Rapport de stage

Évaluation de fonctions élémentaires en virgule flottante sur FPGA

Xavier Pujol

Stage effectué au Laboratoire de l'Informatique du Parallélisme
à l'ENS de Lyon, encadré par Jérémie Detrey et Florent de Dinechin

Juin - juillet 2006

Table des matières

1 Sujet du stage

1.1 Le projet FPLibrary

Mon stage s'est déroulé dans le projet Arénaire, au Laboratoire de l'Informatique du Parallélisme à l'ENS de Lyon. Le thème général de ce projet est l'arithmétique des ordinateurs. C'est un domaine qui comprend le développement d'algorithmes dans différents domaines de l'arithmétique et leur implémentation de manière logicielle ou matérielle. Dans ce projet, j'ai participé à la création de FPLibrary. Cette bibliothèque est développée principalement par Jérémie Detrey, mon encadrant de stage, qui prépare actuellement sa thèse, et Florent de Dinechin, son directeur de thèse qui est maître de conférence à l'ENS Lyon.

Comme son nom l'indique, le but de ce projet est de concevoir une bibliothèque de fonctions mathématiques élémentaires en virgule flottante (*Floating Point* en anglais, terme dont la signification sera détaillée plus tard) en VHDL (Very high speed integrated circuit Hardware Description Language), un langage de programmation conçu pour décrire des circuits électroniques. Cette bibliothèque inclut les opérations de base, qui sont l'addition, la soustraction, la multiplication, la division et la racine carrée, ainsi que les fonctions élémentaires : exponentielle, logarithme, cosinus et sinus. D'autres fonctions pourront être ajoutées pour répondre à des demandes spécifiques. Toutes ces fonctions sont destinées à être utilisées sur des FPGA.

Les FPGA (pour Field-Programmable Gate Array) sont des composants que l'on peut programmer pour qu'ils réalisent des circuits électroniques. Ils sont plus lents que les circuits intégrés mais ont l'avantage de pouvoir être reprogrammés plusieurs fois. Pour rendre cela possible, un FPGA se compose essentiellement d'un grand nombre de LUTs (Look Up Tables) reliées entre elles. Une LUT est un bloc de mémoire adressée par quelques bits qui a un seul bit en sortie. Chaque LUT peut être configurée et jouer ainsi le même rôle qu'une porte logique sur un circuit intégré. Les LUTs sont groupées par deux dans des *slices*. Les routes entre les LUTs sont également configurables. Grâce à cette technique, n'importe quel circuit électronique peut être placé dans un FPGA.

1.2 Calcul en virgule flottante

1.2.1 Représentation des réels

Le calcul avec des nombres réels en informatique est un problème complexe. Comme les réels sont indénombrables, on ne peut pas associer à chaque réel une représentation finie. Il est donc impossible de manipuler les réels en général avec une machine. Le calcul avec des nombres rationnels de taille arbitraire est possible mais assez coûteux.

La solution la plus simple à mettre en œuvre est la représentation en virgule fixe : on fixe le plus petit rationnel ϵ représentable. Un nombre X est alors représenté en mémoire par l'entier $n = \frac{X}{\epsilon}$, codé sur un nombre de bits prédéfini. Ceci ne permet de représenter que des multiples de ϵ et seulement jusqu'à la

valeur maximale de n. L'avantage de ce format est qu'il est facile à manipuler pour une machine. Les opérateurs trigonométriques de FPLibrary permettent d'ailleurs de l'utiliser s'il convient, ce qui permet d'avoir des circuits plus efficaces [3].

Cependant, pour représenter une plage numérique plus grande, le codage en virgule flottante est mieux adapté. Son principe est le même que celui de la notation scientifique, mais en base 2 : un nombre X non nul est représenté sous la forme $s \times m \times 2^E$ où $s \in \{-1, +1\}$ est le signe de X, $m \in [1; 2[$ est sa mantisse et $E \in \mathbb{Z}$ est son exposant (en contraignant l'intervalle de m, on assure l'unicité de cette écriture). En machine, seule la partie fractionnaire de la mantisse est mémorisée puisque sa partie entière vaut toujours 1. Il faut remarquer que cette écriture ne permet pas de coder le réel zéro.

1.2.2 Norme IEEE

Le taille exacte des paramètres mantisse et exposant est standardisée par la norme IEEE 754.

- Un nombre flottant en *simple précision* occupe 32 bits dont 23 bits pour la mantisse, 8 pour l'exposant et un pour le signe.
- Un nombre flottant en *double précision* occupe 64 bits dont 52 pour la mantisse, 11 pour l'exposant et un pour le signe.

Des valeurs particulières sont réservées pour représenter zéro mais aussi l'infini et NaN (*Not a Number* pour le résultat d'un calcul incorrect).

1.2.3 Intérêt de l'utilisation des FPGA

Les microprocesseurs doivent être conformes à cette norme, et donc permettre le calcul flottant au moins sur 32 bits et 64 bits. Mais cela ne laisse que peu de choix sur la précision des calculs. Même si des nombres flottants avec une mantisse de 15 ou de 30 bits suffisent à un algorithme, il doit utiliser la simple ou la double précision. Or utiliser une plus grande précision que nécessaire ralentit l'exécution. De plus certains processeurs calculent systématiquement avec leur précision maximale en interne, et il est alors inutile d'utiliser une précision moindre pour rendre l'exécution plus rapide. A l'inverse, un circuit pour FPGA est mis en mémoire par son utilisateur, qui peut l'optimiser au préalable en fonction de la précision requise. C'est un gain de temps par rapport à un processeur. Les opérateurs de FPLibrary sont conçus pour être paramétrables de cette façon.

En outre, FPLibrary représente les valeurs numérique particulières en utilisant des bits supplémentaires au lieu de réserver certains nombres. Ceci permet d'optimiser les circuits mais cela n'affecte pas la compatibilité avec les normes IEEE. FPLibrary contient aussi des opérateurs qui utilisent un codage des réels différent, LNS (Logarithmic Number System), mais il n'en sera pas question ici.

1.3 Déroulement du stage

1.3.1 Prise en main du sujet

Lorsque j'ai commencé mon stage, certains opérateurs avaient déjà été écrits : les quatre opérations jusqu'à la double précision, la racine carrée, le sinus, le cosinus, l'exponentielle et le logarithme jusqu'à la simple précision. Ainsi, bien que les derniers opérateurs soient conçus de manière générique pour une précision variable, cette précision peut être limitée. En effet, la taille des circuits croît souvent exponentiellement par rapport à la précision requise, et certaines méthodes efficaces jusqu'à la simple précision sont inutilisables au-delà.

Le sujet du stage était d'étendre un opérateur existant jusqu'à la double précision ou d'en concevoir un nouveau. Avant de commencer mes recherches, je me suis familiarisé pendant environ une semaine avec les outils informatiques et le VHDL. En parallèle, je me suis documenté sur les techniques utilisées dans les opérateurs existants, en me basant notamment sur l'article [1]. Au cours de mon stage, j'ai utilisé le logiciel Xilinx ISE, dont la fonction est de générer et d'optimiser un circuit pour FPGA à partir de code VHDL. Pour tester les circuits sur ordinateur sans utiliser de FPGA, je me suis servi du logiciel ModelSim.

Je me suis aussi documenté sur une méthode générale pour créer des circuits calculant des fonctions, bien que je ne m'en soit finalement pas servi. Cette méthode repose sur la compression de tables de valeurs. Elle ne suffit pas à elle seule pour obtenir un opérateur calculant l'exponentielle en simple précision, mais elle est utilisée dans la première version de cet opérateur. L'article [2] expose cette méthode dont l'application est automatisée par le logiciel HOTBM, développé dans l'équipe.

1.3.2 Travail effectué

J'ai choisi de me consacrer à l'opérateur exponentielle, pour le rendre utilisable jusqu'en double précision. Il fallait pour cela trouver une nouvelle technique qui fonctionne également sur tous les intermédiaires entre la simple et la double précision, puisque c'est un des objectifs de FPLibrary.

Je suis assez rapidement parvenu à un premier résultat avec une extension assez simple de l'opérateur existant, mais il n'était pas encore satisfaisant. C'est grâce à une nouvelle idée décrite dans la partie 2.3 que je suis arrivé à une méthode permettant d'obtenir un opérateur vraiment intéressant. Je suis allé jusqu'à la réalisation pratique de cette méthode, en concevant un programme qui l'applique pour générer un composant optimisé en VHDL.

2 Le nouvel opérateur pour l'exponentielle

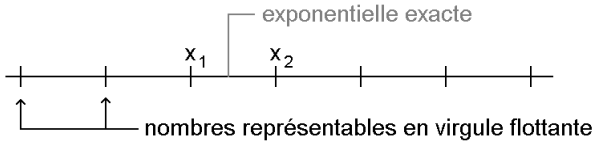
2.1 Fonctionnement du premier opérateur exponentielle

Je vais commencer par expliquer le principe de la première version de l'opérateur, car celui que j'ai conçu utilise les mêmes techniques de base. Cet opérateur

prend en entrée un nombre en virgule flottante dont les champs mantisse et exposant sont de taille configurable. Le résultat est un nombre au même format qui est l'arrondi de l'exponentielle de l'entrée.

2.1.1 Arrondi

L'exponentielle de l'entrée, un rationnel, est toujours irrationnelle (à l'exception de $e^0 = 1$). Le résultat renvoyé par le circuit ne peut donc être qu'un arrondi du résultat exact.



Le résultat exact est situé entre deux nombres représentables en virgule flottante dans le format utilisé. Ce sont ici x_1 et x_2. S'il est garanti que la sortie de l'opérateur est l'un de ces deux nombres - c'est ce qu'on appelle l'arrondi fidèle - on ne peut pas assurer que ce soit le plus proche du résultat exact (arrondi correct).

Ceci est dû à ce qu'on appelle le dilemme du fabriquant de tables, qui peut être illustré par l'exemple suivant. Si $x_1 = 0,10$, si $x_2 = 0,11$ et que l'exponentielle de l'entrée vaut 0,101000000000001... en binaire, il est nécessaire de calculer cette exponentielle à 2^{-15} près pour garantir l'arrondi correct. De manière plus générale, la précision nécessaire pour obtenir l'arrondi correct peut être beaucoup plus grande que le nombre de chiffres demandés en sortie. Ce problème concerne aussi d'ailleurs toutes les autres fonctions élémentaires (logarithme, cosinus, sinus, etc.). Il peut être géré assez efficacement de manière logicielle - c'est par exemple le projet CRlibm d'Arénaire - mais pas en matériel. Le livre [4] donne plus de détails à ce propos.

Obtenir l'arrondi fidèle est plus facile mais demande toutefois une analyse de l'algorithme. Considérons par exemple que l'algorithme renvoie x (x_1 ou x_2) sur une entrée alors que le résultat exact est x_{exact}. La condition de l'arrondi fidèle s'écrit alors $|x_{exact} - x| < x_2 - x_1$. L'algorithme utilise des approximations, et pour satisfaire cette condition, les calculs internes doivent être effectuées avec une précision plus grande que la précision de l'entrée et de la sortie. La dernière étape de l'algorithme est l'arrondi du résultat $x_{interne}$ au nombre x ayant la précision demandée. Dans le pire des cas, $|x_{interne} - x| = \frac{1}{2}(x_2 - x_1)$. L'erreur sur $x_{interne}$ ($|x_{interne} - x_{exact}|$) ne doit donc pas dépasser $\frac{1}{2}(x_2 - x_1)$. L'analyse de cette erreur sera effectuée dans la partie 2.3.3.

2.1.2 Table de valeurs

Pour calculer des valeurs approchées de la fonction exponentielle, on peut d'abord penser à utiliser les premiers termes d'une série comme par exemple :

$$e^x = 1 + x + \frac{x^2}{2} + ... + \frac{x^n}{n!} + ...$$

Mais en fait, le fonctionnement de l'opérateur exponentielle ne peut être basé sur cette série ou même une autre à convergence plus rapide. En effet, la surface d'un FPGA reste relativement limitée. Lors de la première phase de mon travail, j'ai évalué la taille des opérateurs addition et multiplication. Le FPGA sur lequel je travaillais avait 5 000 slices (groupe de deux LUTs). Un circuit multipliant deux nombres de taille l_1 et l_2 occupait de l'ordre de $\frac{1}{2}l_1 \times l_2$ slices, ce qui fait déjà 1 352 slices pour multiplier deux mantisses complètes de nombres en double précision. Certains FPGA peuvent contenir des multiplieurs, ce qui évite de remplir des slices pour faire ces calculs, mais ce n'est pas le cas de tous les FPGA et le nombre de ces multiplieurs est de toute façon restreint. Faire plusieurs itérations d'une suite qui nécessiterait de telles multiplications n'est donc pas une bonne solution. De manière générale, il faut éviter de faire des opérations avec toute la précision de l'entrée.

La première idée de l'algorithme est donc d'éviter de faire des calculs en stockant simplement les valeurs de la fonction dans une table. Bien sûr, il est impossible de mémoriser une table des 2^n valeurs que prend la fonction en précision n dès que n $>$ 12 environ. La taille d'une table sur FPGA peut en effet être estimée par la formule taille $= 2^{(\text{bits en entrée-5})} \times$ (bits en sortie) slices, ce qui remplit 4000 slices lorsqu'il y a 12 bits d'entrée en 32 en sortie. Mais la fonction exponentielle a la propriété mathématique suivante :

$$e^{a+b} = e^a \times e^b$$

En ne tenant compte pour l'instant que de la mantisse de l'entrée, on peut considérer les deux nombres a et b définis ainsi :

m =	1,	m_1	m_2	…	m_i	m_{i+1}	...	m_n
a =	1,	m_1	m_2	…	m_i	0	...	0
b =	0,	0	0	...	0	m_{i+1}	...	m_n

Comme $m = a + b$, $e^m = e^a \times e^b$. Si $i < 10$, on peut stocker les valeurs de e^a dans une table de petite taille. La valeur de e^b est obtenue de façon semblable avec une table, mais des optimisations supplémentaires permettent d'augmenter la longueur maximale de b. En effet,

$$e^b = 1 + b + \mathcal{O}(b^2)$$

Au lieu de stocker e^b dans une table, on stocke $T(b) = e^b - 1 - b$ qui est un nombre de $n - 2i$ bits environ au lieu de n. De plus, la table est assez régulière puisque les valeurs stockées dans la table sont quasiment égales à $\frac{b^2}{2}$. Une technique de "compression de table" peut alors être appliquée [2]. Au final,

$$e^m = e^a(1 + b + T(b)) = e^a + e^a \times (b + T(b))$$

Une seule multiplication suffit alors pour obtenir e^m. De plus, cette multiplication n'est pas de taille $n \times n$ mais de l'ordre de $n \times (n - i)$.

2.1.3 Réduction d'argument

La seconde difficulté de conception tient à la représentation du nombre en virgule flottante, qui n'est pas facilement utilisable telle quelle. Néanmoins, dans le cas de la fonction exponentielle, on peut facilement se ramener à calculer l'exponentielle d'un nombre en virgule fixe compris entre $-\frac{1}{2}$ et $\frac{1}{2}$.

Considérons une valeur d'entrée $X = \pm m \times 2^E$ que l'on souhaite convertir en virgule fixe. Pour pouvoir représenter toute la plage des entrées il faudrait un nombre en virgule fixe sur 2^{w_E} bits, où w_E est la dimension du champ qui contient l'exposant. En double précision, ce nombre occuperait par exemple $2^{11} = 2\,048$ bits.

Mais étant données les variations de la fonction exponentielle, e^X n'est non nul et non infini que pour des valeurs de E assez petites. Dès que E est supérieur à $w_E + 1$, 12 en double précision,

$$e^{m \times 2^E} > e^{2^E} > 2^{2^E} > 2^{2^{w_E+1}}$$

et le résultat dépasse le plus grand nombre représentable (ou est nul dans le cas où l'entrée est négative). La conversion en virgule fixe peut donc se limiter au cas ou l'exposant est dans la plage intéressante et le résultat tient alors sur un nombre raisonnable de bits ($w_M + w_E$,w_M étant la taille de la mantisse).

L'étape suivante est de ramener le calcul de X à celui d'un nombre de l'intervalle $\left[-\frac{1}{2}; \frac{1}{2}\right]$. Pour cela, on calcule d'abord dans le circuit l'arrondi k de $X \times \frac{1}{\log 2}$ à l'entier le plus proche (ici et partout par la suite, la notation log désignera la fonction logarithme népérien). La notation marque le fait que le calcul effectué est un produit et non une division, qui serait beaucoup plus coûteuse en surface. Comme $\frac{1}{\log 2}$ ne peut être stocké qu'avec une précision finie, il est impossible de garantir que k est bien l'arrondi correct du résultat, mais en fait ce n'est pas nécessaire. Il suffit même d'une approximation très large de la constante $\frac{1}{\log 2}$ du moment qu'on remplit les conditions qui vont être détaillées. On réduit donc au maximum la précision pour obtenir le multiplieur le plus petit possible dans le circuit.

On calcule ensuite $Y = X - k \log 2$. Alors $|Y| \leq \frac{\log 2}{2}$, à ceci près qu'on peut avoir commis une erreur sur k. On se contente donc d'une moins bonne approximation, $Y \leq \frac{1}{2}$. La précision de la constante $\frac{1}{\log 2}$ doit alors être suffisante pour que cette deuxième majoration soit valide.

L'exponentielle de Y est finalement obtenue en utilisant la technique du découpage expliquée précédemment au 2.1.2. Comme $X = Y + k \log 2$,

$$e^x = e^y \times 2^k$$

Ce résultat est quasiment au format de sortie attendu, avec comme mantisse e^y et comme exposant 2^k. Néanmoins,

$$e^y \in \left[e^{-\frac{1}{2}} \simeq 0,6; e^{\frac{1}{2}} \simeq 1,6\right]$$

Il peut donc être nécessaire de multiplier la mantisse par 2 et de décrémenter l'exposant k pour que la mantisse soit dans le bon intervalle.

En résumé, pour obtenir l'exponentielle d'un nombre en virgule flottante, une première partie du circuit calcule l'exposant k du résultat (à une unité près) et une valeur Y. Sur ce nombre Y on applique la méthode du "découpage" en deux parties a et b. Les exponentielles de a et b sont récupérées dans des tables et multipliées, ce qui donne après renormalisation la mantisse du résultat.

2.2 Problèmes pour passer à la double précision

2.2.1 Taille des tables

Cet méthode permet de produire des circuits assez petits pour l'exponentielle jusqu'à la simple précision. Cependant, comme cela a déjà été mentionné, la taille d'une table de valeurs croit exponentiellement avec celle de l'entrée, donc la taille de l'opérateur croit exponentiellement avec la précision nécessaire dès que l'on dépasse les précisions les plus faibles. Voici par exemple les tailles des circuits produits pour l'exponentielle selon la précision de l'entrée, d'après l'article [1] . Les pourcentages indiquent la proportion des slices utilisés par rapport au total (5 000) :

Taille de l'exposant	Taille de la mantisse	Slices utilisés
3	6	137 (2,7%)
5	10	258 (5,2%)
6	13	357 (7,1%)
7	16	480 (9,6%)
8	23	948 (19%)

Il n'est donc pas possible d'augmenter la taille de la mantisse jusqu'à 52.

2.2.2 Taille des multiplieurs

La première idée que j'ai testée pour résoudre ce problème a été de réutiliser la technique du découpage de l'entrée. Il est en effet possible de diviser une entrée en un nombre quelconque de parties en appliquant toujours la même formule.

$$X = \underbrace{1, x_1 x_2 ... x_i}_{a} \underbrace{x_{i+1} x_{i+2} ... x_j}_{b} \underbrace{x_{j+1} x_{j+2} ... x_n}_{c}$$

(Ceci est une écriture simplifiée pour représenter le découpage détaillé au 2.1.2.)

$$e^x = e^a \times e^b \times e^c$$

L'augmentation du nombre de morceaux réduit la taille totale des tables mais pour obtenir le résultat final, il faut faire plus de multiplieurs. D'un autre côté, toutes les multiplications ne sont pas de la même taille. En effectuant les produits en sens inverse, les derniers sont beaucoup plus petits. Par exemple, avec ce découpage en trois, $e^b \times e^c$ est calculé par $1 + e^b + e^c + (e^b - 1)(e^c - 1)$, ce donne un produit d'une taille de l'ordre de $(n - i) \times (n - j)$.

Il est possible de diminuer encore la taille des opérandes en faisant une approximation sur le calcul du produit. Pour cela, on néglige les bits du résultat qui sont moins significatifs que le dernier bit de X. Il suffit alors de calculer les $n-i-j$ premiers bits de ce nombre. Mais une approximation à 2^{-n} près de ces premiers bits est donnée en multipliant seulement les $n-i-j+1$ premiers bits de $1-e^b$ et de $1-e^c$. Ceci modifie bien sûr le résultat final et l'erreur maximale doit être contrôlée en calculant en interne avec plus de précision. Cette analyse est détaillée dans la partie 2.3.3.

Pour déterminer si cette méthode était ou non utilisable jusqu'à la double précision, j'ai estimé manuellement la surface nécessaire pour plusieurs découpages. En découpant la mantisse de 52 bits en quatre parties de tailles 9/9/9/25, la surface couverte pour le calcul de l'exponentielle de l'argument déjà réduit entre $-\frac{1}{2}$ et $\frac{1}{2}$ est d'environ 4 000 slices (sur les 5 000 du FPGA), sans toutefois prendre en compte la compression de la dernière table. Ce n'était donc pas satisfaisant.

2.3 Solution trouvée

2.3.1 Principe

J'ai amélioré la méthode précédente grâce à une idée fondée sur un algorithme de Wong et Goto, présenté dans le livre [4]. Elle repose toujours sur le découpage de l'entrée en segments mais a l'intérêt de réduire la taille des multiplieurs. Par la suite, j'appellerai cette méthode "méthode 2" et la précédente "méthode 1".

Voici une description de cette idée. On écrit toujours :

$$X = 0, \underbrace{x_1 x_2 ... x_i}_{a} \underbrace{x_{i+1} x_{i+2} ... x_n}_{b}$$

mais au lieu de d'utiliser une table pour obtenir simplement e^a, on stocke $T_{exp}(a)$, l'exponentielle de a tronquée à i chiffres. Dans une autre table, on stocke $T_{log}(a) = \log(T_{exp}(a))$ avec en sortie la même précision que celle de X. On peut alors réécrire X sous la forme

$$X = T_{log}(a) + (a - T_{log}(a)) + b$$

Par conséquent,

$$e^x = e^{T_{log}(a)} \times e^{a - T_{log}(a) + b}$$

Le calcul de e^b est remplacé par celui de $e^{a-T_{log}(a)+b}$. Mais comme $e^{T_{log}(a)} = T_{exp}(a)$ n'a que i bits après la virgule au lieu de n, le produit est de taille $i \times (n-i)$ environ plutôt que $n \times (n-i)$.

Pour le reste, le principe est le même que pour la première méthode : on calcule l'exponentielle de la partie suivante, soit avec une table, soit en la divisant à nouveau et en appliquant ce même principe, à la différence est que la partie suivante n'est pas seulement b mais la somme de b et d'un "reste", $a - T_{log}(a)$. Pour que l'algorithme soit intéressant, il suffit que ce reste soit du même ordre

de grandeur que b, ce qui est bien le cas. Pour être plus précis, $a - T(a) + b$ occupe au maximum un bit de plus que b, ou deux si a peut être négatif.

L'algorithme de Wong et Goto était prévu pour la double précision seulement, et il n'était pas applicable tel quel pour fabriquer un circuit (la surface aurait largement dépassé la capacité du FPGA). J'ai cependant essayé d'appliquer l'idée de plusieurs façons différentes et j'ai trouvé une solution de surface bien plus faible qu'avec les autres essais en utilisant un nombre maximal de petits morceaux. En raison des approximations commises, une précision de 58 bits est nécessaire pour obtenir au final un arrondi fidèle du résultat sur 52 bits. Voici comment le calcul s'effectue :

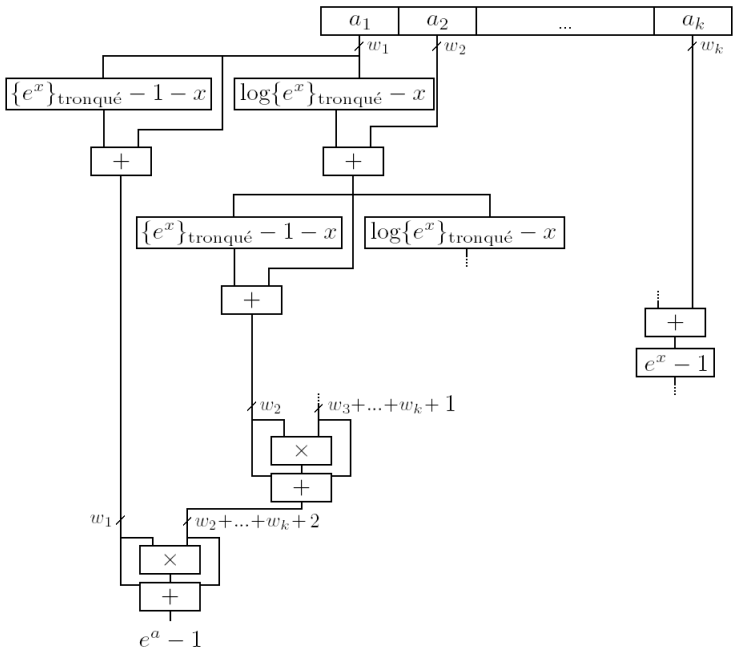
$$X = \pm 0,0\underbrace{x_1x_2x_3}_{a_1}\underbrace{x_4x_5x_6}_{a_2}\underbrace{x_7x_8x_9}_{a_3}...\underbrace{x_{28}x_{29}x_{30}}_{a_{10}}\underbrace{x_{31}x_{32}...x_{57}}_{b}$$

Pour chaque morceau a_i, on utilise l'idée présentée dans ce paragraphe. Ainsi,

$$e^x = T_{exp,1}(a_1) \times e^{[a-T_{log,1}(a_1)]+(a_2+a_3+...+a_{10}+b)}$$

Le calcul de x est donc ramené à celui de $[a-T_{log,1}(a_1)]+(a_2+a_3+...+a_{10}+b)$. On réitère alors l'opération en séparant ce nombre à la limite entre a_2 et a_3. A la différence de la méthode 1, le deuxième morceau n'est pas simplement a_2 mais la somme de a_2 et des premiers bits du reste du morceau précédent, $a - T_{log,1}(a_1)$. L'itération du processus conduit finalement à calculer l'exponentielle de t, somme du dernier morceau b et d'un reste, ce qui est fait simplement avec la formule $e^t = 1 + t$ car $t^2 < 2^{-58}$. Au final, la partie de calcul en virgule fixe n'occupe que 1 200 slices environ.

Voici une représentation de l'architecture d'un composant utilisant la méthode décrite ci-dessus (le shéma ne contient pas les parties de convertion en virgule fixe et traitement du résultat final, arrondi et gestion d'erreurs).



Il est à noter que pour chaque morceau, les tables stockent en fait $T_{exp}(x) - 1 - x$ et $T_{log}(x) - x$ au lieu de simplement T_{exp} et T_{log}, pour gagner de la surface. Pour la dernière partie, le calcul de $e^x - 1$ n'est pas détaillé car selon la taille de cette partie par rapport à la longueur totale de la mantisse, on utilise soit l'approximation $e^x - 1 = x$, soit $e^x - 1 = x +$ une valeur extraite d'une table.

2.3.2 Programmation

Diviser l'entrée en un maximum de morceaux était adapté aux grandes précisions (pour plus de 40 bits de mantisse environ), mais pour des précisions moindres, un découpage intermédiaire utilisant une méthode pour certains morceaux et l'autre pour les morceaux restants devenait meilleur. Cela est possible car on a vu au 2.3 qu'en ajoutant un reste, on augmente la taille de la partie suivante d'un ou deux bits. De plus, deux tables sont nécessaires au lieu d'une pour la méthode originale. La solution optimale n'est donc pas évidente.

Mon travail après avoir trouvé une technique suffisamment efficace a été de réaliser un programme souple pour de générer automatiquement du code VHDL à partir d'un découpage en entrée et du numéro de la méthode à appliquer sur chaque morceau. J'ai écrit ce programme en C++. Pour générer les tables de valeurs des fonctions exponentielle et logarithme, j'ai utilisé la bibliothèque MPFR qui permet de faire des calculs avec une précision arbitraire et en garantissant l'arrondi correct. Cette propriété est utilisée dans la partie sur l'analyse d'erreur.

Dans un second temps, j'ai étendu le programme pour qu'il recherche lui-même la solution de surface minimale pour une précision d'entrée donnée. L'évaluation de la surface n'est qu'approximative mais l'exploration a quand même

permis d'améliorer les résultats trouvés manuellement.

2.3.3 Analyse d'erreur

Il a été mentionné dans la partie 2.1.1 que le composant devait calculer l'exponentielle de l'entrée avec un arrondi fidèle. Or plusieurs approximations sont faites lorsqu'on calcule l'exponentielle en utilisant les méthodes précédentes. D'abord les tables d'exponentielles ou de logarithmes ne peuvent être exactes. Ensuite, plusieurs optimisations dans le circuit introduisent des erreurs supplémentaires. C'est pourquoi un circuit qui calcule des exponentielles de nombres ayant une mantisse de n chiffres utilisera en interne une représentation sur $n+\gamma$ chiffres, où γ doit être évalué en fonction de l'erreur maximale possible.

L'évaluation de cette erreur a été intégrée dans le programme d'exploration puisqu'elle était nécessaire pour optimiser le circuit. Voici les approximations qui peuvent être commises lors du calcul, selon les méthodes utilisées. Certaines pourraient être diminuées (par exemple en remplaçant les troncatures par des arrondis) mais en augmentant la taille du circuit. On note $m = n + \gamma$ la précision interne des calculs, c'est à dire le nombre de chiffres après la virgule. On note $\epsilon = 2^{-m}$ l'ULP (Unit in the Last Place). L'erreur maximale Δv sur une valeur v est évaluée par rapport à ϵ, en mesurant l'écart entre cette valeur et ce qu'elle serait en théorie, si les tables de valeurs et toutes les opérations étaient exactes. Cette valeur théorique est notée v_{th}. Dans cette situation, le résultat final obtenu serait bien l'exponentielle exacte.

- Conversion en virgule fixe : si l'exposant est positif, la mantisse de X est convertie en un nombre à virgule fixe en rajoutant des zéros, ce qui ne modifie rien. Par contre, si l'exposant est négatif et inférieur à $-\gamma$, les derniers bits de la mantisse qui sont moins significatifs que ϵ sont tronqués. La valeur en virgule fixe peut donc être différente de X, d'au plus ϵ.
- Réduction d'argument dans l'intervalle $\left[-\frac{1}{2}; \frac{1}{2}\right]$. On soustrait à X une valeur approchée de $k \log 2$. Un compromis entre la précision et la taille est de stocker suffisamment de bits de $\log 2$ pour assurer que l'erreur sur le produit reste inférieure à $\frac{\epsilon}{2}$. Mais le résultat de $k \log 2$ est alors trop précis et doit être tronqué à la précision m, d'où une erreur ϵ supplémentaire.
- Pour chaque morceau avec lequel on utilise la méthode 2, un reste comprenant une valeur extraite d'une table de logarithmes est ajouté à la partie suivante. Sur ce reste, une erreur de $\frac{\epsilon}{2}$ au plus est possible. C'est d'ailleurs grâce à l'arrondi correct de la bibliothèque MPFR que l'on peut majorer cette erreur par $\frac{\epsilon}{2}$, sinon elle ne serait majorée que par ϵ. Dans le pire des cas, toutes ces erreurs s'additionnent sur le dernier morceau.
- L'exponentielle $y = T(x)$ de la dernière partie x est déduite d'une table dont la sortie est approximée de $\frac{\epsilon}{2}$. On a donc

$$|y - y_{th}| \leq |e^x - e^{x_{th}}| + \frac{\epsilon}{2} \leq \Delta x + \frac{\epsilon}{2}$$

en négligeant le terme de second ordre $x^2 - x_{th}^2 = (x + x_{th})(x - x_{th})$ qui

est bien plus petit que ϵ car $x + x_{th} \ll 1$ et $x - x_{th}$ est de l'ordre de ϵ. On en déduit que

$$\Delta y \leq \Delta x + \frac{\epsilon}{2}$$

- Lorsqu'on utilise la méthode 1 pour calculer l'exponentielle d'une partie x, constituée d'un morceau x_1 de la partie suivante x_2, on commence par évaluer e^{x_2} (en fait $f(x_2) = e^{x_2} - 1$ qui est beaucoup plus petit). Le résultat est donné par la formule $f(x) = f(x_1) + f(x_2) + f(x_1) \times f(x_2)$ qui demande d'évaluer le produit $f(x_1) \times f(x_2)$. L'erreur sur le résultat du produit s'exprime en fonction des erreurs sur les entrées ainsi que des valeurs maximales des paramètres x_1 et x_2. Dans le pire des cas, $x_1 = x_{1,th} + \Delta x_1$ et $x_2 = x_{2,th} + \Delta x_2$. On a alors

$$\begin{aligned} x_1 x_2 &= x_{1,th} x_{2,th} + (\Delta x_1) x_2 + x_1 (\Delta x_2) + \mathcal{O}(\epsilon^2) \\ &\leq x_{1,th} x_{2,th} + (\Delta x_1) x_2^{max} + (\Delta x_2) x_1^{max} \end{aligned}$$

  d'où

$$\Delta(x_1 x_2) = (\Delta x_1) x_2^{max} + (\Delta x_2) x_1^{max}$$

  Comme le produit $x_1 x_2$ est tronqué à m bits pour enlever les bits moins significatifs que ϵ, une erreur supplémentaire de ϵ peut s'ajouter au total. De plus on avait vu que l'on pouvait faire le produit seulement avec les $m+1$ premiers bits des opérandes pour gagner en surface. Pour évaluer l'erreur dans ce cas, ramenons x_1 et x_2 à l'intervalle $[0\,;1[$. L'erreur maximale ajoutée au produit est alors de

$$(x_1 + 2^{-n-1})(x_2 + 2^{-n-1}) - x_1 x_2 = x_1 2^{-n-1} + x_2 2^{-n-1} + 2^{-2(n+1)} \leq 2^{-n}$$

  en négligeant $2^{-2(n+1)}$. En revenant à l'intervalle de départ, l'erreur supplémentaire est au plus égale à ϵ.
- Pour les morceaux utilisant la méthode 2, on fait un produit entre une valeur exacte $T_{exp}(x_1)$ maximisée par $1 + x_1^{max}$ environ et le résultat du calcul de l'exponentielle y de la partie suivante x_2, sur lequel l'erreur maximale Δy est connue ($T_{exp}(x_1)$ est une exponentielle tronquée mais cela fait partie de l'algorithme et n'ajoute pas d'erreur au résultat final). L'erreur sur le résultat du produit est donc $(1 + x_1^{max})\Delta y$ plus ϵ car le produit est tronqué.

Une fois le résultat r obtenu, il est possible qu'il faille multiplier celui-ci par 2 pour ramener le ramener dans l'intervalle $[1\,;2[$. Cela oblige à augmenter d'un bit la précision d'un résultat. Au final, la condition pour que le résultat du calcul soit toujours un arrondi fidèle du résultat exact est que $\Delta r \leq 2^{-(n+2)}$.

3 Conclusion

3.1 Bilan

Au final, je suis parvenu à intégrer à FPLibrary un opérateur en double précision pour l'exponentielle, et de surface assez réduite (2 200 slices environ).

Pour cela, j'ai écrit un programme qui réalise à la fois l'optimisation du découpage pour minimiser la surface et la génération du code VHDL décrivant le circuit.

J'ai pu tester le résultat à l'aide du logiciel ModelSim, d'abord manuellement, puis automatiquement pour des plages de valeurs importantes (même si la vérification exhaustive n'était pas possible). Etant donné que l'erreur maximale n'est presque jamais atteinte, le résultat que donne le circuit est très souvent l'arrondi correct et pas seulement l'arrondi fidèle (98,6% des cas).

Enfin, la méthode que j'ai appliquée pour la double précision est efficace aussi en simple précision, et pour celle-ci mon programme génère un opérateur encore plus petit que l'opérateur précédent.

3.2 Améliorations possibles

Pour la fonction exponentielle, je n'ai pas prévu d'aller au-delà de la double précision, mais la méthode pourrait certainement être utilisée plus loin. En effet, la taille du circuit ne croît pas exponentiellement en fonction de la précision mais quadratiquement. Des améliorations permettraient de garder une taille raisonnable :

- Il serait possible d'utiliser une méthode de compression pour réduire la taille de la table associée au dernier morceau du découpage [2].
- Au lieu d'utiliser la même précision interne à chaque étape du calcul, la précision de chaque étape pourrait être ajustée pour obtenir au final une erreur plus faible avec la même surface.
- Pour optimiser la vitesse, cette fois, il sera possible de pipeliner le circuit, c'est à dire de le diviser en plusieurs sections successives. Ainsi, dès que la première section a terminé le calcul sur la première entrée, l'entrée suivante est chargée. Au final, l'exécution d'un flot de calculs peut se faire beaucoup plus rapidement que les calculs séparés.

Autres opérateurs : la technique de calcul utilise des propriétés spécifiques de l'exponentielle, mais quelques adaptations pourraient permettre de l'appliquer aux autres fonctions.

Remerciements

Ce stage m'a permis de découvrir la vie d'une équipe de recherche, d'entrevoir un autre domaine de l'informatique et de me consacrer à un projet pour y apporter ma contribution. Je remercie Jérémie Detrey et Florent de Dinechin qui m'ont accueilli dans cette équipe et qui m'ont guidé dans mon travail.

Références

[1] Jérémie Detrey et Florent de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs", in *Journal of Microprocessors and Microsystems*, Elsevier, 2006 (à paraître).

[2] Jérémie Detrey et Florent de Dinechin, "Table-Based Polynomials for Fast Hardware Function Evaluation", in *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 328-333. IEEE Computer Society, juillet 2005.

[3] Jérémie Detrey et Florent de Dinechin, "Opérateurs trigonométriques en virgule flottante sur FPGA", in *RenPar'17, SympA'2006, CFSE'5 et JC'2006*, Perpignan, France, octobre 2006 (à paraître).

[4] J.M. Muller, *Elementary Functions, Algorithms and Implementation*, Birkhauser, Boston, 1997.