# L'arithmétique sur le tas

Nicolas Brunie, Florent De Dinechin, Matei Istoan, Guillaume Sergent

**ComPAS'2013** : RenPar'21/ SympA'15 / CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

# L'arithmétique sur le tas

Nicolas Brunie, Florent de Dinechin, Matei Istoan, Guillaume Sergent

LIP, Université de Lyon (CNRS/ENS-Lyon/INRIA/UCBL)
46, allée d'Italie, 69364 Lyon Cedex 07

---

**Résumé**
On appelle un tas de bits une somme non évaluée de variable binaires, chacune pondérée par une puissance de 2. Par exemple, tous les polynômes à plusieurs variables peuvent s'exprimer comme un tas dont chaque variable est un ET logique des bits d'entrée. Cette représentation est pertinente car elle exprime le parallélisme au niveau du bit. La littérature sur les multiplieurs binaires montre comment construire des architectures efficaces qui calculent la valeur d'un tas de bits. Le présent article montre l'intérêt de revisiter un certain nombre d'opérateurs arithmétiques composés pour les exprimer comme des tas de bits.

**Mots-clés :** Arithmétique matérielle, somme de bits pondérés, polynôme, FPGA

---

## 1. Introduction

L'arithmétique binaire classique représente un nombre en virgule fixe comme suit :

$$X = \sum_{i=i_{min}}^{i_{max}} 2^i x_i \tag{1}$$

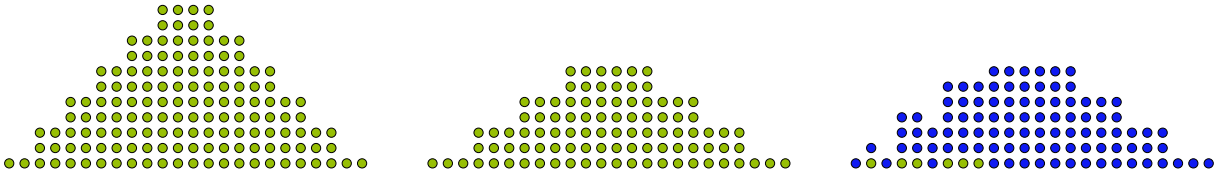
On appellera classiquement *poids* une puissance de 2 comme les $2^i$ dans l'équation ci-desssus : on dira que X est représenté comme une somme de bits pondérés (par des puissances de 2).
Cette notion est bien connue de la littérature sur les multiplieurs : le produit de X par Y peut s'exprimer comme une somme pondérée de produits partiels :

$$\begin{aligned} XY &= (\sum_{i=i_{min}}^{i_{max}} 2^i x_i) \times (\sum_{j=j_{min}}^{j_{max}} 2^j y_j) \\ &= \sum_{i,j} 2^{i+j} x_i y_j \end{aligned}$$

L'addition étant associative, cette représentation ne contraint pas un ordre des calculs : elle expose tout le parallélisme présent au niveau binaire. Cela permet, comme nous allons le voir ci-dessous, de calculer cette somme en temps logarithmique en la taille des nombres [20, 4, 17, 19, 12].

### 1.1. Définition d'un tas de bits

Dans cet article, pour mettre l'accent sur l'absence d'ordre dans cette représentation, nous appellerons une somme de bits pondérés un *tas* (de bits pondérés). On représentera classiquement nos tas de bits par des diagrammes bidimensionnels, avec les poids en abscisse (voir par exemple la figure 1).



(a) Après réécriture des entrées en base $2^2$ pour calculer les produits partiels dans des LUT à 4 entrées

(b) Après réécriture des entrées en base $2^3$ pour calculer les produits partiels dans des LUT à 6 entrées

(c) Idem, mais pour un multiplieur en complément à 2

FIGURE 1 – Tas de bits pour le produit de deux entiers 12 bits

On peut définir pour un tas de bits T
- sa *largeur* $w$, qui est l'intervalle de poids qu'il couvre. Par exemple le produit de deux nombres de $n$ bits est de largeur $w = 2n$ ;
- sa *hauteur* $h$, qui est le nombre maximal de bits de même poids. La hauteur du tas du produit est $n$
- sa *taille* $s$, qui est le nombre total de bits du tas. La taille du produit est $n^2$.

### 1.2. Calcul de la valeur d'un tas

Le calcul de la valeur d'un tas se déroule en deux étapes :
- une étape de *compression*, elle même décomposée en pas de compression élémentaire. Une compression élémentaire consiste à remplacer quelques bits du tas par leur somme. L'archétype du compresseur élémentaire est l'additionneur complet, qui prend trois bits de même poids $i$ et les remplace par leur somme, sur deux bits seulement (de poids $i$ et $i + 1$). En pavant un tas initial avec de tels compresseurs élémentaires travaillant en parallèle, on peut réduire sa hauteur et sa taille d'un facteur $2/3$ (figure 2). On peut ensuite réitérer sur le tas obtenu. La compression doit s'arrêter lorsque le tas est de hauteur 2, ce qui demande un temps logarithmique en la hauteur initiale du tas (pour un log de base $3/2$), et une surface au moins proportionnelle à la taille du tas.
- une étape d'addition rapide des deux lignes du tas compressé, ce qu'on sait également faire en temps logarithmique en la largeur du tas compressé, et surface en $n \log n$.

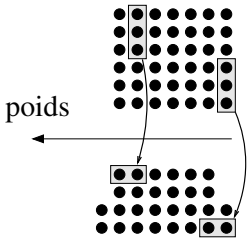



FIGURE 2 – Un étage de compression utilisant des additionneurs complets

Dans ce qui précède on reste volontairement vague sur les termes de complexité : nous allons construire des tas de formes variées, et donner des bornes générales n'a pas grand sens pratique. De plus les compresseurs à utiliser dépendent de la technologie. Il y a eu de nombreux

travaux explorant des compresseurs plus gros pour les multiplieurs VLSI, par exemple 7 en 3 (on réécrit la somme d'une colonne de 7 bits en une ligne de 3 bits seulement). Si l'on sait construire en transistors un tel compresseur avec un délai inférieur à celui d'un assemblage équivalent de compresseurs 3 en 2, cela change les constantes de complexité (la base du logarithme mais aussi la constante multiplicative) et le résultat peut s'avérer meilleur [15]. Il est classique aussi d'utiliser une brique 4 en 2 qui est la fusion, optimisée au niveau des transistors, de deux étages de 3 en 2 [12]. Dans les FPGAs, on va par contre utiliser des compresseurs capables de bien exploiter leurs LUTs (look-up tables) et le matériel qu'ils offrent pour accélérer l'addition [18, 6].

Il existe bien un ordre pertinent pour construire une architecture efficace : c'est l'ordre temporel d'arrivée des bits sur le tas. Dans le cas de la multiplication, si les données sont fournies à l'instant 0, tous les produits partiels $x_i y_j$ arrivent en même temps sur le tas, mais on va voir que ce n'est pas le cas général. Et même dans ce cas simple, une compression élémentaire consiste à retirer quelques bits du tas et à jeter leur somme sur le tas : cette somme arrive sur le tas légèrement plus tard, et ainsi de suite.

### 1.3. Fonctions exprimables comme un tas

On a vu qu'un nombre binaire est un tas de bits. La somme de deux tas de bits est évidemment un tas de bits. Le produit de deux tas reste un tas (le produit de deux sommes de bits pondérés est une somme de bits pondérés). On en déduit donc que tous les polynômes à plusieurs variables sont exprimables comme des tas de bits.

Les tas de bits ne se limitent toutefois pas aux polynômes : on peut jeter sur un tas des valeurs lues dans des tables, par exemple. La section 4 passera en revue certaines fonctions exprimables comme des tas.

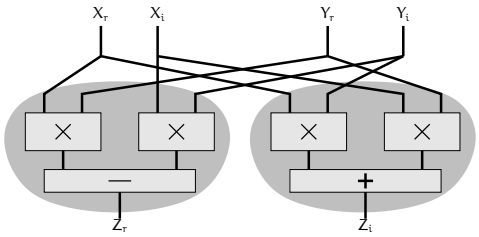
Insistons toutefois sur le fait qu'on parle de polynômes à plusieurs variables. Par exemple, la somme de produits d'un filtre FIR peut s'exprimer comme un tas. Le produit complexe peut s'exprimer comme deux tas (figure 3). Un polynôme sur des variables complexes pourra également s'exprimer directement comme deux tas. Etc.

### 1.4. Manipulation de tas de bits dans FloPoCo

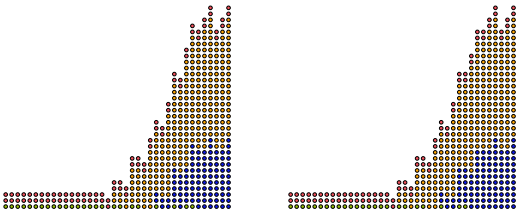
Nous décrivons dans [6] un outil universel pour implémenter des opérations sous forme de tas de bits pour les FPGAs. Cet outil fait partie du générateur de VHDL FloPoCo. Il inclut une structure de donnée qui associe à chaque bit son instant d'arrrivée. Cet outil fournit essentiellement les méthodes suivantes :

- `addBit(weight, expr)` jette sur le tas, au poids `weight`, un bit calculé par l'expression VHDL `expr`. L'instant d'arrivée sur le tas est donné par le temps global géré par FloPoCo [8].
- `addConstantOneBit(weight)` jette sur le tas un bit constant. Les bits constants sont en fait accumulés dans un entier multiprécision, et seule leur somme finale est finalement jetée sur le tas. Cela permet une gestion très simple de l'arithmétique signée en complément à 2 sur le tas [6].
- `generateCompressorVHDL` construit une architecture qui calcule la valeur du tas, en applicant des compresseurs élémentaires à des bits arrivant à peu près au même instant.

Cet outil intègre aussi la gestion des multiplieurs embarqués des FPGAs. En particulier, le générateur de multiplieurs de FloPoCo peut produire un opérateur, mais il peut aussi se contenter de jeter des bits (et des sous-produits) sur le tas d'un opérateur plus grand. L'idée est de retarder au maximum la compression, d'une part pour profiter d'une optimisation globale au lieu de plusieurs optimisations locales, d'autre part pour n'avoir qu'une seule étape d'addi-




(a) Le point de vue arithmétique



(b) Les deux tas pour une précision de 32-bits. Chacune des 4 multiplications utilise le multiplieur 17x24 d'un bloc DSP, complété de multiplieurs 3x3 implémentés en LUTs.

FIGURE 3 – La multiplication complexe sur Virtex5 en deux tas

tion rapide, et non plusieurs en séquence. Un exemple élémentaire d'application est le produit complexe, composé de deux tas regroupant chacun deux multiplieurs (figure 3).
L'objet du présent article est de passer en revue les opportunités qu'offre un tel outil.

## 2. Optimisations algébriques sur le tas

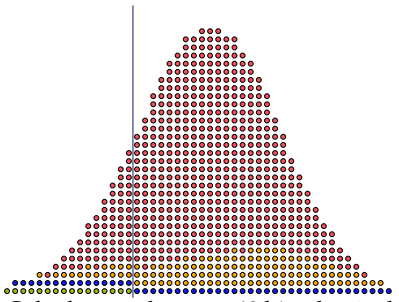
Considérons comme polynôme une approximation de Taylor du sinus : $\sin(X) \approx X - X^3/6$, et cherchons à l'écrire comme un tas de bits. On a :

$$\begin{aligned} X^3 &= \sum_{i=i_{min}}^{i_{max}} 2^{3i} x_i^3 \\ &\quad + \sum_{i_{min} \le i < j \le i_{max}} 3 \cdot 2^{i+2j} x_i x_j^2 \\ &\quad + \sum_{i_{min} \le i < j < k \le i_{max}} 6 \cdot 2^{i+j+k} x_i x_j x_k \end{aligned} \tag{2}$$

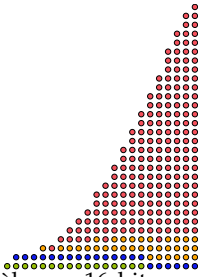
On peut appliquer un certain nombre de simplifications de l'algébre booléenne, comme $x_i^k = x_i$ ou $2 \cdot 2^w a = 2^{w+1} a$. La multiplication par 3 ci-dessus peut s'obtenir par une duplication de bits : $3 \cdot 2^w a = 2^{w+1} a + 2^w a$. C'est moins coûteux qu'une vraie multiplication.
A ce point, le calcul de $X^3$ correspond à un tas de bits dont la taille est d'environ $1/3$ de la taille cumulée de deux multiplieurs. Il sera aussi plus rapide (une compression et une addition finale seulement).
Mais dans ce cas précis on peut aller plus loin, puisque c'est $X^3/6$ qui nous intéresse. On peut réécrire (2) :

$$\begin{aligned} X - X^3/6 &= \sum_{i=i_{min}}^{i_{max}} 2^i x_i \\ &\quad - 1/3 \sum_{i=i_{min}}^{i_{max}} 2^{3i-1} x_i \\ &\quad - \sum_{i_{min} \le i < j \le i_{max}} \cdot 2^{i+2j-1} x_i x_j \\ &\quad - \sum_{i_{min} \le i < j < k \le i_{max}} \cdot 2^{i+j+k} x_i x_j x_k \end{aligned} \tag{3}$$

Il ne reste donc que $i_{max} - i_{min}$ bits à effectivement diviser par 3. Là, il faut faire un arrondi, ce



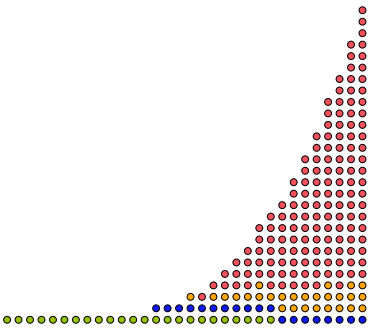
(a) Calcul exact donnant 48 bits de résultat



(b) Calcul fidèle sur 16 bits, par troncature à 16+6 bits

FIGURE 4 – Tas de bits pour l'évaluation de $X - X^3/6$ avec $X \in (0, 1)$ sur 16 bits.

(a) Précision 16 bits ($g = 6$ bits)



(b) Précision 24 bits ($g = 8$ bits)

FIGURE 5 – Tas de bits pour l'évaluation fidèle de $X - X^3/6$ lorsque $|X| < 2^{-4}$.

qui consiste essentiellement à choisir une précision de sortie $p$ qui correspond typiquement à la précision d'entrée : $p = i_{max} - i_{min} + 1$. Pour la division par 3, on peut utiliser une architecture spécifique [5] qui jettera (après un délai) $p$ bits sur le tas. On peut plus simplement remplacer $1/3$ par sa représentation binaire $0.1010101...$ tronquée à la précision nécessaire, et jeter sur le tas les produits partiels correspondant à la seconde ligne. Le coût sur le tas des secondes et troisième lignes de (3) sont alors comparables.

La taille du tas est asymptotiquement dominée par la dernière ligne de (3), qui croît en $p^3/6$ [1]. Si l'on ne s'intéresse qu'aux $p$ bits de poids forts, la croissance est en $p^3/36$ [2]. En pratique, on doit tronquer à la position $w + g$, où $g$ est un nombre de "bits de garde" tels que l'erreur obtenue en tronquant à $w + g$ (la somme de la partie négligée du tas) est inférieure au poids du dernier bit du résultat, ce qu'on appelle l'arrondi fidèle. La figure 4 montre que, sur notre exemple, on peut obtenir l'arrondi fidèle en négligeant plus de la moitié du tas.

De plus, on n'utilisera en pratique un polynôme de Taylor pour évaluer un sinus que pour un argument réduit, donc petit devant 1. Par exemple, si $X < 2^{-4}$, alors $X^3/6 < 2^{-14}$, ce qui repousse d'autant les contributions de ce monôme au tas de bits. La figure 5 illustre ce phénomène.

Dans cet exemple, il y a indubitablement une part de chance dans la compensation du facteur 3 et de la division par 3. Il faut toutefois en retirer que

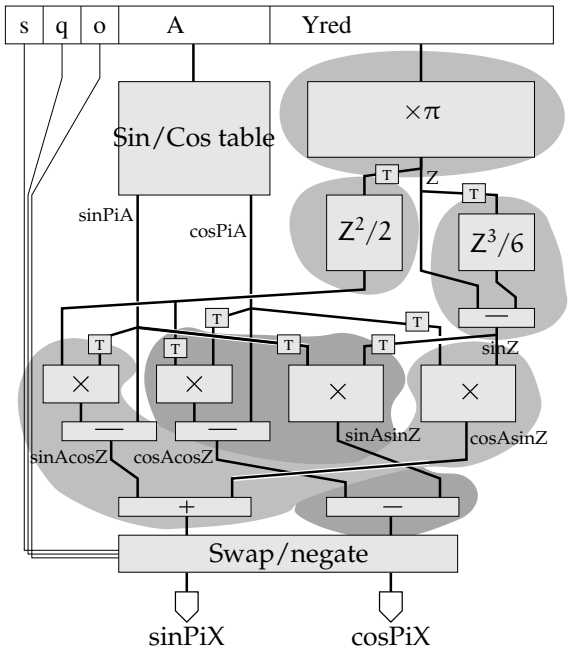



FIGURE 6 – Une architecture trop compliquée pour calculer un sinus et un cosinus, faisant ressortir ses différents tas.

1. la taille en termes de bits sur des tas d'une puissance entière semble toujours très inférieure à celle de la même puissance obtenue par des multiplications naives.
2. l'explosion de la taille du tas pour les hauts degrés peut être compensée par le fait que l'argument est petit devant 1 (et en général devant le coefficient de plus petit degré du polynôme).

### 3. Le tas de bit comme outil de complexité binaire ?

De par sa généralité, la notion de tas de bits parait pertinente pour exprimer et comprendre des questions ouvertes de complexité matérielle, par exemple : quelle est la complexité binaire minimale en espace et en temps pour calculer un sinus en virgule fixe à une certaine précision ?
Pour l'illustrer, la figure 6 décrit un prototype d'architecture actuellement dans FloPoCo pour calculer $\sin(\pi x)$ et $\cos(\pi x)$. Cette architecture tente de s'appuyer sur des propriétés spécifiques des fonctions trigonométriques, comme les identités $\sin(a+b) = \sin(a)\cos(b) + \sin(b)\cos(a)$ ou la simplicité des développements de Taylor de ces deux fonctions.
À l'usage, la performance de cette architecture s'est révélée décevante en comparaison d'une méthode générique réalisant une simple approximation polynômiale. Pourtant, c'est une architecture qui intègre des connaissance spécifique aux fonctions trigonométriques : Pourquoi serait-elle moins performante qu'une architecture générique ?
Rétrospectivement, il apparaît que l'exploitation des propriétés de la fonction s'est traduit par des optimisations arithmétiques (partage de calculs intermédiaires, etc). Mais cela a induit la construction de tas de bits en séquence, et fait perdre du point de vue de la complexité binaire. Mais alors, où est l'optimal ? Cette question reste ouverte. En pratique, il est douteux qu'on puisse y répondre de manière générique. Nous enviseagons plutôt d'utiliser les tas comme un

outil intermédiaire qui permet de mesurer pragmatiquement les coûts en temps et en espace de différentes solutions, afin de les comparer quantitativement.
Il paraît difficile de capturer analytiquement la complexité du tas de bits correspondant à un évaluateur de polynôme multivarié dans le cas général : il y a trop de facteurs (le degré, la taille des entrées, leurs magnitudes, les magnitudes des coefficients, et même leur écriture binaire). Par contre, il est possible de bien poser le problème et de construire un tas de bits qui calcule au plus juste, et c'est ce que nous nous proposons d'explorer.
Ensuite, pour comparer avec le coût d'une approche arithmétique, il faudra préciser celle-ci au niveau du bit avec le même soin. Par exemple,

- pour notre exemple où $X < 2^{-4}$ sur 16 bits, on saurait tronquer X au plus juste avant de calculer $X \times X \times X$, et on pourrait utiliser dans ce contexte des multiplieurs tronqués : sur le fond, les optimisations que nous observons sur le tas de bits sont possibles dans une approche arithmétique, et il n'est pas certain que le gain apporté par un tas aille au delà de celui apporté par les optimisations algébriques.
- Par ailleurs, pour de plus gros degrés, la puissance $d$ d'un nombre se calcule en $\log(d)$ multiplications seulement : ici, l'approche arithmétique exploite de la réutilisation de valeurs intermédiaires, ce qui peut lui redonner l'avantage en terme de complexité asymptotique.
- Il faut distinguer calcul exact (on calcule les $nd$ bits de $X^d$) et calcul approché en virgule fixe (on garde au plus $n$ bits).

## 4. Quelques opérateurs vus comme des tas de bits

Nous listons dans cette section des opérateurs existants qui peuvent être revus comme des tas de bits. Ce travail est en cours. Pour ceux de ces opérateurs qui existent déjà dans FloPoCo sous forme d'opérateurs (correspondant à une entité VHDL), nous ajoutons progressivement une option qui se contente de jeter des bits sur un tas appartenant à une entité plus grosse, pour permettre quand c'est souhaitable de gloabaliser la compression.

### Multiplieurs et multiplieurs constants

Le cas des multiplieurs standards est décrit en détail dans [6]. Nous avons depuis ajouté un opérateur de multiplication-accumulation en un seul tas de bits qui permet par exemple de gagner jusqu'à 30% sur le coût logique d'une évaluation de Horner [7], et 10% sur le délai, par rapport à deux opérateurs séparés. On pourra objecter que ce gain n'illustre peut-être que la piètre qualité des composants séparés. Même dans ce cas, il est plus satisfaisant de passer du temps à l'optimisation d'un unique générateur de compresseurs qu'à l'optimisation séparée des composants qu'il remplace (multiplieurs, multi-additionneurs, etc).
Pour les multiplieurs par une constante, la littérature distingue deux techniques : la méthode à base de table [3, 21] et les méthodes par additions et décalages [10, 13]. Ces dernières tirent leur efficacité de la réutilisation de sommes intermédiaires, et sont donc incompatibles avec un tas unique. Par contre, les méthodes à base de table peuvent s'appuyer sur un tas de bits. Elles sont basées sur la décomposition de l'écriture d'un nombre de $n$ bits X en paquets de $\alpha$ bits :
$X = \sum_{i=0}^{\lceil \frac{n}{\alpha} \rceil - 1} X_i . 2^{\alpha i}$, avec $X_i \in \{0, ..., 2^\alpha - 1\}$.
Le produit de X par une constante entière C sur $m$ bits devient

$$CX = \sum_{i=0}^{\lceil \frac{n}{\alpha} \rceil} CX_i . 2^{-\alpha i}.$$

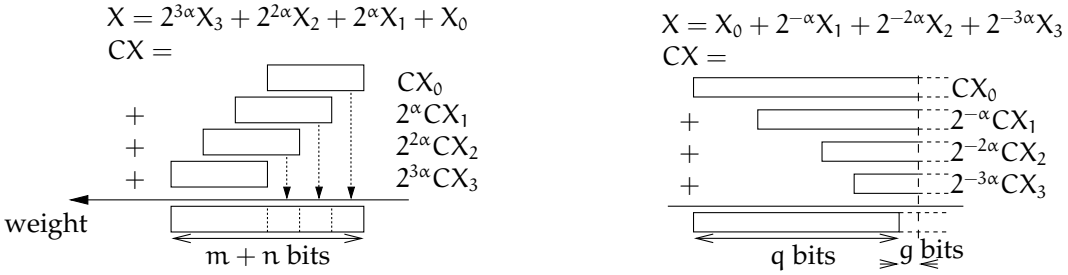



FIGURE 7 – Multiplication par une constante (à gauche, entier par constante entière, à droite, virgule fixe par constante réelle avec arrondi fidèle)

On a une somme de produits partiels $CX_i$ décalés, chacun un entier sur $m + \alpha$ bits. Toute l'idée de cet algorithme est de lire ces $CX_i$ dans des tables précalculées et stockées dans des LUT à $\alpha$ bits d'entrées.
A présent que les LUT ont 6 entrées, on pourrait étendre cette technique à la multiplication d'une constante C par un produit de deux nombres non constants X et Y, chacun décomposés en paquets de 3 bits. Le calcul sur le tas de $CX^2$ sera un cas particulier se prêtant à quelques optimisations.

**Arithmétique complexe et filtres pour le traitement du signal**

La construction d'un filtre FIR ou IIR en virgule fixe fait apparaître des sommes de produits dans lesquelles tous les termes sont alignés [16, 18].
Ici, les avantages de l'approche à base de tas sont :
– une gestion globale de la question de l'arrondi du résultat,
– une construction possiblement plus équilibrée de l'arbre dans le cas où la taille n'est pas une puissance de deux,
– sur FPGA, une meilleure exploitation des LUT à gros grain modernes (5 entrées et plus),
– toujours sur FPGA, une gestion globale du regroupement des petits multiplieurs dans des blocs DSP, comme expliqué dans [6],
– encore et surtout, la simplicité du code, le gros de la complexité étant déporté dans le code du tas de bits.

Le chapitre 5 de la thèse de L. Noury [16] offre, pour un filtre FIR donné dans un contexte ASIC, une étude quantitative des gains offerts par certains de ces points.

**Puissances et polynômes**

Nous avons vu la construction à la main d'un tas de bits pour $X - X^3/6$. Dans le cas le plus général, l'alignement des différents termes d'un polynôme est bien compris (figure 8) : on peut recycler le code existant [7] qui, partant d'une fonction à approcher sur un intervalle, décompose cet intervalle en morceaux de plus en plus petits et construit des approximations polynômiales de plus en plus précises jusqu'à satisfaire la précision requise. La taille de chaque coefficient est définie par la figure 8 : un coefficient n'a pas besoin d'être plus précis que le monôme auquel il participe. L'implémentation actuelle [7] construit ensuite un évaluateur sous forme de Horner. Il nous paraît désormais intéressant d'explorer, pour les petites précisions et les petits degrés, l'alternative d'une évaluation pleinement parallèle en un seul tas. Il est bien connu que l'évaluation d'un polynôme sous forme de Horner : $p(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (... + x \cdot a_d))...)$ compte

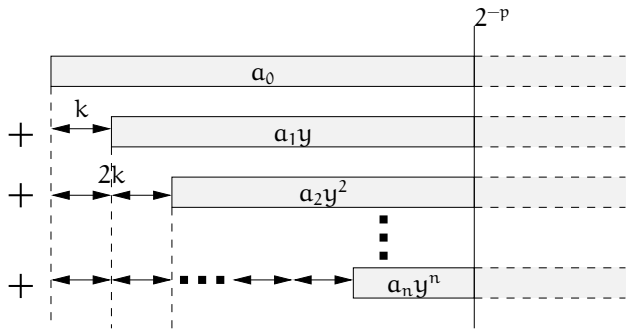



FIGURE 8 – Alignement des monômes d'un polynôme sur le tas, dans le cas où les coefficients sont de poids comparables et $x < 2^{-k}a_0$

moins d'opérations que l'évaluation sous forme développée : $p(x) = a_0 + a_1x + a_2x^2 + ... + a_dx^d$. Mais elle ne s'exprime pas en un seul tas et implique donc plusieurs compressions en séquence : l'approche développée reste donc digne d'intérêt.
La construction des polynômes ayant fixé les tailles et les poids de toutes les variables, il s'agit donc d'implémenter
– la construction d'une structure de données contenant une somme de monômes binaires
– la réduction de cette structure de données par les identités de l'algèbre Booléenne déjà mentionnées,
– éventuellement, si les termes obtenus n'exploitent pas bien les LUTs des FPGA, le regroupement de la somme de plusieurs termes de poids voisins et partageant les mêmes entrées dans des LUTs, voire des tables plus grosses utilisant des blocs mémoire embarqués [14, 9, 11].
Les deux premiers points sont réalisés, le troisième est en cours, et pourra s'appuyer sur des travaux plus anciens, comme la méthode HOTBM *(higher-order table method)* illustrée par la figure 9.

## 5. Conclusion et perspectives

Nous espérons avoir montré le potentiel qu'il y a à repenser systématiquement la construction d'opérateurs arithmétiques en terme de tas de bits. Cette notion, bien connue dans le cadre de la construction des multiplieurs, mérite d'être exposée plus largement à tous les concepteurs de circuits arithmétiques : elle couvre tout ce qui s'exprime comme somme, somme de produit, et en général polynôme à plusieurs variables. Elle a essentiellement trois avantages :
– l'optimisation globale au niveau du bit des opérateurs qui s'y prêtent,
– une approche nouvelle de certaines questions de complexité binaire,
– la simplification du code dans les générateurs arithmétiques comme FloPoCo, puisqu'un seul cadre générique permettant d'obtenir des architectures efficaces pour une classe très large d'opérateurs.
Un travail en cours est de fournir dans FloPoCo les outils pertinents qui permettent d'exprimer cette richesse.

## Bibliographie

1. A000292 : Tetrahedral (or triangular pyramidal) numbers : a(n) = c(n+2,3) =

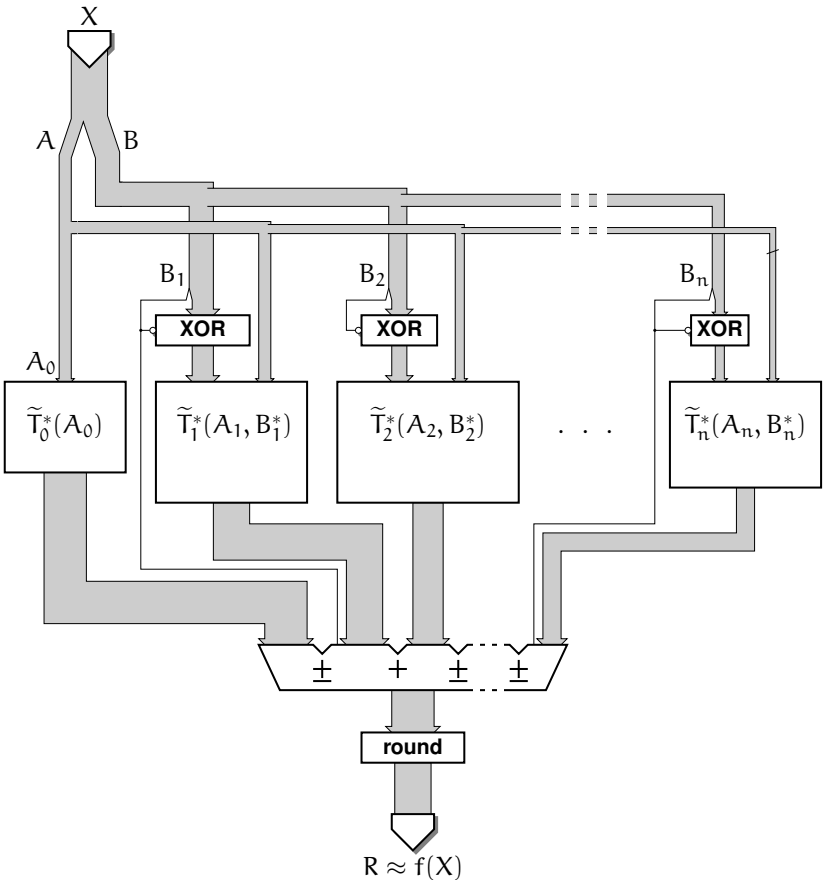



FIGURE 9 – La méthode HOTBM vue comme un tas de bits. Certains des composants de cette figure contribuent plusieurs termes au tas.

n*(n+1)*(n+2)/6. – On-Line Encyclopedia of Integer Sequences. http ://oeis.org/A000292.
2. A000601 : Expansion of $1/((1-x)^2 * (1-x^2) * (1-x^3))$. – On-Line Encyclopedia of Integer Sequences. http ://oeis.org/A000601.
3. Chapman (K.). – Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, no10, mai 1993, p. 80.
4. Dadda (L.). – Some schemes for parallel multipliers. *Alta Frequenza*, vol. 34, 1965, pp. 349–356.
5. de Dinechin (F.) et Didier (L.-S.). – Table-based division by small integer constants. *In : Applied Reconfigurable Computing*, pp. 53–63. – Hong Kong, mars 2012.
6. De Dinechin (F.), Istoan (M.), Sergent (G.), Illyes (K.), Popa (B.) et Brunie (N.). – *Arithmetic around the bit heap*. – Rapport technique, octobre 2012.
7. de Dinechin (F.), Joldes (M.) et Pasca (B.). – Automatic generation of polynomial-based hardware architectures for function evaluation. *In : Application-specific Systems, Architectures and Processors*. – IEEE.
8. de Dinechin (F.) et Pasca (B.). – Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, vol. 28, n4, juillet 2011, pp. 18–27.
9. de Dinechin (F.) et Tisserand (A.). – Multipartite table methods. *IEEE Transactions on Computers*, vol. 54, n3, 2005, pp. 319–330.
10. Dempster (A.) et Macleod (M.). – Constant integer multiplication using minimum adders. *Circuits, Devices and Systems*, vol. 141, n5, 1994, pp. 407–413.
11. Detrey (J.) et de Dinechin (F.). – Table-based polynomials for fast hardware function evaluation. *In : Application-specific Systems, Architectures and Processors*. pp. 328–333. – IEEE.
12. Ercegovac (M. D.) et Lang (T.). – *Digital Arithmetic*. – Morgan Kaufmann, 2003.
13. Gustafsson (O.) et Qureshi (F.). – Addition aware quantization for low complexity and high precision constant multiplication. *IEEE Signal Processing Letters*, vol. 17, n2, 2010, pp. 173–176.
14. Hassler (H.) et Takagi (N.). – Function evaluation by table look-up and addition. *In : 12th Symposium on Computer Arithmetic*. pp. 10–16. – Bath, UK, 1995.
15. Montoye (R. K.), Hokonek (E.) et Runyan (S. L.). – Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, vol. 34, n1, 1990, pp. 59–70.
16. Noury (L.). – *Contribution à la conception de processeurs d'analyse de signaux à large bande dans le domaine temps-fréquence : l'architecture F-TFR*. – Thèse de PhD, Université Paris VI, 2008.
17. Oklobdzija (V.), Villeger (D.) et Liu (S.). – A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Transactions on Computers*, vol. 45, n3, 1996, pp. 294–306.
18. Parendeh-Afshar (H.), Neogy (A.), Brisk (P.) et Ienne (P.). – Compressor tree synthesis on commercial high-performance FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, n4, 2011.
19. Stelling (P. F.), Martel (C. U.), Oklobdzija (V. G.) et Ravi (R.). – Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, vol. 47, n3, 1998, pp. 273–285.
20. Wallace (C. S.). – A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, vol. EC-13, 1964, pp. 14 –17.
21. Wirthlin (M.). – Constant coefficient multiplication using look-up tables. *Journal of VLSI Signal Processing*, vol. 36, n1, 2004, pp. 7–15.