HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Prof. Yann Thoma

Laboratoire VSE
semestre d'automne 2024 - 2025

Laboratoire VSE
Détection de spike - Traitement de signal

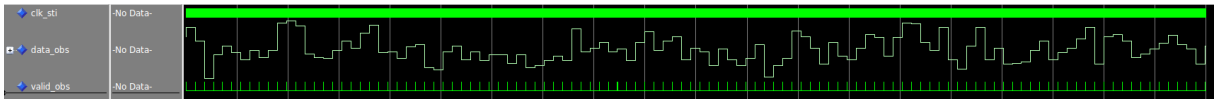
Introduction

Dans le cadre d'un projet de recherche, nous disposons d'un système capable de détecter des signaux électriques dans un puit contenant des neurones réels. Ces neurones émettent de temps en temps un signal qui s'appelle un *spike* et correspond à l'activation du neurone. Pour les biologistes, l'intérêt principal des expériences qu'ils effectuent est la détection de ces spikes.
Dans ce contexte, ce laboratoire vise à valider les interactions entre le logiciel et le matériel responsable de la détection ainsi que l'extraction de zones d'interêts d'un signal autour d'un spike.
Le composant matériel est responsable d'extraire des fenêtres d'intérêt du flux de données reçu en entrée. L'acquisition peut être démarrée et arrêtée depuis le logiciel, et lorsqu'une fenêtre d'intérêt a été sauvegardées, le processeur est averti via une interruption. Il peut alors aller lire les données dans une mémoire partagée.

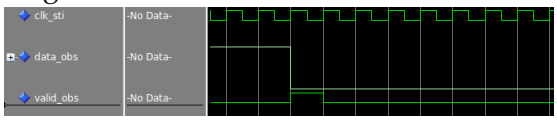
Cahier des charges

Le système FPGA est accessible via une interface Avalon, offrant un registre pour les commandes, un registre de statut et l'espace mémoire pour lire les données sauvegardées.
Il dispose de deux interfaces, une pour l'entrée du flux de données, et une avalon.

En entrée :
Le système reçoit un signal qui correspond à des données mesurées dans un puit de neurones. A chaque fois qu'un échantillon est disponible le flag de validité indique qu'une donnée est disponible.
Pour illustrer le fonctionnement, voici un exemple de signaux qui pourraient être observés en entrée :



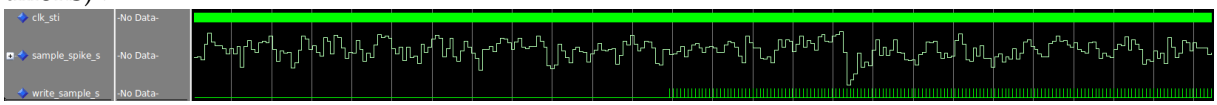
Flag valid zoomé :



Les signaux représentés ci-dessus sont les signaux fournis au système.

En sortie *interne* :

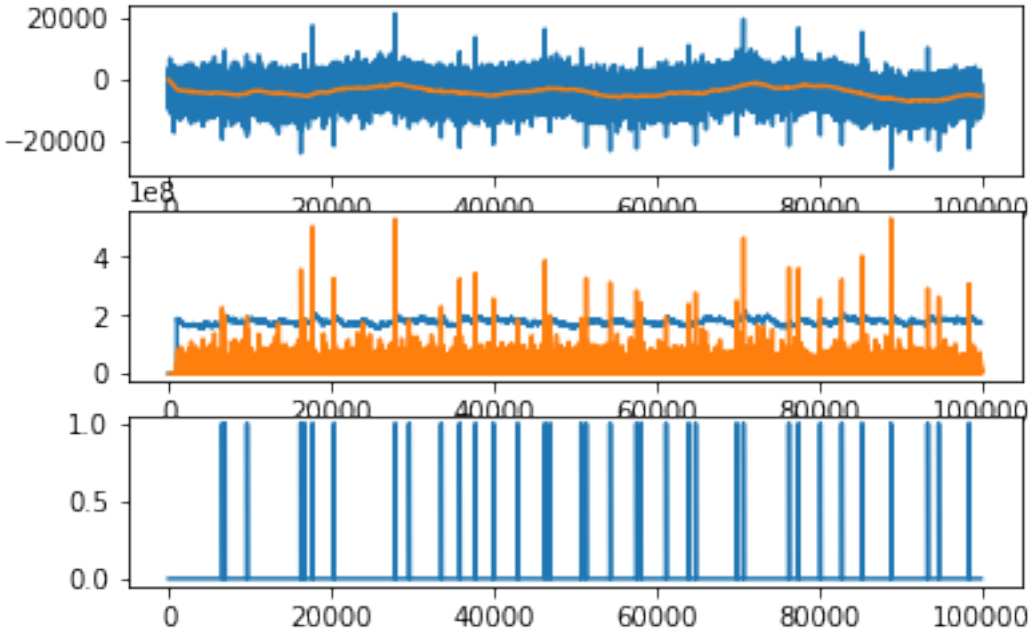
Le détecteur écrit les données sélectionnées dans un block RAM. Les données sont alors à disposition sur le bus avalon. Le chronogramme ci-dessous montre les signaux d'écriture de 150 échantillons dans le bloc RAM (avec une représentation analogique des valeurs d'échantillons) :



Processing de Spikes

Détection de Spikes

Afin d'avoir un système robuste et qui s'adapte largement plus au bruit du signal qu'un simple threshold, nous utilisons le concept suivant : nous regardons si notre échantillon est plus grand qu'un certains nombre de fois (facteur variable) la déviation standard du signal.
Le graphique suivant illustre le signal brut et sa moyenne glissante, le signal en valeur absolue après soustraction de la moyenne, et la détection d'un spike.



Pour ce faire, nous utilisons le développement mathématique suivant (qui offre une simplification intéressante pour l'implémentation sur FPGA).
La formule de base que nous posons consiste en la comparaison entre l'échantillon actuel, la moyenne glissante et un certain facteur multiplié par la déviation standard. Nous posons ainsi :

$$Deviation = |Echantillon - Moyenne| > Deviationstandard * Facteur$$

$$Deviation_t = |x_t - \bar{x}_t| > s \cdot C$$

Premièrement, nous devons calculer la moyenne glissante de notre signal. Cette moyenne glissante se fait sur un certains nombre de points. Dans notre cas, nous définissons que la moyenne glissante se fait sur 128 points (comme nous aurons une division à faire, l'aligner sur une puissance de deux permet de faire un shift (cablâge) au lieu d'une division). Afin de calculer la moyenne glissante sur N points, nous partons de la formule de base suivante :

$$\bar{x}_t = (\sum_{i=t-N+1}^{t} x_i)/N$$

Afin de ne pas stocker la valeur des échantillons pour ensuite les soustraires à la moyenne glissante, nous approximons l'échantillon au temps $t - N$ par l'échantillon actuel (cela provoque une erreur qui serait égale à la différence entre l'échantillon actuel et l'échantillon à $t - N$ divisée par la taille de la fenêtre, mais comme N est relativement grand, nous acceptons cette erreur car elle nous permet de ne pas stocker les valeurs des échantillons dans le temps. On obtient ainsi une formule approximative :

$$\bar{x}_{t+1} \approx \bar{x}_t + (x_{t+1})/N - (\bar{x}_t)/N$$

Où on peut encore simplifier par :

$$\bar{x}_{t+1} \approx \bar{x}_t + ((x_{t+1}) - (\bar{x}_t))/N$$

Nous avons à présent la formule pour la moyenne glissante approximée qui est implémentée en hardware.
⚠ Réfléchissez à ce calcul au lancement du système.
En exploitant cette moyenne est il est alors possible de calculer la déviation standard des échantillons, qui correspond à :

$$s_t = \sqrt{\frac{\sum_{i=t-N+1}^{t}(x_i - \bar{x}_t)^2}{N}}$$

NOTE : il est important de noter que la formule divise par N alors que la formule de la déviation standard divise par $N - 1$. Ceci est un choix afin de pouvoir faire une division câblée comme au point précédent. Afin de s'autoriser à utiliser la formule sous cette forme, il faut calculer l'erreur possible. Celle-ci est de $1/128$ ce qui est moins de 1%, ce que nous avons admis comme acceptable.

Afin d'éviter de devoir faire une racine carrée dans la FPGA, nous élevons cette formule au carré. On obtient (après simplification) :

$$s_t^2 = (\frac{1}{N} \sum_{i=t-N+1}^{t} x_i^2) - \bar{x}_t^2$$

Avec $\sum x_i^2$ qui correspond à la somme des carrés défini par :

$$Sum_t^2 = \sum_{i=t-N+1}^{t} x_i^2 = x_t^2 - \frac{Sum_{t-1}^2}{N} + Sum_{t-1}^2$$

Afin de s'adapter à la modification sur le calcul de la déviation standard, la formule de base pour la détection de Spike devient alors :

$$Spike =' 1' <=> D_t^2 > s^2 \cdot C^2$$

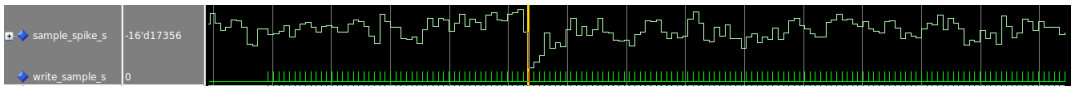
Où :
- `D` : différence entre échantillon et moyenne flottante
- `s` : déviation standard
- `C` : facteur de détection

Extraction de Spikes

Le concept est donc de devoir trouver un spike dans le signal généré. Les biologistes sont intéressés à enregistrer un certain nombre d'échantillons autour du spike. Nous avons défini qu'une fenêtre de 5 ms était nécessaire pour avoir une bonne visibilité du spike. Comme le signal que nous recevons génère des signaux à 30kHz, on peut calculer le nombre de points dans notre fenêtre, soit :

$$N = WindowDuration \cdot SamplingFreq = 0.005 \cdot 30000 = 150$$

Nous avons ainsi défini le nombre de points que nous devons enregistrer dans le block RAM.
Afin de pouvoir avoir une bonne visibilité du Spike, sa position dans la fenêtre est fixée au 50ème échantillon de la fenêtre.
Nous aurons ainsi un signal qui sera enregistré comme l'image présentée ci-dessous :



La barre verticale représente la position du Spike dans la fenêtre. Il y a donc 49 échantillons avant l'échantillon du spike et 100 après (ce qui fait les 150).

Système global hardware

L'interface Avalon offre le plan d'adressage suivant :

Adresse	Taille	Direction	Description
0	2	Rd	Registre de statut
1	2	Wr	Commandes
2	8	Rd	Offset de la prochaine fenêtre de données disponible
0x1000-0x10FF	16	Rd	Fenêtre de données (150 données de 2 octets)
0x1100-0x11FF	16	Rd	Fenêtre de données (150 données de 2 octets)
...	...	...	...
0x1F00-0x1FFF	16	Rd	Fenêtre de données (150 données de 2 octets)

Les bits de status sont les suivants :

Bit	Description
0	Indique qu'il y a une fenêtre de données disponible
1	Indique que le détecteur est en phase d'acquisition

Si le bit de statut 0 indique qu'une fenêtre est disponible, alors le registre à l'adresse 2 contient l'offset de la fenêtre disponible.
A la fin de l'enregistrement d'une fenêtre dans la mémoire, s'il s'agit de la première, alors une interruption est levée. Au niveau du composant lui-même il s'agit uniquement de mettre le signal `avl_irq_o` à `'1'` pendant un cycle d'horloge. Il n'y a pas de gestion des interruptions plus complexe.
Les bits de commande sont les suivants :

Valeur	Description
0	Démarre l'acquisition
1	Stop l'acquisition
2	Indique que toutes les données de la fenêtre ont été récupérées

Ces commandes sont réalisées si la donnée correspondante est écrite. La commande 2 est importante et doit être effectuée après que le logiciel a récupéré les 150 échantillons d'une fenêtre. Ceci permet de mettre à jour l'offset de la prochaine fenêtre de données disponibles.

Organisation mémoire du block RAM

En mémoire, l'espace réservé pour le stockage d'une fenêtre contient 256 cases afin de stocker les données, sur 16 bits. Seuls 150 emplacements sont exploités, pour les 150 échantillons qui composent la région d'interêt.

Index	Echantillons	Size [bits]
0	sample0	16
1	sample1	16
2	sample2	16
3	sample3	16
...	...	...
146	sample146	16
147	sample147	16
148	sample148	16
149	sample149	16
150	unused	16
151	unused	16
...	...	...
255	unused	16

TABLE 1. *Organisation de la RAM*

Système software

L'API logiciel développée offre les fonctions suivantes pour interagir avec le système FPGA :

```
void startAcquisition();

void stopAcquisition();

typedef void (*irq_handler_t)(std::string &);
void setInterruptHandler(irq_handler_t handler);

uint16_t getStatus();

uint16_t getWindowsAddress();

void readWindow(uint16_t *data);
```

Vérification

Un système relativement complet vous est fourni.
Du côté hardware, un banc de test SystemVerilog est responsable d'envoyer les données au système lorsque l'acquisition est actionnée. Ces données viennent d'un fichier que nous avons préparé avec des données pertinentes. Le testbench se connecte, grâce au DPI et à du code C, à un socket pour interagir avec le software embarqué et attend des commandes qu'il exécute.
Du côté software embarqué, le code attend la connection du testbench, active l'acquisition puis récupère les fenêtres lorsqu'une interruption est détectée. Après la récupération d'une fenêtre, une comparaison est faite entre celle reçue du hardware et celle calculée à partir du fichier de données brutes.

A faire

Etape 1

Premièrement, une lecture du code software révèle un problème d'architecture. La classe `FPGAAccess` vise à offrir un accès la FPGA, mais mélange du code haut et bas niveau. Afin de rendre le code plus propre, nous proposons de découper cette classe en deux : une qui gère uniquement l'accès à la FPGA via des méthode `avl_write()`, `avl_read()`, et `setInterruptHandler()`, et une qui instancie celle de bas niveau et offre les autres méthodes. Commencez donc par cette étape, qui ne devrait pas changer la fonctionnalité du système, mais le rendre plus modulaire.

Etape 2

Du côté SystemVerilog, nous pouvons noter qu'actuellement le testbench ne tolère qu'une activation de l'enregistrement, et ne gère pas le fait que celle-ci pourrait être stoppée puis redémarrée.
Modifiez-donc le code de `amiq_top.sv` afin qu'il soit possible de démarrer et stopper une acquisition en cours de simulation.
Modifiez légèrement la partie software pour valider la partie hardware.

Etape 3

Il est maintenant temps de réaliser quelques tests. Modifiez la partie software pour y ajouter quelques tests que vous jugerez pertinents. Il est suggérer d'avoir des testcases "simples", puis d'autres plus complexes. Il vous est également demandé d'exploiter les googletests pour la définition des testcases. A vous de modifier le code existant pour que les googletests puissent être exploités correctement.
Une idée serait de faire que le software puisse indiquer au hardware quel fichier utiliser pour lire les échantillons. Ceci permettrait que les testcases utilisent des fichiers différents.

Exécution

L'exécution du tout implique plusieurs exécutables, qui doivent être lancés dans l'ordre :

1. Soft embarqué
2. Simulation de la partie FPGA

Tout peut se faire en ligne de commande, mais le soft embarqué peut être lancé depuis votre IDE favorit en ouvrant, compilant et lançant le projet correspondant.

Soft embarqué

Nous utilisons cmake pour la compilation du soft embarqué En ligne de commande, allez dans le répertoire `embedded_soft`, puis tapez les commandes suivantes :

```
mkdir build
cd build
cmake ..
make
./embedded_soft
```

Simulation FPGA

En ligne de commande, allez dans le répertoire `fpga_sim`, puis tapez les commandes suivantes :

```
export PROJ_HOME=$(pwd)
./arun.sh -tool questa
```

A rendre

Comme a l'accoutumée, le code ainsi qu'un petit rapport sont à déposer sur cyberlearn.