Pipeline automatique d'opérateurs dans FloPoCo 5.0

Matei Istoan, Florent De Dinechin

Compas'2016 : Parallélisme/ Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Pipeline automatique d'opérateurs dans FloPoCo 5.0

Matei Istoan et Florent de Dinechin

Université de Lyon, INRIA,
INSA-Lyon, CITI, F-69621 Villeurbanne, France
florent.de-dinechin@insa-lyon.fr matei.istoan@inria.fr

Résumé
Cet article présente la troisième évolution de la fonctionnalité de pipeline automatique intégrée dans le générateur de cœurs arithmétiques FloPoCo. La description combinatoire en VHDL d'un opérateur de calcul est d'abord encapsulée dans du C++. Ensuite, l'ajout de primitives simples à ce C++ permet d'obtenir automatiquement des versions pipelinées de ce VHDL, à une fréquence arbitraire et pour toute la gamme de cible supportées par l'outil. Les algorithmes utilisés sont décrits dans les grandes lignes, et la qualité des résultats est évaluée.

Mots-clés : Opérateurs arithmétiques, pipeline, ordonnancement

1. Introduction

1.1. Le projet FloPoCo

Le projet FloPoCo étudie depuis 2008 les opportunités arithmétiques spécifique aux FPGAs, en particulier dans le domaine du calcul flottant [3]. Une motivation était de pouvoir paramétrer chaque opérateur arithmétique le plus finement possible :

- taille (nombre de bits en entrée et en sortie),
- algorithme utilisé (par exemple additionneur flottant à un ou deux chemins [6]),
- paramètres de l'opérateur lui-même, par exemple la constante pour un multiplieur par une constante,
- FPGA cible avec ses spécifités architecturales,
- fréquence à laquelle on souhaite faire fonctionner l'opérateur.

Écrire du VHDL paramétré par autant de paramètres était ingérable. Le choix fut donc fait d'un générateur de VHDL écrit en C++. Cela a permis aussi d'inclure dans ce générateur des optimisations de l'architecture générée de plus en plus poussées. L'optimisation du pipeline est de celles-ci.

FloPoCo a, dès ses débuts, offert une assistance à la conception de pipelines corrects par construction [4]. Ce cadre a ensuite été étendu pour automatiser le placement des registres de pipeline [5]. Cet article présente la troisième génération de cette réflexion, qui épure l'interface programmeur, allège le code, et améliore la performance des circuits générés. Elle est développée dans la branche `newPipelineFramework` du dépôt Git et sera distribuée à partir de la version 5.0.

La figure 1 montre l'interface typique d'un générateur d'opérateur implémenté dans FloPoCo. Elle distingue la spécification fonctionnelle, qui décrit l'opération à implémenter, de la spécification de sa performance.

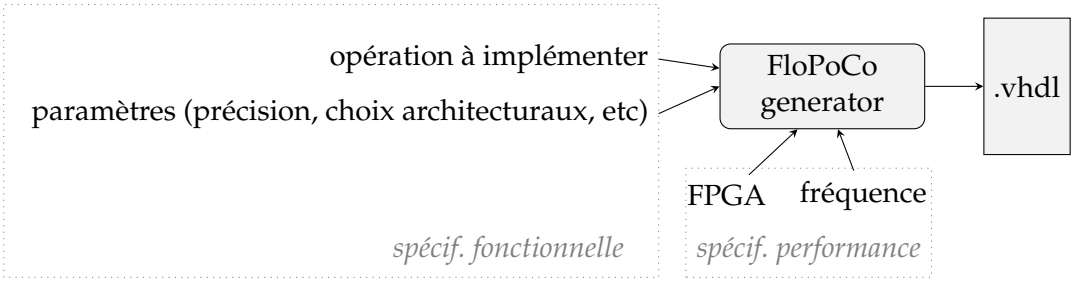


FIGURE 1 – The typical interface to a FloPoCo operator generator

Un opérateur arithmétique au sens de FloPoCo est l'implémentation d'une fonction mathématique (sans mémoire). En terme d'architecture, on peut construire une telle fonction comme un circuit combinatoire. Le concepteur d'un opérateur implémente donc la spécification fonctionnelle en décrivant une architecture combinatoire en VHDL.
Par la suite, on peut vouloir augmenter la performance en pipelinant cette architecture. Le pipeline ne change pas la fonctionnalité numérique de l'opérateur. Il se contente d'augmenter la fréquence de fonctionnement au prix d'un retard (en termes de cycles d'horloges) des sorties sur les entrées.
Pipeliner un opérateur donné pour atteindre une fréquence donnée sur une cible technologique donnée est un travail simple en principe, mais fastidieux et sujet à erreur. Pour un générateur d'opérateurs très paramétré (tant en fonctionnalité qu'en performance), il n'est pas question de pipeliner à la main tous les opérateurs possibles. L'objectif du présent article est donc d'automatiser ce travail.

1.2. Pipeline sous contrainte de fréquence

Le pipeline d'un graphe acyclique direct est une transformation conceptuellement simple. D'abord, la fonction combinatoire peut être représentée par un graphe orienté acyclique (DAG) de portes. Dans ce DAG, on identifie le chemin critique, *i.e.* le chemin qui maximise le délai cumulé (de calcul et de routage), et qui donc impose la fréquence maximale à laquelle le circuit pourra travailler. Ensuite, on coupe ce chemin critique en insérant un ou plusieurs registres dessus. Le nouveau chemin critique sera le plus long délai entre une sortie de registre et une entrée de registre. Si d est le délai du chemin critique, et d_{obj} est l'inverse de la fréquence cible, il faut insérer au moins $\lceil d/d_{obj} \rceil$ registres, le plus équirépartis possibles. De plus, pour préserver la fonctionnalité du circuit, il faut prendre soin d'insérer le même nombre de registres sur tous les chemins d'un nœud A à un nœud B du graphe initial (synchronisation).
Le choix fait dans FloPoCo [5] fut dès 2009 de pipeliner un opérateur (un DAG) pour qu'il fonctionne à une fréquence cible donnée par l'utilisateur. La fréquence est donc une entrée du générateur, qui doit déterminer le nombre de registres à insérer, donc la latence (nombre de cycles entre une entrée et la sortie correspondante). Ce choix distinguait FloPoCo des générateurs du commerce, notamment dans les outils Xilinx et Altera : ceux-ci prenaient en paramètre la latence, et on obtenait la fréquence en synthétisant le circuit pipeliné. Le choix de FloPoCo est meilleur car il est compositionnel : l'assemblage de composants tournant à 200MHz est un circuit qui tourne (presque) à 200MHz. Il a été adopté par des outils plus récents, que nous étudions maintenant.

1.3. Génération automatique des pipelines dans les outils du commerce

Altera présente brièvement dans [2] une implémentation similaire a celle du [8]. Elle consiste aussi a parcourir le DAG représentant le circuit, en insérant des registres en mode glouton. Ils identifient trois problématiques principales. La première est la gestion des registres insérés par l'utilisateur, qu'ils choisissent d'ignorer pendant la temporisation du circuit pour ne la prendre en compte que pendant l'étape de génération du code VHDL. Deuxièmement, c'est la gestion des boucles. Pour attaquer cette problématique, ils transforment leur graphe cyclique en un DAG et coupent les cycles dès qu'ils sont détectés. Finalement, les auteurs parlent de la gestion des sous-composants, traités comme des blocs d'une plus grosse granularité. La fin de l'article suggère une solution basée sur la programmation linéaire.

L'approche de Xilinx [7], même si elle est basée sur les parcours des DAGs, gère le placement des registres sur un circuit déjà synthétisé et placé. La première étape est de déterminer le chemin critique. Une fois trouvé, ils placent des registres sur ce chemin critique. Ensuite, ils déterminent les chemins parallèles avec le chemin critique, puis le nombre minimum des registre a placer sur ces chemins pour équilibrer le graphe. Ce processus et itéré jusqu'à ce que le circuit satisfasse les contraintes de fréquence. Cette solution assure que les registres seront placés avec précision (vu qu'il n'y a plus des sous-estimations a cause du routage et placement). Par contre, elle est très couteuse en terme des calculs, et nécessite un circuit déjà placé.

Simulink et Matlab ont aussi une approche inspirée de [8] dans leur générateur des circuits [9]. Leurs améliorations rendent l'algorithme beaucoup plus performant, mais le problème du retiming reste en soit un problème dur.

Par rapport à ces travaux, nous proposons d'exploiter la connaissance a-priori de l'absence de boucles dans un DAG arithmétique pour obtenir un algorithme plus simple et plus rapide.

1.4. Vue d'ensemble de l'algorithme

La figure 3a donne une vue d'ensemble de la génération de code pipeliné pour un opérateur FloPoCo. Pour illustrer notre propos, nous utiliserons le fragment de code VHDL donné par la figure 2b, extrait de l'architecture d'un additionneur flottant en simple précision. Le code FloPoCo qui généralise ce VHDL à des tailles arbitraires de mantisse `wF` et d'exposant `wE` est donné par la figure 2a. Ce code fait partie du constructeur de l'opérateur (tout en haut de la figure 3a). Ici `wE` et `wF` auraient pu être des `generic` de VHDL. On les a remplacés par des variables C++, ce qui offre de nombreux avantages dont certains vont apparaître dans la suite.

La déclaration des signaux est explicite dans FloPoCo (dans toute la suite nous utilisons le mot "signal" dans son acceptation VHDL). La méthode `declare()` ajoute son argument au dictionnaire des signaux (avec optionnellement sa taille en bits, par exemple 2 bits pour `excR`). Comme le montre la figure 3a, l'éxécution du constructeur produit donc d'une part le VHDL (qui est essentiellement le code de la figure 2b), d'autre part le dictionnaire des signaux.

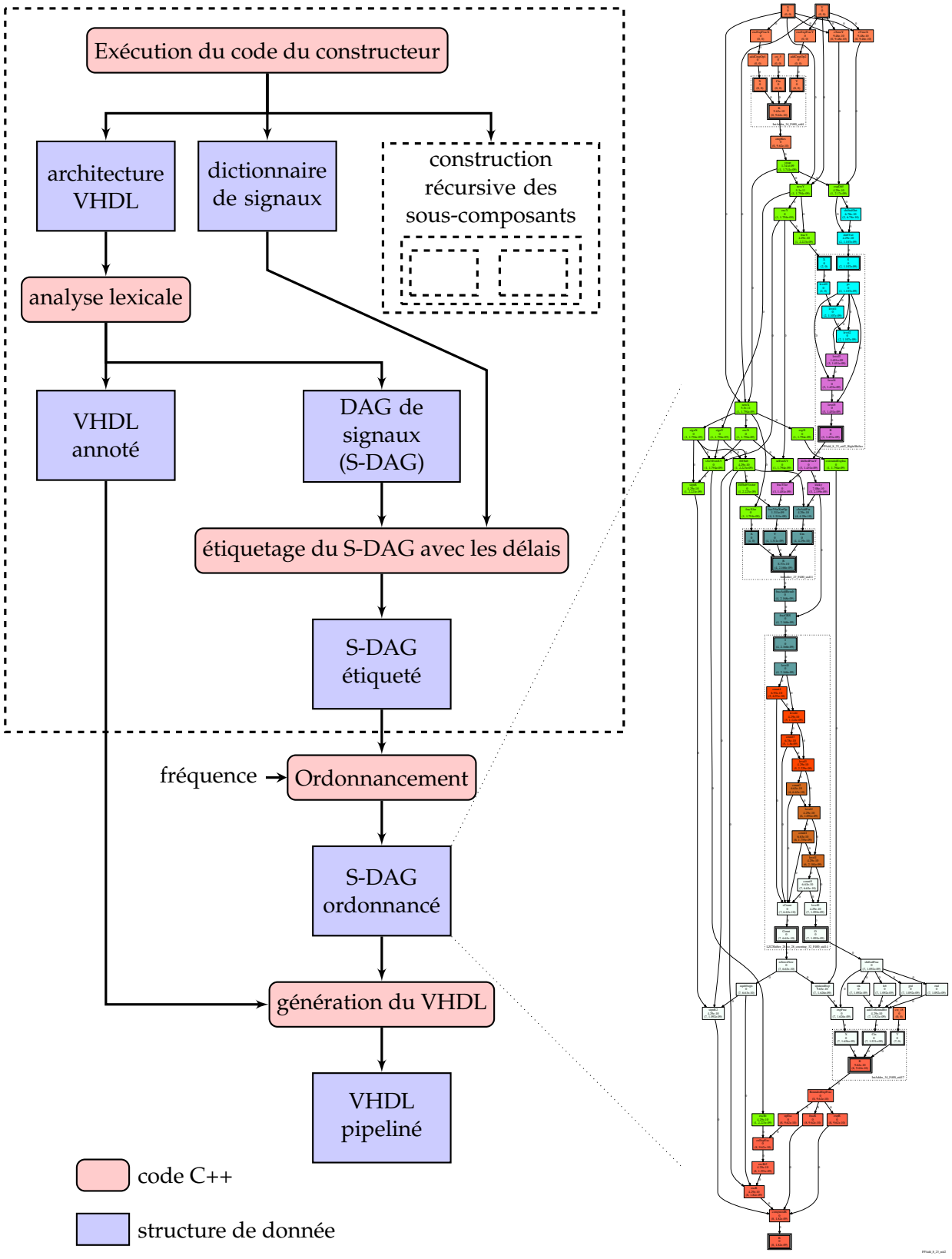
```
vhdl << declare("signX")  << "<= newX("<<wE+wF<<");" ;
vhdl << declare("signY")  << "<= newY("<<wE+wF<<");" ;
vhdl << declare("effSub") << "<= signX xor signY;"   ;
(...)
vhdl << declare("excR",2)
     << " (...) when effSub ='1' (...)"
```

(a) Code du constructeur

```
signX<= newX(31);
signY<= newY(31);
effSub <= signX xor signY;
(...)
excR <= (...)
    when effSub='1' (...);
```

(b) Code généré

FIGURE 2 – Un exemple de génération de code dans FloPoCo



(a) Vue d'ensemble de la construction d'un pipeline

(b) Exemple de S-DAG

FIGURE 3 – Construction et ordonnancement du graphe des signaux

2. Construction du graphe acyclique direct des signaux

L'exécution du constructeur construit donc un dictionnaire des signaux. Il est aussi facile d'isoler les *utilisations* de signaux : il suffit d'un analyseur lexical de VHDL (voir figure 3a), qui isole les identifiants. On récupère ainsi non seulement les noms de signaux mais aussi les noms de fonctions, de variables, d'entités, etc. Un filtrage par le dictionnaire des signaux permet d'isoler les noms de signaux qui apparaissent en partie droite d'une affectation comme `<=`.
Ainsi, le DAG que nous allons considérer n'est pas un DAG de composants, mais un DAG de signaux (S-DAG). Les nœuds de ce graphe sont les noms de signaux déclarés. Il y a une arête d'un signal `A` vers un signal `B` lorsque `A` apparaît en partie droite de la définition de `B`. Par exemple, le code de la figure 2 définit les arêtes `newX`→`signX`, `newY`→`signY`, `signX`→`EffSub`, `signY`→`EffSub` et `EffSub`→`excR`. Le lecteur muni d'une bonne loupe pourra chercher ces arêtes sur le S-DAG complet de l'additionneur flottant simple précision, qui est donné Figure 3b.

2.1. Estimation du temps de calcul

Le S-DAG ainsi construit contient juste les dépendances de données d'un signal à un autre. À présent, intéressons-nous au temps de calcul. En principe, ce sont les portes de base et les fils qui introduisent les délais qui, accumulés, définissent le chemin critique du circuit combinatoire. Dans un flot de synthèse classique, on aura ce type d'information (par exemple le délai induit par le `xor` de la figure 2b) dans les rapports de synthèse. Si on veut s'en servir pour pipeliner le design, on devra itérer sur toute la synthèse, ce qui est possible en principe, mais lourd. Cette solution est d'autant plus difficile à mettre en œuvre que les étapes d'optimisation logique et de *technology mapping* transforment, parfois en profondeur, l'architecture.
Une autre option est de faire confiance à des outils de *retiming* à la Leiserson et Saxe [8]. Il suffit de placer le bon nombre de registres en sortie du circuit (sous forme d'un registre à décalage de la sortie), et ces outils les pousseront à l'intérieur du circuit. Mais il reste la question : comment déterminer ce bon nombre de registres qui permettra de pipeliner un opérateur donné pour une fréquence donnée sur un FPGA donné ? Encore une fois, une approche par essais et erreurs itérant sur la synthèse du circuit est lourde à mettre en œuvre.

2.2. Expression des délais paramétrés

La solution mise en œuvre par FloPoCo est moins précise, mais beaucoup plus rapide que les deux précédentes. Elle demande au concepteur d'étiqueter chaque affectation à un signal en partie gauche du `<=` avec une estimation du délai de calcul de la partie droite. Cet étiquetage a naturellement sa place dans la fonction `declare()` : elle prend un argument optionnel (flottant) qui décrit ce délai.
L'objet `target` est une instance de la classe `Target` qui fournit de nombreuses autres méthodes d'estimation de délai. Par exemple `Target::adderDelay(int n)` retourne le délai d'un additionneur de n bits. C'est ne fonction affine de n sur les cibles Xilinx, et une fonction un peu plus compliquée sur les cibles Altera. Cette complexité est encapsulée dans les classes qui dérivent de `Target` pour modéliser les différents FPGA que supporte FloPoCo. On trouve aussi des méthodes pour le délai de tables de valeurs précalculées, etc.
Inutile par contre de modifier les deux premières lignes de la figure 2a : el les ne réalisent aucun calcul, leur délai restera nul.

À l'usage, l'ajout des délais au générateur de code combinatoire est assez naturel (il est au niveau d'abstraction du VHDL et capture une réflexion qu'il faut avoir pour écrire des opérateurs performants). Ce travail est aussi assez rapide, de l'ordre de quelques minutes pour avoir un premier jet pour un opérateur, à quelques heures si on prend le temps de le raffiner en le confrontant au résultat de synthèses sur différentes cibles. À la fin de cette étape (lorsqu'on sort du cadre en pointillé sur la figure 3a), on dispose d'un S-DAG où chaque nœud représente un signal étiqueté par son temps de calcul.
Le passage par les sous-composants est assez naturel, et on ne le détaillera pas : FloPoCo crée une instance de DAG pour chaque instance de sous-composant, et connecte ensuite les graphes – on voit trois sous-composants sur la figure 3b, ce sont les boîtes encadrées en pointillé. La figure 4 montre un zoom sur le composant de normalisation (LZCShifter). Dans chaque boîte, la première ligne est le nom du signal, et la seconde est le délai de calcul de ce signal à partir de ses signaux en partie droite. La troisième ligne et la couleur décrivent l'ordonnancement dont nous allons parler maintenant.

3. Construction du pipeline à partir du graphe des signaux

3.1. Le temps lexicographique

Dans un circuit pipeliné, le temps écoulé depuis l'arrivée des entrées est mesuré dans un système lexicographique (c, d). Ici c est un entier qui compte le nombres de registres depuis une entrée, et d est un réel qui mesure le délai du chemin critique depuis le dernier registre. La troisième ligne de chaque boîte du DAG représenté Figure 4 donne ce temps lexicographique. Pour deux signaux a et b, l'ordre temporel des calculs de a et b est défini par

$$(c_a, d_a) < (c_b, d_b) \quad \text{ssi} \quad c_a < c_b \text{ ou } (c_a = c_b \text{ et } d_a < d_b).$$

3.2. Ordonnancement lexicographique du S-DAGw

L'ordonnancement (voir Figure 3a) consiste à parcourir le graphe en assignant un instant lexicographique à chaque signal pour respecter la contrainte de fréquence. Il s'agit essentiellement d'un algorithme glouton au plus tôt.
Soit d_{obj} l'inverse de la fréquence cible : c'est le délai de calcul à ne pas dépasser dans un cycle, c'est-à-dire entre deux registres de pipeline.
Pour étiqueter un signal a, on calcule d'abord le maximum lexicographique (c, d) des signaux dont a dépend directement. On ajoute alors à d le délai δ associé (par `declare()`) au calcul de a. Si $d + \delta < d_{obj}$, le calcul de a va être possible dans le même cycle. Par contre si $d + \delta \geq d_{obj}$, il n'est pas possible de calculer a dans le même cycle que son prédécesseur le plus tardif : il faut insérer un registre avant de commencer le calcul de a. Ce registre ajoute 1 à c, et remet à zero le chemin critique d : l'instant lexicographique associé à a est alors $(c + 1, \delta)$.

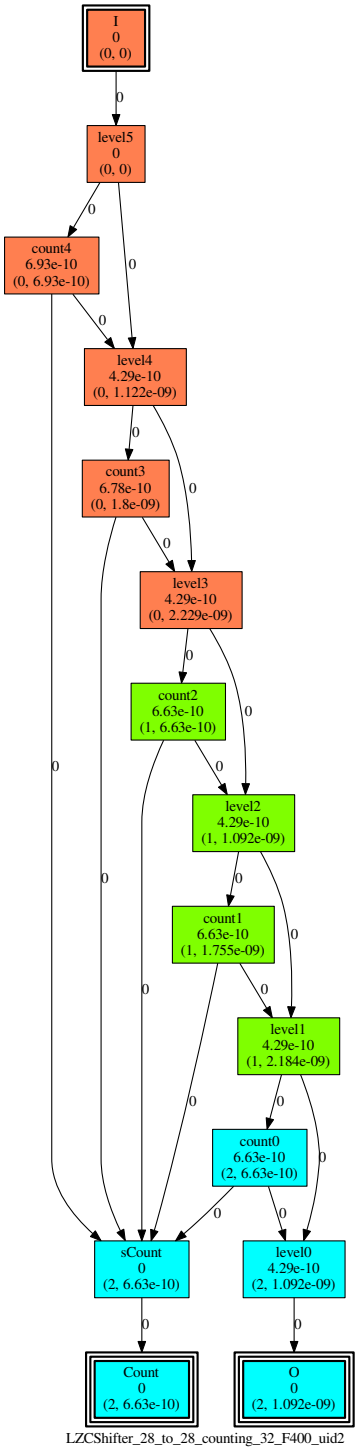


FIGURE 4 – l'unité de normalisation, générée isolément (son ordonnancement est différent de celui qu'elle a au sein de la figure 3b)

Sur les figures 3b et 4, les signaux de la même couleur sont ceux calculés au même cycle.

3.3. Génération du VHDL pipeliné

Une fois tout le S-DAG ordonnancé, il ne reste plus qu'à reparcourir le VHDL en insérant le bon nombre de registres sur les arêtes. Cela se fait simplement en remplaçant les noms de signaux en partie droite par leur version retardée. Par exemple, pour `A <= X;`, si $c_A - c_X = 2$ alors, dans le VHDL produit `X` est remplacé par `X_d2`, ce qui donne `A <= X_d2;`. Un registre à décalage sera aussi produit automatiquement pour produire toutes ces versions retardées de signaux. Le VHDL reste ainsi parfaitement lisible. Cette partie est similaire à ce qui est décrit dans [4].

4. Résultats et conclusion

Le gros du travail présenté ici a porté sur le cœur de FloPoCo : la construction du S-DAG et son ordonnancement sont implémentés dans les classes `Signal` et `Operator`. Nous avons aussi porté dans ce nouveau cadre l'opérateur d'addition flottante `FPAddSinglePath` et toutes ses dépendances (`IntAdder`, `Shifter`, `LZOCShifterSticky`).

4.1. Simplification du code des opérateurs

Dans les versions précédentes de FloPoCo, l'assistance à la construction du pipeline était le résultat d'une évolution qui avait vu apparaître d'abord la gestion des cycles et ensuite la gestion des délais dans un cycle. L'abstraction offerte reflétait cet empilement historique, par exemple les délais de calcul que nous attachons désormais à `declare()` devaient jadis être ajoutés à un chemin critique global qui n'était pas syntaxiquement relié à un chemin critique dans le code et ne résistait pas, par exemple au déplacement et copi-collage de code. Par ailleurs la synchronisation entre signaux devait être explicitée dans le code des constructeurs : il était donc possible de produire des pipelines incorrects en cas d'erreur de programmation.
Le principal résultat de ce travail est donc d'avoir permis une simplification du code des opérateurs. C'est une simplification en terme de nombre de lignes de code : à titre d'exemple, `FPAddSinglePath.cpp` est passé de 1900 à 1800 lignes. C'est surtout une simplification conceptuelle, puisqu'on ne doit plus se préoccuper que d'une information locale de délai. Toutes les questions globales (chemin critique, ordonnancement et synchronisation) sont gérées par l'outil.
Il faut aussi noter que l'outil peut produire le graphe de dépendances au format `dot` : c'est ainsi qu'ont été obtenues les figures 3b et 4. C'est une aide précieuse au développement.

4.2. Amélioration de la performance

La table 1 donne quelques résultats de synthèse pour `FPAddSinglePath` dans la version courante de FloPoCo, et avec le nouveau pipeline. Il faut souligner qu'il n'y a eu aucun changement dans le VHDL combinatoire entre ces deux expériences. On constate que la fréquence du pipeline obtenu est beaucoup plus proche de la spécification, et qu'à latence égale la performance est améliorée et la consommation de ressources réduites.

4.3. Travaux futurs

Le travail sur le moteur de pipeline n'est pas tout-à-fait terminé, il reste à y intégrer les blocs durs (DSP et mémoire) où les registres ont une position fixe. Les tas de bits [1] posent un autre problème : idéalement, leur DAG est construit en fonction du timing des entrées. Ensuite, il faudra reprendre tous les opérateurs de FloPoCo.
Cette refonte de l'outil va aussi permettre d'exprimer proprement en FloPoCo des opérateurs dont la fonction numérique fait appel à une mémoire, comme les filtres de traitement du signal.

Bibliographie

1. Brunie (N.), de Dinechin (F.), Istoan (M.) et Sergent (G.). – L'arithmétique sur le tas. – In *Conférence en parallélisme, architecture et système (COMPAS)*, janvier 2013.
2. Chung (A. J.), Cobden (K.), Jervis (M.), Langhammer (M.) et Pasca (B.). – Tools and techniques for efficient high-level system design on fpgas. *CoRR*, vol. abs/1408.4797, 2014.
3. de Dinechin (F.), Detrey (J.), Creţ (O.) et Tudoran (R.). – *When FPGAs are better at floating-point than microprocessors*. – Rapport technique nensl-00174627, ÉNS Lyon, 2007. http ://prunel.ccsd.cnrs.fr/ensl-00174627.
4. de Dinechin (F.), Klein (C.) et Pasca (B.). – Generating high-performance custom floating-point pipelines. – In *Field Programmable Logic and Applications*, pp. 59–64. IEEE, août 2009.
5. 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.
6. Ercegovac (M. D.) et Lang (T.). – *Digital Arithmetic*. – Morgan Kaufmann, 2003.
7. Gaide (B.). – Methods of pipelining a data path in an integrated circuit, novembre 18 2014. US Patent 8,893,071.
8. Leiserson (C. E.) et Saxe (J. B.). – Retiming synchronous circuitry. *Algorithmica*, vol. 6, n1, 1991, pp. 5 – 35.
9. Venkataramani (G.) et Gu (Y.). – System-level retiming and pipelining. – In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pp. 80–87, May 2014.

spécif. FloPoCo		VHDL produit par FloPoCo 4		VHDL produit par newPipeline	
		performance	ressources	performance	ressources
(8, 23)	400MHz	6 cycles @ 297 MHz	397R + 495L	6 cycles @ 419 MHz	401R + 418L
	300MHz	3 cycles @ 189 MHz	268R + 442L	4 cycles @ 277 MHz	294R + 467L
	200MHz	2 cycles @ 189 MHz	231R + 456L	3 cycles @ 223 MHz	246R + 427L
	100MHz	1 cycle @ 155 MHz	188R + 527L	1 cycle @ 119 MHz	145R + 383L
(11, 52)	300MHz	6 cycles @ 240 MHz	737R + 897L	7 cycles @ 311 MHz	789R + 1085L
	200MHz	4 cycles @ 175 MHz	654R + 958L	4 cycles @ 229 MHz	620R + 867L
	100MHz	2 cycles @ 150 MHz	445R + 948L	2 cycles @ 133 MHz	402R + 880L

TABLE 1 – Comparaison sur le composant FPAdd (simple et double précision) entre l'ancien et le nouveau FloPoCo. Résultats après synthèse sur Virtex6 (6vhx380tff1923-3) avec ISE 14.7. Les options de *register balancing* sont débranchées. Dans ces résultats, n cycles signifient n barrières de synchronisations insérées, donc $n + 1$ étages de pipeline.