Bordeaux INP
ENSEIRB
MATMECA

Filière Électronique
Année 2021/2022
Compte-rendu de Projet Avancé

Optimisations de la réception de trames ADS-B en temps réel

Étudiants :
Léa Segaert
Léa Volpin
Cécilia Caamano
Thomas Nieddu
Léonard Cottin

Encadrant :
Bertrand Le Gal

28 janvier 2022

Table des matières

Table des figures

Acronymes

ADS-B *Automatic Dependent Surveillance - Broadcast*. 1, 2, 4, 5, 7

ADS-C *Automatic Dependent Surveillance - Contract*. 5

AVX *Advanced Vextor eXtension*. 10, 22, 24

AVX2 *Advanced Vextor eXtension 2*. 1, 2, 10, 14, 17

CRC *Cyclic Redundancy Check*. 9, 23

FPGA *Field-Programmable Gate Array*. 2, 6, 7, 18, 19, 21–24

GNSS Navigation par Système de Satellites. 5

PPM *Pulse Position Modulation*. 7, 9, 21, 22

RTL *Register-Transfer Level*. 2, 19, 20

SDR *Software Defined Radio*. 6, 7

SIMD *Single Instruction Multiple Data*. 3, 10

SSE *Streaming SIMD Extensions*. 14

UART *Universal Asynchronous Receiver Transmitter*. 18, 19, 21–24

Glossaire

IP Les IP sont des blocs de propriété intellectuelle encore appelés : IP blocks ou IP cores. Ce sont des composants virtuels sous forme de blocs utilisés dans le Design Reuse, une méthodologie de conception des circuits.. 19

NEON NEON est une extension avancée de l'architecture *Single Instruction Multiple Data* (SIMD) pour les processeurs des séries Cortex-A et Cortex-R d'ARM, comme ceux embarqués sur les cartes Raspberry Pi.. 24

x86 La famille x86 regroupe les microprocesseurs compatibles avec le jeu d'instructions de l'Intel 8086. Le terme x86 fait référence à l'architecture et au jeu d'instruction associé.. 1, 2, 10–14, 16, 17, 22

x86-64 x86-64, ou x64, est une extension du jeu d'instructions x86 d'Intel, introduite par la société AMD avec la gamme AMD64.. 10

1 Introduction

Depuis les années 2000, le nombre de passagers et donc d'avions en circulation a considérablement augmenté, faisant de ce moyen de transport le plus utilisé dans le monde après la voiture. Malheureusement, le nombre d'accidents aériens croit également. Il est donc devenu primordial de pouvoir contrôler le trafic aérien de façon précise, continue et en temps réel afin d'assurer le bon fonctionnement de ce mode de transport et d'éviter tout incident.

Avec l'essor des technologies, il est de plus en plus aisé de surveiller le trafic aérien. Certains sites Internet comme *Fly Radar*[1] en ont fait leur spécialité : on peut y voir toutes les informations sur les avions et hélicoptères affichés comme le présente la figure 1.

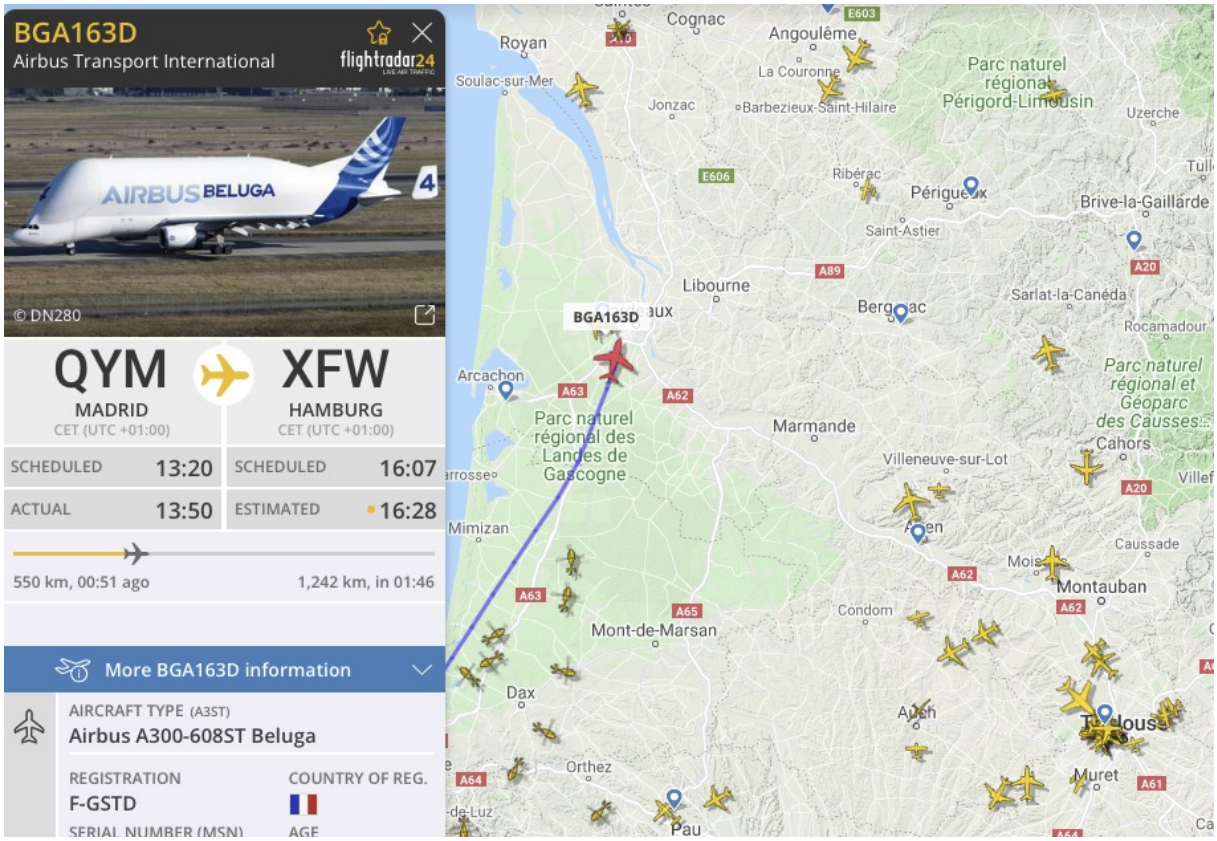


FIGURE 1 – Page d'accueil de Fly Radar et informations sur un avion [1]

L'objectif de ce projet est de reproduire une façon de capter ces avions et d'obtenir toutes les informations associées et ce, en temps réel. Pour cela, nous allons utiliser le protocole ADS-B.

2 Le protocole ADS-B

L'Automatic Dependent Surveillance (ADS) est un moyen coopératif pour le contrôle aérien de connaître la position des avions à chaque instant. Ce système de surveillance est né avec le constat suivant : le développement des satellites a permis aux avions modernes de connaître leur position en temps réel de façon plus précise que les moyens de la surveillance au sol. Les avions équipés de ce système déterminent leur position à l'aide d'un système de Géolocalisation et de Navigation par Système de Satellites (GNSS) avant de l'envoyer aux stations intéressées.

Il existe deux types de surveillance :
- `ADS-C (Contract)` : la surveillance repose sur un mode connecté. L'échange de données entre les avions et les stations nécessite une connexion préalable entre les deux parties. Une fois établie, l'avion peut envoyer ses informations au destinataire ;
- `ADS-B (Broadcast)` : la surveillance repose sur un mode de diffusion. L'avion envoie périodiquement ses informations sans établir de connexion. Les utilisateurs intéressés peuvent alors avoir ces données librement.

L'inconvénient de l'ADS-C est l'établissement de la liaison : souvent satellite, elle est très coûteuse. Cette surveillance est donc principalement utilisée dans les zones désertiques ou océaniques à des fréquences d'émission faibles (de l'ordre de plusieurs minutes). A l'inverse, l'ADS-B ne nécessite pas de radar pour fonctionner mais une simple antenne, faisant diminuer son coût d'utilisation. Cette surveillance est donc principalement développée et de nombreux amateurs peuvent également en bénéficier.

Trois liaisons sont disponibles pour l'ADS-B :
- `1090 MHz Extended Squitter (1090ES)` : il s'agit d'une extension des transpondeurs mode S et TCAS qui émettent sur une fréquence de 1090MHz. La plupart des avions commerciaux sont équipés de ces transpondeurs, c'est donc une liaison peu coûteuse pour eux ;
- `Universal Access Transponder (UAT 978)` : c'est un transpondeur américain spécialement conçu pour l'ADS-B qui fonctionne à 978MHz. Il s'agit d'un système développé pour l'aviation générale qui vise à pallier le coût d'installation de la liaison précédente ;
- `VHF Data Link Mode 4 (VDL mode 4)` : c'est un moyen de surveillance et de communication opérant dans la bande VHF aéronautique pour les moyens de navigation (108-118 MHz). Cet outil est plus abordable que le 1090ES.

Pour ce projet, nous utiliserons le système de surveillance ADS-B avec une liaison 1090ES. Nous ciblerons donc majoritairement les avions commerciaux.

3 Le projet

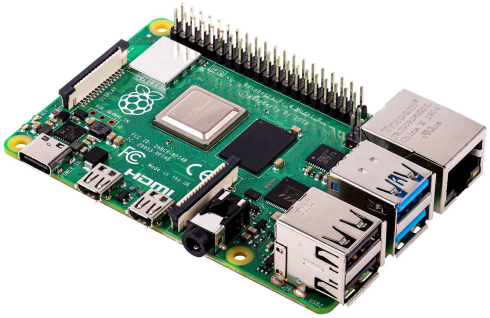
Nous ne sommes pas partis de rien pour ce projet. En effet, M Le Gal nous a donné accès à son dépôt GitHub [2], contenant un projet déjà fonctionnel sur le matériel qui nous est fourni, afin que nous puissions l'étudier et le comprendre.

3.1 Matériel à disposition

Afin de mener à bien ce projet, M Le Gal nous a prêté du matériel. Nous disposons donc d'une antenne (figure 2a) pour capter les trames émises par les avions et d'un module *Software Defined Radio* (SDR), visible en figure 2b, pour transformer les signaux analogiques en numériques. Par ailleurs, le code que nous avons récupéré est fonctionnel sur une carte Raspberry Pi 4B dont une image est donnée figure 2c.

(a) Antenne

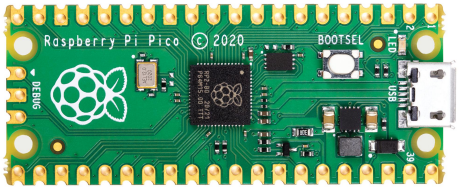
(b) Module SDR



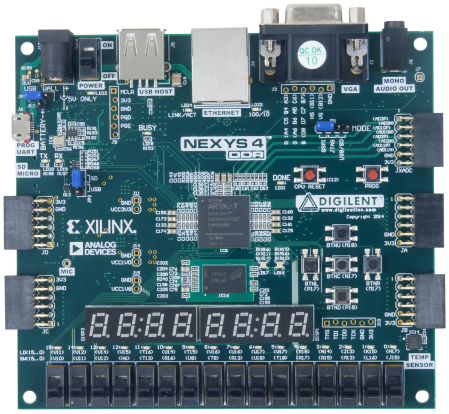
(c) Raspberry Pi 4B

FIGURE 2 – Matériel de réception et de traitement des données reçues

Pour une meilleure intégration et afin de réduire le coût énergétique de notre projet, l'un des objectifs est d'intégrer notre code sur une Raspberry Pi Pico (figure 3a) voire sur une carte *Field-Programmable Gate Array* (FPGA) Nexys A7 (voir figure 3b) qui, au-delà d'être très faible consommation, est capable d'exécuter du calcul en parallèle.



(a) Raspberry Pi Pico



(b) Carte FPGA Nexys A7

FIGURE 3 – Matériel visé pour l'intégration du code

3.2 Détail d'une trame de données

Nous récupérons en sortie module SDR, des trames binaires comme on peut le voir en figure 4. Cette trame est composée d'un préambule et d'un bloc de données.

Le préambule sert à identifier la nature de la trame. Le protocole ADS-B utilise la *Pulse Position Modulation* (PPM) pour le transport des données. Ainsi nous nous retrouvons avec des couples de bits : seule la moitié de ce qui est représenté dans le data block en figure 4 est pertinente.

En outre, une fois qu'une trame est réceptionnée, plusieurs étapes sont nécessaires afin d'extraire les données ordonnées comme en figure 5.

Comme on peut le voir, nous avons accès à pléthore d'informations telles que la longitude, la latitude, l'altitude, la vitesse, le nom de l'avion, etc.

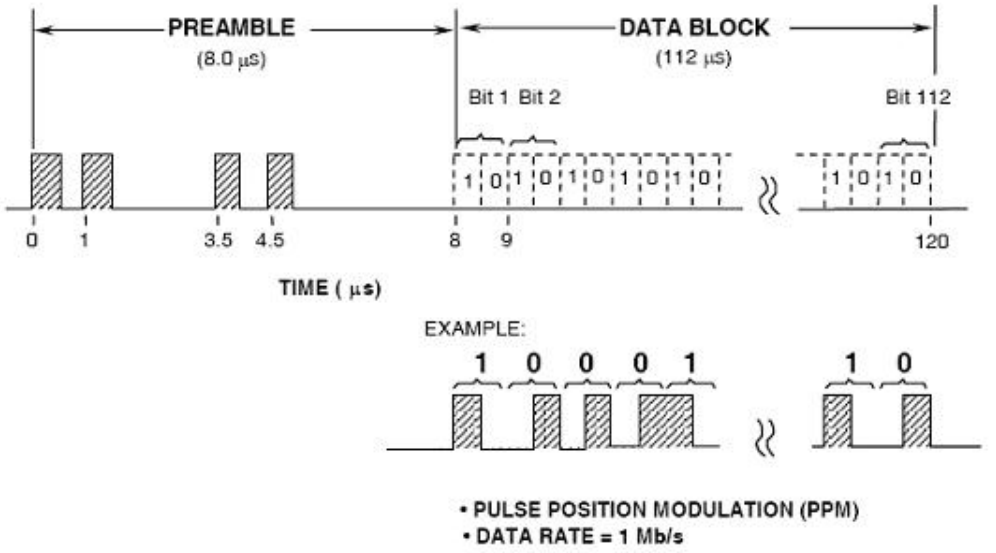


FIGURE 4 – Format d'une trame ADS-B [3]

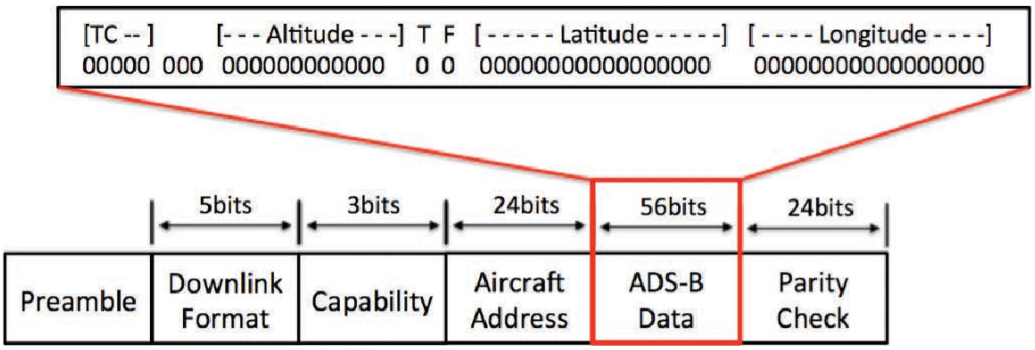


FIGURE 5 – Informations transmises par les avions contenues dans le data block [4]

3.3 Chaîne de réception des données

En réception, le principe est d'effectuer une écoute continue du canal à la recherche de la séquence binaire qui marque le début d'une trame. Les systèmes des radio-logicielles imposent que les échantillons qui arrivent en réception soient de type complexe, donc qu'il y ait pour chaque échantillon une voie I et Q.

Lorsque nous avons récupéré le travail déjà effectué sur ce projet, la chaîne de communication en réception était déjà composée, au niveau du programme, des éléments visibles sur la figure 6 ci-dessous.

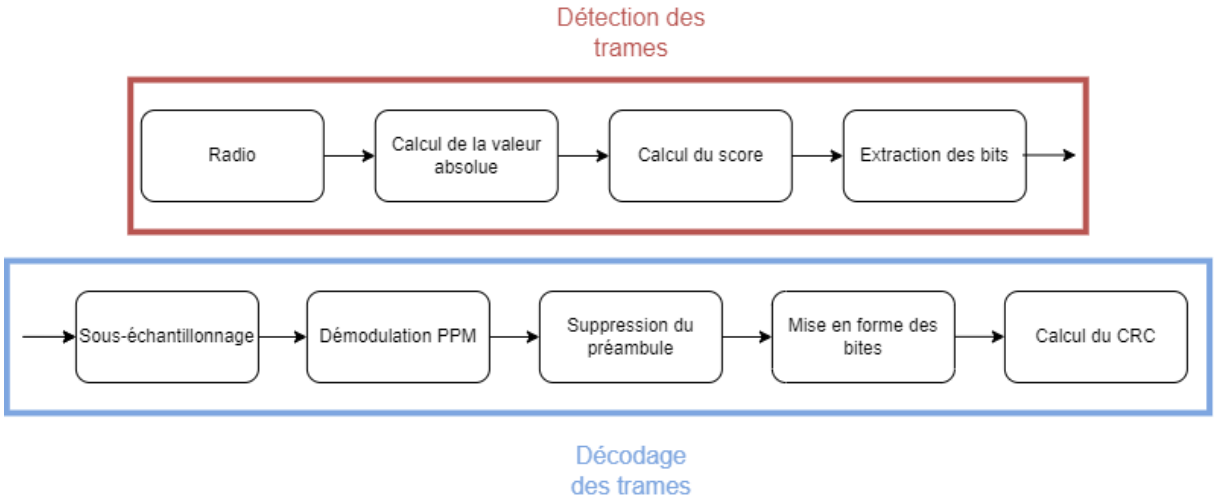


FIGURE 6 – Chaîne de réception

Neuf éléments composent cette chaîne, dont les noms et les rôles propres à chacun sont :

— `Le module radio` : Le module radio effectue une écoute continue du canal pour détecter les trames à partir de leur séquence binaire caractéristique ;
— `Le calcul de la valeur absolue` : Cet élément permet de passer d'un nombre complexe à un nombre réel par le calcul du module ;
— `Le calcul du score` : Un corrélateur permet de détecter le début d'une nouvelle trame, et une trame temporaire est alors formée. Ce module permet de calculer un score et de le comparer à une somme d'échantillons caractéristiques d'une trame.

Si le score dépasse un certain seuil, la trame détectée peut être considérée comme complète ;
— `L'extraction des bits` : Si le score dépasse le seuil fixé, ce module sert à extraire les 120 bits qui composent la trame à décoder ;
— `Le sous échantillonnage` : Ce module fait la somme de deux termes consécutifs. Il y a donc moitié moins d'éléments en sortie ;
— `La démodulation PPM` : Ce module permet de comparer deux termes consécutifs, si celui de gauche est supérieur à celui de droite, un 1 est renvoyé sinon cela sera un 0 ;
— `La suppression du préambule` : Les 8 premiers bits, correspondant au préambule expliqué dans la partie 3.2, sont supprimés ;
— `La mise en forme des bits` : Cet élément permet de créer des octets à partir de paquets de 8 bits ;
— `Le calcul du CRC` : Le *Cyclic Redundancy Check* (CRC) permet au destinataire de vérifier que les trames reçues sont bonnes. Pour cela on vérifie que les CRC calculés avant et après l'envoi ont la même valeur. Si ce n'est pas le cas, il existe des techniques afin de corriger la trame reçue.

3.4 Objectifs du projet

Lorsque nous avons pris en main le projet, nous avons pu établir une liste des objectifs à réaliser pour ce projet. Les deux principaux sont :

1. Traiter les données qui nous parviennent du module SDR en temps réel ;
2. Utiliser du matériel toujours plus petit et plus facilement intégrable pour réduire le coût et la consommation.

Ainsi, pour mener à bien ce projet, nous avons séparé notre groupe en deux sous-groupes afin de mener le développement hardware et software en parallèle.
— Groupe `Software` : composé de Cécilia, Thomas et Léonard, ce groupe est en charge de transformer la chaîne de réception pour manipuler des nombres entiers. Il doit également apporter diverses optimisations sur le code pour d'avantage d'efficacité calculatoire ;
— Groupe `Hardware` : composé de Léa V. et Léa S., l'objectif est d'implémenter la chaîne sur une carte FPGA.

4 Optimisations logicielles

L'un de nos principaux objectifs pour ce projet est d'effectuer des optimisations logicielles et d'adapter le programme fourni afin de gagner en efficacité sur les calculs et en rapidité d'exécution. Cet objectif a pour finalité l'intégration des fonctionnalités sur une Raspberry Pi Pico. Pour cela, nous avons décidé de modifier le format de données des nombres manipulés dans la chaîne de réception. Le programme que nous avons récupéré utilise des nombres flottants qui sont plus compliqués et dont les opérations les manipulant sont plus coûteuses. Nous sommes partis du principe que nous pouvions gagner en efficience en ne manipulant que des entiers plutôt que des nombres flottants. La figure 7 présente les changements à effectuer. Notre travail porte essentiellement sur les trois

premiers blocs de la chaîne à savoir la réception, le calcul du module et le calcul du score.

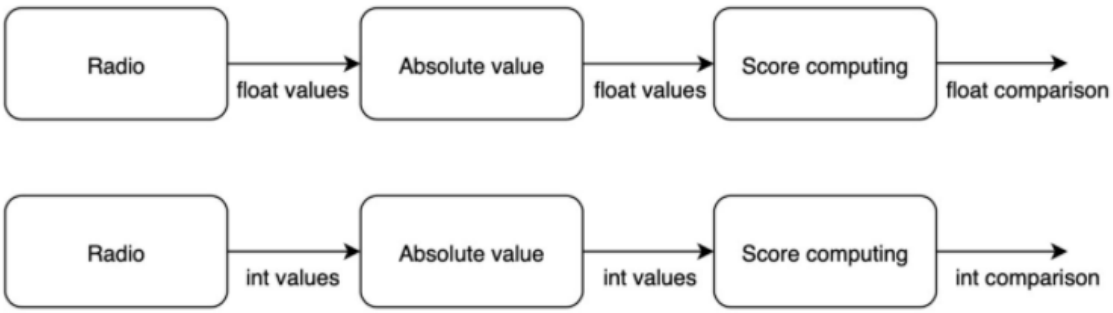


FIGURE 7 – Passage d'une chaîne de réception manipulant des nombres flottants à une manipulant des nombres entiers

Le code source du projet de Monsieur Le Gal, est écrit pour plusieurs jeux d'instructions de processeur. Monsieur Le Gal dispose d'un ordinateur très certainement assemblé après 2003, embarquant un processeur Intel de génération x86-64 qui est l'architecture étendue 64 bits préservant l'intégralité de l'architecture x86 32 bits [5]. Cette génération de processeurs qui équipent nos ordinateurs dans la très grande majorité des cas dispose donc d'un jeu d'instructions x86. Celui de Monsieur Le Gal dispose en outre de l'extension *Single Instruction Multiple Data* (SIMD), nommée AVX2 (*Advanced Vextor eXtension* (AVX) datant de 2008 et AVX2 de 2013).

Un jeu d'instructions regroupe l'ensemble des opérations que peut effectuer un processeur en spécifiant comment elles sont codées (type d'opération, registre source, registre de destination, adresse mémoire source, adresse mémoire de destination ...).

Nous avons à notre disposition une version séquentielle (x86) et une version parallèle (AVX2) toutes les deux manipulant des nombres flottants. Nous allons donc nous atteler à créer deux nouvelles versions, effectuant des calculs sur des nombres entiers, pour les jeux d'instructions x86 et AVX2.

Le passage des nombres flottants en nombres entiers a nécessité une bonne compréhension de l'ensemble des algorithmes composant la chaîne. Par ailleurs, nous avons dû faire des choix stratégiques quant à la façon d'effectuer les différentes opérations mathématiques et logiques car toutes les opérations/conversions ne sont pas disponible dans ces jeux d'instructions.

4.1 Optimisations pour le jeu d'instructions de l'architecture x86

Nous avons commencé par prendre en main le code écrit pour un jeu d'instruction générique à une architecture : la x86. En effet, celui-ci est le plus simple et nous permet de nous concentrer sur la compréhension de l'algorithmique plus que l'implémentation.

Lorsque nous avons étudié le programme, nous nous sommes rendu compte que celui-ci ne fonctionnait pas correctement. En effet, plusieurs opérations sur les vecteurs étaient mal implémentées provoquant l'agrandissement des vecteurs de données plutôt que leur écrasement. Cela avait pour conséquence un temps de détection extrêmement élevé comme nous pouvons le voir sur la figure 8. Notre objectif sur cette partie est donc double : modifier le programme pour manipuler des entiers et réussir à tenir la contrainte de temps réel.

```
Nombre d'acquisition réalisées  : 639
Temps moyen par acquisition     : 615.08 ms
Contrainte tps réel / iter      : 16.384 ms
Number of frames over the detection value      : 22991186
Number of frames with validated CRC value      : 499447
 - Number of initially correct  CRC values     : 499447
 - Number of saved frames with bit-flip (x1)   : 0
 - Number of saved frames with bit-flip (x2)   : 0
 - Number of saved frames with bit-flip (x3)   : 0
Number of discarded frames with strange values : 940 (type != 17)

Nombre de trames emises  (frames)  = 499447

Temps total : 393036 ms
```

FIGURE 8 – Performances avec le jeu d'instructions x86 que nous avons récupéré

4.1.1 Réécriture du code pour des nombres entiers

Le code que nous avons récupéré fonctionne avec des nombres flottants normalisés entre -1 et 1 avec une division par 127. En effet, les données émises par la radio sont des entiers signés sur 8 bits. La première étape a donc été de changer la façon dont sont traités les nombres complexes. En sortie du module radio, nous sommes donc passés de données `complex<float>` à des données de type `complex<int8_t>`, où `int8_t` est un entier signé normalisé sur 8 bits.

Par la suite, nous avons dû modifier les autres éléments de la chaîne et notamment les liens entre les différentes classes du programme. L'algorithme de calcul du module des nombres complexes a été légèrement modifié : nous ne calculons plus le module mais le module au carré des nombres complexes. Ce léger changement permet de nous affranchir du calcul coûteux qu'est la racine carrée. Les parties réelles et imaginaires des nombres complexes signés étant codées sur 8 bits, nous calculons le carré du module sur un nombre entier non signé de 16 bits.

Enfin, nous avons adapté le calcul du score pour tenir compte de la sortie du bloc précédent. Dans la première version, le score retournait un score compris entre 0 et 1. Puisque nous souhaitons utiliser des nombres entiers, nous avons modifié la formule mathématique associée pour obtenir une nouvelle formule capable de sortir ce même score sur une plage de valeurs allant de 0 à 65535, couvrant ainsi toutes les valeurs possibles d'un nombre entier non signé sur 16 bits. Nous remarquons également que nous nous servons du module et du module au carré des valeurs. Une idée a alors été de calculer les différentes racines carrées des modules au début du programme dans un tableau `ModSqrt` et de venir en faire une lecture par la suite. Nous avons donc fait le choix de gagner en nombre de calcul au détriment de la bande passante.

La formule que nous avons récupérée (équation 1) suppose que le vecteur x est composé des modules des complexes. Avec nos changements, nous obtenons l'équation 2 dans laquelle le vecteur x contient cette fois-ci le carré des modules.

$$score = \frac{\sum_{i\in\{0;1;4;5;14;15;18;19\}} x[i]}{\sqrt{8 \times \sum_{i=0}^{32} x[i]^2}} \in [0;1] \tag{1}$$

$$score = \frac{\left(\sum_{i\in\{0;1;4;5;14;15;18;19\}} ModSqrt[\ x[i]\ ]\right)^2}{8 \times \sum_{i=0}^{32} x[i]} \times 65535 \in [0;65535] \tag{2}$$

Une fois ces modifications implémentées, et les changements à faire dans le programme main() appliqués, nous testons notre programme. Celui-ci est fonctionnel et relativement efficace comme l'indique la figure 9. Toutefois, nous ne sommes pas encore totalement satisfaits de la façon dont est calculé le score. Toujours en quête de performances, nous décidons de déformer légèrement la formule.

```
Nombre d'acquisition réalisées  : 5650
Temps moyen par acquisition     : 1.69204 ms
Contrainte tps réel / iter      : 16.384 ms
Number of frames over the detection value      : 308657
Number of frames with validated CRC value      : 12575
 - Number of initially correct  CRC values     : 12575
 - Number of saved frames with bit-flip (x1)   : 0
 - Number of saved frames with bit-flip (x2)   : 0
 - Number of saved frames with bit-flip (x3)   : 0
Number of discarded frames with strange values : 16 (type != 17)

Nombre de trames emises  (frames)  = 12575

Temps total : 9560 ms
```

FIGURE 9 – Performances obtenues avec le jeu d'instruction pour l'architecture x86

4.1.2 Ajouts de diverses optimisations

Le programme est fonctionnel en l'état mais nous pouvons encore améliorer ses performances. Par exemple, nous calculons le score de chaque trame que nous retournons puis que nous comparons au score de référence dans le main. Une idée pourrait être de renvoyer directement un booléen indiquant si la trame pour laquelle nous venons de calculer le score dépasse le seuil ou non. Nous allons nous concentrer sur ce point. Par ailleurs, deux points nous gênent dans la formule mathématique que nous utilisons : la division et les facteurs multiplicateurs. En effet, nous avons enlevé le calcul de la racine carrée afin de gagner en efficacité, il est donc cohérent de vouloir se séparer de la division.

Pour réaliser cela, nous passons le seuil en argument de la méthode et remanions l'expression mathématique pour s'affranchir de la division. Nous retournerons alors le résultat de la comparaison entre le numérateur et le dénominateur multiplié par le score minimal. Par ailleurs, nous multiplions par 65535 au numérateur ($\approx 2^{16}$) et par 8 (2^3) au dénominateur. Nous pouvons assez facilement combiner ces opérations en une seule à l'aide d'un décalage de 13 bits vers la gauche. Finalement, l'opération mathématique à réaliser est donnée en équation 3.

$$trame\_ok = \left(\sum_{i \in \{0;1;4;5;14;15;18;19\}} ModSqrt[\, x[i] \,] \right)^2 \geq \left(\left[seuil \times \sum_{i=0}^{32} x[i] \right] >> 13 \right) \quad (3)$$

Un autre changement que nous avons réalisé a été de faire sortir le module ainsi que le module au carré du bloc de calcul. Cela revient à faire la lecture mémoire dans le bloc de calcul de la valeur absolue plutôt que dans le bloc de calcul du score. Cela ne change rien pour cette partie, en revanche cela nous sera utile par la suite.

Avec ces nouvelles optimisations, nous n'avons pas modifié le fonctionnement du programme mais nous avons en revanche encore gagné en efficacité de calcul. En effet, comme nous pouvons le voir sur la figure 10, le temps de traitement moyen par acquisition a diminué. Toutefois, nous n'avons pas réussi à faire descendre ce résultat plus bas ni à passer la barre symbolique de la milliseconde. Pour atteindre des résultats encore meilleurs, nous allons devoir utiliser des instructions permettant de paralléliser les calculs afin d'effectuer plusieurs opérations simultanément.

```
Nombre d'acquisition réalisées  : 5650
Temps moyen par acquisition     : 1.21469 ms
Contrainte tps réel / iter      : 16.384 ms
Number of frames over the detection value      : 305227
Number of frames with validated CRC value      : 12573
 - Number of initially correct  CRC values     : 12573
 - Number of saved frames with bit-flip (x1)   : 0
 - Number of saved frames with bit-flip (x2)   : 0
 - Number of saved frames with bit-flip (x3)   : 0
Number of discarded frames with strange values : 14 (type != 17)

Nombre de trames emises  (frames)  = 12573

Temps total : 6863 ms
```

FIGURE 10 – Performances avec toutes les optimisations du jeu d'instructions x86

4.2 Optimisations pour le jeu d'instructions AVX2

4.2.1 Le jeu d'instruction AVX2

Maintenant que nous nous sommes familiarisés avec l'algorithmique du programme, nous allons tenter d'améliorer la rapidité d'exécution du code en utilisant le jeu d'instructions AVX2. Celui-ci a la particularité de manipuler des registres de 256 bits, permettant ainsi d'effectuer jusqu'à huit calculs en une fois et donc de gagner en performance comme le présente la figure 11. En effet, l'AVX2 étend le jeu d'instructions du *Streaming SIMD Extensions* (SSE). Il s'agit d'un jeu de 70 instructions supplémentaires spécifiques aux microprocesseurs x86 d'Intel permettant la manipulation de registres de 128 bits.

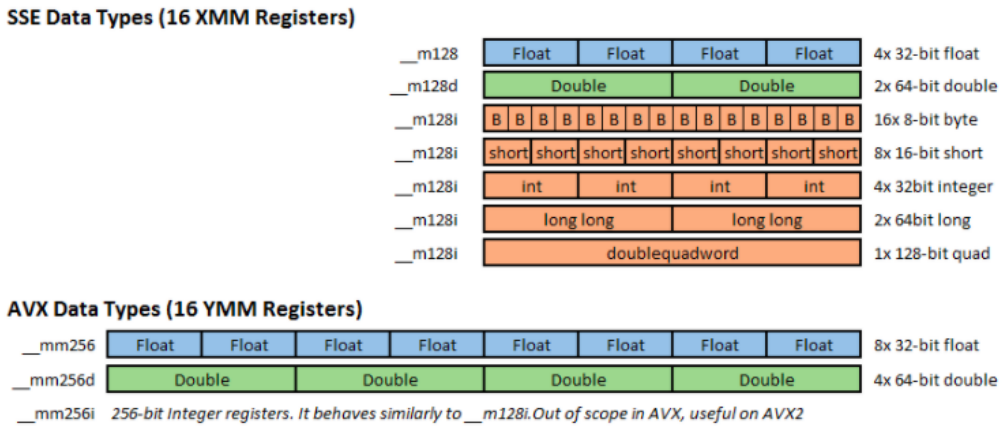


FIGURE 11 – Représentation des données sur le jeu d'instructions AVX2 [6]

Grâce à l'utilisation de ces instructions, nous espérons gagner en rapidité d'exécution. Nous pouvons gagner en théorie un facteur 8, seulement en pratique, nous sommes davantage contraints par les lectures et les écritures en mémoire que nous n'avons pas nécessairement lors de l'utilisation du jeu d'instructions de l'architecture x86.

4.2.2 Réécriture du programme

Le programme, tel que nous l'avons récupéré, ne contient des instructions spécifiques AVX2 que dans les blocs de calcul du module et de calcul du score. Ce sont donc les deux seules parties qu'il nous faut modifier. Nous avons étudié les instructions disponibles pour la manipulation d'entier dans la documentation officielle d'Intel [7]. Nous nous sommes alors rendu compte que nous manquions d'instructions pour faire un simple "copier-coller" des instructions flottantes en instructions entières. Nous avons ainsi dû réfléchir à un moyen de reproduire les mêmes calculs avec d'autres opérations, en modifiant le processus calculatoire.

Nous nous sommes tout d'abord penchés sur le calcul du module des nombres complexes. Afin de minimiser le coût de cette opération, nous avons tenté de conserver un maximum d'instructions binaires car celles-ci s'exécutent en un seul cycle d'horloge. Les parties réelles et imaginaires des nombres complexes étant représentées sur 8 bits, nous pouvons charger 32 valeurs soit 16 complexes dans un registre de 256 bits. L'enchaînement des opérations réalisées pour calculer le module et le module au carré est le suivant :

1. Chargement des 32 valeurs de 8 bits dans un registre de 256 bits ;
2. Extraction sur 8 bits des signes de chaque nombre ;
3. Séparation de la partie réelle et de la partie imaginaire de chaque complexe ;
4. Décalage de 8 bits à gauche pour désaligner les nombres de leur signe ;
5. Extension de signe sur 16 bits de chaque partie réelle et imaginaire extraite à l'aide d'un "OU" logique ;
6. Mise au carré des parties réelle et imaginaire ;
7. Addition sur 16 bits non signés pour obtenir les 16 modules au carré.

Par la suite, nous souhaitons également extraire la racine carrée de ce module. Pour cela, nous souhaitons reproduire le mécanisme de lecture dans un tableau de valeurs pré-calculées. Des étapes intermédiaires sont cependant nécessaires car la fonction à utiliser ne prend des valeurs que sur 32 bits. Nous étendons ainsi nos modules au carré précédemment calculés sur 32 bits à l'aide de la suite d'instructions suivante :

1. Division du vecteur en deux à l'aide d'un masque et d'un "ET" logique ;
2. Extraction des 128 bits non nuls ;
3. Extension sur 32 bits ;
4. Lecture des racines carrées des modules dans le tableau des valeurs pré-calculées.

Cet enchaînement d'étape est résumé dans l'image 12 uniquement pour la partie réelle, la partie imaginaire est traitée de la même façon. Nous observons sur la figure le module des valeurs en bleu et le carré des modules en rouge. Afin de tester le bon fonctionnement du programme, nous l'avons testé dans un programme test séparé en charge uniquement de calculer le module et le module au carré de valeurs générés aléatoirement. Après vérification et validation de ce module, nous pouvons passer au bloc de calcul du score.

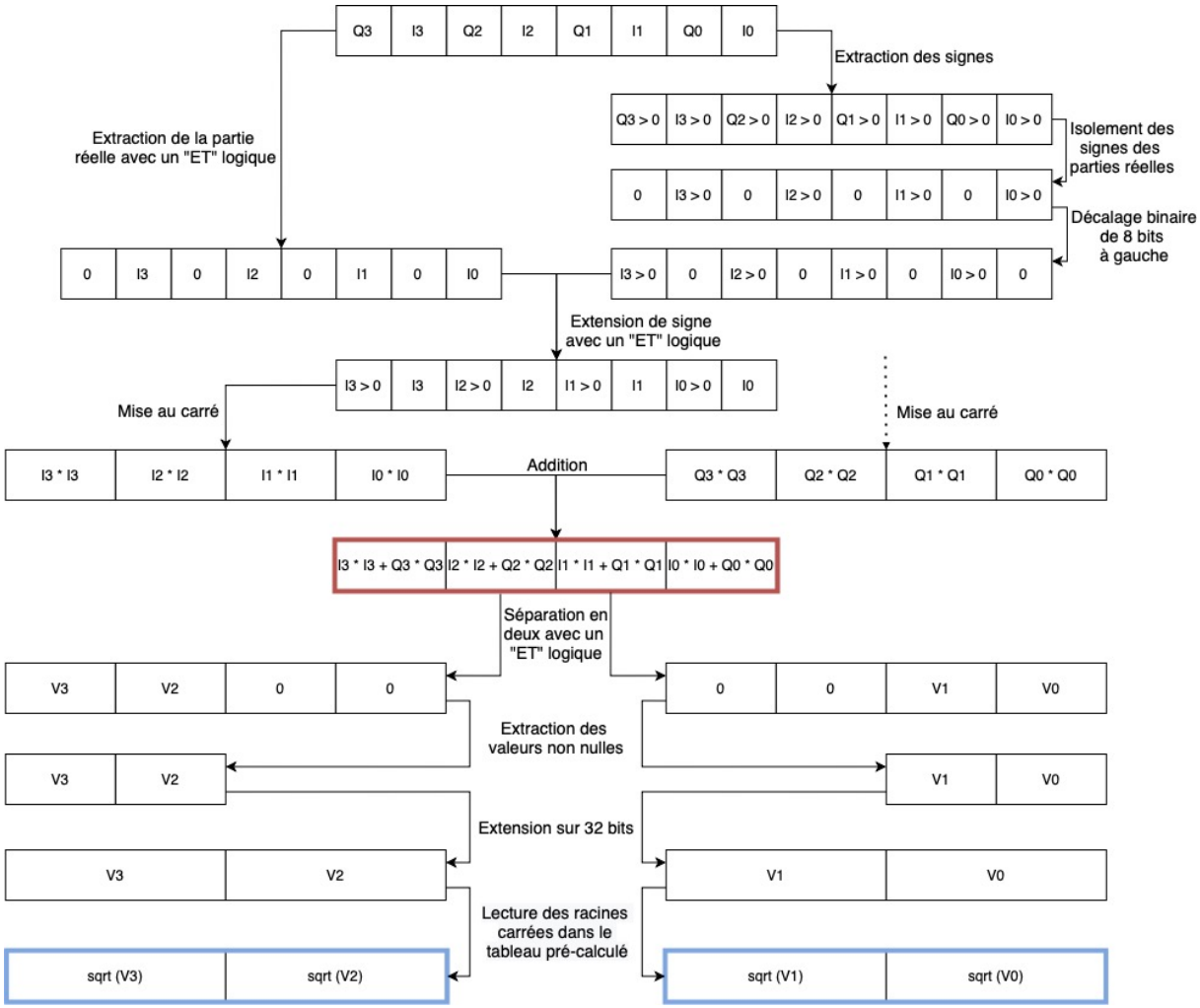


FIGURE 12 – Étapes de calcul du module et de son carré

La réécriture du module de calcul de score nous a posé légèrement moins de problèmes, nous disposons de toutes les instructions nécessaires. Cependant, un nombre flottant sur 32 bits dispose d'une amplitude plus importante qu'un entier de la même taille. Pour palier à ce problème, nous chargeons dans nos registres de 128 bits 8 valeurs sur 16 bits que nous étendons ensuite sur 32 bits. Nous effectuons la plupart des calculs mais lorsque nous réalisons des opérations pouvant mener à des débordements mémoire, nous convertissons nos données entières en données flottantes et nous effectuons la fin du processus. Lorsque nous avons terminé et que nous avons le résultat souhaité, à savoir si une trame est valide ou non, nous écrivons en mémoire ce résultat après l'avoir converti de nouveau en entier.

De la même façon que pour le programme visant l'architecture x86, nous avons dans un premier temps retourner le score de la trame que nous comparons dans le main au score minimal. Par la suite, nous avons adapté l'algorithme afin de nous affranchir de la division. Ces diverses améliorations nous ont permis de gagner de précieuses instructions lors de l'exécution et donc de gagner en efficacité et en rapidité.

```
Nombre d'aquisition réalisées  : 5650
Temps moyen par acquisition    : 0.735575 ms
Constrainte tps réel / iter    : 16.384 ms
Number of frames over the detection value      : 505258
Number of frames with validated CRC value      : 12556
 - Number of initially correct  CRC values     : 12556
 - Number of saved frames with bit-flip (x1)   : 0
 - Number of saved frames with bit-flip (x2)   : 0
 - Number of saved frames with bit-flip (x3)   : 0
Number of discarded frames with strange values : 24 (type != 17)

Nombre de trames emises  (frames)  = 12556

Temps total : 4156 ms
```

(a) Version flottante

```
Nombre d'acquisition réalisées  : 5650
Temps moyen par acquisition    : 0.528319 ms
Contrainte tps réel / iter    : 16.384 ms
Number of frames over the detection value      : 308385
Number of frames with validated CRC value      : 12575
 - Number of initially correct  CRC values     : 12575
 - Number of saved frames with bit-flip (x1)   : 0
 - Number of saved frames with bit-flip (x2)   : 0
 - Number of saved frames with bit-flip (x3)   : 0
Number of discarded frames with strange values : 16 (type != 17)

Nombre de trames emises  (frames)  = 12575

Temps total : 2985 ms
```

(b) Version entière

FIGURE 13 – Performances obtenues avec le jeu d'instructions AVX2

4.3 Comparaison des résultats obtenus

Une fois nos différents algorithmes implémentés, nous pouvons comparer leurs performances.

Nous avons désormais une version fonctionnelle du projet sur l'architecture x86. Cependant, cette version est la plus lente de toutes. Pour palier à cette lenteur d'exécution, nous avons une version fonctionnant avec le jeu d'instructions AVX2. Cette version est la plus rapide de toutes et notamment plus rapide que son équivalent utilisant des nombres flottants. Le tableau 1 permet de faire un récapitulatif du nombres de cycles d'horloges nécessaires au traitement des données. Globalement, les versions pour AVX2 nécessitent le même nombre d'instructions. Cependant, les instructions sur les nombres entiers étant plus rapide, le programme s'exécute plus rapidement.

Enfin, la figure 14 montre le programme fonctionnel lorsque la radio est branchée directement sur l'ordinateur. On détecte les avions et on récupère les informations comme nous le voulions.

	x86 int	AVX2 float	AVX2 int
Conversion (nombre de cycles)	184510	70264	67749
Détection (nombre de cycles)	1130743	287570	299497
Temps de traitement moyen par trame (ms)	1,21	0,73	0,52

TABLE 1 – Tableau bilan des performances logicielles

FIGURE 14 – Illustration du programme fonctionnel

5 Implémentation matérielle

Nous avons un autre objectif qui est d'implémenter la chaîne de réception sur FPGA. Le but est de réduire les coûts matériels et la consommation d'énergie. Pour cela, nous disposons d'un code C/C++, de l'outil Vivado HLS, de Vivado et d'un FPGA : une Nexys A7.

Nous devons implémenter la chaîne bloc par bloc pour être sûr de ne rien "casser". Nous avons alors commencé à implémenter le bloc de sous-échantillonnage.

5.1 Implémentation du module de sous-échantillonnage

Pour pouvoir implémenter ce bloc, nous devons tout de même garder le reste des calculs de la chaîne sur l'ordinateur. Il y a alors une liaison UART entre l'ordinateur et le FPGA comme illustré dans la figure 15.

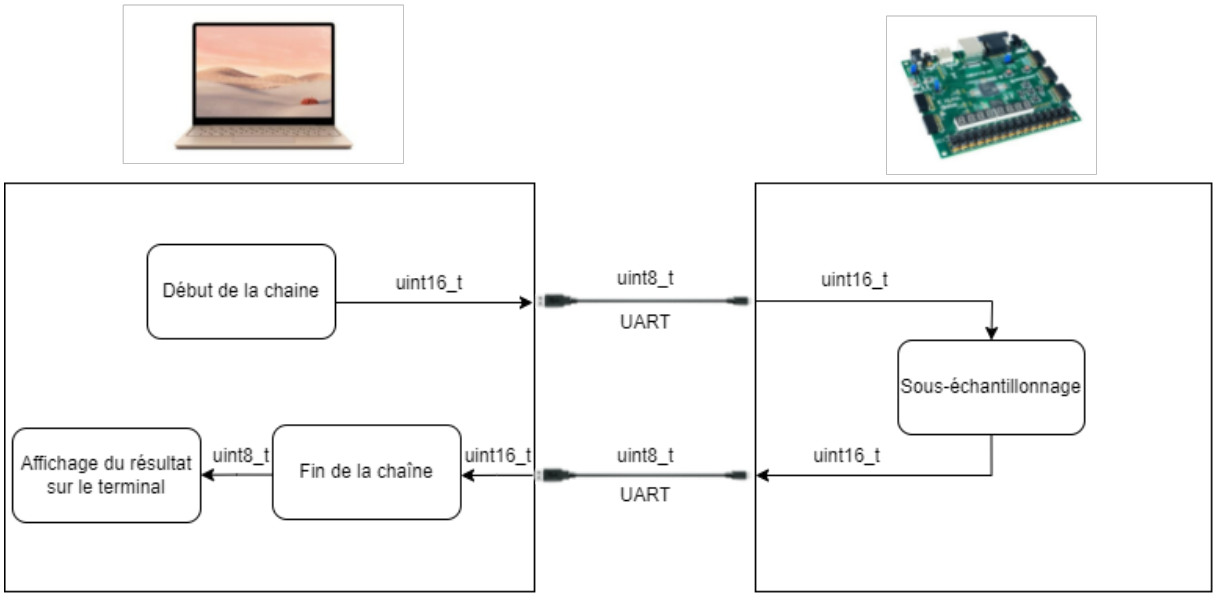


FIGURE 15 – Implémentation du sous-échantillonnage sous FPGA

5.1.1 Synthèse d'architecture : Vivado HLS

Vivado HLS est un outil de synthèse nous permettant de générer une architecture matérielle à partir d'un code C/C++. Dans notre cas nous générons une architecture VHDL que nous implantons sur le FPGA.

Dans notre architecture, nous manipulons des entiers non signés sur 16 bits. Cependant, la liaison UART qui relie l'ordinateur au FPGA ne supporte au mieux que des entiers sur 8 bits. Nous avons donc adapté notre code C/C++ sur HLS pour résoudre ce problème. Nous avons décidé de découper notre code en trois étapes comme observé sur la figure 16. La première étape est de convertir les entiers sur 8 bits en un mot de 16 bits grâce à la fonction `strm_bytes2words`. La deuxième étape est d'effectuer le sous-échantillonnage sur les entiers de 16 bits. La dernière étape est de convertir les mots de 16 bits en octet pour qu'ils puissent être renvoyés à l'ordinateur. Pour cela, nous utilisons la fonction `strm_words2bytes`. Les deux fonctions de conversions nous ont été fournies et sont contenues dans les fichiers suivants : `bytestrm_dwordproc.cpp`, `bytestrm_dwordproc.h` et `bytestrm_util.h`.

```
strm_bytes2words<uint16_t, sizeof(uint16_t), false>(downstrm_in, strm_in, 480 * sizeof(uint16_t));

DownSamp16<uint16_t, 480, 2>(downstrm_in,downstrm_out);

strm_words2bytes<uint16_t, sizeof(uint16_t), false>(strm_out,downstrm_out, 240 * sizeof(uint16_t));
```

FIGURE 16 – Structure du code dans HLS

Dans chaque fonction, nous utilisons en entrée et en sortie des `stream`. C'est un type spécifique à Vivado HLS. Il permet d'échanger plus facilement les données entre chaque IP. Ainsi, à l'entrée de `DownSamp16` nous avons un stream de type `uint16_t`. Ensuite nous recopions toutes les données dans un tableau. Nous faisons les calculs nécessaires à un sous-échantillonnage de facteur 2 et nous mettons les résultats dans un tableau deux fois plus petit. Finalement, nous recopions ce tableau dans un stream de sortie.

Dans le but d'être plus efficace, nous avons rajouté un `pragma dataflow` dans le main. En effet, ce pragma nous permet de pipeliner les trois fonctions. De plus, dans la fonction `DownSamp16`, nous avons deux boucles for dans lesquelles nous avons ajouté un `pragma pipeline`. Ce pragma nous permet de pipeliner les calculs dans les boucles for et donc d'être plus rapide.

Pour finir, nous lançons la synthèse d'architecture et générons une IP compatible avec Vivado.

5.1.2 Implémentation sur carte : Vivado

Au départ, nous avons un fichier contenant un bloc (rcv) permettant de recevoir les données provenant de l'UART et un autre bloc (snd) permettant de renvoyer les données dans le lien UART.

Une fois que l'IP est générée sur Vivado HLS, nous l'importons sur Vivado. Nous relions correctement les blocs rcv, Downsampler et snd. Finalement, nous générons l'architecture et nous observons bien l'apparition des 3 blocs sur le schéma RTL 17.

Nous avons ensuite généré le bitstream associé à cette architecture et nous l'avons implanté sur la carte.

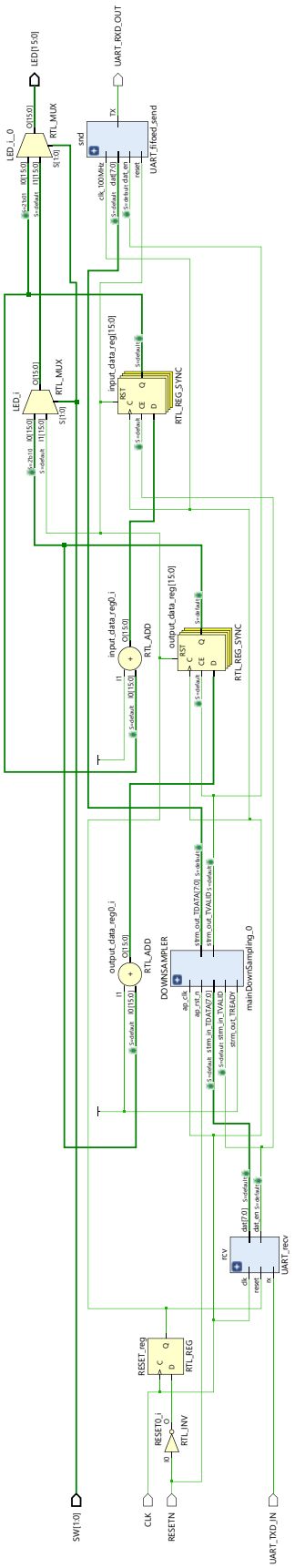


FIGURE 17 – Schéma RTL du début l'architecture

5.1.3 Tests et validation de l'implantation

Une fois notre module de sous-échantillonnage implanté sur le FPGA, nous avons dû le tester. Pour cela, nous avons repris un code permettant l'écriture et la lecture d'une liaison UART et avons mis 480 valeurs aléatoires en entrée. Chaque valeur devait être vérifiée à la main ce qui était assez fastidieux.

Nous avons donc choisi de changer nos valeurs en entrée en mettant une fois sur deux un 1 et une fois sur deux un 0. Il n'y avait que des 1 en sortie ce qui semblait montrer que le programme fonctionnait. Or, avec ces valeurs, seule une addition était testée. L'étape suivante, fut de mettre en entrée une séquence composée de tous les chiffres dans l'ordre, que nous doublons. Elle commence donc par " 0 0 1 1 2 2 3 3 ...". La sortie correspondait aux chiffres pairs, ce qui était assez facile à vérifier.

Cependant, afin d'avoir une vérification pouvant se faire en seulement quelques secondes, nous avons décidé d'utiliser des couleurs dans notre terminal. Pour cela, nous avons ajouté dans notre code de test, la fonction originale du sous échantillonnage en C++ afin de pouvoir comparer nos résultats. Si les deux résultats sont similaires, ils sont affichés en vert sinon ils le sont en rouge, comme nous pouvons le voir dans la figure 18.

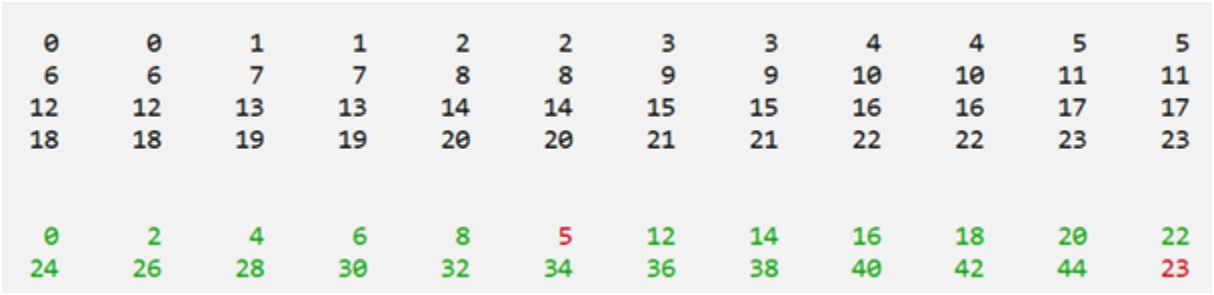


FIGURE 18 – Terminal avec les résultats en couleur

Ici, nous voyons très rapidement que sur nos 24 résultats, deux sont rouges. Cela vient du fait que nous avons injecté deux mauvaises valeurs afin de tester l'affichage rouge. Cela valide donc le fait que si tous nos résultats sont affichés en vert ils sont bons.

Grâce à ces différentes techniques de vérifications, nous avons pu valider notre module de sous-échantillonnage.

5.2 Ajout de modules

Maintenant que le premier module est implémenté et validé, la même démarche est effectuée sur les trois modules suivants de la chaîne de réception, c'est-à-dire :

- La démodulation PPM ;
- La suppression du préambule ;
- La mise en forme des bits.

Nous nous retrouvons donc avec la chaîne présentée en figure 19.

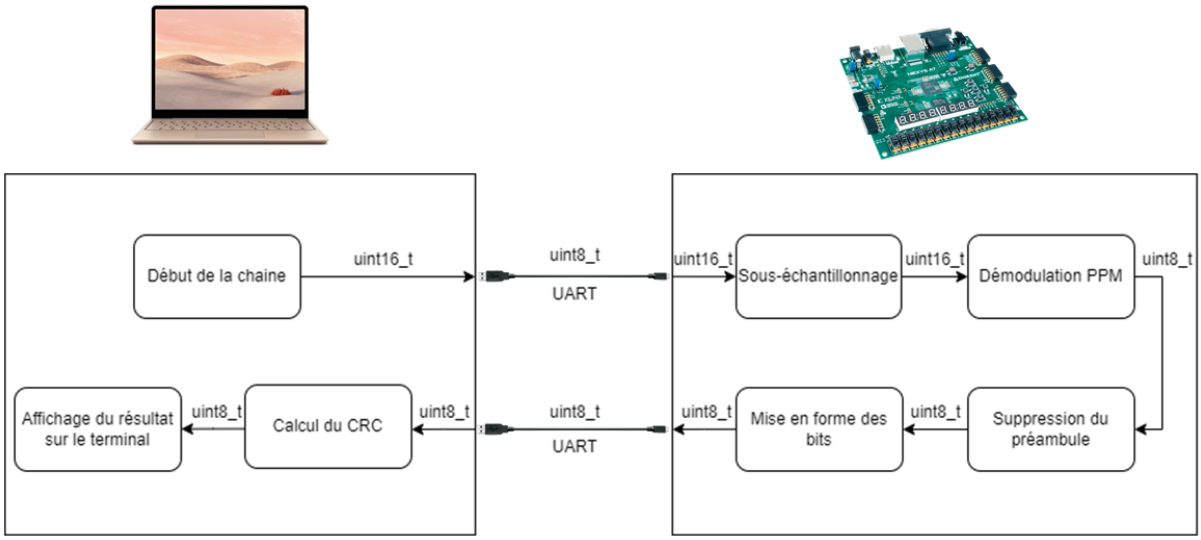


FIGURE 19 – Implémentation d'une partie de la chaîne sous FPGA

Il est bon de remarquer que seul le module de sous-échantillonnage a une entrée et une sortie en `uint16_t`. En effet, la démodulation PPM possède une sortie en `uint8_t` et on utilise cette taille de données jusqu'à l'UART de sortie. Ainsi, contrairement à ce que nous avons fait avec le module de sous-échantillonnage, la fonction `stream_bytes2words` sera la seule que nous aurons besoin d'utiliser.

Une fois les quatre modules implémentés sur notre carte, nous pouvons regarder nos résultats.

5.3 Résultats

En commençant à comparer les résultats, nous nous sommes rendu compte qu'avec notre code nous trouvions un nombre de trames détectées très différents de celui trouvé avec les optimisations logicielles. Après différentes recherches nous avons remarqué que cela venait des OS utilisés. En effet, la partie logicielle s'est déroulée en grande partie sur MAC OS alors que la partie matérielle s'est faite sur Windows. Or, nous avons remarqué que les résultats étaient faux sur Windows lorsque nous utilisions les options d'optimisation `-Ofast`, `-O3`, `-O2`... Nous avons donc fait tourner notre code sans ses optimisations. Il est donc plus lent.

Afin de comparer nos résultats avec ceux précédemment établis, nous dressons un tableau comparatif.

	AVX float	x86 int	AVX int	FPGA
Nombre de trames détectées	12556	12573	12575	12561
Temps/trame (ms)	0.73	1.21	0.52	880

TABLE 2 – Tableau comparatif des performances

Nous pouvons observer un nombre de trames détectées similaire au précédent ce qui nous certifie que nous avons des algorithmes fonctionnels.

Cependant, le temps pour détecter une trame est beaucoup plus élevé comparé aux autres. Cela vient de différentes causes.

Premièrement la liaison UART n'est pas très rapide, ici la vitesse n'est que de 1Mbit/s, l'utilisation d'une liaison plus rapide tel que l'Ethernet pourrait améliorer nos performances.

Le fait que seule une partie de la chaîne se situe sur FPGA dégrade également notre temps de traitement. En effet, des résultats intermédiaires se trouvant sur la carte sont utiles pour le calcul du CRC se faisant pour le moment sur ordinateur. Afin de ne pas gêner ces calculs, nous avons besoin de continuer à faire les calculs en C++ de façon à avoir accès aux résultats intermédiaires. On ne peut donc pas avoir un temps meilleur vu que nous ne remplaçons rien, nous ajoutons juste une autre façon de faire les calculs pour le moment.

De plus, le début de la chaîne étant sur ordinateur nous devons utiliser la liaison UART une première fois pour envoyer les données à la carte et une deuxième fois pour renvoyer les résultats calculés. Idéalement, il faudrait brancher l'antenne et la radio directement sur notre FPGA. Le traitement entier des informations sera réalisé dessus et les résultats seront envoyés à l'ordinateur pour faire l'affichage. La liaison ne serait utilisée qu'une seule fois.

6 Conclusion

Finalement, nous avons réussi à obtenir deux versions software qui sont fonctionnelles et plus efficaces que celles de départ. Nous avons aussi un début d'implémentation matérielle fonctionnelle et prometteuse. Ce qui serait souhaitable pour la suite est de réécrire le projet pour le jeu d'instruction NEON, qui est l'équivalent de l'AVX pour le processeur embarqué sur les Raspberry pi, d'implémenter la totalité de la chaîne sur FPGA en regardant l'impact temporel et matériel et de remplacer le lien UART.

Lors de ce projet, nous avons mis plus de temps que prévu pour avancer. En effet, la compréhension du projet global et des outils à utiliser était plus complexe que nous l'imaginions. De plus, la mise en place de tests pertinents et corrects n'était elle aussi pas évidente.

Les codes sources de notre projet sont accessible sur le dépôt GitHub [8] que nous avons créé et utilisé tout au long de ce projet. Ce dépôt est accessible librement.

Nous avons donc énormément appris sur ce projet aussi bien d'un point de vu compétences logicielles et matérielles que d'un point de vu gestion de projet, ce qui est très enrichissant.

Références

[1] Fly Radar 24. *FlyRadar24 Live Air Traffic*. [En ligne ; Page disponible le 27-Jan-2022]. 2022. URL : https://www.flightradar24.com.

[2] Bertrand Le Gal. *adsb-like-comm-toolbox*. [En ligne ; Page disponible le 27-Jan-2022]. 2020. URL : https://github.com/blegal/adsb-like-comm-toolbox.git.

[3] Xavier Fenard. *Un récepteur ADS-B simple*. [En ligne ; Page disponible le 27-Jan-2022]. 2010.

[4] Yoohwan Kim, Ju-Yeon Jo et Sungchul Lee. « ADS-B vulnerabilities and a security solution with a timestamp ». In : *IEEE Aerospace and Electronic Systems Magazine* (2017). [En ligne ; Page disponible le 27-Jan-2022].

[5] Wikipedia contributors. *X86 instruction listings — Wikipedia, The Free Encyclopedia*. [En ligne ; Page disponible le 27-Jan-2022]. 2022. URL : https://en.wikipedia.org/w/index.php?title=X86_instruction_listings&oldid=1068113341.

[6] CodinGame user : Machete. *SSE & AVX Vectorization*. [En ligne ; Page disponible le 27-Jan-2022]. 2022. URL : https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx.

[7] Intel. *Intel® Intrinsics Guide*. [En ligne ; Page disponible le 27-Jan-2022]. 2022. URL : https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[8] Bertrand Le Gal et al. *Projet ADSB*. [En ligne ; Page disponible le 27-Jan-2022]. 2022. URL : https://github.com/LCottin/Projet_ADSB.