Projet d'Architecture Numérique
Gestion du Microphone de la NEXYS 4
Module EN202

Louis GUÉNÉGO, Électronique 2A groupe D

ENSEIRB-MATMECA, décembre 2020

Introduction

Nous avons choisi pour le projet d'architecture numérique de travailler avec les dispositifs audio présents sur la carte NEXYS4ddr. Nous allons donc acquérir des données audio venant du microphone soudé sur la carte, effectuer des traitement numériques sur ce signal, puis le ressortir sur la prise jack.

1 Cahier des charges

Nous nous sommes fixé différentes étapes à réaliser :

1.1 Étape 1

Acquisition du signal audio du microphone
— Bande passante du signal : 0 à 14kHz
— Maximum de fidélité (le moins de distorsions possible)
— Moins de bruit possible
— Fréquence des échantillon proche de 44100 Hz (qualité CD)
— Résolution d'au moins 16 bits (plus si le FPGA le permet facilement)
Modulation du signal
— Utiliser le dispositif audio de la carte pour restituer le son dans la meilleure qualité possible

1.2 Étape 2

Volume automatique
— Réaliser un dispositif qui permette de réguler le volume du signal audio automatiquement
Filtre de réverbération
— Réaliser un filtre d'effet de réverbération

1.3 Étapes supplémentaires

Réaliser d'autre filtres ou fonctions

Visualisation du spectre sur les LED de la carte

Visualisation du spectre ou effets sur la sortie VGA de la carte

2 Acquisition du signal audio du microphone

2.1 Le microphone de la NEXYS 4

La carte NEXYS 4 DDR est équipée d'un microphone ADMP421 monté sur le côté bottom. Le micro reçois les sons depuis le top à travers le PCB grâce à un via. On peut voir su la figure 1 le branchement du micro.

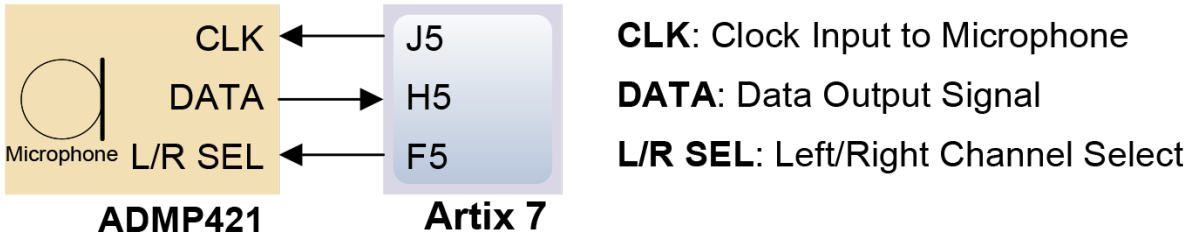

FIGURE 1 – le branchement du microphone de la NEXYS 4 [3]

L'ADMP421 est un microphone MEMS (MicroElectro-Mechanical Systems) intégré accompagné d'un modulateur Σ-Δ. On a donc une seule sortie appelée DATA (figure 1).

Le microphone fonctionne avec une horloge externe qui doit être comprise entre 1 et 3,3 MHz.

On a aussi une entrée L/R SEL qui permet de choisir si la donnée est disponible sur un front montant ou descendant (voir figure 2). Comme on va travailler dans le FPGA sur des fronts montants d'horloge (on aurait pu choisir l'inverse), on utilisera DATA1 et on mettra donc L/R SEL à GND (niveau bas 0V).

Pour la fréquence d'horloge, on utilisera 2,5 MHz, car c'est un multiple entier de 100MHz, la fréquence de l'horloge de base du FPGA ($\frac{100}{40} = 2,5$).

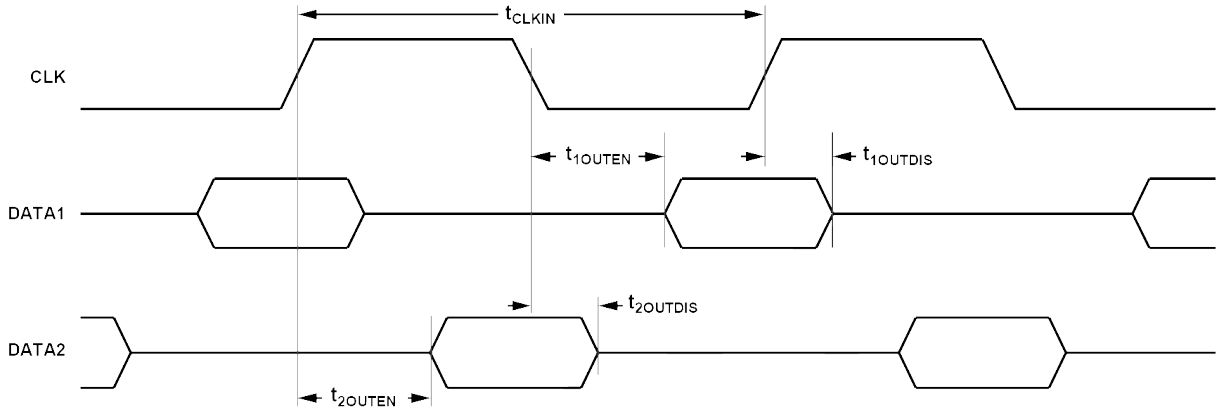

FIGURE 2 – Les timings de l'ADMP421 [4]

2.2 Le signal PDM

Le microphone délivre un signal modulé en PDM (exemple en figure 3). On peut considérer ce signal comme un signal 1bit@2,5MHz. Pour pouvoir travailler plus simplement, nous allons essayer d'avoir un signal à plus basse fréquence, et avec une résolution plus importante.

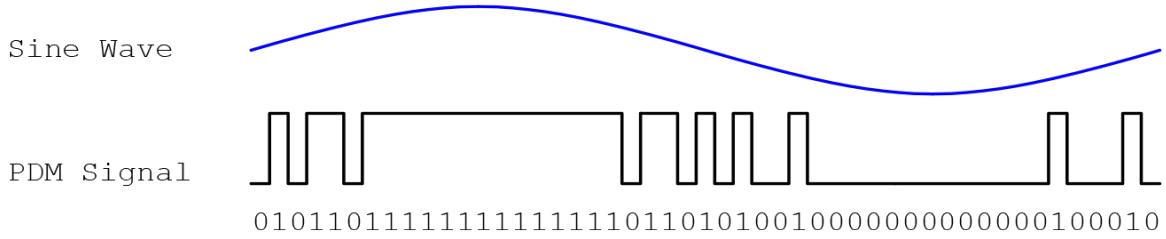

FIGURE 3 – Un exemple de signal PDM [3]

On aimerai se rapprocher de 40kHz, car c'est le double de la fréquence maximum audible par l'oreille humaine, ce qui permet de travailler sur l'ensemble du spectre audio et de respecter le critère de Shannon. Si on division 2,5MHz par 64, on a 39062,5Hz.

2.3 La résolution du signal

Pour choisir le nombre de bits de notre signal, nous allons étudier les blocs DSP disponible dans les Artix 7 (figure 4).

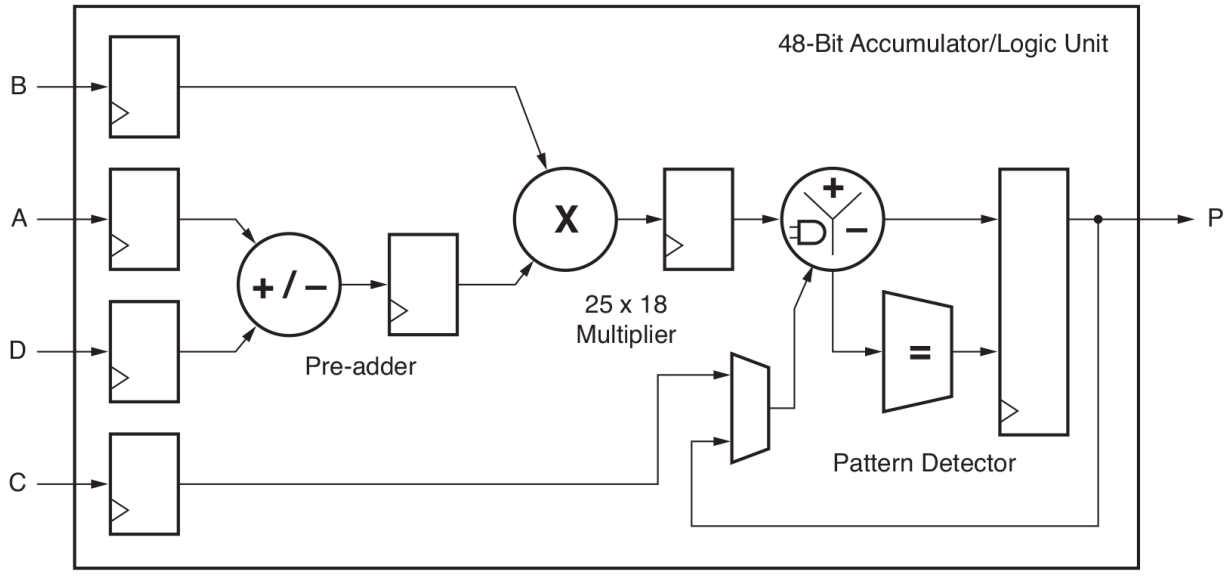

FIGURE 4 – Blocs DSP48E1 [7]

Les blocs DSP disponible dans notre FPGA sont des blocs équipés d'un multiplieur 25x18 bits (soit 43 bits), et d'un accumulateur (additionneur + registre) qui monte jusqu'à 48 bits. On va alors profiter des blocs DSP généreux et utiliser un signal 18bits@39062,5Hz pour tous les filtres et effets dans notre projet.

2.4 Décimation

Pour pouvoir passer de 1bit@2,5MHz à 18bits@39062,5Hz, il va falloir décimer le signal. Dans le cas d'un signal PDM, il va aussi falloir filtrer le bruit haute fréquence que ce type de modulation génère (figure 5).

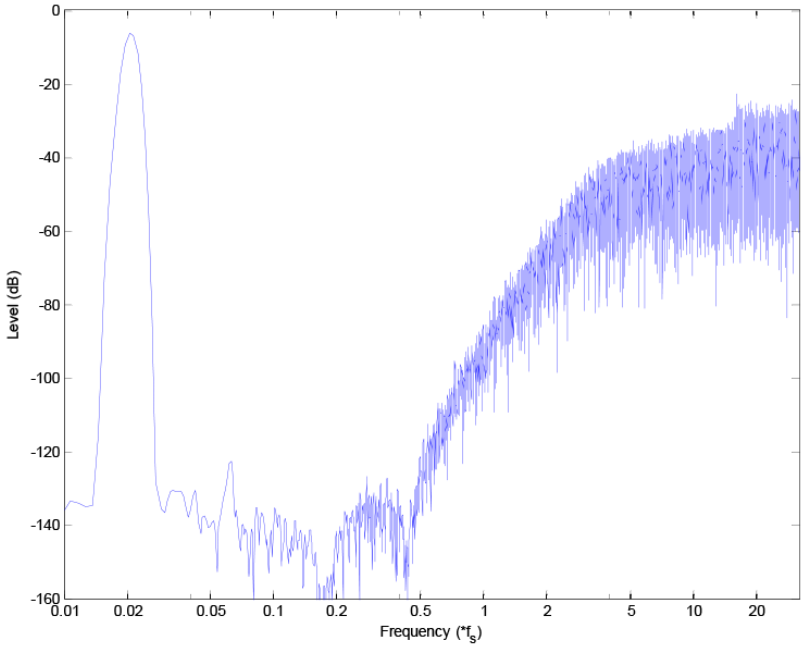

FIGURE 5 – Spectre d'un signal PDM [8]

Comme nous avons un bruit important, nous allons utiliser deux filtres, un décimateur et un filtre passe bas, en cascade avec peu de contraintes sur chacun, plutôt qu'un seul filtre avec des contraintes plus fortes (donc beaucoup de coefficients). On choisi arbitrairement une fréquence d'échantillonage intermédiaire à $\frac{2,5MHz}{8} = 312,5kHz$. On a alors l'architecture suivante :

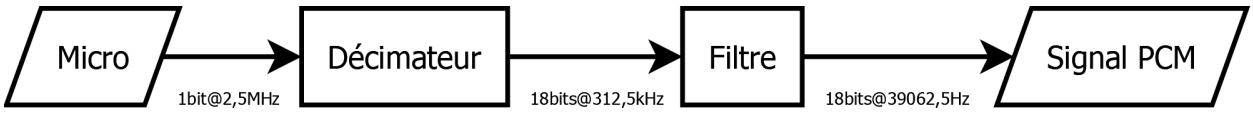

FIGURE 6 – L'architecture d'acquisition

2.5 Réalisation du décimateur (`fir1.vhd`)

Pour notre filtre décimateur, nous allons implanter un filtre passe bas qui va répartir l'information de chaque échantillons sur les échantillons voisin. On pourra alors en sortie ne prendre que 1 échantillon sur 8, sans perdre d'information.

2.5.1 L'architecture du décimateur

En figure 7, nous pouvons voir l'architecture du filtre que nous allons implanter. Pour commencer, nous enregistrons tout les échantillons entrant dans la mémoire RAM `Data_in_mem`. Nous avons de même une ROM qui contient les coefficients du filtre.

Nous avons ensuite un processeur de signal numérique avec 4 étage de pipeline. Ce processeur est commandé à l'aide d'une machine d'état symbolisé par un compteur `cpt`. Les résultats du processeur sont additionnés dans l'accumulateur. Enfin à chaque coup d'horloge `clk_ce_out`, l'échantillon présent dans l'accumulateur est envoyé. Le processeur démarre un nouveau calcul à chaque coup d'horloge de `clk_ce_out`, qui est 8 fois plus lente que `clk_ce_in` : on a bien un décimateur par 8.

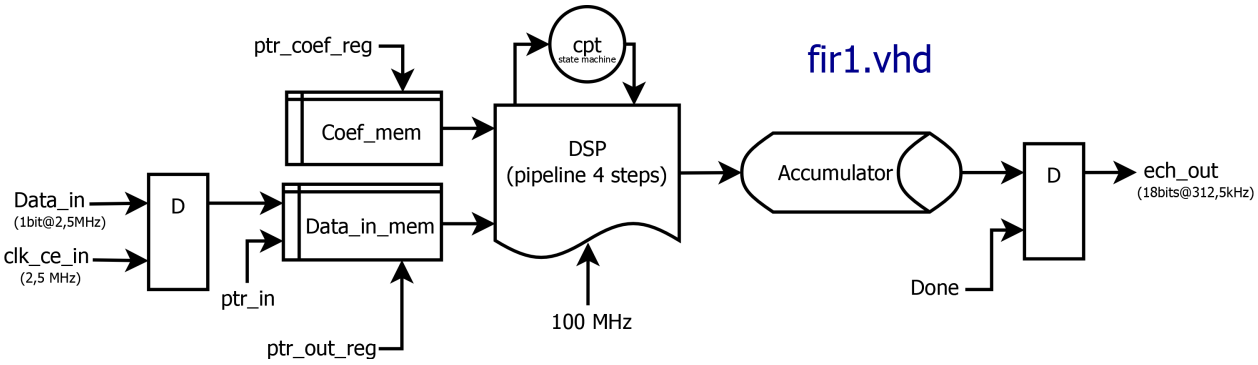

FIGURE 7 – L'architecture du filtre décimateur (`fir1.vhd`)

Remarque 1 : sur la figure 7, l'étape de calcul de `ptr_out` et `ptr_coef` et leur bufferisation est effectué par le processeur de signal numérique et sont comptés comme étape dans le pipeline, même si elles apparaissent à l'extérieur pour une meilleure clarté.

Remarque 2 : le processeur de signal numérique effectue seulement des prises de décision, car comme on a en entrée soit 0, soit 1, on peut pré-calculer les résultats.

2.5.2 Le filtre du décimateur

Le filtre que nous allons implanter sera un filtre décimateur, c'est à dire qu'il devra seulement éliminer les hautes fréquences qui ne pourraient plus être filtrées après décimation, donc tout ce qui est au dessus de $\frac{312,5kHz}{2} = 156,25kHz$. On rappelle aussi que comme on est avec un signal PDM, il va falloir éliminer très fortement ce bruit (On parle de au moins 80 dB d'atténuation à partir de cette fréquence). Aussi, il faut prendre en considération le fait que nous avons seulement 320 (40×8) coups d'horloge pour calculer un échantillons ; sachant aussi que le pipeline met 4 coup d'horloge à se remplir, on se rend compte qu'il faudra éviter de dépasser les 300 échantillons.

Pour générer les coefficients de ce filtre, nous avons utilisé le logiciel gratuit Iowa Hills FIR Filter Design [1] (figure 8). On remarque alors que ce filtre correspond bien à nos attentes : il coupe à au moins 80 dB à partir de 150 kHz, et consomme 128 échantillons.

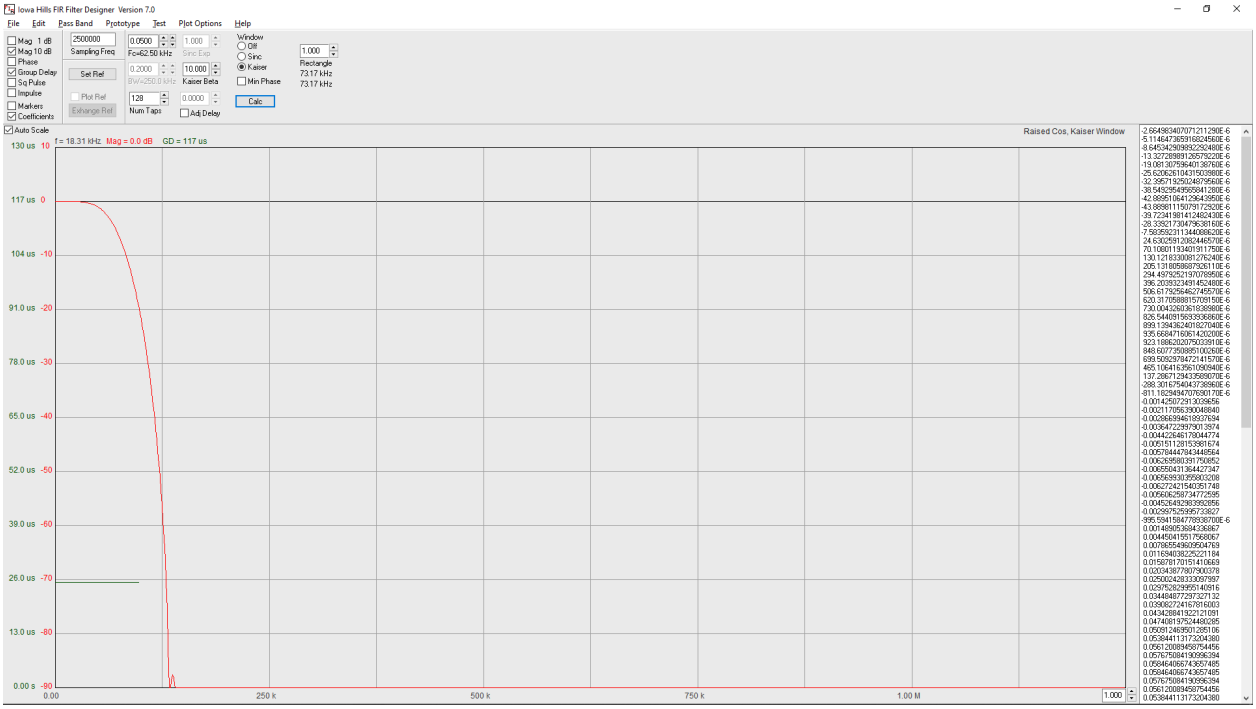

FIGURE 8 – le filtre décimateur sous Iowa Hills [1]

Pour pouvoir utiliser ces coefficients réels dans une architecture numérique, c'est à dire un monde qui ne contient que des nombres entiers, il va falloir les normaliser, ce qui revient à les multiplier par une puissance de 2 en général et arrondir à l'entier. Dans notre cas nous allons les multiplier par 2^{19}, ce qui donnera des nombres signés sur 16 bits.

2.5.3 La validation du système

Pour valider le filtre décimateur `fir1.vhd`, nous allons dans un premier temps générer un signal sinusoïdal modulé en PDM (pour simuler le micro), l'envoyer dans notre filtre, et vérifier qu'on a bien une sinusoïde en sortie.

La deuxième étape sera de visualiser si la machine d'état fonctionne bien, c'est à dire qu'on a bien la phase d'attente, la phase de remplissage du pipeline et la phase de calcul.

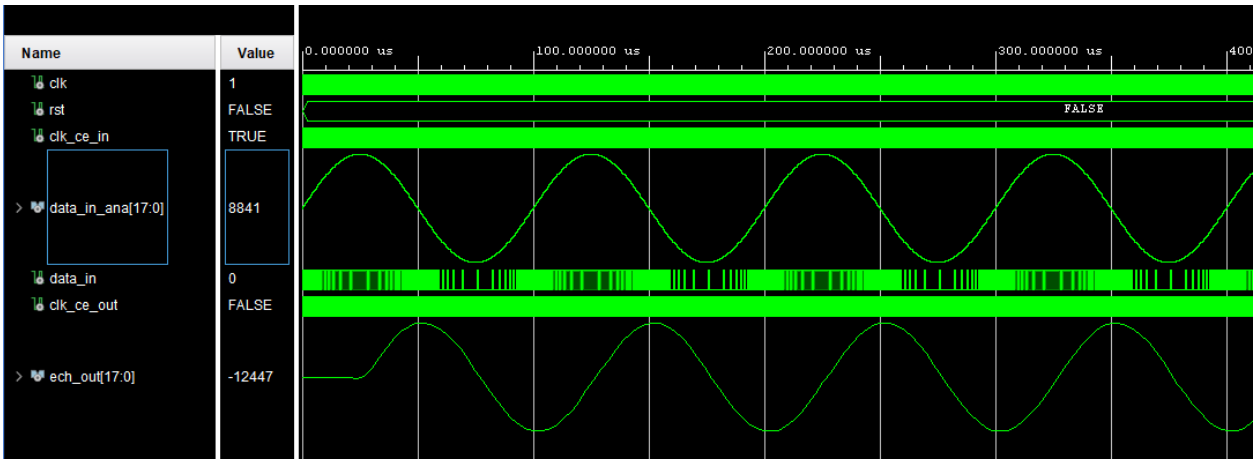

FIGURE 9 – Visualisation de la sinusoïde de sortie

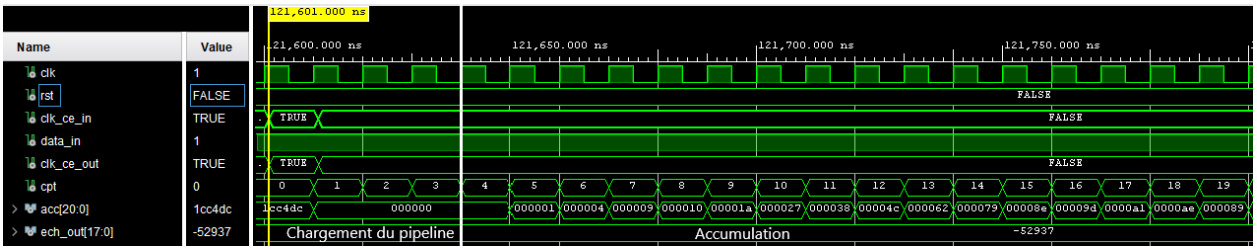

FIGURE 10 – Visualisation de l'initialisation de la machine d'état

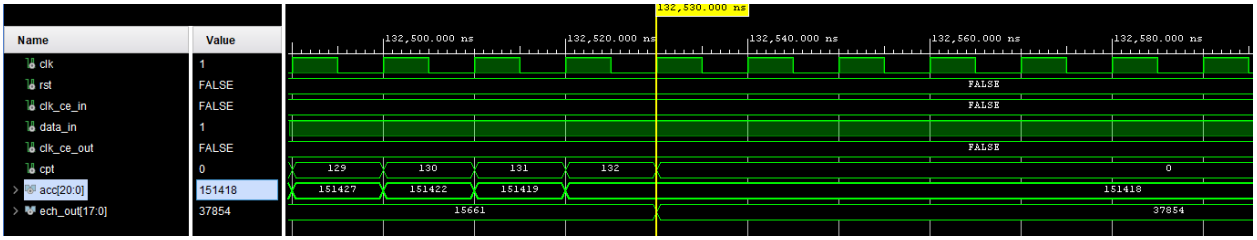

FIGURE 11 – Visualisation de la fin de la machine d'état

On peut voir sur les figures 9, 10 et 11 qu'on a bien un sinus en sortie, et que la machine d'état fonctionne comme prévu.

On a alors les états :
— **Attente :** `cpt` = 0 et `clk_ce_in` = 0.
— **Remplissage du pipeline :** `cpt` = 0 et `clk_ce_in` = 1, puis `cpt` = 1,2 ou 3. `cpt` est incrémenté.
— **Accumulation :** $4 \leq$ `cpt` $< (128 + 4)$ et `clk_ce_in` = 0. `cpt` est incrémenté.
— **Envoie du résultat :** `cpt` = 132. `cpt` revient à 0.

2.6 Réalisation du filtre (`fir2.vhd`)

Pour le filtre, nous allons simplement effectuer un passe bas pour retirer le bruit restant, et aussi décimer le signal par 8. La différence qu'il va y avoir avec le filtre de la partie 2.4 est que celui ci sera plus précis,

et fonctionnera à plus basse fréquence. On devra aussi cette fois utiliser un multiplieur car on ne pourra pas pré-calculer simplement les résultats. On gagne alors deux étape en plus dans le pipeline.

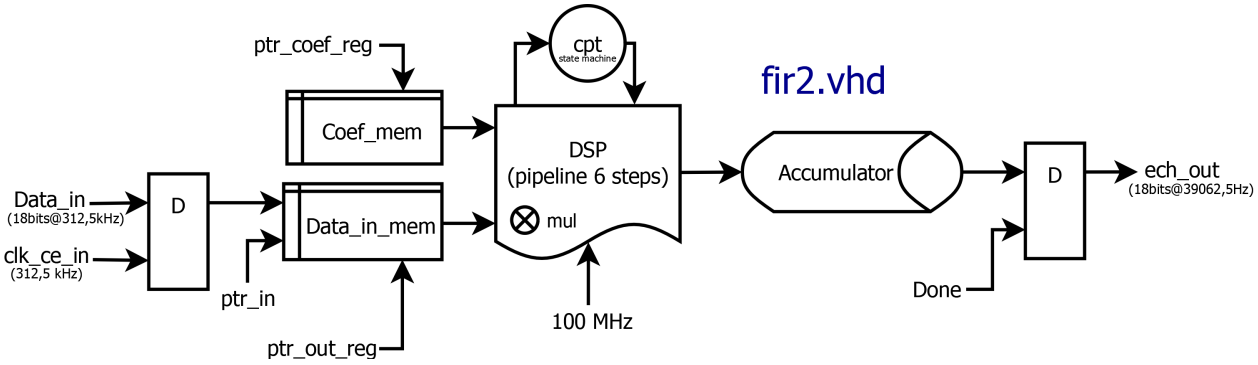

FIGURE 12 – L'architecture du filtre passe bas (`fir2.vhd`)

Pour les coefficients du filtre et la validation de son fonctionnement, nous avons suivi le même cheminement qu'au 2.4.

3 Réalisation du filtre de réverbération

Pour ce qui est du traitement du signal, nous avons ajouté un effet qui est l'effet de réverbération. La réverbération est l'effet qui peut se produire lorsque, dans une pièce par exemple, l'onde sonore rebondit sur les murs et revient une nouvelle fois vers l'auditeur. Ici nous allons simplement effectuer un seul écho de quelques millisecondes.

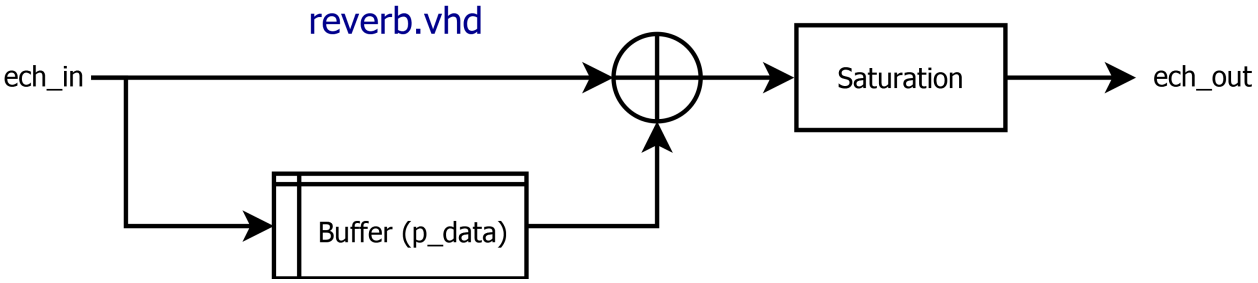

FIGURE 13 – architecture de la réverbération

la réalisation du filtre se compose de deux "process" principaux. Le premier gère l' insertion des échantillons arrivant dans la mémoire, ainsi que l'extraction pour l'addition. Le deuxième process additionne les échantillons rentrant dans le systèmes avec ceux enregistrés dans la mémoire au rythme de `clk_ech` (clock enable d'échantillonage), en prenant en considération les saturations positive ou négative (pour éviter les débordements).

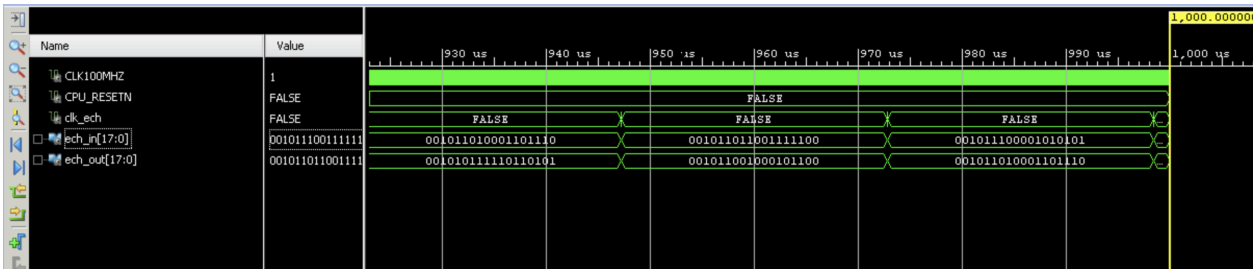

FIGURE 14 – test bench de `reverb.vhd`

4 Volume automatique

4.1 La mauvaise bonne méthode

Pour effectuer un volume automatique, la première approche que nous avons eu a été via une commande feed-forward, comme sur la figure 15.

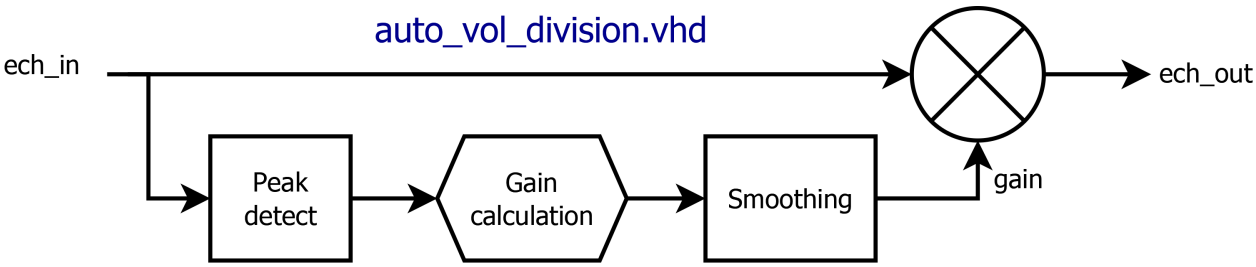

FIGURE 15 – L'architecture du volume automatique en feed-forward

Cette approche est assez simple, on calcule un gain à parti du maximum du signal en entrée, et on prend la précaution de le lisser avant de l'appliquer.

Cette commande est simple et efficace, mais en revanche requiert d'effectuer une division à l'étape "Gain calculation" sur la figure 15 : en effet on réalise l'opération $\frac{A_{voulue}}{A_{signal}} = gain$ pour trouver le bon gain à appliquer. On est donc confronté à un problème temporel (Illustration en figure).

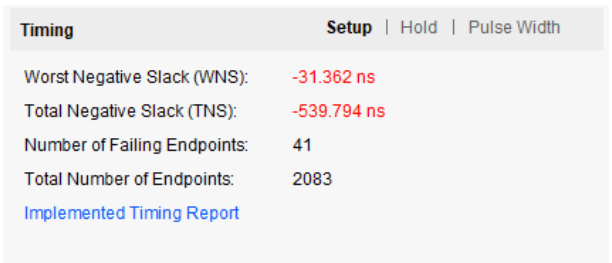

FIGURE 16 – Problème de la division

On utilise alors un bloc DSP et un algorithme qui tente de multiplier une opérande par un facteur et ainsi de retrouver l'autre opérande [6]. On consomme donc relativement peu de matériel mais beaucoup de temps.

En réalité cette méthode serait viable, car on ne fait qu'une division toutes les $\frac{1}{39062,5Hz} = 25,6\mu s$, et notre division prend environ $550ns$.

4.2 La bonne mauvaise méthode

4.2.1 Élaboration du volume automatique

L'architecture feedforward vu au 4.1 nécessite de faire une division et est donc coûteuse. Une autre solution est alors d'utiliser une architecture feedback. On applique d'abord le gain, puis on regarde si le max de la sortie est au niveau souhaité (figure 17).

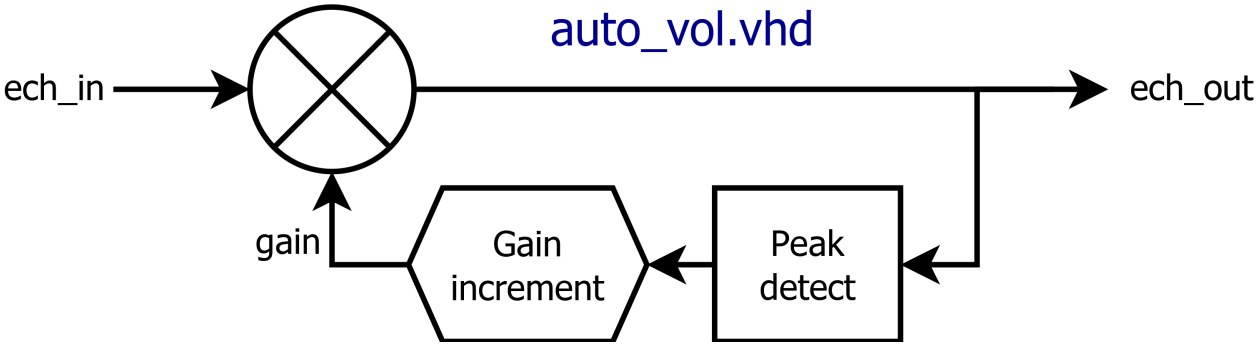

FIGURE 17 – L'architecture du volume automatique en feedback

Le gros avantage de cette architecture est qu'elle consomme alors beaucoup moins de ressources. Le gros désavantage est qu'elle peut être instable comme il y a un rebouclage, et il s'est avéré que ce genre de système est très facilement instable. On va donc assister à un conflit entre moins de distorsion, une meilleure rapidité et une meilleure stabilité.

Un des moyen d'y arriver est d'implanter une variation logarithmique du gain, en utilisant un logarithme en base 2 on n'utilise pas de ressources supplémentaire.

4.2.2 Validation du volume automatique

Pour valider ce système, nous allons comme pour les filtres précédent envoyer un sinus en entrée. En revanche, pour tester si le volume automatique marche bien, nous allons envoyer des sinus d'amplitudes différentes, en espérant avoir un sinus d'amplitude fixe en sortie, et sans distorsions.

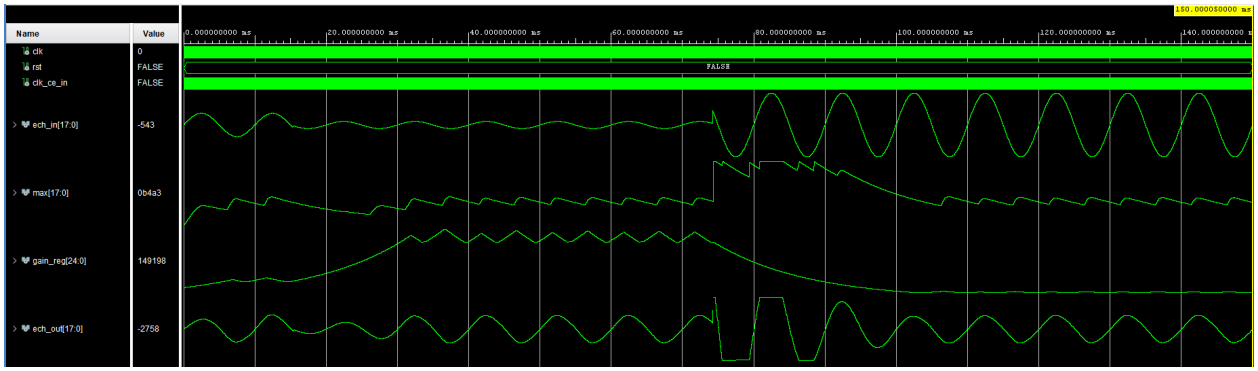

FIGURE 18 – simulation de l'architecture avec un sinus de 100 Hz

On peut voir sur la simulation de la figure 18 qui simule "le pire cas", c'est à dire un sinus à 100Hz qui varie fortement en amplitude, et notre architecture se débrouille pas trop mal.

5 Modulation du signal

Il y a sur la carte un amplificateur audio précédé par un filtre (figure 19). Pour avoir en sortie de la carte un signal de qualité, il faudrait envoyer un signal à au moins 1 MHz. Pour réaliser cela, le plus simple est d'effectuer l'opération inverse de la partie 2. On va alors sur échantillonner le signal avec 2 filtres, puis le moduler en PDM avant de l'envoyer dans la sortie audio (figure 20).

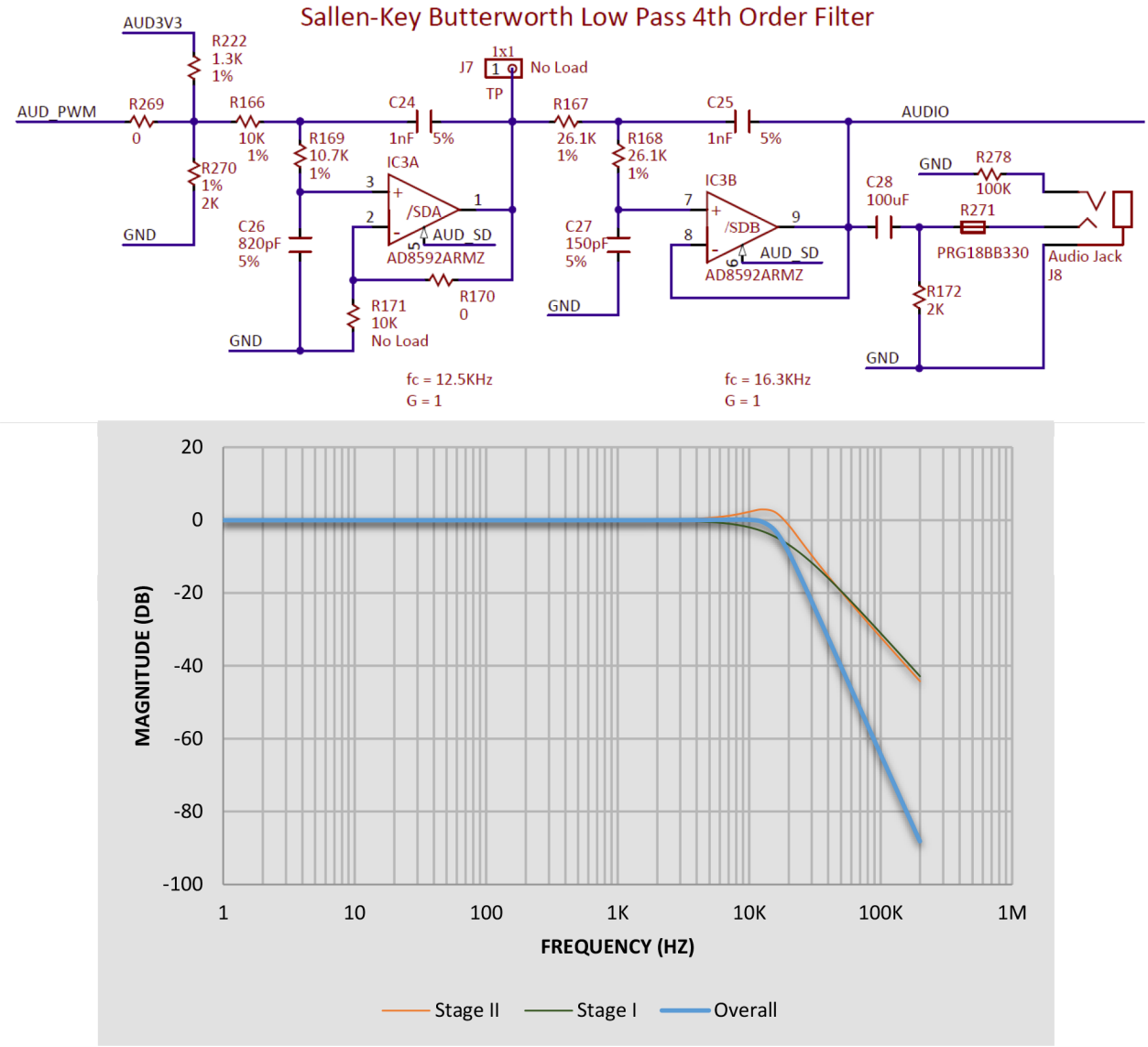

FIGURE 19 – le matériel de sortie audio de la NEXYS 4 DDR [3]

FIGURE 20 – l'architecture de la modulation

5.1 Les filtres de sur-échantillonage

5.1.1 Principe du sur-échantillonage

Pour sur échantillonné notre signal, nous allons d'abord rajouter le nombre d'échantillons à 0 nécessaire, puis appliquer un filtre passe bas sur le signal, ce qui répartira l'information sur les bits qu'on a ajouté.

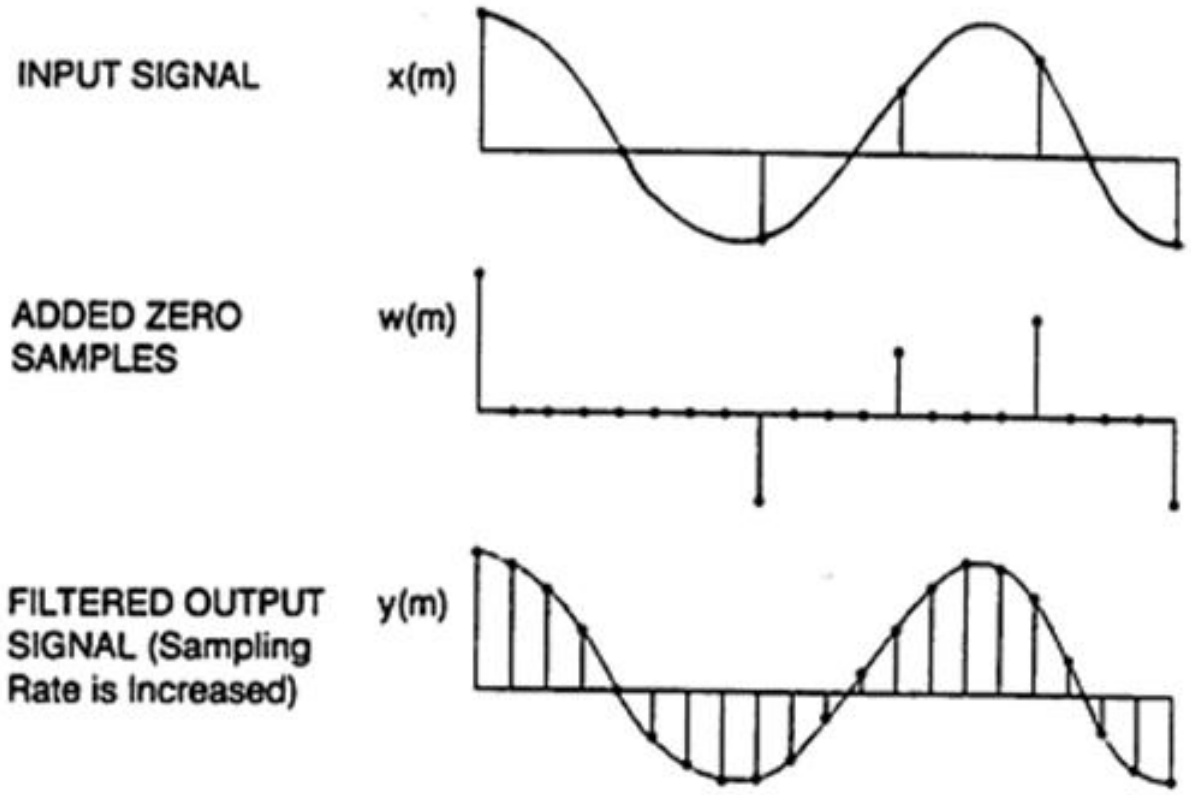

FIGURE 21 – principe du sur-échantillonage [5]

On utilise alors la même architecture qu'au 2.6 (figure 12), à la différence près que nous effectuerons un calcul sur 8 (par qu'on a 7 échantillons sur 8 qui sont nuls), et qu'il faudra faire 8 calcul par période de l'horloge d'entrée, soit un calcul par période d'horloge de sortie.

5.1.2 Validation du sur-échantillonage

Pour valider le bon fonctionnement des deux filtres de sur-échantillonnages `intfir1.vhd` et `intfir2.vhd`, on envoie un sinus plutôt dans les hautes fréquences (10kHz) et échantillonné à 39062,5Hz, et on regarde la sortie du premier et du deuxième filtre (figure 22).

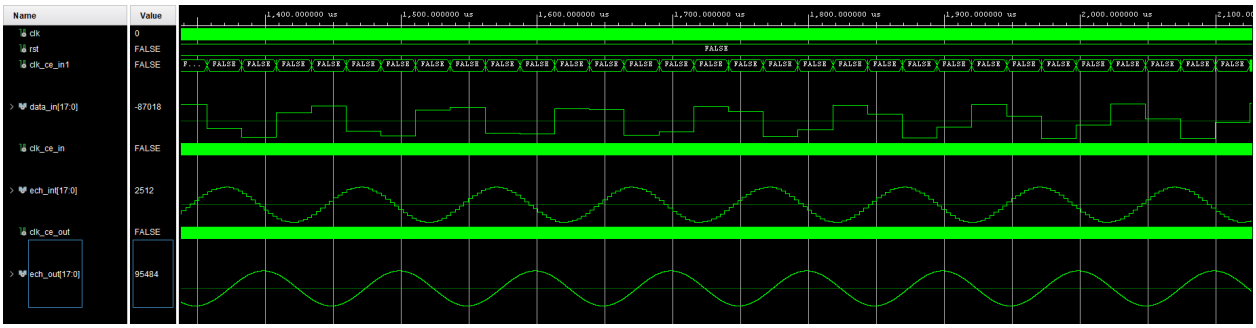

FIGURE 22 – validation de `intfir1.vhd` et `intfir2.vhd`

On retrouve donc bien un sinus à 10kHz échantillonné à 2,5MHz.

5.2 Le modulateur PDM

5.2.1 Étude du modulateur

Un modulateur PDM peut être implémenter à partir d'un signal PCM à l'aide d'un convertisseur $\Delta\Sigma$ comme celui de la figure 23. Le principe est le suivant : on utilise un comparateur numérique Q qui tronque le signal en 1 bit ($>$ ou $\leq$ à 0). Par conséquent, on aura presque toujours une erreur entre le signal d'entrée et de sortie. On va donc récupérer l'erreur qui a été faite et la considérer lors de la prochaine prise de décision. On a alors un soustracteur qui évalue l'erreur, cette dernière passe ensuite dans deux blocs retard (d'où l'ordre 2), qui seront implémenté avec des bascules D. On fini par soustraire l'erreur qu'on a fait au signal d'entrée. Le modulateur travaillera donc pour avoir le moins d'erreur possible entre le signal PCM et PDM (Y proche de 0 sur la figure 23).

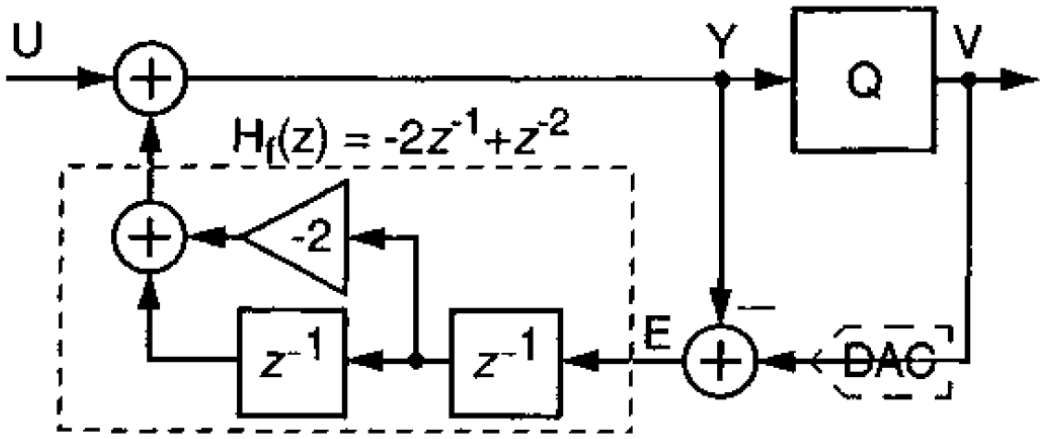

FIGURE 23 – l'architecture d'un convertisseur $\Delta\Sigma$ numérique d'ordre 2 [2]

Pour mieux imaginer le fonctionnement du modulateur, imaginons qu'on ait 0 en entrée et qu'on a un modulateur d'ordre 1 (on a alors qu'un seul bloc retard). Le comparateur prendra alors la décision de mettre un échantillons positif en sortie. Une erreur de $2^{17}-1$ sera enregistré par l'accumulateur d'erreur (comme on est en 18bits PCM). Le prochain coup le comparateur aura $-2^{17}-1$ en entrée, il générera alors un 0 en sortie. L'accumulateur d'erreur enregistrera donc une erreur presque nulle $(-(-2^{17}-1)+(-2^{17})=-1)$. Le modulateur fera donc, presque pour chaque pulsation à 1 une pulsation à 0. On aura alors en sortie une valeur moyenne à environ $0,5 \times V_{out}$, ce qui est le résultat attendu lorsque qu'on a un signal 0 en PCM.

5.2.2 Validation du modulateur `dsmod2.vhd`

Pour valider notre modulateur, nous envoyons un sinus, et nous vérifions que lorsque le sinus est sur un maximum, le modulateur envoie principalement des 1, et lorsque nous sommes sur un minimum le modulateur envoie principalement des 0. La simulation est en figure 24.

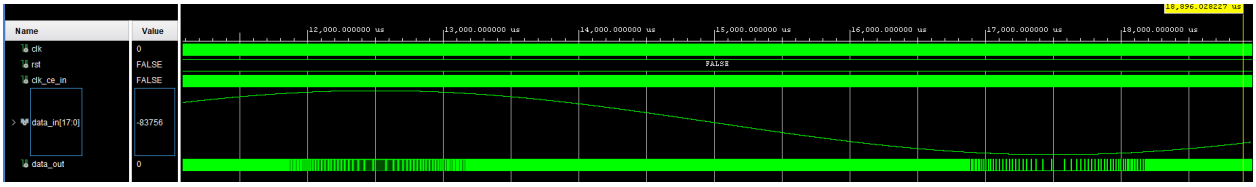

FIGURE 24 – la validation du convertisseur $\Delta\Sigma$

6 Réalisation du système

6.1 Synthèse

Nous pouvons voir sur la figure 25 que notre description VHDL est synthétisable sur Artix 7. Sur la figure 26 nous avons l'utilisation du matériel pour chaque modules, et enfin en figure 27 l'utilisation du FPGA XC7A100T-1CSG324C. Notre description ne pose alors pas de soucis à Vivado (pas de warnings), et nous utilisons qu'une petite fraction du matériel disponible.

Synthesis

Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7a100tcsg324-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

FIGURE 25 – Résultat de la synthèse

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	Block RAM Tile (135)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)
N TOP_ENTITY	691	684	2	3	4	26	1
autoVol (auto_vol)	165	104	0	0	1	0	0
dac (dsmod2)	72	69	0	0	0	0	0
fir1 (fir1)	69	93	2	0.5	0	0	0
fir2 (fir2)	87	98	0	1	1	0	0
gf (gest_freq)	18	17	0	0	0	0	0
reverb (reverb)	89	62	0	0.5	0	0	0
se1 (intfir1)	98	96	0	0.5	1	0	0
se2 (intfir2)	91	90	0	0.5	1	0	0

FIGURE 26 – Utilisation du matériel des différents modules

Resource	Estimation	Available	Utilization...
LUT	691	63400	1.09
LUTRAM	29	19000	0.15
FF	684	126800	0.54
BRAM	3	135	2.22
DSP	4	240	1.67
IO	26	210	12.38
BUFG	1	32	3.13

FIGURE 27 – Utilisation du matériel du système complet

On peut remarquer que nous utilisons presque autant de flip-flops que de LUT, ce qui est gage d'une bonne utilisation du FPGA, car on le rappelle chaque LUT est suivie d'une flip-flop : en effet ces flip-flops seraient gâchées si on ne les utilisait pas, et mettre beaucoup de LUT, donc beaucoup de logique combinatoire à la suite ralentirai le fonctionnement du système.

6.2 Implémentation

6.2.1 Utilisation

Nous avons le résultat de l'implémentation en figure 28. Cette fois ci nous avons un warning qui est le suivant :

```
[Power 33-332] Found switching activity that implies high-fanout reset nets being asserted for excessive periods of time which may result in inaccurate power analysis. Resolution: To review and fix problems, please run Power Constraints Advisor in the GUI from Tools > Power Constraints Advisor or run report_power with the -advisory option to generate a text report.
```

Ce warning signifie que Vivado n'arrive pas à estimer la consommation du signal reset de notre système. Nous avons rencontré ce problème lorsque nous avons commencé à avoir un grand nombre de modules dans notre projet. Apparemment, envoyer le reset à trop de modules en même temps perturbe Vivado. Nous n'avons pas réussis à résoudre ce warning, mais il n'est pas gênant car nous ne considérons pas la consommation dans notre étude.

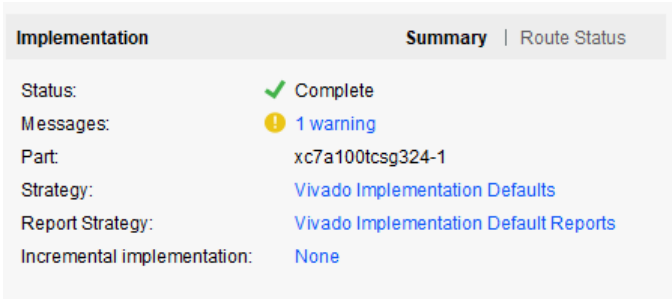

FIGURE 28 – Résultat de l'implémentation

Pour ce qui est de l'utilisation du matériel (figure 29), nous utilisons 15 LUT de moins et 2 flip-flops de plus que lors de la synthèse.

Resource	Utilization	Available	Utilization...
LUT	676	63400	1.07
LUTRAM	29	19000	0.15
FF	686	126800	0.54
BRAM	3	135	2.22
DSP	4	240	1.67
IO	26	210	12.38
BUFG	1	32	3.13

FIGURE 29 – Utilisation du matériel après implémentation

6.2.2 Timings

Nous pouvons voir les timings en figure 30. Nous n'avons pas spécialement de soucis. Néanmoins, il est bon de constater que le maillon faible dans notre architecture est le convertisseur $\Delta\Sigma$ en terme de marge de temps de setup. Si in souhaitait travailler à plus haute fréquence, il faudrait alors améliorer ce module. Nous avons d'un autre côté pas de problèmes avec les temps de hold et de problèmes avec l'horloge.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3.250 ns	Worst Hold Slack (WHS):	0.037 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2373	Total Number of Endpoints:	2373	Total Number of Endpoints:	752

All user specified timing constraints are met.

FIGURE 30 – Timings de l'implémentation

Conclusion

Nous avons dans ce projet pu expérimenter le traitement numérique du signal audio. Nous avons aussi vu la réalisation de filtres numériques, la notion de sur-échantillonnage et le traitement du signal avec l'application d'un gain automatique. Nous enfin mis en oeuvre un modulateur PDM, relatif à la notion de conversion de codage du signal.

Ce projet nous a donc permis de mettre en pratique nos connaissances théoriques et de découvrir de nouvelles façons d'utiliser un FPGA.

Annexes

Les fichiers VHDL, les tests bench et le fichier de contraintes sont disponibles sur GitHub à l'adresse : github.com/Louis-GUENEGO/NEXYS4ddr_microphone

Références

[1] Logiciel : Iowa Hills FIR Filter Design.

[2] Understanding Delta-Sigma Data Converter, Richard Schreier & Gabor C. Temes.

[3] Nexys4 DDR™ FPGA Board Reference Manual.

[4] Datasheet : ADMP421.

[5] THD STUDIO : Convertisseur Numérique-Analogique (thdstudio.com).

[6] Divider Generator v5.1, LogiCORE IP Product Guide, Vivado Design Suite.

[7] Datasheet : DSP Artix 7.

[8] Understanding PDM Digital Audio, Thomas Kite.