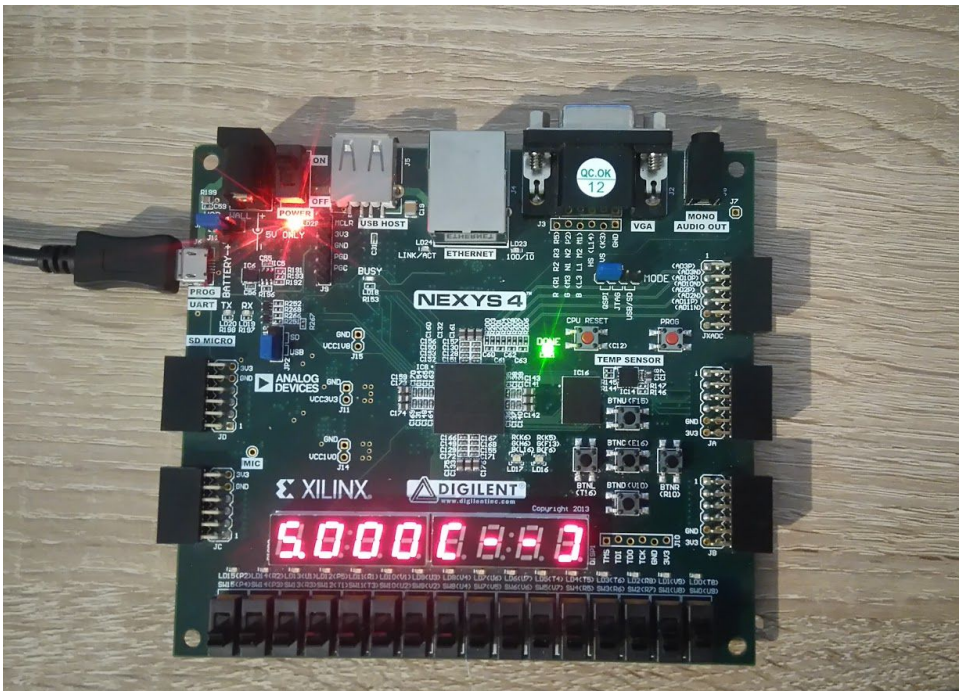
Projet conception numérique

PR209

Projet Lecteur AUDIO



Enseignant : Christophe Jégo (cjego@ipb.fr)

Rédigé par Martin AUCHER et Kevin PEREZ (SEE 2A)

Sommaire :

1) Introduction

Le sujet de ce projet est de mettre en place un lecteur de fichier audio sur une carte de développement Digilent Nexys 4 équipée du FPGA Artix-7 dans le cadre du cours PR209 “Projet de conception numérique”. Le projet devra répondre au cahier des charges donné en partie 2. Le but étant de lire un fichier audio à partir du FPGA en utilisant les connaissances acquises en cours et nos connaissance personnelles.

Manuel de l’utilisateur et code source : https://github.com/arkamor/PR209_MP3

2) Cahier des charges : Le lecteur AUDIO

Le prototype devra au moins satisfaire les exigences suivantes :

- Fréquence d'échantillonnage : **44100 Hz**
- Précision : **11bits**
- Réglage du **volume**
- Gestion **Play / Pause**
- Lecture **normale** / Lire **à l'envers**
- Bande audio **rechargeable** à partir d'un **PC** / **Liaison série**
- Affichage du **temps de lecture** / état du lecteur
- Affichage sur **afficheur** 7 segments
- Commande par **5 boutons** poussoirs
- Fonctionnement du projet sur la carte **Digilent Nexys 4**

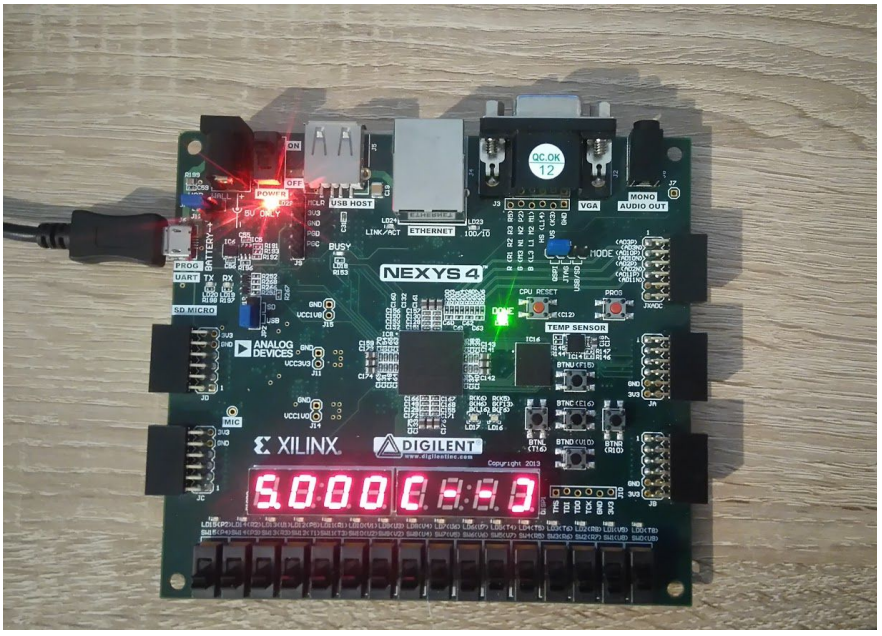


Figure 1 : Carte Digilent Nexys 4

3) La stratégie d'architecture adoptée

a) Architecture générale :

L'architecture globale du projet a été divisée en plusieurs modules simulables individuellement. Cette découpe du projet a permis de mieux structurer le projet et de faciliter le travail en binôme. Par ailleurs, l'utilisation de git a permis de partager les fichiers facilement tout en intégrant un versionning des fichiers en cas d'erreur avérée.

b) Une FSM pour l'interface utilisateur :

Les circuits numériques sont décomposés en deux grandes familles : les circuits combinatoires ainsi que les circuits séquentiels. Dans le cas d'une architecture de projet complexe, les deux familles sont amenées à travailler de paire pour réaliser la fonction souhaitée. Dans le cas d'un circuit sous FPGA on va chercher à regrouper sous différents états des circuits combinatoires. On va ainsi se retrouver avec une FSM (Finite State Machines) qui sera perçue comme un sous-système permettant de générer des états de sorties logiques.

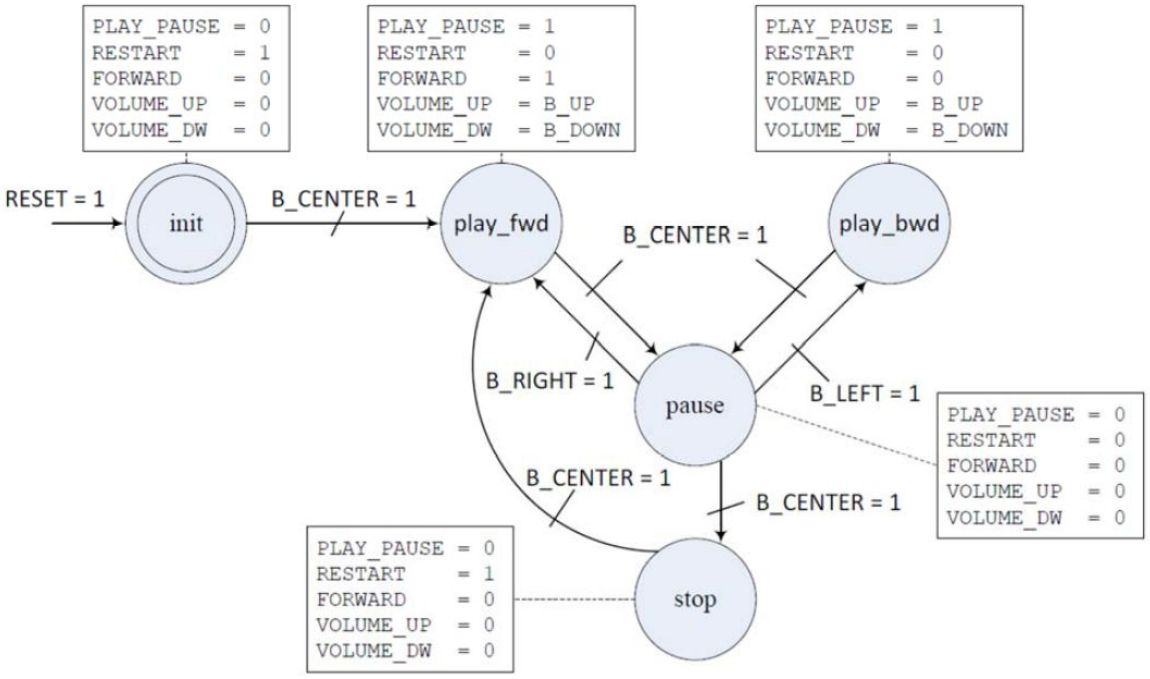


Figure 2 : représentation de la FSM

L'état actuel et l'état futur est stocké dans une variable interne de type *form_state*. Ce type peut prendre 5 valeurs différentes pour les 5 états possibles du système. Voici une représentation d'une énumération UML du type *form_state* dans laquelle on voit apparaître les différents états :

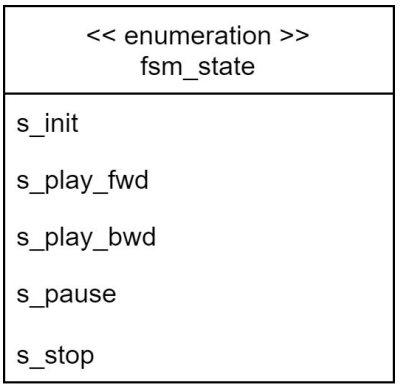


Figure 3 : représentation UML du type fsm_state

Les flèches *(sur la figure 3)* qui relient chaque état entre eux permettra de déterminer la condition de changement vers l'état suivant stocké dans la variable *next_state*. Cette flèche sera représentée en VHDL par une condition if. Dans le cas présent, la FSM permet de contrôler l'interface homme-machine du système. Les conditions de changement des états sont donc définies par des appuis sur les différents boutons de la carte.

Afin d'éviter les rebonds intempestifs lors de l'appui sur les boutons, l'utilisation d'un anti-rebond est nécessaire pour ne pas fausser l'état désiré de la FSM. Pour ce faire, le projet est composé d'un module spécifique *(BP.vhd)* qui a pour objectif d'empêcher les changements rapides d'état des boutons. Les 5 boutons qui sont utilisés et spécifiés dans le cahier des charges seront alors instanciés sur ce module. Le principe de base est de mettre en place un compteur sur front montant d'horloge du FPGA pour attendre quelques millisecondes avant de prendre en compte un nouvel état du bouton.

Les informations qui seront affichées sur les 7 segments dépendra de l'état de la FSM.

c) PWM, la gestion de l'audio :

Cette partie est divisée en 4 modules et d'un top level pour la gestion de l'audio. L'objectif du module global (le top level) est de générer une sortie binaire qui sera ensuite amené à un filtre de Butterworth permettant de transformer le signal

binaire en signal audio analogique. La carte pourra ainsi être connectée via la prise Jack à des écouteurs ou des hauts-parleurs.

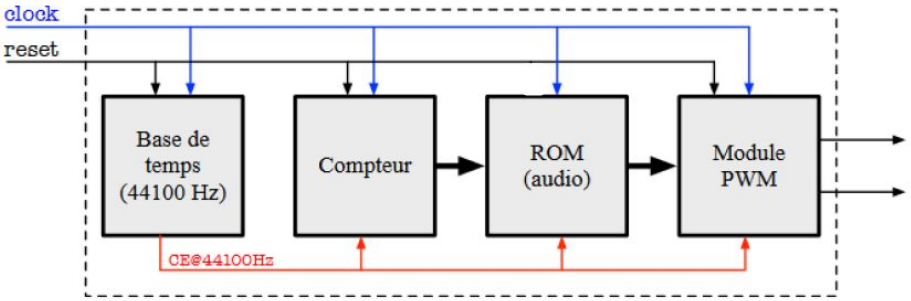


Figure 4 : module PWM de la gestion de l'audio

La fréquence d'échantillonnage précisée dans le cahier des charges est de 44100 Hz sur une fréquence d'horloge du composant qui est sur la carte de 100 MHz. Il est nécessaire de réaliser un module *(CE_gen_44100.vhd)* permettant de diviser la fréquence d'horloge pour générer une horloge à la fréquence d'échantillonnage. L'utilisation d'un compteur permet d'atteindre la fonction souhaitée. Le calcul permettant d'obtenir la valeur maximale du compteur est la suivante :

$$\frac{100\ MHz}{44\ 100Hz} = 2267{,}5$$

Lorsque le compteur atteindra 2266, le signal de sortie passera à 1, ce qui aura pour effet de générer une impulsion de 1 tic d'horloge à environ 44100 Hz. Cette base de temps à 44100 Hz sera ensuite utilisée pour les 3 autres modules de la gestion de l'audio.

Le compteur permet de sélectionner l'index désiré dans le stockage des données. Dans un premier temps, le stockage des données a été effectué dans une ROM sous forme de module VHDL. La donnée est alors transmise dans le module PWM qui a pour objectif de convertir une amplitude en signal binaire modulé (PWM).

d) Stockage des données :

```
21  TYPE ram_type IS ARRAY (0 TO 2**18-1) OF SIGNED (10 DOWNTO 0);
22  SIGNAL memory : ram_type := (others => TO_SIGNED(0,11));
```

Figure 5 : Code VHDL de la représentation des données

Les données sont stockées dans une mémoire RAM émulée au sein du FPGA. Chaque échantillon est stocké en RAM sous forme signée de 11 bits. La mémoire fait une taille de 2^{18}-1 mots de 11 bits soit 262143 mots. La taille de cette mémoire représente un temps d'écoute de :

$$\frac{2^{18}\ bits}{44\ 100\ Hz} \approx 5{,}94 \text{ secondes}$$

Pour télécharger les fichiers audios dans le FPGA, la liaison série est utilisée. Un logiciel en Python permet de téléverser directement la source audio au bon format dans la mémoire RAM du FPGA. Les sources audios peuvent provenir d'un fichier stocké sur le PC ou sur Youtube.

e) Affichage 7 segments :

L'affichage des données pour l'utilisateur se fait par le biais des afficheurs 7 segments de la carte Nexys 4.

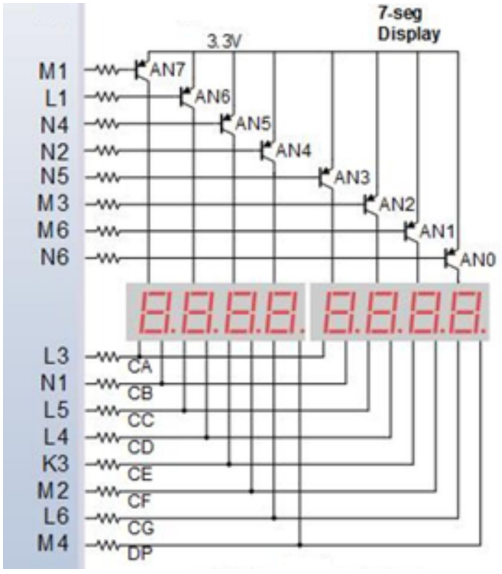


Figure 6 : représentation schématique de l'afficheur 7 segments

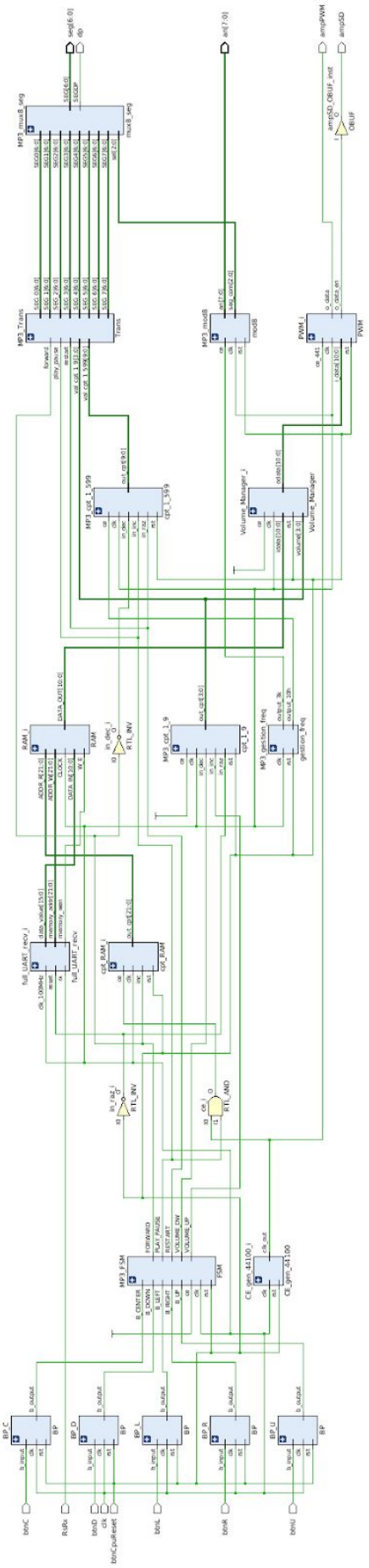
La valeur des segments ne peut pas être modifiée séparément pour chaque afficheur. Cependant, on utilise la sélection au niveau de l'anode pour permettre de sélectionner sur quel afficheur nous voulons appliquer la valeur écrite sur Cx et DP. En utilisant la persistance rétinienne, nous pouvons pouvons allumer un afficheur pendant 10 ms avec la valeur désirée en éteignant tous les autres, puis allumer le second pendant 10 ms et éteindre les autres etc… Pendant les 10 ms durant lesquelles les segments sont allumés, notre oeil garde l'information lumineuse ainsi nous voyons les afficheurs allumés correctement avec une luminosité légèrement plus faible. Le signal à 10 Hz est généré par le module *gestion_freq*. Le signal à 10 Hz est également utilisé pour incrémenter un compteur mémorisant l'afficheur actuellement utilisé. Le module *Transceiver* permets de générer les 8 digits au format 7-segments, un multiplexeur va envoyer l'information correspondante à l'afficheur actuellement utilisé.

4) Le schéma hiérarchique

Le schéma hiérarchique ci-contre définit l'architecture globale du projet final. On voit en entrée les boutons poussoirs sur la gauche du schéma. Ils permettent à l'utilisateur d'interagir avec le système.

Les signaux des boutons poussoirs passent ensuite dans des blocs anti-rebond qui permettent d'éviter au signal entrant dans la FSM de subir les rebonds des boutons physiques.

La FSM permet de gérer le fonctionnement global du système. Tous les blocs sont interconnectés entre eux par des signaux internes.



5) Explication d'une entité

```
5   entity PWM is
6       Port (
7           clk     : in STD_LOGIC;
8           ce_441  : in STD_LOGIC;
9           rst     : in STD_LOGIC;
10
11          i_data : in STD_LOGIC_VECTOR(10 DOWNTO 0);
12
13          o_data : out STD_LOGIC;
14
15          o_data_en : out STD_LOGIC
16      );
17  end PWM;
```

Figure 7 : Code VHDL de la représentation des données

Voici un extrait d'une entité utilisée dans le lecteur audio. Cette entité permet de générer un signal PWM dont le rapport cyclique varie en fonction du mot i_data.

```
50      process(i_data) is
51      begin
52          u_data <= std_logic_vector(to_unsigned((to_integer(signed(i_data))+1024),12));
53      end process;
```

Figure 8 : Code VHDL de la représentation des données

Le code ci-dessus permet la gestion de la PWM. Cette ligne permet de passer d'un nombre signé sur 11 bits à un nombre non signé sur 12 bits, sans perdre en précision.
I_data est la donnée std_logic_vector stockée dans la mémoire sous forme d'un mot de 11 bits signé.
Cette donnée est castée en signed puis convertie en integer avec la fonction to_integer. On ajoute ensuite 1024 pour centrer la valeur autour de 1024 pour pouvoir stocker la donnée dans un entier.

Valeur Signée (i_data)	Valeur non Signée (u_data)
-523	501
0	1024
996	2020

Cette valeur est ensuite utilisée par le second process qui compte jusqu'à atteindre la valeur de u_data et ainsi créer le PWM en sortie sur o_data.

6) Test d'un module

Pour effectuer le test d'une entité, nous générons tout d'abord un banc de test VHDL qui instanciera notre module sous test et le stimulera de manière à obtenir un chronogramme qui nous permet de vérifier son fonctionnement. Tous les modules sont ainsi testés afin de valider leur bon fonctionnement et ainsi assurer la bonne intégration finale du projet lorsque tous les modules seront empaquetés dans le top level.

Pour générer le test bench nous utilisons le générateur de test bench de http://doulos.com qui permet de génerer du code VHDL à partir de la définition de l'entité à tester. Après avoir ajouté les stimulis nous exécutons la simulation afin d'observer les chronogrammes.

a) Module compteur de 0 à 44099 (cpt_0_44099.vhd) :

Le compteur a plusieurs contraintes qui doivent être vérifiées par une simulation à l'aide du testbench :

- Reset : RAZ de la valeur de sortie;
- Lorsque le CE vaut 1, le compteur s'incrémente;
- Incrément sur front montant;
- Lorsque la valeur vaut 44099, le compteur repasse à 0;

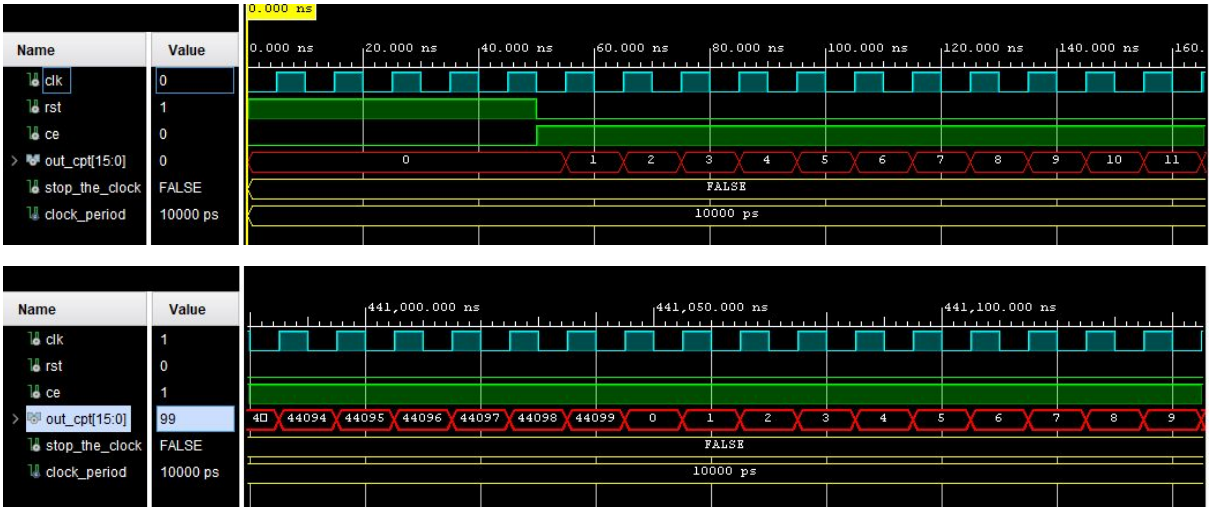


Figure 9 : Chronogramme de simulation du module compteur 0 à 44099

Comme le montre ces chronogrammes, les contraintes sont bien toutes validées à l'aide de ce testbench.

7) Présentation des résultats

L'implémentation du code final du VHDL du projet permet de générer des rapports sur les ressources utilisées sur le FPGA ainsi que sur la consommation et l'analyse des timings.

Resource	Utilization	Available	Utilization %
LUT	677	63400	1.07
FF	470	126800	0.37
BRAM	88	135	65.19
DSP	1	240	0.42
IO	26	210	12.38

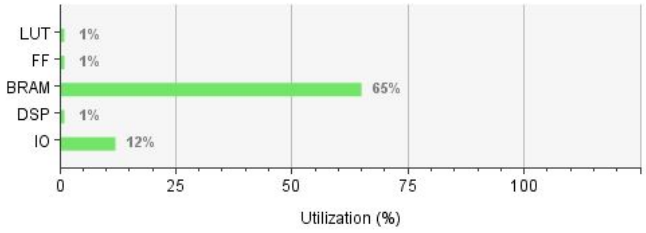


Figure 10 : Ressources utilisées

La figure ci-dessus permet de résumer les ressources utilisées. L'utilisation finale globale de chaque ressource est très faible. Sauf en ce qui concerne la BRAM puisque les fichiers audio sont directement stockés dans la Block RAM du FPGA. En considérant d'utiliser, pour ce projet, un autre système de stockage de fichier, l'utilisation de la BRAM serait fortement réduite. Il en vient donc de conclure sur la pertinence de ce FPGA pour le projet. Dans le cas d'une application industrielle, on pourrait tout à fait imaginer utiliser un FPGA avec moins de ressources internes. Seul 1% des LUT (Look Up Table) et des FF (Flip Flop) sont actuellement utilisées.

Les IO permettent de statuer sur les entrées sortie entre le FPGA et le reste du système. En l'occurence 12% des IO sont utilisées, ce qui est donc très peu comparé aux capacités de ce FPGA.

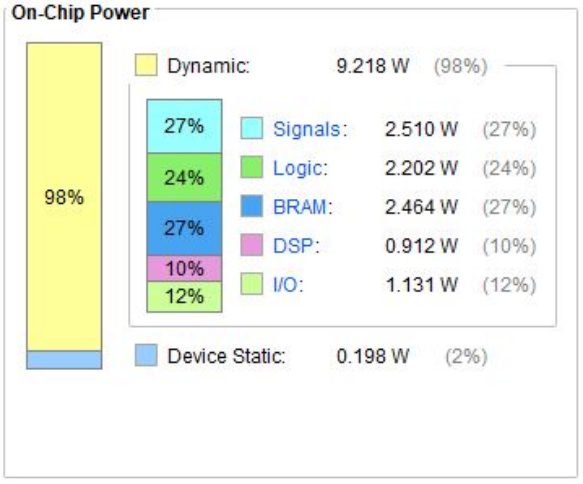


Figure 11 : Consommation du FPGA

La consommation du projet sur le FPGA est d'environ 10W en prenant en compte la charge dynamique. La consommation statique est cependant très faible : aux alentours de 200mW. Ceci s'explique par la faible complexité finale du projet. Seul 1% des LUT, 1% des Flip Flop sont utilisées. La consommation dynamique est en grande partie due au stockage des fichiers audio dans le Block RAM du FPGA.

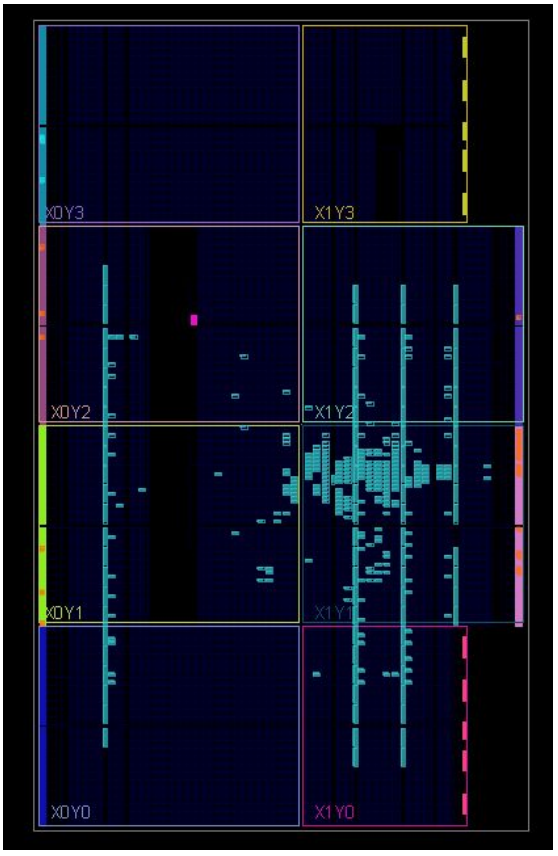


Figure 12 : Représentation de design sur le FPGA

Sur la représentation ci-dessus, on visualise bien que le FPGA n'est que très peu utilisé. La partie à droite (X1Y1 et X1Y2) représente la BRAM (Block RAM) en grand majorité utilisée par le projet final. On remarque que c'est l'une des parties qui prend le plus de place sur le FPGA.

8) Conclusion

Ce projet nous a permis de concevoir un circuit numérique embarqué dans un FPGA. Les connaissances acquises au cours de notre scolarité ainsi que nos compétences personnelles ont été utilisées pour répondre au cahier des charges fourni par notre encadrant, et plus particulièrement les cours du module EN110 pour les manipulations des mots binaires, les modules EN112 et EN225 pour l'analyse de l'utilisation du FPGA ainsi que les cours de communication pour la rédaction de ce document.

Améliorations possibles :

Nous utilisons actuellement les blocs B-RAM du FPGA pour stocker les morceaux de musique. Le design actuel utilise 65% (88 / 135) des blocs B-RAM du FPGA, il est possible d'augmenter le temps maximum du morceau en utilisant tous les blocs B-RAM disponibles.

$$temps\_actuel * \frac{total\_blocs}{blocs\_utilisés} = temps\_max$$

$$\frac{2^{18}\ bits}{44\ 100\ Hz} * \frac{135}{88} \simeq 9,12\ secondes$$

Cependant, si nous voulons augmenter le temps du morceau à 60 secondes, il est nécessaire de disposer de 2646000 cases mémoires.

L'utilisation du slot SD disponible sur la carte Digilent Nexys 4 est une des sources de mémoire disponibles. Cette fonctionnalité permettrait de stocker plus de fichiers ainsi que de réduire l'utilisation de la Block RAM du bitstream final.

La carte Nexys 4 dispose également d'une mémoire RAM de 128Mb. C'est une mémoire SDRAM CMOS pseudo-statique. Elle est arrangée en 8M cellules de 16 bits. Ce qui représenterait plus de 190 secondes d'enregistrement (8M/44100=190). On remarque que cette méthode utilise 68.75% d'un mot (11 bits utiles contre 16 bits dispo). Le temps moyen d'un morceau de musique étant de 3m49s (229s), il est possible d'optimiser encore la mémoire, en fractionnant les mots (Voir figure 13), le temps d'enregistrement peut, par ce biais atteindre 276s.

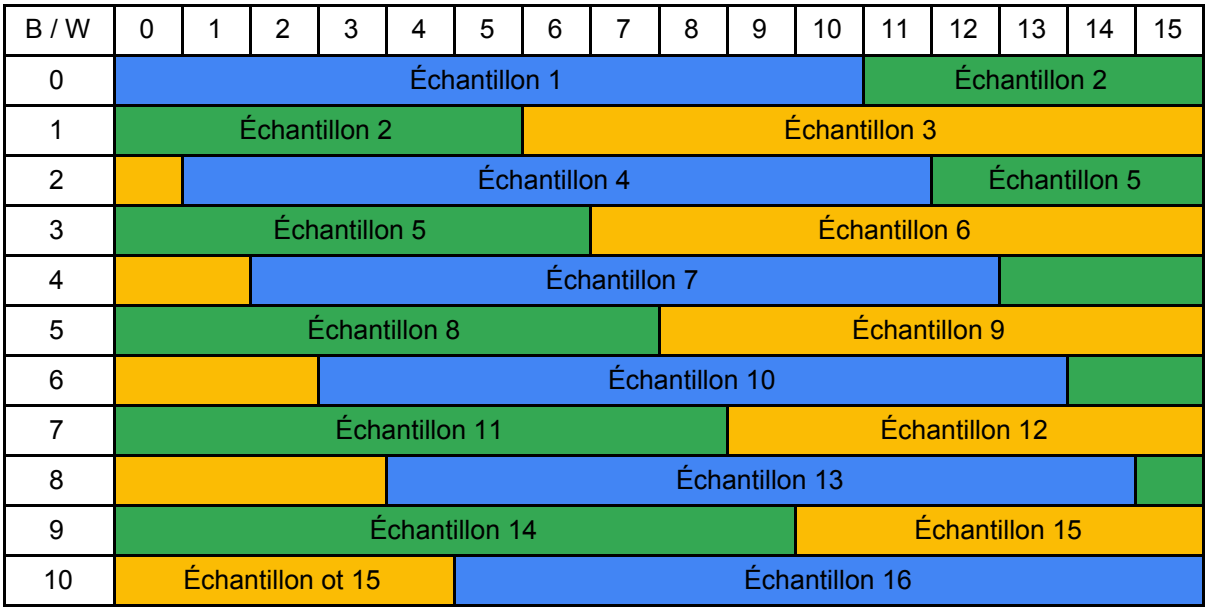


Figure 13 : Représentation de la RAM fractionnée

La réalisation d'un driver de SDRAM n'étant pas chose aisée, nous n'avons pas pu finaliser son implémentation au sein du design dans le temps imparti.

Rappel : Manuel de l'utilisateur et code source :
https://github.com/arkamor/PR209_MP3