Cédric DUMONDELLE E3
Lucas FILLON
Xavier MARINO
Pierre OLIVIER

Projet avancé SE

Yannick BORNAT & Bertrand LE GAL

Table des matières

0 Introduction

0.1 Contexte

0.2 Problématique

1 Présentation du cas d'étude

1.1 Introduction

Afin de mener à bien cette étude nous nou sommes basés sur les travaux de Yannick Bornat. En effet, dans le cadre de l'analyse de signaux biomédicaux, ce dernier a mis au point un module de détection d'activité de cellules biologique, en VHDL.

Dans le cas de signaux provenant de cellules biologiques, l'activité cellulaire peut être détectée par un pic d'amplitude, celui-ci est représenté en figure XX. Cependant ce type de signaux possède une composante en bruit basse fréquence très élevée, c'est pourquoi il doit être filtré en amont. C'est ce signal filtré qui est utilisé à la détermination d'une valeur de seuil nécessaire à la détection d'activité biologique. Ce principe de fonctionnement, qui sera à la base du développement de notre module, est illustré en figure XX.

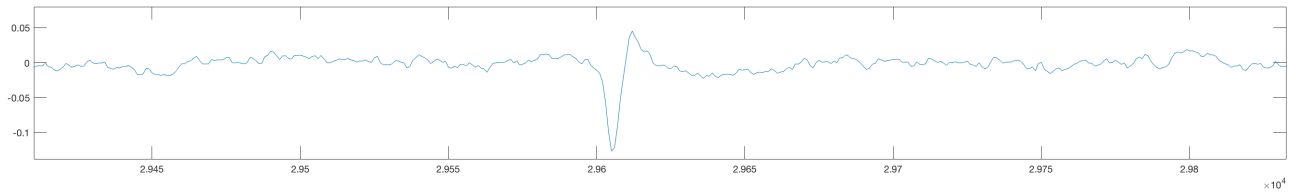

Figure 1 – Pic d'activité dans un signal biologique

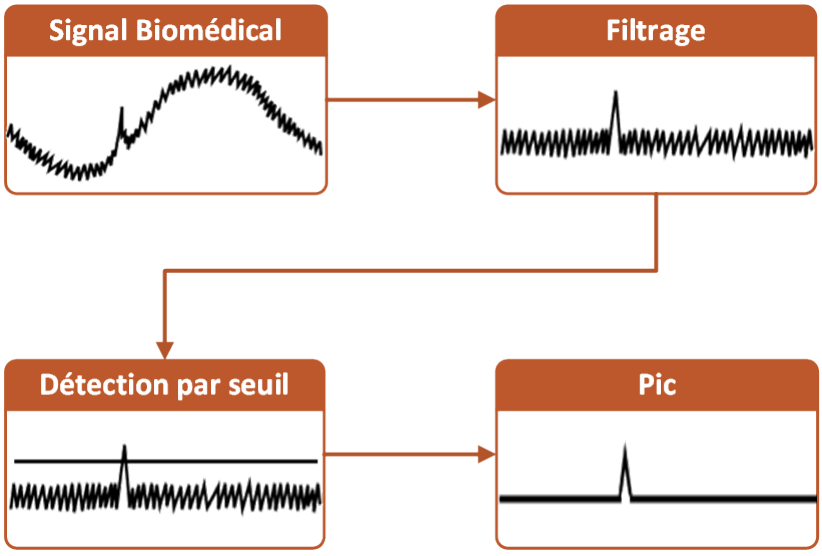

Figure 2 – Principe de fonctionnement du module de détection d'activité cellulaire

1.2 Fonctionnement des différents blocs

Comme énoncé précédement, les signaux issus de capteurs biologiques sont très fortement bruité en basse fréquence, ceci se caractérise par une forte variation de l'amplitude du signal. Cette variation d'amplitude, illustrée en figure XX a) sur 5000 échantillons, est du même ordre, voir plus grand, que les que les pics attestants d'une activité celllulaire. Il convient donc de supprimer ce bruit afin de correctement détecter les pics d'activité.

- **Suppression du bruit basse fréquence contenu dans le signal d'entré** : c'est un filtre à réponse impulsionnelle infinie (IIR) qui est utilisé pour s'affranchir des basses fréquence, suivant l'équation (1).

$$y_n = \frac{63}{64}\left(x_n - x_{n-1}\right) + \frac{31}{32} y_{n-1} \tag{1}$$

  Le résultat issus de cette première étape de filtrage est illustré en figure XX b), on retrouve les pics caractéristiques du signal, mais celui-ci est désormais centré sur 0. On peut, cependant, constater que ce signal est également bruité en haute fréquence. C'est pour limiter l'impact de ce bruit, qu'une valeur de seuil adaptative est utilisée afin détecter les pics.

- **Calcul de la valeur de seuil** : deux modèles différents ont été explorés lors de cette étape, l'un d'entre eux reprennant les travaux de Yannick Bornat en appliquant une boucle de correction sur le signal d'entré filtré, tandis que l'autre méthode utilise la valeur d'écart type, de nouveau appliqué au signal filtré, (à un facteur de proportionnalité près), comme valeur de seuil.

a) Calcul du seuil par boucle de correction :

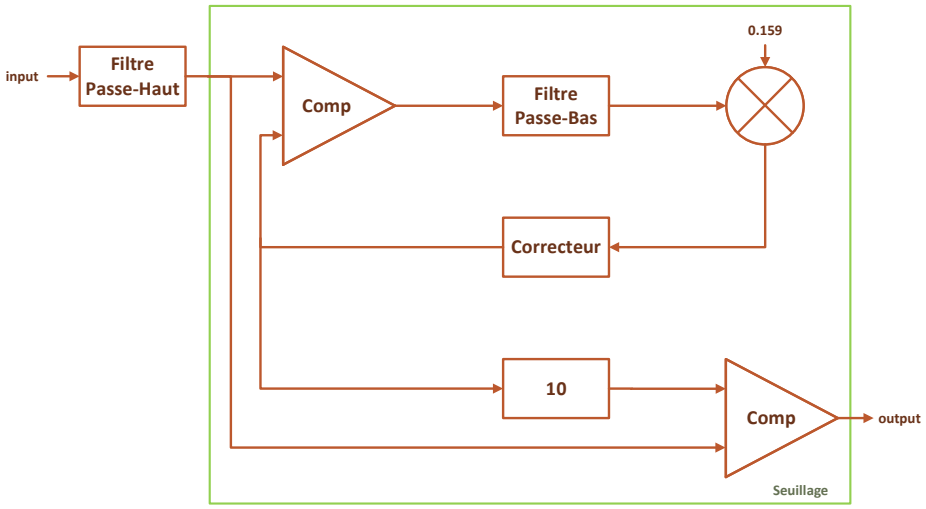

Figure 3 – Architecture du seuillage avec correcteur

Dans ce modèle, le but est de fixer le seuil de façon à ce qu'uniquement 15.9% des échantillons soient supérieurs à celui-ci. Dans ce but, une boucle de contre-réaction est mise en place afin d'adapter la valeur de seuil afin de remplir cette condition. Dans un premier temps, l'échantillon est comparé avec la valeur de seuil actuelle. Le comparateur produira un niveau haut si celui-ci est supérieur au seuil et produira un niveau bas sinon. Le filtre passe-bas qui suit sert de moyenneur. En effet, celui-ci permet de calculer la valeur moyenne des sorties du comparateur sur un certain nombre d'echantillon. En d'autres termes, il donne en sortie le pourcentage d'echantillon au dessus de la valeur de seuil. Après soustraction de 0.159, la valeur de sortie de ce moyenneur est envoyé en entrée du correcteur composé d'un proportionnel integrateur ce qui permet d'annuler l'erreur statique.

b) Calcul du seuil par écart type :

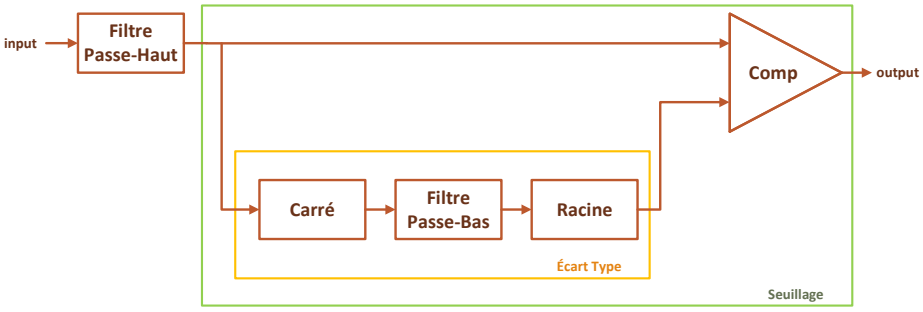

Figure 4 – Flot de compilation

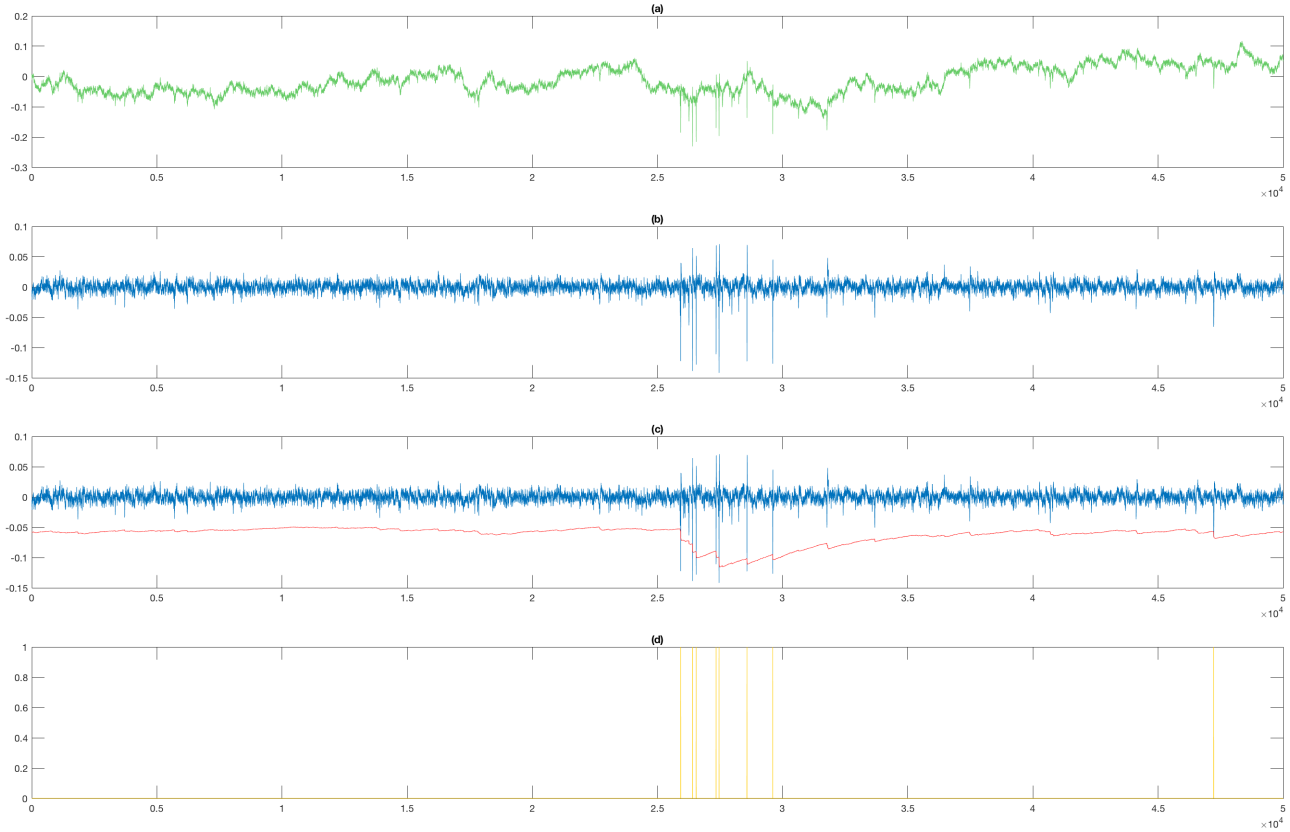

Figure 5 – toto

2 Implémentation depuis le SystemC

Les schémas présentés en Figures X et X permettent de se représenter le fonctionnement de la chaîne globale. La modularité du SystemC nous permet alors de concevoir une paire de fichiers, un fichier source de description ainsi qu'un *header*, pour chacun des blocs composant ces schémas. Il convient de détailler le cheminement suivi à partir de la création de ces fichiers jusqu'à leur implémentation sur la carte Nexys 4. Cette partie vient donc présenter le flot de conception/compilation lié au SystemC et sa mise en oeuvre lors du projet.

2.1 Flot de compilation

L'apprentissage d'un nouveau langage de description ou de programmation passe nécessairement par la compréhension du flot de compilation qui lui est associé. Pour le SystemC, on peut découper ce flot en quatre étapes principales illustrées par la Figure X :

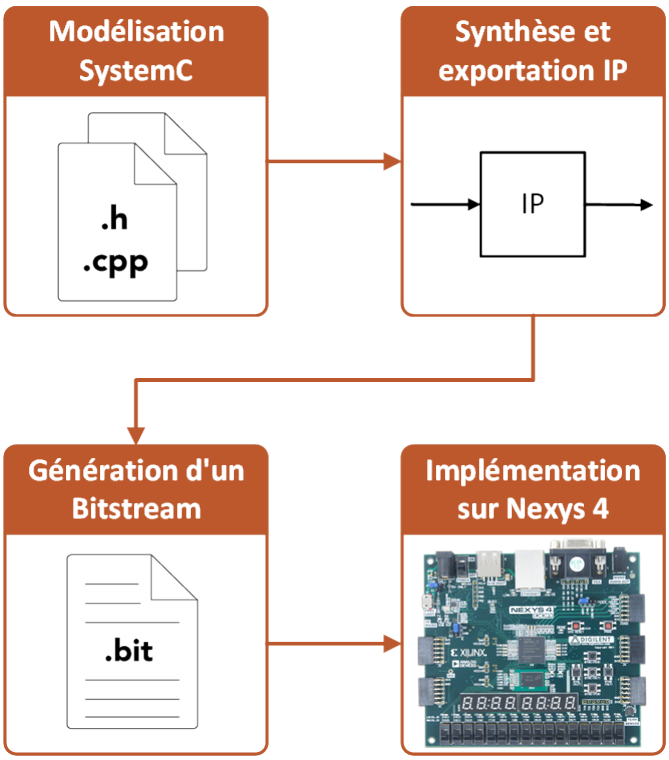

Figure 6 – Flot de compilation

- **Modélisation SystemC** : dans un premier temps il convient de décrire en SystemC le système que l'on souhaite modéliser. Pour cela, on rédige des fichiers sources dans un IDE SystemC pour symboliser chacun des blocs décrits précédemment.

- **Synthèse et exportation IP** : une fois les fichiers SystemC écrits, on peut désormais les passer dans l'environnement *Vivado HLS* afin de les synthéthiser et d'exporter un ou plusieurs blocs sous forme d'IP (*Intellectual Property*). Ces blocs IP sont des blocs logiques, avec des entrées et des sorties, qui vont être implémentés sur FPGA par la suite.

- **Génération d'un Bitsream** : un bloc IP ne pouvant être envoyé directement sur FPGA, il convient de l'intégrer dans un *top-level*, décrit en langage VHDL, pour pouvoir notamment interfacer ses entrées et sorties avec les entrées et sorties physiques de la carte Nexys 4. Pour ce faire, l'environnement *Vivado* est requis et permet de générer un *bitsream* d'extension `.bit`.

- **Implémentation sur Nexys 4** : finalement, le *bitsream* peut être envoyé sur la carte cible, dans notre cas la carte Nexys 4, via le port série de notre ordinateur. A ce stade, les fichiers décrits en SystemC sont implémentés sur FPGA.

Le flot de compilation présenté ici allie le développement *software* de sources SystemC, et l'implémentation *hardware* des blocs ainsi créés. Il s'agit là du flot de compilation suivi lors du projet et les parties 2.2 et 2.3 suivantes viennent détailler sa mise en pratique.

2.2 Modélisation en SystemC et simulations

2.3 Implémentation sur FPGA et validation

Une fois les fichiers SystemC écrits et les simulations en *software* vérifiées, il convient de tester les blocs et chaînes entières en *hardware*. Pour cela, un environnement de test a été mis en place et il est illustré en Figure X.

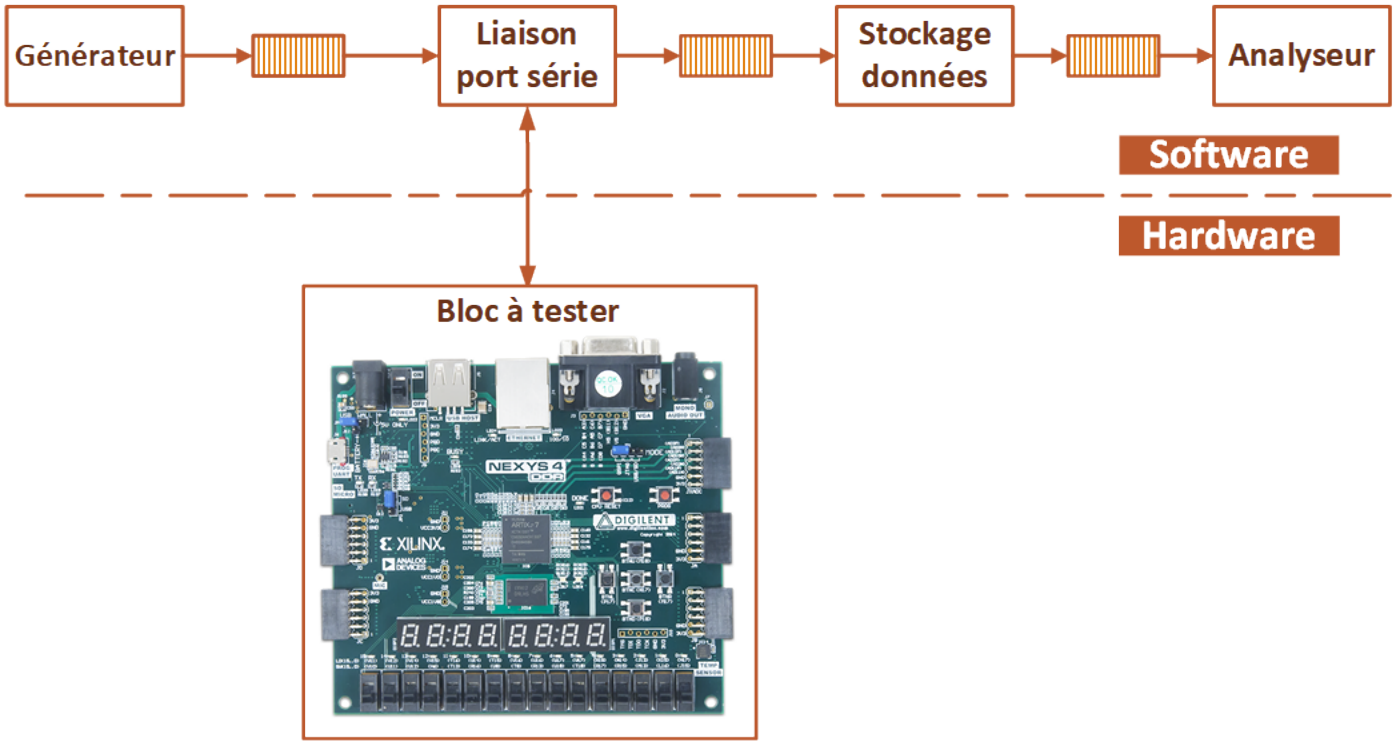

Figure 7 – Environnement de test

On distingue ici deux parties essentielles :

- la partie *software* : elle se déroule sur un ordinateur, sous *Vivado HLS* par exemple, et permet de lancer une simulation,

- la partie *hardware* : elle représente la carte Nexys 4 reliée en USB sur le port série de l'ordinateur, elle contient également l'implémentation du bloc IP à tester.

Avant de tester les chaînes globales développées en SystemC, les blocs réalisant la racine carrée ainsi que la puissance au carré ont notamment été implémentés sur FPGA. Pour ce faire, le bloc "*Générateur*" permet de venir lire les données d'entrée stockées dans un fichier texte, que ce soit les valeurs extraites du signal biomédical ou bien des valeurs plus rudimentaires pour tester la racine carrée. Après passage dans une FIFO, les données sont envoyées sur le port série et traitées sur le FPGA. Ce dernier contient l'IP que l'on souhaite tester en *hardware*, la racine carrée par exemple, et renvoie les données sortantes vers le bloc "*Stockage données*". Ce bloc écrit alors les données calculées sur le FPGA dans un fichier texte. Comme expliqué en 2.2, les données contenues dans une FIFO doivent être consommées, c'est pourquoi le bloc "*Analyseur*" se contente de venir lire dans la FIFO, pour fermer la chaîne de test.

En ce qui concerne l'échange de données entre la partie *software* et le FPGA, le module de communication utilisant l'UART et développé par Yannick Bornat a été utilisé. Cependant, les deux chaînes précédemment décrites manipulent des données de type *float*, donc sur 32 bits, tandis que l'UART ne reçoit et renvoie que des données sur 8 bits. Pour palier à cette incompatibilité, des *wrappers* ont été décrit en SystemC selon le modèle présenté en Figure X. Le *wrapper* IN reçoit les données de la partie *software* sur 32 bits et les envoie par parquets de 8 bits à l'UART, tandis que le *wrapper* OUT fait l'opération inverse et attend de recevoir quatre paquets de 8 bits pour reformer et renvoyer la donnée voulue sur 32 bits au bloc "*Stockage données*" présenté en Figure X. Ces deux wrappers ont alors été synthéthisés et exportés en tant qu'IP avec le bloc à tester.

Figure 8 – UART Wrappers

Lorsque les différents blocs ont été validés un à un sur cet environnement de test, les deux chaînes ont pu successivement être implémentées sur FPGA. Leur validation est, quant à elle, passée par une comparaison entre les données traitées et celles de référence obtenues après développement en VHDL de Yannick Bornat. La principale difficulté de cette démarche d'implémentation *hardware* résidait dans l'envoi et la réception des données entre la chaîne de simulation et la carte Nexys 4. En effet, le goulot d'étranglement se trouve dans la communication avec l'UART qui attend de recevoir une donnée avant d'en transmettre une autre. Une fois cette notion appréhendée il a été possible de vérifier le fonctionnement de n'importe quels blocs.

3 Résultats

3.1 Comparaison des sorties

3.2 Ressources utilisées

3.3 Différences

3.3.1 Débit

3.3.2 Nombre d'entrées

3.3.3 Point sur la consomation

4 Conclusion