MASTER Sciences pour l'Ingénieur 2° année
Spécialité Mécatronique et Microsystèmes
Parcours Électronique et Systèmes Embarqués
Université de Franche-Comté
2009-2010

Réalisation d'un système d'acquisition embarqué haute résolution

-

Utilisation du principe de stroboscopie
Application à l'échographie

Encadrants :
CARRY Émile
FRIEDT Jean-Michel

Rapport de stage
et travaux de :
CHRÉTIEN Nicolas

En collaboration avec

Département Temps - Fréquence

Abstract

The demand for systems capable of acquiring and digitizing RF signals is booming. Indeed, the ability to process and consider a digitized signal is simple and modular as the analog signals. A method of acquiring near the stroboscope can significantly increase the frequency of sampling : Equivalent Time Sampling.

This method, already used in the GPR (Ground Penetrating Radar), applies only to probe the impulse response of a medium. Indeed, the assumption of this principle is that the acquisition signal to be digitized must be either periodic or triggerable identical during acquisition.

The primary objective of this project is to develop a system capable of sampling ultrasonic echoes. Thus ETS makes sense for this application. The system, created during the training course, beyond the defined field of application at the start and managed to be used for radar or delay lines used as sensors. This is possible thanks to the modularity of the platform used, combining a processor (with an embedded Linux) and an FPGA, and allows ease of development.

However, the use of these two components involves programming the system in several languages (C and VHDL). In addition, this platform does not achieve the complete system. Part of the work is to look and operate more efficient external components for precision timing of the acquisition.

The results are inconclusive and the system available after the writing of this report provides greater sampling rate and a broader field of applications.

Keywords : Digital Acquisition, Radio Frequencies, Embedded Linux, Strobe, ETS.

Résumé

La demande de systèmes capables d'acquérir et de numériser des signaux radio-fréquences est en pleine expansion. En effet, la possibilité de traiter et d'étudier un signal numérisé est plus simple et modulaire que le traitement sur des signaux analogiques. Une méthode d'acquisition proche de la stroboscopie permet d'augmenter considérablement la fréquence d'échantillonnage : l'échantillonnage en temps équivalent ou *Equivalent Time Sampling* (ETS).

Cette méthode, déjà utilisée dans les GPR (Ground Penetrating Radar ou radar de sol), ne s'applique que pour sonder la réponse impulsionnelle d'un milieu. En effet, l'hypothèse de ce principe d'acquisition est que le signal à numériser doit être soit périodique soit déclenchable et identique pendant la durée d'acquisition.

L'objectif premier de ce projet est de réaliser un système capable d'échantillonner des échos ultra-sonores. Ainsi l'ETS prend tout son sens pour cette application. Le système, créé au cours de ce stage, dépasse le domaine d'application défini au départ et réussit à être utilisé pour des ondes radars ou encore des lignes à retard utilisées comme capteurs. Ceci n'est possible que grâce à la modularité de la plateforme utilisée, combinant un processeur (avec un système GNU/Linux embarqué) et un FPGA, et permettant une grande facilité de développement.

En revanche, l'utilisation de ces deux composants implique une programmation du système dans plusieurs langages (ici principalement C et VHDL). De plus, cette plateforme ne permet pas de réaliser le système complet. Une partie du travail consiste à chercher et exploiter des composants externes plus performants pour la précision temporelle de l'acquisition.

Les résultats sont concluants et le système encore en modification après l'écriture de ce rapport permet une plus grande fréquence d'échantillonnage soit un domaine d'application élargi.

Mots-clefs : Numérisation, Radio-Fréquences, Linux Embarqué, Stroboscopie.

Remerciements

Je tiens à remercier dans un premier temps, toute l'équipe du département Temps-Fréquence de l'institut Femto-st, ainsi que leur directeur Monsieur S. Ballandras, pour m'avoir accueilli et intégré lors ce stage . Tout particulièrement, Monsieur J.-M. Friedt et Monsieur É. Carry qui m'ont encadré tout au long du stage que ce soit pour des informations aussi bien techniques qu'administratives.

Je remercie également les personnes m'ayant aidé à surmonter les difficultés et les problèmes rencontrés lors de la réalisation du système telles que Monsieur G. Goavec-Mérou pour son aide précieuse sur le fonctionnement de la carte APF9328 ou encore Monsieur G. Martin pour le développement de la partie analogique.

Mes remerciements vont également à l'équipe d'ARMadeus Project pour la qualité des informations et des programmes disponibles et l'équipe pédagogique du Master Mécatronique et Microsystèmes de l'Université de Franche-Comté pour avoir assuré la partie théorique de ma formation et m'ayant apporté les connaissances nécessaires à la réussite de ce stage.

Table des matières

Chapitre 1

Introduction

La demande de systèmes capables d'acquérir et de numériser des signaux radio-fréquences est en pleine expansion. En effet, la possibilité de traiter et d'étudier un signal numérisé est plus simple (reproductibilité, stabilité, flexibilité [1]) et modulaire que le traitement de signaux analogiques. Certains systèmes commerciaux existants permettent une numérisation sur 12 bits avec des fréquences d'échantillonnage de 500 à 800 MS/s mais sont généralement coûteux et souvent intégrés au sein de châssis PCI ou PXI ce qui impose un encombrement minimum important.

L'objectif de ce projet est de réaliser un système embarqué capable d'échantillonner un signal à la fréquence de 4 GS/s. Naturellement, les conditions de fonctionnement d'un tel système sont limitées et définies avant la conception. L'hypothèse principale de départ est que le signal à numériser est reproductible et déclenchable afin d'user d'un principe, expliqué dans le chapitre *Principe de fonctionnement*, proche de la stroboscopie pour l'acquisition. Cette hypothèse limitera les applications possibles de ce système mais son but principal est de numériser des échos acoustiques ou de radar.

Les ondes acoustiques sont utilisées, entre autres, pour la vérification non destructive de soudures ou de collage de wafers, et les dispositifs sont généralement placés en milieu aqueux. Les ondes acoustiques se propageant à une vitesse d'environ 1500 m/s dans l'eau et à 6000 m/s dans l'acier, une période d'échantillonnage de 1 nanoseconde (1 GHz) implique alors une résolution de longueur respectivement 1,5 et 6 μm. Si l'on considère une pièce de 12 mm de profondeur placée à 5 cm du transducteur ultra-sonore, présenté dans la figure 1.1, le dernier écho arrive alors avec un retard de 70 μs $(2 * (0,05/1500 + 0,012/6000))$ par rapport à l'émission de l'onde d'excitation. Malgré cela, seuls les 12 mm de profondeur de la pièce contiennent les informations recherchées. Le système doit donc être capable de générer des long délais (quelques dizaines de μs) puis de numériser avec une grande résolution les échos utiles pour récupérer l'information, ce qui explique les 4 GS/s du système.

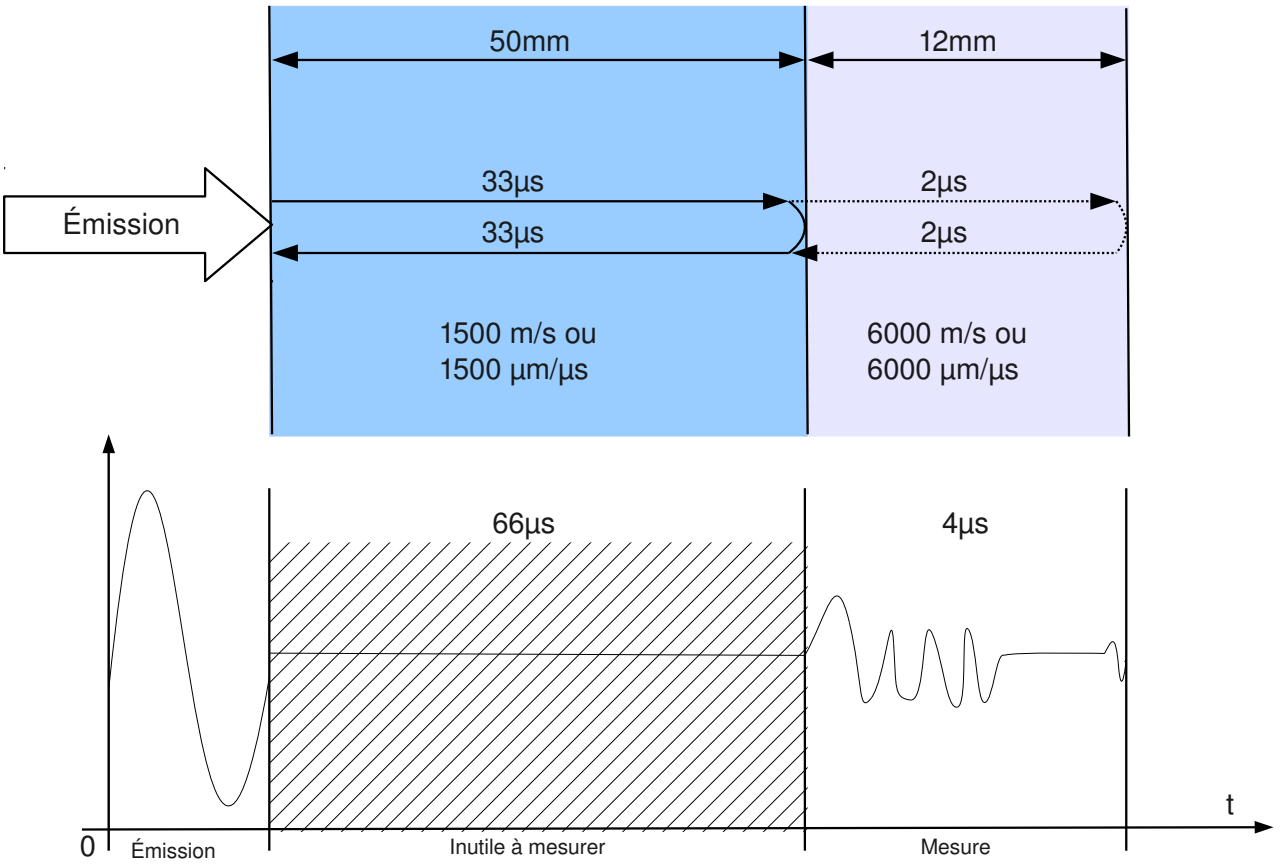


FIG. 1.1 – Schéma d'explication de l'exemple de calcul.

Pour réaliser ce projet, notre système est composé d'un convertisseur analogique-numérique précis et d'une ligne à retard programmable, le tout contrôlé par une carte électronique comportant un FPGA comme détaillé dans le chapitre *Contexte technique et choix des composants*. Une application à de telles fréquences implique un contrôle des temps et délais des signaux à travers chaque élément électronique du système.

La part la plus importante du travail à effectuer est de réussir à maîtriser les différents éléments électroniques interconnectés. Le système de contrôle permet de les synchroniser et de communiquer avec eux afin de récupérer les données utiles. L'ensemble des interactions entre les éléments ainsi que les points critiques de la programmation sont décrits dans le chapitre *Système opérationnel*.

Le système opérationnel conçu peut être testé, tout d'abord, dans le simple objectif de caractérisation et de mesure de ses performances puis pour démontrer son utilité dans des applications plus concrètes comme pour l'échographie acoustique, la caractérisation dans le domaine temporel de micro-systèmes expérimentaux normalement effectuée par des analyseurs de réseaux ou encore l'utilisation dans un radar. Les résultats de ces premiers tests et expériences sont présentés dans le

chapitre *Applications et résultats*.

Bien que l'utilisation commerciale de ce type de système peut être envisagée à long terme, son amélioration et son optimisation garantissent encore des sujets de travaux de recherche et des applications dans des domaines très différents.

Chapitre 2

Principe de fonctionnement

2.1 Introduction

Avant l'arrivée de l'électronique numérique, le traitement des signaux était effectué par des systèmes électroniques analogiques. Ces systèmes comportaient des inconvénients non négligeables pour une étude précise et fiable[1] :

- dérive et dispersion des caractéristiques des composants impliquant un manque de reproductibilité des résultats
- bruit ajouté par le système de traitement sur le signal à étudier
- impossibilité, le plus souvent, d'utiliser le même matériel pour d'autres applications ...

Par opposition, l'électronique numérique va permettre un traitement et une mémorisation des signaux avec une grande immunité au bruit. Sa principale contrainte reste alors de bien définir ses paramètres de quantification et d'échantillonnage.

2.2 Quantification et échantillonnage

2.2.1 Quantification et distorsion

Le problème de la quantification peut être réglé assez facilement, grâce aux progrès de la technologie des convertisseurs analogique-numérique, en augmentant le nombre de bits de codage. En effet, le rapport signal sur bruit théorique dû à la quantification d'un signal sinusoïdale d'amplitude pleine échelle d'un convertisseur N=12 bits donné par l'équation 2.1 nous montre l'immunité au bruit de quantification d'un tel codage (en supposant que la fréquence d'échantillonnage respecte le

[1] Arsène Perez-Mas. "Numérisation des signaux". http://pagesperso-orange.fr/arsene.perez-mas/signal/numerisation/numerisation.htm. Mis en ligne le 19 août 2000, consulté en mars-avril 2010.

théorème de Shannon).

$$SNR_{dB} = 20log_{10}(2) * N \simeq 6 * 12 = 72\ dB \tag{2.1}$$

Donc pour un signal de 3 V (amplitude pleine échelle) codé sur 12 bits, notre bruit de quantification (appelé aussi distorsion) est donné par l'équation 2.2. Ce bruit est considéré négligeable car le bruit analogique d'entrée, pour notre application et après conditionnement du signal, sera supérieur au mV. À titre de comparaison, les différentes valeurs de bruit pour la même amplitude de signal sont données dans le tableau de la figure 2.1.

$$V_{bruit} = V_{max}/2^N = 3/2^{12} \simeq 732\ \mu V \tag{2.2}$$

Nb_{bits}	V_{bruit}(en V)
4	0,18750
6	0,046875
8	0,011719
10	0,0029297
12	0,0007324
14	0,0001831

FIG. 2.1 – Tension théorique de bruit de distorsion généré par une quantification linéaire centrée d'un signal sinusoïdal d'amplitude 3 V (pleine échelle du convertisseur) en fonction du nombre de bits.

2.2.2 Échantillonnage

D'après le théorème de Shannon, la fréquence d'échantillonnage doit être au minimum égale à deux fois la fréquence du signal à étudier. Ainsi l'information utile est restituée en totalité. Bien entendu, il faut faire attention aux phénomènes de repliement de spectre en ajoutant un filtre passe-bas au signal concerné. Cependant pour obtenir une courbe dans le domaine temporel visuellement proche du signal analogique, le respect du théorème de Shannon ne suffit pas. C'est pourquoi la numérisation de signaux, exploitable dans le domaine temporel, en radio-fréquence compris entre 1 et 100 MHz implique des systèmes rapides pouvant échantillonner de 100 MSamples/s à 1 GS/s minimum.

L'inconvénient est que les convertisseurs analogique-numérique (ADC) atteignant 1 GS/s en sont encore au stade de développement[2, 3]. Ceux-ci comportent généralement des mémoires internes afin d'optimiser les temps de conversions et de stockage et dépassent rarement les 6 bits de résolution car ils ont pour objectifs d'être utilisés pour la télécommunication. Une seconde possibilité, largement exploitée par les oscilloscopes et les systèmes d'acquisitions hautes fréquences, est d'utiliser un ou plusieurs convertisseurs moins rapides dans un système permettant de les synchroniser afin d'acquérir

et de stocker les points par des méthodes de multiplexage (Real Time) ou encore, pour les signaux périodiques ou reproductibles, d'*Equivalent-Time Sampling* (ETS)[2].

2.3 Equivalent-Time Sampling

L'avantage principale de l'ETS est de pouvoir numériser un signal reproductible à, par exemple, 5 GS/s lorsque l'acquisition temps réel maximal du même matériel serait de 250 MS/s [3]. Cette fréquence d'échantillonnage est atteignable grâce à la propriété de périodicité ou de reproductibilité du signal à numériser. Le principe fondamental de cette technique, proche de la stroboscopie, est de récupérer l'ensemble des points de la courbe sur plusieurs reproductions ou périodes du signal. Le signal de commande de l'échantillonneur bloqueur[4] du convertisseur est décalé temporellement d'un certain délai incrémenté à chaque itération de reproduction du phénomène jusqu'à la numérisation complète (cf figure 2.2). Le pas d'incrément de ce délai définit la fréquence d'échantillonnage de la numérisation. Ainsi cette méthode est exploitable lorsqu'on excite un système passif pour en déterminer la réponse impulsionnelle : on excite le système et on visualise et mémorise l'état de celui-ci à l'instant $N * \Delta t, N \in \mathbb{N}$.

En pratique, les oscilloscopes effectuent généralement plus d'une mesure par période du signal[5] et réduisent ainsi le temps d'acquisition d'une fenêtre de visualisation. En revanche, la technique utilisée est le "Random-Interleaved Sampling"[6]. Cette technique, légèrement différente, dépend du temps de conversion des ADC utilisés afin d'optimiser le temps d'acquisition. Le signal à numériser est découpé en tranches temporelles. La durée de ces tranches correspond au pas d'échantillonnage souhaité. L'acquisition est effectuée à la vitesse maximum du convertisseur analogique-numérique jusqu'à ce que chaque tranche contienne un point numérisé. Cette méthode est dite aléatoire car, au final, le signal n'est pas échantillonné périodiquement et les points récupérés peuvent être proches ou éloignés entre eux comme montré dans la figure 2.3[7].

[2]National Instruments. "Les 10 points essentiels pour choisir un numériseur/oscilloscope" Chap. 3 "Modes d'échantillonnage". http://zone.ni.com/devzone/cda/tut/p/id/5550. Mis en ligne le 18 février 2008, consulté en mars-avril 2010.

[3]Fréquences d'échantillonnages des modes RIS et RTS des numériseurs PCI/PXI NI 5114

[4]Sample and Hold (S&H) : élément de l'ADC permettant de garder la tension d'entrée stable le temps de la conversion.

[5]Tektronix. "Real-Time Versus Equivalent-Time Sampling". http://www2.tek.com/cmswpt/tidetails.lotr?ct=TI&cs=Application+Note&ci=14295&lc=EN. Mis en ligne le 01 janvier 2001, consulté en mars-avril 2010.

[6]RIS par opposition au RTS : Real-Time Sampling

[7]National Instruments. "Equivalent-Time Sampling and Random Interleaved Sampling". http://zone.ni.com/reference/en-XX/help/370592G-01/digitizers/ris/. Mis en ligne en juillet 2006, consulté en mars-avril 2010.

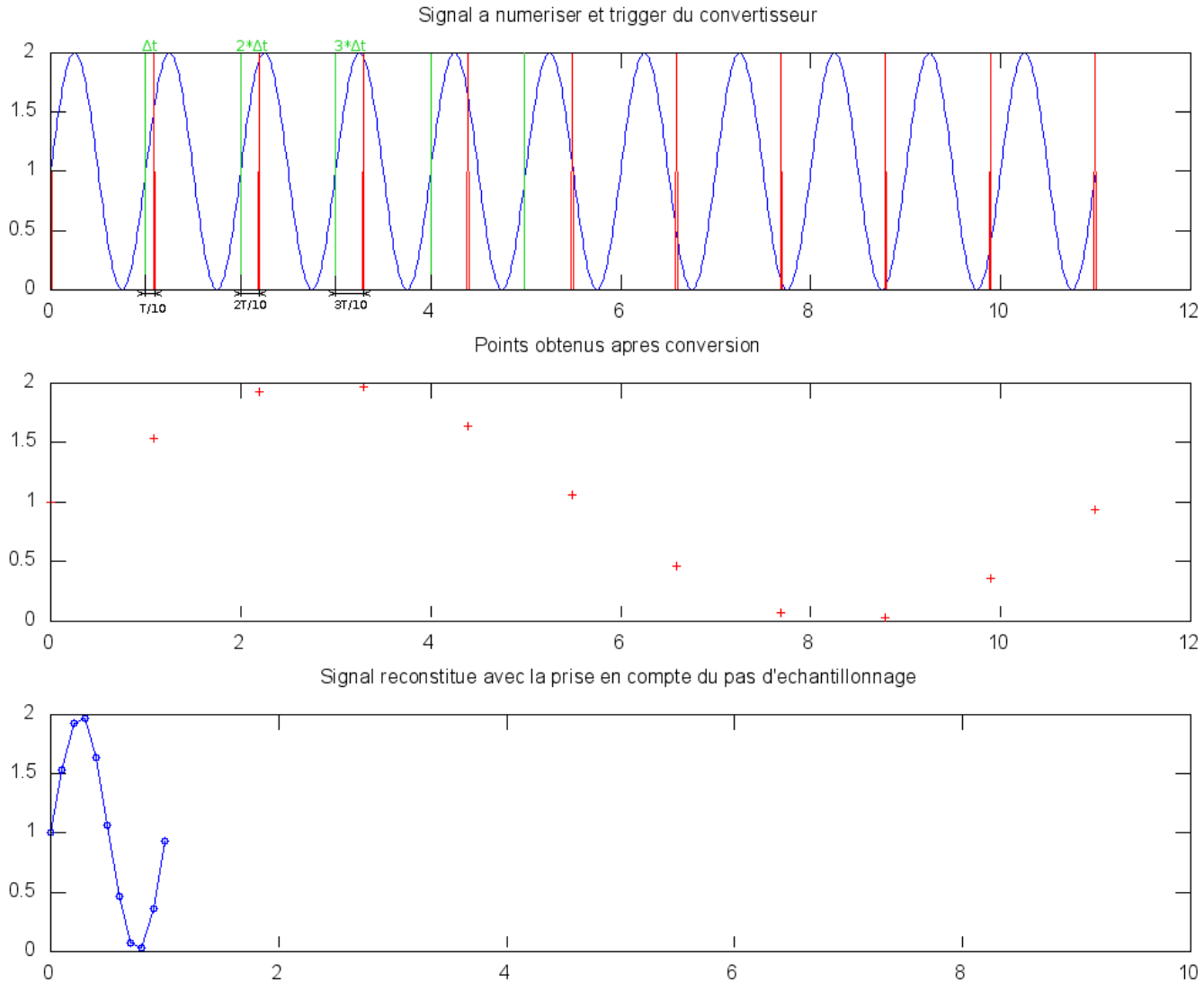


FIG. 2.2 – Présentation simple de l'ETS avec un pas d'incrémentation du délai de $1/10^{e}$ de période soit une période de signal numérisée de 10 points.

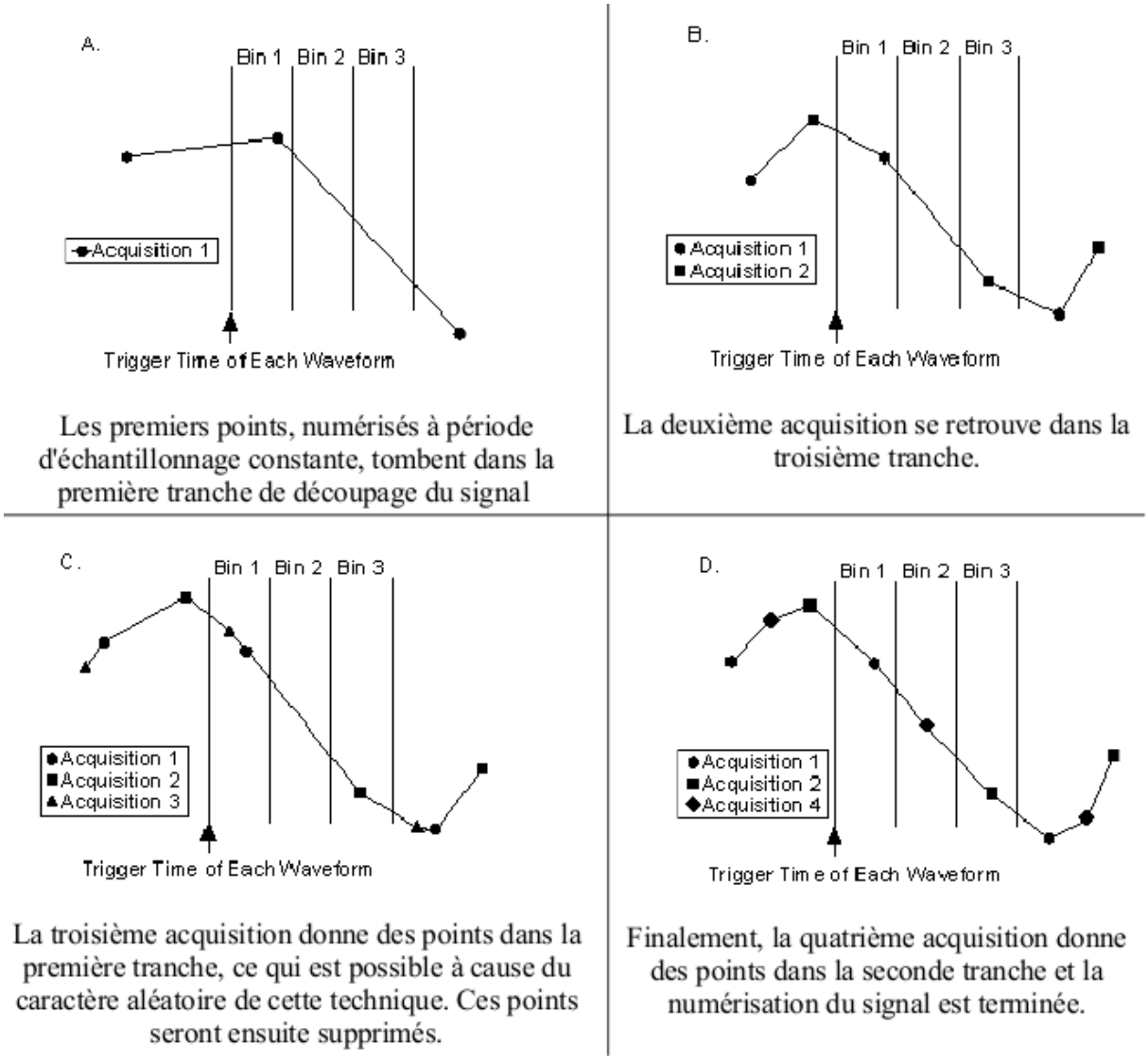


FIG. 2.3 – RIS à une fréquence d'échantillonnage 3x supérieure à la fréquence maximale des ADC.

2.4 Description schématique du système

Le système, décrit dans la suite de ce rapport, utilisera le principe d'ETS décrit plus haut afin d'échantillonner le signal à 4 GS/s. Ce qui implique de pouvoir commander un ADC à 250ps près (avec une précision proche de la picoseconde) à l'aide d'un générateur de délai programmable. Ce générateur de délai programmable va déclencher le S&H du convertisseur pour mémoriser l'état du système à caractériser à un instant défini précisément. Ensuite, la conversion analogique-numérique plus lente peut avoir lieu. Il est nécessaire également de contrôler le signal d'excitation du système passif avec la même précision pour garantir l'instant de déclenchement du S&H par rapport à l'instant correspondant à la stimulation.

C'est pourquoi il est nécessaire de synchroniser et coordonner l'ensemble des opérations à effectuer. Ceci est possible grâce à un système de contrôle qui va produire les différents signaux de déclenchement des générateurs et du convertisseur en respectant des contraintes temps réel dures. Il va également récupérer/stocker provisoirement les données avant de les transmettre à un système de calcul plus puissant pour le traitement du signal.

L'ensemble *générateur d'impulsion* et *système à tester*, présent dans la figure 2.4, est dépendant de l'application pour laquelle la carte sera utilisée. Cela peut être un générateur d'impulsions et un transducteur acoustique pour la caractérisation de soudures (système à tester) par exemple ou encore un amplificateur suivi d'une ligne à retard. Le système de contrôle, le générateur de délai et le convertisseur analogique-numérique constituent l'objet à réaliser lors de ce projet.

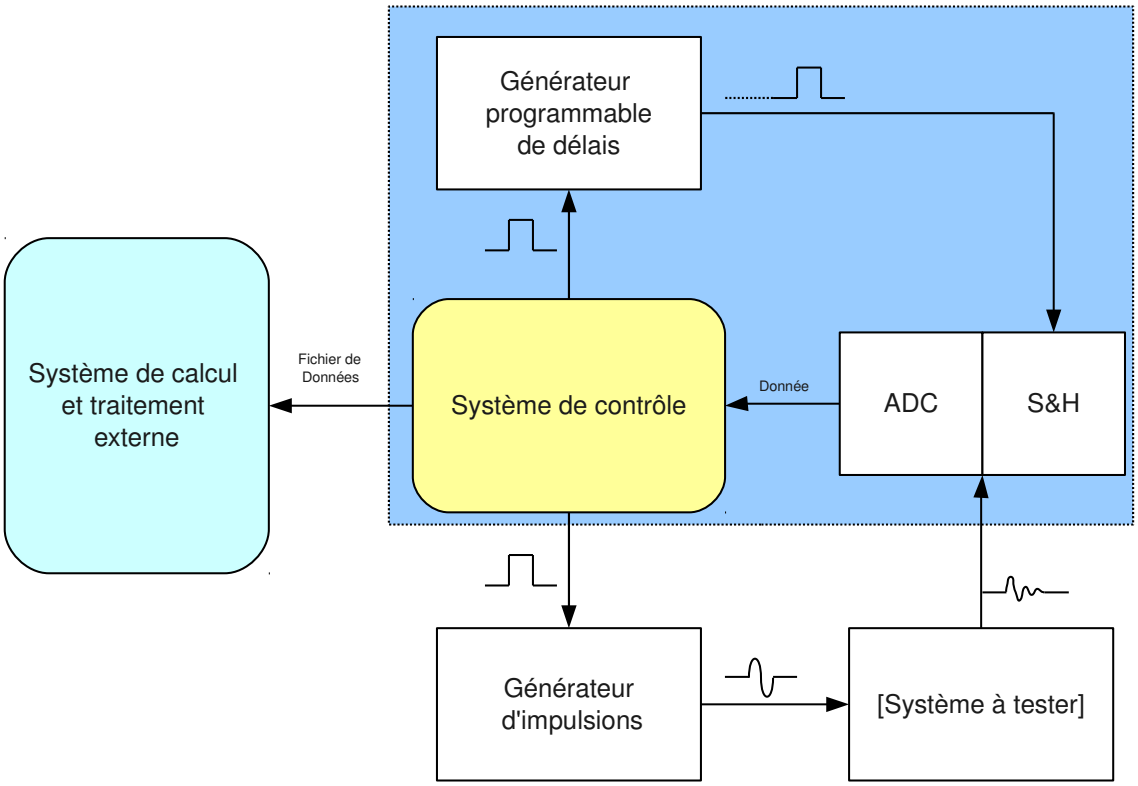


FIG. 2.4 – Schéma de fonctionnement simple du système embarqué d'acquisition numérique supérieur au GHz

Chapitre 3

Contexte technique et choix des composants

3.1 Introduction

Les contraintes induites par les techniques d'acquisitions présentées dans le chapitre précédent, induisent un choix des composants relativement précis sur certaines caractéristiques, telles que les délais de réactions de ceux-ci et leurs incertitudes temporelles sur les déclenchements. Naturellement, les fabricants des synthétiseurs rapides présents sur le marché ne divulguent pas les références des composants utilisés. Ainsi, la recherche de composants et principalement du convertisseur analogique-numérique n'est pas simplifiée. Au final de cette étape de recherche, il a été retenu les composants suivants :

Convertisseur analogique-numérique : LTC1407 (Linear Technologic)

Ligne à retard programmable : DS1023-25 (Maxim)

Carte de contrôle : APF9328 (ARMadeus) accompagnée de la carte de développement DevLight

3.2 Convertisseur Analogique-Numérique

3.2.1 Contraintes sur le choix du convertisseur

Un point important sur le fonctionnement du convertisseur était d'avoir une entrée (une broche) disponible pour déclencher une mesure. Cette entrée doit activer un S&H rapide pour pouvoir garantir une précision temporelle sur la mesure à effectuer[1].

[1]Voir le chapitre *Principe de fonctionnement*

La difficulté est de trouver des S&H ayant un faible jitter[2] sur le déclenchement. Cette incertitude, non nulle et interne au composant, est en générale négligeable pour les signaux basses fréquences mais elle doit être prise en compte pour l'acquisition de signaux hautes fréquences. L'influence du jitter est illustrée dans la figure 3.1[4]. Lors du déclenchement du S&H, une imprécision (notée ici $\pm\Delta t$) provoque une erreur de la tension (notée ici $\pm\Delta V$) à numériser par le convertisseur.

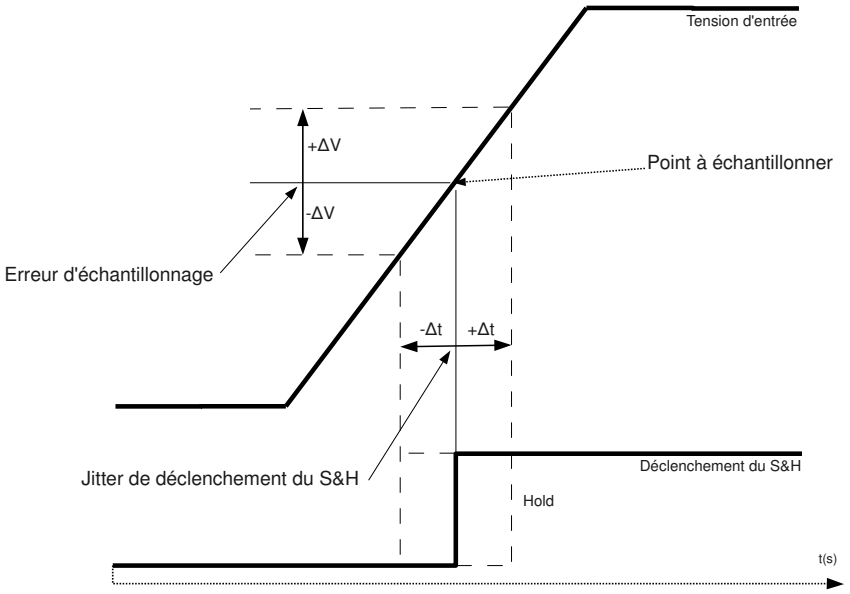


FIG. 3.1 – Influence du jitter de déclenchement d'un S&H sur la numérisation d'un point.

3.2.2 Caractéristiques du LTC1407A

Le composant LTC1407 est un convertisseur analogique-numérique rapide spécialement conçu pour la télécommunication et les systèmes embarqués. Les principales caractéristiques avancées par le constructeur Linear Technologic sont une conversion en continu de 3MS/s, une plage d'acquisition de 0 à 2,5V (alimentation 3,3V), une sortie de référence à 2,5V, une faible consommation, la possibilité d'une programmation série et la présence de mode veille pour le composant. La caractéristique la plus importante pour ce projet est la faible imprécision sur le déclenchement du S&H. Celle-ci est inférieure à la picoseconde[3]. Malgré tout, le convertisseur possède un décalage entre la détection du front montant du signal de conversion et le blocage du signal par le S&H de quelques nanosecondes. Néanmoins, ce décalage étant constant, il ne gênera pas la mesure en utilisant la technique de l'ETS.

[2]Incertitude temporelle

[3]t_{JITTER} : Sample-and-Hold Aperture Delay Time Jitter : 0,3 ps[5]

3.3 Ligne à retard programmable

La ligne à retard, comme son nom l'indique, va permettre de décaler temporellement un signal d'un délai x. Dans le cadre de ce projet, la valeur de ce délai doit être variable et le signal à décaler est le signal de déclenchement du S&H. Ainsi, d'après le principe expliqué précédemment, le signal de sortie de la ligne à retard est envoyé sur le signal de déclenchement de l'ADC. De plus, pour obtenir un échantillonnage rapide (supérieur au GHz), la différence entre deux valeurs de délai (résolution) doit être au maximum d'une nanoseconde (1GHz). Naturellement, ce signal doit présenter une plus faible imprécision (ou équivalente) que le jitter de l'ADC pour le bon fonctionnement du système.

Il existe plusieurs méthodes pour générer des retards mais les méthodes intéressantes pour ce projet sont celles qui peuvent être intégrées facilement (boîtier DIP ou SOIC, par exemple). Ainsi, les méthodes de réflexion de faisceaux d'ultrasons pour les anciennes télévisions SECAM ou encore les lignes à magnétostriction utilisées dans certaines anciennes imprimantes ne seront pas approchées ici. Une méthode analogique, longtemps utilisée dans la télécommunication, consiste à utiliser un réseau d'inductances et de capacités pour créer un délai correspondant aux latences d'établissement du courant et de la tension dans ces éléments. Cette méthode est encombrante et peu efficace (déformation du signal initial, délai de l'ordre de la ms à la centaine de ns généralement).

Les constructeurs ont mis au point une seconde méthode en éliminant les inductances, ce qui permet une plus grande intégration du dispositif. Ici la tension de charge d'un ou plusieurs condensateurs est comparée à une tension de référence. Lorsque la charge dépasse cette tension, la sortie passe à un état logique haut ou bas suivant l'état de l'entrée à retarder. Bien entendu, cette méthode ne convient donc qu'aux signaux numériques. Encore une fois, la limite de cette méthode reste le délai minimum le plus précis ne descendant pas en dessous de la ns et la difficulté de fabrication du système (à cause de la précision demandée pour la valeur des condensateurs) [6, 7].

D'autres lignes à retard utilisent le temps de commutation de portes logiques. Une chaîne de N portes NOT (N étant un chiffre pair) permet ainsi de créer des cellules de délai de 2 ns puis un DAC permet de remplacer le condensateur dans la méthode précédente, pour pouvoir obtenir des délais inférieurs à la nanoseconde (voir figure 3.2). Une gamme de ligne à retard (DS1023) du constructeur Maxim, utilisant cette méthode, présente des caractéristiques idéales pour cette application[8]. Ces composants possèdent des retards programmables, la valeur du retard à appliquer au signal d'entrée étant donnée par un mot de 8 bits au travers d'une communication proche du SPI, et décalent des signaux TTL. Les lignes à retard de cette gamme présentent des résolutions différentes suivants le modèle. Le modèle retenu (DS1023-25) a des pas de 250 ps, ce qui donne la possibilité d'un échantillonnage à 4 GS/s.

En revanche, le délai maximum étant de 255*250 ps soit 63750 ps, le composant seul ne suffit

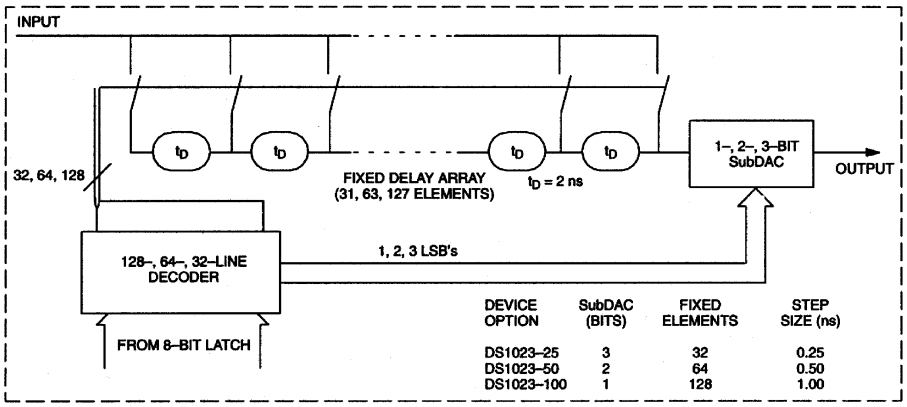


FIG. 3.2 – Fonctionnement des lignes à retard DS1023-25, DS1023-50, DS1023-100.

pas pour avoir une fenêtre de visualisation de mesure correcte (de l'ordre de la microseconde, comme expliqué dans l'introduction). La présence d'un FPGA sur le système de contrôle va pouvoir palier à ce problème en permettant de créer, à l'aide de compteurs simples, des délais par pas d'une période d'horloge de cadencement de celui-ci (voir figure 3.3).

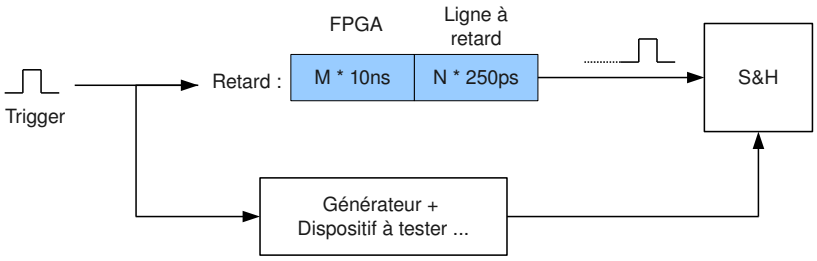


FIG. 3.3 – Génération du retard par le FPGA et la ligne à retard.

3.4 Système de contrôle

3.4.1 Description de la carte ARMadeus

La carte à microprocesseur APF9328 est équipée d'un microprocesseur ARM9 à 200 MHz, d'un FPGA Spartan 3 (200K portes), de SDRAM et de FLASH. Elle est facilement intégrable dans un système embarqué grâce notamment à ses régulateurs et ses convertisseurs de niveau (RS232/USB)[4]. Son architecture permet alors d'utiliser un système GNU/Linux comme système d'exploitation. Pour

[4]ARMadeus systems. "APF9328". http://www.armadeus.com/english/products-processor_boards-apf9328.html. Mis en ligne en 2007-2008, consulté en mars-avril 2010.

le besoin du projet, cette carte est montée sur un kit de développement (APF9328_dev_light) permettant de disposer des connectiques RS232, USB et Ethernet.

Utiliser un système GNU/Linux comme système d'exploitation de la carte permet une grande facilité de développement : chaîne de compilation, compilation croisée, programmation de pilote simplifiée. De plus, la communauté importante de développeurs sous GNU/Linux permet de trouver plus rapidement de l'aide, voir des codes sources de programme à modifier. D'autre part, grâce au noyau Linux et aux paquets déjà présents dans l'utilitaire de configuration de la carte (buildroot), il existe déjà des outils permettant une manipulation simplifiée de celle-ci. Par exemple, la communication par Ethernet (ssh, telnet), le partage de dossier avec l'ordinateur hôte (mount et nfs), l'éditeur de fichier en mode console (vi), l'utilitaire d'accès aux « registres » du FPGA (fpgaregs) sont autant d'outils présents par défaut ou à inclure grâce au buildroot, utiles au développement.

3.4.2 Utilité de l'ensemble FPGA - Microprocesseur

La carte APF9328 possède deux éléments essentiels permettant de séparer les tâches d'une application ayant des contraintes différentes. Le microprocesseur MC9328 (architecture iMx) permet d'effectuer des tâches lentes mais complexes, comme la récupération, le traitement et le partage de données ou encore d'utiliser des pilotes et programmes déjà conçus pour la communication (SPI, Ethernet). Le FPGA, quant à lui, est idéal pour toutes les actions temps réels demandant donc une plus grande précision temporelle. Il peut également servir à tous les traitements d'entrées/sorties en parallèles avec les composants contrairement au microprocesseur qui effectue ces tâches séquentiellement. Enfin, la possibilité de reconfigurer le FPGA « au vol » par le microprocesseur pendant l'exécution d'une application augmente l'importance du couple FPGA/microprocesseur. Naturellement, un outil de communication entre ces deux éléments clés de la carte est nécessaire afin de structurer les échanges : le bus Wishbone[5].

Le FPGA est utile pour une deuxième raison. Comme expliqué précédemment, la ligne à retard utilisée ne permet de créer des délais maximum que de 64 ns environ. Afin d'obtenir des fenêtres de visualisation de l'ordre de la microseconde, le FPGA, à l'aide de compteurs programmés en vhdl, est capable de générer des délais de plusieurs microsecondes (et même au-delà de la seconde pour des gros compteurs) avec une résolution de l'ordre de la dizaine de nanoseconde (suivant la fréquence de l'horloge de cadencement). Ainsi les délais fins (en-dessous de la nanoseconde) sont générés par la ligne à retard et des délais plus important (multiples de 10 ns) sont générés par le FPGA.

[5]Voir le chapitre *Système opérationnel* pour plus de détails

Programmation des délais en peigne

Les latences internes de la ligne à retard imposent un délai maximum de 500 ns entre la fin de la programmation d'une valeur et son influence sur la sortie. De plus, l'horloge de la communication du SPI étant limitée par la ligne à retard à 10 MHz, il faut au minimum 800 ns (8 bits*100 ns) pour la programmer. Pour une meilleure communication, il a été choisi la fréquence de 375 kHz soit un temps de programmation minimum de 20 μs. Le FPGA, quant à lui, est accessible plus rapidement, grâce notamment à la communication entre le FPGA et le microprocesseur cadencée à l'horloge interne de la carte (environ 96 MHz). Afin de limiter le temps d'acquisition, il est préférable d'incrémenter les délais générés par le FPGA plus souvent que les délais générés par la ligne à retard. Ainsi l'acquisition du phénomène est semblable à un peigne, montré dans la figure 3.4. La série 1 correspond aux points numérisés pour un délai nul de la ligne à retard et toutes les incrémentations du FPGA. Ensuite, la valeur de délai de la ligne à retard est incrémentée (série 2), avec les temporisations adéquates dans l'exécution du programme, et le délai du FPGA est réinitialisé. L'acquisition continue ainsi, avec chaque incrémentation de la ligne à retard correspondant aux différentes séries, jusqu'à obtention de tous les points désirés.

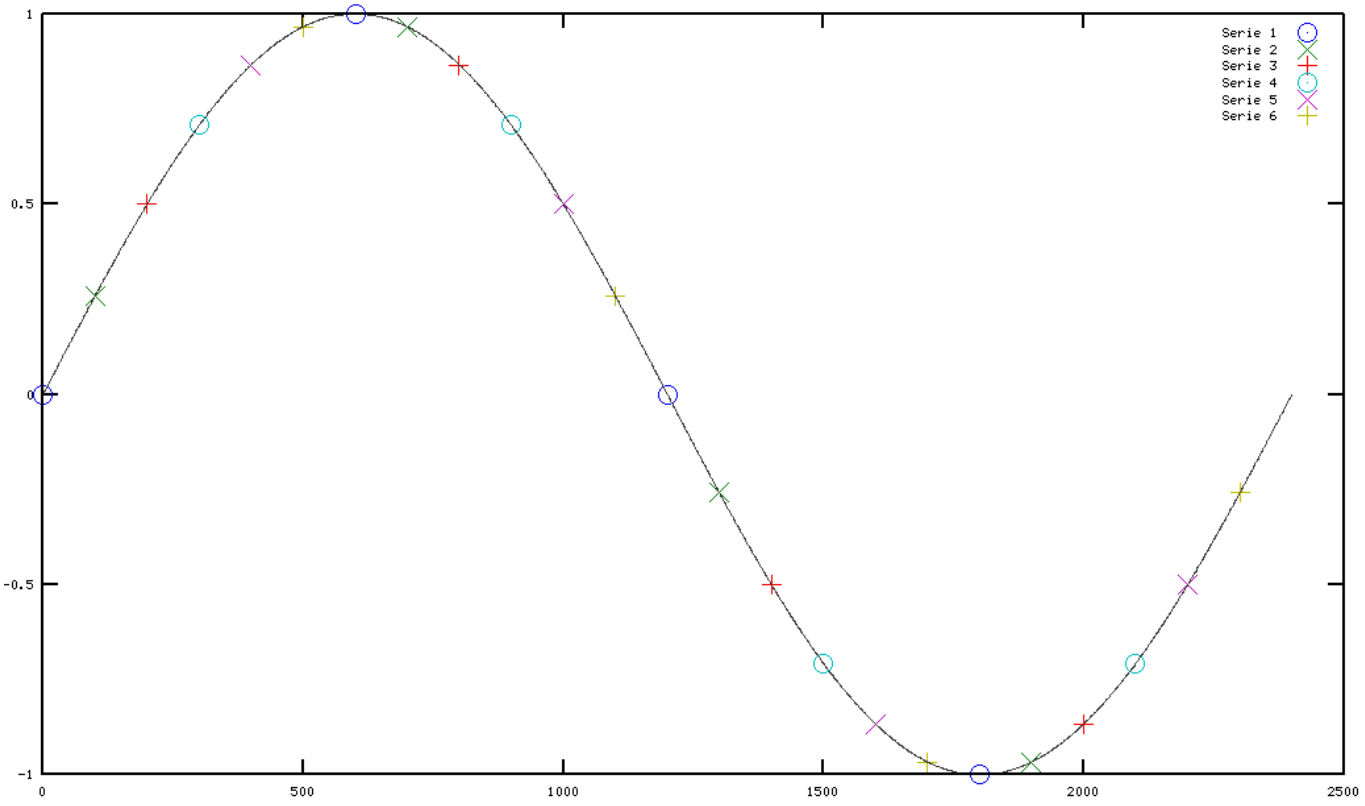


FIG. 3.4 – Solution aux latences de programmation de la ligne à retard : acquisition en peigne

3.5 Schéma complet et détaillé du système

Le schéma, présenté dans la figure 3.5, complète le premier synoptique du projet (figure 2.4 en ajoutant le nom des composants et des broches utiles au fonctionnement du projet. De plus, il indique la présence des signaux et retards fondamentaux entre chaque composant.

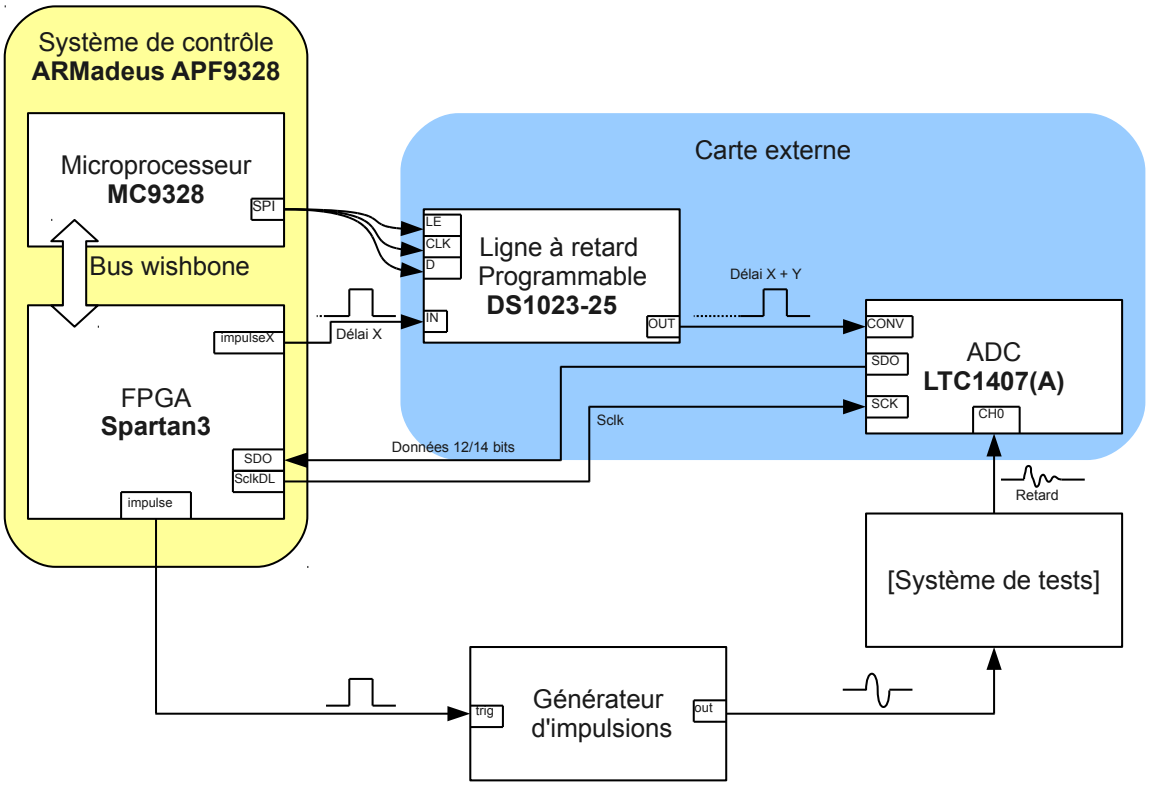


FIG. 3.5 – Schéma complet et détaillé du système avec le nom des ports d'entrées/sorties des programmes et des composants.

Chapitre 4

Système opérationnel

4.1 Utilisation du FPGA

Comme expliqué précédemment, le FPGA est utile pour toutes les applications demandant des contraintes temps-réel fortes mais doit pouvoir communiquer avec le microprocesseur qui s'occupera de stocker les données. Cette section présentera rapidement le bus de communication utilisé entre le FPGA et le microprocesseur, un outil de programmation du FPGA, les contraintes de programmation imposées par le convertisseur analogique-numérique ainsi qu'une description succincte du module final. L'ensemble des programmes commentés sont en annexes mais le code ne sera pas expliqué en détail dans ce chapitre, seul l'organisation générale sera décrite.

4.1.1 Présentation du bus Wishbone

Le bus Wishbone est un bus informatique optimisé pour du matériel reprogrammable (semblable au bus Avalon de la société Altera) dont les spécifications ont été placées dans le domaine public. Le projet ARMadeus a optimisé ce bus pour pouvoir communiquer entre le microprocesseur et le FPGA. Ainsi le FPGA comprend plusieurs modules servant au contrôle du bus et le microprocesseur envoi données et adresses sur certaines broches reliées au FPGA, le tout cadencé à la vitesse d'horloge interne à la carte, pour pouvoir communiquer avec l'ensemble des programmes « utilisateurs » sur le FPGA. En contre-partie, les modules de contrôle du bus peuvent renvoyer des signaux d'interruptions ou des données au microprocesseur.

Le système Wishbone optimisé par ARMadeus qui sera implanté dans le FPGA se compose des programmes VHDL (ou Verilog) suivants[1] :

[1] ArmadeuS Project. "A simple design with Wishbone bus". http://www.armadeus.com/wiki/index.php?title=A_simple_design_with_Wishbone_bus. Dernière modification le 10 juin 2009, consulté en mars-avril 2010.

i.MX Wrapper : l'interface microprocesseur vers le bus Wishbone

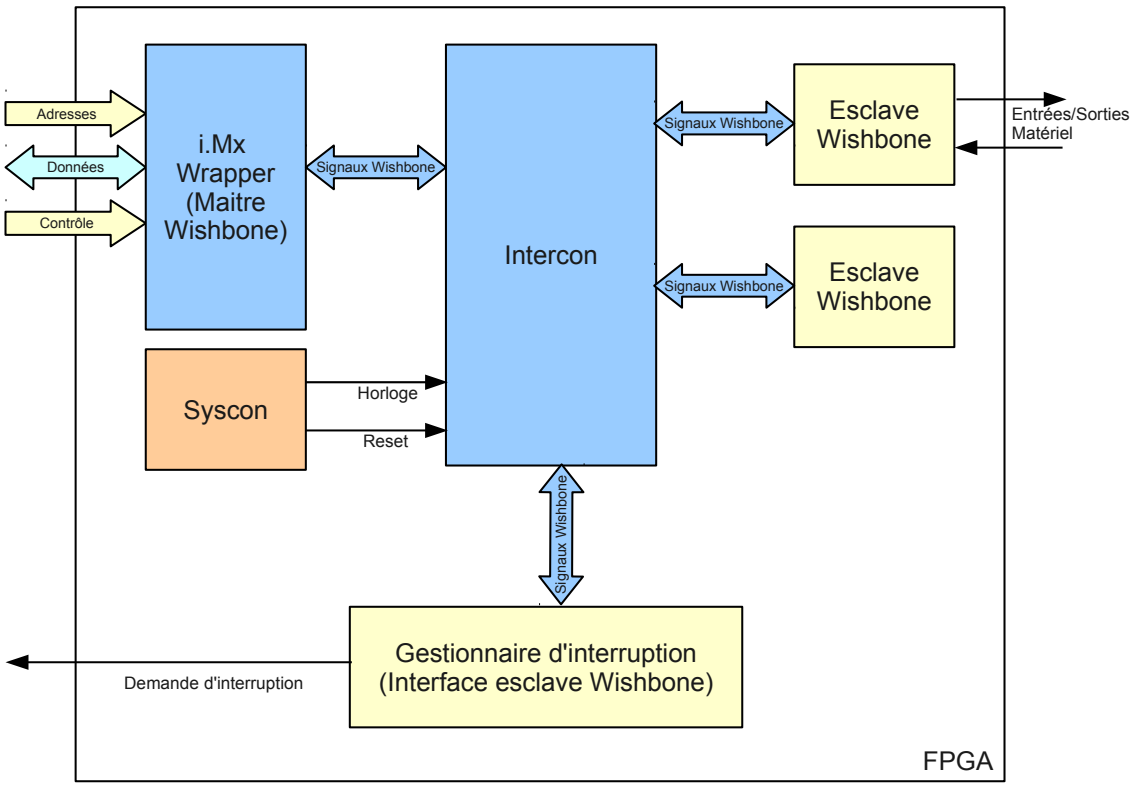
Syscon : ce composant va gérer les signaux CLK (généré par une PLL ou directement issu de l'i.MX) et RESET (synchrone).

Intercon : ce composant devra être généré automatiquement par le programme POD (voir la section suivante), il va faire le lien entre tous les composants faisant parti du système Wishbone.

Gestionnaire d'interruption : ce composant est un esclave Wishbone et va centraliser toutes les demandes d'interruption et les remonter vers l'i.MX.

Esclaves Wishbone : ceux-ci représentent tous les autres composants « utilisateurs » avec une interface Wishbone esclave qui sont accessibles via l'*i.MX Wrapper*. Ces composants peuvent également avoir des entrées/sorties externes.

FIG. 4.1 – Composition du bus Wishbone sur le FPGA de la carte ARMadeus.



4.1.2 POD

« Peripherals On Demand », POD, est une application open-source, codée en python et développée par le projet ARMadeus, simplifiant l'intégration de périphériques virtuels (ou com-

posants) dans un FPGA. Il possède, entre autres, les avantages suivants[2] :

- Il peut utiliser les propriétés des applications propriétaires externes (Xilinx ISE, Altera Quartus) pour configurer le FPGA.
- Il est théoriquement multi-plateforme (Windows, Linux, MacOS).
- Il peut générer les composants utiles au fonctionnement du bus Wishbone en VHDL ou Verilog et les regrouper et connecter avec les composants utilisateurs.
- Il peut également générer des pilotes pour les modules accessibles avec le microprocesseur.
- Il peut générer et connecter plusieurs composants esclaves identiques (avec des entrées sorties distinctes) ainsi que leurs pilotes dans le même projet, ce qui permet d'obtenir par exemple des chaînes de compteur ou de diviseurs complexes en cascades à partir d'un seul composant.

Pour pouvoir générer le projet ISE contenant tous les fichiers utiles au bus Wishbone ou le fichier binaire à flasher dans le FPGA, POD a besoin de quelques fichiers essentiels (disponibles en annexes). Ces fichiers sont structurés dans une arborescence à respecter :

wb16.xml : le fichier de configuration utile à POD.

hdl/multi_sampl.vhd : fichier top du projet VHDL.

hdl/wishbone_interface.vhd : lien entre l'Intercon et le reste du composant esclave.

hdl/*.vhd : les autres programmes utiles au fonctionnement du composant esclave (hdl/delay_sclk.vhd, hdl/gene_impulse.vhd, hdl/diviseur.vhd, hdl/registre.vhd).

Le fichier wb16.xml contient l'ensemble des informations pour la génération du projet défini par les balises obligatoires suivantes :

- generics : les variables « generic » déclarées dans les vhdl (ici l'identifiant du composant).
- hdl_files : les fichiers vhd du projet, dont le fichier top possédant un argument supplémentaire (istop="1").
- interfaces : les différentes interfaces (entrées/sorties de l'esclave) groupées en sous-ensemble dont deux sont essentielles au fonctionnement : candr (**c**lock **and** **r**eset) et swb16 (signaux Wishbone).
- ports : les définitions de chaque port des interfaces.
- registers : les registres, et leurs adresses, accessibles par le microprocesseur au travers du bus Wishbone.

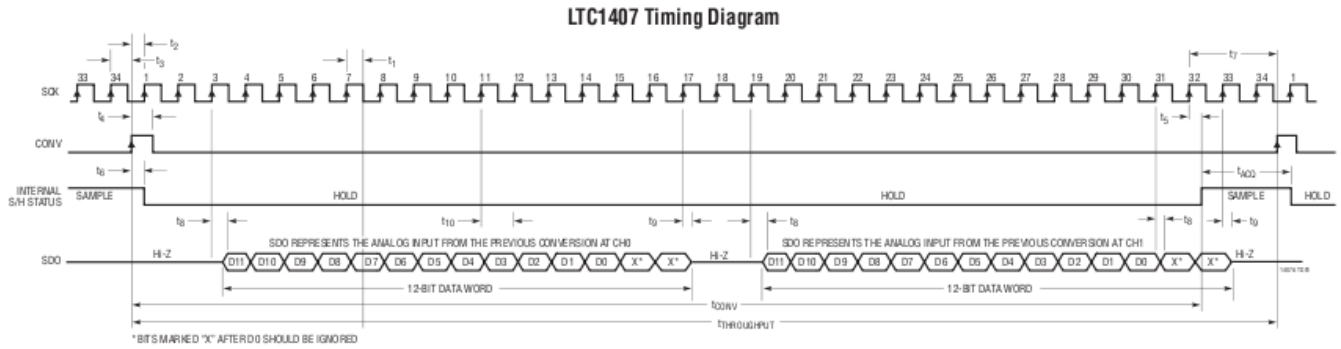
4.1.3 Programmation de l'ADC

Le composant LTC1407(A) présenté dans le chapitre précédent peut être interrogé à l'aide de deux fils. En effet, la communication établie est en série et synchrone et il est indispensable

[2] ArmadeuS Project. "POD Global description". http://www.armadeus.com/wiki/index.php?title=POD_Global_description. Dernière modification le 20 mars 2009, consulté en mars-avril 2010.

d'avoir un fil de données et un fil d'horloge. Les indications pour l'interrogation sont données par la documentation du composant et peuvent être résumés par la chronogramme de la figure 4.2.

FIG. 4.2 – Chronogramme de fonctionnement du LTC1407.



Le fonctionnement de l'interrogation est le suivant. Après avoir reçu la commande de lancement de conversion (créneau visible sur le fil CONV) et les trois premiers fronts montants d'horloge sur le fil SCK, le convertisseur envoie la valeur convertie sur l'entrée CH0 en commençant par le bit de poids fort. Une fois les 14 bits envoyés (pour le LTC1407A ou 12 bits utiles et 2 inutiles pour le LTC1407) et après trois nouveaux fronts montants d'horloge, le convertisseur envoie la valeur convertie sur l'entrée CH1. Tous les bits sont valides sur les fronts descendants l'horloge. Comme l'horloge est fournie par l'utilisateur (plus précisément par le FPGA), il faut respecter les différents timing décrits dans la datasheet. Mais dans le cas de ce projet, la vitesse de lecture de données n'étant pas critique, il suffit de choisir une horloge de fréquence assez faible pour ne pas entrer dans des cas qui peuvent être considérés comme critiques. Par exemple une horloge d'environ 5MHz[3] est comprise entre la fréquence maximum (environ 50MHz) et la fréquence minimum (100kHz) acceptées par le composant et fonctionne parfaitement tout en gardant une vitesse de transmission correcte[4].

4.1.4 Organisation du composant esclave

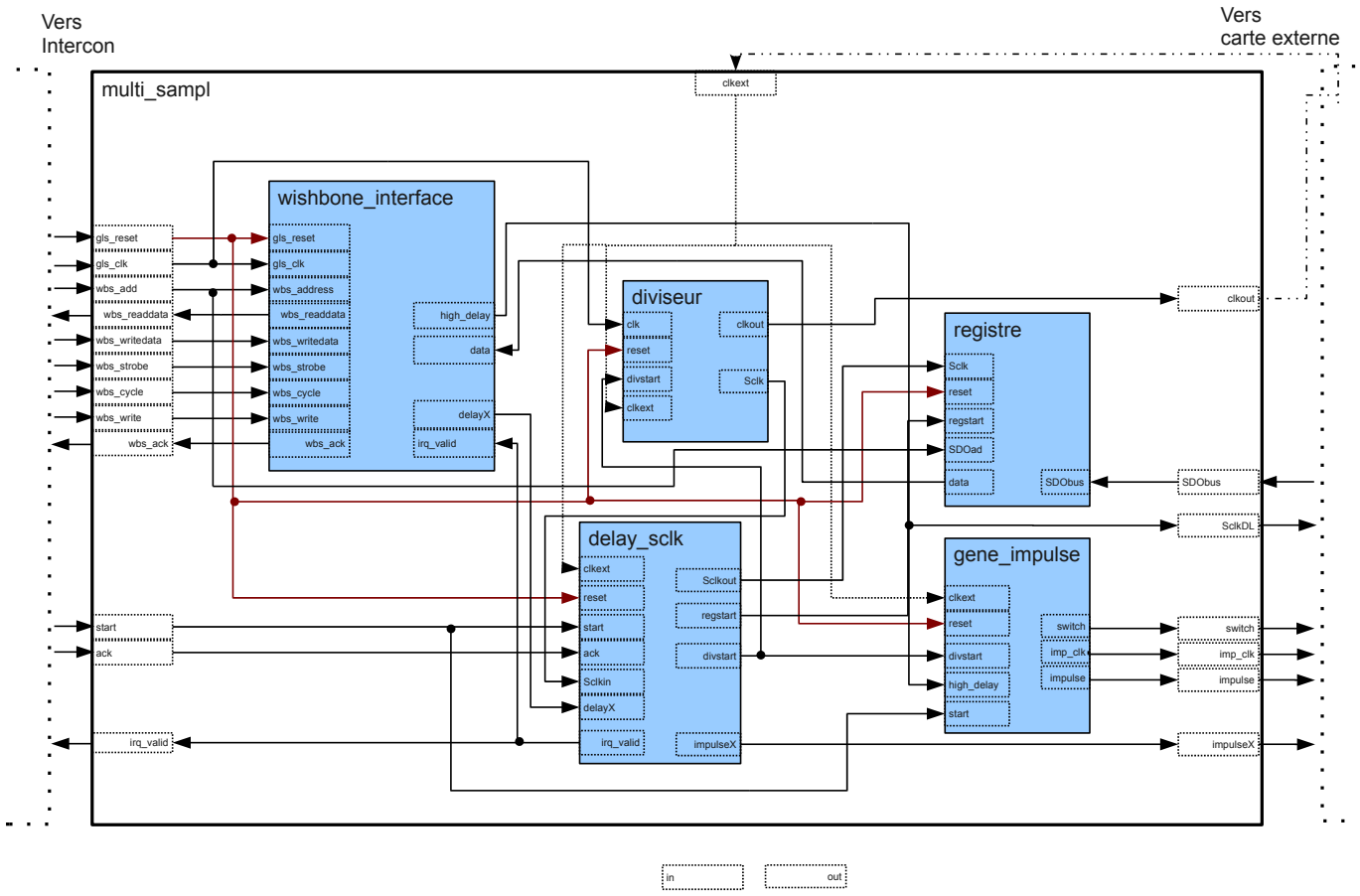
La figure 4.3 présente les connections des différents VHDL du composant esclave créé :

multi_sampl : fichier top du composant esclave. Il regroupe les autres sous-programmes et s'occupe des liaisons entre eux. Il sépare également le signal *wbs_add* entre le programme "wishbone_interface" (2 bits de poids faible) et "registre" (bits de poids fort)

wishbone_interface : il s'occupe de traiter les demandes communiquées par le bus wishbone (principalement les commandes de lecture/écriture sur les différents registres). Il envoie également une

[3] 20 divisions de l'horloge interne
[4] Soit 6,8μs pour la transmission d'une trame complète

FIG. 4.3 – Schéma de l'organisation des sous-programmes du composant wishbone esclave.



valeur précise, connue par le programme utilisateur du processeur, indiquant que la conversion n'est pas terminée lors de la demande de lecture d'une valeur du convertisseur.

diviseur : il s'occupe de la division d'horloge pour la communication avec l'ADC et sort également l'horloge interne de la carte (qui sera branché sur la broche *clkext* si on utilise l'horloge interne pour cadencer le programme).

gene_impulse : il génère le trigger (sortie *impulse*) utile à la synchronisation de générateur d'impulsion externe. De plus, il génère 10 périodes d'horloge (interne ou externe) qui, après amplification, peut servir à l'excitation d'un dispositif (par exemple une ligne à retard) et contrôle une porte pouvant être utilisée comme commande de switch entre l'émission et la réception d'une antenne.

delay_sclk : il est composé d'une machine à 4 états cadencée par l'horloge externe :

- init : initialisation des sorties
- s_delay : comptage du retard interne au FPGA (*delayX*)
- s_sclk : activation du programme registre et envoi de l'horloge de communication avec l'ADC

– s_read : indication de fin de récupération des données (*irq_valid* à 1) et attente de l'acquittement de la part du microprocesseur (signal *ack*)

registre : il lit les valeurs renvoyées par l'ADC lors de l'activation par le programme "delay_ sclk".

Ce composant est relié sur des broches d'entrées/sorties externes ou redirigés à l'i.MX Wrapper grâce au composant Intercon généré automatiquement par POD.

4.2 Linux Embarqué et ARM9

Cette section est consacrée à la programmation du microprocesseur sous GNU/Linux. Pour cela, il est possible de développer le code source sur un ordinateur, à l'aide d'une chaîne de compilation configurée pour le microprocesseur en question, et de transférer le programme une fois compilé sur l'APF9328. La récupération des données n'étant pas critique pour cette application, le gestionnaire d'interruption du bus Wishbone ne sera pas utilisé.

4.2.1 Mise en place de la liaison SPI pour la programmation de la ligne à retard

Présentation du SPI

La liaison SPI, Serial Peripheral Interface, est un bus d'échange synchrone de données, créé par Motorola. Il fonctionne avec un maître et un ou plusieurs esclaves. Le maître cadence la communication en fournissant l'horloge de synchronisation. Les esclaves peuvent écouter les données du maître ou répondre aux requêtes sur deux fils distincts, respectivement MOSI[5] et MISO[6]. La sélection des esclaves se fait à l'aide de fils dédiés à chaque esclave appelés chip select ou encore SS[7]. Le SPI peut fonctionner selon quatre modes de configurations différents. Les différences entre ces modes de configuration portent sur l'état de la ligne d'horloge au repos (état haut ou bas) et le front actif de l'horloge (montant ou descendant).

La programmation de la ligne à retard ne respecte pas toutes les caractéristiques du SPI. La principale différence vient du fait que le composant est activé par un état haut du fils de chip select au lieu d'un état bas. Le mode de configuration utilisé correspond aussi bien au mode 0 qu'au mode 3 (front montant de l'horloge actif) car la ligne à retard n'est pas influencée par l'état de repos de la ligne d'horloge .

[5]Master Output, Slave Input (généré par le maître)
[6]Master Input, Slave Output (généré par l'esclave)
[7]Slave Select, Actif à l'état bas, (généré par le maître)

Modification du module SPI pour l'architecture iMX

Le microprocesseur possède un bus SPI qui est utilisé, sur la carte APF9328, pour la communication avec un convertisseur analogique-numérique et un capteur de température. En revanche, dans l'état actuel de la chaîne de compilation fournie par ARMadeus, il manque quelques modifications pour pouvoir utiliser ce bus depuis l'espace utilisateur. Ce problème a été réglé pour la carte APF27 avant ce projet et sera certainement résolu par la suite avec le patch (disponible en annexe) créé lors de ce projet et communiqué à ARMadeus.

Pour réaliser ce patch, il a été pris comme exemple les programmes déjà existants pour l'APF27. Tout d'abord, le patch rajoute la bibliothèque spidev.h et gpio.h correspondant à l'architecture en question dans le fichier de configuration des périphériques de l'APF9328 (apf9328-dev.c). Entres-autres modifications, il définit les broches à initialiser pour la communication SPI, ceux communes aux autres périphériques SPI (clock, MISO, MOSI) :

```
43+ /* SPI1 GPIOs */
 + imx_gpio_mode(PC14_PF_SPI1_SCLK);
45+ imx_gpio_mode(PC16_PF_SPI1_MISO);
 + imx_gpio_mode(PC17_PF_SPI1_MOSI);
```

Puis le chip select, à l'état haut, utilisant la broche 18 du port B (qui a été choisi pour sa facilité d'accès sur la carte de développement) :

```
 + /* PortB 18 is used as chip select (in GPIO mode) */
49+ DR(1) |= 1 << SPIDEV_CS_GPIOB; /* Initializes it High */
 + imx_gpio_mode(GPIO_PORTB | SPIDEV_CS_GPIOB | GPIO_OUT | GPIO_GIUS | GPIO_DR);
```

L'utilisation du bus SPI dans l'espace utilisateur est possible grâce au pilote spidev[8] et à la modification du fichier de déclaration du matériel présent sur les cartes de développement, pour rendre possible l'utilisation de ce pilote (modification réalisée par le patch).

Enfin, le programme utilisateur donne les paramètres de fonctionnement du module SPI tels que la fréquence d'horloge, le nombre de bits par mot[9], puis envoie les trames en plaçant le mot dans une structure communiquée au pilote au travers de la fonction ioctl[10] :

```
  struct spi_ioc_transfer tr = {
89    .tx_buf = (unsigned long)tx,
      .rx_buf = (unsigned long)rx,
91    .len = ARRAY_SIZE(tx),
      .delay_usecs = delayspi,
93    .speed_hz = speed,
```

[8]Ce driver permet justement d'utiliser des fichiers de périphériques pour communiquer avec un ou plusieurs composants branchés en SPI.

[9]Code source en annexe *B.1 Multi-Sample.c*, lignes 256 à 305.

[10]Code source en annexe *B.1 Multi-Sample.c*, fonction *transfer*, lignes 79 à 98.

```
    .bits_per_word = bits,
};

retour = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
```

4.2.2 Utilisation du bus Wishbone du côté microprocesseur

La communication entre le microprocesseur et le FPGA se fait avec des lectures et écritures (au travers du bus Wishbone) dans des registres définis dans les fichiers VHDL. Pour cela, un programme permet de renseigner une adresse de registre et, en option, une donnée au Wishbone : *fpgaregs*[11]. Il a été défini quatre registres différents dans les programmes VHDL :

ID : contient l'identificateur du composant esclave en question.

DELAY : contient la valeur du délai effectué par le FPGA entre le front montant de l'impulsion servant au trigger et l'impulsion envoyée à la ligne à retard.

HIGH : contient la valeur du nombre de divisions d'horloge effectué pour la communication avec le convertisseur analogique-numérique.

DATA : contient la valeur renvoyée par l'ADC (de 12 ou 14 bits) une fois la conversion finie. Si la conversion n'est pas finie, les deux bits de poids fort de ce registre de 16 bits sont mis à 1 afin d'indiquer au programme utilisateur que la conversion n'est pas terminée.

L'adressage de ces registres est effectué également par POD. Il suffit ensuite d'écrire ou de lire à l'adresse de base du composant (concrètement 0x800 pour le module esclave de ce projet) additionnée à un offset correspondant à chaque registre[12]. Cet offset est obligatoirement un multiple de deux car le bit de poids faible de l'adresse est relié à la masse. Les offsets de ces quatre registres peuvent donc être codés sur 2 bits.

4.2.3 Exécution et fonctionnement du programme utilisateur

Les données récupérées à la lecture du registre DATA pour chaque point échantillonné sont placées tout d'abord dans un tableau créé dynamiquement au lancement du programme. En effet, au lancement du programme, celui-ci va calculer le nombre de points de la fenêtre de visualisation, ainsi que les délais à appliquer à la ligne à retard (delay_DL) et au FPGA (delay_FPGA) pour effectuer correctement la numérisation[13] :

[11] Programme repris par la fonction fpgaregs dans le fichier *Multi-Sample.c*, ligne 12 à 17.

[12] Initialisation de la mémoire partagée avec le FPGA en annexe *B.1 Multi-Sample.c*, ligne 330 à 340 et exemple d'écriture et lecture ligne 342 et 387.

[13] Code source en annexe *B.1 Multi-Sample.c*, ligne 203 à 255

```
230  for(i = 1; i < (nbrpoints); i++){
       delay_DL[i] = (delay_DL[i-1] + step);
232    delay_FPGA[i] =  delay_FPGA[i-1];
       res[i] = 0 ;
234    if (chan == 0){
         res_chan2[i] = 0 ;
236      res_chan3[i] = 0 ;
         res_chan4[i] = 0 ;
238    }
       if ((delay_DL[i]) >= tpsperiodDL){
240      delay_DL[i] = delay_DL[i] - tpsperiodDL;
         delay_FPGA[i] = delay_FPGA[i] + tpsperiodDL;
242    }
     }
```

La création des délais est la suivante. Le tableau delay_DL est incrémenté par pas de valeur step (pas d'échantillonnage). Lorsque cette valeur atteint la valeur maximale avant l'incrémentation du FPGA (valeur calculée en début de programme[14]), la valeur de delay_FPGA est incrémentée et delay_DL remis à sa valeur initiale. Par contre, pour le moment, les tableaux delay_DL et delay_FPGA ne sont pas encore triés pour l'exécution des délais *en peigne*[15]. Ceci est effectué juste après :

```
245  //Tri du tableau avec les valeurs delay_DL croissantes
     quickSort(delay_DL, 1, nbrpoints-2);
247
     for(i = 1; i < (nbrpoints); i++){
249    if (delay_DL[i+1] != delay_DL[i]){
         quickSort(delay_FPGA, j, i);
251      j = i + 1;
       }
253  }
     quickSort(delay_FPGA, j, nbrpoints);
```

Tout d'abord, les tableaux sont triés avec des valeurs de delay_DL croissantes grâce à la fonction quickSort() (algorithme de tri rapide). Ensuite, pour chaque valeur différente de delay_DL, les sous-tableaux sont triés afin d'obtenir des valeurs de delay_FPGA croissantes.

Paramètres du programme

Le programme a besoin de plusieurs paramètres à indiquer lors du lancement du programme par la commande *Multi-Sample delay window step clk_freq trig file [loading]* :

[14] Calcul des temps et nombre de points, ligne 189 à 202 de l'annexe *B.1 Multi-Sample.c.*

[15] Voir le chapitre *Contexte technique et choix des composants - Utilité de l'ensemble FPGA - Microprocesseur*

delay : délai entre le trigger généré et le premier point d'acquisition (en ns). Ce délai est un multiple de la période d'horloge (interne ou externe) car il sera effectué par le FPGA.

window : la largeur temporelle de la fenêtre de visualisation, en nanosecondes également.

step : le pas d'échantillonnage en picosecondes. Pour le moment, ce pas est limité par la résolution de la ligne à retard externe qui est de 250 ps.

clk_freq : la fréquence d'horloge de cadencement du FPGA en MHz. Il est possible d'utiliser l'horloge interne de 96 MHz en mettant 0 à ce paramètre.

trig : indication sur le mode de trigger. Ce paramètre est à 0 si le système génère le trigger et à 1 si on lui fourni un signal de synchronisation.

file : le nom de fichier où stocker les données.

[**loading**] : En option, ce paramètre indique s'il faut recharger le FPGA.

Mise en forme des résultats

Ensuite les résultats de la numérisation, convertis en millivolts, sont placés dans le fichier[16] sous forme d'un tableau de n lignes (n étant le nombre de points numérisés) et 4 ou 7 colonnes présentées ci-dessous :

1. Temps en picosecondes ayant pour origine l'émission du trigger
2. Tension mesurée en mV (1 ou 4 colonnes pour une acquisition multi-voies)
3. Délai appliqué à la ligne à retard en ps
4. Délai appliqué au FPGA en ps

4.3 Matériel

4.3.1 Schéma électrique

Le schéma électrique ci-dessous (figure 4.4), effectué à l'aide du logiciel EAGLE, ne présente pas la version finale de la carte réalisée (dont le schéma et le layout sont disponibles en annexe). Malgré tout, ce circuit à deux entrées analogiques comporte l'ensemble des fonctionnalités essentielles à expliquer. L'ensemble des valeurs des composants et de leurs fonctionnalités est présenté dans le tableau de la figure 4.5. Naturellement, les composants utiles à la mise en forme du signal dépendront des caractéristiques de ce signal et de l'application visée[17].

[16] Copie dans le fichier de résultat, ligne 398 à 420 de l'annexe *B.1 Multi-Sample.c*.
[17] Quelques exemples d'applications sont présentés dans le chapitre suivant.

Le filtre passe-bas peut servir à filtrer un éventuel bruit haute fréquence venant perturber la mesure et le montage suiveur à isoler le convertisseur. En revanche, le réglage de l'offset est indispensable car les tensions mesurables sont comprises entre 0 et 2,5 V.

FIG. 4.4 – Schéma électrique détaillé du système comportant deux voies d'entrées analogiques à numériser.

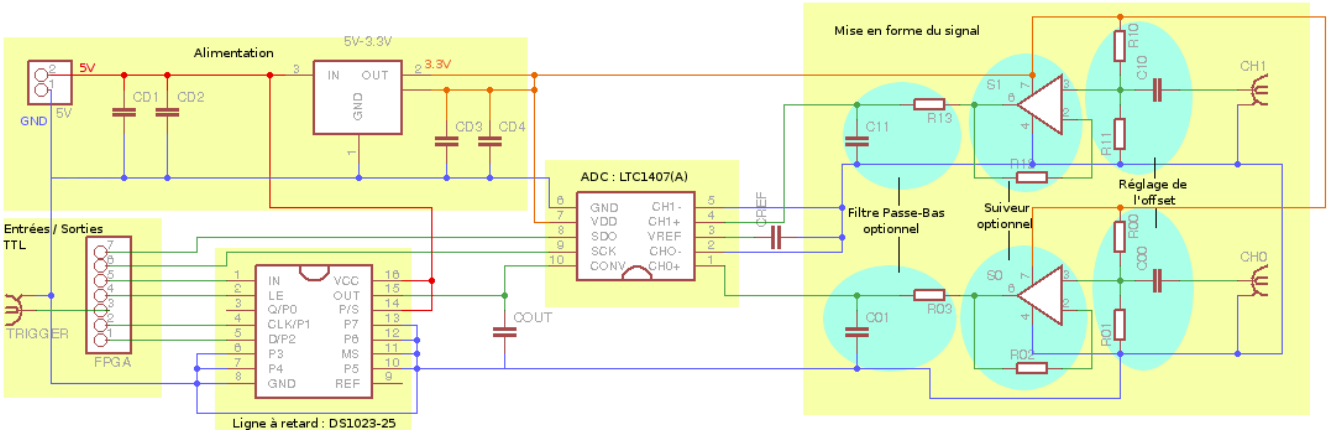


FIG. 4.5 – Liste des composants présentés dans le circuit de la figure 4.4

Alimentation	CD1	10 μF
Capacité	CD2	100 nF
de découplage	CD3	10 μF
	CD4	100 nF
Filtre Passe-Bas	CX1	50 pF
f_c=400MHz	RX3	50 Ω
Montage Suiveur	SX	AD8057
BP=325MHz et G=1	RX2	1 $k\Omega$
Réglage de l'offset	CX0	100nF
	RX0	10 $k\Omega$
	RX1	10 $k\Omega$
	CREF	10 μF
	COUT	optionnelle et variable (quelques pF)

4.3.2 Problèmes rencontrés et solutions

Capacités parasites

Les signaux TTL générés ayant des états logiques de durée de quelques dizaines de nanosecondes, les pistes de cuivre du typon se comportent alors comme des capacités parasites. Ceci engendre alors une perturbation des signaux. C'est pourquoi, lors de la conception du typon, ces problèmes doivent être pris en compte en réduisant la surface des pistes en regard. De plus, afin d'éviter les

perturbations entre pistes, il est préférable d'éloigner les pistes parcourues par un signal analogique d'un signal numérique.

Hautes-fréquences

Le système en question est conçu pour travailler à haute-fréquence (horloge de 50 MHz à 100 MHz, signal carré proche du GHz...). Le problème induit par ces hautes-fréquences est un rayonnement électromagnétique perturbant les signaux proches. Il est possible de réaliser un plan de masse afin de réduire ce phénomène de perturbation par rayonnement. En revanche, le plan de masse implique une augmentation des capacités parasites sur le typon. La conception de celui-ci résulte alors d'un compromis entre ces paramètres pour avoir le meilleur rapport signal sur bruit.

Adaptation d'impédance

Les hautes-fréquences impliquent également des problèmes d'adaptation d'impédance. Ces problèmes dépendent généralement des impédances internes aux composants utilisés. C'est certainement la raison de la présence de la capacité COUT sur le schéma électrique précédent. Suivant la conception du typon, cette capacité est plus ou moins importante (voir nulle pour la carte finale) mais lors de tests sur les premières cartes créées, sa valeur devait être d'au moins 10 pF pour obtenir un bon fonctionnement du système. La seule raison pour laquelle un condensateur a été placé à cet endroit, est que le système fonctionnait uniquement avec une sonde de mesure, ayant une capacité interne de 15 pF, placée sur cette piste lors des tests.

Chapitre 5

Applications et résultats

5.1 Acquisition de signaux déclenchés

5.1.1 Objectif

Cette application simple est une façon de tester le système réalisé. Elle consiste à numériser une sortie de générateur de fonction, déclenchée par un signal de trigger externe. Le résultat obtenu sera ensuite comparé avec l'affichage sur un oscilloscope afin de vérifier que les données obtenues sont correctes et assez précises pour mesurer les informations importantes. Cette application sert également à quantifier les limites du système, avec l'hypothèse de posséder un générateur de fonctions hautes-fréquences. Elle permet par exemple de vérifier la fréquence d'échantillonnage en pratique.

5.1.2 Présentation du dispositif

Le matériel mis en œuvre pour cette application, en plus du système créé :

- Générateur de fonction : Tektronik AFG320 dont l'entrée de déclenchement trigger externe est branchée sur la broche d'impulsion du système d'acquisition.
- Oscilloscope : LeCroy WaveRunner 6200 capable d'échantillonner à 10GS/s, reçoit également la sortie du générateur.

Un schéma de l'expérience est présenté dans la figure 5.1.

5.1.3 Résultats

Les résultats obtenus sont présentés dans les figures 5.2, 5.3, 5.4 et 5.5. La fonction rampe permet de détecter les erreurs de délais facilement car chaque erreur de temps est visible sur la valeur

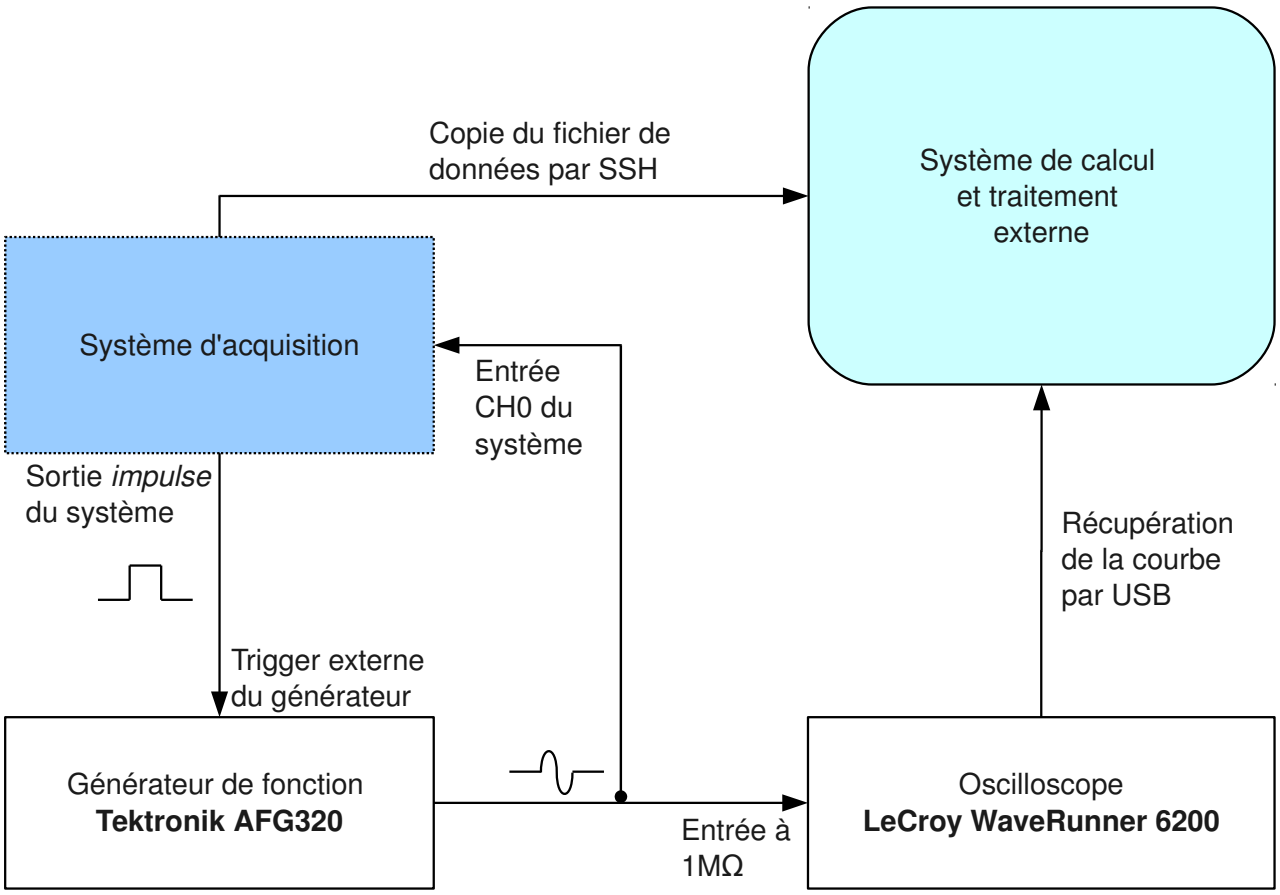


FIG. 5.1 – Schéma de manipulation pour l'acquisition de signaux déclenchés.

de tension et a été beaucoup utilisée lors de l'étape de résolution des problèmes du système. Grâce à ces données, nous pouvons calculer la fréquence d'échantillonnage pratique du système. Malgré le nombre important de point qui rend difficile la distinction du début et de la fin d'une période, il a été compté environ 40000 points par période. Ainsi la fréquence d'échantillonnage est de :

$$f_{ech} = f_{periode} * Nb_{points}$$
$$f_{ech} = 100kHz * 40000 = 4GHz$$

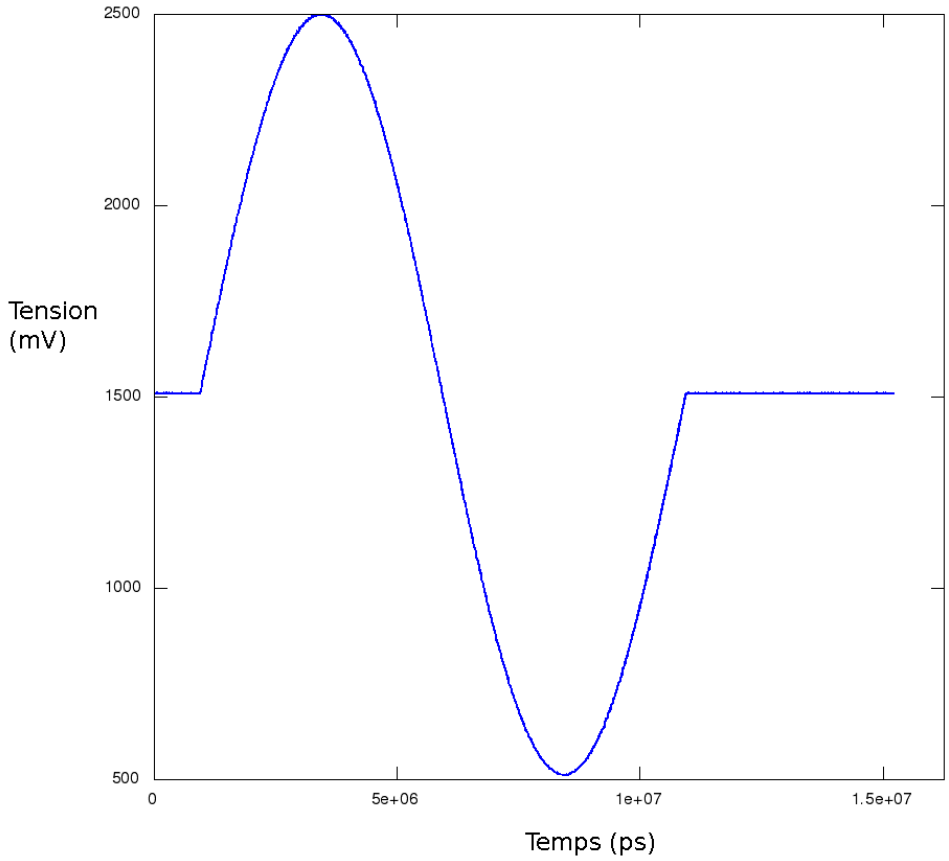


FIG. 5.2 – Sinusoïde de 100kHz mesurée avec 4GS/s par le système.

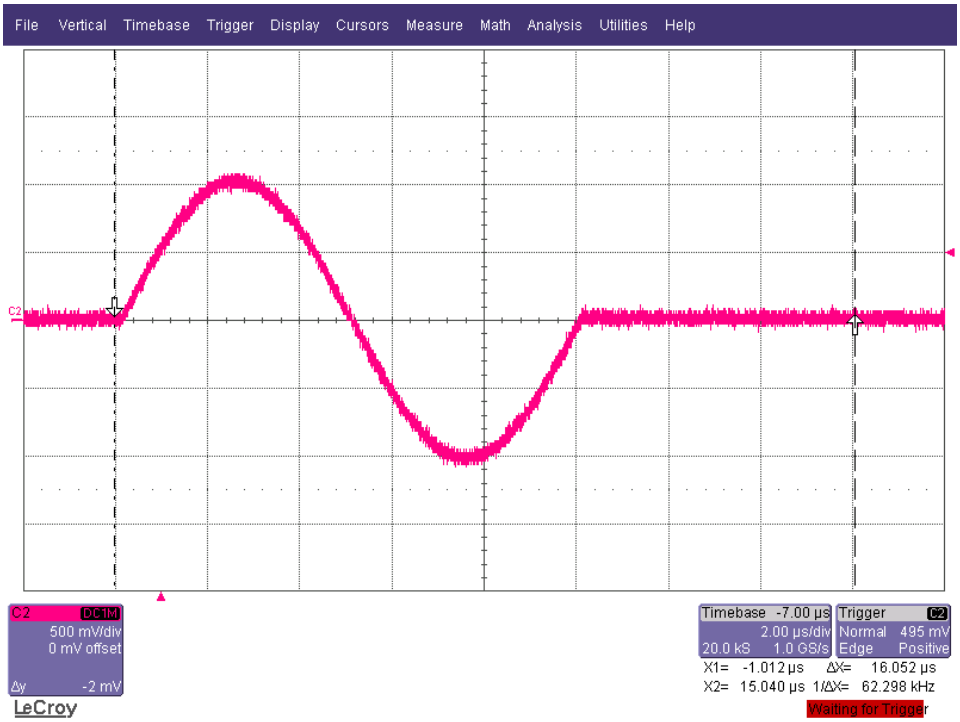


FIG. 5.3 – Sinusoïde de 100kHz mesurée à l'oscilloscope.

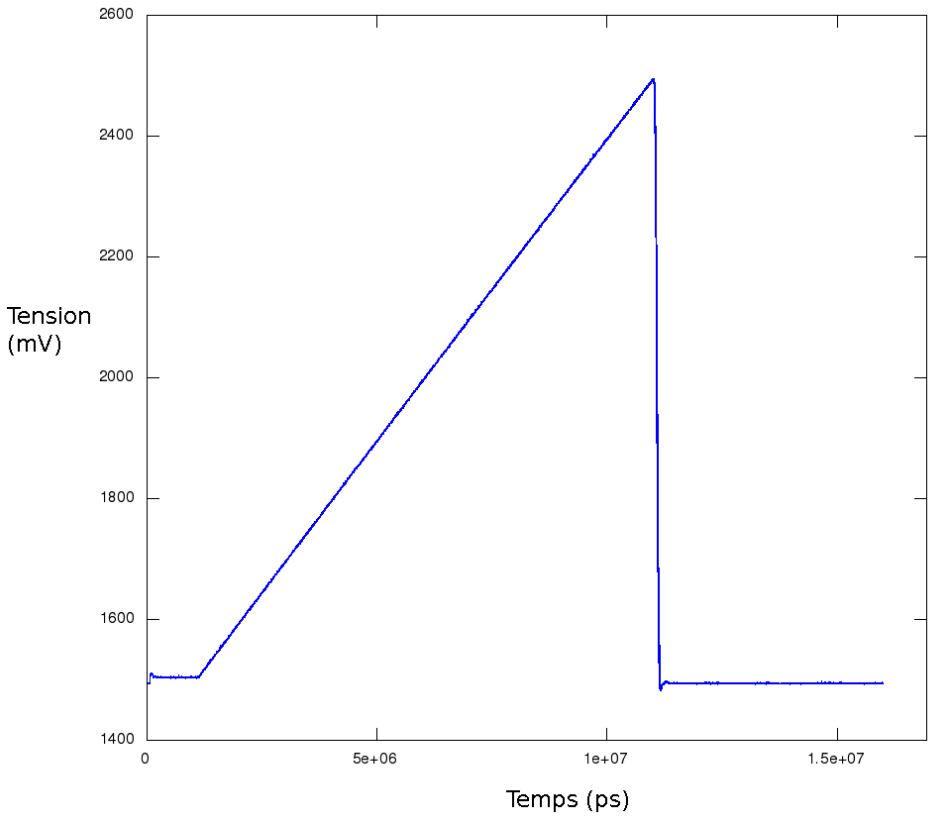


FIG. 5.4 – Rampe de 100kHz mesurée avec 4GS/s par le système.

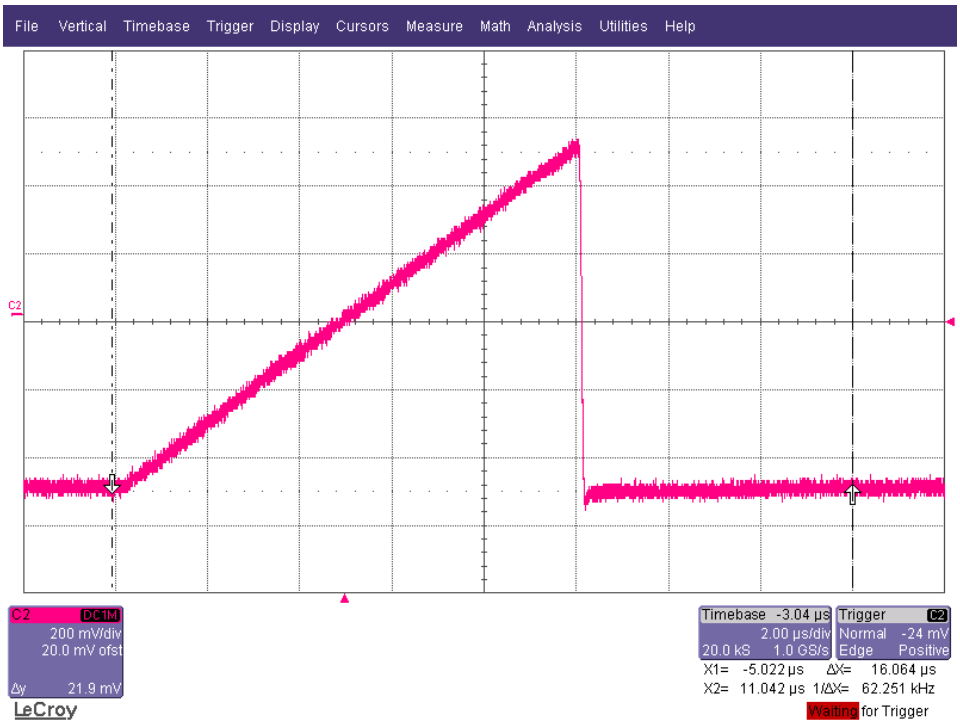


FIG. 5.5 – Rampe de 100kHz mesurée à l'oscilloscope.

5.2 Échographie et transducteur électro-acoustique

5.2.1 Objectif

L'application, pour laquelle était destinée ce projet, est l'acquisition via un transducteur, d'un signal échographique ultra-sonore. Ce signal peut servir à caractériser le collage de deux wafers par exemple. Dans notre cas, l'écho sera produit par une plaque métallique perturbant l'impédance acoustique du milieu aqueux de mesure. Bien entendu, pour garantir le bon fonctionnement de l'acquisition, le milieu de mesure ne doit pas varier pendant l'acquisition. Le temps de l'acquisition varie suivant le nombre de points mais est généralement compris entre quelques centaines de ms à quelques secondes (pour les grandes fenêtres de visualisation demandant beaucoup de points).

5.2.2 Présentation du dispositif

Le dispositif, présenté dans la figure 5.7, utilise un générateur d'impulsions acoustiques externe (Panametrics - Sofranel Model 5800, *Computer controlled pulser/receiver*). En plus de générer les impulsions, cet appareil possède un étage de mise en forme de l'écho (amplificateur) afin d'obtenir un signal exploitable. Les paramètres utilisés lors de cette expérience pour le configurer sont donnés dans la figure 5.6. Certains de ces paramètres (Atténuation, Gain...) ont été trouvé expérimentalement afin d'obtenir un écho exploitable.

Fonction	Nom du paramètre	Valeur
Mode de fonctionnement	Mode	echo
Mode de déclenchement	PRE	ext
Puissance de sortie	Puiss	$12,5\ \mu J$
Amortissement	Amort	$100\ \Omega$
Fréquence de coupure basse	FiltPH	1 MHz
Fréquence de coupure haute	FiltPB	35 MHz
Atténuation d'entrée	Atten entr	30 dB
Atténuation de sortie	Atten sort	0 dB
Gain appliqué à l'écho	Gain	40 dB

FIG. 5.6 – Paramètres de configuration du générateur d'impulsions Panametrics - Sofranel

5.2.3 Résultats

Les résultats obtenus sont présentés dans les figures 5.8, 5.9, 5.10 et 5.11.

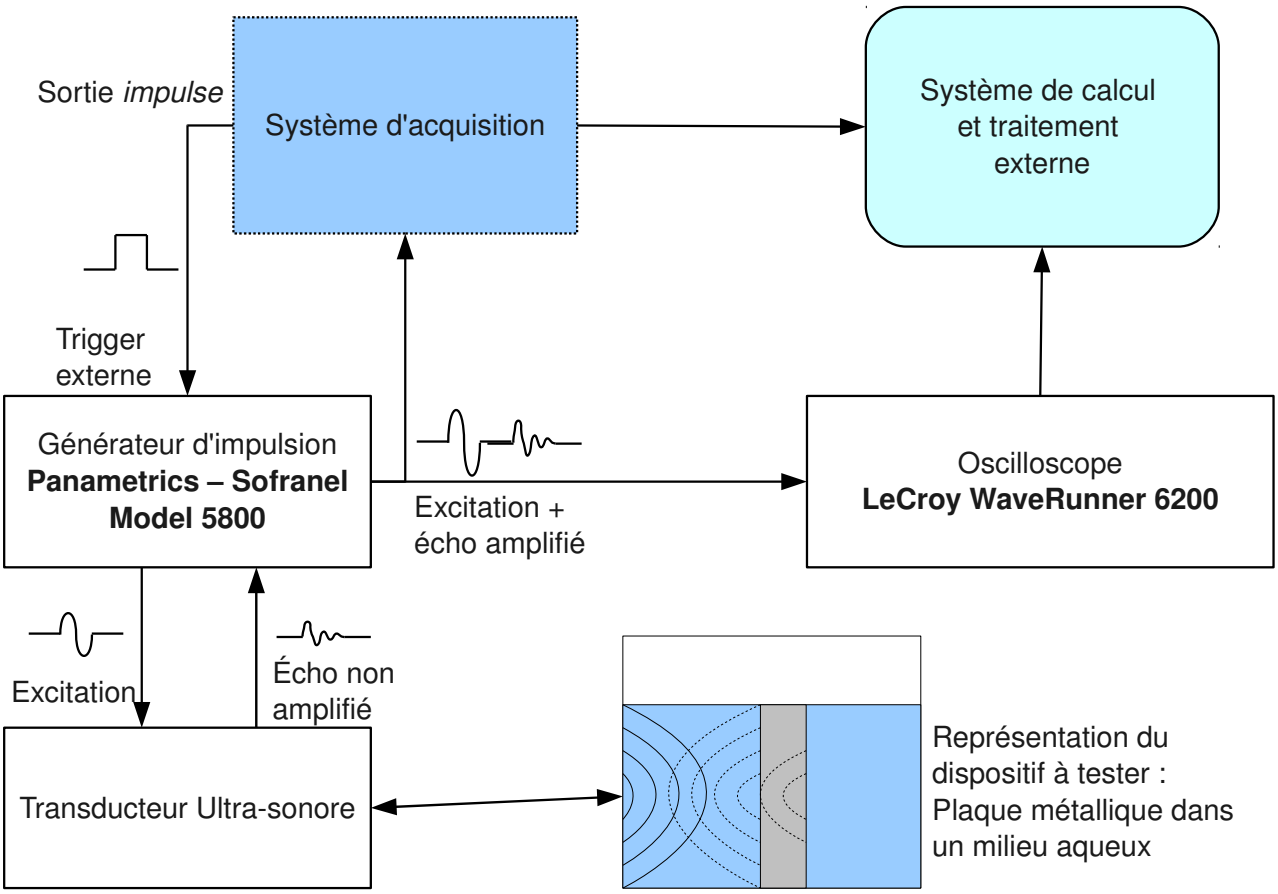


FIG. 5.7 – Schéma du dispositif échographique.

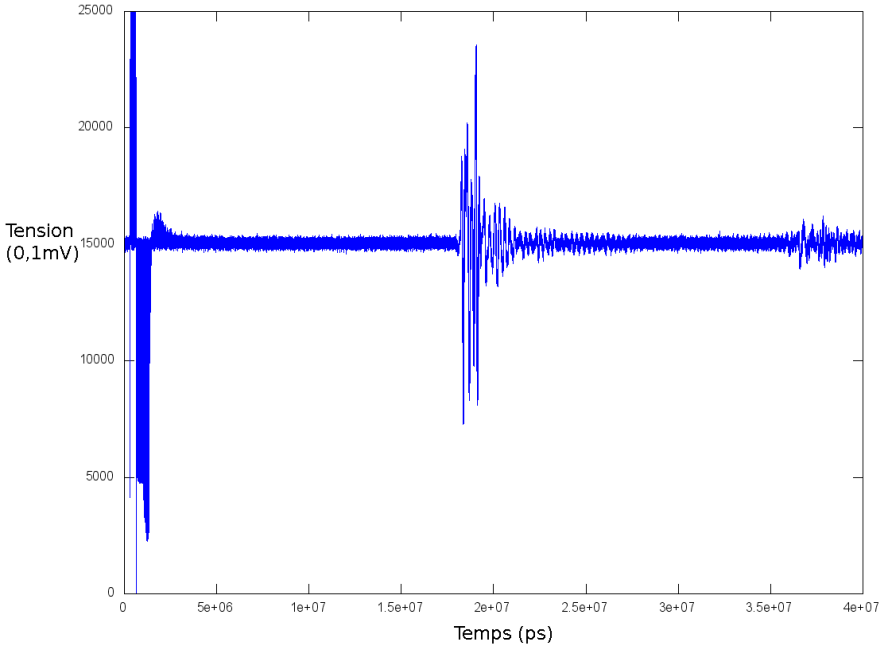


FIG. 5.8 – Stimulation et écho récupéré par le système sur une fenêtre de 0 à 40μ s.

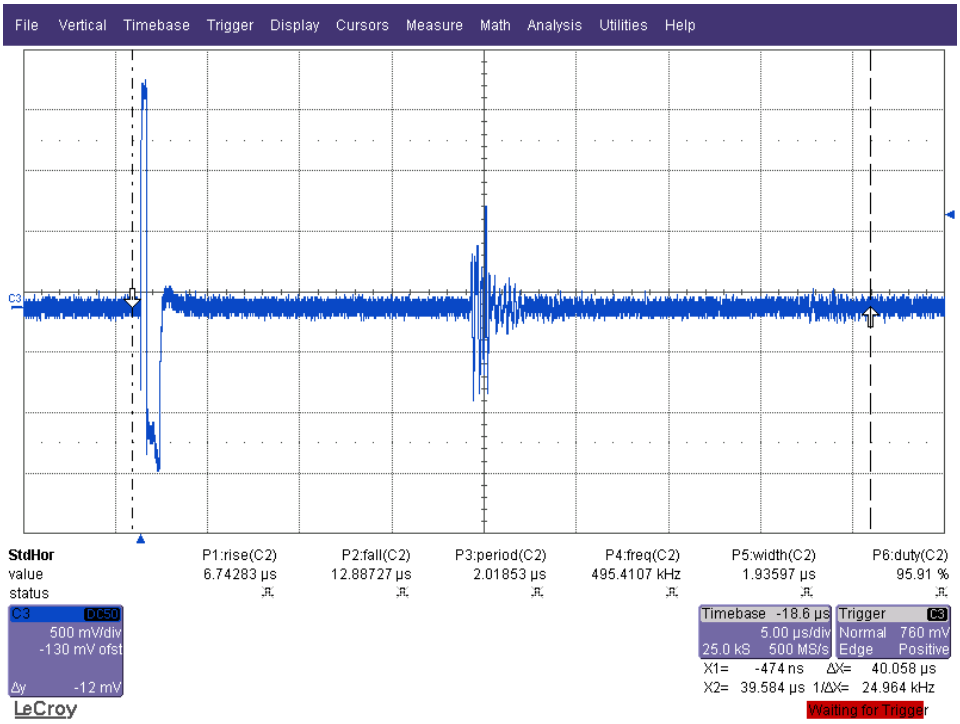


FIG. 5.9 – Stimulation et écho récupéré à l'oscilloscope.

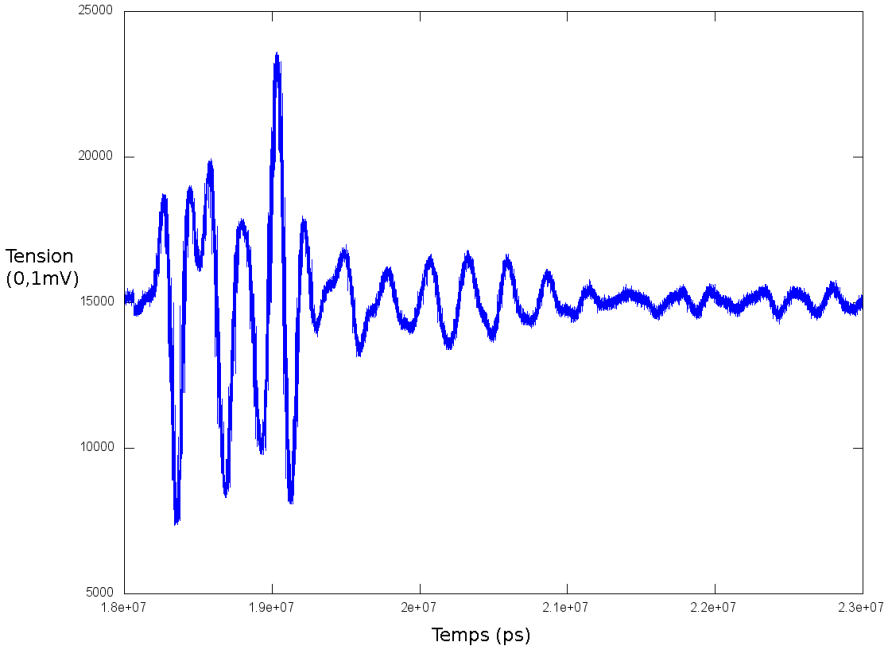


FIG. 5.10 – Écho seul récupérés par le système sur une fenêtre de 18 à 23μ s.

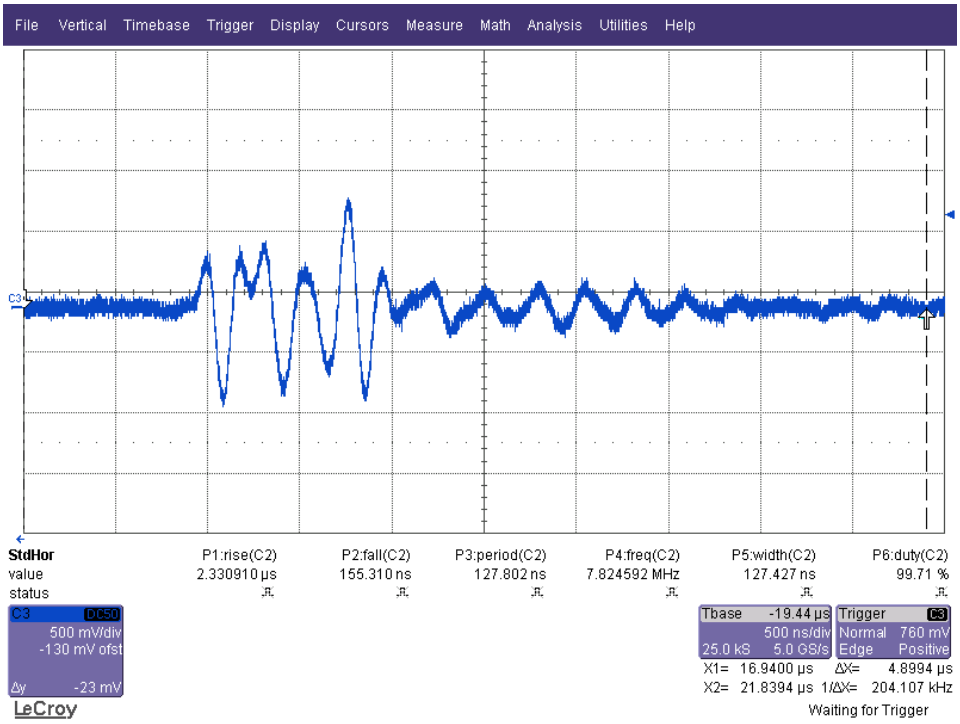


FIG. 5.11 – Écho récupérés à l'oscilloscope.

5.3 Capteurs acoustiques radiofréquences

5.3.1 Présentation des capteurs

Ces capteurs acoustiques, et plus particulièrement les capteurs à ondes de surfaces (SAW), sont une spécialité de l'institut Femto-ST et ont fait l'objet de plusieurs travaux de recherche[9, 10, 11]. Ce sont des dispositifs passifs permettant la mesure de grandeurs physiques telles que la température, la pression ou encore le couple. Ils ont la particularité de ne demander que l'énergie d'une interrogation sans fil (signal radiofréquence) afin de délivrer l'information. En effet, le principe de fonctionnement donnée par la figure 5.12[1] montre un substrat piézoélectrique sollicité par un signal radiofréquence incident, qui est converti en onde mécanique dont les propriétés varient avec l'environnement du capteur, pour ensuite être reconvertie en signal électrique à destination du circuit électronique d'interrogation.

Une autre méthode de test de la surface de travail, présenté par la figure 5.13, est d'avoir seulement une série de peignes interdigités puis de l'autre côté de la surface, un ou plusieurs réflecteurs (créés à partir d'un dépôt différent sur le substrat piézoélectrique). Ainsi, la différence d'impédance acoustique produit des échos qui traverseront à nouveau la surface de travail, et qui produiront une onde électrique RF grâce aux peignes initiaux.

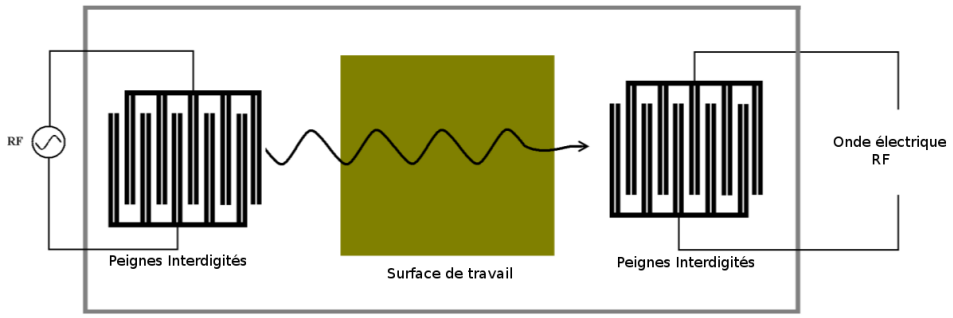


FIG. 5.12 – Schéma de présentation du fonctionnement d'un capteur SAW.

5.3.2 Objectif

Le but de cette application est de mettre en pratique le système présenté, à l'aide d'une électronique de conditionnement du signal, pour visualiser avec suffisamment de précision la réponse impulsionnelle de capteurs. Une première manipulation consistera à récupérer les données d'une ligne à retard Kongsberg, excitée par un signal d'environ 800MHz, afin de compléter une manipulation

[1]Schéma extrait du rapport de stage de D. Rabus intitulé "Mesures de sensibilité gravimétrique de structures HBAR".

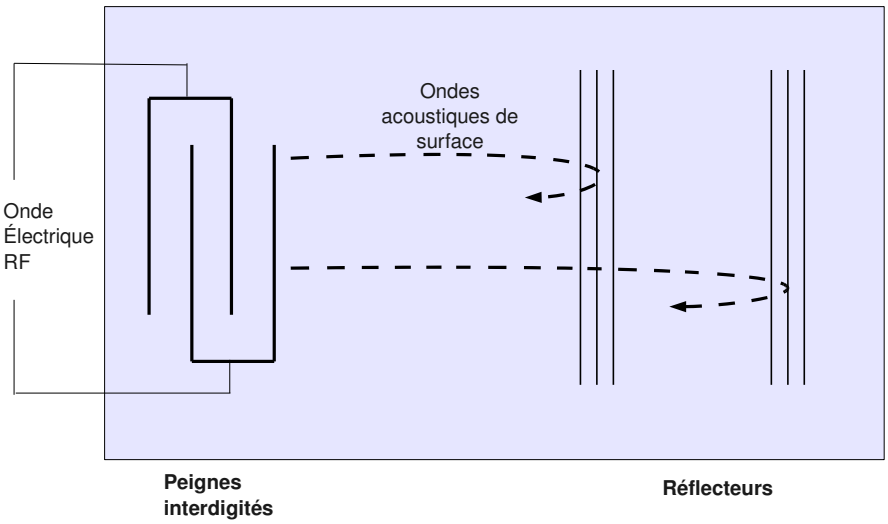


FIG. 5.13 – Schéma de présentation du fonctionnement d'un capteur SAW avec deux réflecteurs.

expérimentale de validation d'une étude sur le retournement temporel itératif appliqué aux lignes à retard [2]. La seconde expérience consiste à tester la réponse impulsionnelle d'une ligne à retard, conçue à l'institut Femto-ST, optimisée à 100 MHz, en générant le signal RF à partir du système présenté.

5.3.3 Présentation du dispositif de test de la ligne à retard Kongsberg

Dans le cas de cette expérience, une autre plateforme, combinant un CPU et un FPGA et spécialisée pour l'implémentation de dispositifs radiofréquences, est mise en œuvre (Ettus Research - USRP, Universal Software Radio Peripheral[3]). Cette plateforme, comme indiquée sur la figure 5.14, fournit le signal RF d'excitation de la ligne Kongsberg (864 MHz), l'horloge de cadencement du FPGA (64 MHz), et le signal de trigger de déclenchement du signal de stimulation(5 kHz). Pour ces raisons, le programme du FPGA contient quelques modifications permettant de choisir si l'horloge de cadencement et le trigger de déclenchement sont internes ou externes. De plus, le programme exécuté sur le microprocesseur demande en argument la fréquence d'horloge de cadencement du FPGA, afin de calculer les retards à effectuer par le FPGA et la ligne à retard Maxim.

[2] Étude en cours réalisée par T. Rétornaz à l'institut Femto-ST, Département Temps-Fréquence
[3] Information sur la plateforme disponible à l'adresse : http://www.ettus.com

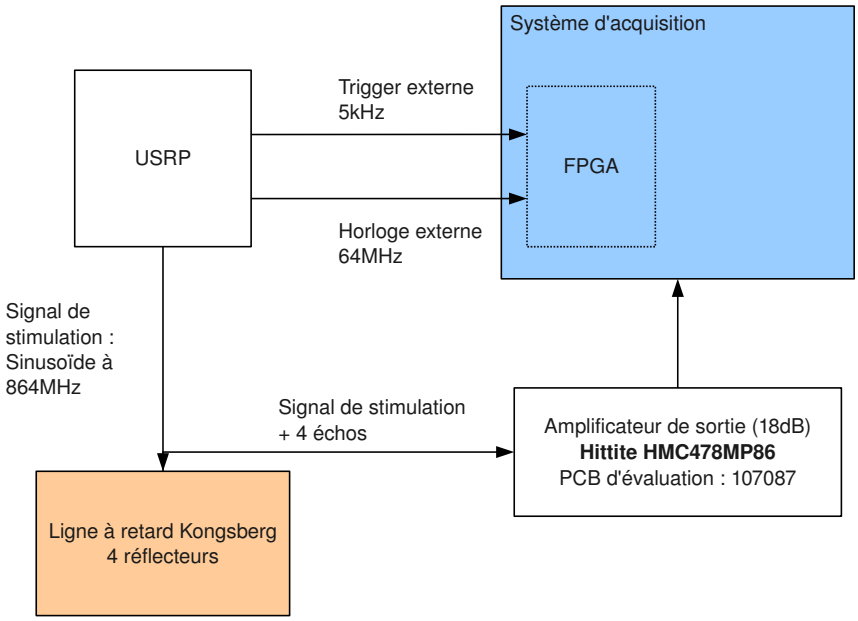


FIG. 5.14 – Schéma du dispositif de test de la ligne à retard Kongsberg.

5.3.4 Résultats obtenus avec la ligne Kongsberg

La figure 5.15 présente les échos obtenus par la ligne Kongsberg visualisés à l'oscilloscope dans le cadre d'une étude sur les résonateurs à ondes de surface utilisés comme capteurs passifs enfouis[10] présentée lors de "The First International Conference on Sensor Device Technologies and Applications" en juillet 2010. La numérisation de ces échos, sur une plus petite fenêtre de visualisation, avec le système à 4 GS/s est présentée par la figure 5.16.

Il a également été réalisé une expérience sur presque 96h avec des prises de mesure toutes les 5 minutes (déclenchement des mesures réalisé par un script shell sur la carte ARMadeus) et des mesures à différentes températures afin de vérifier la possibilité d'utiliser cette ligne à retard comme capteur de température. Les résultats obtenus ne sont pas exploitables sans traitement complexe sur la phase et la magnitude des signaux. De plus, le signal d'excitation étant de plus de 800 MHz et la fréquence d'échantillonnage de 4 GS/s, les résultats deviennent difficiles à exploiter dans le domaine temporel. La possibilité d'augmenter la fréquence d'échantillonnage est une solution à ce problème. Pour cela, l'utilisation d'une seconde ligne à retard Maxim (DS1020-15) et un algorithme différent de distribution des retards aux travers de ces deux lignes permettent de passer à 20 GS/s. Cette solution est en préparation à l'écriture de ces lignes.

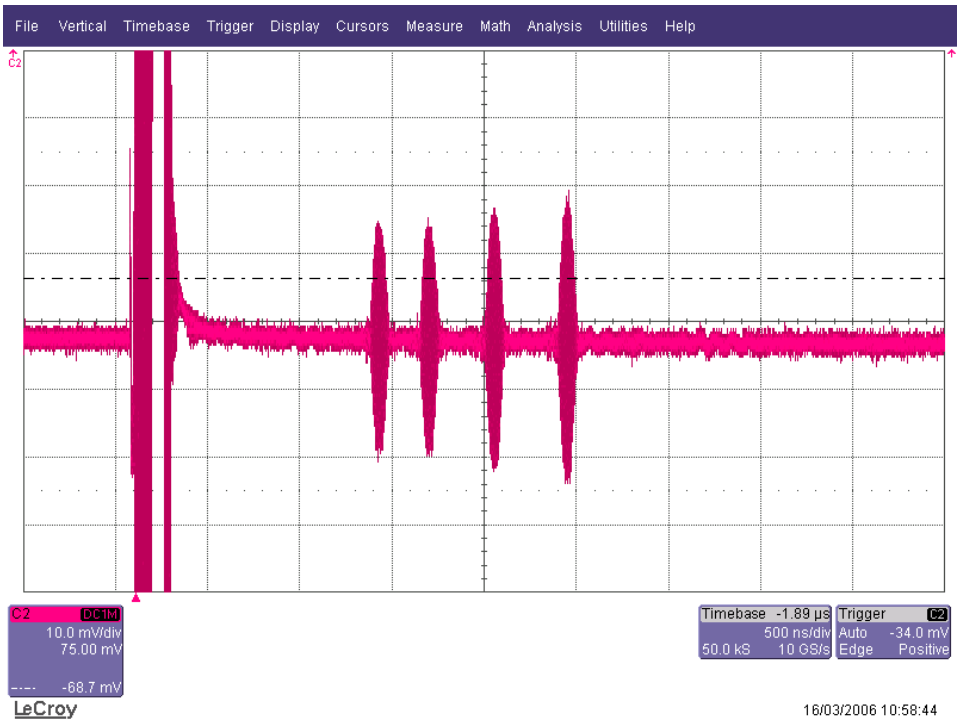


FIG. 5.15 – Visualisation du signal de stimulation et des 4 échos d'une ligne à retard Kongsberg à l'oscilloscope.

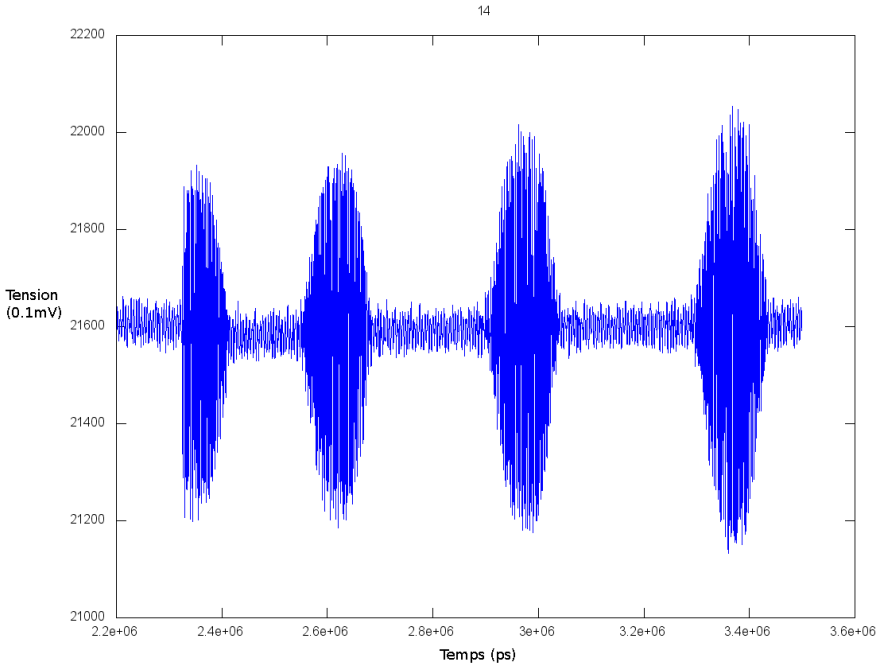


FIG. 5.16 – Visualisation des 4 échos d'une ligne à retard Kongsberg avec le système à 4GS/s.

5.3.5 Présentation du dispositif de test de la ligne à retard à 100MHz

Pour cette expérience, présentée par la figure 5.17, le signal de stimulation à 100MHz est généré par le FPGA et amplifié. De plus, afin ne pas endommager les entrées d'acquisitions, un switch, commandé par le FPGA, alternera la position de stimulation de la ligne et la réception des échos. Enfin, pour une meilleure visualisation des échos, ceux-ci sont également amplifiés en sortie du switch.

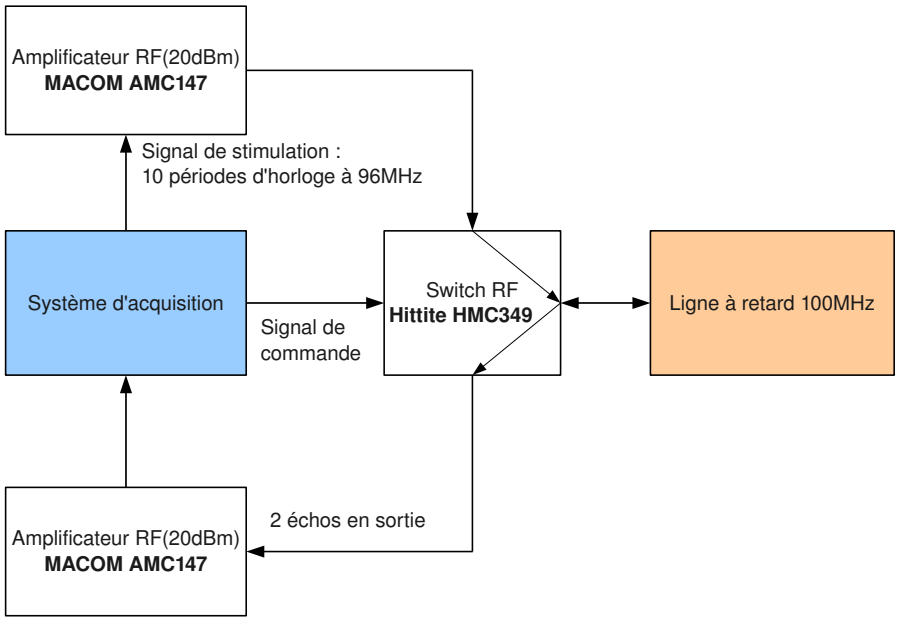


FIG. 5.17 – Schéma de fonctionnement du dispositif de test de la ligne à retard à 100MHz.

5.3.6 Résultats obtenus avec la ligne à 100MHz

La figure 5.18 est le résultat obtenu grâce au dispositif précédent. On peut remarquer la présence d'une impulsion qui est due à la non évacuation des charges de la fin du signal de stimulation, avant la permutation du switch. Les échos sont distincts et exploitables ce qui indique que le signal de stimulation est correct. On peut également conclure sur le fait que le système créé, après quelques modifications, peut également servir à la numérisation de réponses impulsionnelles et d'échos de capteurs à ondes de surfaces.

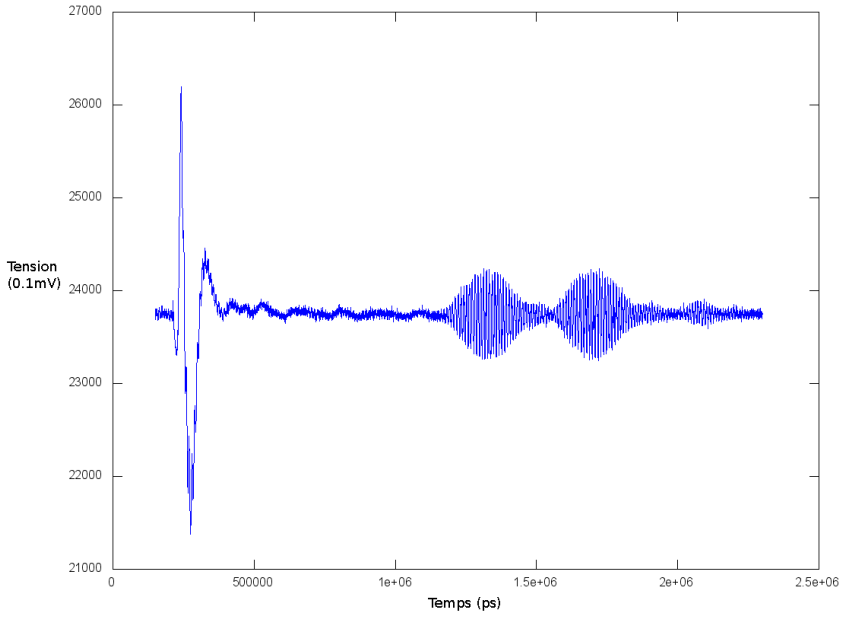


FIG. 5.18 – Visualisation des deux échos de la ligne à 100MHz.

5.4 Radar à pénétration de sol ou GPR

5.4.1 Objectif

Le fonctionnement d'un GPR est semblable à tout autre radar. Des ondes électromagnétiques sont envoyées dans le sol au travers d'une antenne. Lorsque ces ondes rencontrent des changements de milieux, une partie est renvoyée vers la surface et l'écho est récupéré par une antenne avant d'être numérisé. La profondeur et la résolution de sondage du sol dépendra, en plus de la composition de celui-ci, de la fréquence et de l'amplitude du signal envoyé.

Pour cette application, l'objectif est de réaliser un système embarqué capable de générer une impulsion de plusieurs dizaines de Volts dont la fréquence est proche de 100 MHz. À part pour l'amplification des signaux, le système d'acquisition n'a pas besoin de subir de modification.

5.4.2 Fonctionnement de l'émetteur radar

Généralement, les impulsions émises par un GPR sont relativement courtes (rapport cyclique faible) et proviennent du déchargement rapide d'un condensateur chargé au travers d'un transistor à avalanche. La figure 5.19 donne le schéma électrique du circuit de génération des impulsions. Lorsqu'un signal est envoyé sur la base du transistor à avalanche, celui-ci voit sa conductivité augmentée (rétroaction positive de la conductivité) avec le courant qui y circule et donc est capable

de vider rapidement le contenu de C4 avec une constante de temps déterminée par l'impédance de TR1. Une fois C4 vide, au travers de la résistance R9, C4 se recharge lentement lorsque le transistor est bloqué. Le signal de déclenchement des impulsions est fourni par le système d'acquisition (sortie *impulse* des schémas précédents). La sortie est dirigée directement sur une antenne. Le tableau de la figure 5.20 indique le nom et les valeurs des composants utilisés.

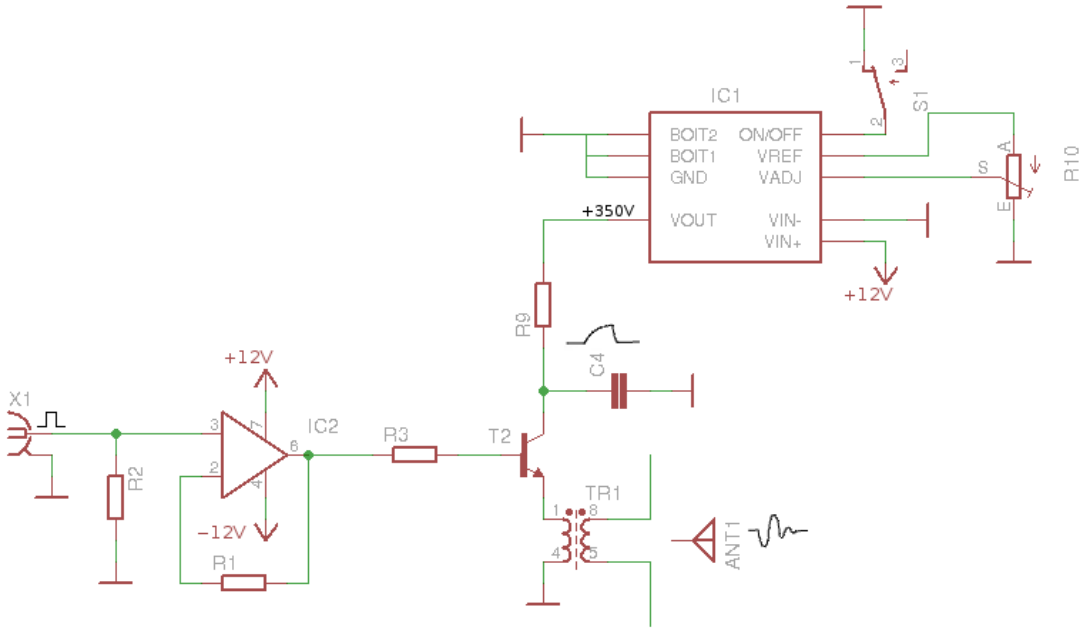


FIG. 5.19 – Schéma électrique du circuit de génération des impulsions du GPR.

FIG. 5.20 – Liste des composants présentés dans le circuit de la figure 5.19

IC1	Convertisseur DC/DC 12/500V - Traco Power	MHV12
IC2	Amplificateur Vidéo - Analog Devices	AD811
T2	Transistor à Avalanche - Zetex	FMMT417
TR1	Transformateur RF	SMC TW-205
R1	Résistance de rétroaction	750 Ω
R2,R3	Résistances	50 Ω
R9	Résistance de puissance	100 $k\Omega$
R10	Potentiomètre	5 $k\Omega$
C4	Condensateur RF	110 pF
S1	Switch manuel	
X1	Connecteur SMA	
Ant1	Antenne	2 fils émaillés de 0,65 mm de diamètre

Le montage suiveur sert à protéger la sortie du FPGA d'éventuels appels ou retours de courant et la valeur résistance de rétroaction est définie pour obtenir un gain de 1 avec l'amplificateur. Pour pouvoir atteindre le régime avalanche, il a été observé que la différence de potentiel, sur ce montage, entre le collecteur et la masse doit être de plus de 280 V (la documentation donne la valeur

de 360 V, en pratique les meilleurs résultats sont obtenus pour 354 V). Cette tension est obtenue grâce au convertisseur DC/DC pouvant délivrer jusqu'à 500 V avec 6 mA et qui est réglable grâce au potentiomètre R10. Une fois le condensateur chargé, un signal supérieur à 0,9 V sur la base du transistor déclenche l'effet avalanche permettant d'atteindre jusqu'à 60 V pic à pic[4] sur le secondaire du transformateur. Ce transformateur est relié à deux fils dont la longueur influence la fréquence de l'onde électromagnétique générée.

5.4.3 Résultats

Pour réaliser ces résultats, la carte de génération des impulsions est pilotée par une générateur de signaux basse fréquence (Tektronic - AFG320) délivrant des créneaux à la fréquence de 10 kHz. Une seconde antenne est reliée à une entrée de l'oscilloscope (LeCroy WaveRunner 6200) par deux amplificateurs (MACOM AMC147 et Hittite HMC478MP86) pour un gain de presque 40 dB. Le schéma de fonctionnement de cette expérience est présentée par la figure 5.21. Le capteur à ondes de surfaces à 100 MHz est une substitution à un terrain à sonder. Celui-ci renvoi également des échos après avoir reçu une impulsion à 100MHz.

La figure 5.22 présente une impulsion en sortie du transformateur d'environ 50V d'amplitude et, après quelques oscillations, de fréquence proche de 100MHz. Cette fréquence de l'impulsion, bien que présentant de nombreuses harmoniques, est mise en avant grâce au spectre d'émission de la figure 5.23. Malgré le bruit important dû aux émissions radiofréquences proches (station de radio de porteuse 102.4MHz émise proche du lieu de l'étude), il est possible de récupérer l'impulsion et les échos du capteur SAW à l'oscilloscope comme montré en figure 5.24 et 5.25. L'étape suivante est de connecter cette électronique avec le système précédent et de récupérer les échos directement sur le système embarqué. Cette application est en cours de préparation au moment de la rédaction de ce rapport.

[4]Valeur maximum observée avec ce montage lors de la manipulation

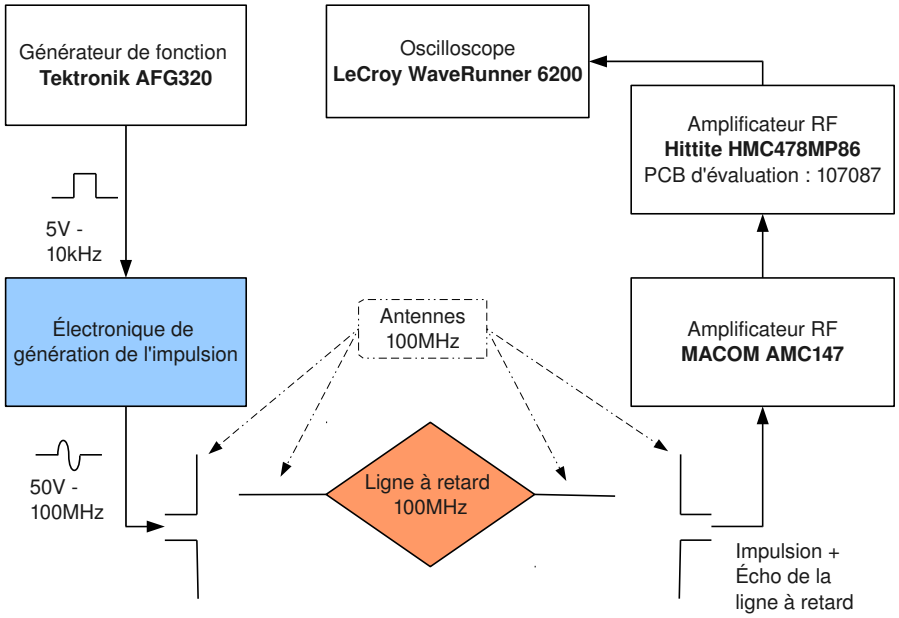


FIG. 5.21 – Schéma de fonctionnement de la manipulation avec l'électronique de génération d'une impulsion pour GPR.

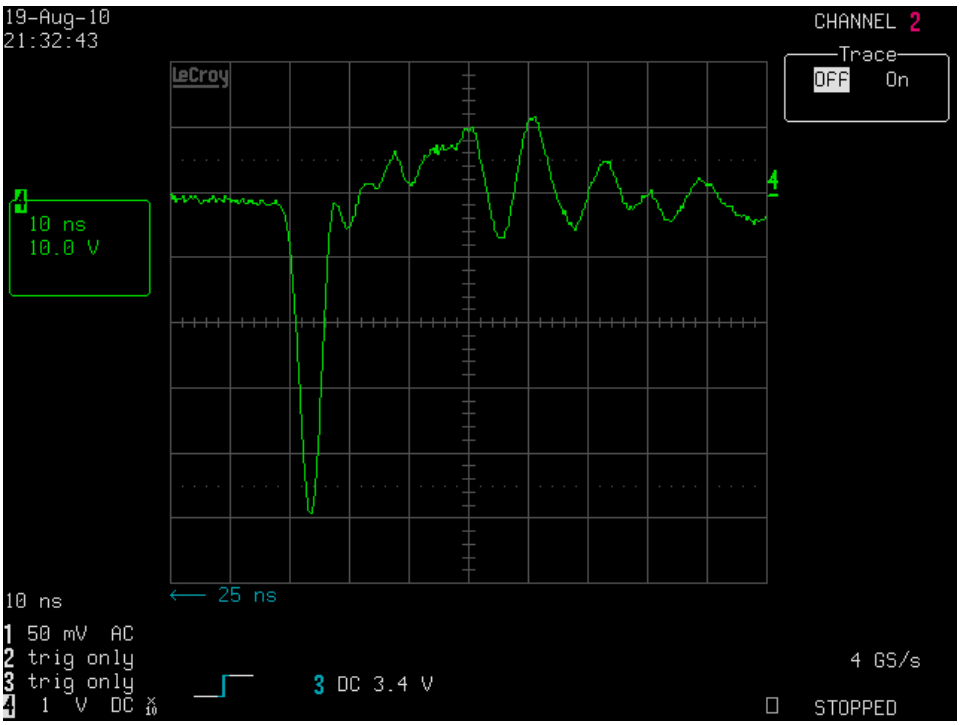


FIG. 5.22 – Impulsion générée visualisée au secondaire du transformateur.

FIG. 5.23 – Spectre d'émission du générateur d'impulsion.

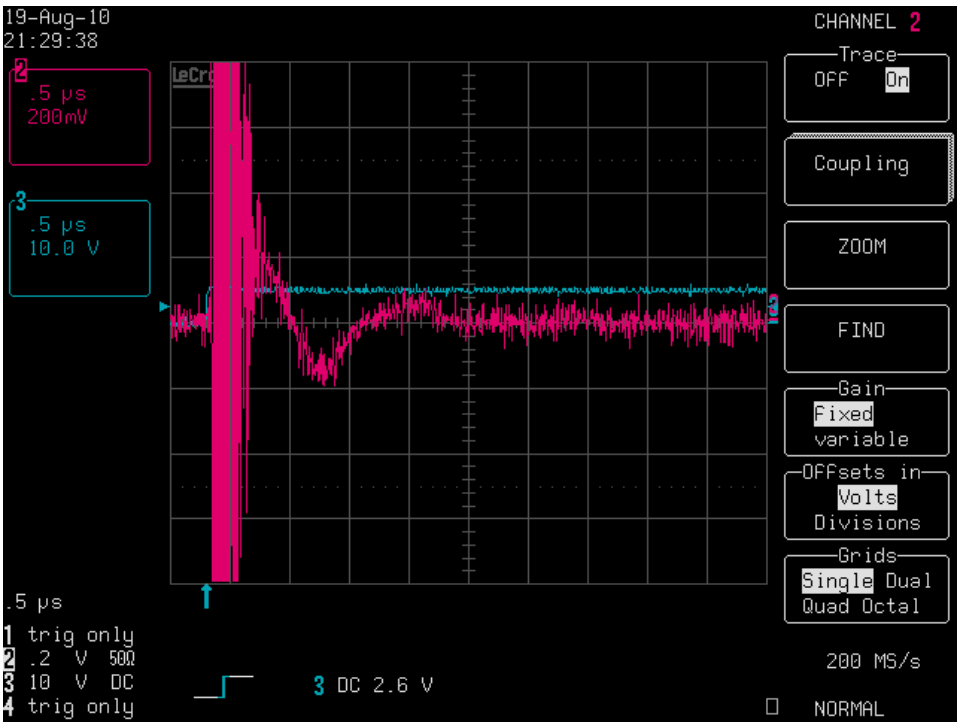


FIG. 5.24 – Réception de l'onde sans le capteur SAW à l'oscilloscope.

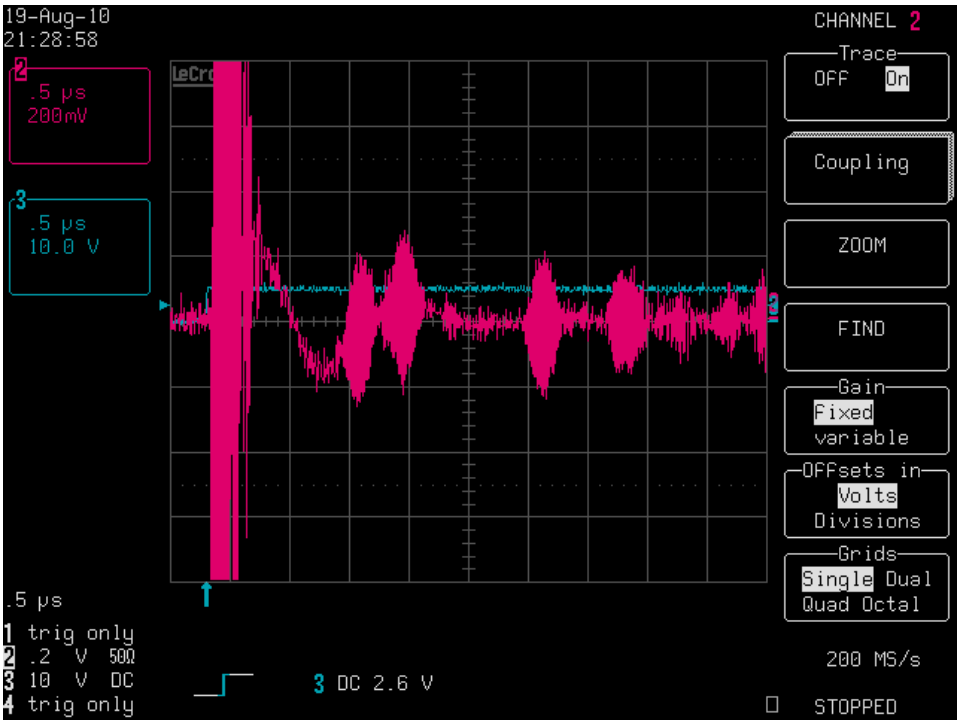


FIG. 5.25 – Réception de l'onde et des échos lors d'une excitation du capteur SAW par une antenne.

Chapitre 6

Conclusion et suites envisageables

Aux vues des résultats, le système créé lors de ce projet est performant et suffisamment polyvalent pour pouvoir toucher à d'autres domaines que les ondes ultra-sonores avec un minimum de modifications. La technique d'acquisition limitant les applications possibles, le système final a été retouché afin de pouvoir utiliser un trigger (si celui-ci ne dépasse pas les 25 kHz) et une horloge externe. Le système est toujours en modification avec la possibilité d'amener la fréquence d'échantillonnage à 20GHz grâce à l'intégration d'une ligne à retard DS1020-15 et à la combinaison avec la ligne déjà présente, ceci afin de récupérer les données de la lignes à retard Kongsberg résonnante aux alentours de 800MHz.

Ce projet a été largement facilité grâce à la plateforme utilisée combinant un microprocesseur et un FPGA. L'avantage de dissocier les applications lentes et rapides, ainsi que le bus de communication rapide entre ces éléments, a permis d'augmenter le nombre de solutions au problème initial. De plus, la possibilité de reprogrammer le FPGA depuis le microprocesseur est utile pour réduire le temps de développement du système et d'augmenter les applications réalisables.

L'autre point fort de cette plateforme est d'avoir des outils libres. Ainsi l'utilisation du SPI depuis le microprocesseur a été simplifiée grâce à des programmes libres déjà existant. De plus, il a fallu compiler le noyau du système d'exploitation avec les fichiers modifiés, ce qui n'est en général pas possible avec un OS propriétaire.

Le système pourrait également être amélioré par l'utilisation du second mode de fonctionnement des ADC. Ceux-ci fonctionnant en continu à 3 MS/s, il serait utile de pouvoir effectuer également une acquisition temps réel avec le système. Une autre amélioration consisterait à exploiter les modes veilles du composant afin de réduire la consommation du système. Ainsi, après l'ajout d'une communication bluetooth, par exemple, et/ou d'un terminal portable pour récupérer les données, le système pourrait être entièrement embarqué et être fonctionnel pour l'application du GPR.

Table des figures

Bibliographie

[1] David A. Mindell. *Digital Apollo : Human and Machine in Spaceflight*. The MIT Press, 2008.

[2] Jing Yang, Thura Naing, and Bob Brodersen. A 1-GS/s 6-bit 6.7-mW ADC in 65-nm CMOS. In *IEEE Custom Integrated Circuits Conference*, pages 287–290, 2009.

[3] Ali Nazemi, Carl Grace, Lanny Lewyn, Bilal Kobeissy, Oscar Agazzi, Paul Voois, Cindra Abidin, George Eaton, Mahyar Kargar, Cesar Marquez, Sumant Ramprasad, Federico Bollo, Vladimir A. Posse, Stephen Wang, and Georgios Asmanis. A 10.3GS/s 6bit (5.1 ENOB at Nyquist) Time-Interleaved/Pipelined ADC Using Open-Loop Amplifiers and Digital Calibration in 90nm CMOS. In *IEEE Symposium on VLSI Circuits*, pages 18–19, 2008.

[4] Walt Kester. *Aperture Time, Aperture Jitter, Aperture Delay Time — Removing the Confusion*. Analog Devices, 2009.

[5] Linear Technology. *LTC1407/LTC1407A - Serial 12-Bit/14-Bit, 3Msps Simultaneous Sampling ADCs with Shutdown*. Datasheet : LT 0109 REV B.

[6] Maxim anciennement Dallas Semiconductor. *Design Considerations for All-Silicon Delay Lines*, Février 2002. Application Note 14. Mots-clé : DS1100, DS1135, DS1110, delay line, hybrid delay line, digital delay line.

[7] Maxim. *How Delay Lines Work*, Août 2002. Application Note 209. Mots-clé :delay lines, cmos delay line, DS1100, DS1135, DS1065, DS1110, DS1073.

[8] Maxim. *DS1023 - 8-Bit Programmable Timing Element*. Datasheet.

[9] S. Ballandras G. Martin E. Carry et V. Blondeau-Patissier D. Rabus, J.-M. Friedt. A high sensitivity open loop electronics for gravimetric acoustic wave-based sensors. EFTF, Avril 2010.

[10] G. Martin T. Laroche S. Alzuaga J.-P. Simonnet-E. Carry S. Ballandras J.-M Friedt, T. Rétornaz. Surface Acoustic Wave Resonators as Passive Buried Sensors. The First International Conference on Sensor Device Technologies and Applications – SENSORDEVICES, July 2010.

[11] S. Ballandras J.-M Friedt C. Droit, G. Martin. A frequency modulated wireless interrogation system exploiting narrowband acoustic resonator for remote physical quantity measurement. *Rev. Sci Instrum.*, Vol 81, Mars 2010.

Annexe A

Fichiers utiles à POD

A.1 wb16.xml

```
<?xml version="1.0" encoding="utf-8"?>
<component name="multi_sampl" version="1.0">
  <description>
    Controle FPGA reception echo
  </description>

  <generics>
        <generic name="id" public="true" value="1" match="\d+" type="natural" destination="both" />
  </generics>

  <hdl_files>
    <hdl_file filename="multi_sampl.vhd" scope="both" istop="1" />
    <hdl_file filename="gene_impulse.vhd" scope="both" />
    <hdl_file filename="delay_sclk.vhd" scope="both" />
    <hdl_file filename="diviseur.vhd" scope="both" />
    <hdl_file filename="registre.vhd" scope="both" />
    <hdl_file filename="wishbone_interface.vhd" scope="both" />
  </hdl_files>

  <interrupts>
  <interrupt interface="irq" port="irq_valid" />
  </interrupts>

    <interfaces>
    <interface name="irq" class="gls">
      <ports>
        <port name="irq_valid" type="EXPORT" size="1" dir="out" />
      </ports>
    </interface>


    <interface name="inpout" class="gls" >
        <ports>
        <port name="start" type="EXPORT" size="1" dir="in" />
        <port name="ack" type="EXPORT" size="1" dir="in" />
        <port name="SDObus" type="EXPORT" size="8" dir="in" />
        <port name="clkext" type="EXPORT" size="1" dir="in" />
        <port name="switch" type="EXPORT" size="1" dir="out" />
        <port name="impulse" type="EXPORT" size="1" dir="out" />
        <port name="impulse_clk" type="EXPORT" size="1" dir="out" />
        <port name="impulseX" type="EXPORT" size="1" dir="out" />
        <port name="clkout" type="EXPORT" size="1" dir="out" />
        <port name="SclkDL" type="EXPORT" size="1" dir="out" />
      </ports>
    </interface>

    <interface name="candr" class="clk_rst">
      <ports>
        <port name="gls_reset" type="RST" size="1" dir="in" />
        <port name="gls_clk" type="CLK" size="1" dir="in" />
      </ports>
    </interface>

    <interface clockandreset="candr" name="swb16" class="slave" bus="wishbone16" >
      <registers>
        <register name="ADD_ID" offset="0x00" size="16" rows="1" />
        <register name="RDELAYX" offset="0x01" size="16" rows="1" />
        <register name="RHIGH" offset="0x10" size="16" rows="1" />
        <register name="RDATA" offset="0x11" size="16" rows="1" />
      </registers>
        <ports>
```

```
        <port name="wbs_add" type="ADR" size="10" dir="in" />
        <port name="wbs_readdata" type="DAT_O" size="16" dir="out" />
        <port name="wbs_writedata" type="DAT_I" size="16" dir="in" />
        <port name="wbs_strobe" type="STB" size="1" dir="in" />
        <port name="wbs_write" type="WE" size="1" dir="in" />
        <port name="wbs_ack" type="ACK" size="1" dir="out" />
        <port name="wbs_cycle" type="CYC" size="1" dir="in" />
      </ports>
        </interface>

    </interfaces>

</component>
```

A.2 multi_sampl.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
----------------------------------------------------------------------------
Entity multi_sampl is
----------------------------------------------------------------------------
  generic(
      id               : natural := 2
  );
  port
  (
    -- global signals
    gls_clk        : in std_logic ;
    gls_reset      : in std_logic ;

    -- Wishbone signals
    wbs_add        : in std_logic_vector(9 downto 0);
    wbs_readdata   : out std_logic_vector( 15 downto 0); --donnee
    wbs_writedata  : in std_logic_vector( 15 downto 0); --donnee
    wbs_strobe     : in std_logic ;
    wbs_cycle      : in std_logic ;
    wbs_write      : in std_logic ; -- si 0 lecture
    wbs_ack        : out std_logic;

    -- in/out
    start          : in std_logic;
    ack            : in std_logic;
    SDObus         : in std_logic_vector(7 downto 0);
    clkext         : in std_logic;
    switch         : out std_logic;
    impulse        : out std_logic;
    impulse_clk    : out std_logic;
    impulseX       : out std_logic;
    clkout         : out std_logic;
    SclkDL         : out std_logic;
    irq_valid      : out std_logic
  );
end entity;


----------------------------------------------------------------------------
Architecture multi_sampl_arch of multi_sampl is
----------------------------------------------------------------------------

component wishbone_interface
  generic(
      id               : natural := 2 --identifiant du composant
  );
  port
  (
  -- global signals
  gls_reset      : in std_logic ;
  gls_clk        : in std_logic ;
  -- Wishbone signals
  wbs_addresse   : in std_logic_vector(1 downto 0);
  wbs_readdata   : out std_logic_vector(15 downto 0);
  wbs_writedata  : in std_logic_vector(15 downto 0);
  wbs_strobe     : in std_logic ;
  wbs_cycle      : in std_logic ;
  wbs_write      : in std_logic ; -- si 0 lecture
  wbs_ack        : out std_logic;
  --irq_valid
    irq_valid      : in std_logic;
  --donnees
  data           : in std_logic_vector(13 downto 0);
  high_delay     : out std_logic_vector(15 downto 0);
  delayX         : out std_logic_vector(15 downto 0)
  );
end component;

component diviseur
  port(
  clk        : in  std_logic;
  reset      : in  std_logic;
  clkext     : in  std_logic;
```

```
    divstart  : in  std_logic;
    clkout    : out  std_logic;
    Sclk      : out  std_logic
    );
  end component;

  component gene_impulse
    port(
    clkext      : in  std_logic;
    reset       : in  std_logic;
    start       : in  std_logic;
    high_delay: in  std_logic_vector(15 downto 0);
    switch      : out  std_logic;
    imp_clk     : out  std_logic;
    impulse     : out  std_logic
    );
  end component;

  component delay_sclk
    port(
    clkext    : in  std_logic;
    reset     : in  std_logic;
    start     : in  std_logic;
    ack       : in  std_logic;
    Sclkin    : in  std_logic;
    delayX    : in  std_logic_vector(15 downto 0);
    regstart  : out  std_logic_vector(1 downto 0);
    divstart  : out  std_logic;
    irq_valid : out std_logic;
    Sclkout   : out  std_logic;
    impulseX  : out  std_logic
    );
  end component;

  component registre
    port(
    Sclk      : in  std_logic;
    reset     : in  std_logic;
    regstart  : in  std_logic_vector(1 downto 0);
    SDOad     : in  std_logic_vector(3 downto 0);
    SDObus    : in  std_logic_vector(7 downto 0);
    data      : out  std_logic_vector(13 downto 0)
    );
  end component;

  ------------------------------------
  --mettre signaux entre les composants
  ------------------------------------
  signal s_Sclk,s_SclkDL,s_divstart,s_irq_valid,s_impulse,s_imp_clk  : std_logic;
  signal s_regstart                   : std_logic_vector(1 downto 0);
  signal s_data                       : std_logic_vector(13 downto 0);
  signal s_delayX                     : std_logic_vector(15 downto 0);
  signal s_high_delay          : std_logic_vector(15 downto 0);

  begin

  wishbone_interface_connect: wishbone_interface
    generic map(
        id => id --identifiant du composant
    )
    port map
    (
      -- global signals
      gls_reset       => gls_reset,
      gls_clk         => gls_clk,
      -- Wishbone signals
      wbs_addresse    => wbs_add(1 downto 0),
      wbs_readdata    => wbs_readdata,
      wbs_writedata   => wbs_writedata,
      wbs_strobe      => wbs_strobe,
      wbs_cycle       => wbs_cycle,
      wbs_write       => wbs_write,
      wbs_ack         => wbs_ack,
      --irq_valid
      irq_valid => s_irq_valid,
      --donnees
      data            => s_data,
      high_delay      => s_high_delay,
      delayX          => s_delayX
    );


  diviseur_inst: diviseur
    port map(
      clk       => gls_clk,
      reset     => gls_reset,
      clkext    => clkext,
      divstart  => s_divstart,
      clkout    => clkout,
      Sclk      => s_Sclk
    );

  gene_impulse_inst: gene_impulse
    port map(
      clkext      => clkext,
```

```
    reset       => gls_reset,
    start       => start,
    high_delay=> s_high_delay,
    switch      => switch,
    imp_clk     => s_imp_clk,
    impulse     => s_impulse
  );

delay_sclk_inst: delay_sclk
  port map(
    clkext      => clkext,
    reset       => gls_reset,
    start       => start,
    ack         => ack,
    Sclkin      => s_Sclk,
    delayX      => s_delayX,
    regstart    => s_regstart,
    divstart    => s_divstart,
    irq_valid   => s_irq_valid,
    Sclkout     => s_SclkDL,
    impulseX    => impulseX
  );

registre_inst: registre
  port map(
    Sclk      => s_SclkDL,
    reset     => gls_reset,
    regstart  => s_regstart,
    SDOad     => wbs_add(5 downto 2),
    SDObus    => SDObus,
    data      => s_data
  );

 impulse_clk <= (s_imp_clk and (not(clkext)));
impulse <= s_impulse;
irq_valid <= s_irq_valid;
SclkDL <= s_SclkDL;

end architecture multi_sampl_arch;
```

A.3 wishbone_interface.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
-------------------------------------------------------------------
Entity wishbone_interface is
-------------------------------------------------------------------
  generic(
    id              : natural := 2 --identifiant du composant
  );
  port
  (
    -- global signals
    gls_reset      : in std_logic ;
    gls_clk        : in std_logic ;
    -- Wishbone signals
    wbs_addresse   : in std_logic_vector(1 downto 0);-- 4 registres: ID, delayX, data, high_delay
    wbs_readdata   : out std_logic_vector(15 downto 0);
    wbs_writedata  : in std_logic_vector(15 downto 0);
    wbs_strobe     : in std_logic ;
    wbs_cycle      : in std_logic ;
    wbs_write      : in std_logic ; -- si 0 lecture
    wbs_ack        : out std_logic;
    --irq_valid
    irq_valid      : in std_logic;
    --donnees
    data           : in std_logic_vector(13 downto 0);
    high_delay     : out std_logic_vector(15 downto 0);
    delayX         : out std_logic_vector(15 downto 0)
  );
end entity;


-------------------------------------------------------------------
Architecture wishbone_arch of wishbone_interface is
 ------------------------------------------------------------------
  signal wb_write      : std_logic ;
  signal wb_read       : std_logic ;
  signal reg_delayX    : std_logic_vector(15 downto 0);
  signal reg_high    : std_logic_vector(15 downto 0);
  constant ADD_ID      : std_logic_vector(1 downto 0) :="00";--identifiant
  constant RDELAYX      : std_logic_vector(1 downto 0) :="01";--delayX
  constant RHIGH       : std_logic_vector(1 downto 0) :="10";--high_delay
  constant RDATA        : std_logic_vector(1 downto 0) :="11";--data

begin

-- register reading process
  pread : process(gls_clk,gls_reset,irq_valid)
  begin
```

```
  if(gls_reset = '1') then--si reset
      wb_read <= '0';
      wbs_readdata <= (others => '0');
  elsif(rising_edge(gls_clk)) then -- si horloge
      wb_read <= '0';
      wbs_readdata <= (others => '0');
      if(wbs_strobe = '1' and wbs_write = '0' and wbs_cycle = '1')then -- si lecture
          wb_read <= '1';
          case wbs_addresse  is
            when ADD_ID =>
              wbs_readdata <= std_logic_vector(to_unsigned(id,16)); --identifiant dans readdata
            when RDELAYX =>
              wbs_readdata <= reg_delayX; --delayX dans readdata
            when RHIGH =>
              wbs_readdata <= reg_high; --valid dans readdata
            when RDATA =>
              if irq_valid = '0' then
                wbs_readdata <= "11" & data; --data fausse dans le registre de lecture
              else
                wbs_readdata <= "00" & data; --data dans le registre de lecture
              end if;
            when others =>
          end case;

        end if;
  end if;
  end process pread;

  wbs_ack <= wb_read or wb_write;

-- register writing process
  pwrite : process(gls_clk,gls_reset)
  begin
  if(gls_reset = '1') then--si reset
    wb_write <= '0';
    reg_delayX <= (others => '0');
  elsif(rising_edge(gls_clk)) then -- si horloge
      wb_write <= '0';
      if(wbs_strobe = '1' and wbs_write = '1' and wbs_cycle = '1')then -- si ecriture
          wb_write <= '1';
          case wbs_addresse  is
            when RDELAYX =>
              reg_delayX <= wbs_writedata(15 downto 0);
            when RHIGH =>
              reg_high <= wbs_writedata(15 downto 0);
            when others =>
          end case;
      end if;
  end if;
  end process pwrite;
   delayX <= reg_delayX;
   high_delay <= reg_high;
end architecture wishbone_arch;
```

A.4 delay_sclk.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity delay_sclk is
port(
  clkext     : in  std_logic;
  reset      : in  std_logic;
  start      : in  std_logic;
  ack        : in  std_logic;
  Sclkin     : in  std_logic;
  delayX     : in  std_logic_vector(15 downto 0);
  regstart   : out  std_logic_vector(1 downto 0);
  divstart   : out  std_logic;
  irq_valid  : out std_logic;
  Sclkout    : out  std_logic;
  impulseX   : out  std_logic
);
end entity;

architecture delay_sclk_arch of delay_sclk is
    type state_type is (init,s_delay, s_sclk,s_read);
    signal current_state: state_type;
    signal compt    : std_logic_vector(15 downto 0);
    signal out_impulseX, start_int, out_sclk, out_divstart,out_irq_valid : std_logic;
    signal out_regstart : std_logic_vector(1 downto 0);
begin
  process (clkext, reset, start)
  variable start_old : std_logic;
  begin
    if (reset = '1') then
        start_int <= '0';
        start_old := '0';
    elsif rising_edge (clkext)  then
```

```
36        if start_old = '0' and start = '1' then
            start_int <= '1';
38        else
            start_int <= '0';
40        end if;
          start_old := start;
42
      end if;
44  end process;

46   -- process#2: impulseX
     process(clkext,reset,current_state,delayX,start_int,Sclkin,ack)
48   variable sclkin_old : std_logic;
     begin
50    if (reset='1') then
        current_state <= init;
52    elsif rising_edge(clkext) then
        case current_state is
54        when init =>
            compt <= (others => '0');
56          out_impulseX <= '0';
            out_sclk <= '0';
58          out_divstart <= '0';
            out_irq_valid <= '0';
60          if start_int = '1' then
              current_state <= s_delay;
62          else
              current_state <= init;
64          end if;

66        when s_delay =>
            compt <= std_logic_vector(unsigned(compt) + 1);
68          if compt = delayX then
              compt <= (others => '0');
70            out_impulseX <= '1';
              out_divstart <= '1';
72            current_state <= s_sclk;
            else
74            current_state <= s_delay;
            end if;
76
          when s_sclk =>
78          if sclkin_old = '0' and Sclkin = '1' then
              compt <= std_logic_vector(unsigned(compt) + 1);
80            out_sclk <= Sclkin;
            end if;
82          if sclkin_old = '1' and Sclkin = '0' then
              out_sclk <= Sclkin;
84          end if;
            sclkin_old := Sclkin;
86          if ((compt > "0000000000000010") and (compt < "0000000000010001")) then
              out_regstart <= "01";
88          elsif ((compt > "0000000000010010") and (compt < "0000000000100001")) then
              out_regstart <= "10";
90          else
              out_regstart <= "00";
92          end if;
            if compt = "0000000000100011" then
94            compt <= (others => '0');
              out_impulseX <= '0';
96            out_regstart <= "00";
              out_divstart <= '0';
98            current_state <= s_read;
            else
100           current_state <= s_sclk;
            end if;
102
        when s_read =>
104       out_irq_valid <= '1';
          if ack = '1' then
106         current_state <= init;
          else
108         current_state <= s_read;
          end if;
110
        when others =>
112       out_sclk <= '0';
          out_regstart <= "00";
114       out_impulseX <= '0';
          out_divstart <= '0';
116       current_state <= init;
        end case;
118   end if;
    end process;
120
    impulseX <= out_impulseX;
122 Sclkout <= out_sclk;
    regstart <= out_regstart;
124 divstart <= out_divstart;
    irq_valid <= out_irq_valid;
126end architecture  delay_sclk_arch;
```

A.5 gene_impulse.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity gene_impulse is
port(
  clkext      : in  std_logic;
  reset       : in  std_logic;
  start       : in  std_logic;
  high_delay  : in  std_logic_vector(15 downto 0);
  switch      : out  std_logic;
  imp_clk    : out  std_logic;
  impulse     : out  std_logic
);
end entity;

architecture gene_impulse_arch of gene_impulse is
    type state_type is (init,s_out);
    signal current_state: state_type;
    signal compt  :std_logic_vector(15 downto 0);
    signal out_impulse,out_imp_clk,start_int,out_switch : std_logic;
begin
  -- process: start
  process (clkext, reset, start)
  variable start_old : std_logic;
  begin
    if (reset = '1') then
        start_int <= '0';
        start_old := '0';
    elsif rising_edge (clkext)  then
        if start_old = '0' and start = '1' then
           start_int <= '1';
        else
           start_int <= '0';
        end if;
        start_old := start;
    end if;
  end process;

  -- process: impulse
  process(clkext,reset,current_state,start_int,high_delay)
  begin
    if (reset='1') then
      current_state <= init;
    elsif rising_edge(clkext) then
      case current_state is
        when init =>
          compt<= (others => '0');
           out_impulse <= '0';
           out_imp_clk <= '0';
           out_switch <= '0';
         if start_int = '1' then
           current_state <= s_out;
         else
           current_state <= init;
         end if;
        when s_out =>
         compt <= std_logic_vector(unsigned(compt) + 1);
         out_impulse <= '1';
         out_switch <= '1';
         out_imp_clk <= '1';
         if compt > "0000000000001010" then
           out_imp_clk <= '0';
         end if;
         if compt = high_delay then
           compt <= (others => '0');
           current_state <= init;
         else
           current_state <= s_out;
         end if;
        when others =>
         out_impulse <= '0';
         out_imp_clk <= '0';
         out_switch <= '1';
         current_state <= init;
      end case;
    end if;
   end process;
  impulse  <= out_impulse;
  imp_clk <= out_imp_clk;
  switch <= out_switch;
end architecture  gene_impulse_arch;
```

A.6 diviseur.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.numeric_std.all;

entity diviseur is
port(
   clk       : in  std_logic;
   reset     : in  std_logic;
   clkext    : in  std_logic;
   divstart  : in  std_logic;
   clkout    : out  std_logic;
   Sclk      : out  std_logic
);
end entity;

architecture diviseur_arch of diviseur is
signal div  : unsigned(15 downto 0);
signal togg : std_logic;
begin
  process (clkext, reset,divstart)
  begin
    if (reset = '1') or (divstart = '0')  then
      div <= (others => '0');
      togg <= '0';
    elsif rising_edge (clkext)  then
      if div = "0000000000010100" then
        div <= (others => '0');
        togg <= not togg;
      else
        div <= div + 1;
      end if;
    end if;
  end process;
  Sclk <= togg;
  clkout <= not(clk);

end architecture  diviseur_arch;
```

A.7 registre.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use IEEE.STD_LOGIC_ARITH.ALL;

entity registre is
port(
   Sclk      : in  std_logic;
   reset     : in  std_logic;
   regstart  : in  std_logic_vector(1 downto 0);
   SDOad     : in  std_logic_vector(3 downto 0);
   SDObus    : in  std_logic_vector(7 downto 0);
   data      : out  std_logic_vector(13 downto 0)
);
end entity;

architecture registre_arch of registre is
TYPE reg_type IS ARRAY (15 downto 0) of std_logic_vector(13 downto 0);
signal reg : reg_type;
begin
  process(Sclk,reset,regstart)
  begin
    if (reset = '1') then
      reg(0) <= (others => '0');
    reg(1) <= (others => '0');
    reg(2) <= (others => '0');
    reg(3) <= (others => '0');
    reg(4) <= (others => '0');
    reg(5) <= (others => '0');
    reg(6) <= (others => '0');
    reg(7) <= (others => '0');
    reg(8) <= (others => '0');
    reg(9) <= (others => '0');
    reg(10) <= (others => '0');
    reg(11) <= (others => '0');
    reg(12) <= (others => '0');
    reg(13) <= (others => '0');
    reg(14) <= (others => '0');
    reg(15) <= (others => '0');
    elsif falling_edge (Sclk) then
    if(regstart = "01") then
        reg(0) <= reg(0)(12 downto 0) & SDObus(0);
        reg(2) <= reg(2)(12 downto 0) & SDObus(1);
        reg(4) <= reg(4)(12 downto 0) & SDObus(2);
        reg(6) <= reg(6)(12 downto 0) & SDObus(3);
        reg(8) <= reg(8)(12 downto 0) & SDObus(4);
        reg(10) <= reg(10)(12 downto 0) & SDObus(5);
        reg(12) <= reg(12)(12 downto 0) & SDObus(6);
        reg(14) <= reg(14)(12 downto 0) & SDObus(7);
    elsif (regstart = "10") then
        reg(1) <= reg(1)(12 downto 0) & SDObus(0);
        reg(3) <= reg(3)(12 downto 0) & SDObus(1);
        reg(5) <= reg(5)(12 downto 0) & SDObus(2);
```

```
        reg(7) <= reg(7)(12 downto 0) & SDObus(3);
        reg(9) <= reg(9)(12 downto 0) & SDObus(4);
        reg(11) <= reg(11)(12 downto 0) & SDObus(5);
        reg(13) <= reg(13)(12 downto 0) & SDObus(6);
        reg(15) <= reg(15)(12 downto 0) & SDObus(7);
    end if;
    end if;
  end process;
  data <= reg(to_integer(unsigned(SDOad)));
end architecture  registre_arch;
```

Annexe B

Code source et makefile espace utilisateur

B.1 Multi-Sample.c

```
#include <linux/ppdev.h>

#include "Multi-Sample.h"

static const char *device = "/dev/spidev1.1";
static uint8_t data = 0x00;
static uint8_t bits = 8;
static uint32_t speed = 375000;
static uint16_t delayspi = 0;
static uint8_t mode = 0;

  /*************************************************
                    FPGA communication
  *************************************************/
unsigned short fpgaregs(unsigned int offset,int cmd,unsigned short value){

  switch(cmd){
    case cmd_WRITE:
        *(unsigned short*)(ptr_fpga+(control_Base+offset)) = (unsigned short)value;
        break;
    case cmd_READ:
        value = *(unsigned short*)(ptr_fpga+(control_Base+offset));
        break;
  }

  return value;
}

  /*************************************************
                    QuickSort
  *************************************************/

int partitionner(unsigned int *tableau, int p, int r) {
    int pivot = tableau[p], k = p-1, l = r+1;
    int temp;
    while (1) {
        do
            l--;
        while (tableau[l] > pivot);
        do
            k++;
        while (tableau[k] < pivot);
        if (k < l) {
            temp = delay_DL[k];
            delay_DL[k] = delay_DL[l];
            delay_DL[l] = temp;
            temp = delay_FPGA[k];
            delay_FPGA[k] = delay_FPGA[l];
            delay_FPGA[l] = temp;
            temp = res[k];
            res[k] = res[l];
            res[l] = temp;
            if (chan == 0){
              temp = res_chan2[k];
              res_chan2[k] = res_chan2[l];
              res_chan2[l] = temp;
              temp = res_chan3[k];
              res_chan3[k] = res_chan3[l];
              res_chan3[l] = temp;
              temp = res_chan4[k];
              res_chan4[k] = res_chan4[l];
              res_chan4[l] = temp;
```

```
            }
        }
        else
            return l;
    }
}

void quickSort(unsigned int *tableau, int p, int r) {
    int q;
    if (p < r) {
        q = partitionner(tableau, p, r);
        quickSort(tableau, p, q);
        quickSort(tableau, q+1, r);
    }
}

/*********************************************************
                    SPI Transfert
*********************************************************/

static void transfer(int fd)
{
  int retour;
  uint8_t tx[] = {data,data};
  uint8_t rx[ARRAY_SIZE(tx)] = {0x00, };
  struct spi_ioc_transfer tr = {
    .tx_buf = (unsigned long)tx,
    .rx_buf = (unsigned long)rx,
    .len = ARRAY_SIZE(tx),
    .delay_usecs = delayspi,
    .speed_hz = speed,
    .bits_per_word = bits,
  };

  retour = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
}


/*********************************************************
                    SIGINT function
*********************************************************/

void quit(){
  close(fd);
  free(res);
  if (chan == 0){
    free(res_chan2);
    free(res_chan3);
    free(res_chan4);
  }
  free(delay_DL);
  free(delay_FPGA);
  close(fdgpio);
  close(fdfpga);
  close(fdstart);
  close(fdack);
  printf("keyboard interrupt\n");
  exit(0);
}

/*********************************************************
                          MAIN
*********************************************************/
int main(int argc, char *argv[])
{
  int i,j = 1, trig = 0,ret = 0;
  unsigned int high = 0x03E8;
  static unsigned int delay, window, step,delay_init, nbrpoints,nbrperiodDL,freq_clk,tpsperiodDL,nbrpointperiod;
  FILE * resFile;
  unsigned int delayX = 0;
  unsigned int lenable,start,ack;
  uint8_t data_old = 0xFF;

  /* quit when Ctrl-C pressed */
  signal(SIGINT, quit);

  /*********************************************************
                Processing Arguments
*********************************************************/

  if ((argc < 8) || (argc > 9)){
    printf("invalid arguments number\n");
    printf("Usage: Multi-Sample delay window step clk_freq trig chan file [loading]\n\n");
    printf("Sampling a signal in a window display with a step size after the trigger delay.\n\n");
    printf("Options:\n");
    printf("        delay        in ns (step size min : clk period)\n");
    printf("        window       in ns (step size min : clk period)\n");
    printf("        step         in ps (step size min : 250 ps)\n\n");
    printf("        clk freq     in MHz (0 if use internal clock)\n\n");
    printf("        trig         0 : interne, 1 : externe\n\n");
    printf("        chan         channel 1 - 4, 0 for all\n\n");
    printf("        file         filename to copy data\n\n");
    printf("        [loading]    if loading fpga and open portD (optionnal)\n\n");
    printf("Example:\n");
    printf("Multi-Sample 5000 10000 1000 0 0 1 1GHzSampling\n");
```

```
    printf("--> 1GHz sampling (channel 1) after a period of 5us in a display window of 10us with internal clock\n");
159   return -1;
  }

  if(argc == 9){
    system("modprobe fpgaloader");
    system("/usr/bin/loadgpio.sh");
165   sleep(1);
    system("dd if=top_pod_multi_sampl.bit of=/dev/fpgaloader");
167   sleep(1);
  }
169 delay = atoi(argv[1]);
  window = atoi(argv[2]);
171 step = atoi(argv[3]);
  freq_clk = atoi(argv[4]);
173 trig = atoi(argv[5]);
  chan = atoi(argv[6]);
175 if (chan > 4) {
    printf("channel 1 - 4, 0 for all\n\n");
177   return 0;
  }
179 delay_init = delay*1000;
  step = (step/250)*250;
181 window = window*1000;

183 /*****************Open results file*******************/
  resFile = fopen (argv[7],"w");
185 if(resFile == NULL)
    return 0;
187 chmod(argv[7],766);

189 /*******Calcul des temps et nombre de points************/
  nbrpoints = window/step;
191 if (freq_clk == 0)
    freq_clk = 96;
193 nbrperiodDL = (63750 * freq_clk)/1000000;
  tpsperiodDL = nbrperiodDL* 1000000 / freq_clk;
195 nbrpointperiod = tpsperiodDL / step;

197 printf("clk frequence : %d\n",freq_clk);
  printf("nombre de points pour la courbe totale : %d\n",nbrpoints);
199 printf("nombre de periode d'horloge pour un parcours de DL : %d\n",nbrperiodDL);
  printf("temps pour un parcours de DL : %d\n",tpsperiodDL);
201 printf("nombre de points pour un parcours de DL : %d\n",nbrpointperiod);

203 /*****************Tableaux & values init*******************/

205 res=(unsigned int*)malloc((nbrpoints)*sizeof(unsigned int));
  delay_DL=(unsigned int*)malloc((nbrpoints)*sizeof(unsigned int));
207 delay_FPGA=(unsigned int*)malloc((nbrpoints)*sizeof(unsigned int));

209 delay_FPGA[0] = delay_init;
  delay_DL[0] = 0;
211 res[0] = 0;

213 if (chan == 0){//Si echantillonnage multi-voies
    res_chan2=(unsigned int*)malloc((nbrpoints)*sizeof(unsigned int));
215   res_chan3=(unsigned int*)malloc((nbrpoints)*sizeof(unsigned int));
    res_chan4=(unsigned int*)malloc((nbrpoints)*sizeof(unsigned int));
217
    res_chan2[0] = 0;
219   res_chan3[0] = 0;
    res_chan4[0] = 0;
221 }else{
    free(res_chan2);
223   free(res_chan3);
    free(res_chan4);
225 }

227 /*************************************************************
                Creation des valeurs de delais
229 *************************************************************/
  for(i = 1; i < (nbrpoints); i++){
231   delay_DL[i] = (delay_DL[i-1] + step);
    delay_FPGA[i] =  delay_FPGA[i-1];
233   res[i] = 0 ;
    if (chan == 0){
235     res_chan2[i] = 0 ;
      res_chan3[i] = 0 ;
237     res_chan4[i] = 0 ;
    }
239   if ((delay_DL[i]) >= tpsperiodDL){
      delay_DL[i] = delay_DL[i] - tpsperiodDL;
241     delay_FPGA[i] = delay_FPGA[i] + tpsperiodDL;
    }
243 }

245 //Tri du tableau avec les valeurs delay_DL croissantes
  quickSort(delay_DL, 1, nbrpoints-2);
247
  for(i = 1; i < (nbrpoints); i++){
249   if (delay_DL[i+1] != delay_DL[i]){
      quickSort(delay_FPGA, j, i);
251     j = i + 1;
    }
```

```
253 }
    quickSort(delay_FPGA, j, nbrpoints);
255
    /*******************************************************
257                         SPI Delay Line Config
    *******************************************************/
259
    printf("Opening %s\n", LE_GPIO_PIN);
261 if ((fdgpio = open(LE_GPIO_PIN, O_RDWR)) < 0) {
      perror("Error");
263   exit(1);
    }
265
    lenable = 0x00;
267 write(fdgpio, &lenable, sizeof(lenable));

269 system("modprobe spidev");
    sleep(1);
271 if (geteuid() != 0) {
      printf("No root access rights !\n");
273   return(-1);
    }
275
    fd = open(device, O_RDWR);
277 if (fd < 0)
      printf("can't open device\n");
279
    /*
281  * spi mode
     */
283 ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
    if (ret == -1)
285   printf("can't set spi mode\n");


287
    /*
289  * bits per word
     */
291 ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
    if (ret == -1)
293   printf("can't set bits per word\n");

295 /*
     * max speed hz
297  */
    ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
299 if (ret == -1)
      printf("can't set max speed hz\n");
301
    printf("spi mode: %d\n", mode);
303 printf("bits per word: %d\n", bits);
    printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);
305

307 /*******************************************************
                         FPGA initialisation
309 *******************************************************/

311 printf("Opening %s\n", START_GPIO_PIN);
    if ((fdstart = open(START_GPIO_PIN, O_RDWR)) < 0) {
313   perror("Error");
      exit(1);
315 }
    if (trig == 0){
317   start = 0x00;
      write(fdstart, &start, sizeof(start));
319 }

321 printf("Opening %s\n", ACK_GPIO_PIN);
    if ((fdack = open(ACK_GPIO_PIN, O_RDWR)) < 0) {
323   perror("Error");
      exit(1);
325 }

327 ack = 0x00;
    write(fdack, &ack, sizeof(ack));
329
    fdfpga = open("/dev/mem", O_RDWR|O_SYNC);
331 if (fdfpga < 0) {
      printf("can't open file /dev/mem\n");
333   return -1;
    }
335
    ptr_fpga = mmap(0, 8192, PROT_READ|PROT_WRITE, MAP_SHARED, fdfpga, FPGA_ADDRESS);
337 if (ptr_fpga == MAP_FAILED) {
      printf("mmap failed\n");
339   return -1;
    }
341
    fpgaregs(control_HIGH_OFFSET,cmd_WRITE,(unsigned short)high);
343 fpgaregs(control_DELAY_OFFSET,cmd_WRITE,(unsigned short)0x00);

345 printf("\n\nAcquisition ");
    /*******************************************************
347                         Data acquisition
```

```
  ********************************************************/
349 for(i = 0; i < (nbrpoints); i++){

351   delayX = delay_FPGA[i]*freq_clk/1000000;
      data = delay_DL[i]/250;
353

355   if (data != data_old){

357     lenable = 0x01;
        write(fdgpio, &lenable, sizeof(lenable));
359
        transfer(fd);
361
        lenable = 0x00;
363     write(fdgpio, &lenable, sizeof(lenable));

365     data_old = data;

367     printf(".");
        fflush(stdout);
369
      }
371
      fpgaregs(control_DELAY_OFFSET,cmd_WRITE,(unsigned short)delayX);
373   /*Temporisation post-ecriture*/
      for(j = 0; j < 750; j++)
375      asm("nop");

377   if (trig == 0){
        start = 0x01;
379     write(fdstart, &start, sizeof(start));
        start = 0x00;
381     write(fdstart, &start, sizeof(start));
      }
383   if (chan != 0)
        while(((res[i]= fpgaregs(control_DATA_OFFSET + (chan-1)*8,cmd_READ,0)) >= 0x4000));
385   else{
        while(((res[i]= fpgaregs(control_DATA_OFFSET_CHAN1,cmd_READ,0)) >= 0x4000));
387     res_chan2[i]= fpgaregs(control_DATA_OFFSET_CHAN2,cmd_READ,0);
        res_chan3[i]= fpgaregs(control_DATA_OFFSET_CHAN3,cmd_READ,0);
389     res_chan4[i]= fpgaregs(control_DATA_OFFSET_CHAN4,cmd_READ,0);
      }
391   ack = 0x01;
      write(fdack, &ack, sizeof(ack));
393   ack = 0x00;
      write(fdack, &ack, sizeof(ack));
395
  }
397
  /*********************************************************
399                     Data copy in file
  *********************************************************/
401 printf("\nCopy in File %s\n",argv[7]);

403 //Tri du tableau avec les valeurs delay_FPGA croissantes
  quickSort(delay_FPGA, 1, nbrpoints-2);
405 j = 1;
  for(i = 1; i < (nbrpoints); i++){
407   if (delay_FPGA[i+1] != delay_FPGA[i]){
        quickSort(delay_DL, j, i);
409     j = i + 1;
      }
411 }
  quickSort(delay_DL, j, nbrpoints);
413
  for(i = 0; i < (nbrpoints); i++){
415   if(chan != 0){
        fprintf(resFile,"%d %d %d %d\n",delay_DL[i]+delay_FPGA[i],res[i]*2500/16383,delay_DL[i],delay_FPGA[i]);
417   }else{
        fprintf(resFile,"%d %d %d %d %d %d %d\n",delay_DL[i]+delay_FPGA[i],res[i]*2500/16383,res_chan2[i]*2500/16383,
            res_chan3[i]*2500/16383,res_chan4[i]*2500/16383,delay_DL[i],delay_FPGA[i]);
419   }
  }
421
  /*********************************************************
423                     Close main
  *********************************************************/
425 fclose(resFile);
  free(res);
427 if (chan == 0){
      free(res_chan2);
429   free(res_chan3);
      free(res_chan4);
431 }
  free(delay_DL);
433 free(delay_FPGA);
  close(fd);
435 close(fdgpio);
  close(fdfpga);
437 close(fdstart);
  close(fdack);
439 printf("ok\n");
  return ret;
441}
```

B.2 Multi-Sample.h

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>

/* converting string */
#include <string.h>

/* sleep, write(), read() */
#include <unistd.h>

/* memory management */
#include <sys/mman.h>

/* file management */
#include <sys/stat.h>
#include <fcntl.h>

#define control_Base (0x800)
#define control_ID_OFFSET      (0x00)
#define control_DELAY_OFFSET   (0x02)
#define control_HIGH_OFFSET    (0x04)
#define control_DATA_OFFSET    (0x06)
#define control_DATA_OFFSET_CHAN1    (0x06)
#define control_DATA_OFFSET_CHAN2    (0x0E)
#define control_DATA_OFFSET_CHAN3    (0x16)
#define control_DATA_OFFSET_CHAN4    (0x1E)

/* ioctl codes */
#define cmd_ID 0
#define cmd_DELAY 1
#define cmd_HIGH 2
#define cmd_DATA 3

#define cmd_WRITE 0
#define cmd_READ 1

#define FPGA_ADDRESS 0x12000000

#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

#define LE_GPIO_PIN "/dev/gpio/PD19"
#define START_GPIO_PIN  "/dev/gpio/PD21"
#define ACK_GPIO_PIN  "/dev/gpio/PD23"


static int fd,fdgpio,fdfpga;
static int fdstart,fdack;
void* ptr_fpga;
static unsigned int *res, *delay_DL, *delay_FPGA;
static unsigned int chan,*res_chan2,*res_chan3,*res_chan4;
```

B.3 Makefile

```
#
# Makefile pour spidev_test.c
# Auteur  : Nicolas CHRETIEN
# Date    : 16.02.2010
#

ifneq ($(CC),)
# Locally compiled:
ARMADEUS_BASE_DIR:=/home/niko/Stage/armadeus
include $(ARMADEUS_BASE_DIR)/Makefile.in
STAGING_DIR:=$(ARMADEUS_BUILD_DIR)/staging_dir/
INSTALL_DIR:=$(ARMADEUS_ROOTFS_DIR)/usr/bin/
CC=$(ARMADEUS_TOOLCHAIN_PATH)/arm-linux-gcc
DEFINES=-D$(ARMADEUS_BOARD_NAME)
endif

EXEC_NAME = Multi-Sample

default: $(EXEC_NAME)

all: $(EXEC_NAME)

$(EXEC_NAME): $(EXEC_NAME).c
	$(CC) $(CFLAGS) $(DEFINES) -Wall -o $@ $^
	cp $(EXEC_NAME) /projet

clean:
	rm -rf $(EXEC_NAME)
```

Annexe C

Patch de correction de la toolchain pour l'APF9328

C.1 apf9328_spidev.patch

```
diff -ru a/arch/arm/mach-imx/apf9328-dev.c b/arch/arm/mach-imx/apf9328-dev.c
--- a/arch/arm/mach-imx/apf9328-dev.c 2010-02-18 10:36:34.131012315 +0100
+++ b/arch/arm/mach-imx/apf9328-dev.c 2010-02-19 16:50:01.000000000 +0100
@@ -25,6 +25,7 @@
 #include <linux/init.h>
 #include <linux/interrupt.h>
 #include <linux/spi/spi.h>
+#include <linux/spi/spidev.h>
 #include <linux/spi/tsc2102.h>
 #include "../../../drivers/net/can/mcp251x.h"

@@ -42,6 +43,7 @@
 #include <mach/usb.h>
 #include <mach/spi_imx.h>
 #include <mach/imx-regs.h> /* imx_gpio_mode() */
+#include <mach/gpio.h>
 #include <mach/imx-alsa.h>
 #include <linux/usb/isp116x.h>
 #include <mach/imx_ssi.h>
@@ -58,6 +60,10 @@
 #ifdef CONFIG_ARMADEUS_ISP1761_MODULE
 #define CONFIG_ARMADEUS_ISP1761 1
 #endif
+#ifdef CONFIG_SPI_SPIDEV_MODULE
+#define CONFIG_SPI_SPIDEV 1
+#endif
+


 /*
@@ -204,6 +210,46 @@

 #endif /* CONFIG_CAN_MCP251X */

+/*
+ * APF9328 CONFIG SPIDEV
+ */
+#ifdef CONFIG_SPI_SPIDEV
+#define SPIDEV_CS_GPIOB 18
+
+static int spidev_init_gpio(void)
+{
+ /* SPI1 GPIOs */
+ imx_gpio_mode(PC14_PF_SPI1_SCLK);
+ imx_gpio_mode(PC16_PF_SPI1_MISO);
+ imx_gpio_mode(PC17_PF_SPI1_MOSI);
+
+ /* PortB 18 is used as chip select (in GPIO mode) */
+ DR(1) |= 1 << SPIDEV_CS_GPIOB; /* Initializes it High */
+ imx_gpio_mode(GPIO_PORTB | SPIDEV_CS_GPIOB | GPIO_OUT | GPIO_GIUS | GPIO_DR);
+
+ return 0;
+}
+
+/* Chip select command for spidev */
+static void spidev_cs(u32 command)
+{
+ /* PortB 18 is used as chip select */
```

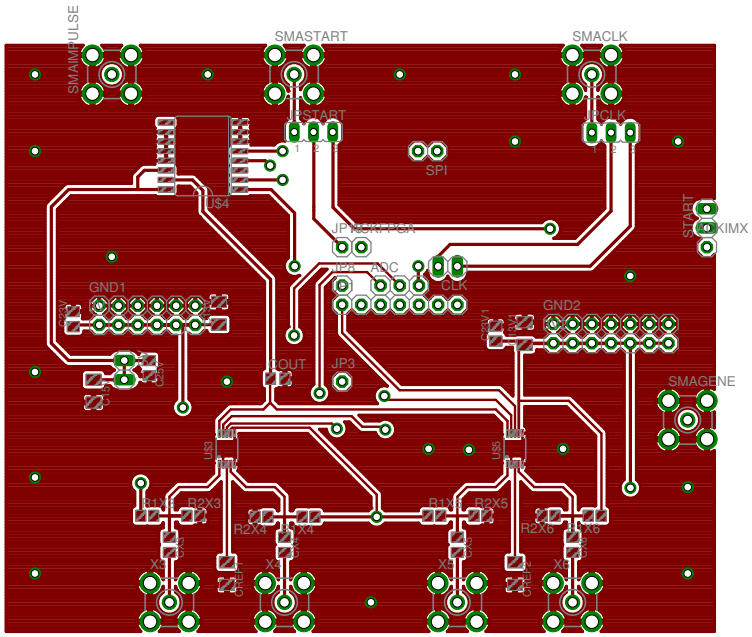
```
59+ if (command == SPI_CS_DEASSERT)
 +   DR(1) |= 1<< SPIDEV_CS_GPIOB;
61+ else
 +   DR(1) &= ~(1<< SPIDEV_CS_GPIOB);
63+}
 +
65+static struct spi_imx_chip spidev_hw = {
 + .cs_control     = spidev_cs,
67+};
 +
69+static struct spidev_platform_data apf9328_spidev_config = {
 + .init = spidev_init_gpio,
71+};
 +
73+#endif /* CONFIG_SPI_SPIDEV */
 +
75
  static struct spi_board_info spi_dev_board_info[] __initdata = {
77 #ifdef CONFIG_SPI_TSC2102
 @@ -230,6 +276,17 @@
79   .platform_data  = &apf9328_mcp251x_config,
  },
81 #endif
 +#ifdef CONFIG_SPI_SPIDEV
83+ {
 +   .modalias          = "spidev",
85+   .controller_data    = &spidev_hw,
 +   .max_speed_hz      = 8000000, /* 8MHz */
87+   .bus_num           = 1, /* SPI1 */
 +   .mode              = SPI_MODE_0,
89+   .chip_select       = 1,
 +   .platform_data     = &apf9328_spidev_config,
91+ },
 +#endif /* CONFIG_SPI_SPIDEV */
93 };

95
 @@ -272,7 +329,7 @@
97 #endif
   platform_add_devices(devices, ARRAY_SIZE(devices));
99
 -#if defined (CONFIG_SPI_TSC2102) || defined (CONFIG_CAN_MCP251X)
101+#if defined (CONFIG_SPI_TSC2102) || defined (CONFIG_CAN_MCP251X) || defined(CONFIG_SPI_SPIDEV)
   spi_register_board_info(spi_dev_board_info, ARRAY_SIZE(spi_dev_board_info));
103 #endif

105diff -ru a/arch/arm/mach-imx/apf9328_lcd_config.h b/arch/arm/mach-imx/apf9328_lcd_config.h
 --- a/arch/arm/mach-imx/apf9328_lcd_config.h  2010-02-18 10:36:34.131012315 +0100
107+++ b/arch/arm/mach-imx/apf9328_lcd_config.h  2010-02-19 16:50:01.000000000 +0100
 @@ -16,7 +16,6 @@
109
  #include <mach/imxfb.h>
111 #include <linux/delay.h>
 -#include <mach/gpio.h>
113
  #ifdef CONFIG_ARCH_IMX
115 #define LCDC_BASE_ADDR IMX_LCDC_BASE
```

Annexe D

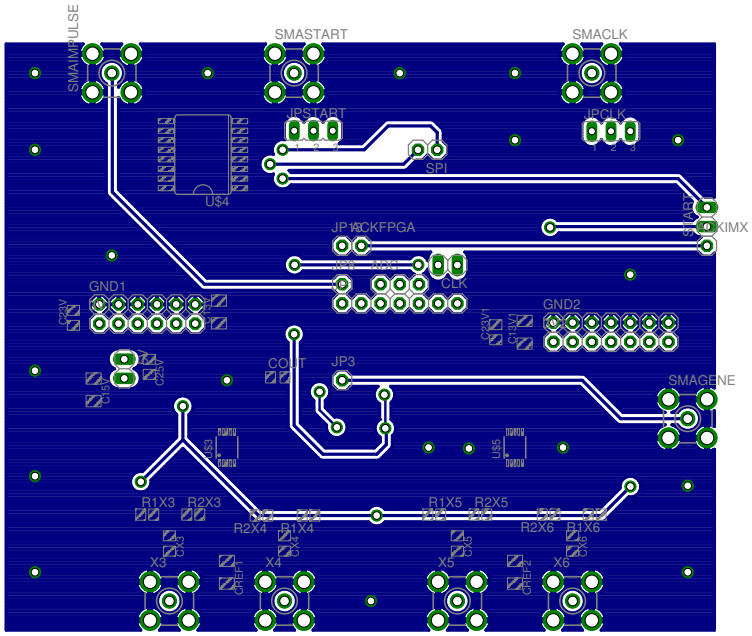
Carte Électronique Finale

D.1 Layout carte 4 voies

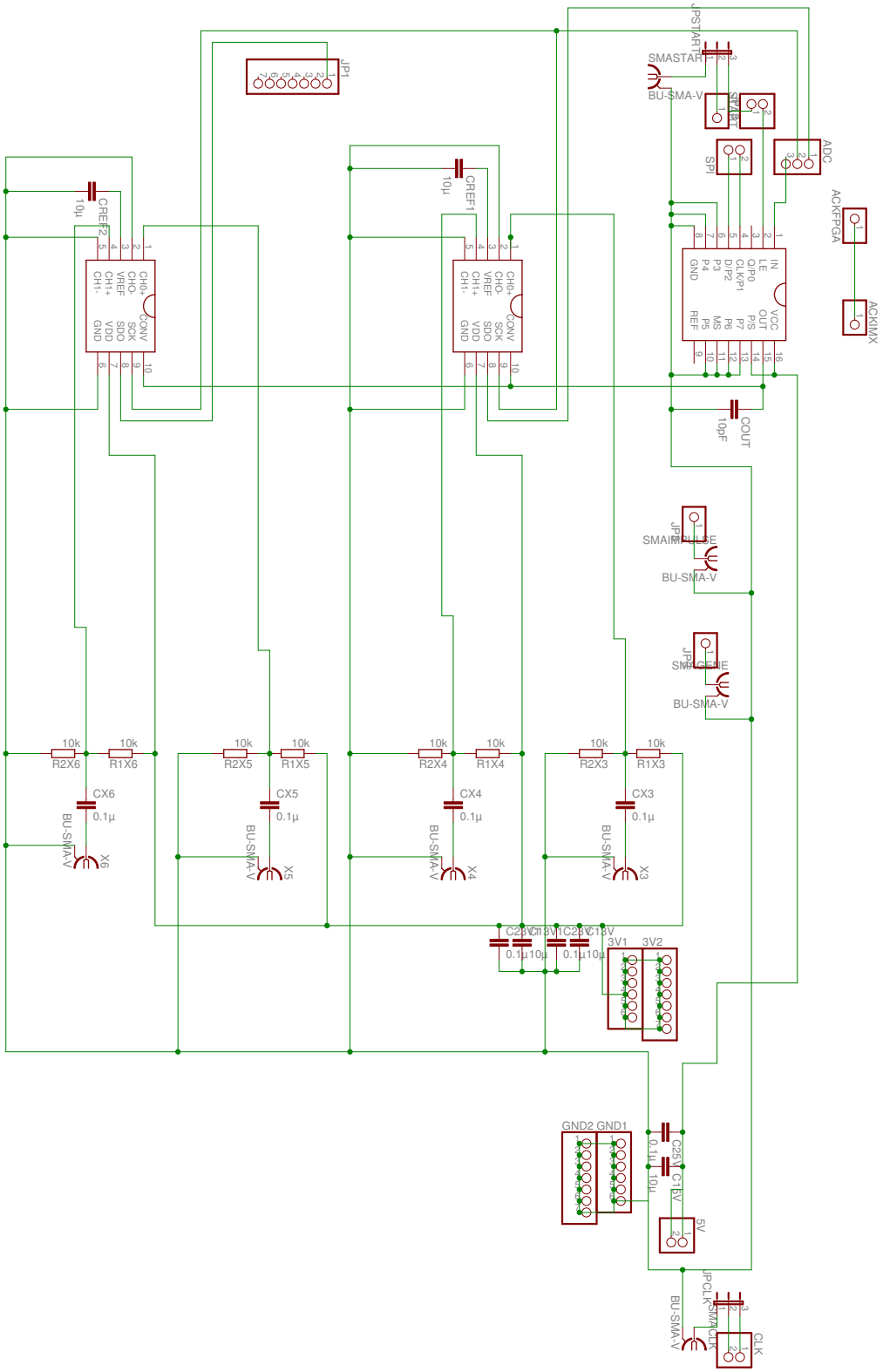
D.1.1 Top



D.1.2 Bottom


SMAIMPULSE
SMASTART
SMACLK
JPSTART
JPCLK
SPI
U$4
JP1
CKFPGA
START
KIMX
CLK
GND1
GND2
JP3
SMAGENE
R1X3
R2X3
R2X4
B1X4
R1X5
R2X5
R2X6
B1X6
X3
X4
X5
X6

D.2 Schéma carte 4 voies



D.3 Photographies