**Mathieu Giraud**
ÉCOLE NORMALE SUPÉRIEURE DE LYON
magiraud@ens-lyon.fr

Stage effectué au sein du Laboratoire des Systèmes Logiques (LSL),
à l'École Polytechnique Fédérale de Lausanne (EPFL, Suisse)

encadré par **Jean-Luc Beuchat**
LSL, EPFL, LAUSANNE
jlbeucha@lslsun.epfl.ch

# Arithmétique en-ligne synchronisée

## théorie et applications sur FPGA

**14 septembre 2000**

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

ÉCOLE NORMALE SUPÉRIEURE DE LYON

# Table des matières

# Introduction

Ce rapport fait suite à un stage de deux mois effectué au LSL, stage que nous détaillerons en première partie.

L'arithmétique en-ligne, présentée en deuxième partie, est un calcul sériel poids fort en tête et pipeliné au niveau du bit. Sa puissance réside dans ce parallélisme de données. Afin de concevoir des circuits de calcul efficace, il faut tenter de maintenir autant que possible ce pipeline.

Pour cela nous présentons en troisième partie un algorithme permettant de connaître précisément les données temporelles de ce circuit puis de le synchroniser globalement et localement. Nous émettons ensuite quelques idées permettant d'améliorer encore ce pipeline.

## Première partie

# Deux mois de stage au LSL

Ce rapport est consécutif à un stage que j'ai effectué début 2000 au Laboratoire des Systèmes Logiques de l'EPFL à Lausanne. Nous présentons ici les conditions de travail de ce stage.

*Vue des laboratoires d'informatique de l'EPFL*

### Le LSL et ses thèmes de travail

Le Laboratoire des Systèmes Logiques de l'EPFL (`http://lslwww.epfl.ch/`), dirigé par le Pr. Daniel Mange, oriente sa recherche autour de deux axes principaux :

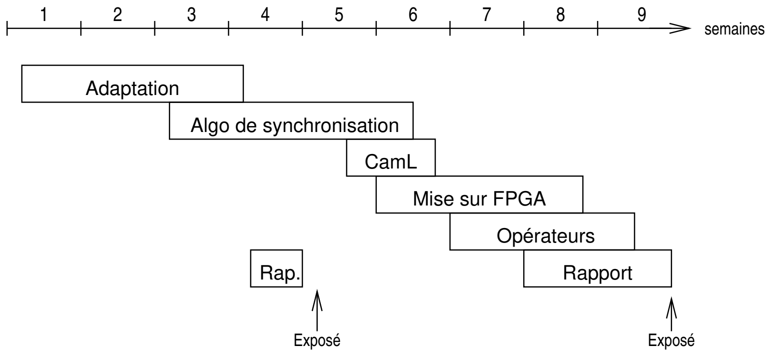
- les **systèmes bio-inspirés**, axe duquel est né le projet "Embryonnique".
- les **systèmes reconfigurables**, avec notamment des études sur les FPGAs.

## Sujet et déroulement du stage

**Jean-Luc Beuchat**, qui prépare sa thèse au LSL, m'a proposé de travailler avec lui sur l'"arithmétique en-ligne sur les FGPAs" dans le cadre de la thématique "systèmes reconfigurables". Le sujet du stage, rédigé en janvier 2000, était libellé ainsi :

> *Le travail proposé pour un stage de deux mois consiste :*
>
> 1. *A découvrir les principes de l'arithmétique en-ligne. Une bibliothèque VHDL d'opérateurs en-ligne est disponible au LSL.*
> 2. *A se familiariser avec les outils de développement utilisés au LSL (Synopsys, outils de placement-routage de Xilinx, ...). (2 semaines pour ces deux premières étapes)*
> 3. *A contribuer à l'un des projets en cours (au choix de l'étudiant). Ce travail pourra consister à développer de nouveaux opérateurs, réaliser une phase de l'un de ces projets ou tester un système sur une carte à FPGA. Cette partie du travail dépendra de l'avancement de nos projets. (4 semaines)*
> 4. *A rédiger un rapport de stage et à préparer une présentation du travail réalisé (2 semaines)*

Le sujet initial était volontairement flou et a permis une grande souplesse dans mon travail. Plus précisément, on peut distinguer quelques phases dans ce travail, comme le montre la figure ci-dessous.




Une grande partie du stage a tout d'abord été conscacrée à la familiarisation avec ce sujet :

– lecture d'articles

– ré-apprentissage du VHDL

– découverte de Synopsis, outil de développement utilisé au LSL

Je me suis ensuite attaché à trouver un algorithme de synchronisation pour des réseaux d'arithmétique en-ligne, algorithme qui sera détaillé en 3$^{e}$ partie.

Enfin, une partie plus pratique a été mise en oeuvre, notamment par l'implémentation de circuits obtenus sur carte FPGAs, ce qui a soulevé d'importants problèmes techniques. De plus, j'ai pu réfléchir à l'élaboration d'opérateurs à faible délai de pipeline.

Tout au long du stage, j'ai eu aussi l'occasion de rencontrer les autres participants du LSL et de lire avec profit quelques documents sur leurs travaux.

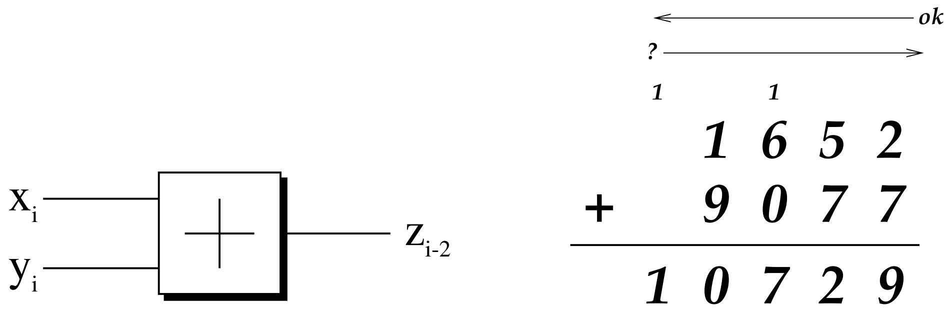
## Deuxième partie

# Présentation de l'arithmétique en-ligne

## I Du calcul en série à l'arithmétique en-ligne borrow-save

### I.1 Les calculs en série : un pipeline au niveau du bit

Les calculs "en série" sont des calculs systoliques au niveau du chiffre, c'est à dire des circuits où les données se propagent chiffre après chiffre. Ces calculs sont en fait très similaires à ceux effectués par l'homme : il "pose" une opération puis donne le résultat chiffre par chiffre, utilisant le même algorithme pour tous les chiffres. Ceci permet d'utiliser de petits opérateurs simples mais qui calculeront en pipeline sans (presque) s'arrêter, d'où une grande efficacité du circuit.




### I.2 L'arithmétique en-ligne

Dans ces calculs en série, la représentation des données joue un rôle important. On peut tout d'abord faire circuler les données poids faible en tête (*Least Signifiant Digit First*, LSDF), ce qui s'utilise naturellement pour faire des additions et multiplications. Mais les divisions ou les comparaisons se révèlent alors impossibles sauf en "stockant" préliminairement tous les opérandes, ce qui fait alors perdre le bénéfice du pipeline.

Si on considère maintenant le calcul *Most Signifiant Digit First* (MSDF) où les données circulent poids fort en tête, les opérations précédentes peuvent se réaliser facilement. Le seul problème reste pour les opérations "à propagation de retenue" comme les additions ou les multiplications, mais Avienzis [Avi61] a montré que ces problèmes pouvaient être évités par l'utilisation d'une base redontante. Ce mode de calcul permet ainsi d'effectuer toutes les opérations usuelles : Trivedi et Ercegovac ont ainsi proposé dans [TE77] l'arithmétique en-ligne comme **calcul en série où les données, codées dans une base redontante, se propagent poids fort en tête**.

Diverses variantes de l'arithmétique en-ligne "traditionnelle" ont été proposées, comme l'arithmétique flottante de [DFAMY92] puis de [Yeh93] qui s'appuie sur une représentation "à la IEEE-754", avec une mantisse et un exposant de taille fixe. Enfin, des représentations plus originales ont été imaginées, comme fractionnaires [Dup] ou p-adiques.

Une application intéressante de l'arithmétique en-ligne se situe dans le *traitement de données converties depuis des signaux analogiques*. En effet, les convertisseurs analogique-numérique fournissent leur résultats de façon sérielle en MSDF. Cette arithmétique permet alors d'exploiter au mieux les données [TD97]. Un autre aspect significatif de l'arithmétique en-ligne se trouve dans sa facile mise en oeuvre sur des *systèmes reconfigurables* tels que des FPGAs. Cette thématique est étudiée au LSL.

## Représentation des données en base redontantes et en borrow-save

Dans une base usuelle $b$, on représente chaque réel $x$ de manière unique par une suite presque nulle $(\alpha_i)_{i\in\mathbb{Z}}$, où chaque $\alpha_i$ est un entier appelé chiffre appartenant à $\{0, 1, \cdots, b-1\}$, cette suite vérifiant :

$$x = \sum_{i=-\infty}^{+\infty} \alpha_i b^i$$

En base redondante $b_a$ (où $a \leq b-1$), on utilise cette fois-ci des chiffres $\alpha_i$ appartenant à l'ensemble $\{-a, \cdots, +a\}$. Par commodité, nous écrirons par la suite le chiffre $-k$ sous la forme de $\bar{k}$. Cette base permet de réprésenter tous les nombres si $2a+1 \geq b$. De plus, si $2a+1 > b$, alors certains nombres ont plusieurs représentations possibles : le nombre $5274$ en base $10$ peut ainsi s'écrire $51\bar{3}4$, $1\bar{5}1\bar{3}4$, $51\bar{2}\bar{6}$ ou $1\bar{5}1\bar{2}\bar{6}$ en base $10_6$.

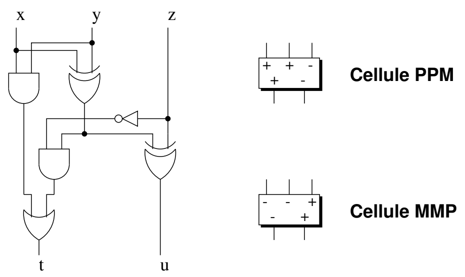
Avienzis [Avi61] montra en 1961 que ces bases permettent d'effectuer des additions sans aucune propagation de retenue. L'approche d'Avinzis impose cependant $2a+1 \geq b+2$, ce qui est possible dès que $b \geq 3$. En base 2, d'autres algorithmes sont possibles dans la base redondante $\{-1, 0, +1\}$ nommée **borrow-save** qui a été proposée dans [GHM89].

Les 4 opérations et la racine carrée sont réalisables dans ce système – on pourra consulter par exemple [Tis97] ou [Beu]. Certains opérateurs comme la multiplication ou la division ont cependant une taille dépendante de la taille des entrées; Muller a d'ailleurs montré dans [Mul94] que seules les fonctions affines par morceaux à coefficients rationnels peuvent être calculées par un opérateur de taille constante.

Concrètement, un chiffre borrow-save $c$ va se matérialiser par une paire de bits $(c^+, c^-)$ telle que $c = c^+ - c^-$. On a ainsi $1 = (1, 0)$, $\bar{1} = (0, 1)$ et $0 = (0, 0) = (1, 1)$.

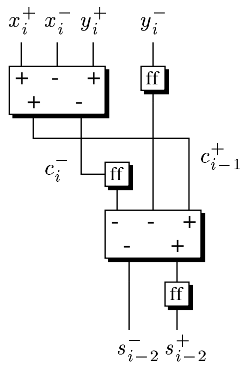
Il est aussi possible d'effectuer des calculs en carry-save (base $\{0, 1, 2\}$), mais on perd alors la possibilité d'écrire simplement les nombres négatifs.

## La cellule PPM et l'additionneur borrow-save de [GHM89]




A partir des entrées $x$, $y$ et $z$, cette cellule calcule $t = \mathsf{maj}(x, y, \bar{z})$ et $u = x \oplus y \oplus z$, ce qui donne $x + y + (-z) = 2t - u$ : c'est une cellule PPM *(Plus-Plus-Moins)*.

La même cellule peut être utilisée en MMP en considérant que l'on effectue le calcul $(-x) + (-y) + z = -2t + u$. Grâce à deux de ces cellules et à trois registres, il est aisé de réaliser l'additionneur en-ligne suivant le circuit ci-dessous.




On se réfererra à [BDKM94] pour une description complète du fonctionnement de l'additionneur cycle par cycle.

# II Aspects temporels

Nous avons déjà évoqué le fait que la puissance de l'arithmétique en-ligne résidait dans le pipeline au niveau du bit. Si on veut concevoir des circuits de calcul efficace, il faut donc tenter de maintenir autant que possible ce pipeline. Or les études précédentes se sont surtout attachées à calculer le délai d'une calcul en-ligne. Par exemple, Mario Fiallos Aguilar construit dans sa thèse [Agu94] des machines complexes composées de plusieurs opérateurs en-ligne et effectue des calculs de synchronisation sur quelques circuits.

Nous allons proposer dans la suite d'évaluer aussi à quel délai plusieurs calculs peuvent *s'enchaîner*.

## II.1 Paramètres d'un opérateur

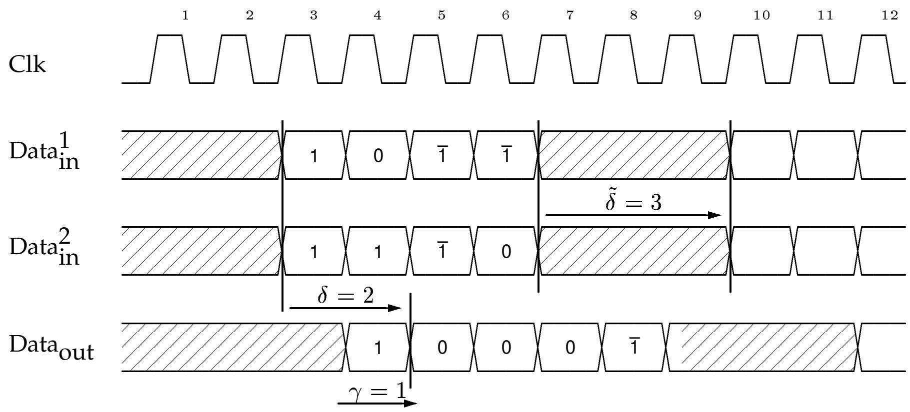
Nous commençons par poser quelques définitions.

Il est classique d'utiliser la notation $\delta$ pour le *délai* d'un opérateur en-ligne, suposé constant. (Dans [Agu94] est tentée une approche avec des délais non constants).

Nous considérerons aussi son *augmentation* $\gamma$, nombre de chiffres de poids fort qu'il rajoute. Enfin nous avons besoin d'un dernier paramètre afin de décrire l'enchainement des calculs dans un même opérateur : nous définissons le *délai de pipeline* $\tilde{\delta}$ comme nombre de cycles d'horloge supplémentaires entre deux entrées (après l'introduction du dernier bit d'entrée).

*Exemple:*

Nous utilisons l'additionneur décrit précédemment pour calculer $10\bar{1}\bar{1} + 11\bar{1}0 = 1000\bar{1}$. Le chronogramme ci-dessous montre, pour chaque cycle d'horloge, l'état des signaux. Ceux-ci peuvent être à 0, à 1, ou à $\bar{1}$. Ils peuvent aussi être non significatifs (en hachuré sur le dessin). La validité des signaux peut être gérée par un autre signal de controle, comme expliqué dans [Beu].




Les bits de poids $i-2$ sortent dans l'opérateur lorsque ceux de poids $i$ rentrent, ce qui donne un délai $\delta = 2$. De plus, l'addition génère un bit de poids fort supplémentaire ($\gamma = 1$). Enfin, on attend classiquement que toute l'opération soit terminée afin de réinitialiser les registres – ici au cycle 9 –, ce qui implique $\tilde{\delta} = 3$. $\triangle$

Le triplet $(\delta, \gamma, \tilde{\delta})$ est appelé *caractéristique* de l'opérateur.

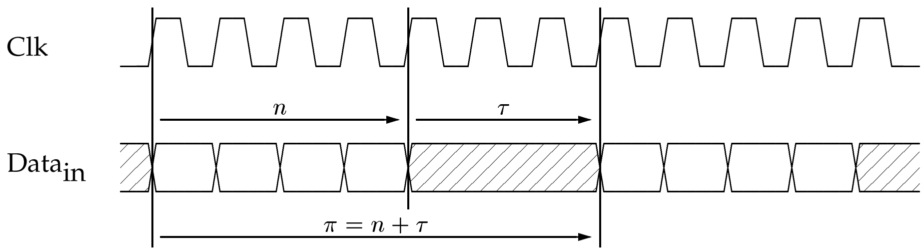
## II.2 Méta-cycles de données

L'arithmétique en-ligne est une arithmétique réalisant un pipeline "au niveau du chiffre" : un cycle d'horloge correspond usuellement à un chiffre codé en borrow-save.

Cependant, lorsqu'on dispose d'un circuit d'arithmétique en-ligne, il devient crucial de savoir à quelle vitesse sont traitées les données et "combien de temps" doit s'écouler entre deux jeux de données. En effet, il faut souvent laisser de "l'espace" entre plusieurs données distinctes (pour laisser le temps aux pipelines du "chiffre" de se vider, entre autres).

Nous avons donc trouvé qu'il était intéressant de définir un **méta-cycle** sur les données : ce méta-cycle correspond à l'introduction des données dans le circuit et s'étend ainsi sur plusieurs cycles d'horloge.

Pour simplifier, nous nous plaçons dans l'hypothèse **(NormData)** : les entrées sont toutes de même taille $n$, et cette taille sera prise comme référence partout dans le circuit (mais les opérateurs pourront générer, on le verra, des tailles supérieures).




De façon plus concrète, on parle, à l'entrée d'un circuit, du **délai supplémentaire** $\tau$ et de sa **méta-période** $\pi = n + \tau$, comme le montre le chronogramme ci-dessus.

# Troisième partie

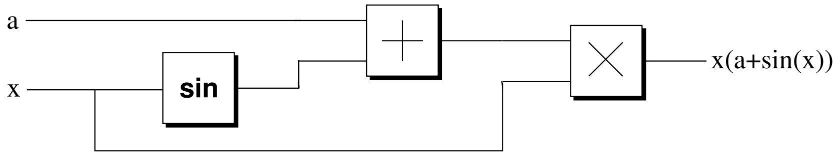
# Synchronisation d'un ensemble d'opérateurs

## III Problème de la synchronisation

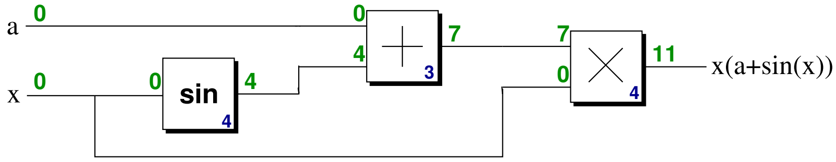
Pour qu'un circuit d'arithmétique fonctionne convenablement, il faut **retarder** certaines entrées en utilisant plusieurs registres en files d'attentes (FIFO). Dans ce paragraphe, nous formalisons ce problème après en avoir donné un premier exemple.

### III.1 Nécessité de la synchronisation : un exemple

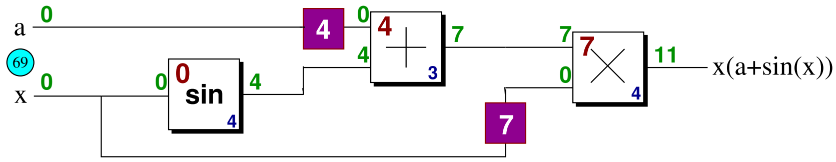
Considérons le circuit d'artihmétique en-ligne suivant, destiné à calculer l'expresion $x(a + sin(x))$ à partir des entrées $a$ et $x$ :




Nous supposons pour simplifier que tous les opérateurs sont normalisés ($\gamma = 0$). Nous noterons "$+^n$" et "$\mathsf{sin}^n$" ces opérateurs. Nous considérons que $\delta_{+^n} = 3$ et $\delta_{\mathsf{sin}^n} = \delta_\times = 4$. La synchronisation de ce circuit s'effectue facilement en prenant les "plus longs délais" rencontrés le long du circuit :




L'opérateur +, par exemple, peut recevoir son première opérande au temps 0 et son second au temps 4. Il va donc falloir retarder le premier opérande de 4 cycles, et cet additionneur va donc "travailler" à partir du 4e cycle :




On se rend ainsi compte que les données seront disponibles en sortie avec un délai $\delta = 11$. De plus, en utilisant des opérateurs classiques avec $\tilde{\delta} = \delta + 1$, on voit que l'opérateur $\mathsf{sin}$ limite le circuit à un délai supplémentaire de $\tau = 5$. Ainsi, si les données sont de 64 bits, on pourra envoyer une donnée dans le circuit tous les $\pi = \tau + n = 69$ cycles.

## III.2 Énoncé du problème

Dans la suite de cette partie, nous exposerons un algorithme permettant de réaliser une synchronisation d'un circuit **quelconque**, c'est à dire où il peut y avoir des augmentations ainsi que des cycles, comme dans le calcul d'une suite $u_n = f(u_{n-1})$ par exemple.

Nous cherchons donc à résoudre le **problème de synchronisation :**

> *Etant donné un circuit d'arithmétique en-ligne $\mathcal{C}$ et une taille de données $n$, placer des registres dans le circuit pour que "tout fonctionne" et évaluer précisément la méta-période $\pi$ minimale, le délai supplémentaire $\tau$ correspondant ainsi que le délai total en sortie $\delta^*$.*

## III.3 Graphe d'un circuit

Nous décrivons ici notre formalisation d'un circuit d'arithmétique en-ligne $\mathcal{C}$.

L'ensemble des opérateurs en-ligne (bibliothèque) est noté $\mathcal{L}$. On définit aussi deux symboles spéciaux, $\perp$ et $\top$, qui schématisent respectivement l'entrée et la sortie de données.

Le circuit $\mathcal{C}$ peut être vu comme un **graphe orienté** $\mathcal{C} = (\mathcal{V}, \mathcal{E})$, où :

- $\mathcal{V} = \mathcal{L} \cup \{\perp, \top\}$ est l'ensemble des sommets.
- $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ est l'ensemble des arêtes reliant les sommets, représentant les connexions entre les opérateurs. Ces connexions sont orientées : une arête $e = (op_1, op_2)$ représente des données sortant de $op_1$ pour aller dans $op_2$.

On appelle "entrées" d'un opérateur $v \in \mathcal{V}$ les arêtes $(v', v) \in \mathcal{E}$.

De plus, chaque sommet $v$ de $\mathcal{V}$ est étiqueté par son délai $\delta_v$, son délai de pipeline $\tilde{\delta}_v$ et son augmentation $\gamma_v$. Les arêtes sont elles étiquetées par un entier $\omega_e$ représentant leur *méta-délai* ou *méta-retard*. Cette notion sera expliquée dans les exemples suivants. Enfin, pour une entrée $\perp_i$, les quantités $\delta_{\perp_i}$, $\tilde{\delta}_{\perp_i}$ et $\gamma_{\perp_i}$ peuvent représenter des contraintes externes sur le circuit.

Un opérateur usuel requiert souvent que le dernier bit de la précédente opération l'ait quitté avant de relancer un cycle de reset puis une nouvelle opération, ce qui donne $\tilde{\delta} = \delta + 1$. Nous verrons dans la quatrième partie comment définir des opérateurs pour que $\tilde{\delta} < \delta + 1$.

Pour un chemin $c = (v_1, v_2, \dots v_n)$ du graphe, on définit de façon naturelle son poids total $\delta^*(c)$ et son méta-délai total $\omega^*(v)$.

Enfin, certains opérateurs $v$ sont non normalisés, et donc pour ceux-ci $\gamma_v > 0$. De plus un *normalisateur universel*, proposé dans [Tis97], permet de normaliser avec un délai 1 toute entrée, autrement dit d'enlever tous les chiffres supplémentaires. Il existe aussi un normalisateur de délai nul, mais il possède un chemin critique très long, ce qui le rend inefficace. Par convention, on posera $\gamma = -\infty$ pour ces normalisateurs universels.

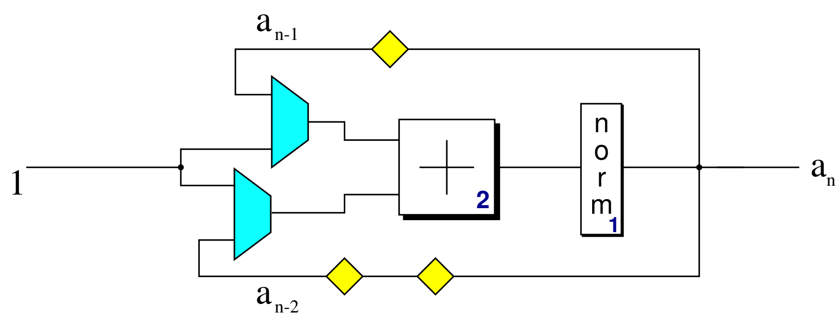
#### Exemples

Avec ces notations, le calcul de $x(a + sin(x))$ où tous les opérateurs sont normalisés se transcrit en :

$$\mathcal{C}_{\mathsf{test}} = (\mathcal{V}, \mathcal{E})$$
$$\mathcal{V} = \{\perp_a, \perp_x, +^n, \mathsf{sin}^n, \times^n, \top\}$$
$$\mathcal{E} = \{(\perp_a, +^n), (\perp_x, \mathsf{sin}^n), (\mathsf{sin}^n, +^n), (+^n, \times^n), (\perp_x, \times^n), (\times^n, \top)\}$$

$$\forall e \in \mathcal{E}, \omega_e = 0$$

Dans la suite, nous détaillerons aussi le calcul de la suite de Fibonacci $(a_n)$ définie par $a_0 = a_1 = 1$ et $\forall n \geq 2, a_n = a_{n-1} + a_{n-2}$. Cette fois-ci, nous considérerons un additionneur non normalisé auquel nous adjoignons un normalisateur universel. De plus, pour réaliser en "dur" le test $n \geq 2$, nous devons utiliser deux multiplexeurs permettant de choisir entre $u_{n-i}$ et la constante 1.




Le circuit pour cette expression se représente donc ainsi :

$$\mathcal{C}_{\text{fibo}} = (\mathcal{V}, \mathcal{E})$$
$$\mathcal{V} = \{\perp, \mathsf{mux}_1, \mathsf{mux}_2, +, \mathsf{norm}, \top\}$$
$$\mathcal{E} = \{(\perp, \mathsf{mux}_1), (\perp, \mathsf{mux}_2), (\mathsf{mux}_1, +), (\mathsf{mux}_2, +), (+, \mathsf{norm}), r_1, r_2, (\mathsf{norm}, \top)\}$$
$$\text{avec } r_1 = (\mathsf{norm}, \mathsf{mux}_1) \text{ et } r_2 = (\mathsf{norm}, \mathsf{mux}_2)$$

.

Sur le dessin, les losange représentent les "méta-retards" des arêtes de retour $(\omega > 0)$ : l'arête $r_1$ transforme $a_n$ en $a_{n-\omega} = a_{n-1}$, tandis que $r_2$ transforme $a_n$ en $a_{n-2}$. On aura donc :

$$\omega_{r_1} = 1 \qquad \omega_{r_2} = 2$$

$$\forall e \in \mathcal{E}/\{r_1, r_2\}, \omega_e = 0$$

# IV Synchronisation globale

Dans cette section, nous allons minorer la méta-période aux bornes du circuit. Nous nous plaçons dans l'hypothèse **(MétaSynchro)**, c'est-à-dire qu'on impose que tous les opérateurs du circuit aient le même méta-cycle.

Dans le cas où il n'est pas imposé que tous les opérateurs aient le même méta-cycle, le problème est beaucoup plus complexe et constitue un des thèmes de nos recherches actuelles.

Nous allons décrire deux facteurs limitant cette méta-période :

- le degré interne de pipeline de chaque opérateur, combiné à l'augmentation de la taille des données (paragraphes IV.1 et IV.2),
- le temps mis par les données pour parcourir chaque cycle (paragraphe IV.3).

Ceci nous permettra de minorer efficacement le méta-cycle au paragraphe IV.4.

## IV.1 Propagation des augmentations $\gamma$

Nous allons tout d'abord propager la fonction $\gamma$ : $\gamma^*_{op}$, augmentation accumulée à l'entrée de *op*, est le plus long chemin en $\gamma$ depuis les entrées jusqu'à *op*.

*Exemple:*

Dans nos exemples, $\gamma^*_{op}$ est nulle partout dans les deux circuits, sauf à l'entrée du normalisateur norm du Fibonnacci où elle vaut 1.

$\triangle$

Cette définition est possible sous l'hypothèse **(CyclAugNulle)** où tous les cycles du graphe ont un poids en $\gamma$ négatif. Concrètement, ceci impose pour un cycle $C$ que :

- soit $C$ contient au moins un normalisateur universel. Dans ce cas, on peut même enlever ce(s) normalisateur(s), puisqu'ils apportent un poids en $\gamma$ de $-\infty$ *(c'est le cas de notre circuit de Fibonnacci).*
- soit $C$ ne contient pas de normalisateur, mais tous ses opérateurs $v$ sont tels que $\gamma_v = 0$.

Si on avait $\gamma^*(C) > 0$, il faudrait considérer dans chaque cycle des registres de taille variable pour réaliser le calcul, ce qui pourrait constituer un problème intéressant.

## IV.2 Degré de pipeline à un opérateur

En application immédiate de nos définitions, nous avons le résultat suivant :

**Lemme 1** *Soit $op \in \mathcal{O}$ un opérateur du circuit. Pour son fonctionnement interne, cet opérateur a besoin d'un délai supplémentaire de $\tau_{pipeline}(op)$ avec :*

$$\tau_{pipeline}(op) = \tilde{\delta}_{op} + \gamma^*_{op}$$

**Corollaire 1** *Pour que tous les opérateurs du circuit fonctionnent, le délai supplémentaire du circuit $\tau$ doit vérifier $\tau \geq \tau_{pipeline}$, où $\tau_{pipeline}$ est défini par :*

$$\tau_{pipeline} = \max\{\tilde{\delta}_{op} + \gamma^*_{op} | op \in \mathcal{O}\}$$

**Preuve:**

Par définition, l'opérateur nécessite, pour son fonctionnement interne, $\tilde{\delta}_{op}$ cycles d'horloge supplémentaires après l'introduction du dernier bit d'entrée. Cependant, en plus des $n$ bits qui se propagent dans tous le circuit, cet opérateur peut avoir a recevoir un certain nombre de bits supplémentaires $\gamma^*_o p$, ce qui conduit à des données de taille $n + \gamma^*_{op}$. La méta-période minimale aux bornes de l'opérateur devra donc être $\tilde{\delta}_{op} + n + \gamma^*_{op}$, d'où le résultat proposé.

$\square$

*Exemple:*

- Pour notre premier circuit, on a pour tous les opérateurs $\gamma^* = 0$. Donc $\tau_{pipeline}(+^n)) = 4$, $\tau_{pipeline}(\mathsf{sin}^n) = 5$ et $\tau_{pipeline}(\times) = 5$ et $\tau_{pipeline}(\mathcal{C}_{\mathsf{test}}) = \max(\tau_{pipeline}(+^n), \tau_{pipeline}(\mathsf{sin}^n), \tau_{pipeline}(\times)) = 5$.
- Dans le cas du Fibonacci, $\tau_{pipeline}(+) = 3 + 0 = 3$ et $\tau_{pipeline}(\mathsf{norm}) = 1 + 1 = 2$, d'où $\tau_{pipeline} = \max(\tau_{pipeline}(+), \tau_{pipeline}(\mathsf{norm}, \tau_{pipeline}(\mathsf{mux})) = \tau_{pipeline}(+) = 3$.

$\triangle$

## IV.3 Parcours des cycles d'un circuit

Nous avons le résultat suivant :

**Lemme 2** *Soit $C$ un cycle élémentaire de $\mathcal{L}$. Alors, pour que les données puissent se propager à temps dans le cycle, les opérateurs de celui-ci doivent travailler à une méta-période minimale de $\pi_{cycle}(C)$ avec*

$$\pi_{cycle}(C) = \left\lceil \frac{\delta^*(C)}{\omega^*(C)} \right\rceil$$

**Corollaire 2** *Pour que la condition précédente soit applicable à tous les cycles du circuit, la méta-période du circuit $\pi$ doit vérifier $\pi \geq \pi_{cycle}$, où $\pi_{cycle}$ est défini par :*

$$\pi_{cycle} = \max\{\pi_C | C \textit{ cycle}\}$$

**Preuve:**

Considérons un opérateur $op$ de $C$, fonctionnant avec une méta-période $\pi$ : il reçoit donc des données aux dates $t, t+\pi, t+2\pi \ldots$
Or une donnée $d_n$ pénétrant dans l'opérateur à la date $t$ et traversant le cycle $C$ arrivera au plus tôt à la date $t+\delta^*(C)$ à l'itération suivante, sous forme de donnée $d_{n+\omega^*(C)}$ devant être ainsi recue à la date $t+\omega^*(C)\pi$, ce qui impose effectivement $t+\delta^*(C) \leq t+\omega^*(C)\pi$, d'où $\delta^*(C) \leq \omega^*(C)\pi$.

□

*Exemple:*

- Il n'y a pas de cycle dans le premier circuit... donc $\pi_{cycle}(\mathcal{C}_{\mathsf{test}}) = 0$
- Pour le Fibonnacci, nous avons deux cycles élémentaires $C_1$ et $C_2$, tous de délai total $\delta_+ + \delta_{\mathsf{norm}} + \delta_{\mathsf{mux}} = 2+1+0 = 3$. Le cycle $C_i$ s'écrit $(+, \mathsf{norm}, \mathsf{mux_i})$ et fournit le résultat $u_{n-i}$ à l'additionneur. Et ainsi $\pi_{cycle}(C_1) = \lceil \frac{3}{1} \rceil = 3$ et $\pi_{cycle}(C_2) = \lceil \frac{3}{2} \rceil = 2$. Donc $\pi_{cycle} = \max(\pi_{cycle}(C_1), \pi_{cycle}(C_2)) = 3$. △

## IV.4 Minoration du méta-cycle du circuit

Les résultats combinés des deux lemmes précédents nous fournissent une minoration du méta-cycle :

**Lemme 3** *La méta-période $\pi$ et le délai supplémentaire correspondant $\tau$ du circuit global d'arithmétique en-ligne $\mathcal{C}$ utilisé avec des données de $n$ chiffres seront au mieux de :*

$$\pi_{opt} = \max(\pi_{cycle}, \tau_{pipeline} + n)$$

$$\tau_{opt} = \max(\pi_{cycle} - n, \tau_{pipeline})$$

Bien que les définitions de $\pi$ et $\tau$ sont assez proches et paraissent parfois redondantes, nous voyons ici que ces deux variables retracent deux phénomènes assez différents – le temps de parcours des cycles et le délai de pipeline dans les opérateurs.

Nous verrons en section V que cette majoration est optimale, ce qui justifie les notations $\pi_{opt}$ et $\tau_{opt}$. Par ailleurs, nous étudierons au paragraphe VII.1 la dépendance par rapport à $n$ de $\pi_{opt}$ et $\tau_{opt}$.

*Exemple:*

Prenons des données de taille $n = 64$. Pour le premier circuit, nous avons $\pi_{opt} = \max(0, 5+n) = 5+n = 69$ et $\tau_{opt} = 5$. Pour le Fibonnacci, on aura de même $\pi_{opt} = 67$ et $\tau_{opt} = 3$. △

Les deux exemples nous montrent déjà qu'un additionneur suivi d'un normalisateur peut avoir comme délai de pipeline total $\max(\tilde{\delta}_+, \tilde{\delta}_{\mathsf{norm}}) = 3$, alors que nous avions pris initialement pour $\tilde{\delta}_{+^n}$ la valeur $\delta_{+^n} + 1 = 4$. Nous voyons ainsi qu'en laissant **deux initialisations distinctes** pour $+$ et $\mathsf{norm}$ nous pouvons réduire le délai de pipeline pour $+^n$ et considérer que $\tilde{\delta}_{+^n} = 3 < 4 = \delta_{+^n} + 1$. Nous verrons dans le dernier paragraphe comment améliorer encore ce résultat.

# V Propagation des données et synchronisation locale

Ce paragraphe a pour but de construire une synchronisation locale qui parvienne effectivement à réaliser $\pi_{opt}$ et $\tau_{opt}$.

## V.1 Délai total après un opérateur

Supposons que nous connaissons la méta-période $\pi$ du circuit. Cela nous permet de "casser" les cycles : les arêtes $e$ telles que $\omega_e > 0$ peuvent être vues comme une avance temporelle de $\omega_e \pi$ cycles.

En effet, si une donnée $d_n$ arrive à la date $t$ à un opérateur, une donnée arrivant avec $\omega$ méta-cycles de retard, $d_{n-\omega}$, sera présente à la date $t - \omega\pi$.

Ainsi la quantité $t - \omega\pi$ représente la **date d'arrivée** effective des opérandes : nous allons donc définir ce **délai total** après un sommet $v$ comme fonction de "plus long chemin" $\delta^*_v$ par :

- si $v = \perp$, $\delta^*_v = \delta_v$,
- sinon, $\delta^*_v = \delta_v + \max\{\delta'_v - \omega_e\pi | e = (v', v) \in \mathcal{E}\}$

Si $e = (v, v') \in \mathcal{E}$, on parlera aussi de son délai total $\delta^*_e = \delta^*_v$.

**Lemme 4** *Appelons* **(Cohérence)** *l'hypothèse "tous les cycles de $\mathcal{C}$ ont un méta-délai strictement positif et tous les sommets de $\mathcal{V}$ sont atteignables à partir d'une entrée $\perp$"*

*Si cette hypothèse est vérifiée et si $\pi \geq \pi_{cycle}$, alors la définition de $\delta^*_v$ est cohérente et totale. De plus, on peut calculer cette fonction en temps linéaire à $|\mathcal{C}|$.*

**Preuve:**

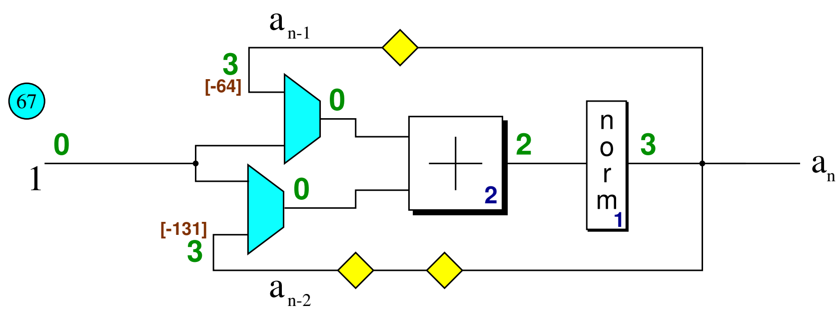
Il nous faut prouver que tous les cycles ont un $\Delta$ positif, ce qui assurera que tous la fonction ne prend jamais des valeurs infinies.
Soit un cycle $C$. On a, par définition, $\pi \geq \pi_{cycle} \geq \lceil \delta^*(C)/\omega^*(C) \rceil$, d'où $\delta^*(C) \geq \pi\omega^*(C)$. Or $\Delta^*(C) = \sum_{e=(v,v')\in C}(\delta_v - \omega_e\pi) = \delta^*(C) - \omega^*(C)\pi \geq 0$. Donc on a bien $\Delta \geq 0$.
Enfin, un parcours en profondeur à partir des sorties permet de calculer effectivement $\delta^*_v$ en temps linéaire.

□

Les conditions du lemme 4 traduisent seulement des propriétés "naturelles" sur de tels circuits : les cycles servent toujours à calculer des expressions du type $u_n = f(u_{n-\omega}, \ldots)$ avec $\omega > 0$, et toutes les portes sont atteignables depuis au moins une entrée, chaque opérateur se réglant sur son entrée "la plus tardive".

*Exemple:*

Pour notre premier circuit, les $\delta^*$ étaient déjà placés sur la figure au début de cette partie. Pour le Fibonnacci avec $n = 64$ et en prenant $\pi = \pi_{opt} = 67$, nous avons ceci :




Seules les deux arêtes "de retour" ont un $\omega$ non nul. Pour ces deux arêtes, nous avons aussi placé sur le dessin la quantité $d - \omega\pi$ (entre crochets).

△

## V.2 Synchronisation locale

Chaque opérateur se réglant sur la date d'arrivée $\delta^*$ la plus tardive de ses entrées, il suffit donc maintenant d'ajouter des registres aux autres entrées :

**Lemme 5** *Soit $\delta_v^*$ le délai total calculé selon le lemme 4. Soit maintenant $op \in \mathcal{O}$ un opérateur en-ligne de délai total $\delta_{op}^*$ dont les $k$ entrées $e_1, e_2 \ldots e_n$ ont pour délai total $\delta_1^*, \delta_2^* \ldots \delta_k^*$.*

*Alors on peut synchroniser localement l'opérateur en adjoignant, devant l'entrée $i$, une FIFO de $r_i$ registres avec :*

$$r_i = \delta_{op}^* - \delta_i^* \geq 0$$

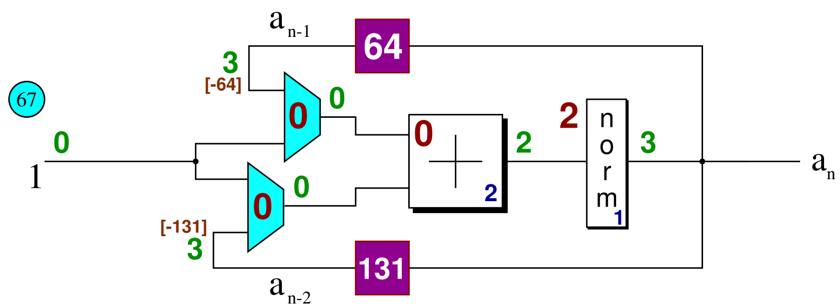
**Preuve:**

D'après la définition de $\delta_{op}^*$, on a effectivement $r_i \geq 0$. En ajoutant alors cette FIFO de longueur $r_i$ à chaque entrée, toutes les entrées vont se retrouver devant l'opérateur à la même date, $\delta_{op}^*$, ce qui effectue la synchronisation.

□

*Exemple:*

Nous avions déjà effectué la sychronisation locale sur notre premier exemple. Pour le Fibonnacci avec $n = 64$ , cela donne :




△

## V.3 Retour à la synchronisation globale

Les résultats précédents nous permettent d'affirmer :

**Théorème 1** *Sous les hypothèses précédentes, la synchronisation locale du lemme 5 conduit effectivement à la méta-période $\pi_{opt}$ du lemme 3 et au délai supplémentaire $\tau_{opt}$ correspondant.*

**Corollaire 3** *Les valeurs $\pi_{opt}$ et $\tau_{opt}$ calculées au lemme 3 sont optimales.*

**Preuve:**

- *Théorème :* Ce n'est que la reformulation du lemme 5.
- *Corollaire :* $\pi_{opt}$ et $\tau_{opt}$ sont atteignables, et le lemme 3 montrait que ces valeurs étaient les meilleures possibles. Elles sont donc optimales.

□

Nous pouvons rappeler que les *hypothèses précédentes* en question sont **(NormData)**, **(Cohérence)**, **(MétaSynchro)** et **(CyclAugNulle)**, et qu'il serait intéressant d'étudier le problème de synchronisation sans les deux dernières.

# VI Point de vue algorithmique

## VI.1 Un algorithme de synchronisation

Les considérations précédentes nous conduisent naturellement à l'algorithme suivant :

**Algorithme 1 (Synchronisation d'un circuit d'arithmétique en-ligne)**

1. Propager $\gamma$
2. Calculer $\tau_{pipeline}$ *(lemme 1)*
3. Énumérer tous les cycles, et calculer $\pi_{cycle}$ *(lemme 2)*.
4. En déduire $\pi_{opt} = \max(\pi_{cycle}, \tau_{pipeline} + n)$ et $\tau_{opt}$
5. Propager les plus longs chemins $\delta^*$ *(lemme 4)*
6. Localement à chaque opérateur, calculer la longueur des registres à mettre devant chaque entrée *(lemme 5)*.

Les étapes 1, 2, 4, et 6 s'effectuent en temps linéaire par rapport à $|\mathcal{C}|$. L'étape 3 est linéaire au nombre de cycles $|\mathcal{Y}|$ du graphe. Enfin, l'algorithme de Floyd-Warshall effectue l'étape 5 en temps $|\mathcal{V}|^3$.

Ce qui nous donne ce résultat :

**Théorème 2** *La synchronisation complète du circuit aboutissant à la méta-période optimale $\pi$ peut être trouvée en temps $O(|\mathcal{V}|^3 + |\mathcal{Y}|)$.*

Il est clair que $|\mathcal{Y}|$ peut être au pire égal à $2^{|\mathcal{E}|}$, donc notre algorithme est en général exponentiel.

**Le problème de décision associée au calcul de $\pi_{cycle}$ est-il NP-complet ?** Malgré de nombreuses recherches personnelles et extérieures, nous n'avons pour l'instant aucun résultat. Nous soupçonnons fortement la NP-complétude – la division $\lceil \delta^*/\omega^* \rceil$ ne donnant rien de bon pour envisager un algorithme polynômial qui fusionnerait des cycles – mais la réduction au circuit hamiltonien ne se laisse malheureusement pas faire. Nous espérons fortement arriver un jour au résultat...

Cependant, dans les cas réels, on a $|\mathcal{Y}| << |\mathcal{E}|$, ce qui mène à un algorithme *polynômial* par rapport à la taille du circuit. Il est aussi raisonnable de considérer le nombre d'entrées et de sorties est petit devant $|\mathcal{V}|$, ce qui mène alors à un algoritme quadratique pour l'étape 5.

## VI.2 Notre implémentation : SyncOL

Nous avons programmé cet algorithme en OCaml[1] de manière modulaire.

Le programme résultant, SyncOL, lit des fichiers décrivant des circuits dans un langage de haut-niveau et effectue leur synchronisation. Il peut aussi directement créer un source VHDL. Nous avons réalisé à cet effet une bibliothèque d'arithmétique en-ligne synchronisée à partir de la bibliothèque d'arithmétique en-ligne développée par Jean-Luc Beuchat au LSL.

SyncOL est un petit programme (700 lignes de code). Bien que documenté, il reste cependant expérimental. Il est disponible librement à l'adresse http://www.ens-lyon.fr/~magiraud/.

1. Ce langage fonctionnel développé à l'INRIA est fortement typé et possède une grande expressivité (http://caml.inria.fr).

## VI.3 Un exemple d'utilisation

Nous reprenons le circuit destiné à la factorielle. Le fichier d'entrée est décrit dans un langage de haut niveau, que l'on charge par 'let fibo = load "tests/fibo"' :

```
{ Suite de Fibonacci }

in (cst_1);                 { Entrée : constante '1' }

s1 = mux (s*,   cst_1);     { Choix de s(n-1) ou de '1' }
s2 = mux (s**2, cst_1);     { Choix de s(n-2) ou de '1' }

t = add  (s1,s2);           { Calcul de la somme }
s = norm (t);               { Normalisation du résultat }

out (s);                    { Sortie du résultat }
```

Après utilisation de la commande 'synchronize 64 fibo', le circuit est synchronisé :

```
fibo: (star = 3, n = 64, pi = 67,
       pi_cycle = 3, tau_pipeline = 3)

      in/out gate 's' : star = 3
          norm_0 ----[0]----> s

      universal normalizer 'norm_0' : star = 3
          add_0 ----[0]----> norm_0

      additionner 'add_0' : star = 2
          mux_1 ----[0]----> add_0
          mux_0 ----[0]----> add_0

      multiplexer 'mux_1' : star = 0
          norm_0 ----[131]-<2>-> mux_1
          cst_1 ----[0]----> mux_1

      multiplexer 'mux_0' : star = 0
          norm_0 ----[64]-<>-> mux_0
          cst_1 ----[0]----> mux_0

      in/out gate 'cst_1' : star = 0
```

Le passage au VHDL n'est ensuite qu'une simple traduction qui se fait ensuite à l'aide de 'vhdl fibo' :

```
-- fibo
-----------------------------------------------------
-- Circuit synchronisé automatiquement par SyncOL V0.1
-- Mathieu.Giraud@ens-lyon.fr, mars 2000

library IEEE;
use IEEE.std_logic_1164.all;

library SOL;
use SOL.sol_lib.all;

-----------------------------------------------------


entity gen_fibo is port (
  Clk, Clr : in  std_logic;
  cst_1: in sol_digit;
  s: out sol_digit);
end gen_fibo;

architecture sync_on_line of gen_fibo is
  signal S_t, S_s2, S_s, S_s1, S_cst_1 : sol_digit;
  signal S_s_delayed_131, S_s_delayed_64 : sol_digit;


begin  -- sync_on_line

  in_reg_cst_1  : fifo_generic_sol
    generic map (N => 1)
    port map (Clk => Clk, Clr => Clr, D => cst_1, Q => S_cst_1);


  norm_0 : norm_sol    --- universal normalizer
    port map (Clk => Clk, Clr => Clr, D => S_t, Q => S_s);

  add_0 : add_sol    --- additionner
    port map (Clk => Clk, Clr => Clr, D0 => S_s1, D1 => S_s2, Q => S_t);

  mux_1 : mux_sol    --- multiplexer
    port map (Clk => Clk, Clr => Clr,
              D0 => S_s_delayed_131, D1 => S_cst_1, Q => S_s2);

  mux_0 : mux_sol    --- multiplexer
    port map (Clk => Clk, Clr => Clr,
              D0 => S_s_delayed_64, D1 => S_cst_1, Q => S_s1);


  s <= S_s;
  delay_s_131 : fifo_generic_sol
    generic map (N => 131)
    port map (Clk => Clk, Clr => Clr, D => S_s, Q => S_s_delayed_131);

  delay_s_64 : fifo_generic_sol
    generic map (N => 64)
    port map (Clk => Clk, Clr => Clr, D => S_s, Q => S_s_delayed_64);

end sync_on_line;
```

## VI.4 Validation expérimentale de SyncOL

– Nous avons testé SyncOL et simulé les résultats sur de nombreux cas.

– Enfin, nous avons implanté le Fibonnacci résultant de notre synchronisation sur un Virtex 1000E monté sur une carte RC1000PP [SB], ce qui nous a permis de découvrir quelques difficultés habituelles à la programmation sur FPGA – la plus notable étant la transmission des données entre le FGPA et l'extérieur. Nous sommes finalement arrivés à lancer le circuit à une vitesse de 38 MHz.

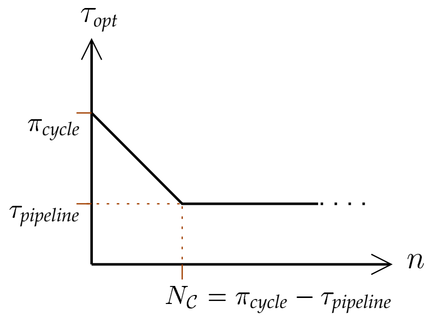
# VII Vers un meilleur pipeline des données ?

Le théorème 1 nous ayant montré que $\pi_{opt}$ et $\tau_{opt}$ sont effectivement atteignables, il devient utile de bien connaître les propriétés de ces valeurs, et, si possible, de chercher ensuite à les réduire.

## VII.1 Variations de $\pi_{opt}$ et de $\tau_{opt}$ en fonction de $n$

Soit un circuit $\mathcal{C}$. Nous avons vu que $\pi_{cycle}$ et $\tau_{pipeline}$ sont des constantes pour $C$ indépendantes de $n$. Par contre, on a $\pi_{opt} = \max(\pi_{cycle}, \tau_{pipeline} + n)$ et $\tau_{opt} = \max(\pi_{cycle} - n, \tau_{pipeline})$. Nous voyons donc que deux comportements peuvent se produire selon que $n$ soit plus petit ou plus grand que $N_{\mathcal{C}} = \pi_{cycle} - \tau_{pipeline}$ :

- si $n \leq N_{\mathcal{C}}$, alors $\tau(n) = \pi_{cycle} - n > \tau_{pipeline}$, et donc $\pi(n) = \pi_{cycle}$. Ce sont alors les cycles du circuit qui limitent sa méta-période.
- si $n \geq N_{\mathcal{C}}$, alors $\tau(n) = \tau_{pipeline}$. Les données sont alors assez "longues" et ont alors le temps nécessaire pour franchir les cycles sans retard : seuls les opérateurs imposent, pour leur pipeline interne, un délai supplémentaire $\tau_{pipeline}$.




On constate ainsi que le délai supplémentaire minimal (correspondant à $\tau_{pipeline}$) est obtenu dès que $n \geq N_{\mathcal{C}}$, condition qu'on essaiera donc de vérifier pour "tirer le meilleur" du circuit.

Ainsi, si l'utilisateur projettait initialement de travailler à précision $n < N_{\mathcal{C}}$, le méta-cycle du circuit sera de toute façon égal à $\pi_{cycle}$, et donc les calculs pourront se faire sans surcoût à précision $N_{\mathcal{C}}$.

*Exemple:*

Pour le circuit de Fibonnacci, nous avions $\pi_{cycle} = 4$ et $\tau_{pipeline} = 3$, d'où $N_{\mathcal{C}} = 4 - 3 = 1$ : le circuit fonctionnera donc toujours de façon correcte.

$\triangle$

## VII.2 Que faut-il améliorer maintenant ?

Il serait utile de pouvoir améliorer les deux valeurs trouvées par notre synchronisation :

- d'un côté, le délai total du circuit aux sorties $\delta^*$
- de l'autre, le couple $(\pi_{cycle}, \tau_{pipeline})$.

Le premier point ne se résout qu'en étudiant un des problèmes fondamentaux de l'informatique qui est l'ordonnancement. On trouvera par exemple dans [Agu94] une étude d'ordonnancement pour les arithmétiques sérielles.

Pour le deuxième point, il est clair que diminuer $\pi_{cycle}$ est aussi un problème d'ordonnancement. Mais nous avons montré ci-dessus que, dès que les données sont suffisament grandes, c'est $\tau_{pipeline}$ qui dicte le méta-cycle du circuit. Ainsi, nous pouvons espérer accélérer nos circuits en améliorant des degrés de pipeline $\tilde{\delta} + \gamma^*$.

Si la composante en $\gamma^*$ rentre aussi dans des problèmes d'ordonnancement, ce n'est pas le cas pour le terme $\tilde{\delta}$ : nous allons l'étudier maintenant dans le but de le réduire.

## VII.3 Bornes pour le délai de pipeline

Dans les études précédentes, il est courant de voir toutes les bascules de chaque opérateur reliées à un unique signal Clr qui commande la réinitialisation du circuit. Un cycle est alors nécessaire après que le dernier chiffre ait quitté l'opérateur pour le réinitialiser : cela mène à un délai de pipeline de $\tilde{\delta}_{op} = \delta_{op} + 1$.

Ce qui nous donne ceci :

**Lemme 6** *Un opérateur en-ligne de délai $\delta_{op}$ et d'augmentation $\gamma_{op}$ vérifie en général $\gamma_{op} \leq \tilde{\delta}_{op} \leq \delta_{op} + 1$.*

Pour une certaine classe d'opérateurs, il est en fait possible d'obtenir $\tilde{\delta}_{op} = \gamma_{op}$. Ces opérateurs ayant été envisagés peu avant la fin de mon stage, le paragaphe suivant ne sera que descriptif.

## VII.4 Idées pour réduire $\tilde{\delta}$

Considérons par exemple l'additionneur simple vu au paragraphe I.2. Nous n'allons plus initialiser les registres entre deux calculs. Pour cela, il faut cependant garantir par un système annexe que les données "non valides" (qui étaient hachurées sur le chronogramme) soient maintenues à zéro. L'additionneur ainsi considéré nécessitera alors un délai de pipeline de $\tilde{\delta} = 1$ au lieu de 3 usuellement.

Plus généralement, nous avons l'intuition du résultat suivant, qu'il faudrait démontrer :

**Intuition 1** *Soit un additionneur généralisé d'augmentation $\gamma_{op}$ ayant un nombre quelconque d'entrées. Alors il est possible d'obtenir sur cet additionneur un délai de pipeline $\tilde{\delta}_{op} = \gamma_{op}$ en n'utilisant pas de réinitialisation entre chaque jeu de données si les entrées sont à 0 lorsque qu'elles ne sont pas valides.*

Enfin, si les multiplieurs et binômieurs peuvent aussi être améliorés de cette manière, il ne semble pas qu'on puisse descendre leur $\tilde{\delta}_{op}$ au niveau de leur $\gamma_{op}$, notamment en raison des automates qu'ils contiennent qui nécessitent absolument une initialisation. Il serait donc très intéressant de décrire précisément la classe des opérateurs où une optimisation peut mener à $\tilde{\delta}_{op} = \gamma_{op}$.

# Conclusion

Ces deux mois de stage au LSL m'auront permis de découvrir les multiples facettes de l'arithmétique en-ligne, et, plus généralement, d'un laboratoire d'architecture.

Ma contribution personnelle peut se résumer à l'énoncé et à une résolution du *problème de la synchronisation* d'un circuit en veillant à formaliser *l'enchaînement des calculs*, tant au niveau d'un opérateur par le délai de pipeline $\tilde{\delta}$ qu'au niveau d'un circuit par les paramètres du méta-cycle $\pi$ et $\tau$. Ces données peuvent prendre de l'importance dans les applications les plus efficaces de l'arithmétique en-ligne comme les chaînes de traitement à la sortie de convertisseurs analogique-numérique.

Cependant le travail effectué est loin d'etre achevé : il faudrait en particulier développer les idées sur les opérateurs à faible délai de pipeline. D'autres questions plus ponctuelles restent, comme la complexité du calcul de $\pi_{cycle}$ évoqué à la partie 3.

**Remerciements**

Je voudrais remercier toute l'équipe du LSL pour son effectif soutien, notamment mon maître de stage Jean-Luc Beuchat. Mes remerciements vont aussi à Arnaud Tisserand, du LIP à l'ENSL, qui m'a judicieusement orienté pour le choix de ce stage.

# Références


[Agu94] Mario Fiallos Aguilar. *Conception et Simulation d'une Machine Massivement parallèle en Grande Précision*. PhD thesis, Laboratoire de l'Informatique du Parallélisme - École Normale Supérieure de Lyon, juin 1994.

[Avi61] Algirdas Avizienis. Signed-Digit Number Representations for Fast Parallel Arithmetic. *IRE Transactions on Electronic Computers*, 10, 1961.

[BDKM94] Jean-Claude Bajard, Jean Duprat, Sylvanus Kla, and Jean-Michel Muller. Some Operators for On-Line Radix-2 Computations. *Journal of Parallel and Distributed Computing*, 22:336–345, 1994.

[Beu] Jean-Luc Beuchat. Thèse en cours. EPFL.

[DFAMY92] Jean Duprat, Mario Fiallos-Aguilar, Jean-Michel Muller, and Hong J. Yeh. Delays of On-line Floating Point Operators in Borrow-Save Representation. In Elsevier, editor, *Algorithms and Parallel VLSI Architectures II*, pages 273–278, 1992.

[Dup] Jean Duprat. *Yet a new redundant number system*. Laboratoire de l'Informatique du Parallélisme - École Normale Supérieure de Lyon.

[GHM89] Alain Guyot, Yavan Herreros, and Jean-Michel Muller. Janus, an on-line multiplier/divider for manipulating large numbers. In IEEE Computer Society Press, editor, *Proc. 9th Symposium on Computer Arithmetic*, pages 106–111, 1989.

[Mul94] Jean-Michel Muller. Some Characterizations of Functions Computable in On-Line Arithmetic. *IEEE Transcations on Computers*, 43:752–755, 1994.

[SB] C. Sweeney and B. Blyth. *RC1000–PP Hardware Reference Manual*. Embedded Solutions Limited. http://www.embedded-solutions.ltd.uk.

[TD97] Arnaud Tisserand and Martin Dimmler. FPGA Implementation of Real-Time Digital Controllers Using On-Line Arithmetic. In *Field-Programmable Logic and Applications. Proceedings of the 7th International Workshop, FPL'97,London*, number 1304 in LNCS, pages 472–481, 1997.

[TE77] Kishor S. Trivedi and Milos D. Ercegovac. On-line Algorithms for Division and Multiplication. *IEEE Transaction on Computers*, C–26(7), July 1977.

[Tis97] Arnaud Tisserand. *Adéquation arithmétique architecture - Problème et étude de cas*. PhD thesis, Laboratoire de l'Informatique du Parallélisme - École Normale Supérieure de Lyon, septembre 1997.

[Yeh93] Hong-Jin Yeh. *Parallel Algorithms and architecures based on digit-seial arithmetic computation*. PhD thesis, Laboratoire de l'Informatique du Parallélisme - École Normale Supérieure de Lyon, July 1993.