De la modélisation par schéma-bloc vers l'implémentation sur carte FPGA.

Adrien Gougeon[†], Bastien Trotobas[†], Patrice Quinton[†], and Martinus Werts[†]

[†]École Normale Supérieure de Rennes

23 août 2017

Résumé

L'exploitation de données en temps réel provenant d'un dispositif externe à un ordinateur peut être rendue difficile par une quantité de données trop importante, la vitesse de transfert étant limitée. Cet article propose des étapes pour implémenter le traitement complet ou partiel de données sur une carte FPGA [1] et ainsi limiter la quantité de données à transférer. Dans cet article nous prendrons l'exemple d'un module compteur de photons pour appliquer notre méthode.

Mots-clés : temps réel ; dispositif externe ; vitesse de transfert ; FPGA ; compteur de photons.

1 Introduction

L'utilisation d'un dispositif matériel externe pour relever des données est courant pour réaliser des mesures physiques. Ces modules ne disposent en général pas de sorties de données complexes tel qu'un port USB ou Ethernet mais plutôt d'une simple sortie TTL (*Transistor-Transistor Logic*) $+5V$, ce qui implique que les modules ne peuvent être connectés directement à un ordinateur. Un moyen simple de traiter un signal TTL est l'utilisation d'une carte FPGA disposant de sorties plus complexes pouvant être envoyées vers un ordinateur et notamment d'un port série UART (*Universal Asynchronous Receiver Transmitter*).

Cependant ce port série a une vitesse de transfert faible et si l'on souhaite traiter les données en temps réel il peut être nécessaire de réduire la fréquence d'envoi et donc la précision finale. Une solution pour pallier à ce problème est de transposer tout ou partie de l'exploitation des données de l'ordinateur à la carte FPGA afin de réduire la quantité de données à transférer. Pour modéliser le traitement des données nous avons décidé d'utiliser un logiciel basé sur des schémas-bloc. Ce type de design est particulièrement adapté pour ce qui concerne la modélisation de circuits synchrones [5] et cette représentation possède aussi l'avantage d'avoir une forte correspondance avec un schéma logique.

Notre idée dans cet article est de présenter une façon de passer des schémas-bloc à du code VHDL (*VHSIC Hardware Description Language*), langage utilisé pour programmer des cartes FPGA. Pour ce faire l'expérimentateur doit au préalable réaliser une modélisation par schéma-bloc de son exploitation des données, cette modélisation sera ensuite abstraite par un graphe permettant d'obtenir les informations nécessaires à la création d'un code dans le

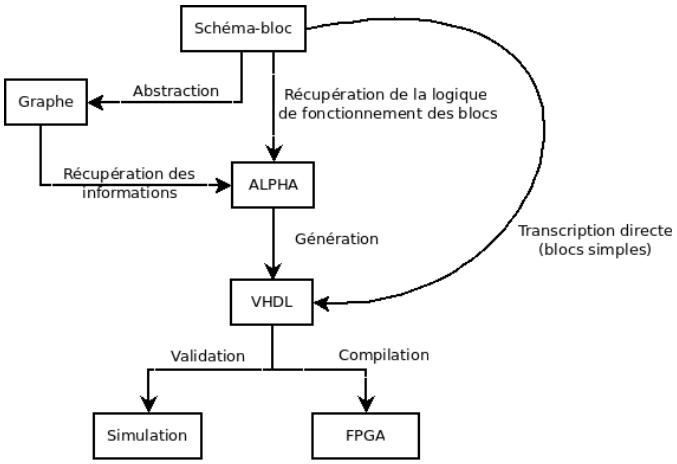

FIGURE 1 – Schéma de la méthode d'exportation d'un schéma-bloc vers une carte FPGA. Lorsque cela est possible, les blocs du schéma sont traduits directement en VHDL, sinon, on passe par leur représentation avec le langage Alpha.

langage Alpha [4], dont une des propriétés est de s'exporter en VHDL. Un schéma général de notre méthode d'exportation est présenté en figure 1.

Afin d'illustrer notre méthode nous avons pris l'exemple d'un module compteur de photons. Le comptage de photons présente de nombreuses applications telles que la détection de rayonnement à très basses températures, la mesure de décroissance phosphorescente, ou encore pour la chimiluminescence [3]. Il existe principalement deux technologies pour le comptage de photons : les photomultiplicateurs, adaptés aux mesures dans l'ultraviolet, et les modules compteur de photons uniques, basés sur des photodiodes à avalanche et adaptés aux mesures dans le visible et dans l'infrarouge. La photodiode à avalanche permet de transformer un photon reçu en une impulsion électrique (signal TTL).

Nous avons choisi cet exemple car nous disposions déjà d'un module compteur de photon relié à une carte FPGA. Actuellement, les impulsions générées sont mémorisées sur la carte à l'aide d'un compteur dont la valeur est envoyée à un ordinateur via le port UART, ce qui limite la fréquence maximale d'envoi de données.

La première partie présente les étapes de l'exportation du traitement des données de l'ordinateur vers la carte FPGA, la seconde partie décrit comment nous l'avons réalisé, enfin la dernière partie conclut.

2 Méthode d'exportation

L'exportation du traitement des données vers la carte FPGA a demandé de faire des choix d'outils de transition. Le premier choix était celui de l'outil de modélisation, Xcos [2]. La carte FPGA nécessitant du code VHDL pour fonctionner, il nous fallait donc le générer mais c'est un langage trop bas niveau pour être obtenu immédiatement, nous avons donc pris un langage intermédiaire : Alpha. Pour créer un bloc dans le langage Alpha il est nécessaire d'avoir des spécifications sur les fréquences de fonctionnement des blocs synchrones modélisés sur Xcos, nous avons pour cela réalisé une abstraction du schéma-bloc en Python sous la forme d'un graphe.

1. *Field-Programmable Gate Array*, réseau de portes logiques programmables in situ.

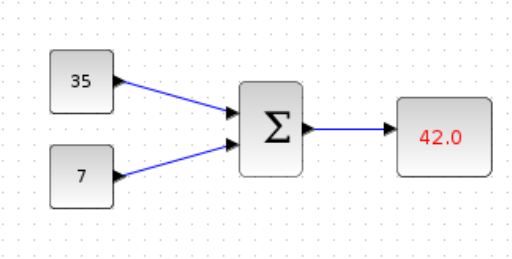

FIGURE 2 – Schéma-bloc d'une somme en Xcos. Le bloc central donne en continu la somme des deux blocs en entrée.

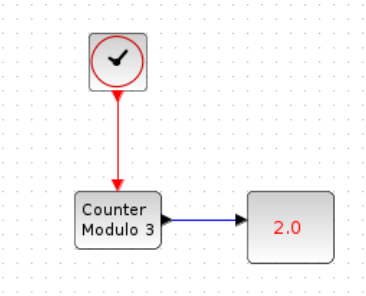

FIGURE 3 – Schéma-bloc d'un compteur modulo en Xcos. Le bloc *Counter* s'incrémente à chaque signal reçu par le bloc horloge.

2.1 Modélisation

Nous avons choisi d'utiliser Xcos pour réaliser les schémas-bloc, un outil de modélisation et de simulation dynamique à base de blocs, fourni avec Scilab, qui représente une alternative *open-source* au logiciel Simulink [1], fourni avec Matlab, couramment utilisé pour réaliser des schémas-bloc.

La modélisation sur Xcos repose sur des blocs combinatoires et des blocs synchrones, ces derniers sont régis par des événements, à l'inverse des premiers. Un bloc combinatoire possède un comportement semblable à une fonction logique combinatoire, ou porte logique sur un circuit électronique. Son fonctionnement est présenté en figure 2.

À l'inverse un bloc synchrone ne s'active que lorsqu'un signal lui est fourni par une horloge. La figure 3 montre un compteur modulo synchrone, à chaque signal reçu le compteur s'incrémente.

Deux autres types de bloc sont au cœur d'une modélisation par schéma-bloc en Xcos, ce sont le super-bloc et le diviseur de fréquence. Les super-blocs permettent une conception structurée d'une application, en effet un super-bloc contient lui-même un design et permet de traiter celui-ci comme un simple bloc. Le second est lui même un super-bloc, les diviseurs de fréquence ont un rôle particulier, puisqu'ils influent sur la synchronisation des blocs synchrones. Chaque impulsion du signal d'horloge entrant est comptée et lorsque le compteur modulo revient à zéro une impulsion est libérée, ce qui permet de diviser la fréquence d'entrée par la valeur désirée. Le design interne d'un diviseur de fréquence par 3 est présenté en figure 4.

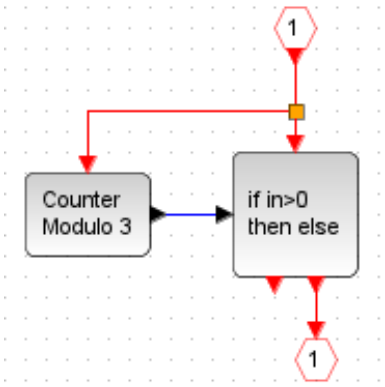

FIGURE 4 – Schéma-bloc interne au super-bloc d'un diviseur de fréquence par 3. Toute les 3 impulsions reçues une impulsion est émise.

2.2 Abstraction

Une fois la modélisation créée, il est nécessaire de faire une abstraction du design par un graphe où les nœuds représentent les blocs et les arcs les connections entre les blocs. L'abstraction a été réalisée à l'aide du langage Python et du module *networkx*. Cette abstraction a pour but de définir la période de fonctionnement et le temps avant initialisation (nombre de fronts d'horloge ignorés) pour chaque bloc synchrone du schéma-bloc, notés respectivement λ et σ. Nous avons implémenté plusieurs fonctions Python dans ce but.

La première étape est de vérifier qu'il n'y a qu'une seule et unique horloge principale d'où découlent toutes les autres, notamment à l'aide de diviseurs de fréquence. En effet, avec plusieurs horloges principales l'ordre d'exécution des blocs dépendant de ces horloges n'est pas garanti [7]. Ensuite la fonction set_lambda, voir algorithme 1, va chercher dans le graphe tout bloc synchrone et lui assigne un λ en fonction du bloc duquel il descend : soit de l'horloge principale soit d'un diviseur de fréquence. Ce λ permet de définir les différentes fréquences de fonctionnement des blocs synchrones. Le premier appel de cette fonction se fait avec comme paramètres le graphe et l'horloge principale. Il s'agit d'un parcours d'arbre avec pour racine l'horloge principale, pour feuilles les blocs synchrones et pour nœuds intermédiaires les diviseurs de fréquences.

Algorithme 1 : set_lambda

Input : G : le graphe, n : le noeud donnant l'horloge à assigner aux blocs suivants

```
clk ← G[n]["lambda"]
div ← G[n]["divider"]
for chaque successeurs s de G[n] do
    G[s]["lambda"] ← clk/div
    if s est un diviseur de fréquence then
        set_lambda(G, s)
    end
end
```

Suite à cela, la fonction set_shift va chercher tous les blocs sans prédécesseur et à partir de ceux-ci descendre dans le graphe en définissant leur σ à $0 +$ nombre de registres traversés. En pratique, les paramètres λ et σ permettent la génération en VHDL de signaux de contrôle *clock enable* qui doivent être appliqués aux registres de l'architecture. Il aurait été logique de vérifier l'absence de cycles combinatoires dans le schéma-bloc mais cette vérification est déjà effectuée par Xcos, en effet, un circuit avec un cycle combinatoire n'est pas accepté par le

compilateur et est donc impossible par construction.

Une fois les périodes et les décalages définis il est désormais possible de décrire les blocs Xcos en Alpha et donc de les traduire en VHDL.

Nous avons aussi remarqué qu'il était possible d'exporter en XML un fichier Xcos dans la version la plus récente de Scilab (6.0), il serait donc intéressant d'extraire des informations du fichier XML afin d'avoir une création automatique de l'abstraction du schéma-bloc dans de futurs travaux.

2.3 Alpha

Suite à l'obtention des périodes et des décalages associés aux blocs que nous voulons générer il nous reste désormais à décrire leur fonctionnement en Alpha. Le langage Alpha a été conçu par Christophe Mauras en 1989 [4] pour faciliter la description de calculs réguliers tels qu'on les trouve dans des boucles, et permettre leur transformation sous forme de calculs parallèles. Le langage Alpha repose sur le modèle polyédrique dans lequel les calculs sont représentés comme des collections de données associées aux points d'un polyèdre entier. Un logiciel appelé MMAlpha a été développé par la suite afin d'automatiser les transformations de programmes Alpha et leur traduction sous forme d'architecture. Dans sa version actuelle, il est possible de traduire des calculs sous forme d'un programme VHDL synthétisable.

Alpha peut aussi être utilisé pour décrire des systèmes flot de données parallèles [6], en interprétant la première dimension des polyèdres comme étant l'indice d'itération des flots de données.

2.4 VHDL

Une fois les blocs décrits en VHDL, il faut désormais assembler ces blocs afin de reconstruire le design créé en Xcos. Cette étape est complexe car il est nécessaire de tester le bon fonctionnement de l'assemblage afin de vérifier que le résultat correspond au design tel qu'il a été fait en Xcos. Nous avons effectué ces vérifications à l'aide de l'outil de simulation ModelSim.

3 Exemple

Pour illustrer notre méthode d'exportation de traitement de données d'un ordinateur à une carte FPGA, nous avons choisi de traiter le cas de la décroissance phosphorescente d'une diode, mesurée à l'aide d'un module compteur de photons.

Nous avons tout d'abord créé des designs permettant de simuler des données provenant du module afin de vérifier le bon fonctionnement du traitement des données. Nous avons ensuite adapté notre design pour pouvoir lire un fichier contenant des données réelles. Enfin, nous l'avons de nouveau adapté pour pouvoir intégrer les entrées et sorties telles qu'elles sont réalisées sur la carte FPGA.

3.1 Simulation

La mesure de la décroissance phosphorescente consiste à détecter et compter les photons émis par une diode avec cumul des différences entre deux mesures successives afin d'obtenir

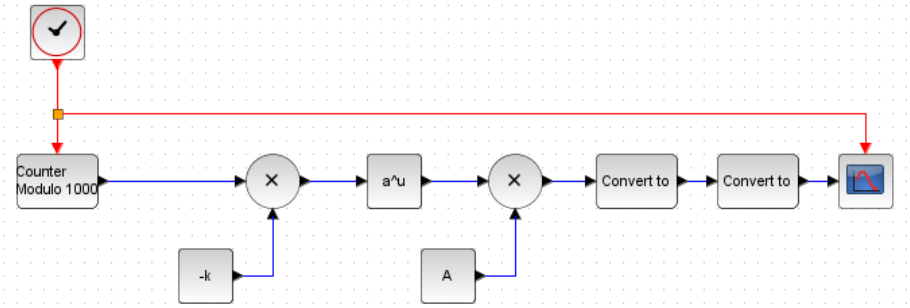

FIGURE 5 – Design de la décroissance phosphorescente.

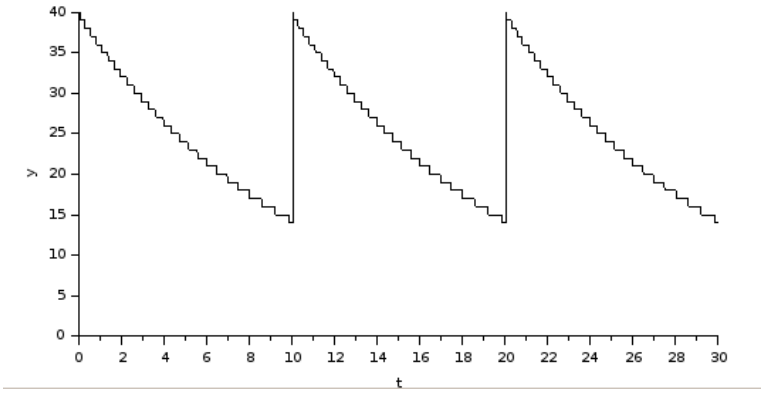

FIGURE 6 – Exemple de la simulation de la décroissance phosphorescente. La décroissance est réinitialisée toutes les dix secondes.

une courbe de décroissance atténuant le bruit provenant du matériel ou de l'expérience en elle-même. *In situ* l'expérience consiste à allumer et éteindre une diode puis de mesurer à intervalles réguliers les photons réémis, il suffit ensuite de répéter ce processus en sommant une à une les mesures faites après le même temps d'extinction de la diode.

Actuellement, la carte FPGA compte les impulsions émises par le module compteur de photons et renvoie la valeur du compteur toutes les 4 μs par le port UART de la carte afin que les données soient traitées par la suite sur un ordinateur.

Avant de porter notre traitement des données sur FPGA nous souhaitions simuler entièrement l'expérience en Xcos. Pour ce faire nous avons donc simulé aussi la partie comptage de photons, partie absente sur le design que nous exporterons.

La première partie consiste à modéliser en Xcos la décroissance phosphorescente sans bruit, comme présenté en figure 5.

La décroissance suit ici une loi de type :

$$A \exp(-kt)$$

Le résultat de cette simulation est présenté en figure 6. Nous avons choisi $A = 40$, $k = 0.001$ et une remise à zéro de la décroissance toutes les dix secondes, ce qui correspond au compteur modulo 1000 car la période est réglée à 0.01s. Cette réinitialisation de la décroissance est nécessaire afin de simuler le fait que la diode est remise sous tension de façon périodique.

La seconde partie a pour but de simuler le registre 8 bits présent sur la carte FPGA,

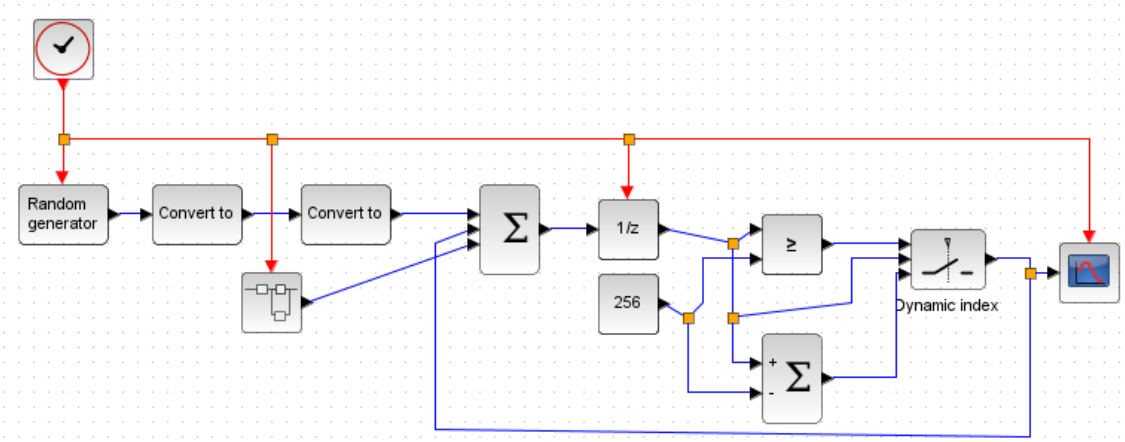

FIGURE 7 – Design du registre de comptage sur la carte FPGA.

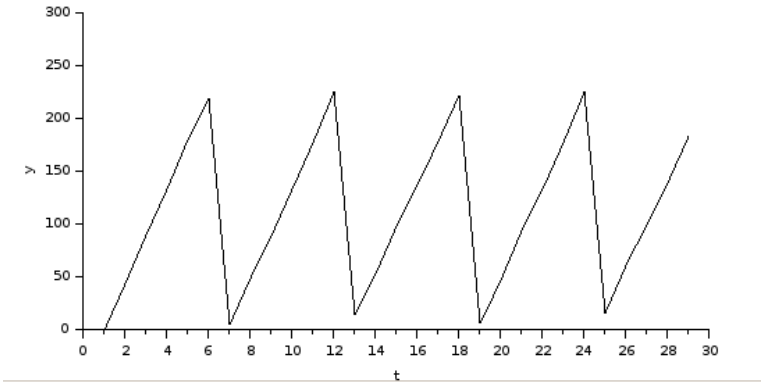

FIGURE 8 – Exemple de la simulation du registre de comptage sur la carte FPGA. Le registre étant représenté par un compteur 8 bits, ses valeurs vont de 0 à 255.

donnant la somme cumulée des photons détectés. Le design Xcos de ce registre est présenté en figure 7.

Dans ce design la partie gauche, avant le bloc $\sum$, sert à générer des valeurs bruitées de décroissance. Le bloc *Random generator* donnant des valeurs réelles, le bloc *Convert to* sert à les transposer en entiers, le vrai compteur donnant évidemment des valeurs entières. Le second bloc *Convert to* sert à repasser en réels, type nécessaire pour les blocs suivants, ces derniers servant à simuler le fait que le compteur fonctionne avec un registre 8 bits. Le bloc en dessous des deux *Convert to* est un super-bloc, il contient le diagramme présenté en figure 5 où l'horloge et la sortie graphique ont été remplacées par de simples liens, permettant ainsi de l'intégrer dans un design plus large. Ces super-blocs permettent d'avoir des designs à plusieurs niveaux de profondeur. Le résultat de cette simulation est présenté en figure 8. Nous avons bien ici un compteur modulo 256 tel qu'il est présent actuellement sur la carte FPGA.

La troisième partie consiste à calculer la différence entre deux relevés consécutifs du registre afin de mesurer la décroissance. Il s'agit simplement de soustraire à la valeur actuelle la valeur précédente, ralentie d'un front d'horloge grâce à un registre.

La dernière partie consiste à faire la somme cumulée des différences entre deux mesures consécutives sur une fréquence, afin de réduire l'impact du bruit.

La figure 9 montre le design complet de l'expérience. Le super-bloc contient le diagramme présenté en figure 7 auquel nous avons rajouté la différence entre deux relevés

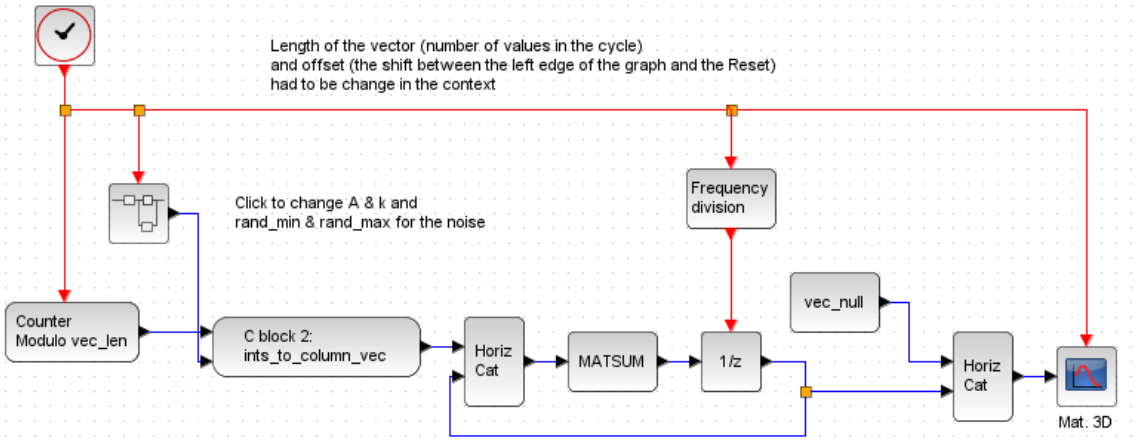

FIGURE 9 – Design complet de la simulation de mesures cumulées de décroissance phosphorescente.

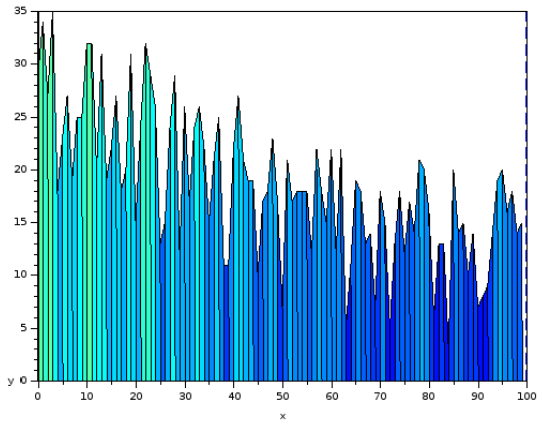

FIGURE 10 – Simulation d'un cycle. La décroissance est à peine visible ici.

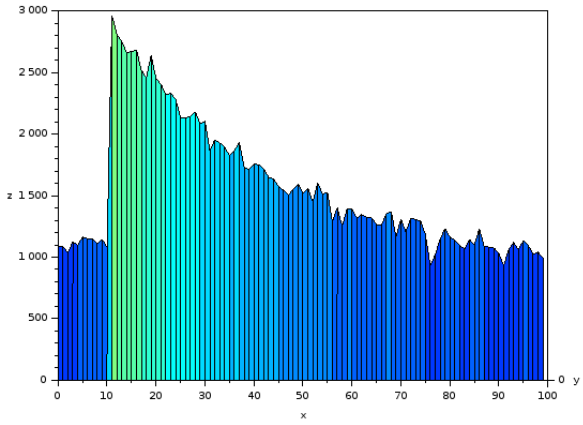

FIGURE 11 – Simulation de 100 cycles. La décroissance est discernable mais reste relativement bruitée.

consécutifs. Ces valeurs sont envoyées vers un bloc contenant du code C et créant un vecteur avec ces valeurs, ce bloc exécute le code C avec lequel il a été configuré de façon séquentielle, il est cependant considéré ici comme un bloc combinatoire et par conséquent au résultat instantané. Les blocs suivants servent à accumuler ces vecteurs en un seul. Le diviseur de fréquence sert à ne déclencher la somme de vecteurs que lorsque le bloc C a traité suffisamment de mesures.

Les figures 10, 11, 12 et 13 présentent le résultat d'une simulation de l'expérience. Les valeurs utilisées sont :

$A = 20$ $k = 0.03$ rand_min $= 0$ rand_max $= 5$ offset $= 10$ vec_len $= 100$

Les paramètres *rand_min* et *rand_max* servent à définir la quantité de bruit généré, *offset* définit le temps avant la réinitialisation de la décroissance et *ven_len* définit la longueur d'un cycle, i.e. la taille du tableau de cumul. L'axe des abscisses représente la position dans le vecteur de cumul et l'axe des ordonnées la valeur associée.

Nous avons ici une bonne observation du principe de cumul afin d'atténuer l'effet du bruit sur la mesure. La figure 13 présente la décroissance sans bruit. Sur la figure 10 la forme de la décroissance est à peine distinguable après un seul cycle, en raison du bruit, alors

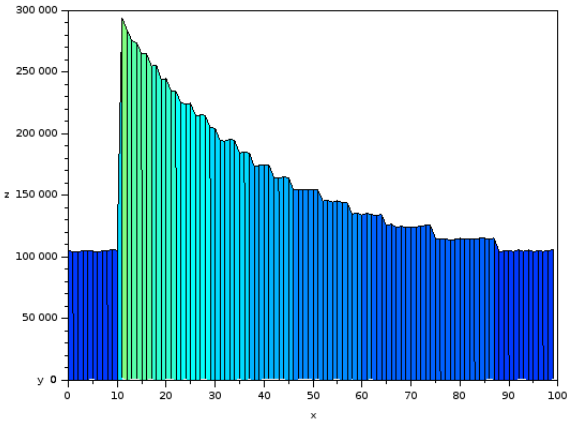

FIGURE 12 – Simulation de 10000 cycles. La décroissance devient clairement visible et s'approche de celle sans bruit.

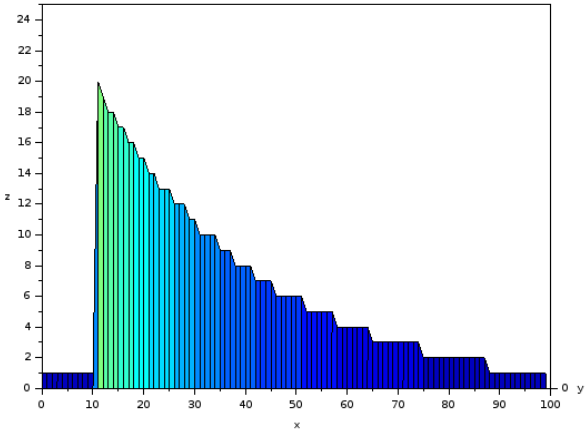

FIGURE 13 – Décroissance sans bruit.

qu'elle devient de plus en plus nette au fil des cycles jusqu'à devenir semblable à la décroissance sans bruit sur la figure 12, après 10000 cycles.

3.2 Exploitation d'un fichier de données

Maintenant que la simulation complète donne des résultats tels qu'attendus, nous avons modifié le design afin de traiter des données réelles provenant de la carte FPGA.

Il effectue le même traitement que le design présenté en figure 9. La seul différence notable est que les données fournies par le super-bloc ne sont pas des données générées mais lues dans un fichier.

3.2.1 Validation

Afin de valider notre design nous avons comparé notre ancien traitement des données, réalisé en Python, avec le nouveau traitement en Xcos. Nous avons d'abord fait une première comparaison avec des valeurs faiblement bruitées puis nous avons ensuite fait une seconde comparaison avec des valeurs fortement bruitées. Les figures 14 et 15 présentent l'exploitation des données des valeurs faiblement bruitées et les figures 16 et 17 présentent l'exploitation des valeurs fortement bruitées. L'axe des abscisses représente la position dans le vecteur de cumul et l'axe des ordonnées la valeur associée.

Nous avons donc pu constater que l'exploitation des données en Xcos nous renvoie un résultat similaire à l'exploitation en Python. Il n'y a pas de décroissance phosphorescente car pour créer le fichier de données nous nous sommes servis d'une diode simple non phosphorescente, par conséquent elle n'émet plus de photons à partir du moment où elle n'est plus alimentée.

3.3 Modification du design pour l'exportation

Maintenant que nous disposons d'un design exploitant les données tel que désiré, nous l'avons de nouveau modifié afin d'y intégrer les parties réellement présentes actuellement sur la carte FPGA, qui sont le compteur d'impulsions et l'envoi de données via le port série UART grâce au composant UAT.

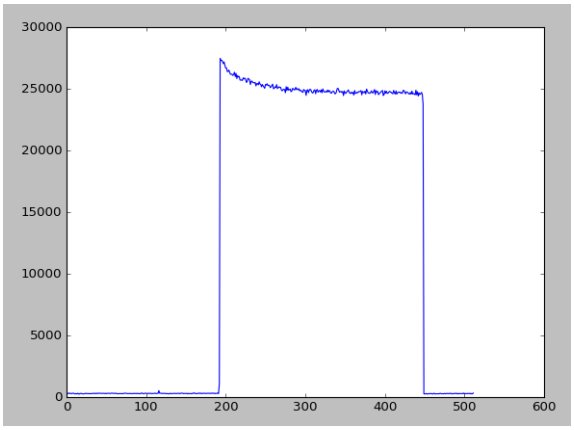

FIGURE 14 – Traitement des données en Python, bruit faible.

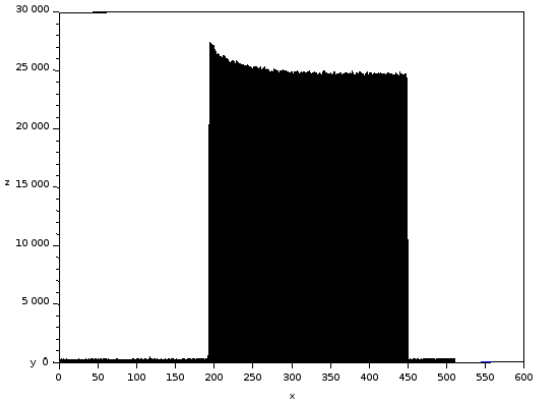

FIGURE 15 – Traitement des données en Xcos, bruit faible.

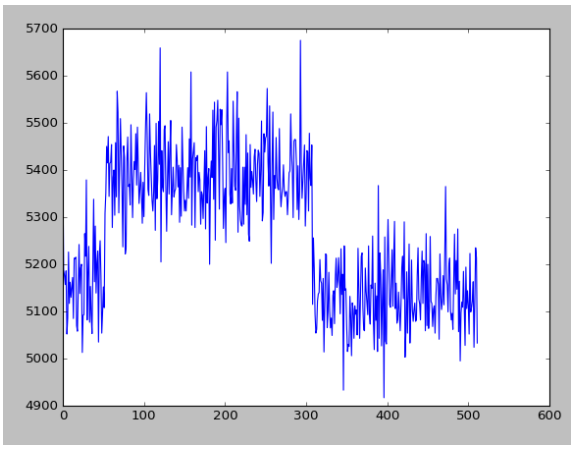

FIGURE 16 – Traitement des données en Python, bruit fort.

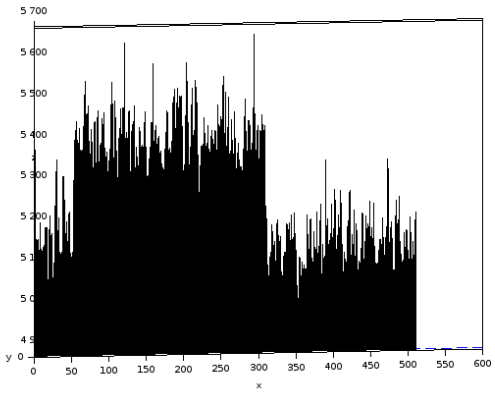

FIGURE 17 – Traitement des données en Xcos, bruit fort.

Le compteur d'impulsions remplace la lecture dans un fichier, ce composant compte en continu, de façon asynchrone, le nombre d'impulsions qu'il reçoit de la part du module compteur de photons et renvoie la valeur du compteur à chaque front d'horloge reçu.

Le composant UAT a pour rôle de transmettre un octet dans le format UART. Ce mode d'envoi était idéal lorsqu'il s'agissait uniquement de renvoyer directement la valeur du compteur 8 bits au PC. Cependant, étant donné que nous avons prévu d'effectuer des calculs sur les valeurs relevées, il nous semblait judicieux de pouvoir envoyer soit des entiers plus grands, soit des flottants. Nous avons pour cela créé un composant *separateModule* qui vient se greffer en amont du composant UAT afin de découper un mot de 32 bits en 4 octets pouvant être envoyés à la suite au PC via l'UART.

3.4 Abstraction

La création du graphe en Python, afin d'obtenir les périodes ainsi que les décalages nécessaires pour créer les blocs en Alpha, se passe comme suit :

1. créer les nœuds et les liens qui représentent les différents blocs de notre graphe Xcos ;
2. définir la fréquence de base de l'horloge principale ;
3. définir les valeurs des diviseurs de fréquence ;
4. définir quels blocs sont des registres ;
5. lancer les fonctions *set_lambda* et *set_shift* qui vont assigner à chaque nœud une valeur correcte pour les attributs *lambda* et *shift* ;
6. récupérer les listes donnant les attributs pour chaque nœud.

3.5 Alpha

Suite à l'obtention des λ et σ, Alpha et MMAlpha ont été utilisés pour modéliser des composants du système, et ensuite les traduire en un programme VHDL qui a été intégré au reste du code VHDL produit à la main. Cette méthode a permis de tester le code VHDL produit par MMAlpha dans un tel contexte et constitue ainsi une perspective de développement futur du projet.

3.6 VHDL

Après la génération en VHDL de certains composants, il nous reste à assembler ceux-ci avec les anciennes parties dont nous disposions (i.e. le compteur d'impulsions et le module d'envoi des données via le port UART). L'assemblage se fait principalement par une description des liaisosn entre les différents modules.

Nous avons tout d'abord lié les modules *incrementedVector* et le compteur d'impulsions afin d'avoir un tableau dont les données proviennent bien du module compteur de photons. Nous avons ensuite lié ceci avec le module *separateModule* et l'envoi des données afin de pouvoir transmettre nos résultats.

Lors de simulations, nous avons réussi à multiplier par 100 la fréquence de relevés du compteur d'impulsions par rapport à l'ancien design. Pour ce faire, nous avons utilisé un tableau de stockage des relevés du compteur de taille 100. Par conséquent, sur une période donnée où nous transmettions précédemment uniquement une valeur relevée depuis le compteur d'impulsions, nous transmettons désormais le résultat d'un calcul effectué sur 100 relevés du compteur.

4 Conclusion

Notre objectif était de réussir à implémenter tout ou partie d'un schéma-bloc sur une carte FPGA. L'idée principale était que nous étions limités par la vitesse d'envoi des données et nous souhaitions donc être capable de traiter les données en amont, sur la carte FPGA, afin d'avoir une quantité plus faible de données à transmettre.

En prenant pour point de départ un schéma-bloc en Xcos, nous avons démontré la faisabilité de l'exportation de ce design vers une carte FPGA en passant par des étapes d'abstraction du design, de transcription en Alpha, de transcripton en VHDL et enfin de liaison des nouveaux modules VHDL entre eux afin d'obtenir un programme VHDL fonctionnel pouvant être implanté sur une carte FPGA.

Il resterait à démontrer que notre implémentation sur la carte FPGA correspond bien au design décrit en Xcos. Il serait aussi intéressant d'intégrer MMAlpha dans la boucle de conception : on peut envisager la traduction de boucles directement en Alpha, ou encore, d'associer simplement à certains éléments de la bibliothèque leur expression en Alpha.

Références

[1] Site officiel de matlab. https://fr.mathworks.com/products/simulink.html. Accédé en juillet 2017.

[2] Site officiel de scilab. https://www.scilab.org/scilab/features/xcos. Accédé en juillet 2017.

[3] W. Becker. *The Bh TCSPC Handbook*. Becker & Hickl, 2014.

[4] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The alpha language and its use for the design of systolic arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 3(3) :173–182, 1991.

[5] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9) :1235–1245, Sept 1987.

[6] Patrice Quinton, Anne-Marie Chana, and Steven Derrien. Efficient hardware implementation of data-flow parallel embedded systems. pages 364–371, 07 2012.

[7] Jean-Philippe Chancelier Stephen L. Campbell and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos*. Springer, 2006.

A Auto-évaluation FFOM

- Forces :
  - une solution à la vitesse de transfert limitée d'une liaison UART ;
  - une méthode pour transposer des schémas-bloc en code VHDL.
- Faiblesses :
  - il n'y a pour l'instant aucune preuve que le code VHDL généré et écrit produise exactement le même résultat que ce que réalise le schéma-bloc Xcos.
- Opportunités :
  - la découverte d'Xcos et des schémas-bloc ;
  - Le mélange de connaissances variées allant de la programmation en VHDL à l'implantation sur carte FPGA en passant par l'utilisation de graphes en Python.
- Menaces :
  - manque de connaissances en VHDL et en Xcos ayant probablement pour conséquence un manque de bonnes pratiques.