HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Département TIC
Filière Informatique
Orientation Informatique embarquée

Travail de Bachelor

Algorithme de traitement par FPGA de signaux provenant de détecteurs de particules

Étudiante Marion Dutu Launay
Travail proposé par Jonathan Emery
CERN
1211 Genève 23
Suisse
Enseignant responsable Yann Thoma
Année académique 2019-2020

Yverdon-les-Bains, le 14 août 2020

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Marion Dutu Launay

Informations

Département TIC
Filière Informatique
Orientation Informatique embarquée
Étudiante Marion Dutu Launay
Enseignant responsable Yann Thoma

Travail de Bachelor 2019-2020

Algorithme de traitement par FPGA de signaux provenant de détecteurs de particules

---

Résumé publiable

Pour la grande mise à jour du LHC au CERN, une nouvelle génération de scanners de faisceaux de particules est en cours de développement. Le système d'acquisition de ces scanners doit comprendre un algorithme de traitement de signaux paramétrable pour des données disponibles en mémoires externes à une FPGA. Les signaux proviennent de quatre détecteurs de particules et sont tout d'abord digitalisés à une fréquence de l'ordre de 500 MHz avec une résolution de 14 bits, pour être stockés dans des mémoires externes à la FPGA. À la fin de l'acquisition, l'algorithme doit réaliser un traitement sur ces données par du filtrage afin de corriger certaines limitations analogiques ainsi que réduire l'information au minimum nécessaire. En effet, l'architecture du système ne permet pas de transférer toutes ces données en un temps raisonnable et c'est pourquoi un traitement par FPGA est indispensable.

L'objectif de ce travail de Bachelor était d'implémenter l'algorithme de filtrage dont il est question. Un des challenges principaux du développement fut d'effectuer le traitement des signaux sur plusieurs données en parallèle, car chaque canal que reçoit la FPGA contient quatre échantillons. Par conséquent, le système doit être en mesure d'appliquer le filtrage sur quatre canaux de quatre échantillons à chaque cycle, à une fréquence de 125 MHz, ce qui a eu des conséquences sur les choix architecturaux des modules.

Le projet comprend l'ensemble du processus d'implémentation, depuis la rédaction du code des descriptions synthétisables en VHDL et des bancs de tests en SystemVerilog, en passant par la simulation des modules avec des données réelles et aléatoires, jusqu'à la synthèse. Cette dernière a été effectuée avec une base Arria V, carte standardisée dans le groupe d'instrumentation du CERN.

Les différents modules implémentés comprennent une médiane, une moyenne glissante, des convolutions à 2, 3 et 5 termes et un filtre *leaky integrator*. Ces blocs étant destinés à être chaînés, une application composée de plusieurs modules a été mise en place afin de valider le fonctionnement de l'algorithme de traitement.

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Marion Dutu Launay

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie / Economie d'entreprise.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Nom du doyen
Chef du département TIC

Yverdon-les-Bains, le 14 août 2020

Authentification

La soussignée, Marion Dutu Launay, atteste par la présente avoir réalisé seule ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Yverdon-les-Bains, le 14 août 2020

Marion Dutu Launay

Remerciements

Je tiens à remercier les personnes suivantes qui m'ont aidée et soutenue lors de ce travail :

Monsieur Yann Thoma, professeur responsable de ce travail de diplôme, pour son suivi régulier, ses précieux conseils et sa disponibilité.

Monsieur Jonathan Emery, mandant du travail de diplôme, pour ses renseignements, ses conseils et sa disponibilité.

Monsieur Rick Wertenbroek, ingénieur Ra&D au REDS, pour avoir consacré une entière matinée à m'aider avec le Timing Analyzer de Quartus, et pour ses nombreuses explications.

Monsieur Isaïa Spinelli, mon compagnon, pour son soutien et ses conseils.

Finalement, je souhaite remercier ma famille et mes proches pour leur soutien constant.

Cahier des charges

Contexte

Le but de ce travail de bachelor est de réaliser un algorithme de traitement de signaux paramétrables pour un flux de données provenant de quatre détecteurs de particules. Après avoir été digitalisés à une fréquence de l'ordre de 500 MHz avec une résolution de 14 bits, ils doivent subir un traitement par FPGA avant d'être stockés dans des mémoires externes de type DDR3. Ce traitement a pour objectif d'effectuer une correction de certaines limitations analogiques ainsi qu'une réduction de l'information au minimum nécessaire, car l'architecture du système ne permet pas de transférer toutes les données en mémoire en un temps raisonnable.

Besoins et contraintes

Les données doivent passer par un système de filtrage qui comprend trois applications : une moyenne, une médiane et diverses opérations de convolution, le tout sur un système tournant à une fréquence de 125 MHz. Pour l'interface d'entrée, les signaux sont représentés sous la forme de quatre canaux de 4x16 bits qui doivent être traités en parallèle. Après le traitement, en sortie, ils doivent conserver cette forme. Le développement du module de filtrage sera implémenté avec le langage de description VHDL.

Le fonctionnement de l'entité VHDL résultante devra être validé à l'aide d'une simulation comprenant des bancs de tests avec différentes valeurs fictives ou réelles disponibles dans un fichier annexe, à l'aide des outils de la suite Questa. De plus, une synthèse de la description sera effectuée avec Quartus Prime. Une intégration du système sur la plateforme utilisée au CERN est également prévue si le temps le permet.

La plateforme hardware utilisée est une carte standardisée dans le groupe d'instrumentation du CERN, la VFC-HD, basée sur une Arria V et une carte FMC de digitalisation commerciale. L'ensemble est destiné à être utilisé dans un fond de panier type VME avec CPU Linux ou en stand-alone avec une connexion Ethernet à un serveur distant.

Résultats attendus

Au terme du travail, une description du module VHDL synthétisable devra être disponible. Ce dernier comprendra l'algorithme de traitement des signaux tout en respectant l'interface donnée, et devra être fonctionnel avec des données réelles.

Table des matières

Glossaire

FMC FPGA Mezzanine Card. Norme ANSI définissant les modules mezzanine I/O avec connexion à une FPGA ou un autre appareil avec une capacité I/O configurable. 12

photomultiplicateur Tube électronique permettant de détecter des photons et les compter individuellement. Abréviation : PM. 10

ultravide Niveau de vide très poussé. 9

émittance Grandeur physique caractérisant le flux lumineux émis par une surface lumineuse. 10

1 Introduction

Le Conseil Européen pour la Recherche Nucléaire, plus communément connu sous son acronyme CERN, a vu le jour en 1954 à Genève, en Suisse. Sa mission principale consiste en la compréhension du fonctionnement et de la composition de l'Univers du point de vue de la physique fondamentale, et offre un épatant exemple de collaboration internationale. Depuis sa création, de nombreuses avancées scientifiques de grande ampleur ont été réalisées en physique des particules et en ingénierie, avec l'aide de chercheurs provenant du monde entier. Une des illustres découvertes du CERN est l'existence du boson de Higgs, une particule qui joue un rôle majeur dans l'histoire de l'Univers. Cette découverte a été permise grâce à la principale réalisation du CERN, le LHC.

Une des principales installations actuelles du CERN est son complexe d'accélérateurs. Le fameux LHC, Grand Collisionneur de Hadrons, est est l'accélérateur de particules le grand et le plus puissant du monde jamais construit. Il s'agit d'un tunnel de 27 kilomètres de circonférence et enfoui à 100 mètres sous terre (figure 1), formé de milliers d'aimants supra-conducteurs et doté de structures accélératrices visant à accroître l'énergie des particules circulant à l'intérieur. Ces particules sont présentes sous la forme de deux faisceaux qui se déplacent à une vitesse proche de celle de la lumière dans des tubes sous ultravide et circulant en sens opposé afin d'entrer en collision. Afin de renforcer la probabilité d'un tel phénomène, l'intervention de divers systèmes composés d'aimants est impérative afin de créer un puissant champ magnétique, les particules étant minuscules : en effet, faire entrer en collision deux faisceaux de particules reviendrait à propulser deux aiguilles éloignées de 10 kilomètres l'une vers l'autre. Les collisions sont prévues en quatre points d'interaction de l'accélérateur, où les détecteurs de particules ATLAS, CMS, LHCb et ALICE sont situés.

FIGURE 1 – Tunnel du LHC (© Brice Maximilien, CERN)

Une mise à niveau du LHC a commencé en juin 2018 et vise à augmenter ses performances à leur maximum dans le but d'accroître le potentiel de découvertes pour fin 2027 et ainsi déchiffrer de nouvelles énigmes de la physique. Pour ce faire, de nombreuses innovations techniques sont effectuées pour amplifier la luminosité d'un facteur 10, c'est pourquoi le projet de mise à niveau porte le nom de LHC à Haute Luminosité.

La luminosité est un indicateur clé des performances d'un accélérateur. Plus elle est grande, plus des collisions vont se produire en un certain temps, et ainsi plus de données seront récoltées, ce qui

mènera à de potentielles manifestations de nouveaux phénomènes très rares qui pourront être observés. Les physiciens ont besoin d'une grande quantité de mesures pour déceler ces derniers. Par exemple, le LHC-HL pourra produire jusqu'à 5 fois plus de bosons de Higgs.

Pour opérer sur ce nouveau scénario, une instrumentation plus précise est en cours de développement depuis plusieurs années. Elle comprend notamment la conception d'une nouvelle génération de scanners de faisceaux utilisés pour calculer l'émittance d'un faisceau, les LIU-BWS pour *LHC Injectors Upgrade Beam Wire Scanners*. Le principe de ce type de scanners consiste à faire passer un très mince fil de carbone au travers du faisceau de particules en circulation. Cette interaction génère une multitude de particules secondaires transformées en courant électrique à travers un photomultiplicateur. Avec cette méthode, il est possible d'effectuer une analyse du profil du faisceau *bunch by bunch* ("grappe par grappe") sur des machines cycliques, où la résolution des mesures est déterminée par la fréquence de révolution de l'accélérateur et la vitesse du scanner.

Le système d'acquisition des mesures récoltées effectue une digitalisation des signaux provenant de quatre détecteurs de particules, en vue de les stocker dans une mémoire de type DDR3. Un filtrage de ces données est impératif, car l'architecture du système n'est pas capable de toutes les transférer en un temps adéquat. Ce traitement doit être intégré sur une FPGA du groupe d'instrumentation du CERN, la VFC-HD, qui fait partie du système d'acquisition.

Ce travail de Bachelor a pour objectif d'implémenter cet algorithme de traitement à l'aide du langage de description *hardware* VHDL. Il comprendra diverses applications de filtre, notamment une moyenne glissante, une médiane ainsi que des convolutions avec différentes matrices. L'implémentation du traitement sera contenue dans un module VHDL, isolé du reste de la description du système contenu par la VFC-HD déjà mis en place par le groupe d'instrumentation du CERN. En effet, une grande partie du système d'acquisition des BWS est déjà fonctionnelle. Il s'agira donc d'intégrer le module de filtrage aux ressources déjà existantes, et donc de porter une attention particulière aux spécifications des entités qui interviennent autour de celui-ci.

Un des challenges principaux du développement sera d'effectuer le traitement des signaux sur plusieurs données en parallèle ; en effet, chaque canal que reçoit la FPGA contient quatre échantillons. Par conséquent, le système doit être en mesure d'appliquer le filtrage sur quatre canaux de quatre échantillons à chaque cycle, à une fréquence de fonctionnement de 125 MHz.

Les premières semaines de travail seront attribuées à la lecture des ressources du CERN en lien avec le sujet, puis à une brève analyse du système existant, avant de procéder aux spécifications plus précises du projet. Dans un premier temps, il s'agira d'établir les spécifications fonctionnelles du système en suivant les contraintes figurant dans le cahier des charges. En d'autres termes, nous définirons clairement ses différentes composantes en respectant l'interface d'entrée et de sortie prévue, pour que l'intégration du module au système global se fasse sans accroc. Ensuite, le processus de réalisation de chaque filtre sera le même.

La première étape du travail consistera en la réalisation d'un plan de vérification pour les sous-systèmes de filtrage élaboré à partir des spécifications du système ; par sous-système, nous entendons la description d'un des filtres. Viendra ensuite l'écriture du code nécessaire à la mise en place du banc de test. Cela comprendra la rédaction des scénarios figurant dans le plan de vérification, puis la rédaction du code de génération des modèles de référence, qui simulera l'application des différents filtres à implémenter en description non-synthétisable.

Les spécifications d'implémentation de la description synthétisable seront effectuées en parallèle avec l'écriture des bancs de tests associés à chaque sous-partie du module de filtrage. La rédaction du code synthétisable suivra, principalement effectuée pendant la partie du travail à plein temps. Les différents algorithmes de filtrage seront donc simulés régulièrement afin d'avancer pas à pas dans les opérations à élaborer.

Une synthèse des descriptions VHDL sera régulièrement effectuée, en vue de procéder à la prochaine

étape d'intégration à la FPGA du groupe d'instrumentation du CERN, la VFC-HD. Cette dernière partie n'a pas été achevée dans le cadre de ce travail, car le temps et les circonstances ne l'ont pas rendue possible. Néanmoins, un essai d'intégration du module de filtrage sur une FPGA de type Cyclone V en prévision de la soutenance est planifié. Cela permettra de valider le design sur une plateforme de remplacement.

La structure du document se présente comme suit : la première partie consiste en une brève analyse afin d'étudier le fonctionnement du système d'acquisition en surface, puis d'expliquer le rôle du filtrage et des différents filtres. Viennent ensuite les spécifications du système et les réflexions sur la conception du module, suivies d'explications détaillées sur l'implémentation de chacun des modules de filtrage. Ensuite, le processus de vérification sera introduit et détaillé pour chaque filtre, puis nous présenterons les résultats des simulations effectuées avec les bancs de tests. Enfin, la dernière partie décrit les synthèses des différents modules en commentant les vues RTL résultantes et la quantité de logique utilisée.

2 Analyse

2.1 Architecture globale des scanners LIU BWS

La figure 2 schématise l'architecture globale des scanners LIU BWS. La caisse d'acquisition et de supervision se trouve sur la partie gauche, où sont dirigés les signaux analogiques via des câbles coaxiaux. Nous pouvons y voir la VFC-HD, responsable de la réception des données et de leur stockage, qui supportera le futur module de filtrage.

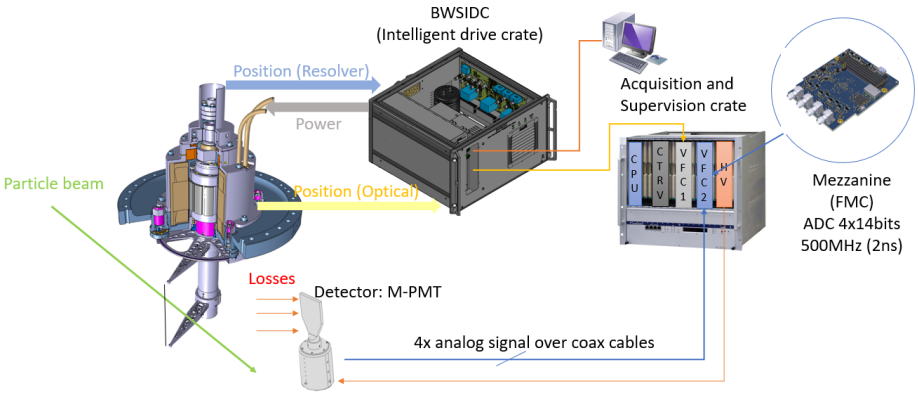


FIGURE 2 – Composants principaux des scanners LIU BWS (J. Emery, février 2020)

2.2 Système d'acquisition

La figure 3 est un schéma simplifié du système d'acquisition des scanners LIU BWS, qui comporte également l'avancement du projet à l'heure du développement.

Ce travail de Bachelor a pour but de réaliser le bloc de filtrage représenté en rouge. Certaines parties du système sont en cours de développement au sein du groupe d'instrumentation du CERN, tandis que d'autres sont déjà en place et fonctionnelles.

Les données des détecteurs de particules proviennent d'une FMC où des mesures sont effectuées toutes les deux nanosecondes. Ces données sont transmises à travers une interface JESD-204, dont le rôle est de standardiser et réduire le nombre d'entrées et sorties entre la FMC et la FPGA pour que les données soient correctement transmises, et ce malgré la différence de fréquence des deux entités.

Comme nous pouvons le voir sur la figure 3, la FPGA reçoit quatre canaux en parallèle par cycle d'horloge, qui est réglée à une fréquence de 125 MHz. Ces canaux contiennent chacun quatre échantillons de données provenant des détecteurs de particules. Il sera donc nécessaire d'effectuer le filtrage sur ces quatre canaux de quatre échantillons simultanément, ce qui ajoute des contraintes sur la façon de concevoir le design.

2.3 Utilité du filtrage

L'application de différents filtres aux données a plusieurs utilités. En premier lieu, cela va permettre d'obtenir des statistiques sur l'ensemble des données récoltées par le système d'acquisition. En second lieu, une réduction des informations à stocker dans la mémoire sera facilement réalisable avec un traitement ultérieur ; les informations seront triées, conservant ainsi seulement les plus pertinentes.

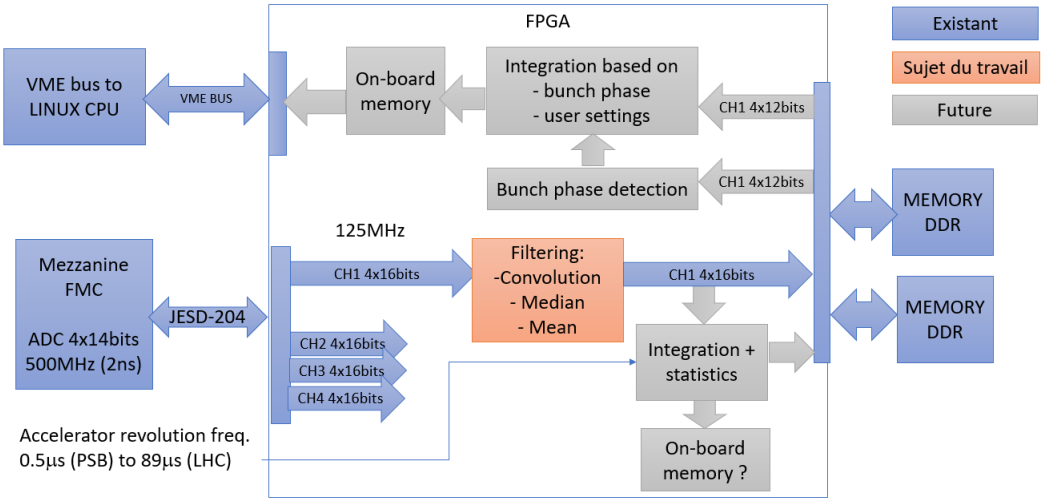


FIGURE 3 – Schéma simplifié du système d'acquisition LIU BWS (J. Emery, février 2020)

Les sections suivantes présentent les différents filtres que nous allons implémenter tout au long du travail et leur rôle respectif.

2.3.1 Leaky Integrator

Couramment, l'on souhaite appliquer une sorte de lissage à notre ensemble de données, ou encore une fonction passe-bas. Une approche commune est celle de la moyenne glissante, mais cette dernière nécessite du *buffering* de données ; un effet semblable peut s'obtenir avec l'utilisation du *leaky integrator*, qui fournit une estimation de la moyenne glissante.

Dans notre domaine d'application, l'usage d'un filtre de type *leaky integrator* sur le flux de données reviendrait en effet à effectuer une sorte de moyenne glissante, mais avec un avantage non négligeable ; rappelons que pour effectuer une moyenne glissante sur un ensemble de données, il est nécessaire de mémoriser un certain nombre d'échantillons et d'effectuer autant d'opérations que nous avons d'échantillons.

Considérons la formule de la moyenne glissante pour un échantillon :

$$y[n] = \frac{1}{M} \sum_{k=0}^{M-1} x[n-k] \tag{1}$$

M est le nombre d'échantillons de la fenêtre d'intégration et n est le numéro de l'échantillon actuel. Avec l'utilisation de ce type de filtre, l'effet "lissant" de la courbe et le nombre d'opérations à effectuer et le nombre d'opérations à stocker est proportionnel à M.

Voici la formule du *leaky integrator* :

$$y[n] = \lambda y[n-1] + (1-\lambda)x[n] \tag{2}$$

Avec

$$\lambda = \frac{M-1}{M}$$

Le nom de ce filtre prend tout son sens : en effet, seulement une fraction λ des valeurs accumulées jusqu'à présent est gardée, et une fraction $1 - \lambda$ n'est pas prise en compte. Puis s'y ajoute uniquement une fraction $1 - \lambda$ de l'estimation courante. Le nom du filtre provient donc du fait qu'à chaque itération, le résultat de l'estimation précédente n'est pas totalement utilisé, une partie a "fui"

Cette estimation ne requiert que deux multiplications et une addition, et produit un résultat à peu près similaire à une estimation par méthode de moyenne glissante. Il s'agit d'une fonction récursive, car elle utilise le résultat précédent de l'opération pour estimer la suite. Plus notre M sera grand, plus le facteur λ s'approchera de 1 et donc stabilisera l'estimation.

Pour notre projet, le filtre *leaky integrator* peut avoir diverses utilisations tout en limitant l'utilisation des ressources matérielles. Le facteur λ, nommé également taux d'oubli, pourra être ajusté selon la fréquence des *bunch* ou à la fréquence de révolution des accélérateurs.

2.3.2 Convolution

Une opération de convolution s'approche mathématiquement de la notion de filtre linéaire. La formule de convolution utilise deux signaux comme opérandes et les combine afin d'en former un troisième. Algébriquement, cela revient à effectuer la multiplication de deux polynômes dont les coefficients sont les éléments composant les deux signaux.

C'est une opération très utilisée dans le monde du traitement du signal, car elle permet d'appliquer de nombreux effets au flux de données d'entrée selon la taille et la configuration du noyau de convolution, dénommé kernel, qui représente la réponse d'impulsion.

Considérons deux signaux représentés par des vecteurs x de longueur m et y de longueur n. Le produit de convolution des deux signaux se note généralement :

$$z(k) = x(k) * y(k) \tag{3}$$

z est le vecteur de longueur $m + n - 1$, où le k^{ime} élément est calculé comme suit :

$$z(k) = \sum_j x(j)y(k - j + 1) \tag{4}$$

En imaginant x comme étant notre flux de données et y le noyau de convolution, de nombreux filtres sont représentables grâce à cette méthode en adaptant les valeurs et le nombre des coefficients du kernel. Certains d'entre eux sont présentés ci-dessous en guise d'exemple et ont été utilisés pour effectuer la vérification des modules utilisant la formule de convolution.

Filtre passe-bas inverse

Un filtre passe-bas permet d'atténuer le signal selon une fréquence de coupure choisie. Nous utiliserons le kernel `[4.5, -3.5]` qui nous permettra de retrouver la dynamique du signal original.

Filtre *high boost*

L'application d'un filtre de type *high boost* sur un flux de données a pour effet d'amplifier ses composants haute-fréquence sans pour autant en éliminer les basses-fréquences.

Pour le représenter, nous allons utiliser le kernel `[-1, 2, -1]`.

Savitzky-Golay

En traitement du signal, l'algorithme de Savitzky-Golay permet de lisser un signal et en prélever les dérivées successives. Dans notre cas, cela nous sert à obtenir une mesure de la pente du signal.

Nous utiliserons le kernel `[-2, -1, 0, 1, 2]` qui représente une fenêtre glissante de 5 points.

Moyennes

Il est trivial de modéliser une moyenne glissante grâce à la méthode de convolution : pour ce faire, il nous faut remplir le kernel de longueur N avec la même valeur, à savoir $\frac{1}{N}$ où N est le nombre d'échantillons pris en compte pour le calcul de la moyenne.

Pour une moyenne glissante de deux échantillons par exemple, le noyau de convolution utilisé sera `[0.5, 0.5]`. Cela aura pour effet un simple lissage du flux entrant.

2.3.3 Médiane

Le filtre médian permet de ne conserver que les valeurs médianes du flux de données ; il est souvent associé à une moyenne, mais qui ne prendrait pas compte des valeurs atypiques. Dans le cas où l'on souhaiterait obtenir la valeur moyenne d'un ensemble d'échantillons, la médiane est souvent plus représentative, car elle n'est pas influencée par des valeurs extrêmes qui pourraient être erronées.

Pour obtenir la valeur médiane d'un ensemble de données, il suffit de suivre deux étapes :

1. Trier les données par ordre croissant.
2. Déterminer l'indice de la valeur médiane : si N est le nombre d'échantillons, il s'agit de l'arrondi supérieur ou inférieur de $\frac{N}{2}$.

2.3.4 Moyenne glissante

Comme nous l'avons vu précédemment, une moyenne glissante permet d'obtenir un effet de lissage des valeurs du flux de données ; la formule d'application requiert la mémorisation d'un certain nombre d'échantillons selon la largeur de la fenêtre d'échantillonnage. Néanmoins, les résultats seront plus précis que ceux fournis par le *leaky integrator*.

Dans notre cas, nous souhaitons obtenir une moyenne glissante sur 25 échantillons. Nous avons constaté que ce type de traitement est facilement réalisable en utilisant la méthode de convolution : ici, le kernel à utiliser comprendra 25 coefficients qui auront pour valeur $\frac{1}{25}$, soit 0.04.

3 Spécifications

3.1 Spécifications fonctionnelles

Globalement, le module de filtrage peut être vu comme une boîte à outils fournissant différents blocs qui appliquent un filtre spécifique aux données que l'on y insère. Cette méthode de conception permettra aux futurs utilisateurs du module de chaîner les blocs à leur guise et de pouvoir changer l'ordre d'application des filtres selon l'objectif à atteindre.

Comme le module a pour but d'être utilisé dans le cadre du projet principal du système d'acquisition des scanners, il est donc nécessaire d'établir avec soin ses spécifications fonctionnelles, puis ses spécifications d'implémentation. Au terme du travail, le design livré devrait être capable de traiter correctement le flux de données circulant réellement dans le système.

3.1.1 Interface

Afin que le module de filtrage soit en mesure d'être intégré au design existant, il est primordial de respecter certaines contraintes vis-à-vis de l'interface utilisé.

Comme nous l'avons vu dans la partie analyse, la FPGA interagit avec la Mezzanine FMC via l'interface JESD-204 pour recevoir les données du convertisseur analogique/digital, plus communément appelé ADC. Les sources de données de ce dernier sont représentées sous la forme de quatre canaux de quatre échantillons de 16 bits que le module recevra en parallèle. Chaque canal doit être traité séparément : ainsi, il faudra multiplier le nombre de modules de filtrage par quatre.

Voici le schéma bloc d'un des modules de filtrage :

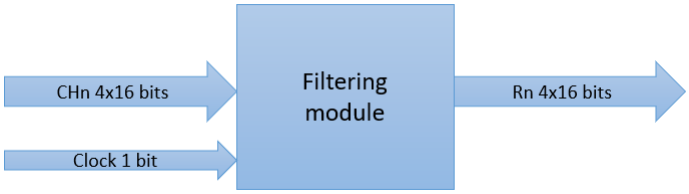


FIGURE 4 – Schéma global d'un module de filtrage

Le flux de données entrant correspond aux données du canal traité par le module. Il s'agit de quatre échantillons de 16 bits contenus dans un seul vecteur. Dans chaque échantillon, les deux bits de poids faibles (ci-après LSB) sont toujours à 0. Les données en sortie seront disponibles en tout temps et auront la même forme que le vecteur d'entrée, à savoir 4 échantillons de 16 bits avec les deux LSB à 0. Rappelons que la durée prise entre chaque échantillon est de 2 nanosecondes.

Le module doit fournir plusieurs sous-modules dont la place peut être interchangée à l'intérieur du design. Ainsi, dans la même optique que pour le module de filtrage dans sa généralité, il est impératif de garder le même nombre d'entrées et sorties liées aux données entrantes et aux résultats pour chaque bloc, avec le même format de données. Dans notre cas, il s'agit des 4 échantillons de 16 bits que contient un canal.

3.1.2 Horloge

Une horloge commune sera connectée à chaque sous-module de filtrage afin d'assurer la synchronisation des données entre les sous-systèmes.

3.1.3 Filtres

Leaky Integrator

Le sous-module appliquant le filtre *leaky integrator* au flux de données doit pouvoir fournir en sortie l'estimation courante de la moyenne glissante pour chacun des 4 échantillons, avec des valeurs différentes de M.

Rappelons que la formule du *leaky integrator* requiert l'utilisation de la valeur de l'estimation précédente, qui est indéfinie au premier coup d'horloge. Pour le tout premier échantillon traité, cette valeur sera donc de 0.

Convolution

Chaque sous-module utilisant de la convolution doit pouvoir fournir en sortie le résultat de l'application du filtre représenté par le kernel de convolution, pour chacun des échantillons et pour différentes valeurs de coefficients.

Nous allons implémenter différents modules de convolution selon le nombre de termes que contient le kernel ; dans notre cas, il s'agira de convolutions à 2, 3 et 5 termes pour divers filtres. Il doit être possible et rapide de changer la valeur des coefficients, qui seront des nombres décimaux positifs ou négatifs.

Médiane

Le sous-module appliquant le filtre médian au flux de données doit pouvoir fournir en sortie la valeur de la médiane sur 5 échantillons pour les quatre données d'entrée.

Moyenne glissante

Le sous-module appliquant une moyenne glissante au flux de données doit être en mesure de fournir la valeur de la moyenne mobile sur 25 échantillons pour les quatre données d'entrée.

3.2 Spécifications d'implémentation

Représentation des nombres

Une problématique importante des designs FPGA est la façon de stocker et représenter les nombres dans les descriptions synthétisables. Dans certains filtres qui vont être intégrés au système de filtrage, il sera nécessaire de manipuler des données représentant des nombres décimaux. Pour ce faire, deux possibilités se présentent : utiliser une représentation en virgule flottante ou en virgule fixe.

L'interprétation des nombres ne sera pas la même dans tous les sous-systèmes. Pour notre premier filtre, le *leaky integrator*, les nombres seront représentés en format virgule fixe. Bien évidemment, le format utilisé sera pris en compte lors de la vérification du design afin d'adapter le calcul de la référence et/ou d'appliquer une marge d'erreur adéquate.

La représentation avec virgule flottante offre une meilleure précision et une plus grande plage de valeurs représentables, grâce à leur système de représentation avec mantisse et exposant. Cependant, elles nécessitent beaucoup de logique pour effectuer des calculs arithmétiques.

À contrario, la représentation en virgule fixe engendre une utilisation de logique plus réduite, au détriment d'une précision de valeurs plus faible ; quand la place de la virgule est fixée, cela réduit la plage de valeurs représentables sur le nombre de bits utilisés pour la partie entière ou décimale du nombre à coder.

Dans notre cas, l'utilisation de nombres avec virgule fixe est préférable, car nous voulons économiser les modules de logique ; avec nos 14 bits pour représenter les données contenues dans les échantillons et

les résultats du filtrage, nous pouvons avoir une plage de valeurs suffisante en adaptant correctement la place de la virgule. De plus, une précision *exacte* n'est pas requise pour notre type de traitement.

Afin de paramétrer cette représentation en virgule fixe, nous avons là encore deux choix : nous pouvons régler le nombre de bits de fraction de façon statique, ou le rendre dynamique. Dans la première option, le nombre de bits utilisés pour coder la partie décimale peut être un paramètre générique ; les calculs tiendront compte de la place de la virgule choisie pour représenter les données des échantillons, ceci afin d'effectuer des arrondis et de recadrer le résultat des opérations selon la valeur de celui-ci. Cependant, une fois cette position fixée à la synthèse, elle ne sera plus modifiable. Dans la seconde option, les calculs ne considéreront pas la place de la virgule des entrées, et le résultat final sera recardé en tronquant les bits en surplus. Cela laissera le choix à l'utilisateur du module quant à l'interprétation des données.

Le filtre *leaky integrator* a été réalisé en suivant les deux méthodes, ceci afin d'effectuer une comparaison entre les variantes. L'implémentation est détaillée tout au long du chapitre suivant, et le chapitre concernant la simulation du système fait part des différences observées. Dans ce premier filtre, les données sont toujours des nombres non-signés, comme les valeurs des échantillons d'entrée ; la formule appliquée à ces derniers ne produira jamais de résultats négatifs, ce qui est également le cas de la moyenne mobile qui suit les mêmes spécifications quant à la représentation des nombres.

Il en va de même pour les différents filtres qui utilisent de la convolution, à la différence que les valeurs en interne sont représentées en nombre signés, compte tenu des valeurs négatives que peuvent avoir les coefficients des kernels de convolution utilisés.

Le filtre médian, quant à lui, exploite des entiers non-signés ; comme les seules opérations effectuées en interne sont des comparaisons, les données en sortie conservent leur format d'entrée.

Fréquence

La fréquence du système existant implanté sur la FPGA est de 125 MHz. Par conséquent, la fréquence de fonctionnement maximale de notre système, post placement-routage, devra être supérieure ou égale à 125 MHz.

Nos modules de filtrage doivent exploiter des opérations arithmétiques et/ou des comparaisons en interne, et tous sont tenus de traiter quatre échantillons par cycle ; la fréquence requise ne sera pas toujours atteinte si l'on implémente les traitements à effectuer avec des systèmes purement combinatoires, car les délais combinatoires peuvent être trop longs pour fournir quatre résultats par coup d'horloge. Il sera nécessaire d'adapter les modules en suivant le principe du pipelining afin de respecter les contraintes d'implémentation.

Un système pipeliné découpe un système combinatoire en différents niveaux pour effectuer le traitement séquentiellement. Chaque niveau de pipeline comprend un ensemble de registres dont le rôle est de sauvegarder les opérandes et les résultats des blocs combinatoires et assurer la cohérence des données. Une fois le processus enclenché, il faudra attendre un certain nombre de cycles avant d'avoir un résultat correct, qui correspond au nombre d'étages du pipeline. Une fois ce délai passé, le système sera en mesure de fournir des informations correctes à chaque cycle.

Afin de concevoir chaque bloc en niveaux de pipeline, nous avons séparé le traitement appliqué aux échantillons d'entrée en plusieurs phases, où chaque petite étape de traitement peut s'appliquer sur les quatre échantillons en un même étage de pipeline. Le découpage de nos systèmes combinatoires en systèmes séquentiels sont développés dans la partie traitant de l'implémentation. Cette méthode a prouvé son efficacité, car la fréquence maximale de fonctionnement des nouvelles versions des modules s'est souvent avérée bien au-dessus des 125 MHz requis. Cependant, si cela n'avait pas été le cas, nous aurions pu subdiviser le traitement une fois de plus, par exemple en séparant la valeur d'un échantillon en deux (bits de poids forts d'un côté, bits de poids forts de l'autre) et les utiliser pour deux calculs distincts, puis combiner les résultats intermédiaires par la suite.

4 Conception et implémentation

4.1 Généralités

4.1.1 Structure du projet

Le fichier de description VHDL du module de filtrage entier comprendra tous les sous-systèmes qui contiennent les filtres internes sous forme de composants. En vue de modifier le traitement du flux de données, l'ordre des blocs pourra donc être changé dans ce fichier de référence, en reliant les entrées et sorties des modules internes selon le résultat recherché.

Chaque sous-système aura un fichier source associé contenant sa propre description synthétisable. Ces fichiers se trouvent dans le dossier `src_vhdl`.

4.1.2 Librairies externes

Représentation des nombres en virgule fixe

La librairie `fixed_pkg`, développée par David Bishop, est une librairie synthétisable qui nous permet d'effectuer des opérations en virgule fixe de manière rapide avec une syntaxe intuitive. Elle offre la possibilité de manipuler des signaux de type `sfixed` et `ufixed`, pour les nombres à virgule fixe en signé et non-signé respectivement.

L'indice `high` d'un signal de l'un de ces deux types représente le nombre de bits utilisés pour coder la partie entière et l'indice `low` représente le nombre de bits pour la partie fractionnaire, en nombre négatif. Il faut donc que la précision pour représenter les données soit décidée à la compilation, car les valeurs des indices doivent être constantes tout comme pour la création d'un signal de type `std_logic_vector`.

La librairie `fixed_pkg` surcharge la plupart des opérateurs mathématiques pour les utiliser avec des opérandes de type `sfixed` et `ufixed`, en retournant un signal dont la taille s'adapte en adéquation avec la précision du résultat. Il est aussi possible d'utiliser des constantes numériques pour les opérandes, en les convertissant au préalable à l'aide des fonctions `to_ufixed` et `to_sfixed`.

4.2 Leaky Integrator

Le premier filtre développé dans ce travail est le *leaky integrator*. Il était pratique de commencer le projet par un module avec une complexité d'implémentation peu élevée, afin de se familiariser avec les nouveaux concepts découverts pendant le travail. La figure 5 représente un schéma décrivant la structure interne de ce module de filtrage, notamment comment sont gérés les quatre échantillons que nous avons à chaque cycle.

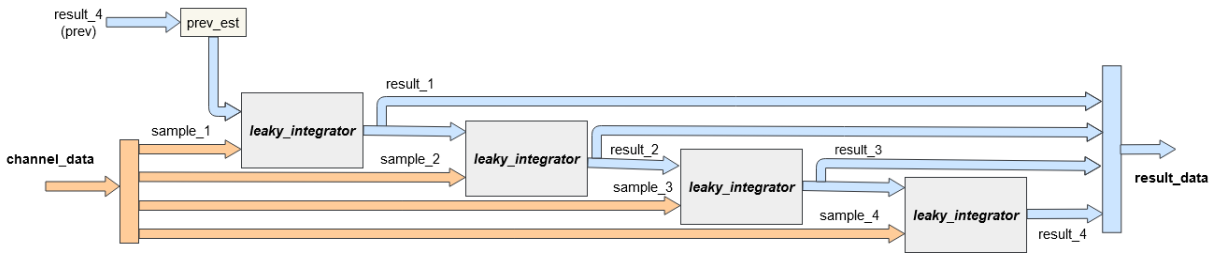


FIGURE 5 – Schéma général du module *leaky integator* (version combinatoire)

Les échantillons sont récupérés depuis le vecteur du canal d'entrée, sans les deux bits de poids faible, avant d'être utilisés en opérandes dans la formule du *leaky integrator*, qui nécessite deux données : la valeur de l'échantillon actuel et la valeur de l'estimation précédente, donc du résultat précédemment obtenu. Dans la même optique, deux bits à 0 sont apposés à la fin des quatre résultats, qui sont regroupés en une seule sortie.

Au démarrage du système, la valeur de l'estimation précédente est évidemment indéfinie. Le signal interne `pres_est`, qui représente cette valeur prend donc la valeur de 0. À partir du coup d'horloge suivant, il prend la valeur du dernier résultat obtenu, soit celui du quatrième échantillon d'entrée.

Pour développer ce premier filtre expérimental, deux implémentations différentes ont été réalisées.

La première version de la description utilise la librairie `fixed_pkg` présentée plus haut. Elle comprend un paramètre générique `FRACT_BITS` qui définit le nombre de bits utilisés pour représenter la partie fractionnaire des données en entrée. Elle intègre un second paramètre générique `M` qui correspond au facteur du même nom utilisé dans la formule du *leaky integrator*.

La seconde version n'utilise aucune librairie externe et ne prend aucun paramètre générique ; ainsi, contrairement à la première version, la façon de représenter les données n'est pas fixée à la compilation, tout comme le facteur λ. Ce dernier est directement donné en entrée du module avec un format Q1.15 : 1 bit pour la partie entière et 15 bits pour la fraction, afin d'avoir la meilleure précision possible. Il revient donc à l'utilisateur du module de choisir comment les données des échantillons et des résultats doivent être interprétées. Naturellement, la précision utilisée pour un échantillon et le résultat obtenu pour le traitement de ce dernier doit être la même.

4.2.1 Première version : précision fixe

La description synthétisable de l'application du *leaky integrator* aux données d'entrée est implémentée à l'aide de la librairie `fixed_pkg`, qui fournit les opérations en virgule fixe dont nous avons besoin : multiplication, addition et soustraction. Au préalable, les valeurs des 4 échantillons d'entrée sont convertis en données de type `ufixed`.

La valeur du facteur λ est calculée selon le paramètre générique `M` comme dans notre formule de référence. Le facteur `M` est converti en `ufixed` afin d'effectuer la division. Comme λ se situera toujours entre 0 et 1, elle ne nécessite qu'un bit pour la partie entière et 8 pour la partie fractionnaire selon le format du résultat de la division. En outre, la librairie ne permet pas de représenter un nombre composé uniquement de bits pour la partie fractionnaire, il est nécessaire d'avoir au moins un bit pour la partie entière. Le calcul effectué pour l'obtenir se situe au début de la description synthétisable, mais comme cette valeur est constante, cela n'utilisera pas de ressources à l'intérieur de la FPGA et sera traduit comme une constante par le synthétiseur.

Comme les résultats doivent conserver la même forme que les données d'entrée, à savoir une représentation sous 16 bits avec les deux bits de poids faible à 0, il est nécessaire de redimensionner le résultat final de l'opération avant de l'envoyer en sortie du module. Pour ce faire, la librairie `fixed_pkg` fournit la fonction `resize` qui permet de redimensionner un signal `sfixed` ou `ufixed` avec un nouveau nombre de bits pour la partie entière et fractionnaire. Cette opération de reformatage est effectuée selon une politique fixée dans les génériques du package, afin d'arrondir ou de tronquer le nombre de départ. Dans notre cas, nous effectuons des arrondis si la taille d'un signal doit diminuer, et traitons les dépassements de capacité en attribuant au signal problématique la plus grande valeur représentable possible.

4.2.2 Deuxième version : précision variable

La seconde version comprend deux différences majeures par rapport à la première : le facteur λ est une entrée à part entière du module et n'est pas calculé selon un paramètre générique, et la façon d'interpréter

les données des échantillons et des résultats n'est pas apparente dans la description synthétisable ; en d'autres termes, les calculs ne tiennent pas compte de la place de la virgule dans ces données et le même traitement est appliqué quelle que soit cette dernière.

La structure de la description de l'architecture est très similaire à celle de la première version. Ici, les opérations mathématiques sont faites sur des signaux de type `unsigned` à l'aide de la librairie `ieee.numeric_std`. Les opérations de mise à l'échelle par rapport au format des résultats en sortie sont donc effectués manuellement. Nous verrons lors de la partie vérification que cette méthode offre moins de précision que celles de la librairie `fixed_pkg`, car les bits de trop sont simplement tronqués et aucune opération d'arrondissement n'est effectué.

Notons toutefois que, si dans la première version nous changeons le paramètre `round_style` de la librairie `fixed_pkg` de `fixed_round` à `fixed_truncate`, nous obtenons des résultats similaires à ceux fournis par le module de la deuxième version : les opérations d'arrondi sont remplacés par une troncature "pure et dure", comme nous le faisons dans la deuxième version.

4.2.3 Version pipelinée

Dans deux versions du *leaky integrator* présentées ci-dessus, les quatre calculs sont effectués de façon combinatoire et engendrent avec beaucoup de logique à la synthèse. Rappelons que le système doit tourner à une fréquence de 125 MHz, or les versions combinatoires ne peuvent guère fonctionner aussi rapidement. Une comparaison sera effectuée dans le chapitre traitant de l'intégration.

Il est donc nécessaire d'adapter le module en utilisant le principe de pipelining : cela permettra d'augmenter la fréquence maximale de fonctionnement du système, mais ajoutera un peu de latence selon les niveaux du pipeline. Néanmoins, ce petit inconvénient ne sera pas contraignant pour l'application finale. La figure 6 montre le découpage du module combinatoire en plusieurs niveaux de pipeline, dont les registres sont représentés en violet. Les quatre couleurs visibles sur les blocs de calculs sont liés aux quatre résultats.

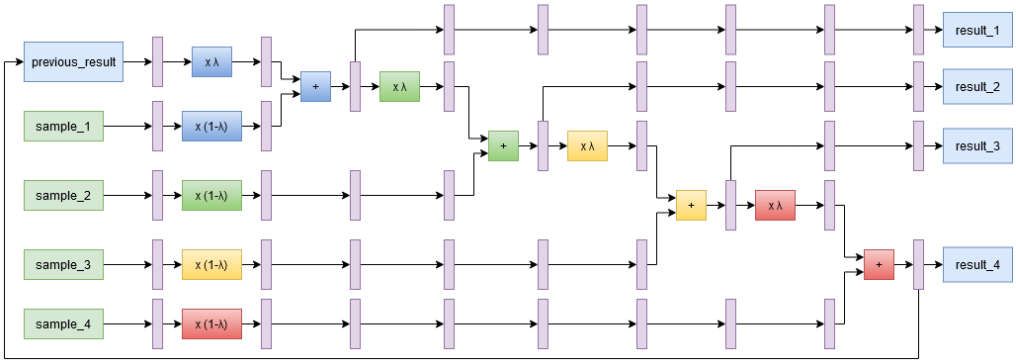


FIGURE 6 – Schéma du module *leaky integator* (version pipeline)

La première version (précision fixe) requiert **10** niveaux de pipeline, la seconde (précision variable) en nécessite **11** ; elle a besoin d'un niveau de plus pour effectuer des opérations d'adaptation du format des résultats.

Comme nous pouvons le remarquer sur le schéma, il est nécessaire de récupérer la valeur du résultat précédent lorsqu'il s'agit de traiter le premier échantillon `sample_1`. Or, ce résultat n'est disponible qu'à partir de 10 (ou 11) cycles d'horloges ; au second cycle, la valeur de `previous_result` sera donc toujours de zéro. Cela aura pour conséquence de fausser les premiers résultats fournis par le module. Néanmoins,

les résultats se stabiliseront peu à peu. Ce petit bémol sera analysé durant le chapitre exposant les résultats des simulations.

4.3 Convolution

Les trois modules de convolution développés, à savoir la convolution à 2, 3 et 5 termes, ont été conçus de manière à pouvoir changer la valeur des coefficients aisément. Comme une opération de convolution sur un échantillon requiert les valeurs des échantillons précédents, un système de sauvegarde a été mis en place. Par exemple, la convolution à deux termes doit utiliser la valeur de l'échantillon précédent, et celle à cinq termes requiert les quatre échantillons précédents. Certaines de ces valeurs proviennent parfois du cycle antérieur.

La figure 7 présente la structure générale d'un module de convolution. Pour chaque opération de convolution appliquée à un échantillon, un vecteur `conv_ops` est préalablement construit, composé du ou des échantillon(s) précédent(s) que requiert le calcul. Au démarrage du système, ces vecteurs sont naturellement indéfinis et contiennent par conséquent la valeur 0. Ils se remplissent au fur et à mesure de l'exécution.

Comme les registres de sauvegarde sont initialisés à 0 au démarrage du système, cela nous permet d'utiliser la même formule pour chaque bloc qui calcule le résultat de la convolution, à savoir :

- $conv\_ops[0] \cdot kernel[1] + conv\_ops[1] \cdot kernel[0]$ pour la convolution à deux termes,
- $conv\_ops[0] \cdot kernel[2] + conv\_ops[1] \cdot kernel[1] + conv\_ops[2] \cdot kernel[0]$ pour la convolution à trois termes,
- $conv\_ops[0] \cdot kernel[4] + conv\_ops[1] \cdot kernel[3] + conv\_ops[2] \cdot kernel[2] + conv\_ops[3] \cdot kernel[1] + conv\_ops[4] \cdot kernel[0]$ pour la convolution à cinq termes.

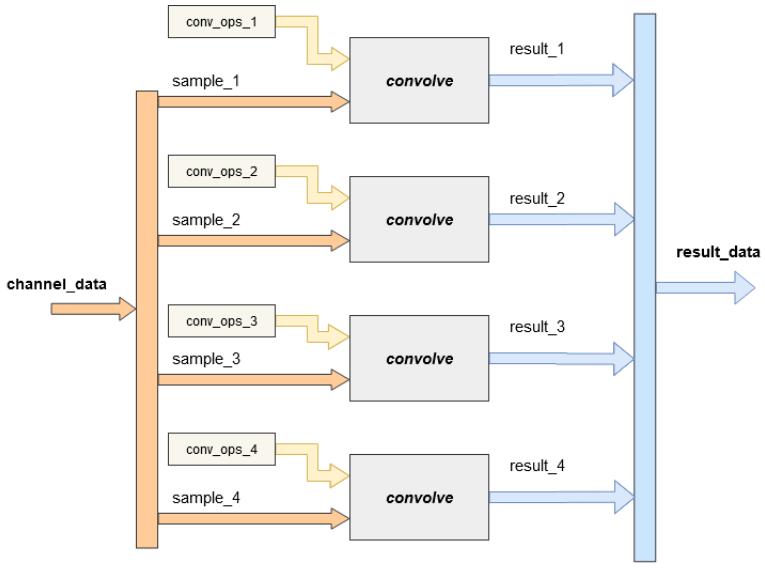


FIGURE 7 – Schéma général des modules de convolution (version combinatoire)

La problématique principale de l'implémentation de ces modules est la façon de stocker les 2, 3 ou 5 coefficients de convolution. Plusieurs méthodes ont été imaginées : dans la première, le module contient le kernel de façon fixe à l'intérieur de sa description. Un package a été mis en place pour conserver tous les kernels utilisés sous forme de tableaux de constantes ; cette première version utilisant la librairie

`fixed_pkg`, le format des coefficients doit être fixe. La seconde alternative prend les coefficients de convolution en entrée du module, dans un format défini au préalable par un paramètre générique. Elle utilise la librairie `numeric_std` pour les calculs, comme la seconde version du *leaky integrator*. Enfin, la troisième version est une extension de la deuxième, avec pour supplément le choix du format des coefficients via une entrée : cela permet ainsi de paramétrer le module après la synthèse et de choisir le format des coefficients le plus adapté, selon la valeur de ces derniers.

4.3.1 Première version : précision et coefficients fixes

Le modèle de la description synthétisable des filtres de convolution est similaire à la première version du *leaky integrator*, à savoir l'utilisation de la librairie `fixed_pkg` et le nombre de bits de fraction des échantillons en paramètre générique.

Afin de conserver les différents kernels, un package de notre invention `coeff_conv_pkg` a été créé. Chaque kernel possède un identifiant précis passé en paramètre générique, qui sert à initialiser les signaux représentant le noyau de convolution à l'intérieur du module. Les coefficients sont récupérés depuis notre package et sont stockés en `sfixed` au format Q8.8 ; les valeurs traditionnelles utilisées en traitement du signal étant généralement des nombres inférieurs à 10 et avec une partie fractionnaire précise à quelques décimales seulement, ce format était un bon compromis pour représenter la plupart des kernels.

4.3.2 Deuxième version : précision variable, coefficients variable avec format fixe

La seconde version est implémentée de la même façon que la seconde variante du *leaky integrator* : les coefficients sont une entrée du module sous la forme d'un `std_logic_vector` de 2, 3, ou 5 valeurs sous 16 bits. Tout comme les valeurs des échantillons, ils sont stockés en `signed`. Le nombre de bits utilisés pour la partie fractionnaire des coefficients est configurable grâce à un paramètre générique `FRACT_BITS_COEFFS`.

Ici, il faut porter une attention particulière aux résultats des calculs qui peuvent vite provoquer des dépassements de capacité ; la librairie `numeric_std`, contrairement à la librairie `fixed_pkg`, ne gère pas ces derniers dans sa fonction `resize`. Si le résultat d'un calcul n'est pas représentable sous les 14 bits utilisés pour les résultats en sortie du module, il est converti en la valeur de la borne inférieure ou supérieure.

4.3.3 Troisième version : précision variable, coefficients variables avec format variable

Cette dernière version est similaire à la précédente, à la différence que le nombre de bits utilisés pour la partie fractionnaire des coefficients est une entrée du module sous 4 bits ; les coefficients étant représentés sous 16 bits, cette valeur peut donc aller de 0 à 15.

Le fait de pouvoir changer le format des coefficients de façon versatile est bien pratique, car l'utilisateur du module peut représenter un grand nombre de kernels et choisir la précision des valeurs la plus judicieuse pour chaque cas d'utilisation des modules de convolution.

4.3.4 Versions pipelinées

Comme pour le *leaky integrator*, il a été nécessaire d'implémenter une variante pipelinée des modules de convolution afin d'atteindre la fréquence de fonctionnement requise. Les figures 8, 9 et 10 présentent les structures en pipeline implémentées pour les module de convolutions.

Le tableau 1 présente le nombre de niveaux de pipeline mis en place dans l'implémentation des modules de convolution.

Nombre de termes	Version	Niveaux de pipeline
2	1	5
2	2	6
2	3	6
3	1	6
3	2	7
3	3	7
5	1	7
5	2	8
5	3	8

TABLE 1 – Nombre de niveaux de pipeline pour les modules de convolution

Nous pouvons remarquer que plus le numéro de la version est petite, plus le sera le nombre de niveaux de pipeline ; cela est dû aux opérations supplémentaires à effectuer dans les variantes 2 et 3. En effet, la librairie `fixed_pkg` utilisée dans les premières versions offre des méthodes aisées de reformater le résultat, le format des données en interne étant fixe ; cela n'est pas le cas du format des données dans les autres variantes. Les versions 2 nécessitent de recadrer le résultat selon le nombre de bits de fraction utilisés pour les coefficients, et les versions 3 nécessitent de multiples comparaisons supplémentaires pour performer ce recadrage, le nombre de bits de fraction des coefficients étant variable.

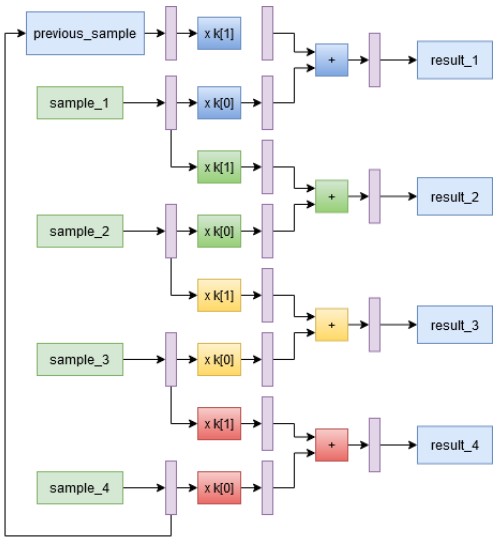


FIGURE 8 – Schéma du module de convolution à 2 termes (version pipelinée)

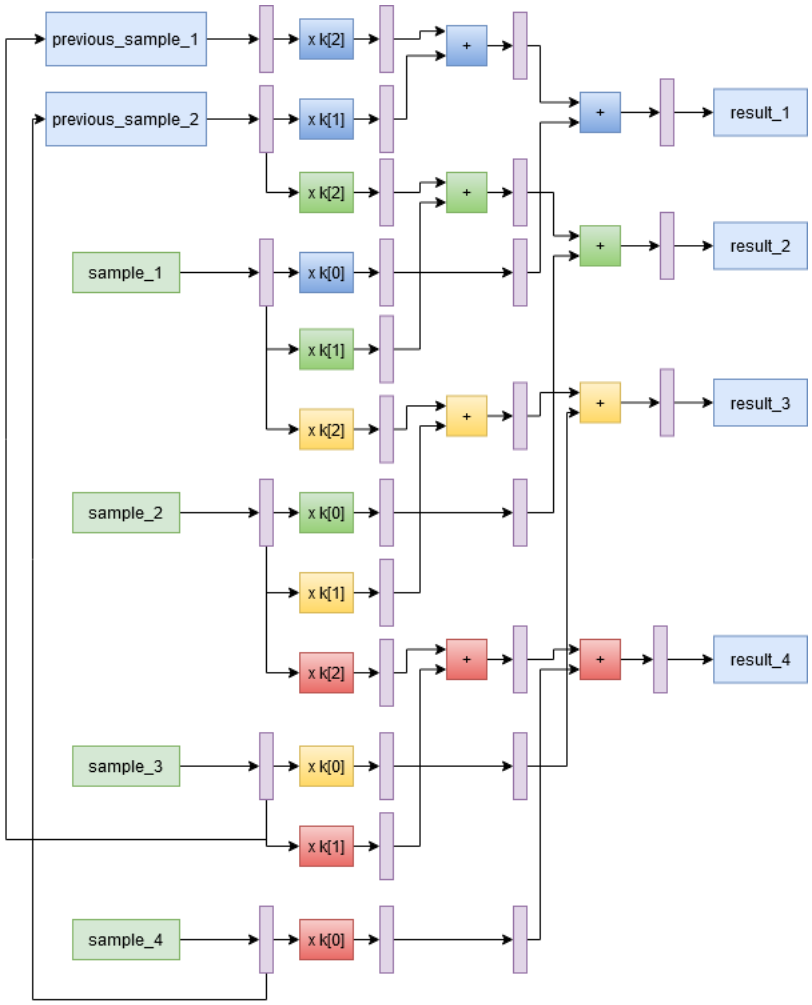


FIGURE 9 – Schéma du module de convolution à 3 termes (version pipelinée)

4.4 Médiane de 5 échantillons

Ce troisième type de filtre, contrairement aux autres, n'utilise pas les échantillons comme opérandes pour des calculs ; il s'agit uniquement d'effectuer des comparaisons dans un ensemble échantillons afin d'en déterminer la valeur médiane. Cette dernière étant mesurée sur cinq échantillons, il est nécessaire de sauvegarder de 1 à 4 échantillons du cycle précédent, tout comme pour le filtre de convolution à 5 termes.

La structure du module du filtre médian est similaire à celle des modules de convolution, comme le montre la figure 11.

La version combinatoire du module a été plutôt aisée à implémenter, car l'arrangement trié des échantillons pris en compte pour le calcul précédent pouvait être réutilisé pour le calcul courant : il suffisait simplement d'extraire la valeur la plus ancienne puis d'ajouter la nouvelle valeur à la bonne place, le reste des valeurs étant déjà classées par ordre croissant. Naturellement, cette implémentation

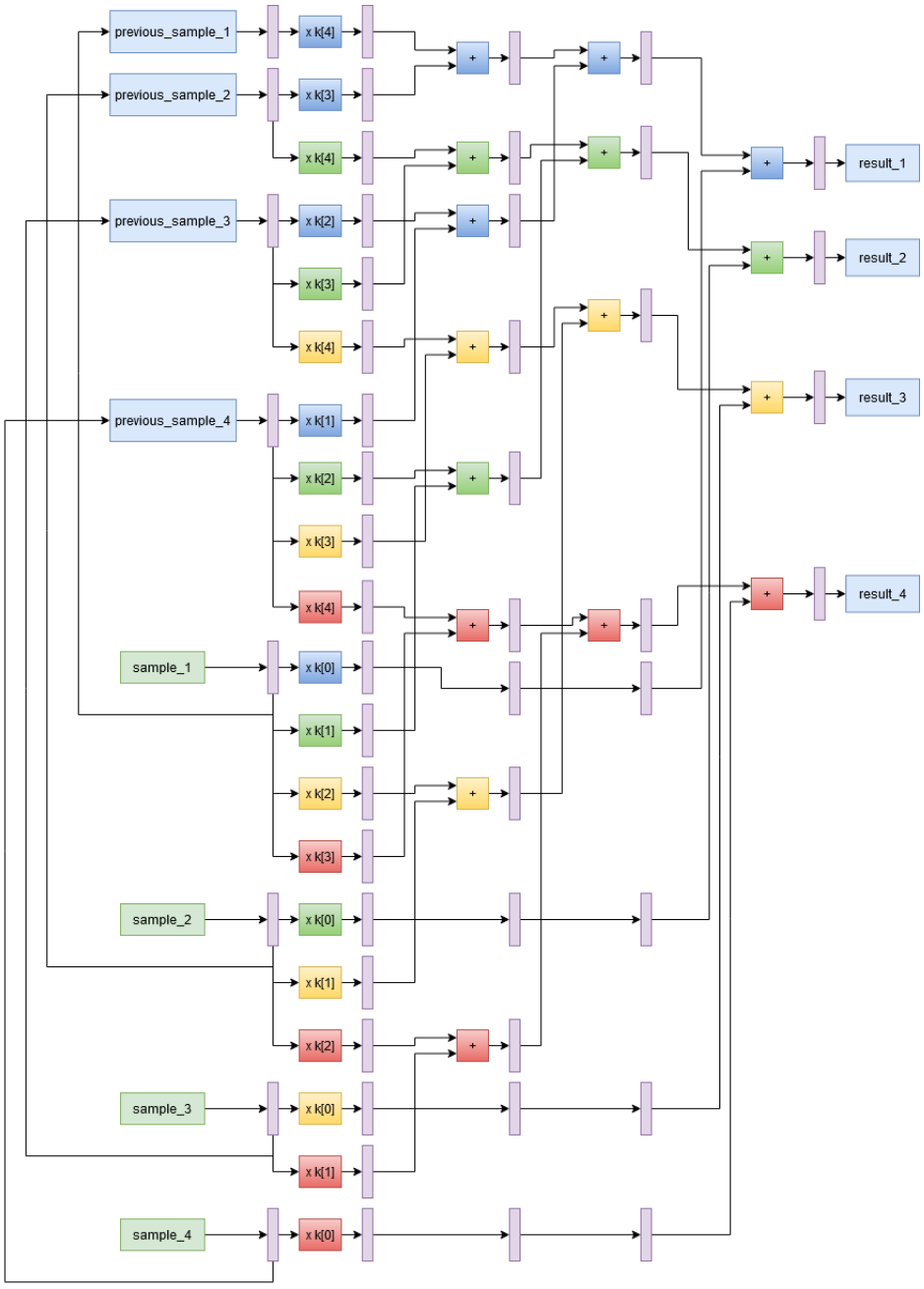


FIGURE 10 – Schéma du module de convolution à 5 termes (version pipelinée)

n'est pas favorable à une adaptation en version pipelinée, car le résultat précédent est utilisé pour le calcul du suivant, comme nous l'avons vu pour le *leaky integrator*.

La version pipelinée du module doit donc considérer les cinq échantillons compris dans le calcul pour

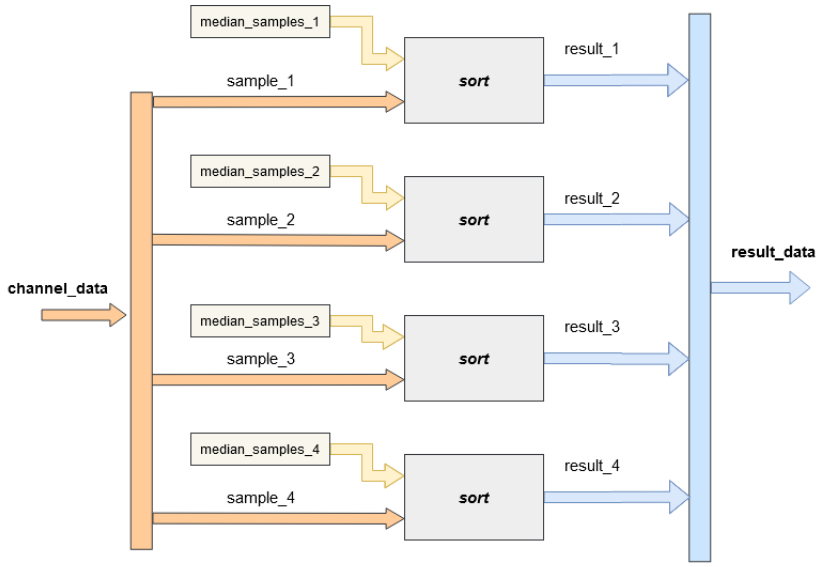


FIGURE 11 – Schéma général du module de la médiane sur 5 échantillons (version combinatoire)

faire un tri par ordre croissant, dont le schéma est visible sur la figure 12. Elle nécessite **7** niveaux de pipeline compte tenu des multiples blocs de comparaisons effectués ; en effet, chaque évaluation requiert 4 blocs de comparaisons, qui génèrent un arrangement trié des échantillons de plus en plus grand (2, 3, 4 puis 5 valeurs triées).

Au terme du tri des cinq valeurs, il suffit d'extraire l'échantillon situé à l'indice 2 de la structure qui contient l'arrangement ordonné et de l'envoyer en sortie du module.

4.5 Moyenne glissante de 25 échantillons

L'implémentation de notre dernier filtre suit la même structure que celle utilisée pour les convolutions ; en effet, rappelons qu'une moyenne glissante de 25 valeurs correspond à effectuer une convolution de 25 termes avec des facteurs valant $\frac{1}{25}$.

Ainsi, l'architecture du module est similaire à la deuxième version des modules de convolution, à la différence que les coefficients ne sont pas en entrée du module, car ils sont constants pour ce cas d'utilisation. Il en va de même pour le format de ces derniers ; la valeur des 25 facteurs de convolution étant de 0.04, il est nécessaire d'utiliser 13 bits de fraction pour représenter la partie fractionnaire de ce facteur. Bien entendu, il n'y a qu'un coefficient constant stocké en interne dans le bon format, et non un kernel de 25 valeurs similaires.

Naturellement, le module a été réalisé avec du pipelining. Afin d'imaginer une telle structure, nous pouvons reprendre le schéma de la figure 10 qui représente la convolution à 5 termes et imaginer qu'il y en a 25 cette fois-ci ; il nous faut mémoriser jusqu'à 21 échantillons des cycles précédents, appliquer la convolution avec pour coefficient constant 0.04, et effectuer 24 additions des résultats intermédiaires au lieu des 4 additions nécessaires pour la convolution à 5 termes. Tout ceci va nécessiter **9** niveaux de pipeline.

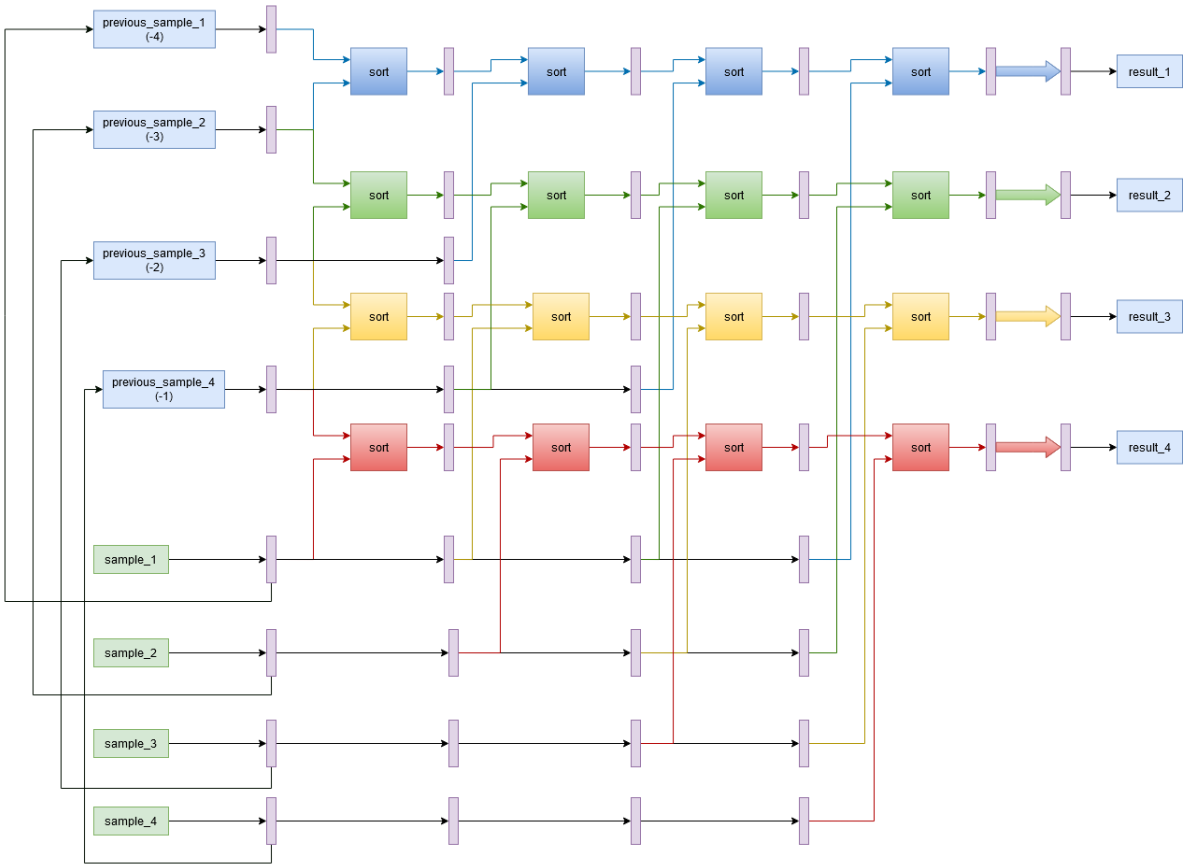


FIGURE 12 – Schéma du module de la médiane sur 5 échantillons (version pipelinée)

4.6 Application de test

Comme les sous-modules de filtrage sont destinés à être chaînés, il est nécessaire de mettre en place un bloc de test qui utilise la combinaison de plusieurs filtres. Rappelons également que le système global doit effectuer un traitement sur quatre canaux à la fois : pour que le module qui contiendra l'application de test soit le plus fidèle possible aux cas d'utilisation de la réalité, ce dernier aura l'interface du système de filtrage complet.

Dans la pratique, les filtres seraient principalement chaînés par groupe de deux. La combinaison médian et convolution avec kernel Savitzky-Golay, par exemple, nous permettrait d'abord de corriger le saut des signaux, puis d'obtenir une mesure de la pente du signal.

Quatre entités ont donc été créées, qui comprennent un couplage de deux modules de filtrage. Le schéma de la figure 13 présente la configuration des filtres que nous avons sélectionnés pour l'application de test ; en réalité, il s'agit d'une habile composition de chaque catégorie de filtre développé. Cela nous permet de confirmer le fonctionnement de tous les filtres en une seule et même entité.

Ceci dit, nous n'utiliserons que la version 2 du *leaky integrator* et les versions 3 des trois modules de convolution ; il s'agit des implémentations les plus susceptibles d'être utilisés en situation réelle, comme il s'agit des variantes les plus configurables.

Dans un nouveau module, il nous suffit de connecter les couples de blocs avec des signaux intermé-

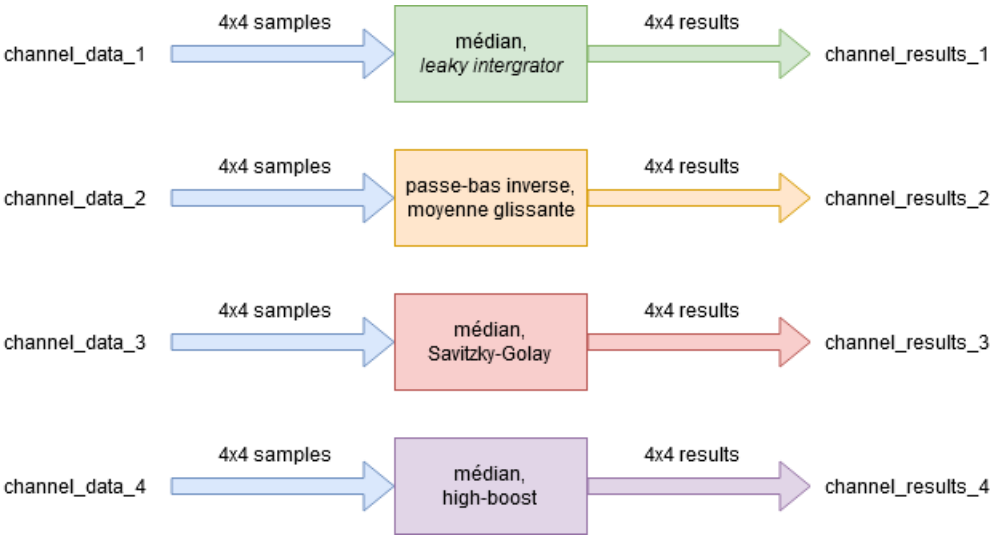


FIGURE 13 – Schéma du module d'application de test

diaires. Notons que le format utilisé n'est pas le même dans tous les modules, certains exploitent des nombres décimaux signés ; il est donc nécessaire de convertir les valeurs qui sortent du module médian en nombres signés pour le premier et les deux derniers blocs. Pour ce faire, nous ajoutons simplement un bit de signe à gauche à 0, ce qui a pour effet de perdre un bit de précision à droite. Il est évident que ce stratagème sera pris en compte lors de la vérification du module.

Cela ne sera pas dramatique pour notre cas d'utilisation, car les données d'entrée seront adaptées pour que ce désagrément ne soit pas pénalisant. Néanmoins, cela pourrait l'être selon les format des données des échantillons dans les cas réels. La problématique de la différence de format en interne des modules n'a pas été explorée dans ce travail. Toutefois nous pourrions imaginer plusieurs options : un bloc convertisseur entre les modules qui n'utilisent pas le même format pour traiter leurs données, l'utilisation d'un des deux bits de poids faible pour effectuer la conversion signé vers non-signé ou inversement, ou encore l'emploi d'une entrée booléenne renseignant sur la présence d'un bit de signe ou non.

5 Vérification

Dans le processus complet de réalisation du design, l'étape de vérification est cruciale. Elle a pour but de contrôler la conformité du système avec les spécifications du cahier des charges, ainsi que de détecter les potentielles erreurs dissimulées dans la description synthétisable du design. C'est une des raisons pour laquelle il est préférable que ce ne soit pas le même développeur qui écrive les bancs de tests et la description synthétisable ; ce projet de travail de Bachelor étant (hélas !) individuel, il est donc important d'essayer de faire abstraction de l'implémentation du design pour l'étape de vérification.

Pour faire ceci, une méthode astucieuse est de décrire le module de banc de test en description non-synthétisable. Cela facilite la rédaction du code en tirant profit du panel d'outils que fournissent les langages utilisés pour la vérification. Dans notre cas, nous rédigerons le code de vérification avant le code de description de chaque filtre : cela nous permettra de nous familiariser avec les opérations effectuées dans les différents modules, sans penser tout de suite aux contraintes d'implémentation dans le design.

En général, le temps utilisé pour écrire un banc de test est tout aussi important que celui consacré à l'implémentation du design en lui-même, c'est pourquoi l'étape de vérification occupe une place considérable dans le planning.

Les fichiers contenant les bancs de tests sont situés dans le dossier `src_tb` du projet. Pour chacun des filtres à implémenter, un fichier test-bench correspondant sera produit. Cela permettra de vérifier le fonctionnement de chaque filtre indépendamment des autres.

5.1 Méthode

Choix du langage

Le langage utilisé pour la vérification du système sera le SystemVerilog, et ce pour diverses raisons ; par rapport au VHDL, ce langage offre des façons plus aisées et plus rapides d'implémenter un banc de test, permettant au développeur d'utiliser un style de code axé logiciel. De plus, l'utilisation d'un langage différent pour cette étape de contrôle accroît la possibilité de dénicher d'éventuelles erreurs tout au long du processus de vérification.

Choix de la méthodologie

Nous utiliserons une méthodologie spécifique à notre cas d'utilisation, car le système n'est pas très complexe à tester. En effet, il ne requiert pas d'utiliser des signaux de synchronisation ou de contrôle. Une librairie tierce spécialisée dans la vérification n'est donc pas nécessaire.

Choix technologique

La vérification sera principalement basée sur les résultats de simulation, avec des bancs de test dirigés comprenant l'utilisation de génération aléatoire pour les signaux de stimulis. La validation sera donnée par la liste des scénarios exécutés et la preuve écrite des résultats des vérifications.

Choix interactif

La vérification s'effectuera à l'aide de simulations automatiques, avec l'exécution d'un test-bench via un script de lancement de simulation pour chaque sous-système de filtrage. Ces scripts, situés dans le dossier `scripts` du projet, comprendront les commandes suivantes :

— Chargement de la librairie de travail
— Compilation des fichiers test-bench et des fichiers de descriptions VHDL synthétisables
— Lancement de la simulation avec adaptation des paramètres génériques

— Ajout des signaux significatifs au chronogramme de simulation avec chargement du `wave.do` associé au *design under test*
— Exécution de la simulation

Les scripts sont au nombre de cinq : un pour le *leaky integrator*, un pour l'ensemble des convolutions, un pour la médiane, un pour la moyenne glissante, puis un dernier pour l'application finale. La liste des descriptions synthétisables étant plutôt dense, il est nécessaire de pouvoir configurer les simulations via des paramètres génériques passés aux scripts, qui peuvent renseigner sur certaines variables : numéro de version du filtre, nombre de bits de fraction des échantillons, activation de l'écriture dans les fichiers de log...

La compilation et le lancement dudit test-bench se feront avec le logiciel QuestaSim.

VRM

Afin d'automatiser les tests, il sera également possible d'utiliser le *Verification Run Manager* de QuestaSim dans les cas propices. Ce dernier offre la possibilité de définir les tests à effectuer avec différents paramètres génériques dans un fichier XML et de les lancer en parallèle. Cela permet d'exécuter tous les scénarios d'un test pour un filtre en particulier et analyser le rapport généré à la fin de son exécution : cette analyse est décisive, elle permet de nous assurer que tous les tests n'ont pas provoqué d'erreurs, ou dans le cas contraire de cibler les scénarios problématiques en vue de les corriger.

Chaque module de filtrage aura son fichier VRM associé afin de lancer facilement les tests de régression, contenus dans le dossier `vrm_files`. Le fichier contiendra l'ensemble des scénarios à jouer pour la vérification de l'entité cible.

5.2 Architecture du test-bench

Comme le design à tester ne comprend pas de signaux de synchronisation ou de contrôle autre que son horloge, il n'est pas nécessaire d'utiliser des librairies tierces spécialisées dans la vérification des systèmes numériques. Une solution de notre invention est suffisante, pour autant qu'elle soit rigoureusement établie.

Cela étant, nous allons utiliser une structure de test-bench de style TLM, pour *Transaction Level Modeling* ; les composants d'un tel système sont modélisés par des modules, qui seront des instances de classes SystemVerilog dans notre cas. Ces modules communiquent entre eux via des canaux, reliés grâce à des ports de communication, et l'ensemble des données est échangé au moyen de transactions. La synchronisation globale du système se fait à l'aide d'actions spécifiques entre modules.

Les différents composants possèdent des ports en tant que producteur ou consommateur et s'échangent les transactions via des méthodes bloquantes ou non-bloquantes. Ces transactions sont placées dans des FIFOs de taille fixe.

Une telle architecture permet de découper le banc de test en deux parties : une partie abstraite au niveau transaction, rythmé par les synchronisations des ces dernières, et une partie concrète au niveau RTL, rythmé par l'horloge du système.

La figure 14 est un schéma de notre structure TLM, générique à tous les bancs de tests des filtres, où apparaissent les différents composants instanciés.

Chaque composant possède un rôle bien défini :

— Le **DUT**, *Design Under Test* représenté en vert, est une instanciation du module de filtrage à tester.
— Le **séquenceur** permet de générer les valeurs de stimulis que l'on souhaite envoyer en entrée du DUT. Parallèlement, il transmet ces mêmes valeurs générées au scoreboard.
— Le **driver** récupère les données du séquenceur et les transmet au DUT par l'intermédiaire d'une interface d'entrée, avec une fréquence adaptée au fonctionnement du DUT. Pour effectuer cette

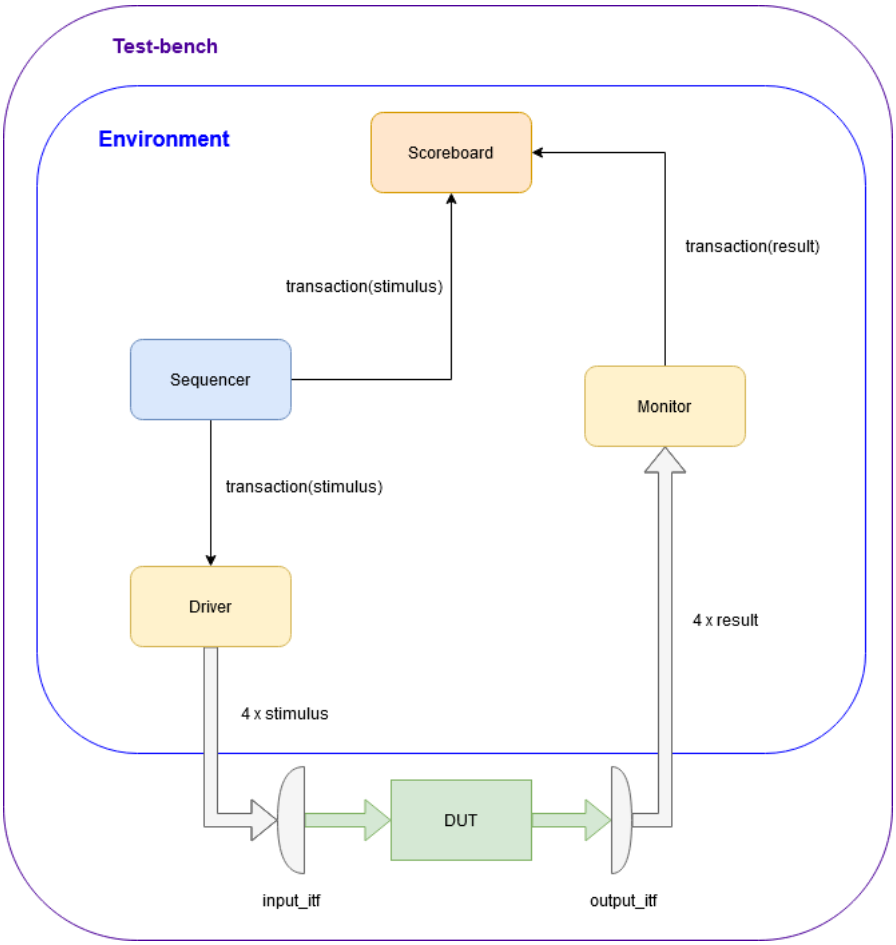


FIGURE 14 – Structure d'un banc de test avec la méthode TLM

opération de transfert, le driver doit attendre d'avoir quatre valeurs du séquenceur, car rappelons que le module de filtrage traite quatre échantillons à la fois.

— Le **moniteur** récupère les données en sortie du DUT et les transmet au scoreboard, en envoyant les résultats obtenus un par un.

— Le **scoreboard** génère des résultats corrects selon la spécification du module de filtrage grâce à un algorithme, qui dans notre cas sera l'application du filtre en question, en description non-synthétisable. Il compare ces résultats avec les données reçues par le moniteur pour détecter les erreurs.

Les différentes transactions transitent entre les composants par l'intermédiaire de trois FIFOs qui peuvent contenir jusqu'à un certain nombre d'éléments. Dans notre cas, nous avons fixé ce nombre à 10 pour nous assurer qu'assez de transactions puissent être stockées, dans le cas où les composants qui récupèrent ou insèrent des transactions ne soient pas totalement synchrones. Ici, les appels aux méthodes `get` et `put` sont bloquants, ceci afin de ne pas oublier de traiter une donnée.

Le rôle de l'**environnement** est d'instancier tous les composants présentés ci-dessus, et les trois FIFOs. Le code du test-bench à proprement parler est un module SystemVerilog standard : il s'occupe

de créer cet environnement, les deux interfaces `input_itf` et `output_itf` correspondant à l'interface du module de filtrage testé. Il a aussi comme tâche d'instancier le DUT avec les bons paramètres.

Afin de pouvoir modulariser le banc de test, pour utiliser la structure TLM avec tous nos filtres, une classe générique `Filter_Parameters` est mise en place. Ses sous-classes contiennent les paramètres spécifiques à chaque filtre et la fonction de calcul du résultat de référence selon le filtre utilisé. Il est également du ressort du module de banc de test d'instancier un objet d'une des sous-classes de `Filter_Parameters` et de le lier aux composants inférieurs dans la hiérarchie qui utilisent les paramètres ; dans notre cas, il s'agit du scoreboard et du séquenceur.

Chaque filtre possède un identifiant unique, qui est connu de tous les composants TLM. Cela permet à ces derniers d'adapter ses actions selon le module de filtrage qui est en cours de simulation : envoyer les bonnes valeurs de stimulis selon le testcase, appeler la bonne fonction de calcul de la référence, attendre le bon nombre de cycles avant de commencer la vérification des résultats.

Il est important de remarquer qu'en vue de s'abstraire de l'implémentation du traitement effectué dans le design, le séquenceur génère et envoie les données de stimulus une par une. Les transactions échangées entre les différents composants ne contiennent donc qu'une seule valeur, que ce soit une donnée de stimuli ou un résultat récupéré du DUT. La synchronisation de ces données aux bornes du design est effectuée par le moniteur et le driver. Ainsi, le module de filtrage à tester est vu comme une boîte noire : cela permet de s'abstraire de l'implémentation du design et cela simplifie grandement la rédaction du banc de test. Néanmoins, il faut établir le plan de vérification méticuleusement pour ne pas oublier certains scénarios qui couvrent les cas limites.

5.3 Généralités

Pour les différents scénarios à mettre en place, nous pouvons lister les paramètres communs entrant en jeu.

Représentation des nombres

Dans le design à tester, les données d'un échantillon sont parfois représentées sous la forme de nombres à virgule fixe, où la place de la virgule, donc le nombre de bits utilisés pour représenter la partie fractionnelle du nombre, peut être passé au module via paramètre générique. Il est donc nécessaire d'effectuer les calculs du résultat de référence en utilisant des données avec la même précision, afin d'obtenir des résultats aussi proches que possible. Ainsi, pour les filtres concernés, nous allons jouer plusieurs scénarios qui prennent en compte différentes valeurs pour ce paramètre.

De plus, les nombres sont représentés sous la forme de 14 bits, car rappelons que les deux bits de poids faible sont toujours à 0. L'utilisation de variables de type `shortreal`, utilisant une représentation en virgule flottante, fausserait les résultats car la précision ne sera pas la même. Cependant, une comparaison entre les résultats de référence calculés selon la méthode du design et ceux générés avec des `shortreal` sera effectuée, ceci afin de mesurer la précision offerte par les différentes techniques de calcul.

Randomisation

Un autre acteur important des scénarios est la génération de nombres aléatoires pour changer les valeurs d'entrée du design. En effet, les tests dirigés peuvent être trop radicaux et trop restreints, tandis que les tests aléatoires permettent de générer des scénarios totalement arbitraires. Cela a pour effet d'accélérer la couverture fonctionnelle.

Le langage SystemVerilog offre une méthode convenable pour la génération de nombres aléatoires : dans notre cas d'utilisation, nous faisons un appel à la fonction `randomize()` directement sur la transaction, avec une contrainte forçant les deux derniers bits à 0.

Marge d'erreur

En raison de la représentation des nombres sous 14 bits seulement, la précision des données n'est pas toujours parfaite ; l'étape de comparaison du résultat de référence avec la sortie observée doit se faire avec quelques précautions.

En effet, dans le cas d'une utilisation de nombres décimaux, le niveau de précision de la partie fractionnelle change selon la place de la virgule : la marge d'erreur doit donc être adaptée pour que la souplesse lors de la vérification des résultats soit plus ou moins marquée.

Rappelons également que le langage utilisé pour la vérification et la description synthétisable n'est pas le même. Ainsi, certaines opérations n'offrent pas la même précision. De petites différences peuvent donc être observables ; ce sujet est traité dans le prochain chapitre. L'application de la marge d'erreur ne sera donc pas effectuée de la même façon pour tous les filtres, cela dépendra des opérations effectuées.

Fichiers de log

Pour récupérer les valeurs d'entrée, les résultats de référence et ceux observés en sortie du module testé, le banc de test met à disposition un paramètre générique permettant l'activation d'une écriture dans 3 fichiers `stimulus_data.txt`, `result_data.txt` et `result_ref.txt`. Ainsi, il est possible d'utiliser ces fichiers pour générer des graphes ou effectuer des mesures pour des comparaisons, comme nous le verrons dans la section suivante.

5.4 Scénarios

Les plans de vérification complets sont fournis en annexe. Les fichiers originaux sont situés à l'emplacement `publi/verif_plans` du projet.

La séquence de test de tous nos scénarios sera la même, à savoir l'instanciation du DUT avec les bons paramètres puis l'envoi des données de stimuli pendant au minimum 400 itérations, à savoir 100 périodes d'horloge car les filtres sont destinés à traiter quatre échantillons par cycle. Dans la même optique, le critère de vérification sera, pour tous les scénarios, l'obtention d'un résultat correct pour chaque échantillon. Concernant les priorités, l'accent est mis sur les scénarios de tests ayant le plus de chances d'être exécutés dans la pratique.

5.4.1 Leaky Integrator

Dans le cadre de la vérification du filtre *leaky integrator*, il est essentiel de rappeler le rôle du paramètre déterministe de l'effet du filtre sur le flux de données : il s'agit du nombre d'échantillons pris en compte dans la fenêtre d'échantillonnage, noté M dans notre formule de référence. Notons que le facteur λ est calculé à partir du paramètre générique M du fichier du banc de test, ceci pour éviter à l'utilisateur de devoir le déterminer lui-même.

Pour aller plus loin dans la vérification, il est aussi nécessaire de changer le nombre de bits représentatifs de la partie fractionnaire tout au long des scénarios. Le tableau 2 présente une version condensée du plan de vérification du module *leaky integrator*, avec la liste des scénarios à jouer lors de la vérification afin de s'assurer du bon fonctionnement du traitement.

Afin de lancer la vérification de la première description du module *leaky integrator*, nous allons utiliser l'outil VRM afin de pouvoir changer les paramètres génériques `M` et `FRACT_BITS`.

Pour la deuxième version, qui ne comporte pas de paramètres génériques, c'est au séquenceur d'envoyer la bonne valeur du facteur λ en entrée du DUT selon le paramètre `M` spécifique au banc de test. Le nombre de bits utilisés pour la fraction, quant à lui, sera défini selon le testcase utilisé ; ce sera au

Testcase	Valeurs échantillons	Valeur de M	Bits de fraction	Priorité
1	aléatoires	10	**1**	2
2	aléatoires	10	**2 à 5**	1
3	aléatoires	10	**6 à 12**	1
4	aléatoires	10	**13**	2
5	aléatoires	**1 à 4**	5	1
6	aléatoires	**5 à 7**	5	1
7	aléatoires	**8 à 9**	5	1
8	aléatoires	**11**	5	2
9	aléatoires	**20**	5	2
10	aléatoires	**100**	5	2
11	**tous bits à 0**	10	5	3
12	**tous bits à 1**	10	5	3
13	**même valeur pour les 4 échantillons**	10	5	3
14	**Sinus bruité**	10	8	1
15	**Impulsions**	10	1	1

TABLE 2 – Testcases *leaky integrator*

scoreboard d'utiliser cette information pour interpréter correctement les données des échantillons et des résultats.

Ces remarques s'appliqueront également au lancement des simulations pour les filtres suivants qui traitent des données en virgule fixe et qui comportent différentes versions d'implémentation.

Calcul de la référence

Pour la première version avec paramètres génériques, rappelons que la description synthétisable utilise des signaux de type `ufixed`. Cela a une influence sur la manière de vérifier les résultats ; en effet, la librairie `fixed_pkg` essaie de conserver la meilleure précision possible quant il s'agit de reformater un signal en réduisant sa taille en bits. Ainsi, le calcul de la référence dans le code du banc de test doit se faire avec les mêmes précautions. Il est donc nécessaire d'avoir deux fonctions de calcul de référence distinctes pour les deux implémentations.

Pour la seconde version, qui comporte le facteur λ en entrée de l'entité, le calcul de la référence s'effectue de façon similaire à celle de la description synthétisable, avec des décalages à droite et une troncature des bits de poids faible selon la précision voulue.

Il est crucial de rappeler que le facteur λ n'est pas représenté avec le même format dans les deux variantes ; la fonction de calcul de la référence doit donc utiliser une variable avec la même précision que dans la description. Cela transgresse la règle d'un test-bench "boîte noire" pour la première version,

car nous devons savoir que le facteur λ est représenté en Q1.8 ; cela est important pour ne pas avoir de grosses différences entre le résultat de référence et le résultat observé. En effet, si la précision du facteur n'est pas la même, cela peut engendrer des différences considérables, notamment dans les cas où il y a très peu de bits de fraction. La cause principale est l'arrondissement effectué sur les résultats par la librairie `fixed_pkg` ; comme la formule du *leaky integrator* utilise le résultat de l'estimation précédente pour calculer l'estimation suivante, la propagation de l'arrondi peut avoir des conséquences dramatiques.

Application de la marge d'erreur

Naturellement, il est nécessaire d'appliquer une certaine marge d'erreur selon le nombre de bits de fraction. Rappelons que deux langages différents sont utilisés pour l'implémentation et la vérification, il peut donc y avoir une infime différence entre le résultat de référence et celui observé en sortie du module. Généralement, il s'agit du bit de poids faible qui change de valeur selon l'arrondi.

Le tableau 3 présente les valeurs que cette marge d'erreur peut prendre, en adéquation avec le nombre de bits de fraction.

Nombre de bits de fraction	Marge d'erreur
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
8	0.00390625
9	0.001953125
10	0.0009765625
11	0.00048828125
12	0.000244140625
13	0.0001220703125

TABLE 3 – Marge d'erreur pour le module *leaky integrator*

Vérification de la version pipelinée

Les remarques ci-dessus sont valables pour les versions combinatoires du module. La vérification de la version pipelinée du module s'est avérée plutôt complexe : dans cette variante, rappelons que le module nécessite un certain temps d'exécution avant que les résultats fournis se stabilisent. Même en adaptant le calcul de la référence au comportement du module, c'est-à-dire en prenant une première valeur d'estimation précédente (`previous_result`, sur la figure 6) de 0 pendant les premières périodes

de l'exécution, la différence entre les résultats observés et les résultats de référence reste assez élevée et il a été difficile de cibler la marge d'erreur à appliquer sans qu'elle soit trop grande et qu'elle ne soit plus vraiment significative. Ce premier filtre étant expérimental et ne figurant pas dans le cahier des charges, la recherche n'a pas abouti et la vérification du fonctionnement de ce module de filtrage en version pipelinée s'est effectuée en comparant visuellement les graphes générés par les données des résultats, comme nous le verrons au chapitre suivant.

5.4.2 Convolutions

Dans nos trois modules de convolution, le paramètre déterministe de l'effet du filtre sur le flux de données est le kernel utilisé ; en effet, les coefficients du noyau de convolution représentent le traitement à appliquer aux échantillons. Ainsi, la vérification de chacun des trois modules doit se faire avec l'utilisation d'un kernel employé dans la pratique. Introduire des valeurs farfelues avec 13 chiffres significatifs n'aurait pas beaucoup de sens, nous utiliserons donc les kernels présentés dans la section traitant de l'analyse des filtres.

Comme pour le plan de vérification du *leaky integrator*, il est nécessaire d'introduire des scénarios où le nombre de bits représentatifs de la partie fractionnaire des échantillons varie ; il en va de même pour les bits de fraction des coefficients dans les versions 2 et 3. Les tableaux 4, 5 et 6 résument les plans de vérification des modules de convolution.

Testcase	Valeurs échantillons	Kernel	Bits de fraction	Priorité
1	aléatoires	[4.5, -3.5] (Q15.1)	**1**	2
2	aléatoires	[4.5, -3.5] (Q15.1)	**2 à 5**	1
3	aléatoires	[4.5, -3.5] (Q15.1)	**6 à 12**	1
4	aléatoires	[4.5, -3.5] (Q15.1)	**13**	2
5	aléatoires	**[0.5, 0.5] (Q15.1)**	5	1
6	**minimales**	[4.5, -3.5] (Q15.1)	5	2
7	**maximales**	[4.5, -3.5] (Q15.1)	5	2
8	**constantes**	[4.5, -3.5] (Q15.1)	5	3
9	**0**	[4.5, -3.5] (Q15.1)	5	3
10	**minimales**	[0.5, 0.5] (Q15.1)	5	2
11	**maximales**	[0.5, 0.5] (Q15.1)	5	2
12	**constantes**	[0.5, 0.5] (Q15.1)	5	3
13	**0**	[0.5, 0.5] (Q15.1)	5	3
14	**Impulsions**	[4.5, -3.5] (Q15.1)	1	1
15	**Impulsions**	[0.5, 0.5] (Q15.1)	1	1

TABLE 4 – Testcases convolution à deux termes

Testcase	Valeurs échantillons	Kernel	Bits de fraction	Priorité
1	aléatoires	[-1, 2, -1] (Q16.0)	**1**	2
2	aléatoires	[-1, 2, -1] (Q16.0)	**2 à 5**	1
3	aléatoires	[-1, 2, -1] (Q16.0)	**6 à 12**	1
4	aléatoires	[-1, 2, -1] (Q16.0)	**13**	2
5	**minimales**	[-1, 2, -1] (Q16.0)	5	2
6	**maximales**	[-1, 2, -1] (Q16.0)	5	2
7	**constantes**	[-1, 2, -1] (Q16.0)	5	3
8	**0**	[-1, 2, -1] (Q16.0)	5	3
9	**Impulsions**	[-1, 2, -1] (Q16.0)	1	1

TABLE 5 – Testcases convolution à trois termes

Testcase	Valeurs échantillons	Kernel	Bits de fraction	Priorité
1	aléatoires	[-2, -1, 0, 1, 2] (Q16.0)	**1**	2
2	aléatoires	[-2, -1, 0, 1, 2] (Q16.0)	**2 à 5**	1
3	aléatoires	[-2, -1, 0, 1, 2] (Q16.0)	**6 à 12**	1
4	aléatoires	[-2, -1, 0, 1, 2] (Q16.0)	**13**	2
5	**minimales**	[-2, -1, 0, 1, 2] (Q16.0)	5	2
6	**maximales**	[-2, -1, 0, 1, 2] (Q16.0)	5	2
7	**constantes**	[-2, -1, 0, 1, 2] (Q16.0)	5	3
8	**0**	[-2, -1, 0, 1, 2] (Q16.0)	5	3
9	**Impulsions**	[-2, -1, 0, 1, 2] (Q16.0)	1	1

TABLE 6 – Testcases convolution à cinq termes

Pour la première description des modules de convolution, il nous utilisons VRM pour lancer les tests et changeons les paramètres génériques dans le fichier VRM, à savoir l'identifiant du kernel à utiliser et le nombre de bits de fraction des échantillons. Pour les deux autres descriptions, c'est au séquenceur d'envoyer les bons paramètres aux bons formats au DUT selon le testcase.

Calcul de la référence

Le calcul de la référence se fait de façon similaire au calcul des résultats dans la description synthéti-

sable ; la fonction de calcul prend en paramètre le kernel de convolution de notre choix et les échantillons à utiliser comme opérandes. Chaque résultat de référence est calculé en `shortreal` et en virgule fixe au format souhaité.

Il faut naturellement prendre en compte les dépassements de capacité possibles, c'est pourquoi les résultats de référence sont comparées aux bornes maximales et minimales prédéterminées, et prennent leur valeur si les valeurs ne sont pas représentables.

Cette mesure n'était pas nécessaire pour le filtre *leaky integrator*, car les facteurs appliqués aux opérandes n'étaient pas susceptibles de provoquer des *overflow*.

Application de la marge d'erreur

Les valeurs des coefficients de convolution utilisés n'ayant généralement guère plus d'un ou deux chiffres significatifs, l'application d'une marge d'erreur sur les résultats générés par le DUT n'est pas forcément utile ; néanmoins, si l'utilisateur souhaite utiliser un filtre qui nécessite des coefficients très précis, il se peut qu'une petite adaptation soit nécessaire. Dans ce cas, la marge d'erreur prend la même valeur que celle utilisée pour le *leaky integrator*, dont les valeurs sont présentées sur le tableau 3.

Nous l'avons quand même prise en compte lors de la vérification des résultats, bien qu'une petit étude a été effectuée pour vérifier son utilité : en raison de la simplicité des kernels utilisés dans les scénarios, elle n'est jamais indispensable.

5.4.3 Médiane

Le plan de vérification du filtre médian sur 5 échantillons est sans doute le plus sommaire ; en effet, aucun paramètre n'entre en jeu et aucun calcul n'est effectué sur les échantillons. De plus, les valeurs des échantillons sont considérées comme des données entières, il n'y a donc aucun paramètre à modifier quant à l'interprétation des données de sortie. Il est néanmoins important d'observer le comportement du filtre médian lorsque les données d'entrée sont constantes, ou sont des valeurs atypiques.

Le tableau 7 présente les scénarios à jouer lors de la vérification.

Testcase	Valeurs échantillons	Priorité
1	aléatoires	1
2	minimales	2
3	maximales	2
4	constante	2
5	impulsions	1

TABLE 7 – Testcases médiane sur 5 échantillons

Calcul de la référence

Pour calculer le résultat de référence, une fonction prenant comme argument un tableau de 5 échantillons a été mise en place. Un simple tri à bulles réarrange les valeurs du tableau par ordre croissant, puis la fonction renvoie comme résultat la valeur stockée à l'indice 2 du tableau. Il n'y a aucune application

de marge d'erreur particulière à mettre en place, car aucune opération ne risque de modifier les valeurs des échantillons.

5.4.4 Moyenne glissante

Le plan de vérification de la moyenne glissante sur 25 échantillons, visible tableau 8 est similaire à celui de la médiane, avec pour supplément un changement du format des valeurs d'entrée qui sont représentées en virgule fixe, en variant leur nombre de bits de fraction.

Testcase	Valeurs échantillons	Bits de fraction	Priorité
1	aléatoires	**1 à 13**	1
2	minimales	4	2
3	maximales	4	2
4	constante	4	2
5	impulsions	1	1

TABLE 8 – Testcases moyenne glissante de 25 échantillons

Bien que l'implémentation de ce filtre soit semblable à celle des convolutions, rappelons que le kernel comprend des coefficients fixes et ne changera donc pas de configuration, c'est pourquoi le plan de vérification n'est pas aussi fourni que pour les convolutions.

Calcul de la référence

Le calcul des résultats de référence se fait de la même façon que pour les convolution classiques, avec 25 termes dans notre cas. Attention toutefois à la représentation du facteur de convolution 0.04, qui est doit être stocké avec *exactement* le même format que dans la description synthétisable, à savoir virgule fixe avec 1 bit pour la partie entière et 13 pour la partie fractionnaire.

La même marge d'erreur que pour les modules de convolution standards est appliquée, en raison de la précision du facteur de convolution utilisé. Aucune comparaison du résultat avec des bornes inférieures ou supérieures de représentation n'est effectuée, car le calcul appliqué aux données ne risque pas d'engendrer des dépassement de capacité.

5.4.5 Application de test

Du point de vue du code de vérification, l'application de test a été traitée comme un filtre normal afin de faciliter la rédaction du banc de test. Néanmoins, adapter les composants de l'architecture TLM à une interface de sortie et d'entrée différente aurait été long et finalement peu utile, les quatre sous-modules pouvant être vérifiés individuellement avec la structure existante. Ainsi, la génération du résultat de référence est adaptée selon le sous-module testé parmi les quatre, et exploite les fonctions de calcul mises en place pour les contrôles spécifiques de chaque filtre.

Un seul scénario a été mis en place pour le test de cette application, à savoir l'introduction de valeurs réelles représentant des impulsions ; l'application de test ayant pour but de représenter un cas d'utilisation pratique des filtres, il serait incongru d'introduire des données qui ne se conformeraient pas à la réalité.

De plus, des nombreux tests comprenant de la génération aléatoire de stimulis ont déjà été effectués de manière indépendante pour chaque filtre.

6 Simulation

6.1 Démarche

La première phase de simulation de chaque module de filtrage s'effectue avec le lancement du banc de test associé et la vérification des résultats en sortie avec les résultats de référence générés par le scoreboard (voir chapitre précédent). Ce dernier génère une erreur si les résultats diffèrent après l'application de la marge d'erreur. Ainsi, il est trivial de vérifier si la simulation a engendré une erreur ou non dans les logs de QuestaSim. Si VRM est utilisé, le petit rapport généré au terme des tests de régression nous renseigne rapidement sur la bonne exécution de ces derniers et sur l'exactitude des résultats.

Depuis le chronogramme de QuestaSim, il est possible de représenter les données des échantillons et les résultats sous la forme d'un graphe grâce à la commande suivante :

```
add wave -noupdate -format Analog-Step <options> <nom du signal>
```

Cela permet de vérifier rapidement si l'effet du filtre en question génère une courbe respectant nos attentes. La figure 15 montre un exemple avec l'application du filtre *leaky integrator* sur un signal correspondant à un sinus légèrement bruité.

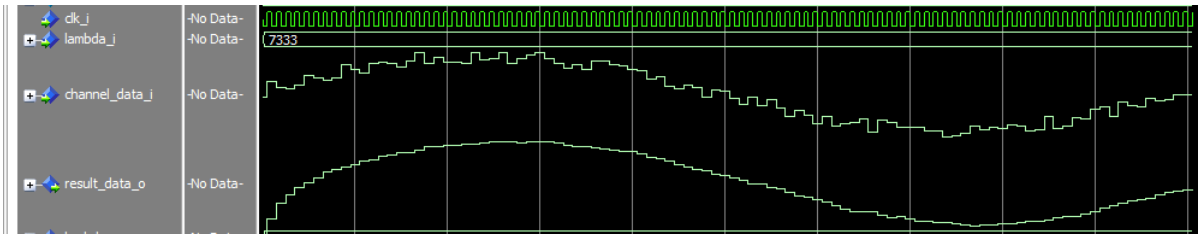


FIGURE 15 – Exemple de vue *wave* QuestaSim avec représentation analogique

Un fichier `wave.do`, qui enregistre le format souhaité pour l'affichage des signaux dans le chronogramme de simulation, a été mis à disposition pour le test de chaque filtre.

Afin de faciliter la visualisation des résultats, des graphes ont été générés à partir des fichiers de logs récupérés après une simulation, ceci à l'aide de MATLAB. Les simulations effectuées pour la génération de ces graphes ont été exécutées sur 400 itérations, soit 100 cycles d'horloge ; rappelons que quatre échantillons sont traités à la fois. Cela nous permet d'avoir une bonne vision d'ensemble de l'effet du filtre obtenu sur les données.

La deuxième phase consiste en l'introduction d'un flux de données représentant une structure d'impulsions à 25 nanosecondes, récupérées depuis un fichier contenant des mesures réelles du CERN. Cela permet de valider le bon fonctionnement du filtre avec la forme de flux de données la plus adaptée à la réalité.

6.2 Résultats et analyse

6.2.1 Leaky Integrator

Données aléatoires

La partie gauche de la figure 16 représente les valeurs des échantillons d'entrée lorsqu'on les génère aléatoirement, avec un nombre de bits de fraction de 4. La partie droite nous montre les estimations obtenues pour ces données d'entrée, après l'application du filtre avec le paramètre `M` à 10, soit λ à 0.9

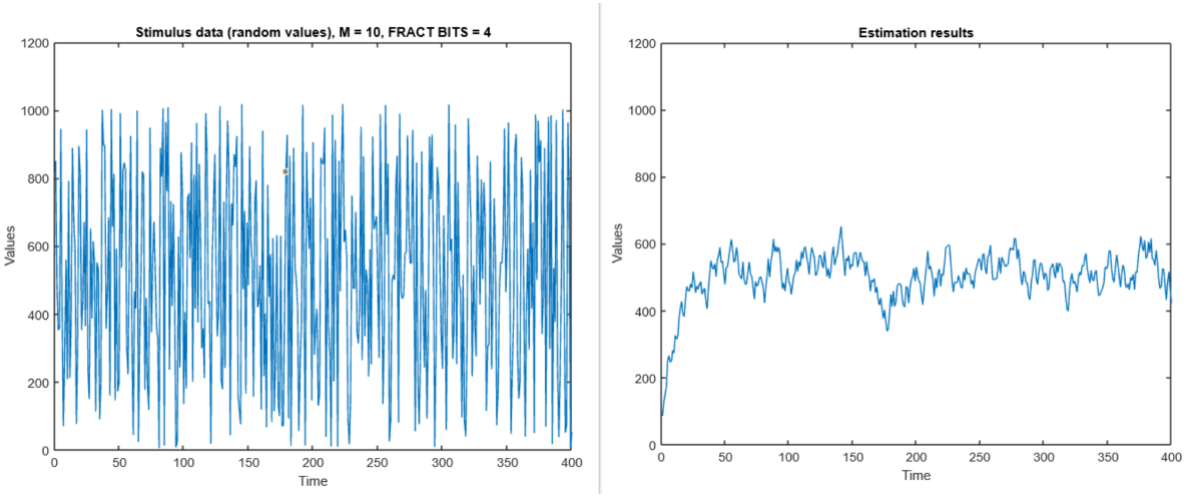


FIGURE 16 – Données de stimulus aléatoires avec 4 bits de fraction et résultats avec $\lambda = 0.9$

Les mêmes graphes, figure 17, sont générés avec un nombre de bits de fraction de 6 et `M` à 6, soit λ à 0.8333. Comme plus de bits de fraction sont utilisés, la plage de valeur se retrouve réduite par rapport au graphe précédent.

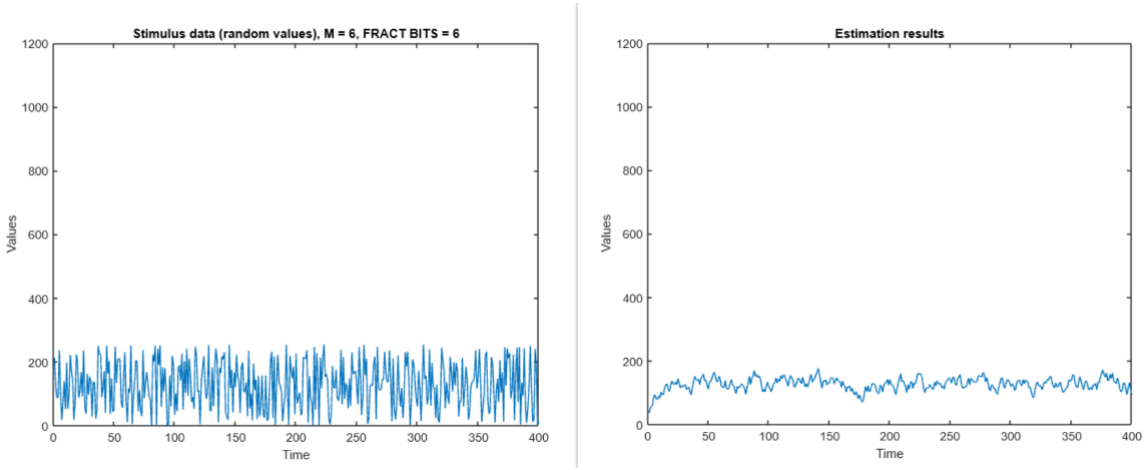


FIGURE 17 – Données de stimulus aléatoires avec 6 bits de fraction et résultats avec $\lambda = 0.8333$

Réflexions sur la précision

Comme les nombres sont représentés en virgule fixe, les résultats ne sont pas aussi précis qu'en virgule flottante. Nous pouvons effectuer une comparaison des deux représentations afin de quantifier

la différence. Pour ce faire, un résultat de référence de type `shortreal`, fourni par SystemVerilog, est également calculé dans le code du banc de test, pour les deux versions du module.

Attention toutefois à la représentation du facteur λ utilisée dans les calculs : pour la première version, il s'agit du résultat d'une division qui utilise le paramètre `M`. Il n'est parfois pas possible de stocker la valeur *exacte* de λ dans la description synthétisable. Par exemple, avec un `M` $= 10$, λ devrait être égal à 0.9, or dans la description il est très légèrement plus petit (un peu plus de 0.89), compte tenu du nombre de bits utilisés pour le représenter.

Il faut donc que sa valeur stockée en `shortreal` soit la même que dans la description, sinon de grosses différences seront observables. Pour la première version, λ est stocké sous le même nombre de bits que dans la description (Q1.8), puis cette valeur est convertie en `shortreal`. Pour la deuxième version, nous suivons le même principe : λ est une entrée du module représenté en format Q1.15, convertie en `shortreal` avant de procéder aux calculs.

Le tableau 9 présente la différence que peuvent avoir les trois représentations du facteur λ. Aussi petites soient-elles, ces divergences peuvent vite influer sur les résultats.

M	λ virgule flottante	λ virgule fixe Q1.8	λ virgule fixe Q1.15
1	0.0	0.0	0.0
2	0.5	0.5	0.5
3	0.66666667	0.66796875	0.666656494140625
4	0.75	0.75	0.75
5	0.8	0.80078125	0.79998779296875
6	0.83333333	0.83203125	0.833343505859375
7	0.85714286	0.85546875	0.857147216796875
8	0.875	0.875	0.875
9	0.88888889	0.890625	0.888885498046875
10	0.9	0.89843750	0.0.899993896484375
11	0.90909091	0.91015625	0.909088134765625
50	0.98	0.98046875	0.95001220703125
100	0.99	0.98828125	0.989990234375

TABLE 9 – Valeurs de λ en représentation en virgule flottante et en virgule fixe pour le filtre *leaky integrator*

Le tableau 10 présente les divergences observées, pour les deux versions du calcul de la référence. La différence est calculée comme suit :

```
abs( résultat de référence (virgule fixe) - résultat de référence (shortreal) )
```

Il est vite visible que l'utilisation de la première méthode offre une précision un peu plus optimisée quant aux résultats obtenus ; rappelons que la première version utilise la librairie `fixed_pkg`, dont les

Nb. bits de fraction	Diff. max v1 (λ Q1.8)	Diff. max v2 (λ Q1.15)
1	0.25	0.499542236328125
2	0.123046875	0.2497711181640625
3	0.0615234375	0.12488555908203125
4	0.03125	0.062442779541015625
5	0.015625	0.031221389770507813
6	0.0078125	0.0156106948852539
7	0.00390625	0.0078053474426269531
8	0.001953125	0.0039026737213134766
9	0.0009765625	0.0019513368606567383
10	0.00048828125	0.0009756843032836914
11	0.000244140625	0.0004878342151641 8457
12	0.0001220703125	0.0002439171075820 9229
13	0.00006103515625	0.0001219585537910 4614

TABLE 10 – Différence entre les résultats de référence en virgule flottante et virgule fixe pour le filtre *leaky integrator*

méthodes offrent une meilleure précision que la seconde. Il est intéressant de constater que la différence dans la première version équivaut environ à la moitié de la différence de la deuxième.

Si toutefois nous avions calculé le résultat de référence en `shortreal` avec la valeur de λ en virgule flottante, nous aurions obtenu des différences considérables. La figure 18 montre la différence obtenue sur 400 itérations lorsqu'on soustrait le résultat calculé avec la valeur *exacte* de λ codée en `shortreal` au résultat calculé avec la valeur de λ en virgule fixe sur 9 bits, avec $\lambda = 0.9$.

Cela montre les conséquences désastreuses que peuvent avoir une succession d'opérations d'arrondissement sur les données, et appuie le fait qu'il faut s'adapter à la représentation des données en interne du design lorsqu'il s'agit de vérifier le fonctionnement de ce dernier. Une autre méthode de vérification aurait été d'utiliser un résultat de référence de type `shortreal` uniquement, puis d'appliquer une marge d'erreur plus grande. Toutefois, c'est une alternative dangereuse car elle pourrait ne pas détecter les véritables erreurs d'implémentation.

Données réelles

Comme nous l'avons vu au chapitre traitant de l'implémentation du module, l'adaptation du système combinatoire en version pipelinée produit un petit écart des résultats au début de l'exécution, provoqué par l'indisponiblité du dernier résultat du cycle précédent.

Pour pallier à ce problème, plusieurs solutions ont été explorées, mais vite abandonnées. Le challenge principal était d'obtenir une façon d'assigner une valeur correctrice au signal qui fausse les premiers résultats, à savoir la valeur de l'estimation précédente qui est de 0 trop longtemps. Il ne fallait pas non

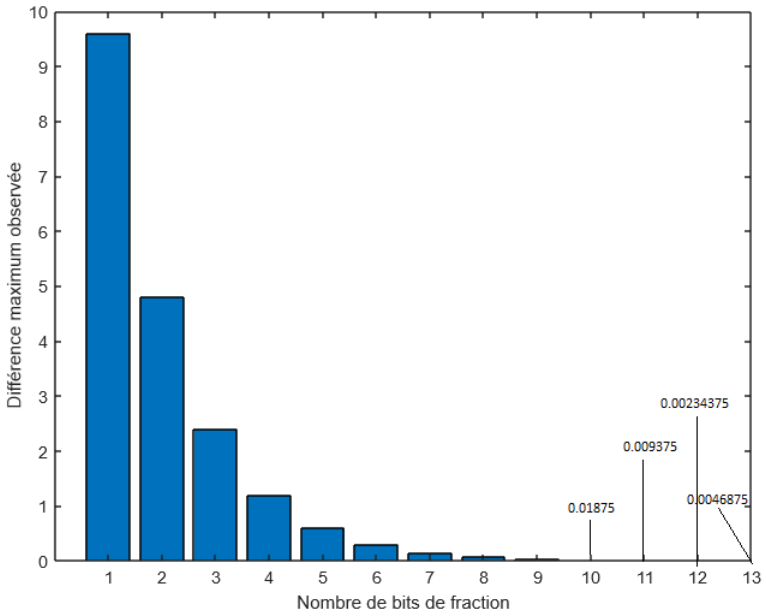


FIGURE 18 – Différence entre les résultats de référence en virgule flottante avec valeur exacte de λ et résultats en virgule fixe avec λ sous 9 bits pour le filtre *leaky integrator*

plus ajouter de calculs supplémentaires, car cela aurait ajouté de la logique au circuit ; c'est un prix trop cher à payer pour une correction qui n'est utile que pour les premiers cycles.

Si l'on voulait tout de même effectuer une correction du signal, il était nécessaire d'avoir un compteur interne jusqu'à 10 ou 11 cycles d'horloge selon la version du module *leaky integrator* et modifier la valeur du signal contenant la dernière estimation du cycle précédent : un test a été effectué en lui attribuant la valeur du dernier échantillon du cycle antérieur (`sample_4` sur le schéma de la figure 6, qui présente le schéma du module pipeliné). Néanmoins, la valeur de cet échantillon pouvant être arbitraire, c'est une solution peu fiable selon la forme du flux entrant.

La figure 19 montre un exemple du résultat obtenu sur un flux de données représentant des impulsions, après application du *leaky integrator*. Le signal original est représenté en bleu, la référence générée par le banc de test est en rouge, les résultats du module standard sont en jaune et les résultats du module modifié sont en violet. Au début, les résultats fournis par le module "corrigé" sont moins aberrants que ceux du module standard ; néanmoins, lorsque le temps s'écoule, nous pouvons observer une nette stabilisation du signal du module standard avec toutefois des valeurs un peu en dessous de la courbe de référence, alors que le signal du module corrigé possède toujours des valeurs plus élevées que la courbe de référence.

Dans tous les cas, nous sommes parvenus à la conclusion qu'il n'était pas nécessaire d'effectuer la correction, les premières valeurs n'étant pas extrêmement importantes dans notre cas d'utilisation.

6.2.2 Convolutions

Données aléatoires

Bien que la vérification d'un filtre de convolution avec des données aléatoires soit essentielle pour

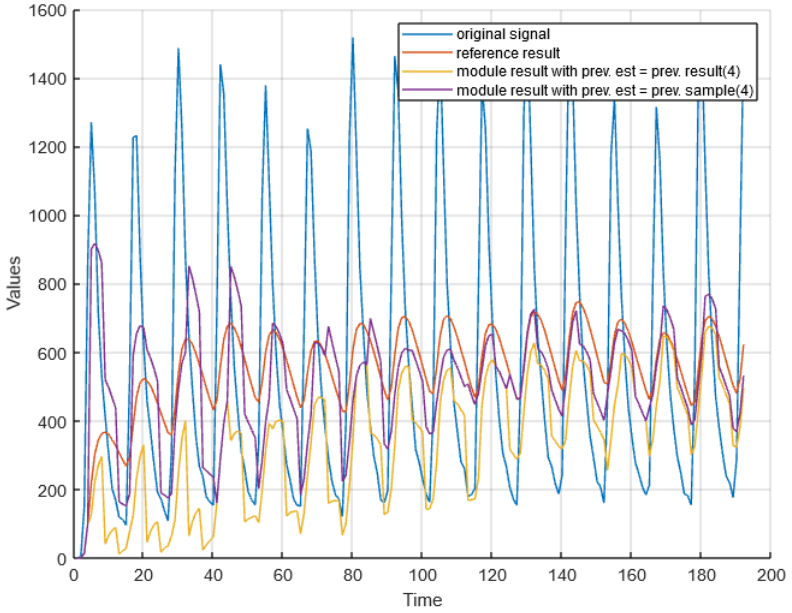


FIGURE 19 – Comparaison des résultats du *leaky integrator* avec flux de données réelles, 1 bit de fraction et $\lambda = 0.9$

la preuve de son bon fonctionnement, les graphiques résultants ne sont pas visuellement expressifs de l'authenticité du module selon le kernel utilisé, c'est pourquoi nous n'en présenterons qu'un seul dans cette section.

Les données récoltées avec le filtre "lissant" qui utilise le noyau de convolution `[0.5, 0.5]` peuvent être visuellement parlant, comme le montre la figure 20 où les stimulis au format Q8.6 sont représentés en bleu et le résultat du filtre en rouge : ici, nous pouvons observer une légère réduction des valeurs de stimulus.

Données réelles

Il est bien plus intéressant de découvrir l'effet des filtres de convolution sur les mesures réelles du CERN. La première simulation met en évidence l'effet du filtre passe-bas inverse sur le flux de données, soit la convolution à deux termes avec le kernel `[4.5, -3.5]`. La même opération a été effectuée de façon logicielle avec la fonction MATLAB `conv()` pour comparer les deux résultats.

Sur le graphe de la figure 21, nous pouvons voir que les résultats fournis par le module sont pratiquement identiques à ceux que fournit la fonction `conv()` de MATLAB ; la seule différence est visible sur les pics de valeurs les plus hautes, car comme le module utilise des valeurs codées sur 14 bits, certaines ne sont pas représentables et sont bloquées à la valeur de la borne supérieure.

Les simulations des modules de convolution à trois et cinq termes suivent le même processus. Les graphes générés depuis les résultats du module sont visibles figures 22 et 23.

La dernière simulation combine l'utilisation de deux kernels de convolution à deux termes : d'abord le filtre passe-bas inverse puis la moyenne sur deux échantillons comme dans la section précédente. L'effet obtenu est visible à la figure 24.

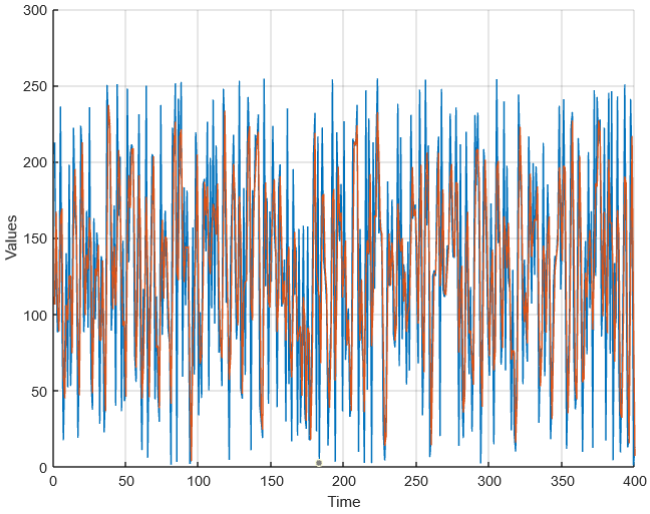


FIGURE 20 – Application du filtre lissant (convolution à deux termes) sur des données aléatoires (bleu : stimulis, rouge : résultats)

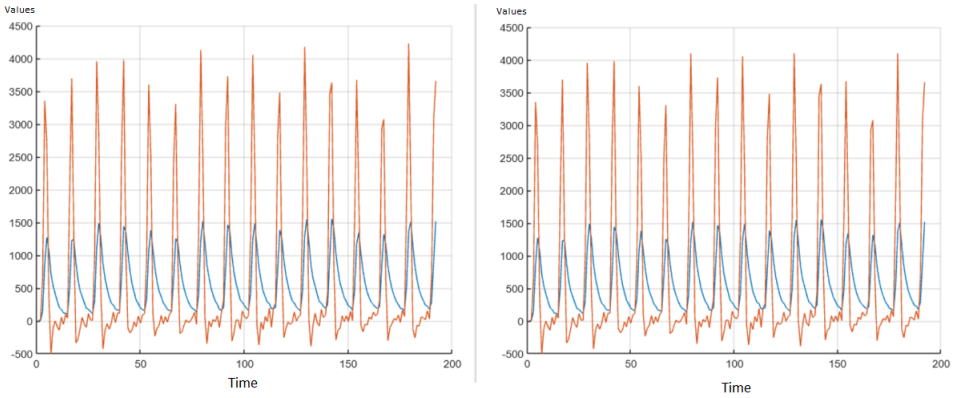


FIGURE 21 – Filtre passe-bas inverse (convolution à deux termes) avec données réelles, module (droit) et convolution MATLAB (gauche)

6.2.3 Médiane

Données aléatoires

La figure 25 nous montre l'effet obtenu par le filtre médian sur un flux de données aléatoires.

Données réelles

La figure 26 montre l'effet du filtre médian sur le flux de données réel. Nous pouvons clairement voir que le flux de données sortant suit la même dynamique que les impulsions, avec des valeurs légèrement plus petites.

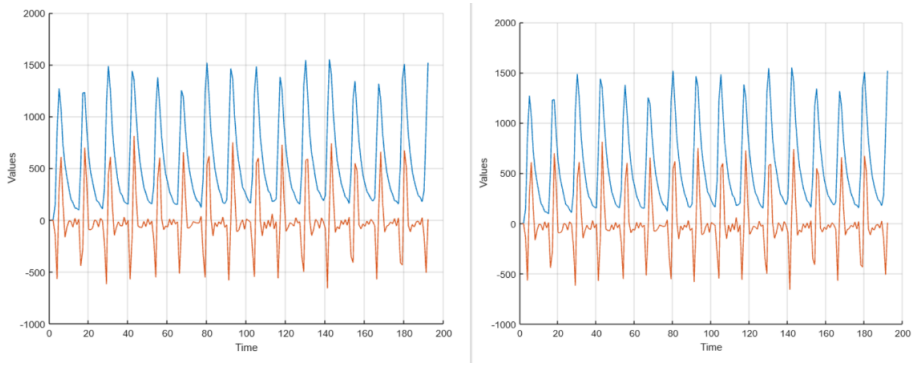


FIGURE 22 – Filtre *high boost* (convolution à trois termes) avec données réelles, module (droit) et convolution MATLAB (gauche)

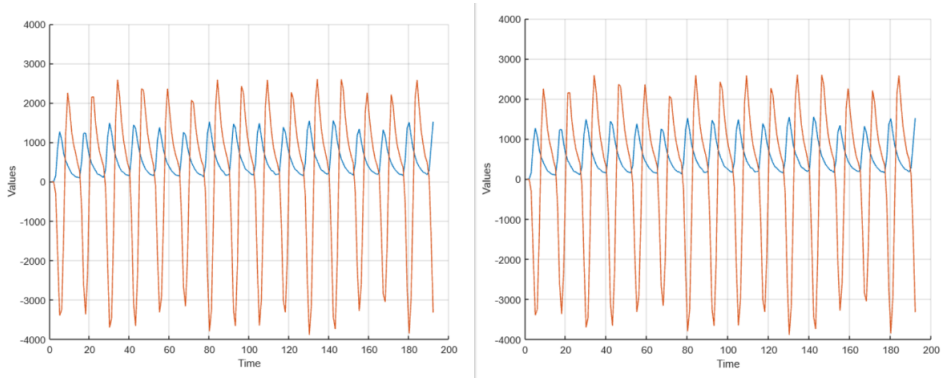


FIGURE 23 – Filtre Savitzky-Golay (convolution à cinq termes) avec données réelles, module (droit) et convolution MATLAB (gauche)

6.2.4 Moyenne glissante

Données aléatoires

La figure 27 nous montre l'effet obtenu par la moyenne glissante de 25 échantillons sur un flux de données aléatoires. Comme nous pouvons le deviner, il s'agit du même type de résultat que celui que nous pouvons obtenir avec un *leaky integrator* et un facteur λ proche de 1.

Données réelles

La figure 28 montre l'effet de la moyenne glissante sur le flux de données réelles. Comparé au *leaky integrator*, le résultat obtenu est plus lisse et nous avons perdu la structure d'impulsions de départ, qui était toujours légèrement marquée avec *leaky integrator*. Plus il y a d'échantillons pris en compte dans le calcul, plus la forme d'impulsions de base s'effacera.

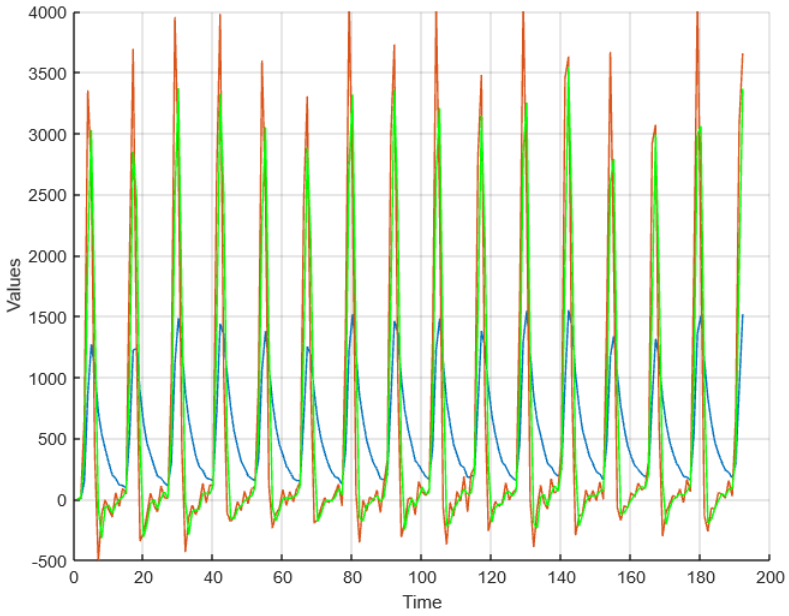


FIGURE 24 – Chaînage de deux convolutions (bleu : signal original, rouge : filtre passe-bas inverse, vert : filtre passe-bas inverse + moyenne glissante de 2 échantillons)

6.2.5 Application de test

La figure 29 présente le premier graphe généré avec les résultats du premier bloc de l'application de test. Il présente les données observées pour le chaînage des filtres médian et *leaky integrator* avec un facteur λ de 0.9. Afin de pouvoir observer la stabilisation du signal à cause du temps de configuration du *leaky integrator*, la simulation a été effectuée sur un grand nombre de cycles. Nous pouvons faire le constat de cette stabilisation à partir de l'itération 250, où la dynamique du signal résultant semble plus solide.

La figure 30 comporte les résultats pour le second bloc de l'application, à savoir le filtre passe-bas inverse suivi de la moyenne glissante. Visiblement, cette combinaison a pour effet de stabiliser totalement le signal produit par le filtre passe-bas.

Le graphe de la figure 31 nous présente les résultats du troisième bloc, qui comprend une médiane sur cinq échantillons puis un filtre *high boost*. La médiane a pour effet de corriger les sauts des signaux, ce qui ne risquera pas de les mettre en évidence avec l'application du *high boost*.

Finalement, les résultats du dernier bloc sont présentés à la figure 32. Il s'agit de la combinaison d'un filtre médian suivi d'une convolution avec kernel de type Savitzky-Golay, qui nous permet d'obtenir une mesure de la pente du signal.

En bonus, un dernier graphe a été généré avec un chaînage médiane + moyenne glissante, figure 33. Nous pouvons constater que contrairement au *leaky integrator*, effectuer une moyenne glissante sur 25 échantillons avec ce genre d'impulsions a pour effet de stabiliser presque totalement le signal et de perdre la dynamique première du flux d'entrée, comme nous l'avons vu précédemment.

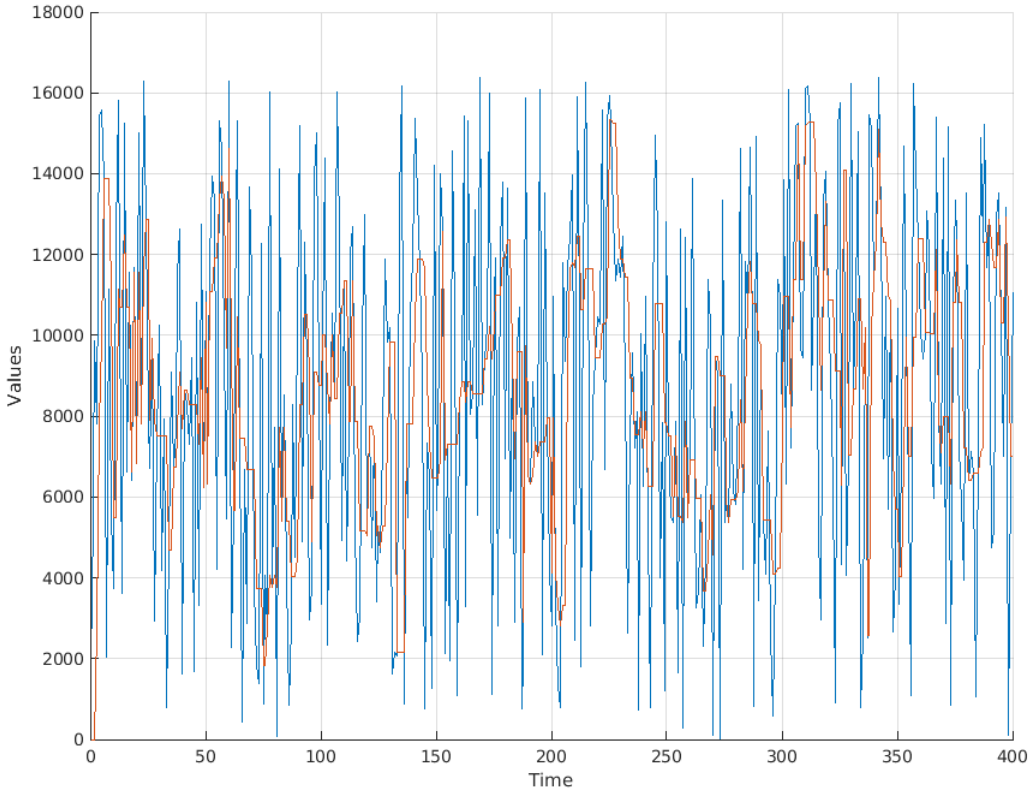


FIGURE 25 – Application du filtre médian sur les stimulus aléatoires (bleu : signal original, rouge : résultats après passage dans le filtre médian)

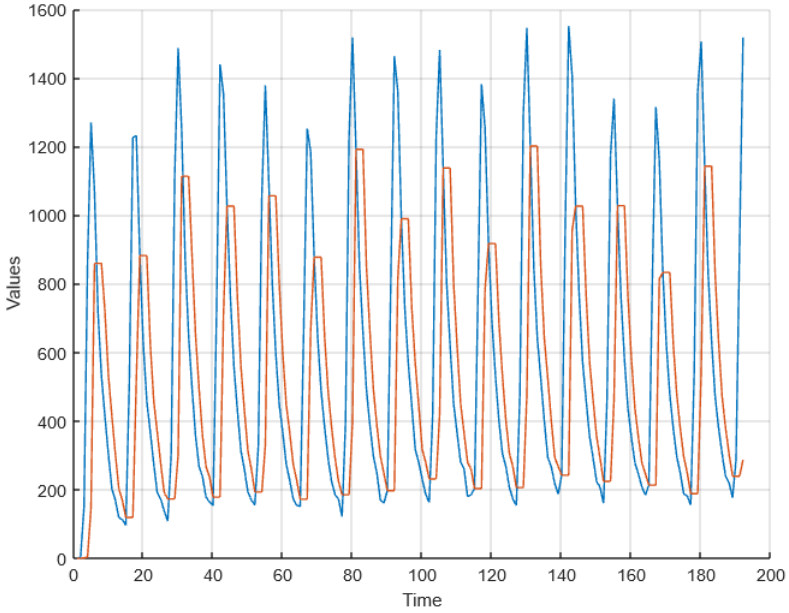


FIGURE 26 – Application du filtre médian sur les données réelles (bleu : signal original, rouge : résultats après passage dans le filtre médian)

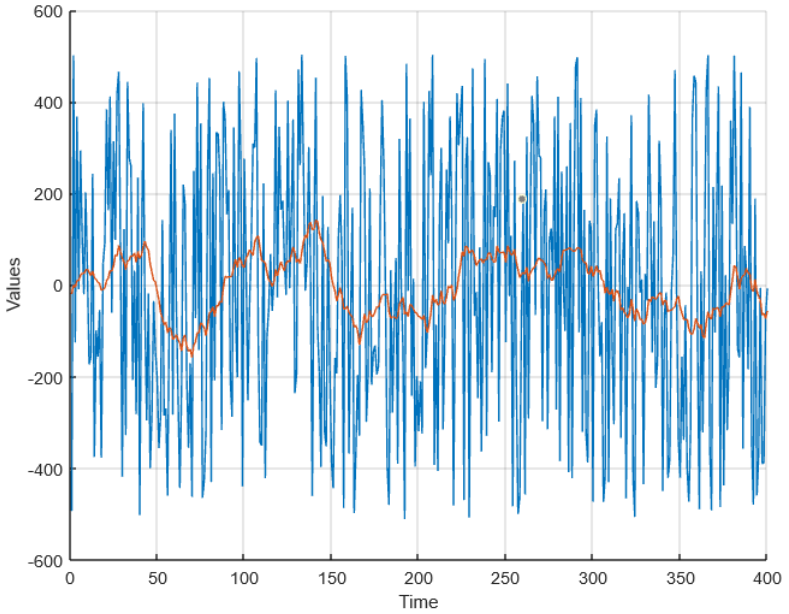


FIGURE 27 – Application de la moyenne glissante sur les stimulus aléatoires (bleu : signal original, rouge : résultats après passage dans le filtre)

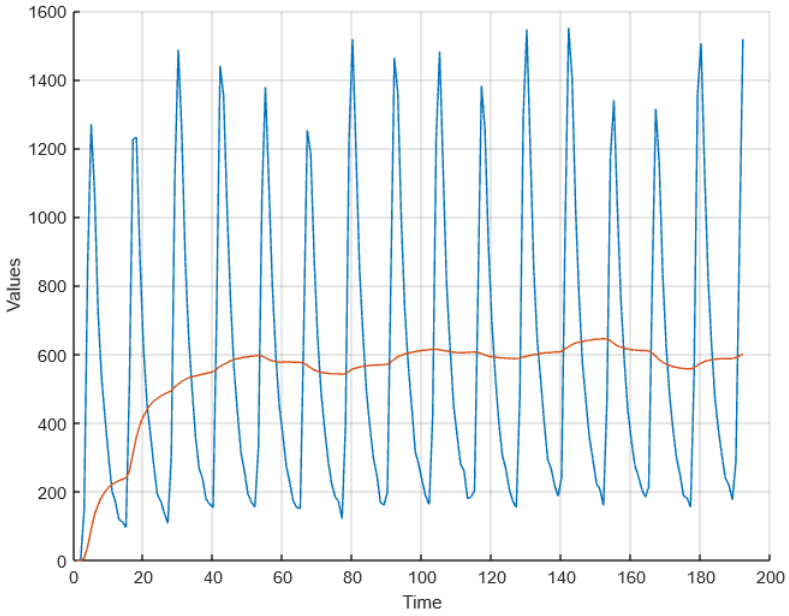


FIGURE 28 – Application de la moyenne glissante sur les données réelles (bleu : signal original, rouge : résultats après passage dans le filtre

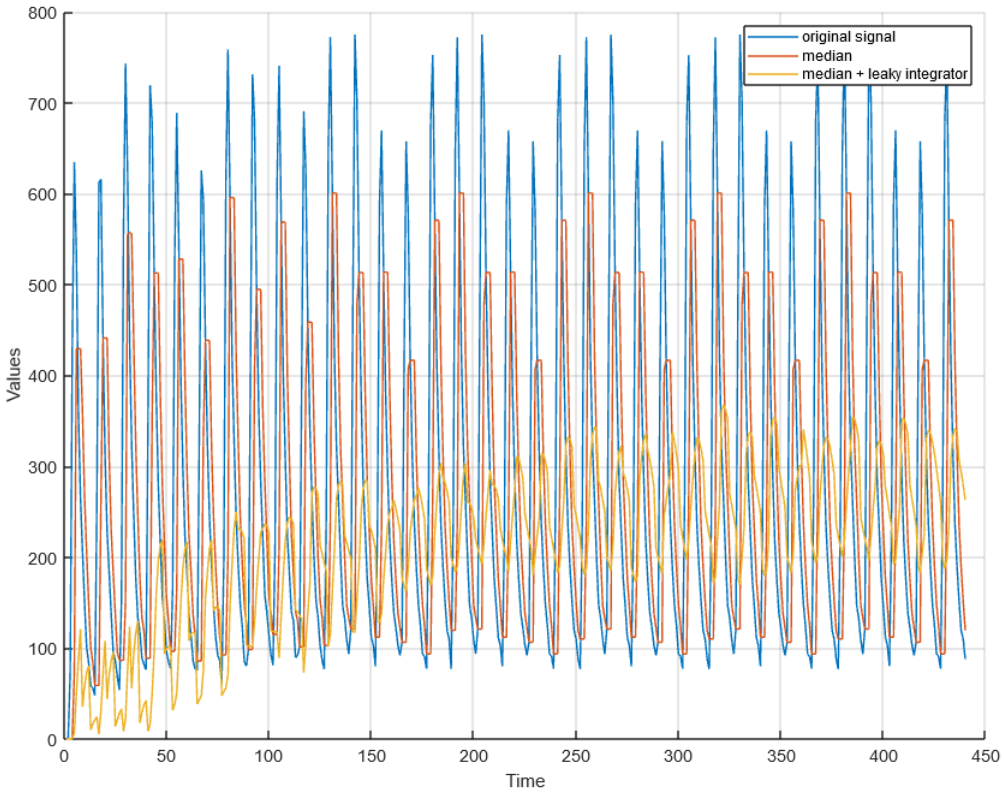


FIGURE 29 – Résultats pour le premier composant de l'application de test

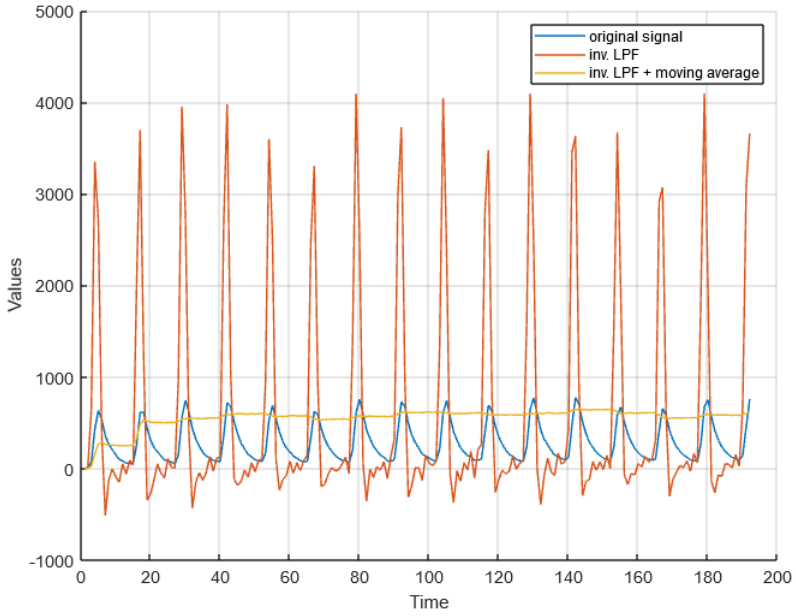


FIGURE 30 – Résultats pour le second composant de l'application de test

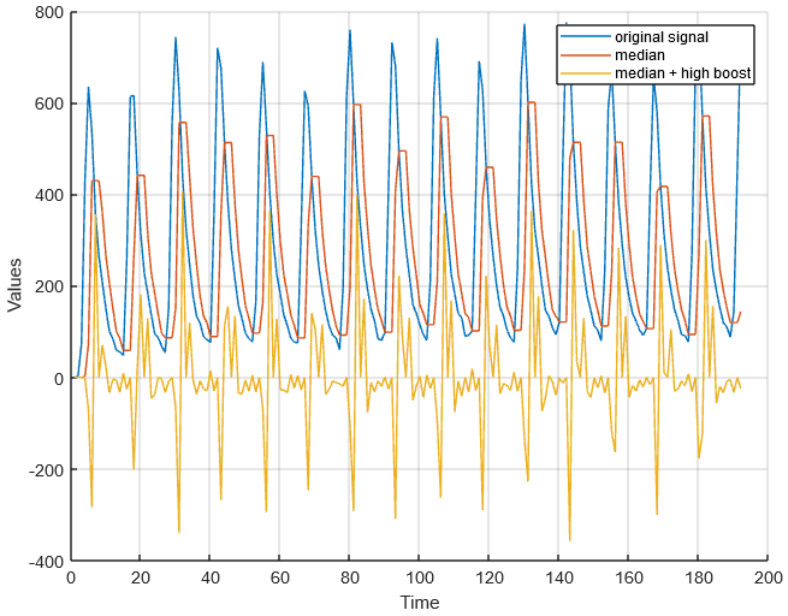


FIGURE 31 – Résultats pour le troisième composant de l'application de test

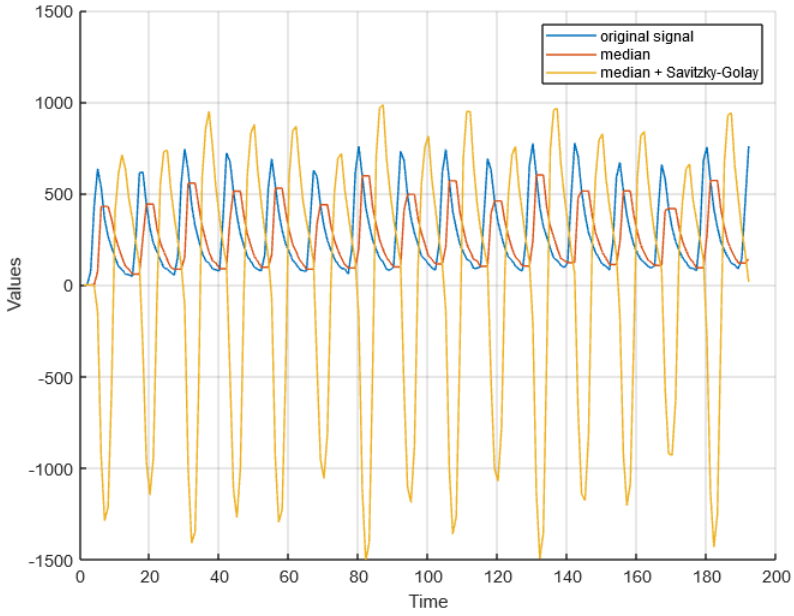


FIGURE 32 – Résultats pour le troisième composant de l'application de test

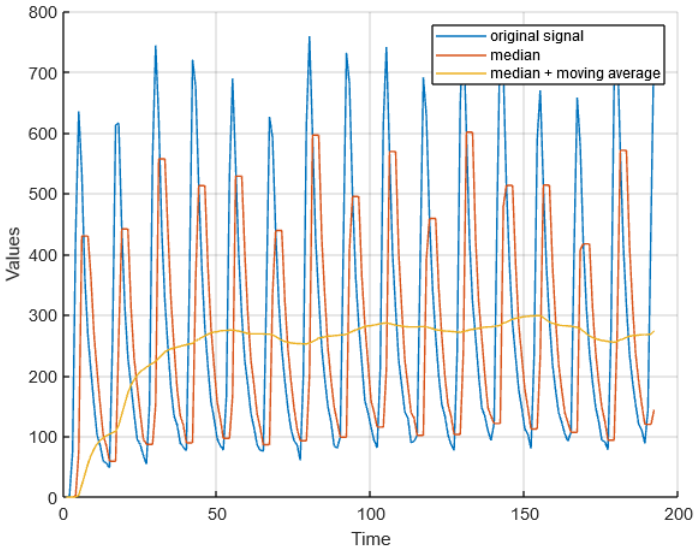


FIGURE 33 – Résultats pour chaînage médiane + moyenne

7 Synthèse

Afin de valider le design de filtrage, il était souhaitable d'intégrer ce dernier à une FPGA. Les circonstances ne permettant pas l'intégration dans la FPGA utilisée par le groupe d'instrumentation du CERN, à savoir la VFC-HD, une carte Cyclone V pourrait être utilisée pour cette ultime étape. Cela n'a pas pu être fait dans le temps imparti pour le projet, mais il est prévu de faire un essai avant la soutenance.

Bien que l'intégration à la VFC-HD n'ait pas été possible, la synthèse de l'ensemble des descriptions VHDL du module de filtrage s'est quand même effectuée avec une base de FPGA Arria V, modèle 5AGXMB1G4F40C4, à l'aide du logiciel Quartus Prime 19.1. Des synthèses intermédiaires ont été effectuées pour chacun des sous-systèmes afin d'analyser le rapport du logiciel sur les ressources utilisées pour chaque design.

Comme plusieurs versions ont parfois été implémentées pour chaque module, nous n'effectuerons pas de commentaires détaillés sur chacune d'entre elles ; nous nous intéresserons surtout à celles qui sont les plus susceptibles d'être utilisées. Dans la même optique, nous ne commenterons pas la synthèse de toutes les versions combinatoires des modules, à l'exception du *leaky integrator* afin d'effectuer des comparaisons et de tirer des conclusions qui seront également valables pour les autres filtres.

7.1 Leaky Integrator

Version 1, combinatoire

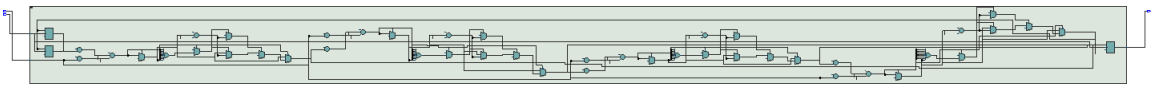


FIGURE 34 – Vue RTL du module *leaky integator* (première version, combinatoire)

Cette première synthèse nous génère une vue RTL (figure 34) où les quatre blocs d'opérations arithmétiques sont aisément visibles. Dans chacun de ces blocs, nous pouvons voir la logique ajoutée aux calculs de reformatage composée essentiellement de multiplexeurs et de portes OR.

Si nous changeons le paramètre `round_style` de la librairie `fixed_pkg` à `fixed_truncate`, cette logique supplémentaire est supprimée, comme le montre la figure 35. En effet, les bits en surplus sont simplement tronqués lors des opérations de recadrage des résultats. Il reste tout de même des multiplexeurs servant certainement à tester les dépassements de capacité, car nous n'avons pas modifié le paramètre associé aux *overflow*.

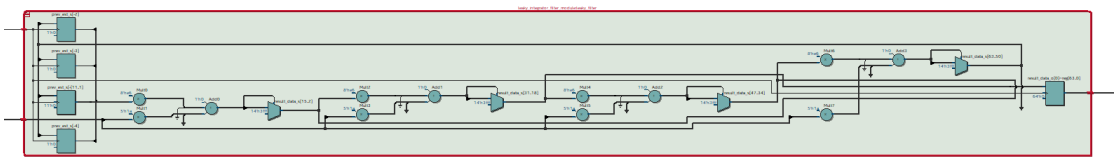


FIGURE 35 – Vue RTL du module *leaky integator* (première version combinatoire avec mode troncature)

Version 2, combinatoire

Sur la figure 36, nous pouvons retrouver nos 8 blocs multiplicateurs et nos 4 additionneurs venant de la formule du *leaky integrator*, et un additionneur supplémentaire qui calcule le facteur 1-λ qui, cette fois, n'est pas fixé à la compilation.

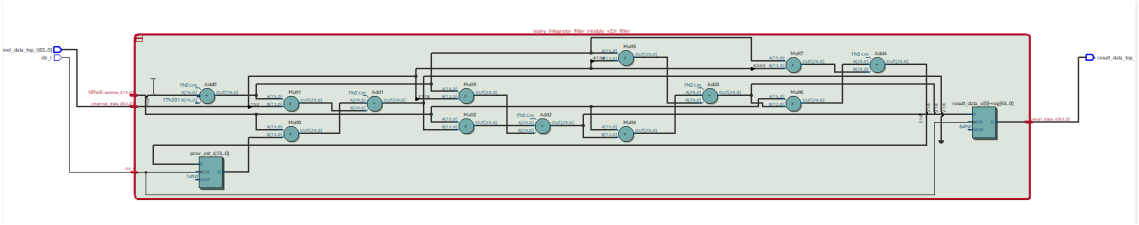


FIGURE 36 – Vue RTL du module *leaky integator* (deuxième version, combinatoire)

Version 2, pipeline

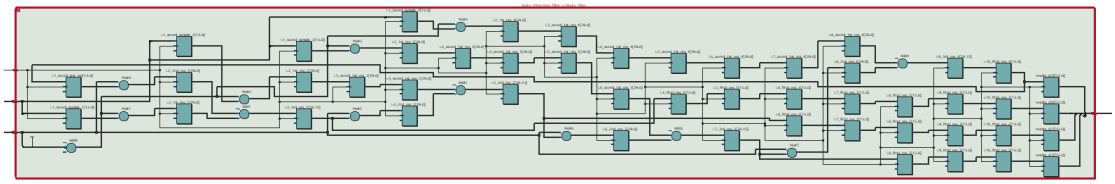


FIGURE 37 – Vue RTL du module *leaky integator* (deuxième version, pipeline)

La figure 37 nous montre la vue RTL pour la version 2 avec son adaptation en architecture pipelinée. Nous retrouvons les mêmes composants arithmétiques que pour la version combinatoire, avec cette fois les registres ajoutés pour la mise en place des étages de pipeline.

Quantité de logique

Le rapport de Quartus situé dans `Analysis and Synthesis > Resource Usage Summary` nous renseigne sur la quantité de logique utilisée après synthèse.

Le tableau 11 résume le rapport des versions combinatoires en présentant les éléments les plus importants.

La première information du tableau est spécifique à la famille de FPGA utilisée comme base pour la synthèse. Il s'agit du nombre d'*Adaptive Logic Modules* de la FPGA requis pour notre design, dont la quantité totale est de 136'880 pour la carte Arria V. Les deux versions du design en utilisent donc moins de 1%.

Nous pouvons constater que la seconde version n'utilise pas d'ALUTs. Cela n'est guère étonnant, car les blocs d'opérations arithmétiques sont entièrement contenus dans les blocs DSP. Rappelons que la première version effectue des opérations supplémentaires afin de reformater le résultat en gardant la meilleure précision, ce qui ajoute de la logique supplémentaire après les blocs DSP, visible dans la vue RTL.

Le tableau 12 présente les mêmes informations après la synthèse du module en version pipelinée.

Ressource	Version 1	Version 1' (trunc. mode)	Version 2
Logic Utilization (estimation, in ALMs)	67	35	35
Combinational ALUTs	124	56	0
Dedicated Logic Registers	70	70	70
Total DSP blocks	4	4	4

TABLE 11 – Rapport Quartus : logique utilisée pour le filtre *leaky integrator* (combinatoire)

Ressource	Version 1	Version 2
Logic Utilization (estimation, in ALMs)	252	220
Combinational ALUTs	236	130
Dedicated Logic Registers	446	416
Total DSP blocks	7	7

TABLE 12 – Rapport Quartus : logique utilisée pour le filtre *leaky integrator* (pipeline)

Par rapport aux versions combinatoires, nous pouvons vite remarquer que bien plus de logique est utilisée pour les versions pipelinées, à tous les niveaux ; en effet, de nombreux registres intermédiaires nécessaires à la structure en pipeline s'ajoutent aux ressources logiques utilisées par le design. Néanmoins, cela a permis de faire monter la fréquence maximale de fonctionnement en flèche, qui n'était guère plus de 30 MHz pour les versions combinatoires.

Fréquence de fonctionnement

L'outil Timing Analyzer de Quartus nous informe que nos deux systèmes, en versions pipelinées, peuvent fonctionner à une fréquence de 125 MHz comme le voulaient les spécifications du module de filtrage. La fréquence maximale de fonctionnement est de **151.65** MHz pour la première version et de **235.36** MHz pour la seconde.

Lorsque l'on revient sur les vues RTL des versions combinatoires, il est clairement visible que les données doivent parcourir un chemin combinatoire plutôt long, ce qui compromet les exigences de *timing*. Avec le Timing Analyzer et sa macro *Report Top Failing Paths*, nous pouvons voir les principaux chemins défaillants qui profanent les spécifications d'implémentation du module : dans notre cas, il s'agit du chemin parcouru par signaux d'entrée jusqu'aux signaux de sortie, où les données ont un délai de propagation d'environ 15 ns, bien au-dessus des 8 ns secondes que prennent une période d'horloge. Cela nous alerte qu'une restructuration du module est impérative.

7.2 Convolutions

Deux termes

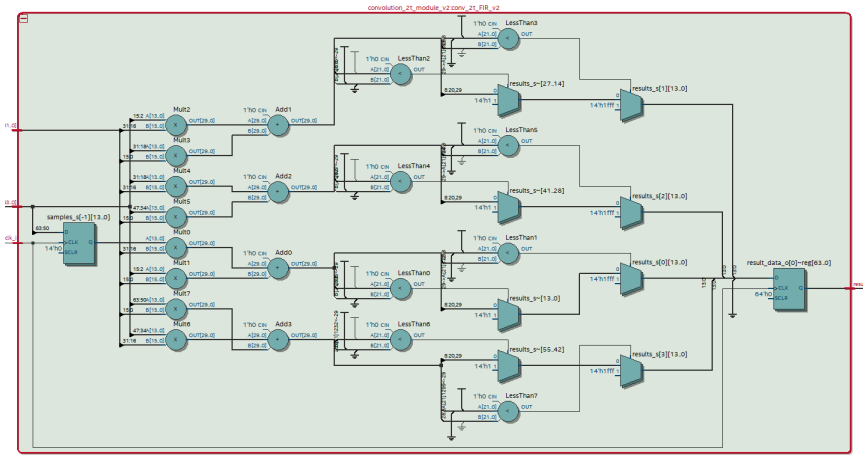


FIGURE 38 – Vue RTL du module de convolution à deux termes (deuxième version, combinatoire)

Cette première vue RTL, figure 38, présente la synthèse du module de convolution à deux termes, dans sa version combinatoire et avec les coefficients en entrée. Sur la gauche, nous pouvons noter la présence de nos huit blocs multiplicateurs avec les coefficients, puis les quatre additions. Viennent ensuite les opérations de reformatage du résultat où apparaissent quelques multiplexeurs et comparateurs, puis le test avec les valeurs maximales et minimales représentables sur la partie droite du bloc.

Nous pouvons voir que le chemin que doivent traverser les signaux d'entrée afin de subir le traitement du filtre est peuplé d'obstacles logiques, ce qui aura pour conséquence d'avoir une fréquence maximale de fonctionnement plutôt basse.

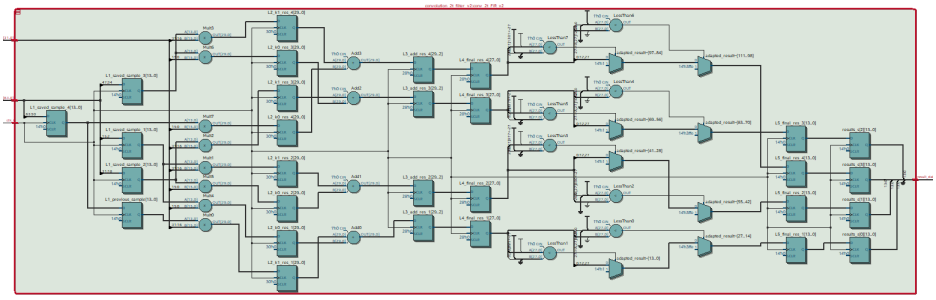


FIGURE 39 – Vue RTL du module de convolution à deux termes (deuxième version, avec pipeline)

Cette seconde vue RTL, figure 39 nous présente la synthèse de la version pipelinée. Nous pouvons retrouver les mêmes opérations, avec cette fois-ci nos registres intermédiaires pour le découpage des blocs combinatoires en niveaux de pipeline.

Sur la figure 40, nous pouvons clairement retrouver une structure similaire au schéma du découpage

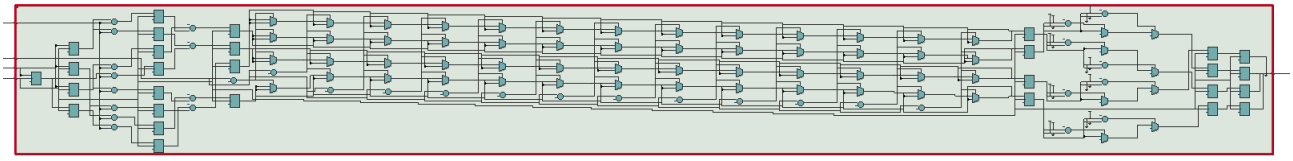


FIGURE 40 – Vue RTL du module de convolution à deux termes (troisième version)

en niveaux de pipeline du module de convolution à deux termes. Par rapport à la deuxième version, un arsenal de multiplexeurs et comparateurs se sont invités, régis par le nombre de bits de fraction des coefficients qui est variable dans cette version.

Rappelons ici que la troisième version des modules de convolution prend en entrée les coefficients des noyaux de convolution et le nombre de bits de fraction utilisés pour représenter ces derniers.

Le tableau 13 présente la quantité de logique nécessaire aux trois versions du module de convolution à deux termes, en implémentation pipelinée.

Ressource	Version 1	Version 2	Version 3
Logic Utilization (estimation, in ALMs)	221	269	399
Combinational ALUTs	348	72	473
Dedicated Logic Registers	389	518	542
Total DSP blocks	0	4	4

TABLE 13 – Rapport Quartus : logique utilisée pour la convolution à deux termes (versions pipelinées)

Ici, nous pouvons observer que la version 3 utilise le plus de ressources matérielles, compte tenu de sa modularité. Dans les deux autres versions, rappelons que certains paramètres sont constants, c'est pourquoi moins de logique est utilisée.

Nous pouvons également voir que la version 1 n'a pas pu combiner les calculs dans des blocs DSP, tandis que les autres ont pu en utiliser 4. Cela a permis de réduire le nombre d'ALUTs de la version 2, qui reste très élevé pour la version 3.

Trois termes, version 3 avec pipeline

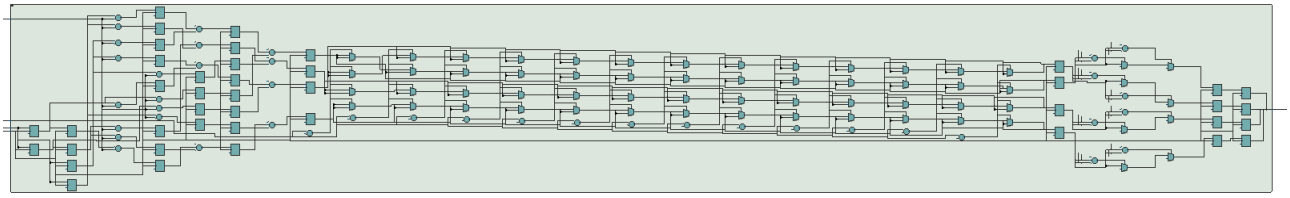


FIGURE 41 – Vue RTL du module de convolution à trois termes (troisième version)

La figure 41 présente la vue RTL obtenue pour le module de convolution à trois termes. Tout comme

pour celle à deux termes, nous pouvons retrouver notre structure de pipeline avec nos 12 mutliplicateurs et 8 additionneurs, puis le système de redimensionnement du résultat.

Ressource	Version 1	Version 2	Version 3
Logic Utilization (estimation, in ALMs)	257	462	586
Combinational ALUTs	220	192	593
Dedicated Logic Registers	511	900	916
Total DSP blocks	0	8	8

TABLE 14 – Rapport Quartus : logique utilisée pour la convolution à trois termes (versions pipelinées)

Concernant le rapport du tableau 14, les mêmes conclusions peuvent être faites par rapport à la quantité de logique utilisée pour la convolution à deux termes, avec une différence plus marquée ; la version 3 utilise presque deux fois plus de ressources que la version 1.

Cinq termes, version 3 avec pipeline

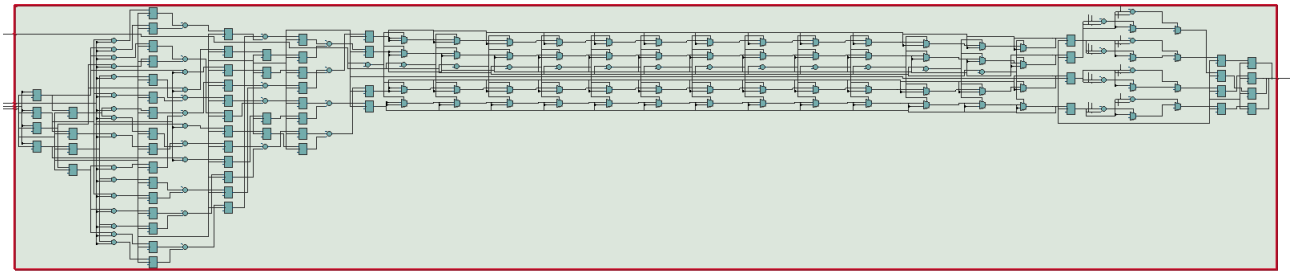


FIGURE 42 – Vue RTL du module de convolution à cinq termes (troisième version)

Dans la même optique que pour les deux vues RTL présentées ci-dessus, la figure 42 montre les 20 multiplicateurs et les 16 additionneurs dont le module de convolution à cinq termes a besoin, puis également le système de redimensionnement du résultat.

Ressource	Version 1	Version 2	Version 3
Logic Utilization (estimation, in ALMs)	342	716	840
Combinational ALUTs	274	312	713
Dedicated Logic Registers	680	1408	1424
Total DSP blocks	0	12	12

TABLE 15 – Rapport Quartus : logique utilisée pour la convolution à cinq termes (versions pipelinées)

Comme pour les ressources utilisées par le module de convolution à trois termes, notre module de convolution à cinq termes en utilise de plus en plus selon la version du module comme le montrent les valeurs du tableau 15.

Fréquence de fonctionnement

Le tableau 16 présente les fréquences maximales de fonctionnement des trois versions des trois modules de convolution implémentés. Nous pouvons constater que toutes passent bien le seuil des 125 MHz requis.

Il est normal que les versions 1 soient plus limitées compte tenu de la logique ajoutée par la librairie `fixed_pkg` pour le reformatage des résultats et l'absence de blocs DSP, pour les convolutions à deux et trois termes ; pour celle à cinq termes en revanche, il semblerait que le synthétiseur ait trouvé des optimisations permettant d'avoir une fréquence maximale très élevée.

Nb. termes	Version	FMax [MHz]
2	1	184.06
2	2	310.2
2	3	225.19
3	1	185.72
3	2	299.85
3	3	221.29
5	1	310.95
5	2	250.1
5	3	238.56

TABLE 16 – Fréquences maximales de fonctionnement des modules de convolution

7.3 Médiane

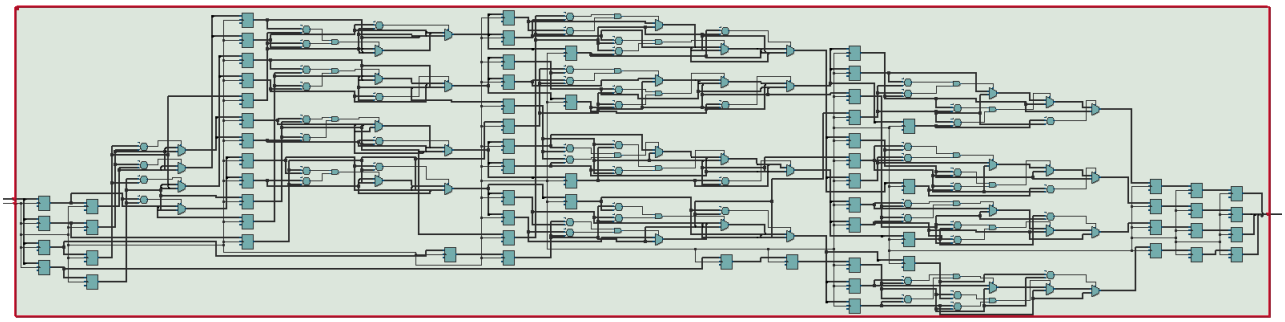


FIGURE 43 – Vue RTL du module du filtre médian

Sur la vue RTL du filtre médian figure 43, nous pouvons retrouver tous les comparateurs qui servent à trier les cinq échantillons pris en compte dans la détermination de la médiane.

Quantité de logique

La quantité de logique utilisée par le filtre médian, tableau 17, correspond plutôt à nos attentes, car il y a un grand nombre de systèmes comparateurs à mettre en place. Il est normal qu'aucun bloc DSP ne soit utilisé, car aucune opération arithmétique n'est effectuée.

Ressource	Valeur
Logic Utilization (estimation, in ALMs)	709
Combinational ALUTs	788
Dedicated Logic Registers	840
Total DSP blocks	0

TABLE 17 – Rapport Quartus : logique utilisée pour la médiane sur 5 échantillons

Fréquence de fonctionnement

Le module de filtre médian sur cinq échantillons peut fonctionner à une fréquence maximale de **177.72** MHz ; le dernier étage de tri des cinq valeurs nécessite beaucoup de logique et par conséquent un long chemin combinatoire, ce qui explique cette valeur.

7.4 Moyenne glissante

La vue RTL de notre dernier filtre, figure 44, est sans doute la plus visuellement dense. Néanmoins, elle correspond à nos attentes : nous pouvons retrouver les niveaux de pipeline établis, avec d'abord les multiplicateurs permettant d'appliquer le coefficient aux 25 échantillons, puis les nombreux additionneurs qui combinent tous ces premiers résultats pour former le résultat de sortie.

Quantité de logique

La quantité de logique utilisée par le module du filtre médian, présentée au tableau 18, est plutôt élevée ; cela n'est guère surprenant si l'on revient rapidement sur sa vue RTL. Notons toutefois qu'il y a tout de même moins de *Dedicated Logic Registers* qui sont utilisés que pour la médiane, qui n'est pourtant que sur cinq échantillons ; le grand nombre de comparaisons effectuée dans le filtre médian est donc plus coûteux pour cette ressource que les nombreuses opérations effectuées pour le calcul de la moyenne glissante. Le nombre d'ALUTs et d'ALMs, par contre, est deux fois plus élevé dans le second filtre.

Fréquence de fonctionnement

Ce dernier module de filtrage possède une fréquence maximale de fonctionnement de **215.34** MHz.

Ressource	Valeur
Logic Utilization (estimation, in ALMs)	1490
Combinational ALUTs	1871
Dedicated Logic Registers	309
Total DSP blocks	28

TABLE 18 – Rapport Quartus : logique utilisée pour la moyenne glissante en version pipeline

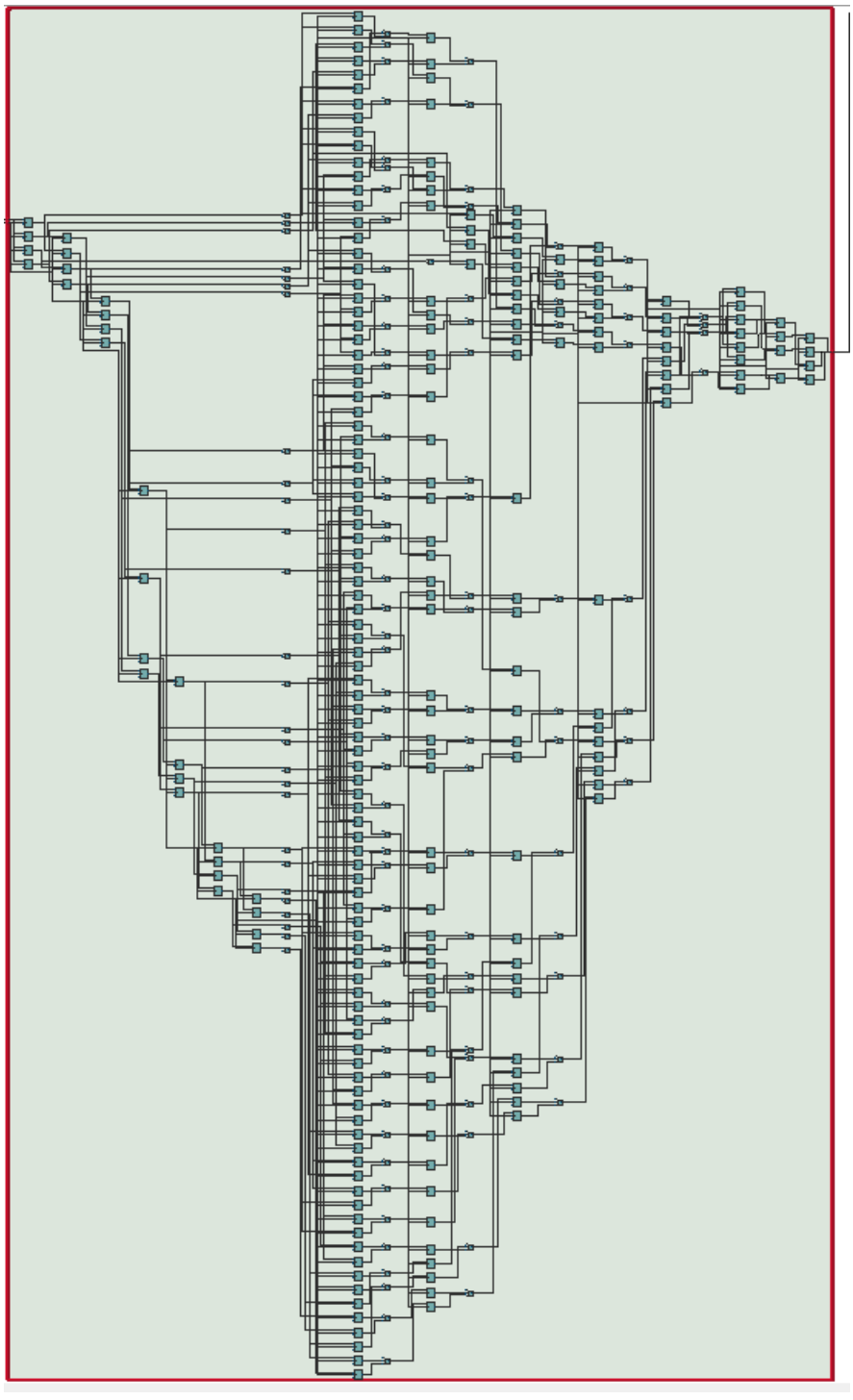


FIGURE 44 – Vue RTL du module de la moyenne glissante sur 25 échantillons

8 Conclusion

Pour conclure ce projet de Bachelor, nous pouvons faire le bilan du travail accompli jusqu'ici en nous intéressant au respect du planning et du cahier des charges, puis aux problèmes rencontrés durant l'ensemble du projet. Finalement, une conclusion personnelle sera établie.

Afin de prendre en main l'environnement de développement, l'implémentation et la vérification d'un premier sous-système de filtrage assez simple a été effectuée pendant le travail à temps partiel, selon la suggestion du professeur responsable. Ce premier filtre, le *leaky integrator*, m'a permis de me familiariser avec le SystemVerilog, langage que je n'avais jamais utilisé auparavant, et d'améliorer mes connaissances en design de systèmes numériques. Le suivi du cours Conception de Systèmes Numériques sur FPGA m'a été très bénéfique, j'ai notamment découvert le monde de la vérification de façon pédagogique.

Dans la première version du planning établie, le filtre *leaky integrator* n'apparaît pas car il n'était pas prévu de l'implémenter dans le cahier des charges ; cela produit un grand décalage par rapport au temps consacré aux autres filtres prévus initialement, et m'a quelque peu préoccupée dans un premier temps. Néanmoins, toute la structure de banc de test étant mise en place, cela a rendu les prochaines étapes de vérification bien plus rapides. En effet, les quelques difficultés d'implémentation rencontrées durant le développement du premier filtre n'étaient plus à résoudre par la suite, ou l'ont été bien plus rapidement.

Durant le travail à plein temps, j'ai pu me concentrer sur l'implémentation des modules de filtrage restants en suivant le modèle de développement mis en place pour le *leaky integrator*. Cependant, un mois avant la date de remise, j'ai réalisé que la fréquence de fonctionnement maximale de mon système n'atteignait pas les 125 MHz des spécifications fonctionnelles. Lors des premières synthèses, une mauvaise manipulation dans le logiciel n'avait pas connecté correctement l'horloge de mon système ; j'ai dû revoir la conception de l'ensemble de mes filtres pour adapter leur architecture comportementale en une architecture en pipeline, ce qui m'a pris plusieurs heures de développement et a engendré un petit retard sur mon planning de référence. Ceci dit, c'était une étape fort instructive qui m'a permis d'apprendre à concevoir un système séquentiel un peu plus complexe que ceux que j'ai eu l'occasion de mettre en place durant les travaux pratiques des derniers semestres.

En raison de l'amoncellement de petits retards pris sur le planning d'origine, le filtre médian et la moyenne glissante ont dû être implémentés et vérifiés rapidement pendant les trois dernières semaines du projet. De plus, le développement de l'application de test n'avait pas été évoqué dans la planification de base ; cela a empiété sur la semaine consacrée à l'intégration sur la FPGA.

En effet, il était prévu de conclure le projet en programmant une carte Cyclone V avec le design de l'application de test, ceci afin de valider le fonctionnement du système qui n'a été testé qu'en simulation. Comme le module de filtrage est destiné à s'intégrer à un design existant en cours de développement au CERN, il aurait fallu effectuer des adaptations supplémentaires en vue de l'intégration à la FPGA et du système d'envoi de données au module de filtrage. Cela n'a malheureusement pas pu être mis en oeuvre par manque de temps, néanmoins il est prévu d'essayer d'effectuer cette intégration au début du mois de septembre pour en présenter les résultats pendant la soutenance.

La période de confinement, qui a eu lieu pendant toute la première partie du projet, ne m'a pas pénalisée. J'ai pu mettre en place tout l'environnement de travail sur mon ordinateur personnel sans accroc, et mon professeur et moi avons quand même pu effectuer nos séances hebdomadaires à distance. Nous avons également eu l'occasion de nous entretenir avec M. Emery du CERN à plusieurs reprises, pour discuter du projet et présenter quelques démonstrations des résultats obtenus.

Au début du travail, il m'a été un peu difficile de me plonger dans le vif du sujet, car le monde du traitement de signal ne m'est pas vraiment familier, et le monde des accélérateurs de particules l'est encore moins. Fort heureusement, j'ai pu trouver de nombreuses ressources en ligne pour combler certaines de mes lacunes et pouvoir apprivoiser le sujet en surface, notamment pour transcrire les formules mathématiques des filtres en code adapté pour les descriptions synthétisables.

Si, au tout début du projet, le parcours du travail à accomplir m'a paru obscur et semé d'embûches, aujourd'hui je suis satisfaite de mon implémentation et des résultats que j'ai pu obtenir avec l'ensemble de mes modules de filtrage. Les objectifs qui figurent dans le cahier des charges ont été atteints : l'algorithme de traitement est fonctionnel avec des données réelles.

Le design FPGA étant une matière que j'ai découvert il y a moins d'un an, j'ai pu renforcer mes compétences dans ce domaine et je souhaite continuer à le faire durant ma carrière professionnelle. Pouvoir apporter ma contribution, aussi modeste soit-elle, à un projet du CERN aussi prestigieux que le LHC à Haute Luminosité a été une solide source de motivation tout au long du développement. Cela m'a confortée dans mon souhait de travailler dans le domaine de la recherche scientifique, et j'espère grandement en avoir l'occasion durant mon futur parcours professionnel.

Table des figures

Liste des tableaux

Planning

Deux plannings de Gantt sont fournis en annexe. Le premier est le planning original, établi au début du travail à la fin du mois de mars. Le second montre le travail qui a été effectué dans la réalité jusqu'au vendredi 14 août qui marque la fin de ce travail de diplôme.

Ils sont générés avec le logiciel Gantt Project.

Références

[1] Jose Luis Sirvent Blasco. *Beam secondary shower acquisition design for the CERN high accuracy wire scanner*. 2018

[2] Site officiel du CERN
`https://home.cern`

[3] David Belohrad. *BWS Acquisition - Reflection on state of matters*. 2020

[4] Paolo Prandoni et Martin Vetterli. *Cours EPFL : Traitement numérique du signal - Module 4.2*. 2020

[5] Clay S. Turner. *Leaky Integrator*. 2008

[6] Yann Thoma. *Cours HEIG-VD : Slides des unités d'enseignement CSF et VSN*. 2020

[7] Fixed point numbers in Verilog, Will Green, 2018
`https://timetoexplore.net/blog/fixed-point-numbers-in-verilog`

[8] David Bishop. *Fixed point package user's guide*. 2008

[9] Documentation MATLAB : Convolution and polynomial multiplication
`https://ch.mathworks.com/help/matlab/ref/conv.html`

[10] Wikipédia : Algorithme de Savitzky-Golay
`https://fr.wikipedia.org/wiki/Algorithme_de_Savitzky-Golay`

[11] Wiki VME FMC Carrier (VFC-HD)
`https://ohwr.org/project/vfc-hd/wikis/home`

A Plans de vérification

A.1 Leaky Integrator

# Test (v1) (v2)	Fonctionnalité	Détails	Priorité	Séquence de test	Critère de vérification
1	Valeurs aléatoires, M par défaut, **1 bit de fraction**	Valeurs échantillons : aléatoires Nb de bits fraction : 1 M : 10	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
2 (2, 3, 4, 5)	Valeurs aléatoires, M par défaut, **petit nombre de bits de fraction**	Valeurs échantillons : aléatoires Nb de bits fraction : 2 à 5 M = 10	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
3 (6, 7, 8, 9, 10, 11, 12)	Valeurs aléatoires, M par défaut, **grand nombre de bits de fraction**	Valeurs échantillons : aléatoires Nb de bits fraction : 6 à 12 M = 10	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
4 (13)	Valeurs aléatoires, M par défaut, **13 bits de fraction**	Valeurs échantillons : aléatoires Nb de bits fraction : 13 M = 10	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
5 (14, 15, 16, 17)	Valeurs aléatoires, **M petit**, moyen nombre de bits de fraction	Valeurs échantillons : aléatoires Nb de bits fraction : 5 M = 1 à 4	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
6 (18, 19, 20)	Valeurs aléatoires, **M moyen**, moyen nombre de bits de fraction	Valeurs échantillons : aléatoires Nb de bits fraction : 5 M = 5 à 7	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
7 (21, 22)	Valeurs aléatoires, **M grand**, moyen nombre de bits de fraction	Valeurs échantillons : aléatoires Nb de bits fraction : 5 M = 8 et 9	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
8 (23)	Valeurs aléatoires, **M = 11**, moyen nombre de bits de fraction	Valeurs échantillons : aléatoires Nb de bits fraction : 5 M = 11	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
9 (24)	Valeurs aléatoires, **M = 20**, moyen nombre de bits de fraction	Valeurs échantillons : aléatoires Nb de bits fraction : 5 M = 20	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
10 (25)	Valeurs aléatoires, **M = 100**, moyen nombre de bits de fraction	Valeurs échantillons : aléatoires Nb de bits fraction : 5 M = 100	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
11 (26)	**Valeurs minimales**, M = 10, moyen nombre de bits de fraction	Valeurs échantillons : tous bits à 0 Nb de bits fraction : 5 M = 10	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
12 (27)	**Valeurs maximales**, M = 10, moyen nombre de bits de fraction	Valeurs échantillons : tout bits à 1 Nb de bits fraction : 5 M = 10	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
13 (28)	**Valeur constante**, M = 10, moyen nombre de bits de fraction	Valeurs échantillons : la même pour les 4 Nb de bits fraction : 5 M = 10	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
14 (29)	**Sinus bruité**, M = 10, grand nombre de bits de fraction	Valeurs échantillons : génération d'un sinus bruité Nb de bits fraction : 8 M = 10	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects
15 (30)	**Impulsions**, M = 10, petit nombre de bits de fraction	Valeurs échantillons : impulsions réelles Nb de bits fraction : 1 M = 10	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats des estimations corrects

A.2 Convolution à deux termes

# Test	Fonctionnalité	Détails	Priorité	Séquence de test	Critère de vérification
1	Valeurs aléatoires, kernel invLPF, **1 bit de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [4.5, -3.5] (Q15.1) Nb de bits fraction : 1	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
2	Valeurs aléatoires, kernel invLPF, **petit nombre de bits de fraction**	Valeurs échantillons : aléatoire Valeurs kernel : [4.5, -3.5] (Q15.1) Nb de bits fraction : 2 à 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
3	Valeurs aléatoires, kernel invLPF, **grand nombre de bits de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [4.5, -3.5] (QQ15.1) Nb de bits fraction : 6 à 12	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
4	Valeurs aléatoires, kernel invLPF, **13 bits de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [4.5, -3.5] (Q15.1) Nb de bits fraction : 13	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
5	Valeurs aléatoires, **kernel smooth invLPF**, moyen nombre de bits de fraction	Valeurs échantillons : aléatoires Valeurs kernel : [0.5, 0.5] (Q15.1) Nb de bits fraction : 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
6	**Valeurs minimales**, kernel invLPF, moyen nombre de bits de fraction	Valeurs échantillons : minimum représentable Valeurs kernel : [4.5, -3.5] (Q15.1) Nb de bits fraction : 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
7	**Valeurs maximales**, kernel invLPF, moyen nombre de bits de fraction	Valeurs échantillons : maximum représentable Valeurs kernel : [4.5, -3.5] (Q15.1) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
8	**Valeur constante**, kernel invLPF, moyen nombre de bits de fraction	Valeurs échantillons : la même pour les 4 Valeurs kernel : [4.5, -3.5] (Q15.1) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
9	**Valeur 0**, kernel invLPF, moyen nombre de bits de fraction	Valeurs échantillons : tous bits à '0' Valeurs kernel : [4.5, -3.5] (Q15.1) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
10	**Valeurs minimales**, kernel smooth, moyen nombre de bits de fraction	Valeurs échantillons : minimum représentable Valeurs kernel : [0.5, 0.5] (Q15.1) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
11	**Valeurs maximales**, kernel smooth, moyen nombre de bits de fraction	Valeurs échantillons : maximum représentable Valeurs kernel : [0.5, 0.5] (Q15.1) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
12	**Valeur constante**, kernel smooth, moyen nombre de bits de fraction	Valeurs échantillons : la même pour les 4 Valeurs kernel : [0.5, 0.5] (Q15.1) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
13	**Valeur 0**, kernel smooth, moyen nombre de bits de fraction	Valeurs échantillons : tous bits à '0' Valeurs kernel : [0.5, 0.5] (Q15.1) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
14	**Impulsions**, kernel FIR, moyen nombre de bits de fraction	Valeurs échantillons : données représentant des impulsions Valeurs kernel : [4.5, -3.5] (Q15.1) Nb de bits fraction : 1	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
15	**Impulsions**, kernel smooth, moyen nombre de bits de fraction	Valeurs échantillons : données représentant des impulsions Valeurs kernel : [0.5, 0.5] (Q15.1) Nb de bits fraction : 1	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects

A.3 Convolution à trois termes

# Test	Fonctionnalité	Détails	Priorité	Séquence de test	Critère de vérification
1	Valeurs aléatoires, kernel HBF, **1 bit de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 1	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
2	Valeurs aléatoires, kernel HBF, **petit nombre de bits de fraction**	Valeurs échantillons : aléatoire Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 2 à 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
3	Valeurs aléatoires, kernel HBF, **grand nombre de bits de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 6 à 12	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
4	Valeurs aléatoires, kernel HBF, **13 bits de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 13	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
5	**Valeurs minimales**, kernel HBF, moyen nombre de bits de fraction	Valeurs échantillons : minimum représentable Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
6	**Valeurs maximales**, kernel HBF, moyen nombre de bits de fraction	Valeurs échantillons : maximum représentable Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
7	**Valeur constante**, kernel HBF, moyen nombre de bits de fraction	Valeurs échantillons : la même pour les 4 Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
8	**Valeur 0**, kernel HBF, moyen nombre de bits de fraction	Valeurs échantillons : tous bits à '0' Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
9	**Impulsions**, kernel FIR, moyen nombre de bits de fraction	Valeurs échantillons : données représentant des impulsions Valeurs kernel : [-1, 2, -1] (Q16.0) Nb de bits fraction : 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects

A.4 Convolution à cinq termes

# Test	Fonctionnalité	Détails	Priorité	Séquence de test	Critère de vérification
1	Valeurs aléatoires, kernel SG, **1 bit de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 1	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
2	Valeurs aléatoires, kernel SG, **petit nombre de bits de fraction**	Valeurs échantillons : aléatoire Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 2 à 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
3	Valeurs aléatoires, kernel SG, **grand nombre de bits de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 6 à 12	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
4	Valeurs aléatoires, kernel SG, **13 bits de fraction**	Valeurs échantillons : aléatoires Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 13	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
5	**Valeurs minimales**, kernel SG, moyen nombre de bits de fraction	Valeurs échantillons : minimum représentable Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
6	**Valeurs maximales**, kernel SG, moyen nombre de bits de fraction	Valeurs échantillons : maximum représentable Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
7	**Valeur constante**, kernel SG, moyen nombre de bits de fraction	Valeurs échantillons : la même pour les 4 Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
8	**Valeur 0**, kernel SG, moyen nombre de bits de fraction	Valeurs échantillons : tous bits à '0' Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 5	3	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
9	**Impulsions**, kernel SG, moyen nombre de bits de fraction	Valeurs échantillons : données représentant des impulsions Valeurs kernel : [-2, -1, 0, 1, 2] (Q16.0) Nb de bits fraction : 5	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects

A.5 Médiane

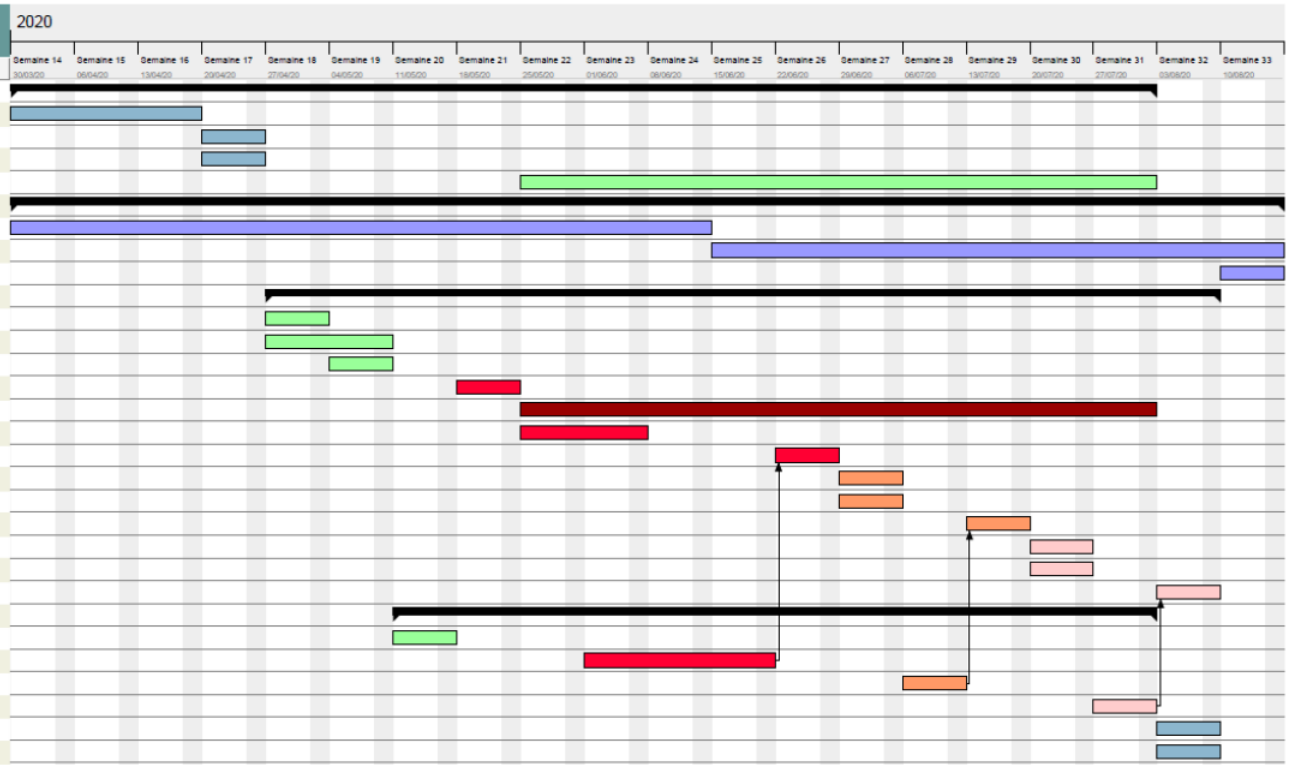
# Test	Fonctionnalité	Détails	Priorité	Séquence de test	Critère de vérification
1	**Valeurs aléatoires**	Valeurs échantillons : aléatoires	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
2	**Valeurs minimales**	Valeurs échantillons : minimum représentable	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
3	**Valeurs maximales**	Valeurs échantillons : maximum représentable	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
4	**Valeur constante**	Valeurs échantillons : la même pour les 4	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
5	**Impulsions**	Valeurs échantillons : données représentant des impulsions	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects

A.6 Moyenne glissante

# Test	Fonctionnalité	Détails	Priorité	Séquence de test	Critère de vérification
1	**Valeurs aléatoires,** chaque nombre de bits de fraction	Valeurs échantillons : aléatoires Bits de fraction : de 1 à 13	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
2	**Valeurs minimales**	Valeurs échantillons : minimum représentable Bits de fraction : 4	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
3	**Valeurs maximales**	Valeurs échantillons : maximum représentable Bits de fraction : 4	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
4	**Valeur constante**	Valeurs échantillons : la même pour les 4 Bits de fraction : 4	2	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects
5	**Impulsions**	Valeurs échantillons : données représentant des impulsions Bits de fraction : 1	1	Instanciation du DUT avec les bons paramètres Envoi des données de stimulus sur 400 itérations	Résultats corrects

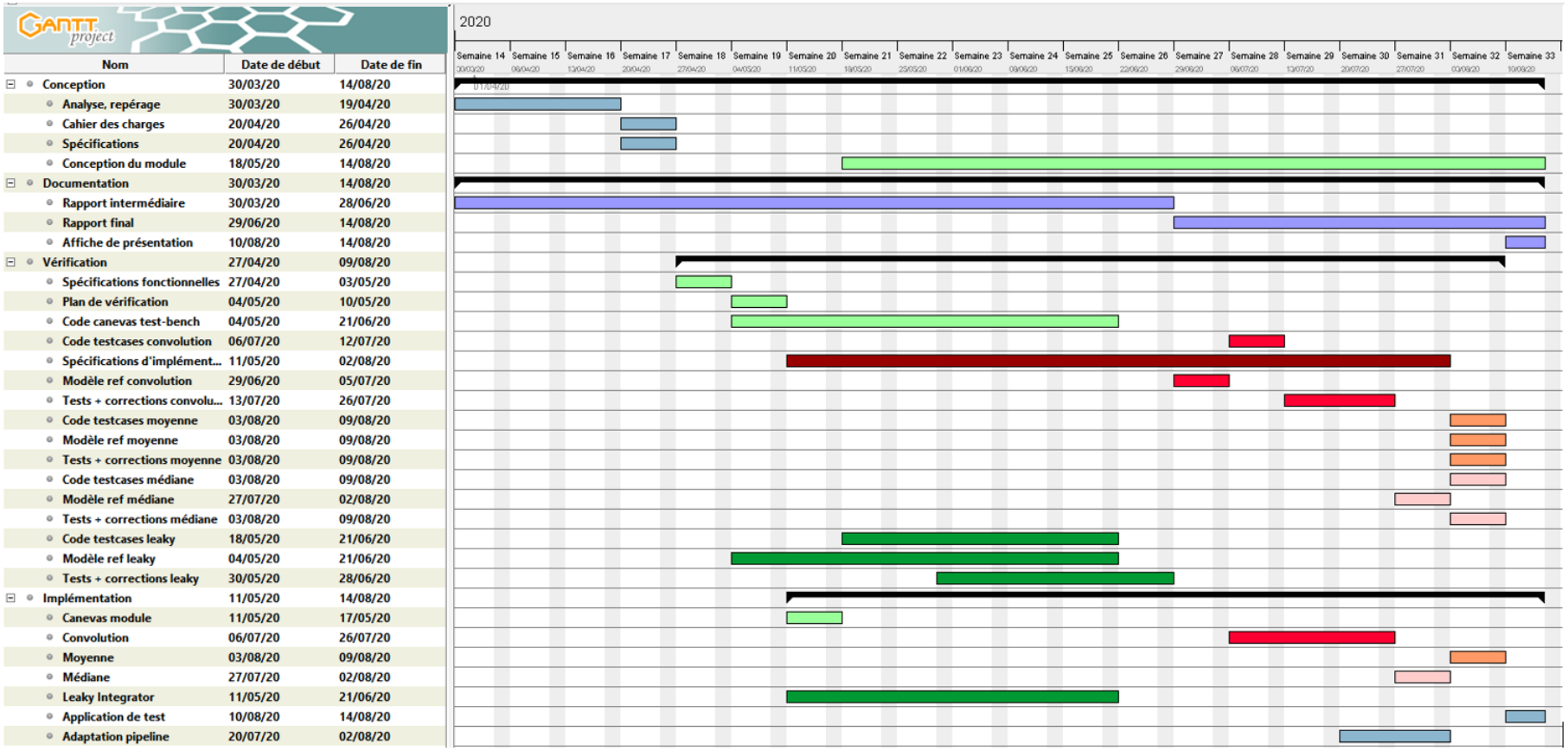
B Plannings

Nom	Date de début	Date de fin
Conception	30/03/20	02/08/20
Analyse, repérage	30/03/20	19/04/20
Cahier des charges	20/04/20	26/04/20
Spécifications	20/04/20	26/04/20
Conception du module	25/05/20	02/08/20
Documentation	30/03/20	16/08/20
Rapport intermédiaire	30/03/20	14/06/20
Rapport final	15/06/20	16/08/20
Affiche de présentation	10/08/20	16/08/20
Vérification	27/04/20	09/08/20
Spécifications fonctionnelles	27/04/20	03/05/20
Plan de vérification	27/04/20	10/05/20
Code canevas test-bench	04/05/20	10/05/20
Code testcases convolution	18/05/20	24/05/20
Spécifications d'implémentation	25/05/20	02/08/20
Modèle ref convolution	25/05/20	07/06/20
Tests + corrections convolution	22/06/20	28/06/20
Code testcases moyenne	29/06/20	05/07/20
Modèle ref moyenne	29/06/20	05/07/20
Tests + corrections moyenne	13/07/20	19/07/20
Code testcases médiane	20/07/20	26/07/20
Modèle ref médiane	20/07/20	26/07/20
Tests + corrections médiane	03/08/20	09/08/20
Implémentation	11/05/20	02/08/20
Canevas module	11/05/20	17/05/20
Convolution	01/06/20	21/06/20
Moyenne	06/07/20	12/07/20
Médiane	27/07/20	02/08/20
Intégration VFC-HD	03/08/20	09/08/20
Mise au propre code et commentair.	03/08/20	09/08/20



Planning original

Marion Dutu Launay



Nom	Date de début	Date de fin
Conception	30/03/20	14/08/20
Analyse, repérage	30/03/20	19/04/20
Cahier des charges	20/04/20	26/04/20
Spécifications	20/04/20	26/04/20
Conception du module	18/05/20	14/08/20
Documentation	30/03/20	14/08/20
Rapport intermédiaire	30/03/20	28/06/20
Rapport final	29/06/20	14/08/20
Affiche de présentation	10/08/20	14/08/20
Vérification	27/04/20	09/08/20
Spécifications fonctionnelles	27/04/20	03/05/20
Plan de vérification	04/05/20	10/05/20
Code canevas test-bench	04/05/20	21/06/20
Code testcases convolution	06/07/20	12/07/20
Spécifications d'implément...	11/05/20	02/08/20
Modèle ref convolution	29/06/20	05/07/20
Tests + corrections convolu...	13/07/20	26/07/20
Code testcases moyenne	03/08/20	09/08/20
Modèle ref moyenne	03/08/20	09/08/20
Tests + corrections moyenne	03/08/20	09/08/20
Code testcases médiane	03/08/20	09/08/20
Modèle ref médiane	27/07/20	02/08/20
Tests + corrections médiane	03/08/20	09/08/20
Code testcases leaky	18/05/20	21/06/20
Modèle ref leaky	04/05/20	21/06/20
Tests + corrections leaky	30/05/20	28/06/20
Implémentation	11/05/20	14/08/20
Canevas module	11/05/20	17/05/20
Convolution	06/07/20	26/07/20
Moyenne	03/08/20	09/08/20
Médiane	27/07/20	02/08/20
Leaky Integrator	11/05/20	21/06/20
Application de test	10/08/20	14/08/20
Adaptation pipeline	20/07/20	02/08/20

Planning réel

Marion Dutu Launay