Polytech Lille
IMA4 - 2019/2020
Projet IMA3/4

Elèves
Stephen Andriambolisoa - Robin Coubelle
Quentin Bargibant - Simon Lecoutère

# Rapport de projet S8

# P23 - Parure augmentée à but purement narcissique

Encadrants
M. Xavier Redon
M. Alexandre Boé
M. Thomas Vantroys


# Table des matières

**Introduction** **3**

**1.Le projet** **4**

1.1. Approche 4

1.2. Résumé du S6 & S7 4

**2. Réalisations du S8** **6**

2.1. Mise au point 6

2.2. Création des PCB 7

2.2.1. FPGA 7

2.2.2. USB to JTAG 9

2.3. Développements VHDL 10

2.3.1. Convertisseur analogique-numérique 10

2.3.2. Transformation de Fourier rapide 19

2.3.3. Panneau leds 21

2.3.4. Bluetooth 25

2.3.5. Application mobile 27

2.4. Résultat final 28

**Bilan** **30**

**Sources** **31**

**Annexes** **32**

# Introduction

Dans le cadre de la formation d'ingénieur Polytech'Lille du département IMA, il nous a été demandé de réaliser un projet par groupe d'élèves qui se déroulera sur la période du semestre 6, 7 et 8. Ce projet a pour but de développer nos capacités d'études et de mettre en pratique nos connaissances techniques acquises dans l'enseignement supérieur. En effet, cela nous permet de nous mettre dans les conditions du métier d'ingénieur.

Ce projet a pour but de concevoir un vêtement (robe, jupe, veste ou accessoires) agrémenté d'animations lumineuses. Ce vêtement serait le complément idéal pour des occasions spéciales (anniversaires, mariages, soirées) afin d'attirer l'attention sur l'utilisateur et d'intriguer les personnes aux alentours.

Dans ce rapport, nous présenterons le projet de façon générale. Puis, nous reviendrons sur ce qui a été réalisé aux semestres 6 et 7. Ensuite, nous verrons les évolutions apportées au cours du semestre 8 se concluant sur la version finale de nos travaux. Enfin, nous établirons un bilan et un retour sur ce projet de groupe.

# 1.Le projet

Dans un premier temps, nous allons revenir sur les débuts de ce projet réparti sur les 2 semestres précédents en évoquant les directives entreprises et les points clés à retenir pour comprendre son évolution.

## 1.1. Approche

Lors de la prise en main de ce sujet, différentes étapes de réflexions et de pistes ont été évoquées dans le but de se positionner sur l'objectif que l'on souhaitait atteindre. En effet, avec un projet intitulé "Parure augmentée à but purement narcissique", beaucoup de possibilité semblait s'offrir à nous avec comme plus-value majeure, des capteurs fixés capables d'interagir avec des rangées de leds.

A partir là, nous avons pu segmenter les différents points que nous souhaitions aborder dans ce projet. Tout d'abord, la capacité d'interaction avec l'environnement, où les capteurs devront récupérer le signal sonore ambiant tel que dans la musique pour effectuer des animations sensibles au son. Ensuite, la création de diverses animations fixes dans le temps pour augmenter le choix des visuelles, sur un vêtement tel qu'un tee-shirt. Et surtout, de la mise en place d'une communication à distance avec tout le matériel afin que l'utilisateur puisse se divertir autant qu'il le souhaite.

Avant de se lancer dans le développement, nous nous sommes penchés sur le marché actuel et des potentiels concurrents. Cette phase nous a permis de connaître l'état des produits disponibles à l'achat et de nous en apprendre plus sur les technologies employées pour de multiples utilisations. De part ces analyses, nous avons pu élaborer notre scénario d'usage.

Nous avons pu à partir des informations consultées établir notre cahier des charges face aux concurrents. Notre produit étant destiné au grand public, plusieurs critères doivent être respectés. Le produit devra être simple d'utilisation et agréable à porter pour son utilisateur, ainsi que d'assurer sa sécurité face aux circuits électroniques. Nous allons donc réaliser un prototype qui devra tendre vers nos objectifs et dans la finalité vers nos contraintes.

## 1.2. Résumé du S6 & S7

Un premier prototype du projet a été réalisé durant le semestre 6 de notre cursus. Son schéma électronique est le suivant:

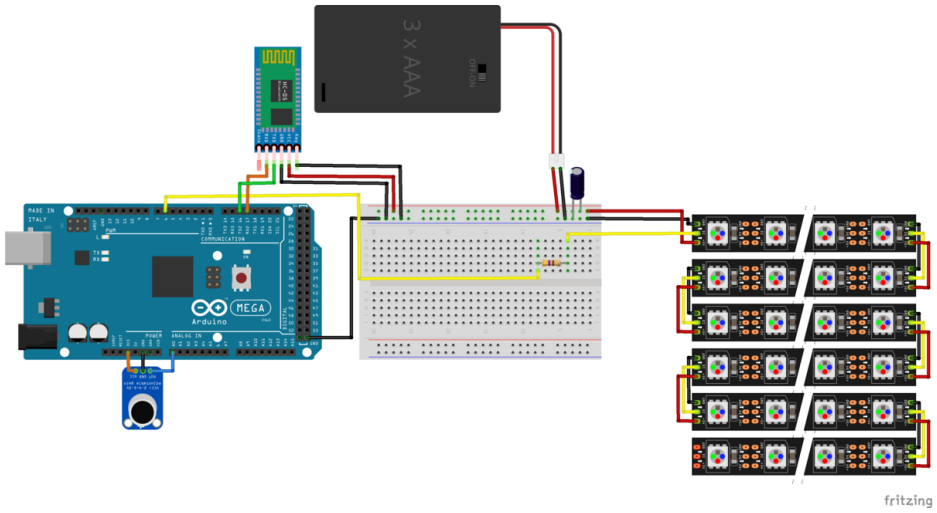



*Figure 1 - Schéma S6*

Ce prototype s'articule autour de 3 modules importants:

- 1 Microphone MAX4466
- 1 Module Bluetooth HC-05
- 1 matrice de 24*6 LEDs conçu à partir d'un bandeau de LEDs SJ-100144-2811.

Ce prototype présente plusieurs facettes intéressante selon nous, le bandeau de LEDs contrôlé par la carte Arduino permettait une grande diversité d'animations, facilement codables.
Les deux autres modules permettent d'interagir avec l'environnement (essentiellement la musique), mais également de contrôler les animations à distance, via son téléphone, grâce à la connexion Bluetooth.
Cependant notre groupe s'est attaché à une animation principale: l'égaliseur sonore, qui représente le spectre des fréquences audible dans l'environnement ambiant. Cette animation a été réalisé artificiellement par nos soins, en effet notre égaliseur sonore réagit en fonction de l'intensité du signal sonore reçu, et non en fonction des fréquences.

Nous avons alors choisi de nous attarder sur cette animation lors de notre 4ème année d'études. Après l'arrivée de Stephen et Simon, nous avons décidé de complexifier le projet pour avoir un résultat optimal pour l'animation. Nous allons récupérer le spectre de fréquences après une Transformée de Fourier Rapide (FFT). Une FFT est un ensemble d'opérations mathématiques assez lourdes, pour obtenir un résultat temps réel rapide, nous avons remplacé le microprocesseur de la carte Arduino par une puce FPGA, composant électronique permettant d'effectuer un grand nombre d'opérations rapidement.
Utiliser un FPGA dès le semestre 7 s'est finalement révélé très ambitieux: devant notre manque de connaissances, ce semestre a été très laborieux et nous avons eu très peu de résultats physiques. Cependant ce semestre de recherches, ainsi que le module CNP de la formation IMA, nous a permis d'acquérir de nombreuses connaissances indispensables à la réalisation du projet nous permettant de mieux appréhender le semestre qui a suivi.

# 2. Réalisations du S8

Dans un second temps, nous allons aborder toutes les réalisations apportées au déroulement du projet lors du semestre 8. L'évolution de ce dernier est disponible dans son intégralité sur le wiki à disposition de notre groupe.

## 2.1. Mise au point

Suite à la soutenance du S7, nous décidons de ne plus gérer les LEDs à l'aide d'un microcontrôleur. Nos encadrants estimaient que ce n'était pas la bonne direction à prendre en continuant comme cela. En effet, nos réalisations n'étaient pas convenables pour un projet de quatrième année. C'est pourquoi, nous allons nous appuyer sur les bases acquises lors du semestre 7, notamment en Commande Numérique & Programmable, pour la suite.

Avant de penser à la structure de notre parure personnalisable, nous voulons aboutir sur un véritable égaliseur avec un FPGA. Il semblerait que certains d'entre eux possèdent des IP permettant d'effectuer des ADC ou des FFT, que nous pourrons coder et personnaliser. Nous nous penchons ainsi vers une carte à réaliser. Cependant, nous voulons également expérimenter et tester nos codes VHDL pendant la production de notre carte. C'est pour cela que nous aurons une board Basys 3 mise à disposition par nos professeurs. Nous l'avons choisie car par rapport à ce que nous avions, c'est celle qui se rapproche le plus de notre FPGA en terme de fonctionnalités qu'on souhaite lui apporter. La board nous servira également à effectuer des démonstrations.

Nous avons donc pu répartir le travail comme l'indique l'organigramme ci-dessous, avec comme référent principale la personne mentionnée en première.

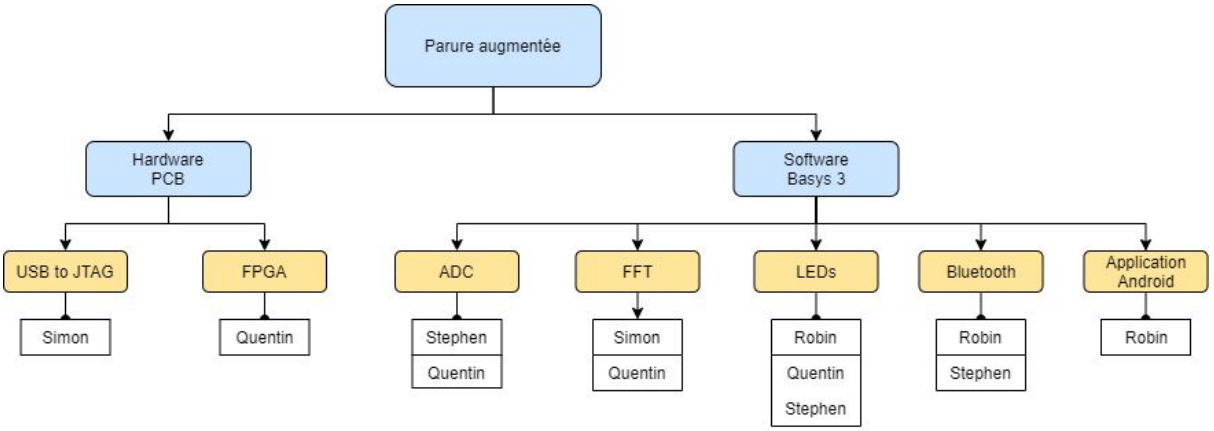



*Figure 2 - Gestion de notre projet*

L'objectif de ce semestre est d'effectuer une véritable analyse fréquentielle du signal sonore reçu par le microphone. Le FPGA pourra traiter le signal et commander les LEDs en fonction des fréquences correspondantes. Nous ajouterons également une application Android qui permettra à l'utilisateur de choisir un motif sur le panneau de LEDs lorsque l'égaliseur est

actif. Il faudra donc gérer un module Bluetooth. Le PCB sera alors spécifique à notre utilisation du FPGA. C'est pourquoi le traitement du signal se fera grâce au logiciel Vivado dans un premier temps, pour la Basys 3, puis nous l'adapterons à notre propre carte. Une des difficultés sera de créer les interactions entre les différents codes.

Pour démarrer notre projet, voici le matériel acquis, même pendant la période de confinement :

- 1 Microphone MAX4466
- 1 Module Bluetooth HC-05
- 1 Panneau de 144 LEDs (dont une ligne de 24 LEDs hors-service)
- 1 Board Basys 3
- Câbles USB et fils
- 1 Breadboard
- Arduino ATMEGA2560 (uniquement pour configurer le module Bluetooth)

## 2.2. Création des PCB

Pour notre application, nous réaliserons une carte électronique spécifique et optimisée par rapport à nos besoins. Pendant la réalisation, nous avons rencontré une difficulté concernant la programmation de la puce sur cette carte, qui a permis de scinder la réalisation de la carte en deux parties bien distinctes qui finalement, rassemblées, permettra de réaliser notre application.

### 2.2.1. FPGA

Une carte électronique dont le composant principal est un FPGA s'articule autour de 4 pôles majeurs:

- La puce FPGA
- Sa mémoire de programme
- Les entrées/sorties du système
- L'horloge

Après la remise à zéro du projet, il a été nécessaire de choisir à nouveau les composants des différents pôles.

Choix des composants

Premièrement, le choix du FPGA a été compliqué pour notre groupe. Ayant de faibles connaissances concernant en son sujet, nous avons été confronté à l'immensité de l'offre présente sur le marché. Nous avons décidé de nous tourner vers les éléments que nous avons déjà utilisés en cours: les FPGA Xilinx, qui peuvent être programmés via les logiciels Vivado et ISE.
Notre application ne nécessite pas une puce très performantes, les deux premières gammes, Spartan et Artix de la marque peuvent alors correspondre à nos attentes. Cependant, les dernières versions de ces puces contiennent un très grand nombre d'entrées, et leurs positions nous empêchent de les souder nous mêmes. La solution à ce

problème a été de choisir un FPGA parmi une série plus ancienne du constructeur: la série 3.
Notre application possède peu d'interfaces, en suivant la documentation Xilinx, nous avons choisi la famille Spartan-3E et plus particulièrement le XC3S500E-4VQG100C.

La mémoire de programme est un élément indispensable pour notre application, en effet, sans celle-ci la carte électronique doit être reprogrammée via logiciel à chaque allumage. Après avoir étudié la documentation et en s'inspirant des boards de programmation existantes, nous avons choisi d'utiliser la mémoire PROM XCF04SVOG20C de 4Mb, qui permet de stocker le bitstream de notre FPGA.

Les interfaces sont définies par notre application, comme évoqué précédemment, nous utiliserons des modules que nous avons déjà en notre possession:

- La matrice de led créée au S6 à partir du bandeau SJ-100144-2811
- Le module Bluetooth HC-05
- Le microphone MAX4466

Ce dernier composant nous envoie une donnée analogique, pour exploiter ses données sur la puce, il est nécessaire que le signal soit converti via un ADC. Nous utilisons le ADS7868IDBVR, notre donnée sera récupéré sur 8 bits, transmis en série. L'égaliseur sonore affiché sur notre matrice est composé de seulement six pixels de hauteur, il n'est donc pas nécessaire d'avoir un ADC plus précis pour notre application.

Nous utiliserons une horloge classique de 100 MHz de référence CB3LV-3I-100M0000.

Réalisation de la carte

Secondement, la réalisation des cartes électroniques a été faite sous le logiciel Altium. Pour réaliser les branchements de la puce FPGA, notamment avec la mémoire de programme, nous avons choisi de nous inspirer de boards existantes. Après quelques recherches, nous avons trouvé une board utilisant la même puce que la nôtre, et étant très bien renseignée : Open3S500E, on utilisera cette board comme référence.
La plupart des cartes de développement existantes sont programmés via un câble USB, cependant les constructeurs cachent les fichiers comprenant la connexion USB-FPGA. En effet, la programmation du de la puce, selon les schémas constructeurs, se fait tout le temps via une arrivée de donnée JTAG. Pour éviter l'achat d'un câble de programmation USB-JTAG, et pour rendre la programmation de notre carte plus accessible, nous avons réalisé le circuit de conversion USB-JTAG (voir la partie 2.2.2.).

Les fichiers Altium fournit par Xilinx sont différents des blocs classiques. En effet les puces FPGA, contenant un grand nombre de pins, sont illisibles en tant que bloc unique Altium. Pour pallier ce problème, les FPGA sont représentés en différentes parties, désignant chacune une fonction de la puce (voir *Figure 3 - Exemple bloc*).

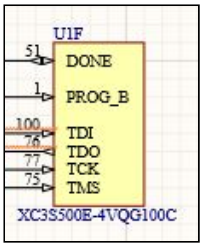



*Figure 3 - Exemple bloc*

Notre application possède très peu d'interfaces (trois entrées/sorties), et malgré le choix d'un FPGA en possédant peu, nous n'utilisons qu'une infime partie des ports de la puce (une centaine). Après concertation avec les encadrants, il a été décidé de ne pas laisser libres les ports non utilisés. Des banques d'entrées/sorties ont donc été ajoutées, pour éventuellement permettre de rajouter des interfaces, ou d'utiliser la carte pour une autre application.

La réalisation du routage de la carte nous a présenté quelques difficultés, la finesse des nombreux ports du FPGA et la nécessité de la propreté de l'alimentation (et donc de la présence des capacités de découplage au plus proche de la puce) nous a obligé à retravailler notre routage.

Les schémas électroniques ainsi que le routage de la carte sont disponibles en annexe ainsi que sur le wiki du projet (annexe 1 à annexe 4).
On reliera les ports JTAG d'entrées aux sorties JTAG de notre seconde carte.

### 2.2.2. USB to JTAG

Pour envoyer les données des programmes que nous allons créer de l'ordinateur jusqu'à la mémoire du FPGA, nous allons utiliser une connexion grâce à un bus JTAG. On rappelle que le JTAG est une norme IEEE initialement utilisée pour résoudre les problèmes de fabrication des cartes électroniques ainsi que pour déboguer le code d'un processeur. Dans notre cas, le JTAG permettra de programmer notre FPGA via le port JTAG.

Il y a deux manières de transférer ses données, en utilisant un câble USB->JTAG ou en intégrant directement un module FTDI au PCB. Nous avons opté pour la seconde option qui est moins coûteuse (environ 60€ pour un cable JTAG contre 25€ pour un module), plus pratique car les câbles USB sont facilement mis à disposition et le transfert des données se fait plus rapidement.

Le module que nous avons choisit d'intégrer à notre PCB est un FT2232H Mini Module. Comme précédemment, nous avons décidé de le réaliser sous altium en achetant les pièces séparément. L'ensemble des composants se trouve sur notre Wiki.


*Figure 4 - FT2232H Mini Module*

Réalisation du PCB

Pour réaliser le PCB, nous nous sommes inspirés du schematic donné dans la documentation du Mini Module. Ici, notre circuit intégré comporte beaucoup de broches. Cependant, le bus JTAG se compose vraiment de quatre signaux logiques, appelés :

- TMS, Signal d'activation de la communication JTAG,
- TCK, Horloge,
- TDI, Entrée des données,
- TDO, Sortie des données.

Grâce à la datasheet du module FT2232H qui sera l'élément principal, on peut voir que ces signaux sont reliés aux broches 16,17, 18 et 19 (page 14). Ce qui permet de réaliser notre schematic de manière à n'avoir accès qu'à ces 4 broches. Pour ajouter chaque composant, il a fallu télécharger Altium Library qui permet de trouver un composant avec son empreinte ou sa référence fab.

Une fois le schematic du JTAG terminé, nous avons réalisé le routage du PCB. Le principal soucis était le grand nombre de connexions qui nous a obligé à faire notre PCB sur les deux faces de la cartes en y insérant un plan de masse. Enfin, Il devra être soudé sur la carte FPGA personnalisée avec les connexions nécessaires.

Les schémas électroniques ainsi que le routage de la carte sont disponibles en annexe ainsi que sur le wiki du projet (annexe 5 et annexe 6).

## 2.3. Développements VHDL

Le travail qui va suivre a été réalisé avec l'environnement Vivado proposé par Xilinx et nous permettra de vous présenter le premier prototype sur la Basys 3.

### 2.3.1. Convertisseur analogique-numérique

Paramétrage de l'IP :

L'objectif de notre convertisseur analogique-numérique (ADC) est de récupérer le signal sortant du microphone pour le convertir en un signal numérique pouvant être traité par la suite. Pour se faire, nous utilisons l'IP XADC. Avant toute chose, il faut savoir que lors de la création du projet, nous déclarons le FPGA *xc7a35tcpg236-1* et non la board Basys3, d'après cette source. Après avoir tout de même tenté de déclarer la Basys3 pour l'utilisation de notre convertisseur, la génération de notre IP ne s'arrêtait pas.

Afin d'effectuer une démonstration, nous avons paramétré au plus simple notre convertisseur pour mesurer une tension. En effet la sortie du micro délivre des variations de tensions qui vont être traitées par l'entrée de notre ADC.

Nous utiliserons alors le mode « Single Channel » avec une interface DRP (Dynamic Reconfigurable Port) pour convertir notre signal d'entrée. Nous nous intéresserons uniquement à la chaîne VAUXP6/VAUXN6. Une chaîne correspond à une paire de vecteurs pour lire une différence de potentiel à faible bande-passante lorsqu'elle reçoit un signal analogique (d'où le P de positive et N de negative). Cette chaîne correspond aux pins J3/K3 soit les ports JXADC 1 et 7 de la Basys3 d'après ce mapping.

Tous les paramètres de calibrage, de multiplexages externes et d'alarmes seront décochés. Concernant le fichier de simulation, nous choisissons dans un premier temps l'emplacement par défaut. Voici le résultat de la génération de notre IP :

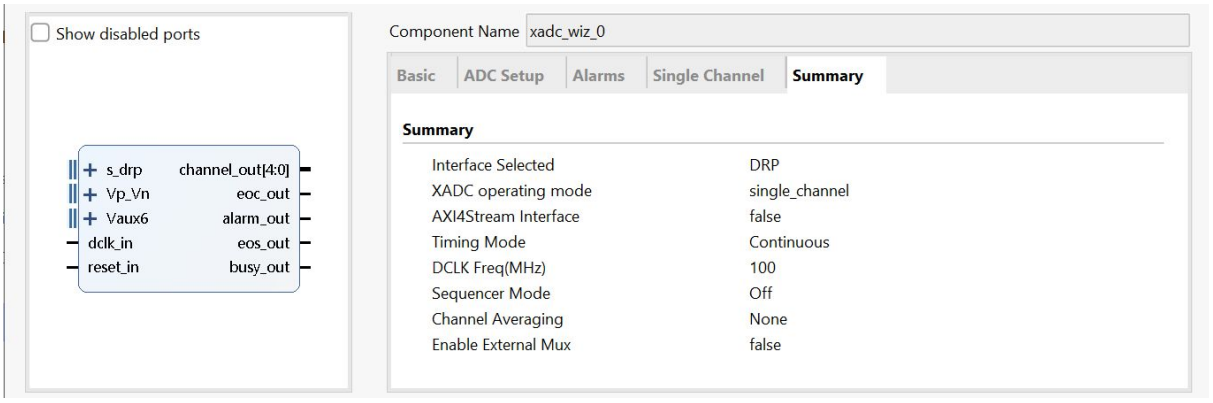



*Figure 5 - IP xadc_wiz_0 paramétré*

Instanciation de notre composant :

Nous avons alors un fichier *xadc_wiz_0.vhd* qui s'est créé. Il correspond à notre IP personnalisé pour notre utilisation. Il est en mode *read-only* car il est uniquement configurable lors de la « customisation » de l'IP. Nous n'aurons donc pas à configurer les registres de notre xadc_wiz_0.

Dans la façon dont nous l'avons paramétré, nous pouvons voir que seule la chaîne 6 nous intéresse (voir *annexe 7 - xadc_wiz_0 instancié*). Ainsi, l'entrée de l'IP et du xadc_wiz_0 sont connectés via vauxp6 et vauxn6.

Après avoir déclaré le composant xadc_wiz_0 dans notre fichier VHDL, nous pouvions créer les signaux nécessaires à la création de notre port map, voir *annexe 8 - Signaux de notre composant*.

Nous pouvons alors créer le port map de notre propre ADC, correspondant finalement à l' *annexe 9 - Port map de notre ADC*. Le port map consiste à créer les liaisons entre différents composants pour en avoir un plus spécifique. Nous avions longuement hésité pour les

entrées VAUXP et VAUXN. En effet, dans nos premières versions, nous mettions VAUXP6 et VAUXN6 déclarés directement en STD_LOGIC. Au final, nous verrons plus tard que cela va poser des problèmes, notamment avec la simulation. En ce qui concerne la sortie numérique de notre composant, MEASURED_VCCINT correspond à un vecteur de 16bits.

Nous pensons avant tout à coder en machine d'états. Ainsi nous pouvions mémoriser l'état interne de notre composant dans le registre d'état, et les déclarer comme suit :

```
type state_type is (init_read, read_waiteos,
                    read_reg00,
                    reg00_waiteos );
signal state : state_type; --next_state ?
```

*Figure 6 - Déclaration de notre machine d'état*

Première version de notre code :

Nous commençons alors par apprendre à utiliser les machines d'états finies pour coder notre ADC (voir *annexe 10 - Notre machine d'états*). Nous avons 4 états que nous avons déclaré précédemment :

- Init_read : initialisation de notre code lorsque RESET passe à 1.

- Read_waiteos : Nous attendons la fin d'une séquence de notre signal pour changer d'état. Sinon, nous restons dans celui-ci. Dans le cas où nous changeons d'état, nous réinitialisons l'entrée *di_drp* du xadc_wiz_0 à l'aide d'un ET logique et de *do_drp*. Le but est de pouvoir mémoriser cet état lorsque nous avons une nouvelle séquence. Nous rappelons que *do_drp* est la sortie de notre xadc_wiz_0 donc il s'agit d'un signal analogique déjà converti en numérique, sur 16 bits.

- Read_reg00 : c'est un état tampon permettant d'anticiper l'état futur, soit la lecture de notre donnée.

- Reg00_waiteos : c'est avec cet état que nous pouvons attribuer une valeur numérique à la sortie de notre composant. C'est MEASURED_VCCINT qui la récupérera via *do_drp* lorsque la lecture d'une séquence sera terminée. Sinon, nous mémorisons l'état dans lequel nous sommes actuellement.

Le Block Diagram de notre machine d'état est assez complexe, comme vous pouvez le voir ci-dessous :

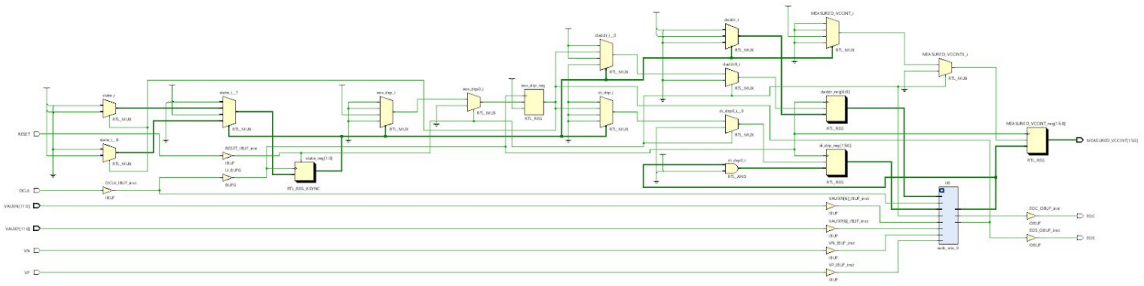


*Figure 7 - Block Diagram de notre machine d'état*

Nous vous résumons alors son organisation en un seul schéma, avant de lancer une simulation pour valider son bon fonctionnement.

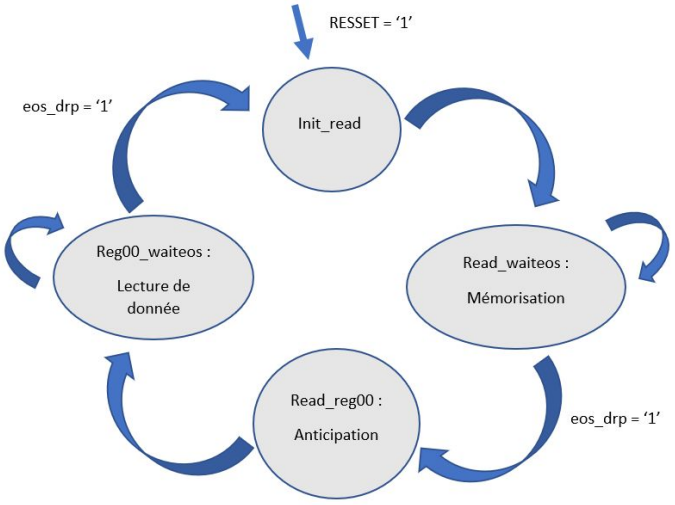



*Figure 8 - Résumé de notre machine d'états*

Simulation :

Pour lancer une simulation sur Vivado, il faut créer un testbench. Ce dernier permet de modéliser des situations particulières pour notre code. On sait que notre composant reçoit un signal analogique en entrée. Cependant, Vivado ne peut pas en créer un. C'est pour cela que nous devons générer un fichier nommé design.txt pour simuler une entrée analogique.

Lors de notre toute première instanciation, nous avions eu un problème lié aux noms des ports de notre entité, comme dit précédemment. En effet, nous avions des entiers de type STD_LOGIC en entrée : VAUXP6 et VAUXN6. On souhaitait récupérer un signal

uniquement sur une chaîne donc on ne voyait pas tellement l'intérêt de déclarer un vecteur entier. Finalement, ce n'était pas possible car notre fichier de stimulation demande la position de la chaîne dans un vecteur d'entrée. Vous pourrez retrouver comment créer un fichier de stimulation dans l'annexe 11 - Comment créer un fichier de stimulation.

Nous avons donc pu créer notre propre fichier de stimulation :

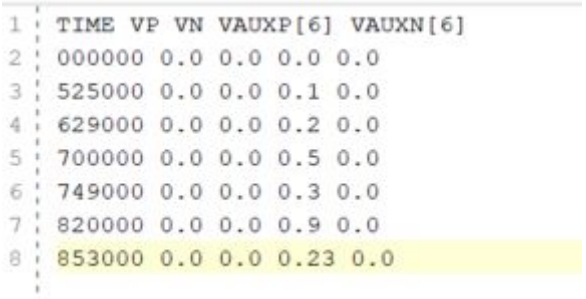



*Figure 9 - Notre fichier de stimulation*

On peut remarquer que nos paramètres VP et VN restent à 0. Nous aurions pu nous en passer mais nous l'avions ajouté lors de tests non-concluants. De plus, c'est bien la chaîne VAUXP6 et VAUXN6 qui doit recevoir le signal en entrée.

Pour créer un testbench, il faut avant tout ajouter une nouvelle source de simulation. Vu que notre design.txt est à un emplacement par défaut, il faut qu'il soit dans le même répertoire. Pour notre simulation, nous ne créons pas d'entité. Cependant, nous déclarons les composants qui vont intervenir dans notre simulation avec les même entrées et sorties. Dans un premier temps, c'est celui de notre ADC qui va nous intéresser, voir *annexe 12 - Déclaration de notre ADC dans le testbench*. Il faut ensuite créer des signaux avec le même nom que les ports du composant. Cela permettra une nouvelle fois de faire un port map, soit de connecter les composants entre eux, voir *annexe 13 - Port map de l'ADC pour le testbench*.

Ensuite, nous créons une horloge de fréquence 100 MHz ainsi qu'une initialisation de RESET à 1 pendant 0.050ms. Pour cela, il faut créer des « process » qui vont être générés pendant la simulation, voir *annexe 14 - Process de notre horloge et RESET*.

Enfin, nous devons récupérer le signal analogique de notre fichier pour vérifier la conversion effectuée par notre composant. Cependant, nous allons voir que cette étape nous aura causé bien des problèmes.

<u>Datasheets et UG480 :</u>

Pour récupérer un signal analogique, nous voulions d'abord nous inspirer d'un exemple de notre IP. Lorsqu'on essaye d'en générer un directement depuis le logiciel, l'exemple est en langage VERILOG. Ce n'était pas approprié. Nous décidons alors de faire nos propres recherches et de consulter plusieurs datasheets (voir dans **Sources** les datasheets).

Dans un premier temps, il s'avère que si le fichier de stimulation est au bon emplacement, il est généré par défaut. C'est inscrit dans la section Simulation de la datasheet de l'XADC, page 53. Cependant, nous n'obtenons rien en lançant notre simulation avec le fichier dans l'emplacement par défaut. Vivado ignore les modifications du fichier de stimulation de notre IP après l'avoir instancié. Nous décidons alors de copier notre projet et créer un IP avec un « Relative path » indiquant le chemin de notre design.txt. Malheureusement, le fichier n'est toujours pas généré en lançant une simulation.

Nous espérons alors comprendre le fonctionnement de cette simulation via un exemple indiqué sur la datasheet de l'XADC pour Séries 7, page 82 section XADC Software Support. Il s'agit de l'UG480, créé par un ingénieur Xilinx. Nous ne pouvons pas accéder directement à l'exemple car il semblerait qu'il soit confidentiel. Nous envoyons donc une demande à Xilinx, avec notre propre compte, pour se le procurer. Malheureusement, le testbench de cet exemple ne possède aucune gestion du fichier. Vous pouvez le retrouver dans le GIT, nommé ug480_tb.vhd. Dans le readme.txt de l'exemple, il faut créer une simulation via la TCL Console à l'ouverture du logiciel. Cependant, une erreur persiste nous poussant à nous diriger vers une autre méthode de résolution.

Au final, c'est en parcourant plusieurs forums que nous nous penchons vers la librairie qui va suivre.

La librairie STD.TEXTIO.ALL :

Nous décidons alors de directement chercher les valeurs de notre signal analogique via cette librairie. Pour récupérer ces données, il faut déclarer certaines variables importantes dans le processus (voir *annexe 15 - Variables pour lecture de design.txt*) :

- La déclaration d'ouverture du fichier : nous l'appellerons *text_file* et initialiserons en mode lecture.
- Une variable *text_line* qui permettra de parcourir les lignes de notre fichier.
- Une variable de type *time* pour lire le temps en première colonne.
- Toutes les variables (vauxp_active6 et vauxn_active6) qui permettront de récupérer le signal analogique. Notre premier problème venait du type à définir pour ces variables. En effet, nous voulions le déclarer de la façon suivante :

```
variable vauxp_active6 : subtype'VAUXP;
variable vauxn_active6 : subtype'VAUXN;
```

Le subtype permet de faire des affectations entre deux paramètres lorsqu'ils ne sont pas de même type. Cela aurait été utile pour la récupération du signal analogique, mais le logiciel ne semble pas aimer cette utilisation. Nous décidons donc de les déclarer comme suit :

```
variable vauxp_active6 : std_logic_vector(11 downto 0);
variable vauxn_active6 : std_logic_vector(11 downto 0);
```

Lors du lancement d'une simulation, le logiciel semble ne pas faire la différence entre STD_LOGIC_VECTOR et STD_uLOGIC_VECTOR dans notre cas. Le dernier type correspond à un vecteur "unsigned". Comme Vivado ne connaît pas les signaux analogique, il serait logique de déclarer nos entrées comme ceci. Nous laissons tout de même en STD_LOGIC_VECTOR et aviserons en fonction des résultats.

Ensuite, on attend que RESET prenne la valeur 1 pour rentrer dans le fichier. Nous placerons plus tard une note « Hello wait until » à la suite pour vérifier que la condition soit bien respectée :

```
wait until RESET='1';
        report "Hello Wait until" ;
```

Puis, nous rentrons dans une boucle qui définit l'entrée dans le fichier. Nous restons dedans tant que ce n'est pas la fin du fichier. D'ailleurs le fichier n'a pas besoin d'être manuellement fermé puisque cela est fait par défaut en sortie de boucle. De même, nous placerons plus tard une note « Hello while not » pour vérifier si on est bien rentré dans le fichier.

```
while not endfile(text_file) loop
                report "Hello While not";
```

Pour la suite de notre testbench, nous utiliserons les fonctions READ et HREAD de notre librairie. Elles ont besoin d'au moins deux paramètres pour se repérer dans le fichier : *text_line* et une variable de lecture de données. Malheureusement, la fonction HREAD n'était pas reconnu alors qu'on avait bien déclaré la librairie comme dans cette source. Après plusieurs recherches, il semblerait qu'il fallait également ajouter la librairie standard utilisées dans IEEE :

```
use IEEE.STD_LOGIC_TEXTIO.ALL
```

Nous ajouterons plus tard une nouvelle variable de type *boolean* nommée « ok » pour vérifier la lecture d'une ligne. Cette variable peut être ajoutée en troisième paramètre dans les fonctions READ et HREAD. Lorsqu'une ligne du fichier est lue, « ok » prend la valeur 1, sinon 0. Ainsi nous vérifierons plus tard, avec la partie de code suivante qui est lue, si « ok » vaut 0 :

```
read(text_line,wht_time,ok);
assert ok
                report "Read 'wht_time' failed for line: " & text_line.all
                severity failure;
```

Malheureusement, le code s'arrête à « Severity Failure » puisqu'il n'a pas réussi à lire la première ligne. Nous nous demandons alors si ce n'est pas dû à l'en-tête. En effet, comme vous pouvez le voir ci-dessous, c'est à la toute première ligne du fichier que le programme ne parvient pas à lire notre variable wht_time qui est de type *time*.

```
Note: Hello Wait until
Time: 0 ps  Iteration: 1  Process: /ADC_v3_tb2/PROC_FILE  File: C:/Users/steph/OneDrive/Bureau/Cours/Projet/ADC_V3/ADC_v3.sr
Note: Hello While not
Time: 0 ps  Iteration: 1  Process: /ADC_v3_tb2/PROC_FILE  File: C:/Users/steph/OneDrive/Bureau/Cours/Projet/ADC_V3/ADC_v3.sr
Failure: Read 'wht_time' failed for line: TIME VP VN VAUXP[6] VAUXN[6]
```

*Figure 10 - Erreur de simulation*

Cependant, il est impossible de se passer de l'en-tête puisque Vivado en a besoin pour reconnaître les signaux. On peut tout de même remarquer que nos notes confirment notre entrée dans le fichier. C'est donc un problème de lecture. Nous décidons de voir si le problème n'est pas uniquement dû à la lecture du temps. Nous commentons alors la section de code liée à « assert ok ».

Pour la suite, nous utilisons la fonction HREAD qui nous permettra d'utiliser nos vecteurs de lecture vauxp_active6 et vauxn_active6. Voici comment nous procédons :

```
HREAD(text_line, vauxp_active6)
                VAUXP <= vauxp_active6;
```

Le but est de récupérer en entrée (VAUXP/VAUXN) le signal analogique. Cependant, lorsque nous lançons une simulation, toutes nos variables restent en UNSIGNED. Ce qui signifie qu'aucune valeur n'a été attribuée à nos variables.

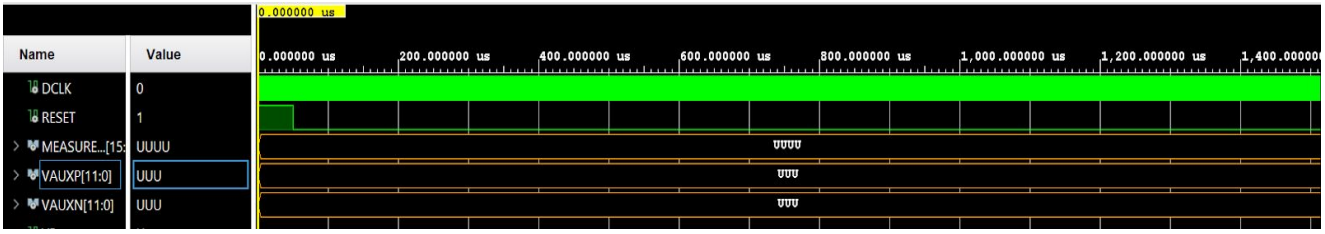



*Figure 11 - Simulation échouée*

De même, lorsque nous ajoutons le troisième paramètre « ok » sur chaque fonction, il s'avère que toutes les « Severity Failure » s'activent. Nous essayons différentes façons de récupérer le signal analogique notamment en changeant le type de nos variables, l'attribution du signal analogique ou encore l'entrée dans le fichier en s'inspirant de cette source. Malheureusement, rien n'est vraiment convaincant et concluant.

Nous décidons alors de tout reprendre depuis le début pour tenter de retrouver une erreur inaperçue. C'est alors qu'on crée une deuxième version de notre composant, beaucoup plus simple. Vous pouvez le retrouver en *annexe 16 - Deuxième version de l'ADC*.

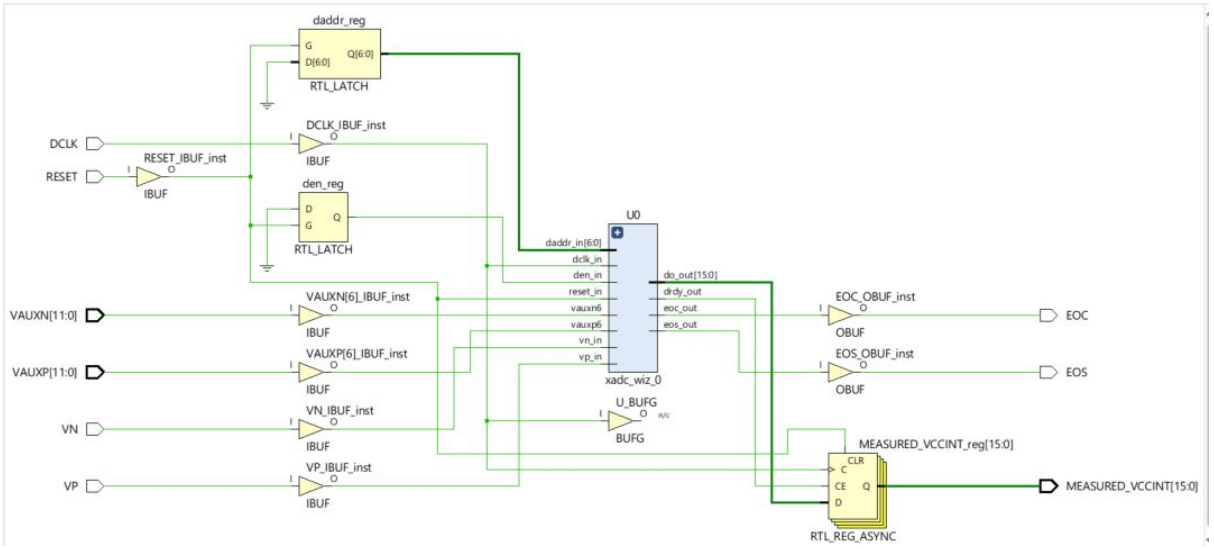



*Figure 12 - Block Diagram de la deuxième version*

L'objectif était de pouvoir débugger plus facilement le testbench si le problème venait du code principal. Dans celui-ci, nous nous basons uniquement sur drdy qui passe à 1 lorsque la donnée est prête à être lue. Ainsi nous pouvons attribuer la sortie do_drp de l' xadc_wiz_0 à la sortie du composant. Malheureusement, le problème est toujours le même… nous ne pouvons pas récupérer le signal analogique.

Pour passer la simulation sur logiciel, nous avons pensé à directement effectuer un test physique sur la board. Cependant, le microphone délivre plus d'1V en sortie alors que l'ADC ne prend qu'une tension comprise entre 0 et 1V. Pendant la période de confinement, nous n'avions pas accès à des résistances pour effectuer un diviseur de tension. Nous pouvons éventuellement connecter des pins d'entrée à la masse puis en l'air, mais le résultat serait trop aléatoire pour savoir si notre ADC fonctionne réellement sans être passé par la simulation logicielle.

Alternative :

Dans le but de pouvoir donner en entrée de la FFT des valeurs numériques cohérentes similaire à celle de l'ADC, nous allons générer ce signal d'entrée grâce à des sinus de fréquences différentes. Nous avons utilisé l'IP DDS Compiler permettant de produire des formes d'ondes sinusoïdales.

Dans notre cas, nous choisirons les fréquences sinusoïdales suivantes : 31, 62, 125, 250, 500, 1 000, 2 000, 4 000, 8 000 et 16 000 Hz.

Nous configurons l'IP et associons 1 switch à chaque fréquence avec la configuration du premier tableau ci-dessous. Puis, nous réalisons une simulation où l'on injecte des sinus, pour le signal d'entrée de la FFT, à chaque changement de switch dans l'ordre précis du second tableau :

| Fréquences (Hz) | switch (bit n°) |
|---|---|
| 31 | 9 |
| 62 | 8 |
| 125 | 7 |
| 250 | 6 |
| 500 | 5 |
| 1 000 | 4 |
| 2 000 | 3 |
| 4 000 | 2 |
| 8 000 | 1 |
| 16 000 | 0 |

| switch (binaire) | switch (hexa) | Fréquences (Hz) |
|---|---|---|
| 10_0000_0000 | 0x200 | 31 |
| 11_0000_0000 | 0x300 | 62 + 31 |
| 11_1000_0000 | 0x380 | 125 + 62 + 31 |
| 11_1100_0000 | 0x3c0 | 250 + 125 + 62 + 31 |
| 11_1110_0000 | 0x3e0 | 500 + 250 + 125 + 62 + 31 |
| 11_1111_0000 | 0x3f0 | 1 000 + 500 + 250 + 125 + 62 + 31 |
| 11_1111_1000 | 0x3f8 | 2 000 + 1 000 + 500 + 250 + 125 + 62 + 31 |
| 11_1111_1100 | 0x3fc | 4 000 + 2 000 + 1 000 + 500 + 250 + 125 + 62 + 31 |
| 11_1111_1110 | 0x3fe | 8 000 + 4 000 + 2 000 + 1 000 + 500 + 250 + 125 + 62 + 31 |
| 11_1111_1111 | 0x3ff | 16 000 + 8 000 + 4 000 + 2 000 + 1 000 + 500 + 250 + 125 + 62 + 31 |

*Figure 13 - Tableaux de fréquences correspondants aux switchs*

Nous obtenons la simulation suivante où la sortie numérique (avec une visualisation analogique) "m_axis_data_tdata_sine[15:0]" transporte les sinusoïdes :

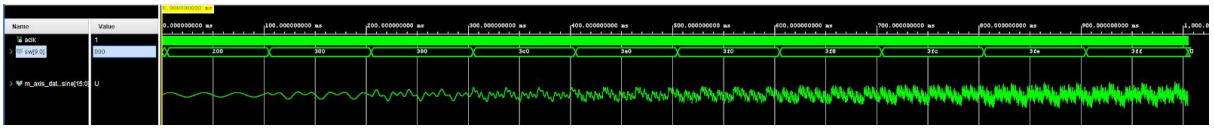


*Figure 14 - Simulation d'un signal analogique converti en numérique*

### 2.3.2. Transformation de Fourier rapide

Le but de cette partie est de récupérer les données en sortie du convertisseur analogique-numérique et d'effectuer une FFT pour avoir les amplitudes des signaux en sortie.

On rappelle qu'une FFT est un algorithme de calcul de la transformée de Fourier d'une fonction. Cette transformée de Fourier se présente sous forme de raies dont on va chercher à récupérer l'amplitude et les LEDs vont s'allumer en conséquence. Nous allons implémenter cette FFT sous Vivado également. Pour cela, nous allons utiliser l'IP FFT déjà proposé par Xilinx.

On commence par réaliser un Block design en y ajoutant l'IP FFT. Grâce à la datasheet proposée par le fabricant Xilinx, nous allons pouvoir configurer l'IP.
Dans un premier temps, il faut configurer le module de la fft. Nous décidons de nous concentrer tout d'abord sur la reconnaissance des signaux audibles, dont la fréquence est comprise entre 20Hz et 20kHz. D'après le théorème de Shannon, notre fréquence d'échantillonnage doit être supérieure à 40kHz.

Pour éviter le stockage de données sur notre carte électronique, nous décidons de lire les échantillons directement à partir du signal sortant de l'ADC, et de ne pas les stocker. L'IP FFT nous permet alors une fréquence d'échantillonnage minimale de 1MHz.
Nous souhaitons enregistrer au minimum, une période totale des signaux échantillonnés. Nous devons alors échantillonner sur une période minimale de 50 ms:

$$T_{max} = \frac{1}{f_{smin}} = \frac{1}{20} = 0,05s$$

On choisit le nombre de points de la FFT en fonction de cette formule:

$$T_e * nb_{echantillons} \geq T_{smax}$$

$$nb_{echantillons} \geq \frac{T_{smax}}{T_e}$$

$$nb_{echantillons} \geq 50\ 000$$

Nous devrions alors réaliser une FFT de 65 536 points. Pour simplifier l'IP, nous décidons d'ignorer les signaux de fréquence inférieure à 30Hz, et de réaliser une FFT 32 768 points.

On aura alors en sortie de FFT, 32 768 données fréquentielles. On additionnera ces fréquences pour trouver 12 plages de fréquences, qui seront ensuite affichées sur la matrice LED. On trouvera la configuration à l'annexe 17.

A l'aide de l'IP DDS compiler, nous simulons une onde sinusoïdale en entrée de la FFT à défaut d'avoir les résultats de l'ADC. On configure cet IP de manière à avoir une fréquence d'entrée de 100 MHz avec une largeur en sortie de 16 bits qui est la taille de sortie de celle de l'ADC.

En sortie de la FFT, on récupère les composantes réelles et imaginaires du signal généré par le DDS compiler que l'on sépare en deux blocs bien distincts. Nous avons nos composantes, il nous reste maintenant à trouver l'amplitude associée. Pour cela, on utilise la formule suivante : $Amplitude = \sqrt{Re^2 + Im^2}$. En ajoutant des multiplieurs, un additionneur et l'IP Cordic pour réaliser la racine carrée du calcul, on arrive au diagramme final pour la partie FFT (*Figure 15 - Diagramme bloc*).

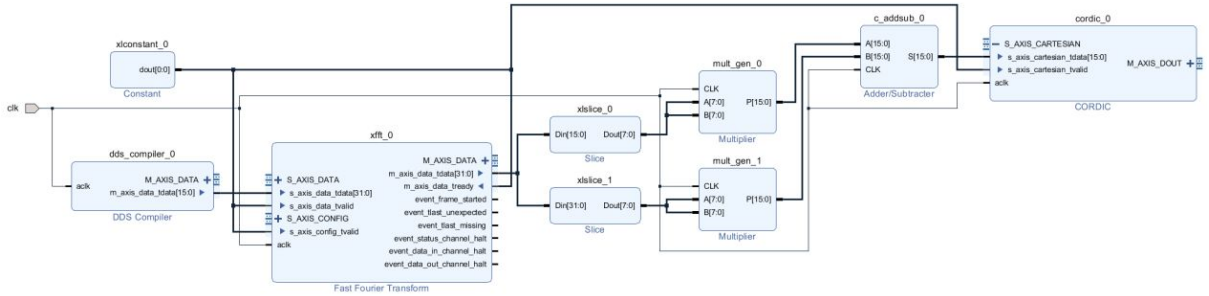



*Figure 15 - Diagramme bloc*

Après avoir réalisé le diagramme bloc, nous avons effectué la simulation pour obtenir un signal de sortie. Lorsque l'on effectue la simulation, on observe malheureusement pas le résultat attendu. On peut voir qu'en injectant bien une sinusoïde dans le compiler, on n'observe pas le pic attendu en sortie du module Cordic (voir *annexe 18 - Simulation fft*).

En essayant de changer les valeurs de fréquence, nous obtenons toujours des résultats infructueux.

### 2.3.3. Panneau leds

Dans cette partie, nous allons nous focaliser sur la méthode de développement pour aboutir au contrôle total du panneau leds. Ce dernier est composé de 6 lignes et de 24 colonnes, pour un total de 144 leds. Cependant, la soudure connectant la dernière ligne a été cassée, et par un manque de matériel à notre disposition dû au confinement, nous sommes contraint de travailler sur 120 leds. Malgré cela, l'ensemble des utilisations prévues initialement pourront être réalisées.

Tout d'abord, il a fallu identifier le protocole de communication du bandeau leds ayant pour référence SJ-100144-2811 et s'alimentant sous une tension continue de 5V. Sur le site du constructeur, nous pouvons remarquer que le protocole est le WS218B. Les leds de ce protocole sont la combinaison de 3 leds de couleurs (rouge, vert et bleu) et d'un circuit de commande numérique. Une led se contrôle par une trame binaire de 24 bits, avec 8 bits par couleur (vert, rouge, bleu), offrant une possibilité de 256 niveaux de couleurs. La composition des données de la trame sur 24 bits est la suivante :

| G7 | G6 | G5 | G4 | G3 | G2 | G1 | G0 | R7 | R6 | R5 | R4 | R3 | R2 | R1 | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

*Figure 16 - Ordre de la trame de 24 bits*

Lors de la transmission des bits, il faut prendre en compte que l'émission d'un 0 ou 1 logique est défini par des temps logiques hauts et bas (horloge asymétrique) comme le montre la figure ci-dessous.

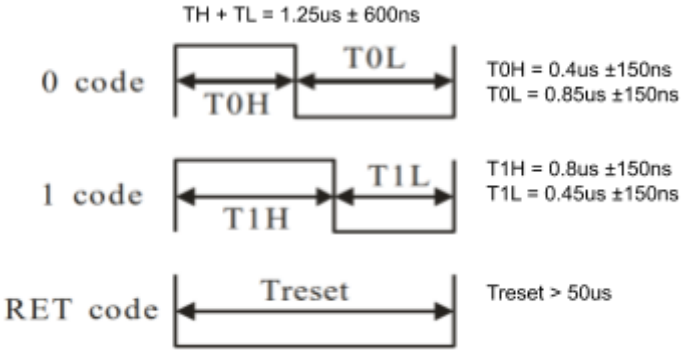



*Figure 17 - High & Low voltage time*

L'avantage de ces leds est la possibilité de les mettre en cascade. Autrement dit, les données de toutes les leds sont transmises dans la première led qui récupère ses 24 bits de couleurs et qui transmets le reste de la trame aux leds suivantes. Il faut donc être capable de générer une suite de N x 24 bits pour pouvoir contrôler l'ensemble des N leds. Il faut noter également que chaque led garde en mémoire la couleur reçue en attendant de recevoir une nouvelle trame.

Avec toutes ses informations, nous pouvons démarrer le développement VHDL. Les différentes étapes accordées au panneau leds seront de tester en simulation et en réel avec la Basys 3, le contrôle d'une led, puis le contrôle de N leds, et enfin le contrôle des animations sur les N leds.

Pour commencer, l'objectif est de réussir à communiquer avec la première led du bandeau. Pour cela, nous avons créé une entité capable de recevoir des paramètres d'entrées pour configurer la couleur d'une seule led (voir *Annexe 19 - Entité one_led*). Ce premier fichier VHDL nous a permis de générer en sortie la trame binaire capable de piloter une led en respectant l'ordre des 24 bits de données et de l'horloge asymétrique. En simulation, nous avons obtenu la visualisation qui suit (voir *Figure 18 - Test bench de one_led*), avec en entrée logique l'horloge présente sur la Basys 3 de 100 MHz qui nous servira de référence pour le compteur, une activation d'envoi de la trame, des vecteurs pour le choix des couleurs, et en sortie logique la trame générée. Un test avec la board a été réalisé pour confirmer la simulation.

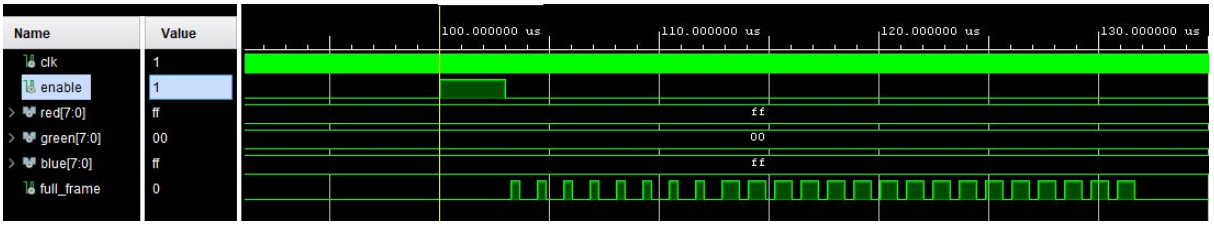



*Figure 18 - Test bench de one_led*

Une fois le contrôle d'une led opérationnel, il est nécessaire de pouvoir exploiter cette entité sur N leds. Une nouvelle entité est créée et aura pour objectif d'appeler N fois l'entité "one_led", et se nommera "all_leds" (voir *Annexe 20 - Entité all_leds*). Cette entité reçoit en entrée un tableau de vecteurs à 2 dimensions. Le type de ce tableau est propre à notre utilisation et a été créé dans le but de stocker la couleur de chaque led (voir *Annexe 21 - Librairie myLibrary*). Pour la gestion des couleurs RGB de chaque led, nous avons remarqué de part notre réalisation au S6, que la luminosité maximum d'une couleur, c'est-à-dire 255, est trop élevée et désagréable visuellement, et a par extension une consommation de courant supplémentaire. Nous avons fait le choix de n'utiliser et de ne mettre à 1 ou 0 que le bit[3], représentant une faible consommation de courant en gardant un aspect visuel correct. De plus, nous nous sommes restreints à un choix de couleurs limités (7 couleurs possibles + 1 couleur dite "éteinte") dans le but de diminuer également la lourdeur du programme VHDL, d'où le tableau de notre librairie. Les couleurs choisies sont les suivantes :

| En décimal | R | G | B | Couleur |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | LED éteinte |
| 1 | 0 | 0 | 1 | bleu |
| 2 | 0 | 1 | 0 | vert |
| 3 | 0 | 1 | 1 | cyan |
| 4 | 1 | 0 | 0 | rouge |
| 5 | 1 | 0 | 1 | magenta |
| 6 | 1 | 0 | 1 | jaune |
| 7 | 1 | 1 | 1 | blanc |

*Figure 19 - Couleurs possibles*

Le contenu de l'entité "all_leds" est de cadencer à un rythme précis, égal au temps d'émission des 24 bits qui est de 28.8 ms. Ce calcul est obtenu en multipliant un coup d'horloge à 100 MHz, par le temps d'envoi en ns d'un bit à 0.4us de l'horloge asymétrique, puis par le nombre d'état pour envoyer 1 bit en 1.2us de cette même horloge (voir *Figure 17 - High & Low voltage time*), puis par le nombre de bits total à envoyer. L'opération est $10ns * 40 * 3 * 24 = 28800\ ns$.

Il faudra donc 28.8 us x N leds pour émettre la trame complète, dans notre cas $28.8 * 120 = 3456\ us = 3.456\ ms$ en théorie. En simulation, nous obtenons un temps de 3.45719 ms (voir *Figure 20 - Trame complète avec all_leds*). Cette différence $3.45719 - 3.456 = 0.00119\ ms = 1.19\ us$ s'explique simplement par un retard de 10 ns par led dans le VHDL, sachant que la sortie logique "trame_busy" décroche 10 ns avant la fin complète, d'où la différence $120 * 10\ ns\ - 10\ ns\ = 1.19\ us$.

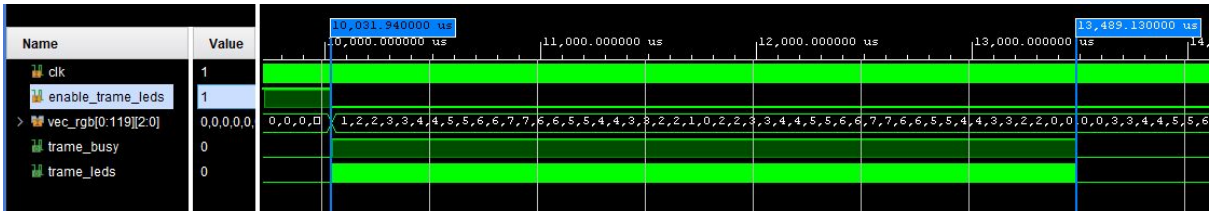



*Figure 20 - Trame complète avec all_leds*

Dès que le contrôle de toutes les leds est acquis, on s'intéresse maintenant aux animations que l'on souhaite afficher sur ce panneau. Une nouvelle entité est créée et se nommera "animations" (voir *Annexe 22 - Entité animations*). Cette entité permettra de récupérer le choix de l'animation que l'utilisateur souhaite, cette donnée est transmise par Bluetooth et envoyée à cette entité en entrée. Grâce à cette information, le contenu du process ira configurer le vecteur de couleurs "vec_rgb" (voir *Figue 21 - vec_rgb*) en entrée de "all_leds" en fonction du choix d'animation.

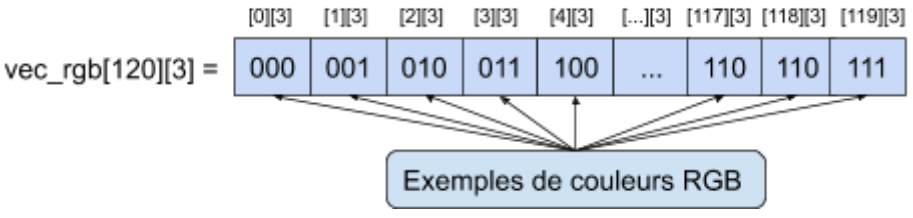


*Figure 21 - vec_rgb*

Nous pouvons à présent représenter par des blocs les différents VHDL créés pour une meilleure visualisation du fonctionnement (voir *Figure 22 - Fonctionnement des entités leds*).

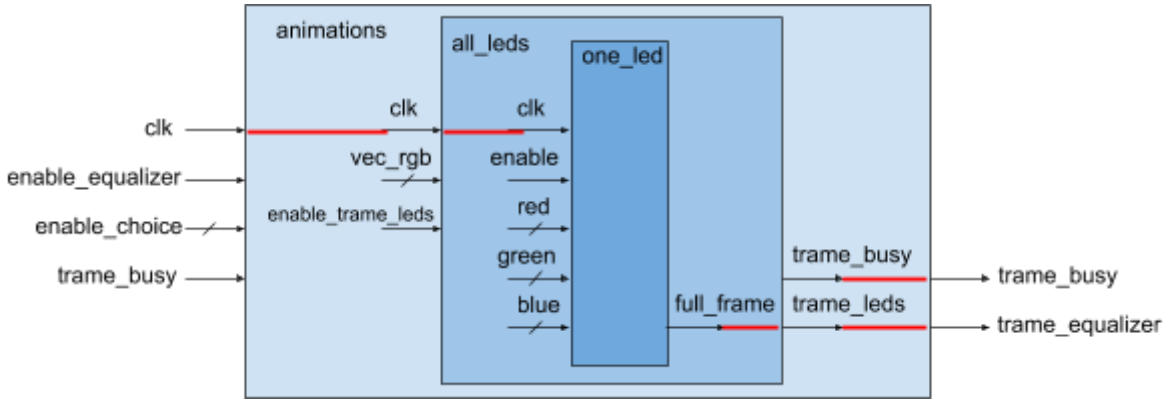



*Figure 22 - Fonctionnement des entités leds*

Les différentes animations développées sont :

- un égaliseur fixe (voir *Annexe 23 - Animation "égaliseur fixe"*).
- les 8 couleurs possibles affichées par pas de 8 leds sur les 120 leds, rythmées de 2 affichages différents toutes les secondes (voir *Annexe 24 - Animation "couleurs rythmées"*).
- un chenillard de 7 couleurs zigzaguant de la première à la dernière led indéfiniment (voir *Annexe 25 - Animation "chenillard"*).
- ainsi que l'extinction des leds.

Le test bench à partir de l'entité "animations" en choisissant l'égaliseur fixe, puis les couleurs rythmées aura pour effet la simulation qui suit (voir *Figure 23 - Test bench de 2 animations*). Le choix de l'égaliseur fixe se fait par la réception de la donnée 0x31 et des couleurs rythmées par 0x32. On remarque que la trame en sortie émet à un intervalle régulier un changement d'affichage des leds à partir d'un unique bit d'activation au début (à 10 ms sur la figure) pour la deuxième donnée reçue.

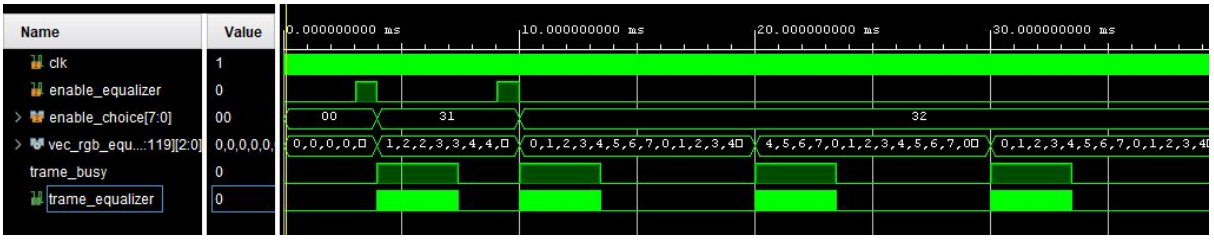



*Figure 23 - Test bench de 2 animations*

Une fois cette partie validée en simulation et en pratique, il serait intéressant à l'utilisateur de pouvoir contrôler à distance les différentes animations dont il souhaite, d'où l'intégration du module Bluetooth à notre projet.

### 2.3.4. Bluetooth

Dans cette partie, nous allons nous intéresser à la communication Bluetooth entre un smartphone et la Basys 3. Nous avons à notre disposition le module Bluetooth HC-05, connu pour sa facilité d'utilisation avec l'environnement d'Arduino, ayant pour caractéristique principale une communication par liaison série TTL en respectant le protocole UART. De plus, ce module est configurable et offre les fonctionnalités nécessaires pour notre utilisation. Dans notre cas, il sera alimenté en 5V et les broches RX TX émettront sous une tension de 3.3V, cette tension est adaptée pour les ports d'entrées de la Basys 3.

Tout d'abord, il est important de connaître le fonctionnement du protocole UART (Universal Asynchronous Receiver Transmitter) et permet avant tout une communication asynchrone en tant qu'émetteur-récepteur. Les données sont envoyées à travers la liaison série respectant une suite de bits de la trame UART (voir *Figure 24 - Trame UART*). Pour qu'une transmission soit comprise, il est nécessaire que les deux périphériques aient les mêmes configurations. Autrement dit, il faut qu'ils aient la même vitesse de transmission, la même taille (bits) de donnée, l'utilisation du bit de parité et avec une parité paire ou impaire.

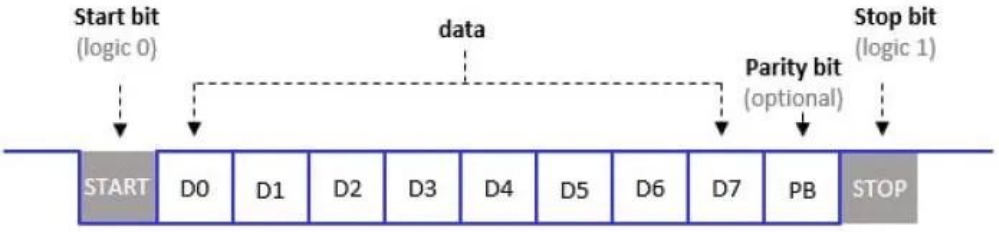



*Figure 24 - Trame UART*

Dans notre application, nous avons décidé de travailler à une vitesse de 9600 bauds, l'équivalent à 9600 bits par seconde, et sans bit de parité. Pour le développement de l'UART, pour commencer nous avons repris le code source présent sur Digikey afin de comprendre son fonctionnement et de le tester rapidement avec la board. Nous l'avons gardé pour son optimisation et sa configuration facile d'accès, mais surtout parce que nous avons rencontré de multiples problèmes que nous évoquerons à la fin de cette partie.

Premièrement, il est important de savoir à quel moment dans le temps l'on doit échantillonner les données. En effet, si l'on échantillonne au mauvais instant, on risque de lire des mauvaises données. Le FPGA devra donc échantillonner en continue sur la ligne de réception. Une fois qu'il remarquera le bit de start, c'est-à-dire au premier passage à l'état bas (l'équivalent à un front descendant), il comprendra qu'une donnée va être reçue. Le taux de sur-échantillonnage est fixé à 16, soit 16 échantillons par période de bauds. Dès qu'il détectera que l'entrée RX est à l'état bas pendant 8 échantillons consécutifs, il aura compris qu'il s'agira du bit de start. À partir de là, le FPGA lira l'état du milieu de chaque bit de réception et le fera pour les 8 bits de données (voir *Figure 25 - Trame UART vue par le FPGA*).


*Figure 25 - Trame UART vue par le FPGA*

Pour illustrer le fonctionnement du test bench qui nous montre la réception de la donnée, il est intéressant de connaître l'entité de cet UART (voir *Annexe 26 - Entité UART*). Dans cette simulation (voir *Figure 26 - Test bench RX*), on décide d'envoyer sur RX la première donnée binaire "00110001" à raison de 9600 bits par seconde, soit un bit toutes les 0.10416ms sur cette broche. On remarque que le front descendant est perçu à 3ms, mais que le signal "rx_busy" n'est à l'état 1 qu'une demi-période d'un bit (soit $3 + 0.10416 / 2\ ms$), ce qui confirme ce que l'on a présenté dans le paragraphe précédent. Dans cet exemple, la donnée reçue 0x31 est bien celle émise, puis la deuxième émission correspond à la donnée 0x32. Ces données sont la représentation en ASCII des caractères "1" et "2".

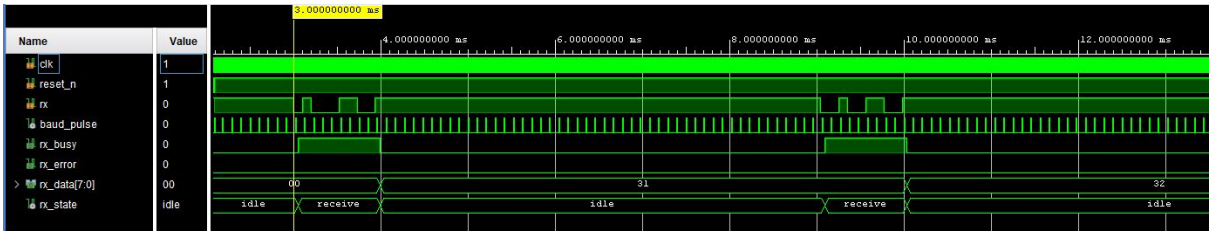



*Figure 26 - Test bench RX*

Puis du côté TX, on veut émettre le message "ok:[donnée RX reçue]". Par exemple, si l'on reçoit le code 0x31 ("1" en ASCII), on émettra le message "ok:1" à l'utilisateur. La simulation est concluante, on constate que le message est correctement émis (voir *Figure 27 - Test bench TX*).

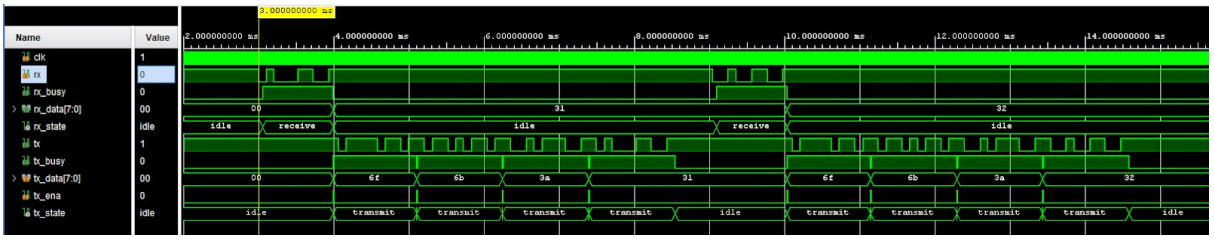



*Figure 27 - Test bench TX*

Pour réaliser les ordres d'envois sur la broche TX et orienter la donnée reçue vers d'autres programmes VHDL, nous avons créé une entité capable de contrôler l'entité UART en reprenant ses entrées/sorties (voir *Annexe 27 - Entité control*). Le schéma bloc simplifié ci-dessous permettra une meilleure visualisation de nos 2 entités qui plus tard, sera fusionnée avec les autres entités.

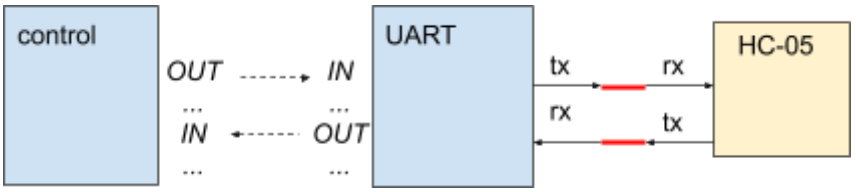



*Figure 28 - Fonctionnement des entités pour l'UART*

Après avoir validé les tests en simulation, on réalise des essais avec un terminal Bluetooth sur un téléphone. On constatera que cela fonctionne correctement (voir *Figure 29 - Terminal Bluetooth*).

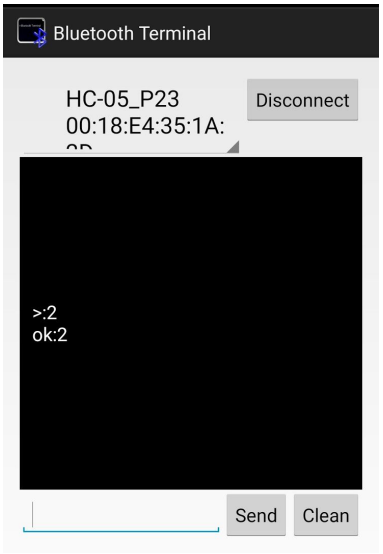



*Figure 29 - Terminal Bluetooth*

Les problèmes rencontrés lors de la mise en pratique de ces tests ont été variées. Tout d'abord, on a constaté une mauvaise communication entre le HC-05 et la Basys 3. Pour comprendre d'où pouvait venir le problème, on a été visualisé sur l'IDE Arduino la configuration du HC-05, qui avait pour paramètres 9600[bauds],0[parité],0[bit de stop]. Nous avons redéfini le bit de stop à 1 pour une compatibilité entre les composants. De plus, pour être certain que l'émission sur la broche TX de la Basys 3 était correcte, nous l'avons redirigé vers son RX. Le test a été concluant puisqu'il comprenait la donnée émise, nous permettant d'être sûr que le problème ne pouvait pas venir de la board. Cependant, ces modifications n'ont pas résolu le problème initial. Finalement, la solution était simple et fut un oubli de masse commune, cette erreur nous a permis d'explorer toutes les causes possibles. D'autres part, des warnings sont présents lors de l'implémentation cependant ils sont dû aux entrées et sorties asynchrones et nous pouvons les ignorer.

### 2.3.5. Application mobile

Pour une utilisation et une expérience utilisateur réussie, il est nécessaire d'avoir un environnement agréable et simple à prendre en main. C'est la raison pour laquelle nous avons décidé de développer une application mobile, contrairement aux tests expérimentaux

que nous avons réalisé jusqu'ici où nous passions par un terminal Bluetooth série. Cette application a été développée sous MIT App Inventor car nous n'avons pas besoin de fonctionnalité poussée comme pourrait proposer d'autres IDE.

L'interface de cette application est visualisable sur un seul écran (voir *Figure 30 - Interfaces application Android*). Lorsqu'on la lance, on nous informe qu'elle n'est connectée à aucun appareil Bluetooth. Une fois connecté grâce à l'appui sur le bouton correspondant, l'interface s'actualise pour nous donner la main sur les leds. Les boutons de commandes envoient respectivement le texte "0" pour l'extinction des leds, "1" pour l'égaliseur fixe, "2" pour les couleurs rythmées, et ainsi de suite. Lorsque cette donnée est reçue et traitée par la board, elle affiche l'animation demandée et envoie une chaîne de caractères à l'application. Cette chaîne s'affiche dans la section "Messages de la Basys 3" et sera de type "ok:3" pour le chenillard par exemple.

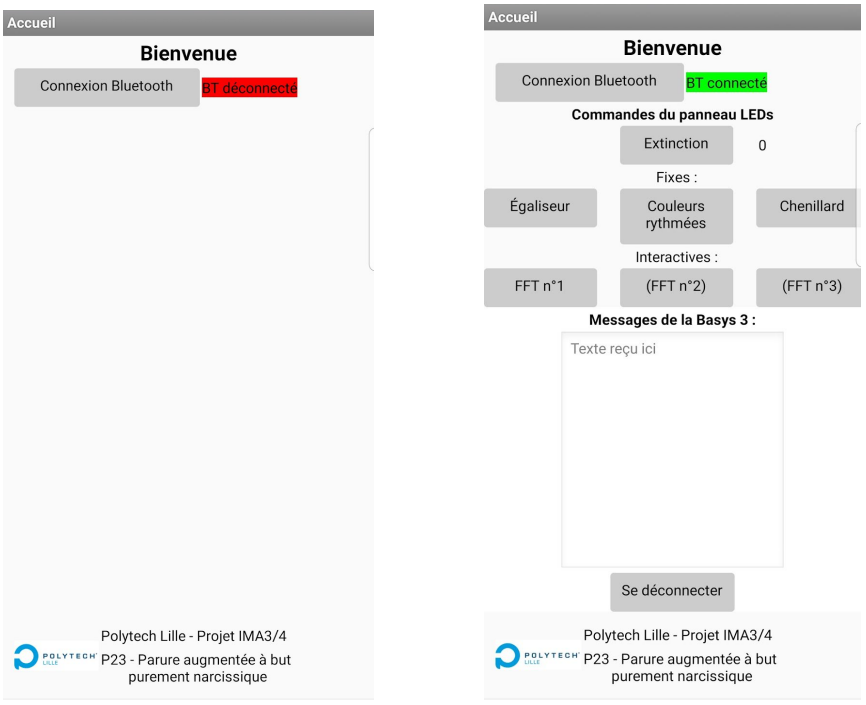



*Figure 30 - Interfaces application Android*

## 2.4. Résultat final

Après avoir abordé toutes les parties essentielles de notre projet, il est maintenant temps de les fusionner entre elles afin qu'elles puissent interagir ensemble. Nous pouvons illustrer les entitées par le schéma qui suit.

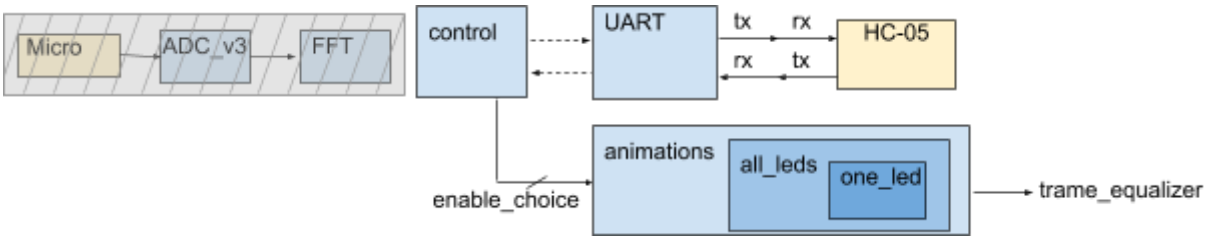



*Figure 31 - Entités fusionnées*

Le schéma de câblage des fonctionnalités opérationnelles se présente sous cette forme :

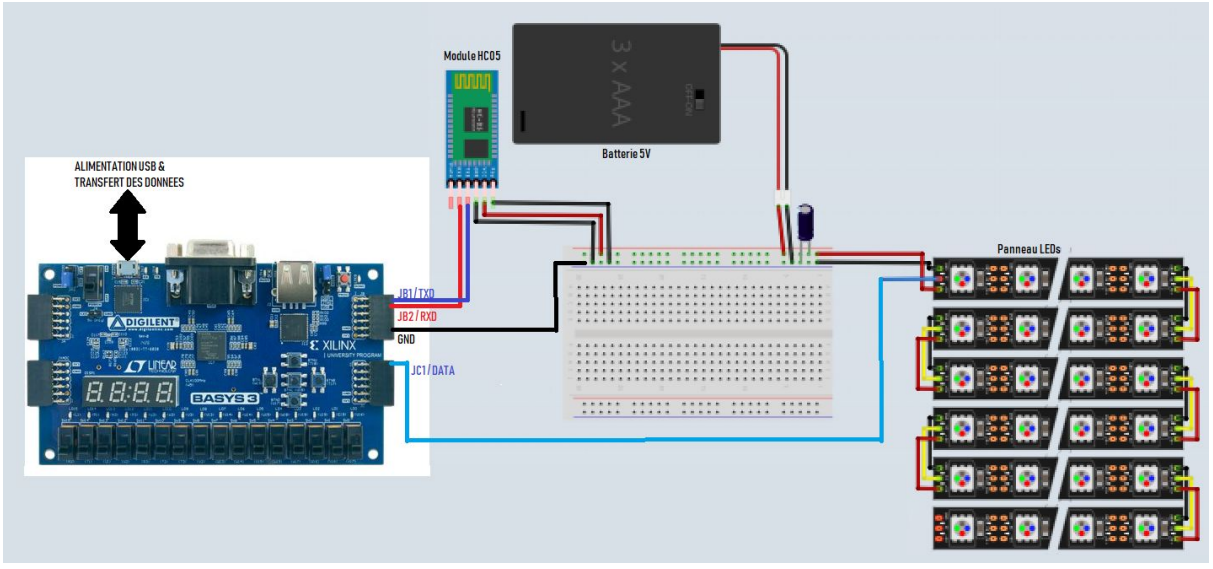



*Figure 32 - Schéma de câblage*

Nous avons réalisé un essai avec notre application en alternant différentes animations. Dans ce cas précisément, nous exécutons une suite de commandes avec initialement les leds éteintes :

- égaliseur (“ok:1”)
- extinction (“ok:0”)
- couleurs rythmées (“ok:2”)
- extinction (“ok:0”)
- chenillard (“ok:3”)
- etc.

Cet essai sera diffusé lors de notre présentation orale et est disponible sur notre wiki (voir 6.4.13.Résultat final). Toutes ces actions auront provoqué cet affichage sur l’application (voir *Figure 33 - Affichage application*). Malgré le non fonctionnement de l’ADC et de la FFT, ce prototype montre le résultat de l’ensemble de nos travaux.

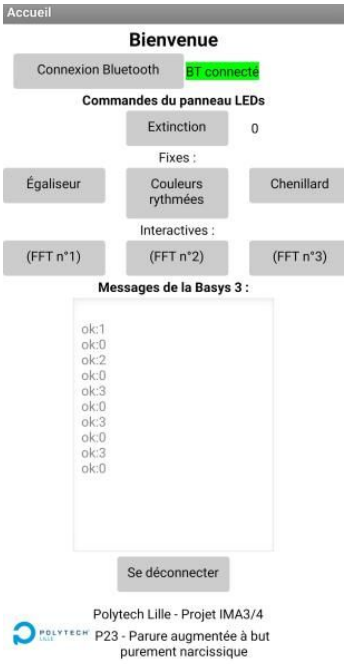



*Figure 33 - Affichage application*

# Bilan

Ce projet fut une expérience de travail de groupe enrichissante et nous a permis de nous confronter à la réalité du rythme de travail sur un projet de groupe. Nous avons eu plusieurs problèmes liés aux fonctionnements de l'ADC puis de la FFT, mais également lors des animations lumineuses et de l'UART. Cela nous a appris à raisonner sur toutes les causes possibles. Ce travail en langage de description matériel (VHDL) pour représenter le comportement et l'architecture d'un système électronique-numérique fût prenant de part sa complexité qui lui est associé, comme les contraintes imposées lors des phases de synthèse et d'implémentation. Ainsi, ces développements en VHDL nous ont sortis de notre façon de programmer habituelle. Lorsque les simulations fonctionnaient mais que les tests réels et physiques ne suivaient pas, cela nous encourageait à relever le défi. De fil en aiguille, c'est en réussissant à résoudre nos problèmes petit à petit que nous pouvions rester motiver face à notre projet. Concernant les PCB, la réalisation de ceux-ci, malgré la complexité des différents composants et de leurs fonctionnements nous a permis de concevoir une carte élaborée et de se perfectionner sur le logiciel de CAO Altium.

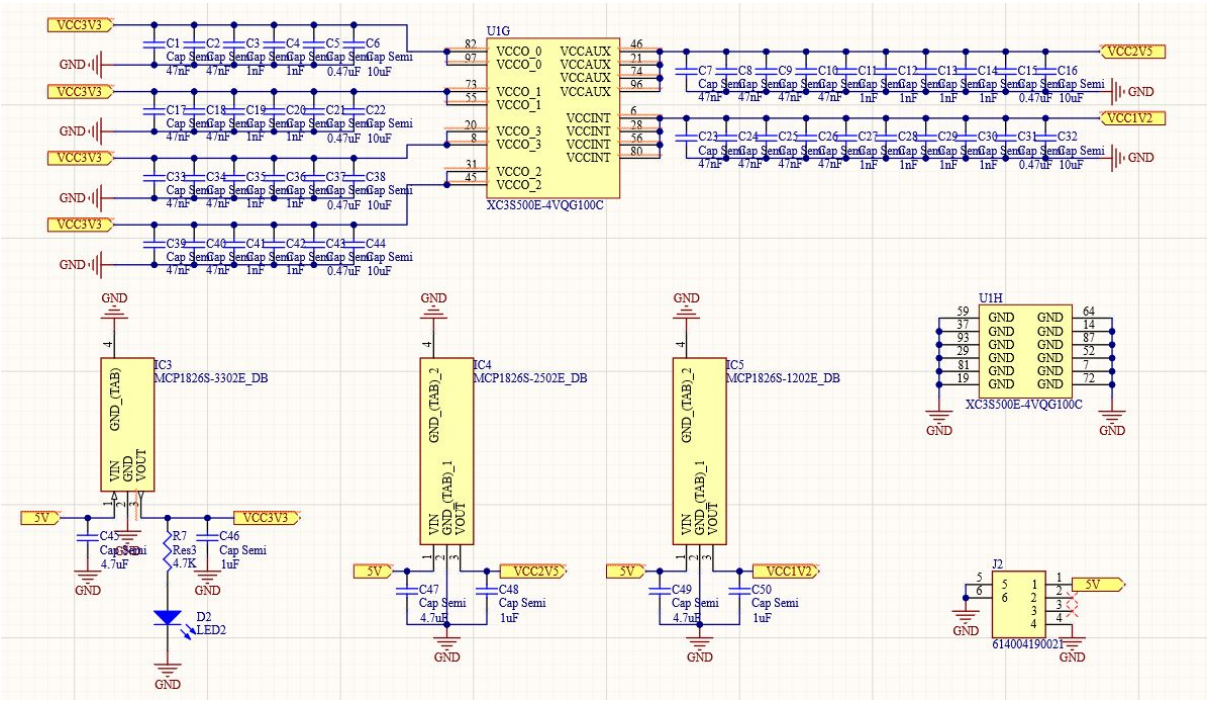
Pour conclure ce rapport, ces 3 semestres ont été rythmées par des choix qui n'ont pas toujours été judicieux et ont causés des difficultés d'orientations du projet. Cependant, avec les remarques remontées, nous avons pu nous ré-orienter et proposer un projet à la hauteur des attentes des encadrants. Pendant le S8, la période de confinement a changé notre façon de travailler. Une approche différente a donc été mise en place pour pouvoir poursuivre notre collaboration. Nous avons adapté par conséquent notre communication et la répartition du travail selon les contraintes de chacun.

# Sources

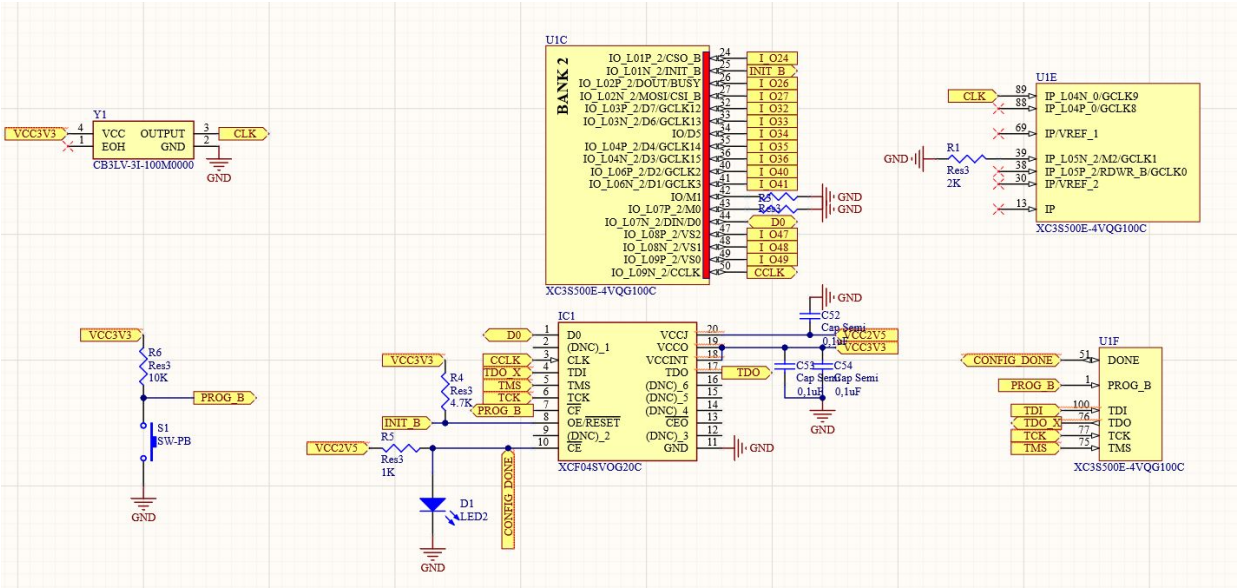
De nombreux hyperliens sont présents à travers ce rapport pour faciliter leurs consultations.

Manuel référent de la Basys 3
Datasheet de l'ADC pour FPGA série 7
Datasheet de l'IP XADC
Datasheet : Comment simuler sur Vivado
Datasheet : Guide des testbench
Exemple d'utilisation d'un ADC
Forum sur la simulation de l'XADC
Implantation des machines d'états
Librairie TextIO : lecture/écriture dans un fichier .txt

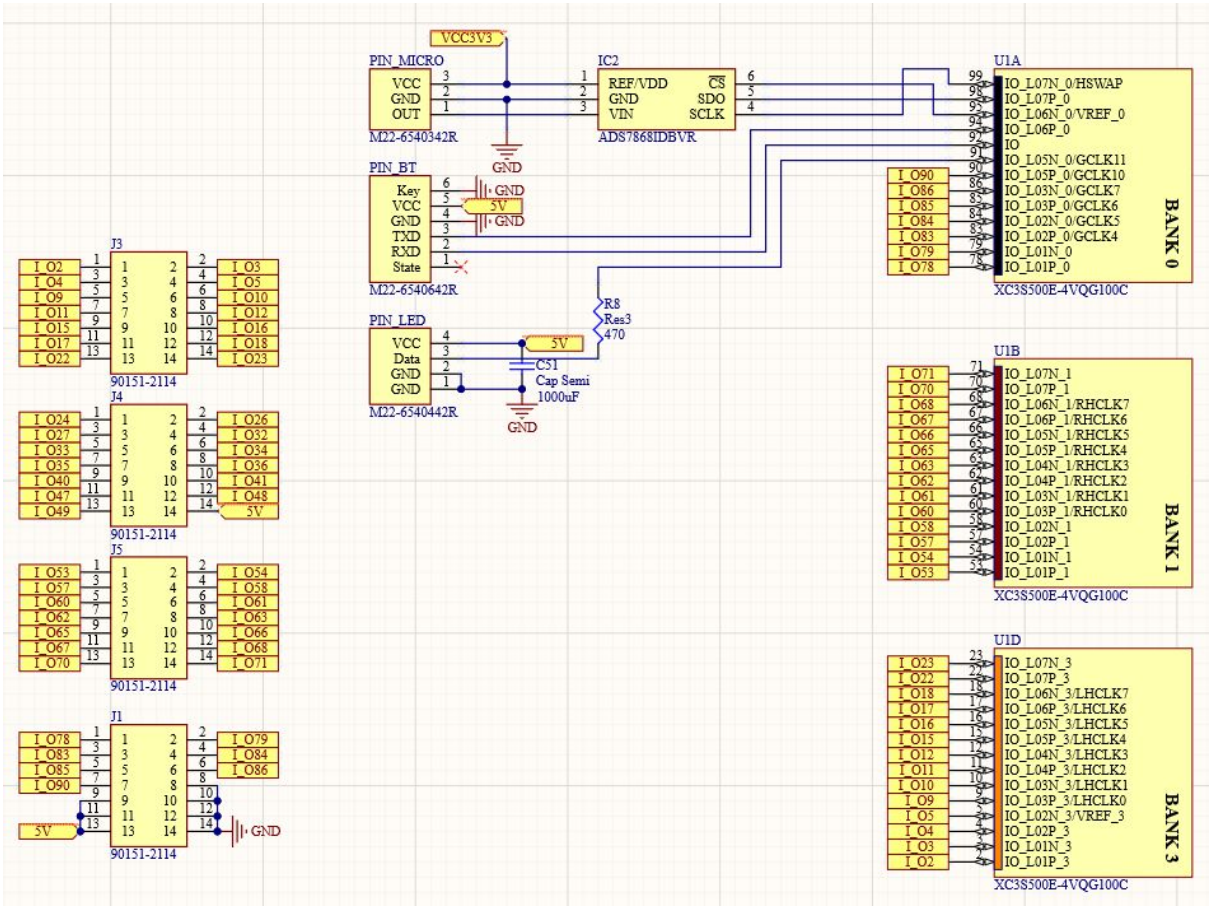
# Annexes



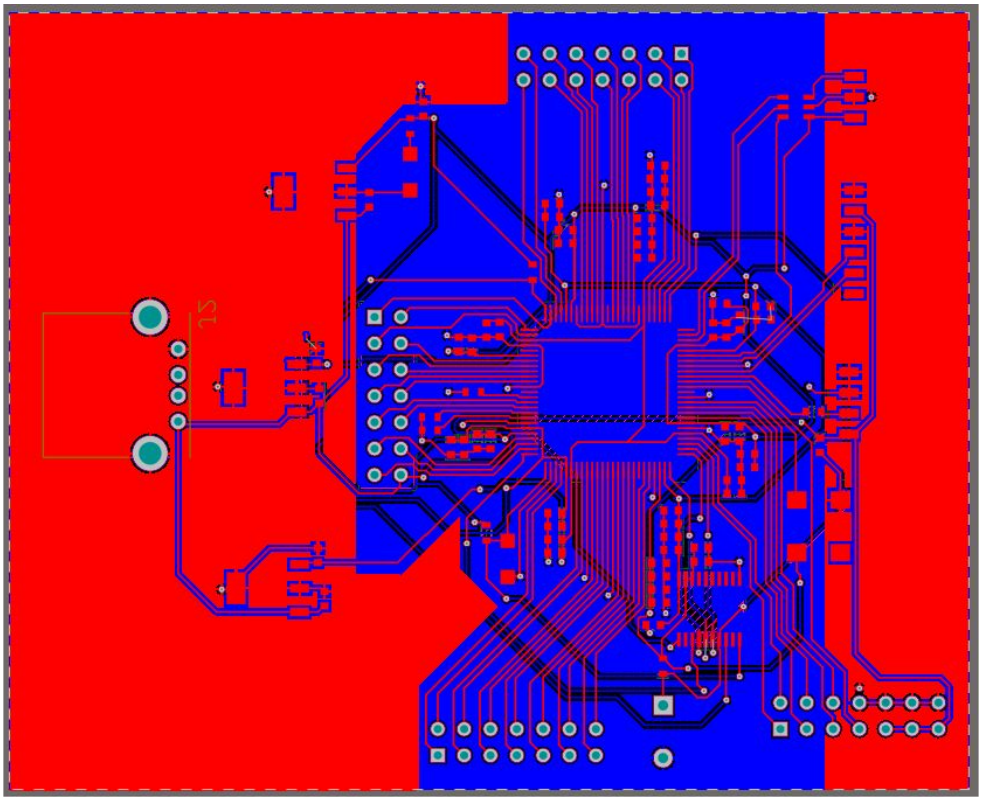

*Annexe 1 - Schéma électrique: alimentation*



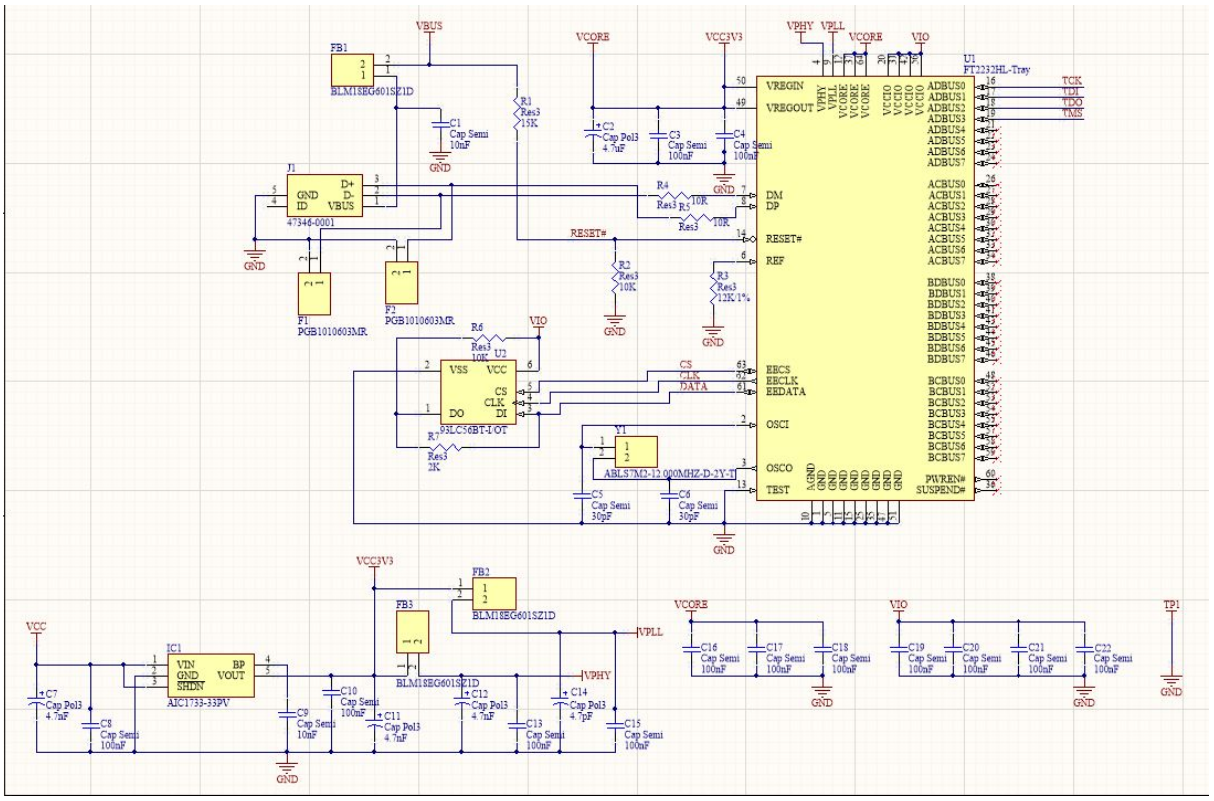

*Annexe 2 - Schéma électrique: configuration*



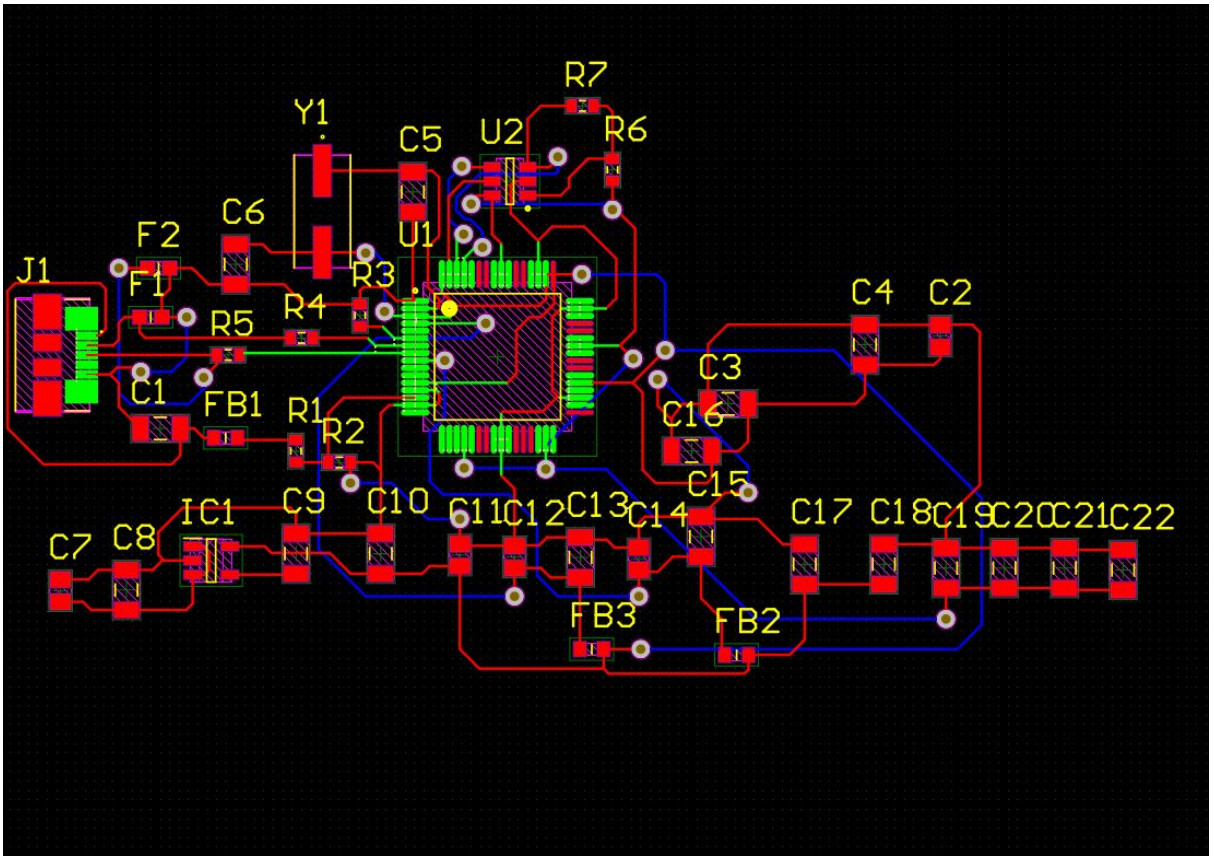

*Annexe 3 - Schéma électrique: interfaces*



*Annexe 4 - Routage carte FPGA*




*Annexe 5 - Schematic USB to JTAG*




*Annexe 6 - Routage USB to JTAG*

```
97          aux_channel_p(0) <= '0';
98          aux_channel_n(0) <= '0';
99
100         aux_channel_p(1) <= '0';
101         aux_channel_n(1) <= '0';
102
103         aux_channel_p(2) <= '0';
104         aux_channel_n(2) <= '0';
105
106         aux_channel_p(3) <= '0';
107         aux_channel_n(3) <= '0';
108
109         aux_channel_p(4) <= '0';
110         aux_channel_n(4) <= '0';
111
112         aux_channel_p(5) <= '0';
113         aux_channel_n(5) <= '0';
114
115         aux_channel_p(6) <= vauxp6;
116         aux_channel_n(6) <= vauxn6;
117
118         aux_channel_p(7) <= '0';
```

```
172 port map (
173         CONVST              => '0',
174         CONVSTCLK           => '0',
175         DADDR(6 downto 0)   => daddr_in(6 downto 0),
176         DCLK                => dclk_in,
177         DEN                 => den_in,
178         DI(15 downto 0)     => di_in(15 downto 0),
179         DWE                 => dwe_in,
180         RESET               => reset_in,
181         VAUXN(15 downto 0)  => aux_channel_n(15 downto 0),
182         VAUXP(15 downto 0)  => aux_channel_p(15 downto 0),
183         ALM                 => alm_int,
184         BUSY                => busy_out,
185         CHANNEL(4 downto 0) => channel_out(4 downto 0),
186         DO(15 downto 0)     => do_out(15 downto 0),
187         DRDY                => drdy_out,
188         EOC                 => eoc_out,
189         EOS                 => eos_out,
190         JTAGBUSY            => open,
191         JTAGLOCKED          => open,
192         JTAGMODIFIED        => open,
193         OT                  => open,
```

```
55  entity xadc_wiz_0 is
56     port
57     (
58      daddr_in        : in  STD_LOGIC_VECTOR (6 downto 0);
59      den_in          : in  STD_LOGIC;
60      di_in           : in  STD_LOGIC_VECTOR (15 downto 0);
61      dwe_in          : in  STD_LOGIC;
62      do_out          : out  STD_LOGIC_VECTOR (15 downto 0);
63      drdy_out        : out  STD_LOGIC;
64      dclk_in         : in  STD_LOGIC;
65      reset_in        : in  STD_LOGIC;
66      vauxp6          : in  STD_LOGIC;
67      vauxn6          : in  STD_LOGIC;
```

*Annexe 7 - xadc_wiz_0 instancié*

```
signal FLOAT_MUXADDR    : STD_LOGIC_VECTOR (4 downto 0);
  signal den_reg, dwe_reg: STD_LOGIC_VECTOR (1 downto 0);
  signal vauxp_active    : STD_LOGIC_VECTOR (15 downto 0);
  signal vauxn_active    : STD_LOGIC_VECTOR (15 downto 0);
  signal daddr           : STD_LOGIC_VECTOR (6 downto 0);
  signal den             : STD_LOGIC;
  signal di_drp                : STD_LOGIC_VECTOR (15 downto 0);
  signal dwe             : STD_LOGIC;
  signal do_drp          : STD_LOGIC_VECTOR (15 downto 0);
  signal drdy            : STD_LOGIC;
  signal eoc_drp         : STD_LOGIC;
  signal eos_drp         : STD_LOGIC;
  signal busy            : STD_LOGIC;                    -- ADC Busy signal
  signal dclk_bufg       : STD_LOGIC;
```

*Annexe 8 - Signaux de notre composant*

```
U0 : xadc_wiz_0
port map (
        daddr_in(6 downto 0)    => daddr(6 downto 0),
        dclk_in                  => DCLK,
        den_in                   => den,
        di_in(15 downto 0)       => di_drp(15 downto 0),
        dwe_in                   => dwe,
        reset_in                 => RESET,
        vauxn6                   => VAUXN(6),
```

```
        vauxp6                  => VAUXP(6),
        busy_out                => busy,
        --channel_out(4 downto 0) => CHANNEL(4 downto 0),
        do_out(15 downto 0)     => do_drp(15 downto 0),
        drdy_out                => drdy,
        eoc_out                 => EOC,
        eos_out                 => EOS,
        vn_in                   => VN,
        vp_in                   => VP
          );
```

*Annexe 9 - Port map de notre ADC*

```
NEXT_STATE_DECODE: process (dclk_bufg, RESET)
        begin
                if (RESET = '1') then
                                        state <= init_read;
                elsif (dclk_bufg'event and dclk_bufg = '1') then
                         case (state) is
                                when init_read =>
                                        daddr       <= "1000000";
                                        den_reg     <= "10";
                                        dwe_reg     <= "00";
                                        state <= read_waiteos;
                                when read_waiteos =>
                                        if eos_drp = '1' then
                                                state <= read_reg00;
                                                di_drp  <= do_drp  AND "0000001111111111";
                                                daddr   <= "1000000";
                                                den_reg <= "10";
                                                dwe_reg <= "10";
                                        else
                                                state <= read_waiteos;
                                                den_reg <= "0" & den_reg(1) ;
                                                dwe_reg <= "0" & dwe_reg(1) ;
                                        end if;
                                when read_reg00 =>
                                               daddr       <= "0000001";
                                                den_reg     <= "10";
                                                --eos_drp     <= '1';
                                                state <= reg00_waiteos;
                                when reg00_waiteos =>
                                       if eos_drp = '1' then
                                            MEASURED_VCCINT <= do_drp;
                                                den_reg <= den_reg;
                                                dwe_reg <= dwe_reg;
                                                state <= init_read; --reg01
                                        else
                                                den_reg <= "0" & den_reg(1) ;
                                                dwe_reg <= "0" & dwe_reg(1) ;
                                                state <= reg00_waiteos;
                                        end if;

                         end case;
                end if;
        end process;
```

*Annexe 10 - Notre machine d'états*

```
***** XADC Simulation Analog Data File Format ******
NAME: design.txt or user file name passed with generic sim_monitor_file
FORMAT: First line is header line. Valid column name are: TIME TEMP VCCINT VCCAUX VCCBRAM VCCPINT VCCPAUX VCCDDRO VP VN VAUXP[0] VAUXN[0] ....
TIME must be in first column.
Time value need to be integer in ns scale
Analog  value need to be real and contain a decimal  point '.', zero should be 0.0, 3 should be 3.0
Each line including header line can not have extra space after the last character/digit.
Each data line must have same number of columns as the header line.
Comment line start with -- or //
Example:
TIME TEMP VCCINT  VP VN VAUXP[0] VAUXN[0]
000  125.6  1.0  0.7  0.4  0.3  0.6
200  25.6   0.8  0.5  0.3  0.8  0.2
```

*Annexe 11 - Comment créer un fichier de stimulation*

```
architecture TB of ADC_v3_tb2 is
    component ADC_v3
        port
       (
        DCLK               : in  STD_LOGIC;
        RESET              : in  STD_LOGIC;
        VAUXP, VAUXN     : in  STD_LOGIC_VECTOR(11 downto 0);
        VP, VN             : in  STD_LOGIC;
        MEASURED_VCCINT    : out  STD_LOGIC_VECTOR (15 downto 0);
        EOC          : out  STD_LOGIC;
        EOS          : out  STD_LOGIC
        );
    end component;
```

*Annexe 12 - Déclaration de notre ADC dans le testbench*

```
UUT: ADC_v3 port map
    (DCLK => DCLK,
    RESET => RESET,
    VAUXP => VAUXP,
    VAUXN => VAUXN,
    VP => '0',
    VN => '0',
    EOC => EOC,
    EOS => EOS
    );
```

*Annexe 13 - Port map de l'ADC pour le testbench*

```
DCLK_gen : process
        begin
                DCLK <= '0'; wait for 5ns;
                DCLK <= '1'; wait for 5ns;
    end process;

RESET_gen : process
        begin
        RESET <= '1';
        wait for 0.050ms;
        RESET <= '0';
        wait for 1000ms;
    end process;
```

*Annexe 14 - Process de notre horloge et RESET*

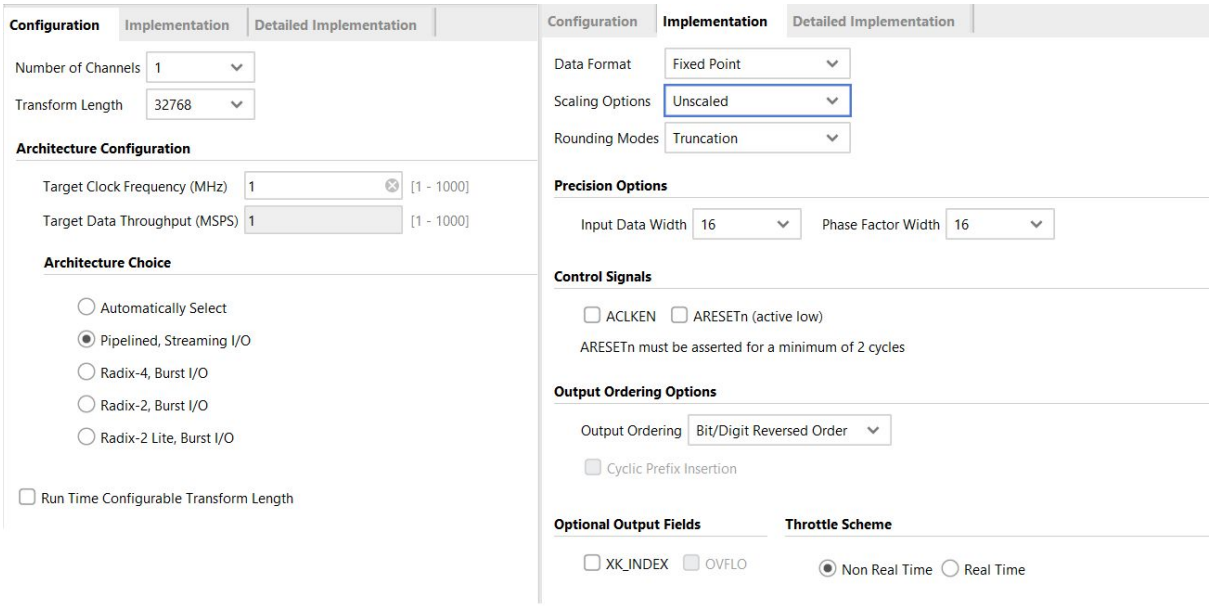
```
PROC_FILE : process
        file text_file : text open read_mode is "design.txt";
        variable text_line : line;
        variable ok : boolean;
        variable wht_time : time;
        variable vauxp_active6 : std_logic_vector(11 downto 0);
        variable vauxn_active6 : std_logic_vector(11 downto 0);
```

*Annexe 15 - Variables pour lecture de design.txt*

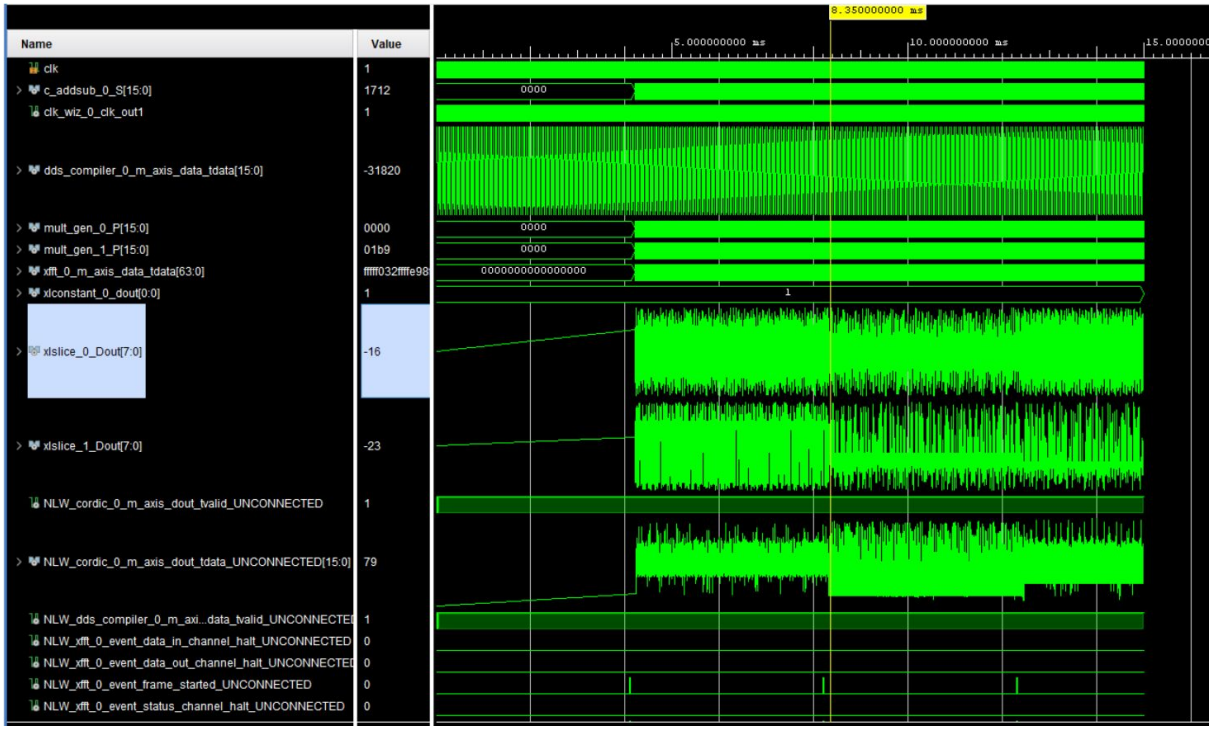
```
DECODE: process(DCLK, RESET)
    begin
    if (RESET = '1') then
--        count <= (others => '0');
        den <= '0';
        daddr <= (others => '0');
        MEASURED_VCCINT <= (others => '0');

    elsif(rising_edge(DCLK)) then
        if (drdy = '1') then
            MEASURED_VCCINT <= do_drp(15 downto 0);
        end if;
    end if;
    end process;
```

*Annexe 16 - Deuxième version de l'ADC*




*Annexe 17 - Configuration de la fft*




*Annexe 18 - Simulation fft*

```vhdl
entity one_led is
        generic(
                N : INTEGER := 8
        );
    Port(
        clk : in STD_LOGIC; --horloge de 100 MHz reçue
        enable : in STD_LOGIC; --active l'envoi de la trame
        red : in STD_LOGIC_VECTOR (N-1 downto 0); --donnée red
        green : in STD_LOGIC_VECTOR (N-1 downto 0); --donnée green
        blue : in STD_LOGIC_VECTOR (N-1 downto 0); --donnée blue
        full_frame : out STD_LOGIC --émission série de la trame
        );
end one_led;
```

*Annexe 19 - Entité one_led*

```vhdl
library myLibrary;
use work.tab_rgb_type.all;

entity all_leds is
        generic(
                Nb_leds: integer := Nb_leds_library --Nombre de LEDs, ici 120
        );
        Port(
                clk : in STD_LOGIC; --horloge de 100 MHz reçue
                enable_trame_leds : in STD_LOGIC; --Enable de N leds
                vec_rgb : in myTab_rgb; --tableau RGB pour une led, explication dans ma librairie
                trame_busy : out STD_LOGIC; --Trame occupée, émission en cours
                trame_leds : out STD_LOGIC --Emission série reliée à la sortie de one_led
        );
end all_leds;
```

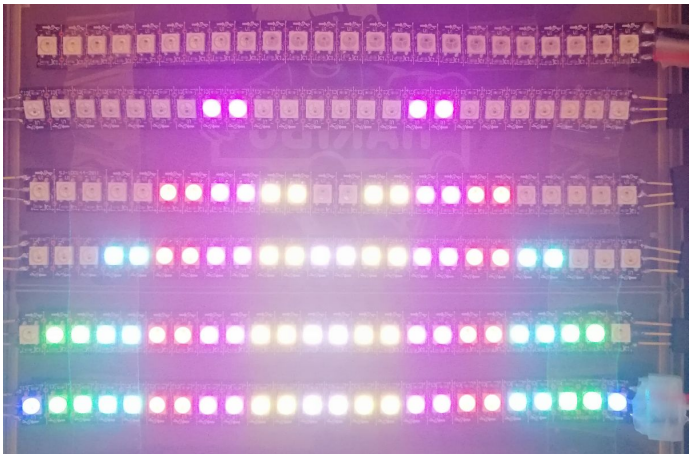
*Annexe 20 - Entité all_leds*

```
package tab_rgb_type is --Notre propre type pour pouvoir l'utiliser dans le "port" de "entity"
        constant Nb_leds_library: integer := 120; --cas général 144, ici 120
        type myTab_rgb is array(0 to (Nb_leds_library-1)) of std_logic_vector (2 downto 0);
--myTab_rgb [120][3], ce tableau a pour but de contenir l'information RGB de chaque led, un vecteur de taille 3 permet de stocker les couleurs sur 1 bit pour une couleur, nous nous sommes limités à 8 couleurs possibles par conséquent
end package tab_rgb_type;
```

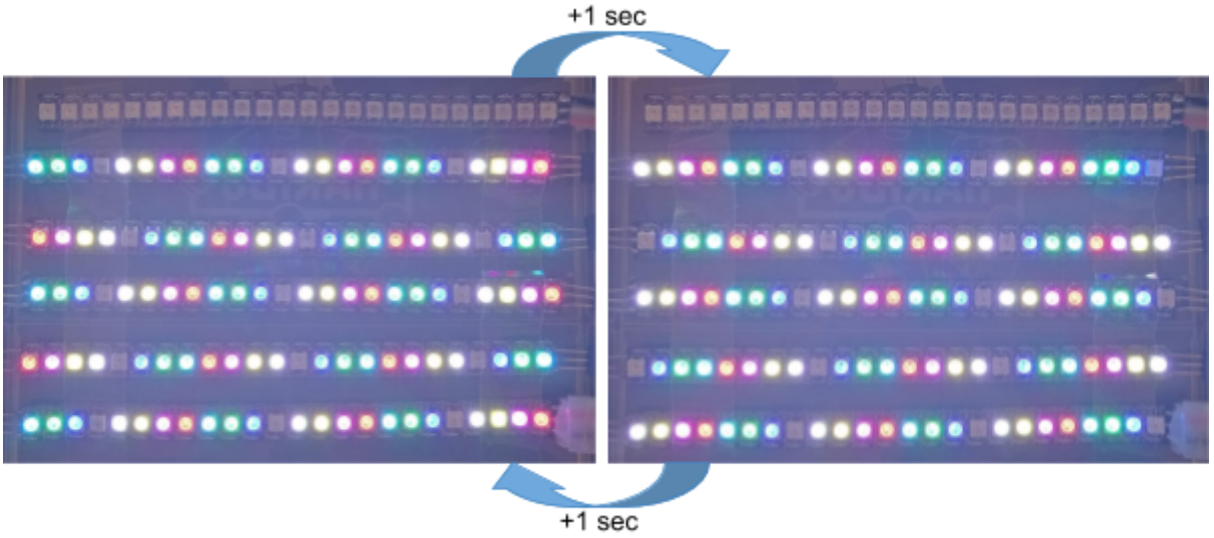
*Annexe 21 - Librairie myLibrary*

```
entity animations is
        Port(
                clk : in STD_LOGIC; --horloge de 100 MHz reçue
                enable_equalizer : in STD_LOGIC; --Réception de l'ordre d'envoi de la trame
                enable_choice : in STD_LOGIC_VECTOR (7 downto 0); --choix de l'animation, ce choix sera plus tard associé à la donnée RX reçu par le Bluetooth
                trame_busy : inout STD_LOGIC; --Trame occupée, émission en cours
                trame_equalizer : out STD_LOGIC --Émission de la trame vers le panneau LEDs
        );
end animations;
```

*Annexe 22 - Entité animations*



*Annexe 23 - Animation "égaliseur fixe"*




*Annexe 24 - Animation “couleurs rythmées”*

*Annexe 25 - Animation “chenillard”*

```
entity uart is
	generic(
		clk_freq : INTEGER := 100_000_000;--frequency of system clock in Hertz
		baud_rate : INTEGER := 9_600; --data link baud rate in bits/second
		os_rate : INTEGER := 16; --oversampling rate to find center of receive bits (in
samples per baud period)
		d_width : INTEGER := 8; --data bus width
		parity : INTEGER := 0; --0 for no parity, 1 for parity
		parity_eo : STD_LOGIC := '0' --'0' for even, '1' for odd parity
	);
	port(
		clk : IN STD_LOGIC; --system clock
		reset_n : IN STD_LOGIC; --ascynchronous reset
		tx_ena : IN STD_LOGIC; --initiate transmission
		tx_data : IN STD_LOGIC_VECTOR(d_width-1 DOWNTO 0); --data to transmit
		rx : IN STD_LOGIC; --receive pin
		rx_busy : OUT STD_LOGIC; --data reception in progress
		rx_error : OUT STD_LOGIC; --start, parity, or stop bit error detected
		rx_data : OUT STD_LOGIC_VECTOR(d_width-1 DOWNTO 0); --data received
		tx_busy : OUT STD_LOGIC; --transmission in progress
		tx : OUT STD_LOGIC --transmit pin
```

```
                );
end uart;
```

*Annexe 26 - Entité UART*

```
entity control is
        generic(
                d_width : INTEGER := 8 --data bus width
        );
        Port (
                clk : IN STD_LOGIC; --system clock
                trame_busy : IN STD_LOGIC;
                rx_busy : IN STD_LOGIC; --data reception in progress
                rx_error : IN STD_LOGIC; --start, parity, or stop bit error detected
                rx_data : IN STD_LOGIC_VECTOR(d_width-1 DOWNTO 0); --data received
                tx_busy : IN STD_LOGIC; --transmission in progress
                tx_ena : OUT STD_LOGIC; --initiate transmission
                tx_data : OUT STD_LOGIC_VECTOR(d_width-1 DOWNTO 0); --data to transmit
                reset_n : OUT STD_LOGIC; --ascynchronous reset
                leds_data : OUT STD_LOGIC_VECTOR(d_width-1 DOWNTO 0)--allumer les 8 LEDs
correspondant à la donnée RX reçue pour la visualiser sur la board
  );
end control;
```

*Annexe 27 - Entité control*