Erik Verhagen

Master 2 Pro CSINA
Université Joseph Fourier / INP Grenoble

Rapport de stage

Conception d'une fonctionnalité d'autotest pour la chaîne d'acquisition des pertes de faisceau du LHC

Organisation : **CERN**

Group : **AB – BI**

Superviseur : **Dr. Bernd Dehning**

Remerciements :

Je tiens avant tout à remercier celui qui a rendu possible cette expérience unique durant mon cursus universitaire : en la personne du **Dr. Bernd Dehning**, Directeur de la section BL au sein du groupe d'instrumentation du CERN. Sa bienveillance et son appréciation auront permis de faire de ce stage un succès autant sur le point pédagogique que personnel.

Une pensée particulière s'adresse également à **Jonathan Emery**, collègue dans la section BL. Sa patience ainsi que son savoir faire m'auront permis de trouver rapidement une solution chaque fois qu'un challenge supplémentaire est venu s'ajouter au travail de conception. Dans le même esprit, merci également à **Christos Zamantzas** qui a réalisé la conception du design hôte, et dont les explications m'auront permis de réaliser un système adaptable à la configuration existante.

Tous les membres de **la section BL** ont également leur place ici, pour m'avoir accueillis chaleureusement d'une part, mais aussi pour m'avoir fournit l'interaction nécessaire à la réussite de mon stage de fin d'études au sein du CERN.

SOMMAIRE :

Version française

INTRODUCTION

Dans le cadre du Master deuxième année en Conception de Systèmes Intégrés Numériques et Analogiques, la validation du diplôme requiert d'effectuer un stage industriel de quelques mois. Ce rapport a donc pour objectif de rendre compte du travail effectué durant cette période afin de le soumettre à un jury composé d'enseignants de la filière. Dans ce sens il tentera de faire le pont entre les connaissances théoriques acquises durant l'année universitaire et les problèmes pratiques de conception rencontrés durant cette période de travail. Dans un second temps, ce rapport a également un but de documentation à usage de l'organisme d'accueil, pour rendre possible la poursuite du travail effectué et l'utilisation du projet final après mon départ. Dans ce sens il essayera d'apporter un maximum de précisions dans les parties susceptibles d'être l'objet d'une étude ultérieure, ou dans le cas d'une réutilisation du procédé.

Le stage en question s'effectue au sein de l'Organisation Européenne pour la Recherche Nucléaire à Genève, CERN (Suisse). Ce rapport vient compléter le rapport intermédiaire fourni au mois de juillet. Une lecture préalable de celui-ci est donc souhaitable pour comprendre le cadre dans lequel viens s'insérer le projet. Il porte sur la conception et l'implémentation d'un système autonome de test de la chaîne d'acquisition des pertes du faisceau du futur accélérateur du CERN, le LHC. Dans un premier temps, nous allons faire une description du travail à réaliser. Ceci implique une description du point de départ, l'objectif à atteindre et les moyens mis en œuvre pour parvenir à un résultat en septembre. Nous ferons ensuite un gros plan sur le travail de conception réalisé, en mettant l'accent sur ce que la formation de CSINA a apportée pour arriver à terme. Dans un troisième temps, nous aborderons les résultats obtenus à la fin du travail, et nous élaborerons les perspectives que ceux-ci nous offrent dans les mois qui viennent.

Ce rapport s'adresse principalement à un public initié à l'activité de conception de systèmes intégrés. Afin de faciliter la compréhension du rapport lors de sa lecture, l'annexe contenant un certain nombre de compléments d'information a été séparée de ce présent document. De plus, pour garantir la pertinence des annexes en évitant de les surcharger, seul la description de quelques composants clés, y a été reproduite. Pour obtenir la totalité de la description du système, un outil de navigation de code a parallèlement été conçu, sous la forme d'une page web. Il se trouve à l'adresse http://cern.ch/csina et permet d'obtenir le code de l'élément sélectionné en un seul click de souris. Si une interrogation subsiste toutefois à propos du travail réalisé, une adresse mail où me joindre est fournie en annexe.

I) DESCRIPTION DU PROJET

A. Rappel du projet, état initial

Comme il l'a été mentionné dans le rapport intermédiaire fourni au début du mois de juillet, l'objectif du projet est de concevoir un système de test autonome mesurant les caractéristiques de la chaîne d'acquisition des pertes de faisceau du LHC. Cette chaîne d'acquisition permet la mesure du niveau d'interaction entre le faisceau et le tube sous vide, afin d'être en mesure d'initier immédiatement son extraction dans le cas de la perte de contrôle de ce faisceau, qui aurait pour conséquence une détérioration de l'accélérateur. Il s'agit donc d'un point important pour le bon fonctionnement du LHC, ce qui nécessite de connaître avec précision les caractéristiques de la chaîne et leurs évolutions au cours du temps. L'objectif de ce paragraphe est de détailler ces caractéristiques, afin d'établir une approche méthodique au problème.

Le chaîne d'acquisition en question est composée d'un capteur physique (chambre ionisante), entouré d'électronique d'acquisition (analogique et numérique) le tout relié par un certain nombre de lignes de transmission (câbles, fibres optiques) de longue distance. Durant la période d'exploitation du LHC, les caractéristiques de ces trois éléments vont évoluer du fait qu'ils sont soumis à des contraintes telles que :

- des radiations ionisantes
- variations de température (ambiante ou radiative) et de pression
- ainsi qu'à un champs électromagnétique fort.

Pour ne citer qu'un exemple, les fibres optiques ont tendance à s'obscurcir en présence de ces radiations. Il est donc essentiel d'ajouter de l'électronique de contrôle permettant de quantifier l'influence de ces contraintes sur la fiabilité de la chaîne d'acquisition. C'est dans ce cadre que vient s'insérer le projet à réaliser durant ce stage.

Afin d'extraire les caractéristiques d'un système, une approche possible est de réaliser son étude harmonique. C'est l'approche qui a été choisie dans le cas présent. Voici la composition schématique détaillée de la partie mesure de la chaîne d'acquisition. Dans un premier temps, le projet va se concentrer sur cette partie précise, et pourra éventuellement s'élargir sur la chaîne globale telle qu'elle a été décrite dans le rapport intermédiaire.

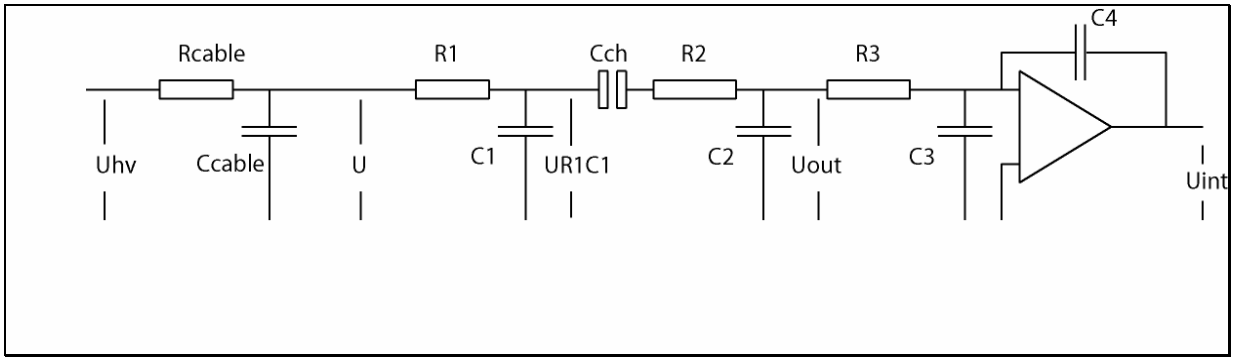


Figure 1 : schéma de la composition de la chaîne d'acquisition

R_{cable} et C_{cable} sont les constantes linéiques du câble de commande haute tension (de quelques centaines de mètres), C1 est une capacité utilisée comme réserve de charges en cas de pertes élevées afin de pouvoir fournir suffisamment de courant au système d'acquisition, et C_{ch} est la capacité représentant la chambre ionisante. Les composants situés au-delà de la chambre se décomposent à nouveau en paramètres linéiques d'un câble puis de l'étage d'entrée du convertisseur courant-fréquence, reconnaissable à l'intégrateur situé en bout de chaîne. Connaissant la composition de la chaîne, on peut dès lors trouver sa fonction de transfert. Le calcul est détaillé en page 4 des annexes.

En accord avec le diagramme de Bode également inclus en annexe, il apparaît que la fonction de transfert est de type passe-bas, définissant ainsi une fréquence d'étude maximale. Cette donnée nous donne déjà une indication sur le design à réaliser en terme de traitement du signal. En effet, la fréquence du signal appliquée au système est basse en regard de la fréquence d'échantillonnage (donnée par le rafraîchissement des sommes présentes en amont, voir chapitre suivant). Ceci implique que le traitement devra s'effectuer sur un nombre choisi d'échantillons, pour éviter la création d'une avalanche de registres durant la synthèse logique.

B. Objectifs du projet

Il a été mentionné également dans le rapport intermédiaire que la cible du projet est un FPGA grande échelle (FPGA de la famille Stratix de **Altera**). Contrairement au travail de conception typique d'un ASIC sur silicium, le FPGA possède la contrainte supplémentaire pour le concepteur de devoir s'adapter à la logique que renferme le circuit de manière pré câblée. Ainsi, un travail exhaustif de documentation à été nécessaire sur ce que renferme le FPGA, ainsi que les règles d'implémentations spécifiques à cette technologie qu'applique l'outil de synthèse. A cette contrainte vient s'ajouter la compatibilité du projet au design déjà existant. En effet l'objectif est de pouvoir simplement ajouter le bloc dans l'espace laissé libre à cet effet dans le FPGA de la chaîne d'acquisition. Une étude complète du système existant a donc été nécessaire pour permettre une adaptation parfaite des deux entités (signaux disponibles, format, représentation).

Le signal harmonique appliqué à la chaîne d'acquisition est généré à l'aide d'un synthétiseur de type DDS (Direct Digital Synthesis, une ROM et un compteur), présent sur le même circuit que celui qui effectue la mesure. Nous aurons donc une référence pour l'étude du gain et de la phase. Ce signal est converti en une tension de commande d'un générateur haute tension. Cette tension étant variable, ceci permet d'effectuer une modulation en tension sur la chambre ionisante. Elle restitue ainsi un courant représentatif du signal modulant. Une conversion courant-fréquence (abordée dans le rapport intermédiaire) et un traitement de sommes à fenêtre glissante permettent de restituer la forme du signal modulant.

Le débit des données entre la carte tunnel (conversion courant-fréquence) et l'électronique de surface dans lequel se trouvera le projet, est d'une nouvelle valeur toutes les 40 µs. Cette valeur est ajoutée à la première somme dont la fenêtre est petite. Au cours du temps, la fenêtre se remplie et le résultat de la somme est fourni à la somme cascadée derrière, dont la taille de fenêtre est plus grande. Il y a ainsi 12 sommes de tailles de fenêtre croissante, de 40 µs à 83 secondes. Le fait que le résultat d'une somme soit transmis à la somme suivante permet de réaliser tout le traitement avec des registres de taille fixée. Ainsi le résultat d'une somme ne sera transmis que quand la fenêtre est pleine ce qui augmente la période de rafraîchissement des sommes, conjointement avec la taille de la fenêtre.

Les données d'entrée (issue d'une conversion courant-fréquence) ont la forme d'une succession de rampes, et sont donc inutilisables pour une étude harmonique. Les opérations successives de sommes à fenêtre glissante permettent de retrouver la forme du signal modulant. Elles nous fourniront donc les données d'entrée pour l'analyse de gain et de phase. L'état actuel du système de mesure de pertes est composé d'un système fournissant les données de toutes les sommes sur 4 canaux simultanément. Le schéma RTL complet du circuit conçu durant ce stage peut être trouvé en annexe page 6. Il permettra de faciliter la navigation dans les blocs fonctionnels, détaillée dans la deuxième partie de ce rapport

C. Environnement de travail, méthodologie

Le CERN est un laboratoire de recherche en physique fondamentale. Le travail à réaliser au sein de l'organisme se situe dans le processus de développement du nouvel accélérateur de particules, prévu pour début 2008. De ce fait, le flot de conception dévie complètement du schéma habituel de l'industrie, dont les contraintes sont le « Time To Market » et la réduction de coûts induits par la production de masse. Dans le contexte qui nous intéresse ici, les contraintes sont la précision, la fiabilité et la robustesse, trois notions qui, au CERN, dépassent les standards présents dans l'industrie.

Il n'existe pas au CERN une méthodologie de conception standardisée à la manière d'un livret de contraintes à respecter comme dans certaines industries. Il y a toutefois une collection plus que complète d'outils parmi lesquels il faut faire des choix, et dont on doit faire un usage efficace. La méthodologie suivit ici est celle qui a été développée par les intervenants industriels durant la formation de CSINA. Elle s'articule en trois principales parties :

- le développement et la simulation d'un modèle de référence (en langage évolué)
- l'implémentation et la simulation fonctionnelle du modèle validé à l'aide d'un langage de description matérielle (VHDL)
- la synthèse puis le test de la description sur une plate-forme de prototypage (FPGA) qui sera réalisée sur une version allégée du contrôleur final de pertes de faisceau.

Les outils choisis pour la réalisation de ce travail sont les suivants :

- Un **compilateur C** pour la réalisation et la simulation du modèle de référence. Le CERN met à la disposition de ses employés une grappe de calcul (Linux) pour réaliser des travaux ponctuels mais intensifs (batch processing) de ce type.

- La simulation fonctionnelle est réalisée à l'aide de **ModelSim**, logiciel de référence dans ce domaine, édité par Mentor Graphics. Le choix s'est porté sur ce logiciel car les notions de son utilisation étaient déjà acquises.

- Lorsque l'aspect de timing était critique (comme cela a été le cas pour l'unité de division) une simulation post-synthèse a également été réalisée à l'aide de ModelSim. La synthèse s'effectuait alors avec Mentor Graphics **Precision** (successeur de Leonardo) et **Synplify Pro** de Synplicity pour réaliser les netlists VHDL.

- La compilation, synthèse et placement & routage ont été effectués avec **Quartus**, le logiciel propriétaire d'Altera permettant de finaliser le prototypage sur FPGA, et réaliser une interface de mesure des signaux internes à des fins de débogage

II) REALISATION

A. Travail de conception

L'objectif de ce stage, et en accord avec la formation, est de réaliser un travail de conception. Malgré l'aspect atypique du domaine d'application des connaissances en microélectronique, ce stage répond exactement aux attentes du Master CSINA, grâce à la confiance et l'autonomie accordée par **Bernd Dehning**. En effet, toutes les étapes de la conception classique ont été abordées. Ces étapes sont : l'étude du système existant et les possibilités d'interfaces possibles puis l'élaboration de la méthode de mesure, ensuite la simulation des différents modèles envisageables (« golden model » en C), l'implémentation et la simulation du modèle choisi (VHDL), et pour finir la caractérisation du résultat à des fins de documentation. Ce chapitre a pour but de détailler ces différentes étapes, pour mettre en évidence les liens entre le stage et la formation.

Comme il l'a été abordé dans la partie précédente, il faut réaliser une interface avec le circuit existant. En effet, les données que nous allons exploiter sont regroupées par sommes, chaque somme rassemblant 4 canaux (32 bits par canal, donc128 bit par somme). Voici le bloc auquel le projet sera cascadé :

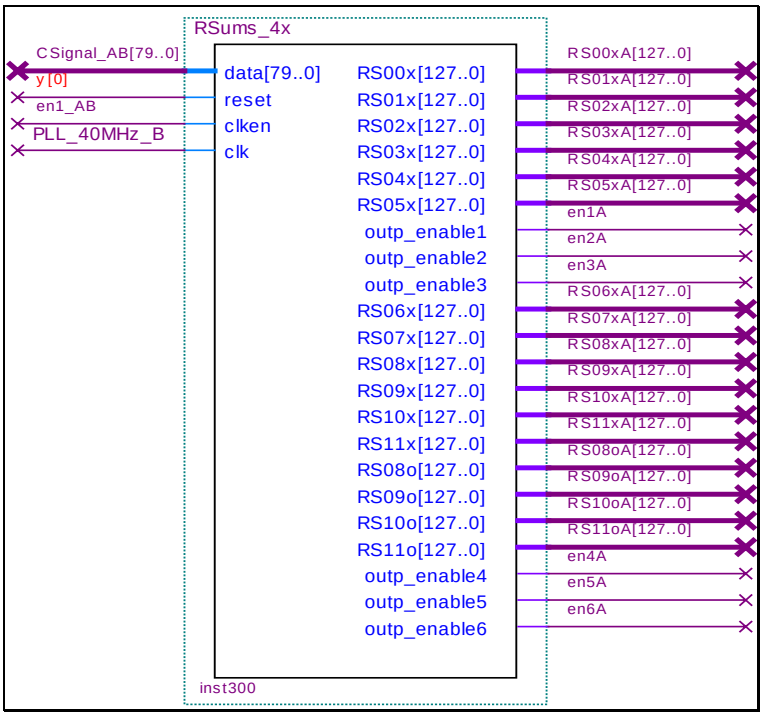


Figure 2 : Bloc auquel viendra se cascader le projet

Le traitement ne s'effectuera que sur les données d'une seule somme de chaque canal. Le numéro de cette somme n'est pas encore connu à ce stade et nécessitera une étude approfondie ainsi que des tests. Un décodeur devra donc fournir de manière séparée les données issues de la même somme pour chaque différent canal. Afin de limiter le nombre d'interconnexions, un paramètre générique indiquant le numéro de la somme est donc utilisé et renseigné au moment de la synthèse.

Pour se faire une idée plus précise de l'entité, voici une vue du bloc générique de décodage, qui sera cascadé au bloc représenté précédemment.

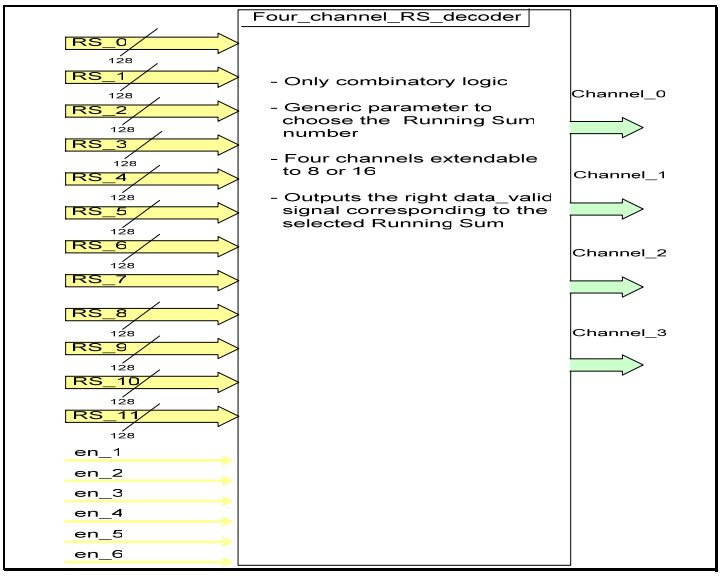


Figure 3 : Décodeur d'entrée du projet

Dans un premier temps, nous allons nous intéresser à la mesure du gain de la chaîne. Un gain se mesure toujours en effectuant le rapport des amplitudes des signaux d'entrée et de sortie. Le premier bloc fonctionnel à réaliser a donc été un extracteur de valeur crête à crête. Pour chaque bloc fonctionnel il y a deux approches : le circuit synchrone ou le bloc combinatoire. Cet extracteur de valeur crête à crête est noté *RS_peak_to_peak* dans le schéma RTL fourni en annexe. Ce circuit doit réagir chaque fois qu'une nouvelle valeur se présente en entrée. Il utilise ensuite deux fonctions de comparaison purement combinatoire pour savoir si cette valeur est supérieure ou inférieure à la précédente jusqu'à ce que l'on atteigne une crête. Ce circuit est donc synchrone de l'arrivée d'une nouvelle valeur. Ceci va se traduire en VHDL par une structure ressemblant à ceci :

```
process (data_available, local_reset, local_RSxx_in, max_memory, min_memory)

begin
  if local_reset = '1' then                                  -- reset asynchrone

  elsif data_available 'event AND data_available = '1' then
                                                             -- instructions synchrones
                                                             -- à l'horloge locale
  end if;
    local_peak2peak_out <= max_memory - min_memory;          -- instructions concurrentes
end process;
```

Ceci sera interprété comme une horloge locale lors de la synthèse. D'après la documentation du constructeur, le design peut contenir 16 arbres d'horloge différents à condition, pour le concepteur, d'être averti des mécanismes de synchronisation entre elles. Quatre de ces extracteurs seront nécessaires pour les 4 canaux. Pour réaliser le calcul du gain, la référence doit également fournir une valeur crête à crête régulièrement. Afin de limiter les problèmes de synchronisme liés à l'utilisation d'un calcul commun sur des données issues de deux horloges différentes, une fréquence commune doit être choisie. Rappelons en effet que la somme fournissant les données relatives aux canaux n'est pas choisie, chaque somme possédant sa fréquence de rafraîchissement. Elle s'étend de 40 µs à 1.3 secondes. La fréquence de rafraîchissement de la référence quant à elle est fixée à 255 échantillons par période (DDS).

Nous choisirons donc celle-ci comme fréquence commune. Ceci permettra également de normaliser le calcul de la phase et rendre le traitement indépendant de la fréquence appliquée à l'entrée de la chaîne. Un arbre d'horloge commun à ces 5 extracteurs de valeur crête à crête sera donc créé. Voici le schéma bloc d'un extracteur de valeur crête à crête :

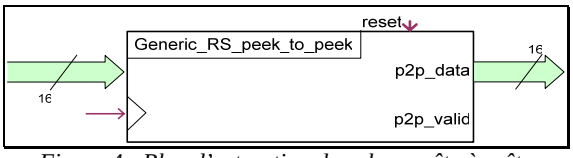


Figure 4 : Bloc d'extraction de valeur crête à crête

Après les premières simulations fonctionnelles de ce bloc (réalisées en C), il est apparu que les sommes donnent une restitution fidèle de la forme du signal, auquel est ajoutée un très fort offset. Ainsi, dépendamment des sommes, les bits de poids le plus fort restent à '1'. Seuls les bits de poids faible sont donc représentatifs du signal. Dans les cas usuels (choix de la somme RS_07 et amplitude modérée du signal modulant), on a constaté que seuls les 16 bits de poids le plus faible sont suffisants pour représenter les variations du signal d'entrée. C'est ce qui explique la largeur des bus d'entrée et de sortie. Cette largeur a été choisie pour réaliser les tests, mais naturellement, le design doit offrir une souplesse suffisante afin de laisser jusqu'au bout le choix de la somme. Cette souplesse sera détaillée dans un prochain chapitre réservé à l'implémentation de la description. On peut voir également en sortie un signal *p2p_valid*. Ce signal est validé durant un cycle d'horloge locale à chaque période complète du signal d'entrée (quand le signal est passé par un maximum puis un minimum) pour indiquer aux blocs suivants qu'une nouvelle valeur crête à crête est disponible.

Le calcul du gain implique la création d'une unité de division. A ce stade, 2 possibilités étaient offertes. La première est d'implanter une unité arithmétique et logique trouvée dans la littérature. Cette solution aurait été coûteuse en place, et disproportionnée pour un seul diviseur. La solution retenue a donc été de réaliser une unité de division de toute pièce. Pour faire partie efficacement du projet, ce diviseur devra satisfaire à quelques contraintes. Premièrement, la place occupée en terme d'éléments logiques devra bien sûr être maîtrisée. En effet les diviseurs sont habituellement des circuits complexes et coûteux en place. Deuxièmement, et afin de faciliter l'exploitation des résultats (réalisée par un logiciel tournant sur une unité de traitement indépendante) la sortie devra être conforme à la représentation standard des nombres en virgule fixe, et le nombre de décimales doit être configurables lors de la synthèse. Ce nombre (garant de la précision de la mesure) ne sera connu qu'après un certain nombre d'essais réels. Pour rappel, voici la représentation standard des pondérations de nombres à virgule fixe (mot de 10 bits, convention à 4 digits après la virgule) :

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

La troisième contrainte porte sur la réalisation de la division en un nombre finis de cycles d'horloge. Ceci est indispensable pour permettre par la suite la réalisation d'un bilan d'exécution et ainsi définir combien de cycles sont nécessaires pour la production d'un jeu de mesure gain/phase. Ce bilan permettra notamment de spécifier au final la fréquence d'entrée maximale (ou le nombre d'échantillons par période) à appliquer au circuit, ainsi que le choix de sa fréquence d'horloge.

Comme nous allons le voir plus loin dans ce rapport, ce bloc sera constitué d'un soustracteur combinatoire pour le calcul des restes et d'une partie synchrone réalisant les

différentes manipulations de bits de manière « pipelinée » afin de gagner du temps d'exécution. Un signal de « *start* » sera présent, permettant de mettre en route la division quand les données en entrée sont prêtes et devant rester à '1' durant toute la durée de la division. Ce signal servira également de reset synchrone dans le cas '0'. La présence d'un bit indiquant un débordement (division par zéro par exemple) est également obligatoire afin d'indiquer un résultat erroné. La présence d'un signal indiquant la fin de la division est présent également, mais reste en cours de discussion, sa nécessité n'étant pas assurée du fait que le nombre de cycles pour réaliser la division est fini. Voici la représentation du bloc diviseur tel que vous pouvez le retrouver dans le schéma RTL fourni en annexes :

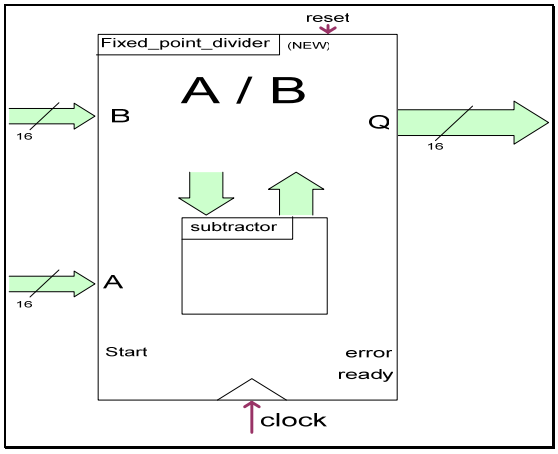


Figure 5 : vue de l'unité de calcul de division

Intéressons-nous à présent à la phase. Elle s'obtient en mesurant la position relative d'une crête du signal à mesurer par rapport à une crête du signal de référence. Nous avons vu dans la description de l'extracteur de valeur crête à crête qu'un signal nommé *p2p_valid* indique justement quand le signal passe par chaque minimum. Il est dès lors facile d'implanter 2 compteurs, le premier pour compter le nombre d'échantillons (cycles d'horloge locale) que comporte une période du signal mesuré (on admet ici que les fréquences sont identiques), puis un deuxième permettant le compter le nombre de cycles de décalage entre les 2 signaux. Le rapport de ces deux nombres permet ensuite d'obtenir une valeur du déphasage, et ainsi de la phase induite du système à cette fréquence. Afin d'obtenir un maximum de précision, la division s'effectue avec le compte du nombre de cycles par période (plus grand) au numérateur. Ceci implique qu'il faudra prendre l'inverse du résultat et multiplier ce nombre par 360 pour un résultat en degrés ou par 2π pour un résultat en radians.

Les deux compteurs sont d'un type un peu particulier. En effet, leur signal de remise à zéro (reset) est couplé avec un mécanisme gardant en mémoire la valeur atteinte par le compteur juste avant ce reset. Ceci est nécessaire car suite au déphasage, les compteurs ne produisent pas leur valeur en même temps, et la division requiert les valeurs produites aux minima déphasés des 2 signaux. D'un point de vue de la logique, ceci se traduirait par la génération d'une bascule sur niveau (appelée « latch »). Cette catégorie de circuit n'est pas présente dans le FPGA. D'un point de vue de la synthèse donc, cette structure fera appel à une bascule D. Comme toutes les bascules D du FPGA ont leur entrée d'horloge connectée à un arbre d'horloge, l'outil de synthèse génèrera un signal de validation « clock enable », émulant ainsi une bascule RS classique.

Cette connexion systématique des entrées d'horloge de toutes les bascules à un arbre d'horloge est un point qu'il faut garder en mémoire. En effet ceci interdit formellement l'usage

simple de bascules pour garder une valeur en mémoire dans un système qui à première vue n'inclut pas d'horloge globale. Les machines à états finis sont par exemple concernées s'il existe des signaux « trans-état » c'est à dire gardant leur valeur au cours de plusieurs états, et effectuant un traitement sur ces signaux dans un des états seulement. Voici une représentation d'un des deux compteurs présents pour le calcul du déphasage :

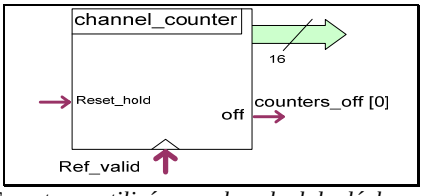


Figure 6 : Compteurs utilisés pour le calcul du déphasage

Afin de faciliter le contrôle interne du système de test, et pour éviter la production de données erronées concernant la phase, un signal indiquant si le compteur a atteint sa valeur maximale est prévu. Ce signal est activé en cas de débordement.

La division devra bien entendu se faire à l'aide de la même unité de calcul que celle utilisée pour le gain. Ceci implique un ordonnancement des instants de mesure et de calcul, conduisant tout naturellement à la réalisation d'une unité de contrôle. Elle aura pour fonction d'assurer le séquençage correct des opérations, à commencer par l'extraction des mesures d'amplitude puis de déphasage, de réaliser ensuite les divisions l'une après l'autre en assurant la présence des données adéquates, et pour finir, de stocker les résultats dans les registres de sortie prévus à cet effet. Ce séquençage devra s'effectuer de manière continue et circulaire pour les 4 canaux.

Une unité de contrôle est généralement réalisée à l'aide d'une machine à états finis. Dans notre cas, chaque état de la machine sera défini par une action du système, ce qui simplifiera la logique de calcul de l'état suivant. Les entrées nécessaires sont donc les suivantes :

- présence d'une nouvelle valeur crête à crête sur la référence
- présence d'une nouvelle valeur crête à crête sur les canaux
- division terminée avec succès
- division terminée avec erreur
- débordement d'un compteur
- activation globale du système de test

En sortie et dépendant uniquement de l'état courant de la machine (configuration de Moore), nous aurons :

- la sélection du registre de sortie
- un signal permettant le cas échéant de connaître les défaillances internes du système à l'aide de codes d'erreurs
- un signal contrôlant les multiplexeurs d'entrée du diviseur afin de choisir la présentation de données d'amplitude ou de phase à ce dernier
- un signal permettant de mettre en route l'unité de division pour un nouveau calcul
- un signal de sélection du canal à étudier (parcourant un à un tous les 4 de manière séquentielle)

Précision qu'il n'est pas possible d'effectuer les calculs relatifs aux 4 canaux les uns après les autres à l'intérieur de la même période du signal d'entrée. En effet, les chambres ionisantes sont réparties en des points stratégiques du tunnel. Il s'ensuit que la longueur des câbles de commande haute tension et de récupération des données sont de longueurs variables d'une chambre à l'autre, induisant un déphasage différent pour chaque chambre. Les 4 canaux ne produiront donc pas leurs *p2p_valid* en même temps, rendant impossible un calcul groupé sur les 4 canaux. C'est la raison de la présence du multiplexeur 4 vers 1 de largeur 1 bit. Il permet de fournir un reset au compteur de déphasage au moment précis d'apparition d'une nouvelle valeur. Ceci s'effectuera cependant à la période suivante du calcul sur le canal précédent.

Voici une vue du bloc de contrôle. Il est à noter que le nombre de sorties est de 8, pour l'amplitude puis la phase des 4 canaux. Le fonctionnement de la machine à états finis sera détaillé dans le chapitre consacré à l'implémentation de la description.

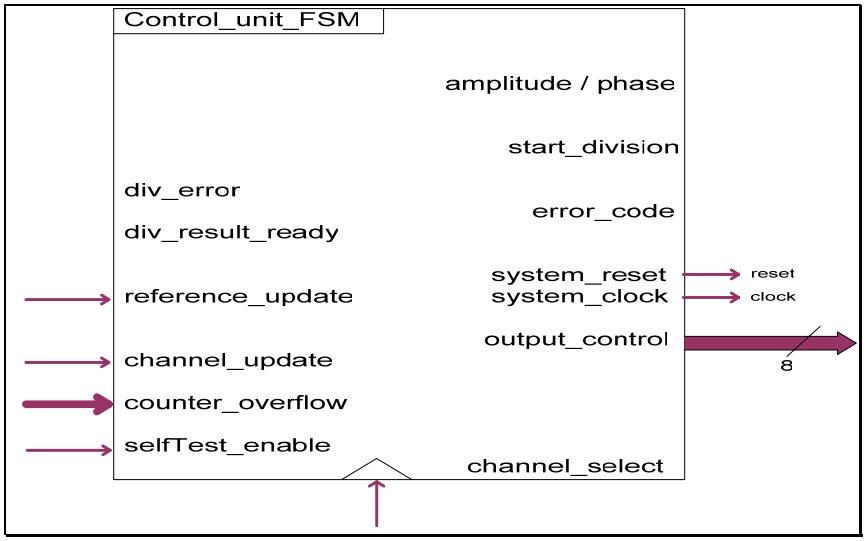


Figure 7 : Gros plan sur l'unité de contrôle du projet

La présence d'un générateur de codes d'erreur a été dévoilée précédemment mais ne nécessite pas que l'on y consacre un paragraphe entier. Du point de vue de la synthèse, il s'agit simplement d'un multiplexeur commandé par l'unité de contrôle et possédant sa première entrée raccordée à la sortie du diviseur pour laisser passer les informations si elles sont pertinentes, et une série de constantes prédéfinies servant de code d'erreur sur les autres entrées. Le code indique l'erreur qui s'est produite, et sa place dans les signaux de sortie de canal où elle s'est produite.

Pour finir sur le travail de conception, il est important de préciser que les valeurs de phase et d'amplitude issues de ce système sont calculées sur l'ensemble de la chaîne d'acquisition. Ceci comprend donc les équipements de mesure proprement dit et dont nous nous sommes entrepris d'extraire la fonction de transfert, mais également toute l'électronique de traitement de ces mesures. Le traitement final de ces données devra donc inclure de retrancher aux mesures obtenues une constante d'amplitude et de phase, ce qui ne posera aucun problème puisque ces constantes sont entièrement statiques. Ces constantes incluront le décalage en phase induit par le codage, l'envoie, le décodage puis le calcul du CRC (code redondant d'erreur) et le contrôle d'intégrité de l'information sur les deux fibres optiques

redondantes dans un premier temps. Viendront s'y ajouter la phase et le gain induit par le calcul des sommes à fenêtre glissante. Vous trouverez en annexe les courbes de l'étude réalisée en premier travail durant le stage sur l'influence de l'opération de calcul de ces sommes sur la phase et l'amplitude. On peut voir notamment que la phase et l'amplitude (amplitude normalisée par rapport à la taille de la fenêtre dans cette étude) induite par ce calcul est une fonction linéaire du nombre d'échantillons regroupés dans la somme. Il faudra ensuite tenir compte du gain et de la phase produite par le circuit de test lui-même, et qui est loin d'être négligeable comme nous allons le voir dans le dernier chapitre de cette partie

B. Création de l'environnement de simulation

Comme il l'a été énoncé dans le précédent chapitre, un travail systématique de validation de l'implémentation pour s'assurer de sa conformité avec le « golden model » (réalisé en C) a été accompli pour chaque bloc fonctionnel. Afin de réaliser cette étape, il a été nécessaire de créer un environnement de simulation fonctionnel, flexible et permettant de maximiser la couverture de fautes. Ce chapitre permet de faire une description de l'environnement réalisé, donner un aperçu de ses points forts, sans oublier de détailler sa construction.

Au fur et à mesure de l'avancement du stage, l'implémentation du design hôte auquel doit venir se greffer le projet était également en cours de finition. Un certain nombre de tests de ce design ont donc été effectués, produisant des données dont la ressemblance s'approchait de plus en plus du résultat final. Ces données m'ont été fournies sous forme de fichiers textes. Ils contiennent plusieurs colonnes, chaque colonne restituant une somme d'un canal au cours du temps. L'objectif est de construire un environnement de test similaire pour le modèle en C et celui en VHDL, afin d'offrir une comparaison raisonnable de leurs fonctionnement.

Intéressons nous d'abord au cas de la simulation des modèles de référence en C. Dans ce cas de figure, le traitement des données issues du fichier et leur application au modèle à simuler sont assez triviaux. En effet la lecture du fichier se fait colonne par colonne, et chaque valeur correspondante à un instant de mesure est insérée dans un tableau. Chaque tableau correspond donc aux données d'une somme d'un canal. Nous obtenons donc un ensemble de tableaux qui fourniront les données d'entrée au modèle de référence à simuler. Bien entendu cette simulation ne permet d'effectuer qu'une simulation fonctionnelle du modèle, mais ceci s'est avéré très utile lors de l'élaboration de l'algorithme de division, ainsi que les filtres comme nous le verrons plus loin. Une description détaillée de cette simulation fonctionnelle du modèle de référence est en effet abordée dans un prochain chapitre, en prenant comme exemple le travail de synthèse d'un filtre numérique.

La réalisation du testbench VHDL à partir des données réelles quant à lui est nettement moins triviale. Le canevas habituel d'un testbench VHDL rassemble tous les composants à simuler, les connectent entre eux à l'aide de signaux, puis un processus affecte séquentiellement à tous les signaux d'entrées une valeur, en séparant chaque nouvelle affectation par `wait for`. Malheureusement, le fichier de données initial contenait plus de 280 000 valeurs distinctes, ce qui est difficilement gérable dans un seul fichier. De plus, réaliser un seul fichier de testbench n'offre aucune souplesse à cette échelle, puisque le testbench devra être généré à nouveau pour chaque entité sous test. Le choix s'est donc porté

sur la création d'un véritable environnement de simulation décrit en VHDL, dans lequel on vient greffer le circuit sous test, à la manière d'un composant dans un support sur carte électronique. Les données sont présentes dans des ROM couplées à un générateur d'adresse (compteur) venant parcourir toutes les adresses des ROM, et fournir ainsi séquentiellement les données au circuit à simuler. Le fichier testbench quant à lui sera donc réduit au plus simple, c'est à dire une horloge de commande du générateur d'adresse, un signal de commencement de test, et un reset.

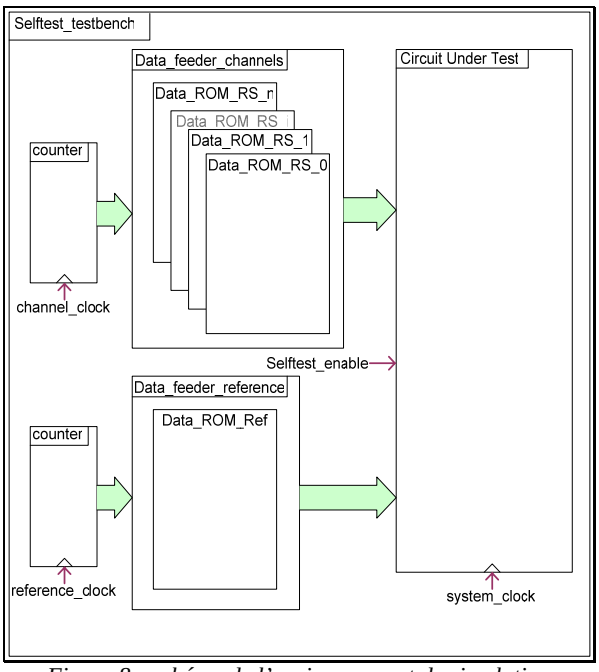


Figure 8 : schéma de l'environnement de simulation

La difficulté réside à présent dans la réalisation des ROM contenants les données. Après une rapide réflexion sur les outils à disposition, le plus simple semblait de réaliser un générateur de code VHDL en C. En effet la structure canonique d'une ROM en VHDL est assez simple, et ceci permettrait de réaliser ces ROM de manière automatique à chaque nouveau fichier de données. La structure canonique d'une ROM est la suivante :

```
entity simple_data_ROM is
  port  (
         address_in              : in    natural;
         data_out                : out   std_logic_vector (3 downto 0)
        );
end entity;

architecture valid of simple_data_ROM is
begin
  process(address_in)
  begin
    case address_in is
      when 0 =>
        data_out <= conv_std_logic_vector(2, 4);
      when 1 =>
        data_out <= conv_std_logic_vector(3, 4);
      .
      .
      when others =>
        data_out <= "----";
    end case;
  end process;
end architecture;
```

Cette structure répétitive (**when**) permet d'inclure la génération de code de condition dans une boucle en C et remplacer la valeur à envoyer en sortie par la case adéquate du tableau. Le principe est le suivant :

- on effectue une lecture du fichier de données et comme précédemment, toutes les valeurs sont mises dans des tableaux représentant les ROM à réaliser (une ROM par somme donc 12 au total, chaque valeur de somme regroupe 4 canaux, afin de se rapprocher le plus possible du format des entrées du projet, cf. chapitre A.)

- on génère la description de l'**entité** (qu'on appellera *data_ROM*) de telle sorte qu'elle corresponde exactement aux entrées du circuit à tester en terme de sorties. Tous les bus des sommes doivent être présents (même si une seule sera choisie). Deux **paramètres génériques** sont ajoutés, le premier étant pour choisir la somme à sortir, le deuxième est un paramètre de gain qu'on pourra multiplier à chaque valeur en sortie afin de tester ultérieurement l'exactitude des calculs :

```
when 0 =>
  data_out <= conv_std_logic_vector( GAIN * 2, 32);
```

- on parcours un à un les tableaux en remplissant les conditions **when** par les valeurs des cases du tableau correspondantes. Chaque tableau (chaque ROM) correspond à une **architecture** distincte. Nous aurons donc 12 architectures différentes de l'entité *data_ROM* dans le même fichier VHDL

- on crée ensuite une **configuration** VHDL (appelée *data_feeder*) qui associe à l'entité *data_ROM* l'architecture correspondante au choix de la somme (fournie en paramètre générique) sur laquelle on veut effectuer la simulation.

Le fichier VHDL ainsi généré produit, après une longue compilation, un bloc qui fournira les données de manière séquentielle au circuit à simuler à l'intérieur de l'environnement de test. Tous les composants de cet environnement sont rassemblés dans l'architecture d'une entité, que l'on aura tout de même appelée testbench pour des raisons de commodité à la relecture. Cette architecture comprend les paramètres génériques cités plus haut, comme le numéro de la somme choisie, et le gain appliqué aux données de cette somme. L'affectation de ces paramètres génériques se fait par l'intermédiaire de constantes définies au début de la description de l'architecture. Cependant, afin de réaliser une simulation exhaustive, on rajoute des paramètres supplémentaires à cet endroit, permettant d'augmenter la **couverture de fautes**. C'est à ce stade que l'on peut pleinement apprécier la souplesse de l'environnement de simulation. Ces paramètres permettent également de rajouter artificiellement une phase (insertion de périodes et d'échantillons de retard), les différentes fréquences de rafraîchissement ainsi qu'un décalage temporel entre les deux. De cette manière on peut simuler la réponse du circuit dans le cas d'une perte totale de synchronisation entre les signaux, chose qui sera le cas dans le fonctionnement réel.

```
architecture testbench_selfTest_amplitude_phase_4_channels_v2 of testbench is

  constant reference_signal          : natural range 1 to 3 := 1;-- choose which reference
  constant running_sum_number        : natural range 0 to 11 := 7;-- choose de sum
  constant selfTest_gain             : natural range 1 to 100 := 1;-- artificial gain
  constant selfTest_delay_periods    : natural range 0 to 46 := 0;-- artificial phase (periods)
  constant selfTest_phase            : natural range 0 to 16 := 0;-- artificial phase (samples)
  constant reference_period          : time := 2.6 us; -- reference sample period
  constant channel_period            : time := 1.5 us; -- reference sample period
  constant selfTest_skew             : time := 0.23 us; -- skew between reference and channels
```

Le déphasage des horloges est décrit comme suit :

```
reference_clock : process
begin
  clock_reference <= '1', '0' after (reference_period / 2);
  wait for reference_period;
end process;

channels_clock : process
  variable first_time : boolean := TRUE;
begin
  if first_time = TRUE then
    clock_channels <= '0';
    first_time := FALSE;
    wait for selfTest_skew;
  else
    clock_channels <= '1', '0' after (channel_period / 2);
    wait for channel_period;
  end if;
end process;
```

On peut observer que l'on réalise simplement un décalage de *selfTest_skew* à la première période de l'horloge. Ce décalage restera donc le long de la simulation, permettant de s'assurer que les 2 horloges ne se recouvrent à aucun moment.

C. Implantation de la description

Dans ce chapitre, nous entrerons dans les détails de la description matérielle, en mettant l'accent sur l'organisation du travail réalisé (aspects de réutilisation de code, paramètres génériques), ainsi que sur certains points clés du design, tels que la machine à états finis et le diviseur.

Intéressons nous pour commencer à l'unité de division. Pour repousser au plus tard possible la réalisation finale de ce composant complexe, une version intermédiaire a été utilisée pendant plusieurs mois. Cette version était composée d'une machine à états finis. L'algorithme était simple : on retranche le diviseur (dénominateur) au dividende (numérateur) en mettant le résultat dans le dividende. Chaque soustraction incrémente un compteur jusqu'à ce que le dividende est plus petit que le diviseur. Le résultat est fourni par la valeur du compteur, et le reste est fourni par la dernière valeur du dividende. En effectuant un décalage à gauche du reste et en l'injectant à nouveau dans le dividende, on peut calculer ensuite la partie décimale. Malheureusement ce circuit ne répondait pas à l'une des contraintes énoncées précédemment. En effet, le nombre de cycles d'horloge nécessaires à la division n'était pas fixe, puisque dépendante des opérandes (c'est ce qui explique la présence du signal de fin de division en entrée du l'unité de contrôle). Cependant, ce diviseur aura permis de réaliser des premières mesures, ainsi que de se familiariser à nouveau avec l'arithmétique à virgule fixe.

La version finale de l'unité de division est quant à elle basée sur 2 opérations binaires simples : la soustraction combinatoire, et le décalage séquentiel à gauche. De plus nous allons voir que l'exécution de la division au fil des cycles d'horloge permet de choisir facilement le nombre de digits significatifs. Nous verrons également que pour faciliter la réutilisation future de cette entité, le dividende et le diviseur ne nécessitent pas d'être de la même taille binaire. C'est pourtant le cas dans notre projet, mais ce n'est pas imposé par l'architecture. Voici le détail de l'algorithme :

- La première étape est de charger les registres internes avec le diviseur et le dividende présents à l'entrée. Ceci occupe un cycle d'horloge et devra être déclenché dès la validation du signal *division_start*. Le diviseur doit être placé dans un registre verrouillé car il ne variera pas durant l'exécution de la division. Le dividende par contre va effectuer un décalage séquentiel à gauche. Il sera donc placé du coté des bits de poids le plus faible d'un registre dont la taille est la somme des tailles de vecteurs de bits du dividende et du diviseur (`divider'length + dividend'length`). Dans notre cas, ces tailles sont identiques, donc le registre accueillant le dividende sera de taille double. Voici une représentation simplifiée :

  Registre de stockage du dividende (à décalage) :

0	0	0	0	0	0	0	0	D	I	V	I	D	E	N	D

  Registre de stockage du diviseur (verrouillé) :

	D	I	V	I	D	E	R

- Le registre contenant le diviseur ainsi que la partie haute du registre contenant le dividende forment les opérandes du soustracteur combinatoire. Ainsi, un résultat sera toujours présent dans le résultat de la soustraction. De la même manière qu'un additionneur complet possède un bit de retenue, le soustracteur affiche son débordement à l'aide d'un bit d'emprunt (« borrow » en anglais). Ce bit est à '1' lorsque l'opérande soustracteur est plus grand que la quantité sur laquelle on veut réaliser une soustraction. Il correspond également au signe du résultat, l'opération fournissant un nombre négatif de ce cas de figure.

		0	0	0	0	0	0	0	0	D	I	V	I	D	E	N	D
-			D	I	V	I	D	E	R								
=	B		R	E	M	I	N	D	E	R							

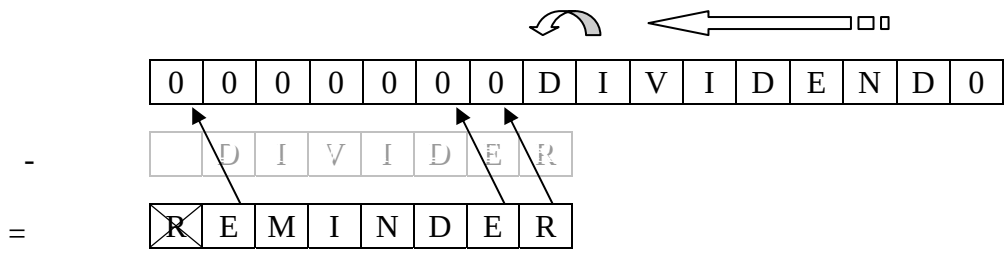
  B indique le bit d'emprunt, le résultat fourni le reste de la division partielle

- Ce bit d'emprunt indique ainsi si une soustraction a été possible. Si ce n'est pas le cas, cela indique que le diviseur ne rentre pas dans le dividende partiel (nous l'appelons « partiel » car il ne contient pas tous les bits du dividende). Dans ce cas, le bit d'emprunt est à '1' (débordement) donc le résultat de la division partielle est 0. Dans le cas opposé d'une soustraction possible, il indique (par un '0') que le diviseur est présent au moins une fois dans le dividende partiel. Un registre à décalage contenant le résultat devra donc être mis à jour avec l'inverse du bit d'emprunt avant d'effectuer un décalage pour préparer la division partielle (soustraction) suivante.

Q	O	T	I	E	N	T		⟸	Not(B)

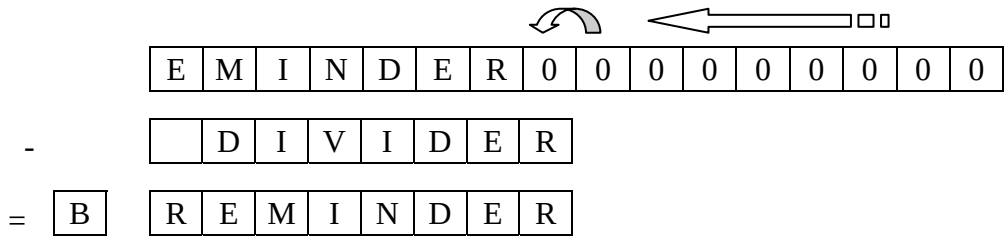
- Dans un même temps (concurrence des actions : création d'un pipeline), le registre contenant le dividende devra effectuer un décalage à gauche pour remplir le premier bit (de poids le plus faible) de l'opérande supérieur de la soustraction. Simultanément toujours, les bits de poids le plus fort de l'opérande de la soustraction devront être remplis par le résultat de la soustraction si elle a eu lieu, ou par des zéros dans le cas contraire.

Cas d'une soustraction réussie (le diviseur rentre dans le dividende partiel) :



Dans le cas d'une soustraction avec emprunt, le registre du dividende n'effectue que le décalage vers la gauche, jusqu'à ce que la soustraction ne produise plus de bit d'emprunt à '1'. Comme la taille des registres de résultat de soustraction et de la partie opérande du registre de dividende est la même, on peut se demander si on ne risque pas de tronquer la division et rendre le résultat incorrect. Mais du fait du choix de l'opérande supérieur de la soustraction (issue d'un décalage) le bit de poids le plus fort du résultat sera toujours à '0' dans le cas d'une soustraction réussie. Ce bit peut dès lors être oublié.

Venons-en à présent au nombre de cycles et à la précision. Le décalage du dividende n'est pas soumis à une condition. En effet si l'opérande supérieur de la soustraction est suffisant pour effectuer l'opération, la soustraction suivante devra s'effectuer sur le reste concaténé d'un bit du dividende décalé. Dans le cas contraire, la soustraction nécessite au moins un bit de plus vers la gauche. La division entière est donc finie après que le dividende ai effectué un décalage complet vers la gauche, c'est à dire un décalage du nombre de bits qui le constitue (`dividend'length`). Une division d'un dividende de 16 bits produira donc un résultat entier après 16 cycles d'horloge. Cependant, notre projet nécessite une précision supérieure à la division entière. En effet nous aimerions avoir une lecture plus précise en particulier de la phase, à 1 ou 2 chiffres décimaux (base 10) après la virgule. Comme dans le cas de la division décimale à la main, les chiffres après la virgule du résultat sont obtenus en « abaissant » un zéro pour le placer à droite du reste et continuer l'opération. Ici ces zéros sont fournis par la poursuite du décalage à gauche du dividende.



Ce mécanisme de division peut ainsi se prolonger jusqu' à l'infini, il ne cessera de rajouter des chiffres significatifs. La limitation vient du registre de résultat (quotient). Ce

registre à décalage est de taille fixée, ce qui implique que sur 16 bits par exemple, si l'on veut 6 bits de partie décimale, la partie entière ne devra pas excéder 2^{10} (sur 10 bits) au risque de la tronquer. Afin de compter le nombre de chiffres significatifs (partie entière + décimale), un compteur sera utilisé. Dès que ce compteur atteint sa valeur maximale (nombre de bits significatifs fournie au diviseur par l'intermédiaire d'un paramètre générique dans l'entité VHDL), le résultat est prêt et le signal *result_ready* à destination de l'unité de contrôle pourra être validé. Se reporter aux annexes page 7 pour la description VHDL de l'unité de division à virgule fixe.

Nous allons à présent focaliser notre attention sur cette unité de contrôle, aussi appelé séquenceur dans la littérature. En effet son rôle est de produire dans le temps une séquence de signaux de contrôles permettant d'effectuer des commandes choisies sur les blocs qui l'entourent. La manière conventionnelle de décrire le fonctionnement d'un tel circuit est la machine à états finis. Comme nous l'avons abordé précédemment, chaque état de l'automate est représentatif d'une action à l'intérieur du projet. Ceci permet de faciliter la description VHDL de l'automate, ainsi que de simplifier le travail de simulation et de test. Vous trouverez la description VHDL en annexe.

Voici la représentation formelle de la machine à états finis :

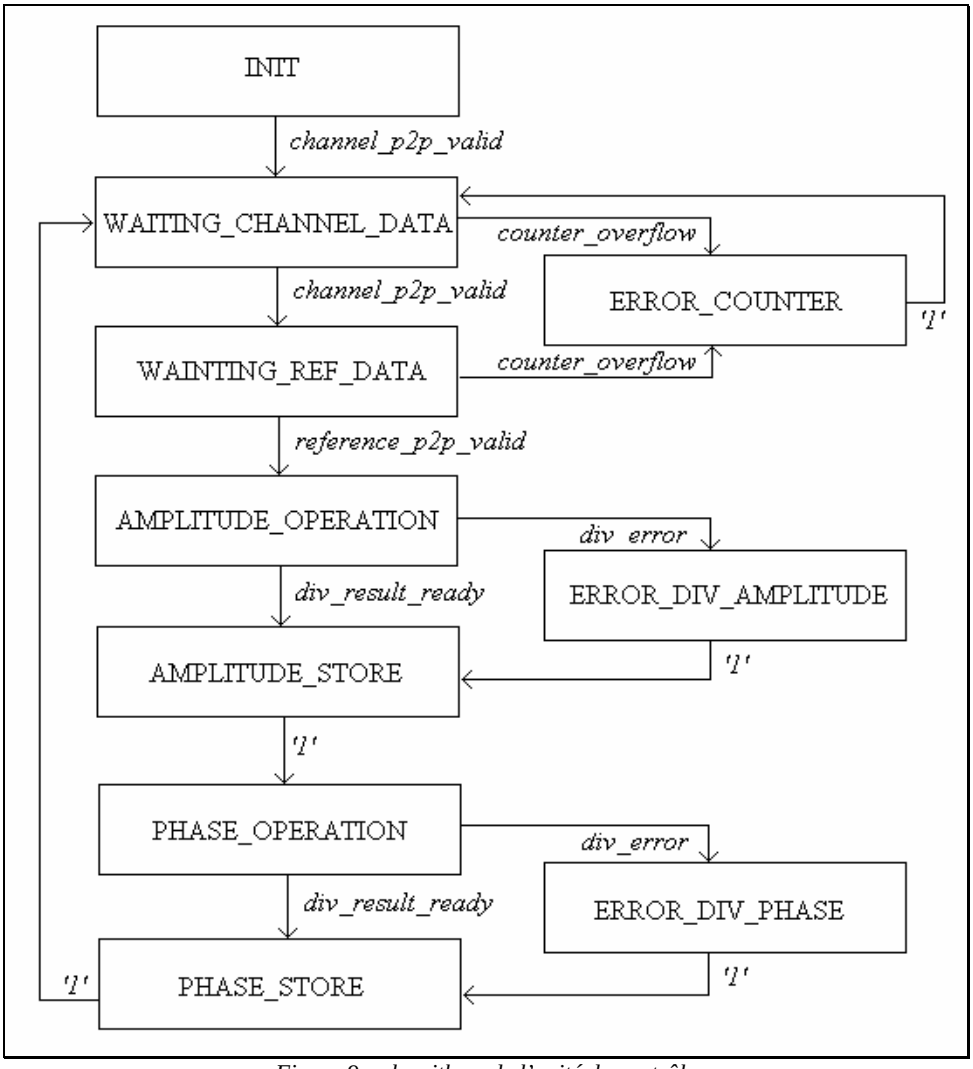


Figure 9 : algorithme de l'unité de contrôle

L'affectation des sorties se fait de manière combinatoire. Il est à noter tout de même qu'un des signaux nécessite une fois de plus de prendre en compte la connexion systématique des horloges de toutes les bascules à un arbre d'horloge. En effet le signal *line_out* (8 bits, codage « onehot » pour activer le registre de stockage en sortie) doit effectuer une rotation vers la gauche dans deux états particulier. Or en VHDL la représentation des états est faite de signaux combinatoires. Un processus possédant l'horloge dans sa liste de sensibilité doit donc être créé afin de connecter un arbre d'horloge à chaque bascule du registre à décalage. Le signal représentant les états durant lesquels la rotation doit s'effectuer sera synthétisé à la manière d'un « clock enable ». Ce signal *line_out* est propagé tel quel en sortie pour activer le bon registre de sortie (8 registres, 4 pour l'amplitude et la phase). La sélection du canal (4 bits, également « onehot ») se fait en ne propageant que les bits impairs de ce signal *line_out*.

Pour simplifier l'utilisation ultérieure du projet, tous les composants possédant des versions variables selon le choix de l'utilisateur (taille des bus, numéro de somme) sont pourvus de paramètres génériques. Ces paramètres pouvant varier selon le choix d'implémentation, ils sont propagés jusqu'à l'entité hiérarchique la plus haute qui est un **package**. Dans ce package, ces choix sont renseignés par l'intermédiaire de constantes. L'utilisateur n'a dès lors plus qu'à définir quelques paramètres clés du design et la synthèse effectue toutes les adaptations automatiquement sur tous les composants du projet. Voici les paramètres génériques en question :

```
package selfTest_amplitude_phase_4_channels_package is
-- global design parameters
  constant running_sum_number: natural := 7; -- choose the RS to process data on
  constant channel_data_width: natural := 40; -- width of the decoder data
  constant reference_data_width: natural := 16; -- width of the reference data
  constant internal_data_width: natural := 16; -- width of the division input/output
  constant number_of_symetric_channels: natural := 4 ; -- number of channels
  constant output_number_decimals     : natural := 6; -- number of fixed point decimals
```

D. Gros plan sur le filtrage

Après une première série de simulations et afin d'augmenter d'un échelon la robustesse du système, il est apparu qu'apporter une immunité au bruit serait un atout. La solution retenue est la plus triviale : implanter un filtrage du signal d'entrée. C'est sur le travail de conception et d'implantation du filtre qu'est consacré ce chapitre. En effet, le temps dédié à sa conception justifie ce gros plan. De plus, il s'agit d'une illustration parfaite de la méthodologie de conception utilisée, puisque nous allons retrouver toutes les étapes décrites dans le chapitre A. de cette partie.

Pour commencer, posons les contraintes que le filtrage doit respecter. Les données arrivent de 4 canaux différents, et n'ont a priori aucune corrélation entre eux. Le système devra donc contenir 4 filtres distincts, un par canal. Ceci implique une contrainte sur la place que devra occuper chaque filtre, et nous verrons que cette contrainte aura un effet sur le type de filtre numérique utilisé. Ensuite, le gabarit devra être de type passe-bas, car on ne veut filtrer que le bruit, qui se trouve (après analyse de Fourier) autour de 2 octaves au dessus de la fréquence appliquée au système. De plus nous voulons garantir un gain unitaire autour de cette fréquence, ce qui suppose une pente de coupure relativement raide. Une dernière recommandation concerne la souplesse du design. En effet la fréquence à appliquer au système n'est pas encore connue avec précision. Pourtant elle est essentielle pour définir un gabarit de filtre. La fréquence de coupure devra donc être ajustable simplement lors de la synthèse. Ceci permettra en outre de rendre plus aisée l'utilisation du projet après mon passage au CERN.

Après avoir défini les contraintes, regardons les solutions qui nous sont offertes. Tout d'abord, il y a la solution du filtre calculant la moyenne sur plusieurs échantillons. Ce type de filtre est en fait équivalent à un filtre à réponse impulsionnelle finie dont tous les coefficients sont égaux. L'étude de cette solution sera donc faite dans le cadre de la deuxième possibilité, les filtres de ce type. Ces filtres semblaient de prime abord la solution idéale au problème, puisque leur implémentation est assez simple du fait de leur faible complexité et leur stabilité inhérente. De plus, Altera offre des générateurs de code VHDL pour ce type de filtres, ce qui aurait permis de gagner du temps de conception. Le choix c'est donc porté en premier sur cette solution, ce qui explique la réalisation d'une étude complète à leur propos. La troisième solution est le filtre à réponse impulsionnelle infinie (IIR) aussi appelé filtre récursif du fait que la sortie d'un tel filtre dépend d'un ou plusieurs échantillons présent en sortie pendant les cycles précédents. Il est donc facile de comprendre que ces filtres sont en proie à de l'instabilité (divergence) assez facilement. Ils permettent pourtant de se rapprocher au plus près des caractéristiques d'un filtre analogique (possédant toujours un chemin de feedback). Ceci est au prix d'une complexité plus grande que les FIR en terme de chemin de données (à cause de la récursivité) mais en occupant nettement moins d'éléments logiques à réponse fréquentielle égale.

C'est dans le design de ces filtres que l'utilisation d'un modèle de référence développé en C aura été le plus utile. En effet, les données d'un signal fortement bruité étant en ma possession, il était simple de réaliser une simulation permettant de valider ou pas le choix du filtre à implémenter. Ceci permettra de gagner du temps de conception en définissant le nombre de coefficients du filtre ainsi que leurs valeurs.

Fort de 5 années d'études comprenant à chaque fois un semestre de traitement du signal, un modèle de référence de filtre FIR développé en C a vite été conçu et compilé :

```
[lxplus063] /afs/cern.ch/user/e/everhage/work/filters > ./fir rsum.txt 5 1,-3,5,-5,3,-1
#
#       FIR filtering - Erik Verhagen
#
#Usage: fir [file] [column] [coefs]     where
         [file]          : text (ascii) file containing the data
         [column]        : data column (eg. specific Running Sum), starting column 1
         [coefs]         : list of comma separated filter coeficients from lowest to highest polynomial degree


#       Data file size : 180060 bytes
#       Number of lines : 3001
#       Number of columnss : 6
#
#
#       number of coeficients = 6
#       y(n) = 1*x(n) + -3*x(n-1) + 5*x(n-2) + -5*x(n-3) + 3*x(n-4) + -1*x(n-5)
#


[lxplus063] /afs/cern.ch/user/e/everhage/work/filters >
```

Figure 10 : interface du modèle de référence de filtre FIR

Cet outil permet d'extraire et visualiser la courbe en sortie d'un filtre de ce type, en choisissant soi-même les coefficients (représentatifs de l'équation aux différences), et en précisant quelles données (colonne d'un fichier) sont à lui appliquer.

Le premier type de filtre essayé a été un filtre moyenneur. Tous les coefficients sont mis à 1, et une moyenne est ainsi calculée sur le nombre d'échantillons correspondant au nombre de coefficients. Le résultat était attendu, la courbe est identique à celle produite par l'étude de l'influence sur le gain et la phase des sommes à fenêtre glissante (principe

identique). Le filtrage offre des résultats médiocres, même avec une moyenne sur 15 échantillons, limite au delà de laquelle l'impacte sur le gain (aplatissement) ne satisfait plus les contraintes fixées au début.

Le deuxième essai se porte sur le filtre FIR classique. Les coefficients sont choisis afin de respecter le principe de causalité, ainsi que la linéarité de phase (symétrie). Contrairement aux attentes, le résultat est peu encourageant. Là aussi, le nombre de coefficients pour s'approcher d'un filtrage correcte devient grand. Ceci offre une limitation de taille car chaque registre formant un monôme de l'équation aux différences est de taille supérieure à 16 bits (dépendamment de la représentation binaire des coefficients car `(A*B)'length` = `A'length` + `B'length`). La quantité de bascules nécessaires pour réaliser ce filtre devient donc grande, d'autant plus que les filtres seront au nombre de 4. Après une réflexion approfondie, cette nécessité d'un grand nombre de coefficients est due au ratio d'échantillonnage. En effet la période de rafraîchissement de la somme choisi (RS_7) est de 2 ms, la période du signal à mesurer dépasse la seconde. Ceci donne un ratio d'échantillonnage de l'ordre de 500. Avec un faible nombre de registres, l'application de l'équation aux différences ne permet donc pas d'amortir une variation brusque d'un seul échantillon. Les filtres FIR sont nettement plus efficaces en présence d'un ratio entre 2 (théorème de Nyquist) et 10.

La dernière catégorie de filtres restant à explorer (IIR) semblait dès à présent être la plus adaptée. Elle est malheureusement aussi la plus difficile à mettre en œuvre, à cause notamment des critères de stabilité. La synthèse d'un tel filtre s'effectue pourtant de manière plus naturelle et intuitive, puisque le point de départ est un filtre analogique. Dans notre cas, on souhaite avoir un gain unitaire à la fréquence appliquée au système, et la pente ne nécessite pas d'être trop raide pour un filtre IIR. Le modèle de filtre sera donc du type Butterworth et l'ordre sera à choisir entre 2 et 3 d'après l'étude du gabarit. La fonction de transfert d'un filtre de Butterworth s'obtient à l'aide d'un polynôme générateur dit « de Butterworth », mais les coefficients du polynôme sont également aisément trouvable dans la littérature de manière tabulée. Le détail du calcul ne sera pas reproduit ici (voir le calcul complet en annexe page 13) :

- La fonction de transfert analogique d'un filtre Butterworth d'ordre 2 est la suivante :

$$H(p) = \frac{1}{1 + \sqrt{2}.\frac{p}{\omega_c} + \left(\frac{p}{\omega_c}\right)^2}$$

- Transformation bilinéaire

$$H(z) = \frac{z^{-2} + 2.z^{-1} + 1}{b1.z^{-2} + b2.z^{-1} + C} = \frac{Y(z)}{X(z)}$$

- Equation aux différences.

$$y(n) = \frac{x(n) + 2.x(n-1) + x(n-2) - b1.y(n-1) - b2.y(n-2)}{C}$$

Sachant à présent l'utilité de l'étude d'un modèle de référence, nous ne nous priverons pas de passer ce filtre au même banc de test. Connaissant l'équation aux différences, un modèle en C a ainsi été écrit et compilé :

```
[lxplus063] /afs/cern.ch/user/e/everhage/work/filters > ./iir_butterworth_2 rsum.txt 5 0.00256 1
#
#       Second order Butterworth IIR filtering - Erik Verhagen
#
#Usage: iir_butterworth_2 [file] [column] [Te] [Fc]     where
         [file]          : text (ascii) file containing the data
         [column]        : data column (eg. specific Running Sum), starting column 1
         [Ts]            : sampling period (seconds)
         [Fc]            : cut off frequency (Hz)


#       Data file size : 180060 bytes
#       Number of lines : 3001
#       Number of columns : 6
#
#
#       Ts = 0.002560   wc = 6.283185
#        y(n) = (1/C)x(n) + (2/C)*x(n-1) + (1/C)*x(n-2) - (b1/C)*y(n-1) - (b2/C)*y(n-2)
#        1 Biquad :
#               b1 = -30918.771217      b2 = 15285.543123       C = 15637.228638
#
#               1/C = 0.000064   b1/C = -1.977254               b2/C = 0.977510
#


[lxplus063] /afs/cern.ch/user/e/everhage/work/filters >
```

Figure 10 : interface du modèle de référence de filtre IIR

Une étude et un modèle identique à été créé pour un filtre Butterworth d'ordre 3 ainsi qu'un Chebyschev d'ordre 2 pour effectuer une comparaison (voire Annexe). Il est apparu qu'un Butterworth d'ordre 2 serait suffisant car moins complexe, moins gourmant en éléments logiques, à condition de prêter une attention toute particulière au choix de la fréquence de coupure. Notons également un déphasage dont il faudra tenir compte ultérieurement.

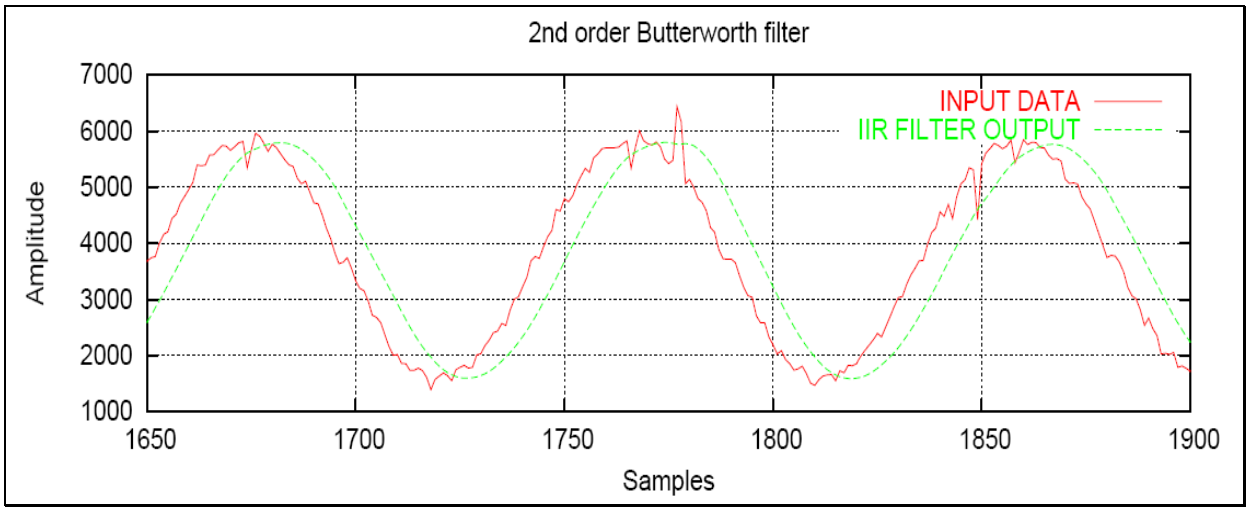


Figure 11 : simulation du modèle de référence d'un filtre IIR Butterworth d'ordre 2

Etudions à présent de manière détaillée la création de la description VHDL de ce filtre, car cette étape n'a pas été triviale, et plutôt gourmande en temps. La difficulté réside principalement en l'utilisation de ressources matérielles présentes dans le FPGA avec le format de données choisie précédemment dans le projet. En effet le FPGA renferme 112 blocs DSP de 9 bits, pouvant réaliser des calculs rapides et en cascade sur des entiers signés. Les coefficients étant des nombres réels, on est face à un problème puisque les outils de synthèse actuels sont incapable d'interpréter ce type de valeurs. Après un travail de documentation très complet, le choix s'est porté sur la résolution du problème à l'aide de l'arithmétique à virgule fixe. Cette

notation permet faire une représentation de nombres réels comme nous l'avons vu au cours du travail réalisé sur le diviseur, mais également d'effectuer des opérations arithmétiques entre ces nombres. Ces opérations sont effectuées en appliquant à chaque nombre un multiplicateur implicite commun permettant de s'affranchir de la position de la virgule. Ainsi le nombre 10,101 par exemple aura comme multiplicateur implicite le nombre 8 (2^3) afin de réaliser toutes les opérations sur celui ci comme si il s'agissait d'un entier. Bien sur, tous les nombres décimaux intervenants dans la chaîne d'opérations de ce nombre devront avoir le même multiplicateur implicite, et une division par 8 pour remettre la virgule à sa place sera indispensable en fin de calcul.

Afin de s'affranchir de la réflexion nécessaire sur tous les chemins de données à propos de cette représentation en virgule fixe, une bibliothèque existe en VHDL. Elle permet de simplifier l'utilisation de cette représentation en créant un type `fixed` contraint du MSB (positif) au LSB (négatif, à droite de la virgule). L'utilisation de celle-ci est permise puisqu'elle fera partie intégrante de la nouvelle mouture de VHDL-200x et certifiée par l'organisme IEEE. Cette bibliothèque est le fruit du travail de *David Bishop* (*dbishop@vhdl.org*), se nomme *fixed_pkg_c.vhd* et est disponible par l'intermédiaire du groupe de réflexion sur le VHDL (*vhdl.org* ou *eda.org*). Outre le nouveau type de vecteur de bits (présent sous la forme signée et non-signée), elle effectue une surcharge de tous les opérateurs arithmétiques classiques, et est entièrement synthétisable. Cette nouvelle librairie a été spécialement développée pour les applications nécessitant de l'arithmétique virgule fixe comme la synthèse de DSP pour le traitement du signal. Une documentation exhaustive est également fournie avec ce package.

Le modèle de référence étant validé, et une solution ayant été trouvée pour palier à cette question de représentation de nombres, la production de la description VHDL du filtre n'a pas été longue. Une simulation de ce circuit a pu être effectuée après apport de la nouvelle bibliothèque dans ModelSim. Le résultat est le suivant :

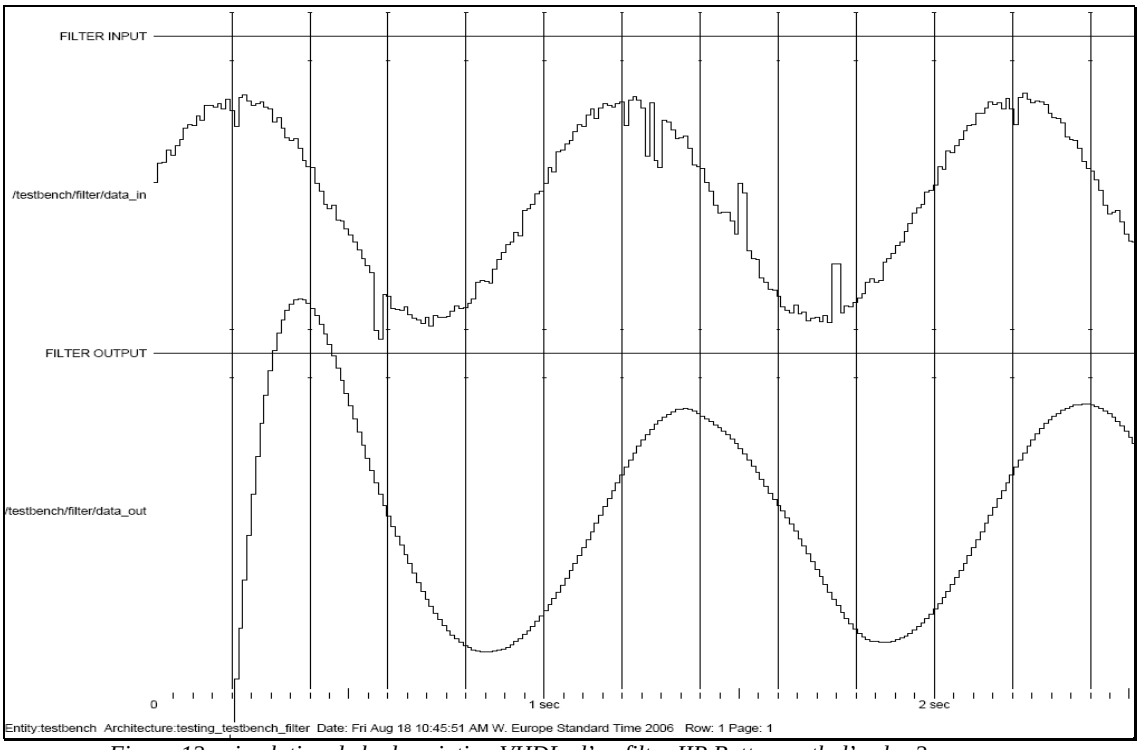


Figure 12 : simulation de la description VHDL d'un filtre IIR Butterworth d'ordre 2

La dernière étape consiste à augmenter la souplesse de notre description, en intégrant les paramètres du filtre comme constantes dans le package du design et qui seront propagées ensuite par l'intermédiaire de paramètres génériques aux filtres. L'utilisateur souhaitant synthétiser le projet n'aura donc plus qu'a renseigner quelques paramètres supplémentaires dans ce fichier, et pourra ainsi générer un circuit sans avoir à ce soucier de la structure de celui-ci. Ces paramètres seront la fréquence appliquée au système et le choix de la somme (nécessaire pour la fréquence d'échantillonnage) déjà renseigné auparavant. Le calcul des coefficients se fait de manière automatique à partir de ces deux informations.

```
-- arithmetic represetation considerations (fixed point) for the filtering
  constant filter_internal_MSB : integer := 21; -- fixed point MSB for data representation
  constant filter_internal_LSB : integer := -14; -- fixed point LSB for data representation
  constant filter_coef_MSB : integer := 1 ; -- fixed point MSB for filter coefficients
  constant filter_coef_LSB : integer := -18 ; -- fixed point LSB for filter coefficients

-- frequency applied to the design (for filter cut off calculation)
  constant F : real := 0.153; -- input Frequency (in Hz)
```

A partir d'ici les constantes servent au calcul automatique des paramètres. Le numéro de la somme par exemple correspond à une case d'un tableau contenant les valeurs de périodes de rafraîchissement de chaque somme. Les coefficients du filtre sont également calculés statiquement à cet endroit. Ceci nous permettra d'effectuer un changement de type de filtre le cas échéant. En effet, seuls les coefficients changent dans l'équation aux différences entre un filtre Butterworth et Chebyshev par exemple.

```
-- LEAVE UNTOUCHED, inherited constants:
  type running_sum_table is array(0 to 11) of real;
  constant RS_updates : running_sum_table := (0.00004, 0.00004, 0.00004, 0.00004, 0.00008,
0.00008, 0.00256, 0.00256, 0.08192, 0.08192, 0.65536, 0.65536); -- Running sums update times
  constant Ts : real := RS_updates(running_sum_number); -- gathering sample period from array

-- Filter parameters
  constant wc : real := (2.0*MATH_PI) * (2.0*F); -- pulsation (and cut off frequency = 2 * F)

  -- Calculus of the common denominator of the filter transfer function
  constant C : real := 1.0 + ((sqrt(2.0)*2.0) / (Ts*wc)) + (4.0 / ((Ts*wc)*(Ts*wc)));

  -- Calculus of the biquad coefficients
  constant b1 : real := 2.0 - (8.0 / ((Ts*wc)*(Ts*wc)));
  constant b2 : real := 1.0 - ((sqrt(2.0)*2.0) / (Ts*wc)) + (4.0 / ((Ts*wc)*(Ts*wc)));

  -- Normalized coefficients (coef / C)
  constant na1 : sfixed (filter_coef_MSB downto filter_coef_LSB)
              := To_sfixed (1.0/C, filter_coef_MSB, filter_coef_LSB);
  constant na2 : sfixed (filter_coef_MSB downto filter_coef_LSB)
              := To_sfixed (2.0/C, filter_coef_MSB, filter_coef_LSB);
  constant na3 : sfixed (filter_coef_MSB downto filter_coef_LSB)
              := To_sfixed (1.0/C, filter_coef_MSB, filter_coef_LSB);
  constant nb1 : sfixed (filter_coef_MSB downto filter_coef_LSB)
              := To_sfixed (b1/C, filter_coef_MSB, filter_coef_LSB);
  constant nb2 : sfixed (filter_coef_MSB downto filter_coef_LSB)
              := To_sfixed (b2/C, filter_coef_MSB, filter_coef_LSB);
```

III) RESULTATS

A. Synthèse

Le premier critère d'appréciation de l'efficacité du travail réalisé est de regarder le résultat de la synthèse logique du design. C'est ce que nous allons faire dans ce chapitre de manière objective, en tentant de trouver les points forts et faibles en parcourant le schéma RTL fourni en annexe.

D'après la documentation du constructeur, le FPGA cible contient plus de 41 000 éléments logiques (macro cellules combinatoires) ainsi que plus de 3 millions de cellules mémoires de type SRAM (permettant de stocker des tables de vérités). En outre, 112 blocs de calcul DSP permettent de réaliser des fonctions mathématiques complexes telles que des FFT ou d'autres transformées courantes en traitement du signal. Seize arbres d'horloge permettent également d'augmenter le design en complexité, ainsi que 822 broches d'entrées et sorties sont présentes. Nous sommes donc en présence d'un des circuits les plus vastes de cette famille de FPGA disponibles chez ce constructeur.

Le design global dans lequel viendra se greffer notre projet est relativement gourmant en place dans sa version complète. Afin de réaliser des tests réels, une version allégée de ce design a du être créée afin de gagner en temps de synthèse (15 minutes contre plus d'une heure précédemment). Ce design fournit le résultat des 12 sommes sur 4 canaux, permettant ainsi de satisfaire les entrées de la fonctionnalité de test réalisée durant ce stage. Vous trouverez en annexe le détail du rapport de synthèse.

On peut dès à présent indiquer que le bloc fonctionnel le plus gourmand en interconnexions est le décodeur d'entrées. Et ceci malgré la présence du paramètre générique qui ne synthétise pourtant que le chemin de données de la somme choisie (`if` combinatoires sur une condition statique). Ensuite le bloc le plus consommateur d'éléments logiques est l'unité de division. Ceci vient essentiellement du soustracteur combinatoire.

En terme de choses pouvant êtres améliorées, citons l'implémentation des extracteurs de valeur crête à crête. En effet, cette description a été fait en VHDL comportemental. Or la synthèse comportementale possède la caractéristique de générer des structures RTL différentes selon l'outil de synthèse, sans être toujours optimales. L'opération de base de cette fonction d'extraction est une soustraction des valeurs maximum et minimum. Dans le cadre du diviseur, un soustracteur combinatoire a été réalisé. Il est donc envisageable de d'effectuer une nouvelle description plus proche du matériel (VHDL structurel) de cette fonction d'extraction en réutilisant le soustracteur.

En terme de bilan d'exécution, pour produire un couple amplitude / phase, le nombre de cycles nécessaires comporte deux fois le nombre de cycles nécessaires pour chaque division, auquel viennent s'ajouter 2 cycles de stockage des valeurs produites (amplitude + phase). Dans notre cas de figure (données sur 16 bits, précision de 6 digits après la virgule), la durée de production d'un couple de valeur est donc de : 2 + 2 * (16 + 6) = 46 cycles d'horloge.

B. Tests réels

Le deuxième critère d'appréciation permettant de porter un jugement sur le travail réalisé est simplement les résultats obtenus lors des tests réalisés dans les conditions réelles de fonctionnement du futur circuit. C'est ce que nous allons détailler dans ce chapitre, en tenant compte du fait que ces résultats sont obtenus au même moment que la rédaction de ce rapport.

La première source d'information vient de l'environnement de développement et de prototypage propriétaire d'Altera, qui porte le nom de **Quartus**. Ce logiciel permet d'utiliser un certain nombre de blocs mémoires à l'intérieur du FPGA pour y stocker les données relatives à certains signaux choisis. Cette fonctionnalité s'appelle le *signal tap*. Lors de la synthèse on peut donc choisir de câbler ces blocs de mémoire pour stocker les états de certains signaux, et lors de l'exécution, cette acquisition peut être lancée à n'importe quel moment. Lorsque les blocs mémoires ont atteint le remplissage défini, une chaîne de type JTAG permet de récupérer ces données et d'en réaliser un chronogramme à l'intérieur de Quartus.

Ce *signal tap* a permis de réaliser la validation fonctionnelle et temporelle des filtres ainsi que de la machine à états finis de l'unité de contrôle. Voici le résultat d'exécution d'un cycle complet de la machine à états. Le codage des états est de type « onehot » pour simplifier l'analyse et l'ordre de leur affichage est respecté (se reporter à la représentation de la machine à états finis page 20).

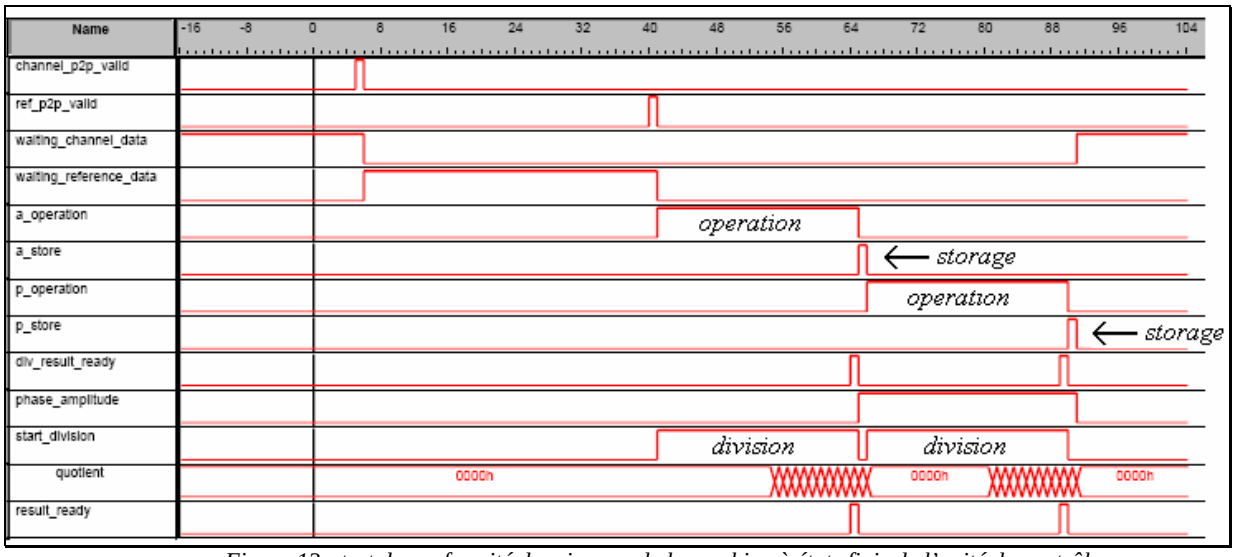


Figure 13 : test de conformité des signaux de la machine à états finis de l'unité de contrôle

Afin de récupérer les données produites par notre système, il a été nécessaire de créer un environnement de test, connecté directement au châssis VME accueillant la carte sur laquelle est présente le projet. Cet environnement a été réalisé à l'aide de **Labview**, un environnement de contrôle industriel, par une personne hautement compétente. Cet environnement tourne sur un PC de contrôle et la communication avec le projet se fait à l'aide d'un lien série. Du coté interne du FPGA, les données sont envoyées dans le registre de sortie (lien série) séquentiellement par le parcours des entrées d'un multiplexeur 32 vers 1 à l'aide d'un compteur. Nos 8 sorties font partie de ces 32 entrées (de l'adresse 6 à 14). Voici une capture d'écran du résultat dans l'environnement **Labview**. Attention, les représentations sont en hexadécimal :

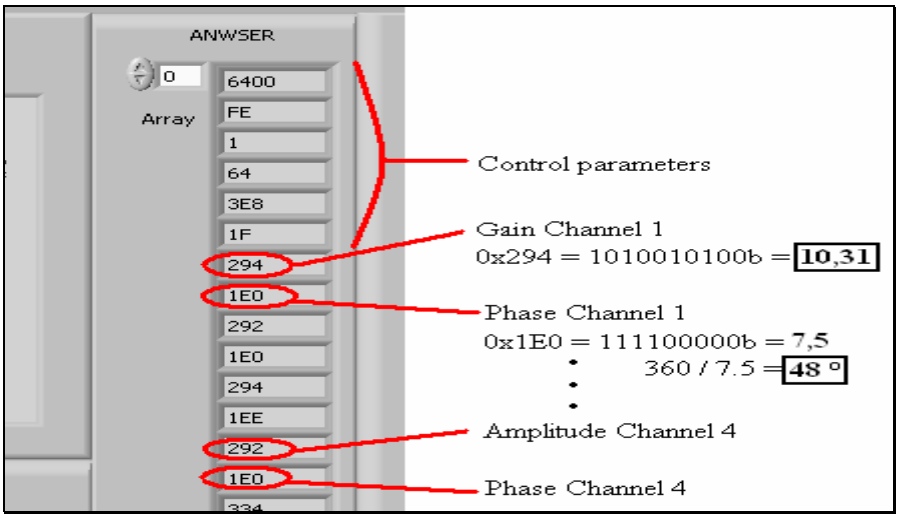


Figure 14 : affichage des sorties du projet en temps réel

Cet environnement de simulation permet également de sauvegarder un grand nombre de données dans un fichier texte. Vous trouverez en annexe (page 14) une courbe représentant les signaux en entrée et en sortie d'un filtre sur les données extraites de cette manière. On constate que le choix du filtre IIR de type Butterworth 2 est conforme aux attentes.

C. Perspectives

Ce dernier chapitre permet de faire une synthèse des choses à améliorer. De plus, comme mon contrat au CERN se prolonge au-delà de la durée imposée par l'université, nous allons faire un tour d'horizon des fonctionnalités soumises à discussion actuellement, et qui seront implémentées en cas de validation.

La première chose à améliorer sera la réécriture de l'extracteur de valeur crête à crête en VHDL structurel. Ceci permettra de mieux maîtriser la synthèse de ce bloc. L'étape suivante sera de réaliser un comparateur de magnitude pour les amplitudes et phases extraites. Cette opération permettra de n'extraire qu'un seul signal indiquant si la chaîne d'acquisition répond encore aux spécifications. Ceci impliquera également une étude approfondie des valeurs tolérées d'amplitude et de phase qui déclencheront une alarme en cas de dépassement. Le design sera alors compatible avec le système global de mesure de pertes du faisceau, dont le but est de définir si l'accélérateur est fonctionnel ou pas en termes de pertes.

Il est également envisageable dans le futur de pousser l'étude plus loin qu'une simple analyse harmonique. En effet, ce design pourrait offrir les bases d'une unité permettant de mesurer d'autres paramètres de la chaîne d'acquisition. En effet, les éléments critiques de la chaîne sont les chambres ionisantes disposées dans le tunnel. Ces chambres sont conçues pour ne pas varier durant toute la durée d'exploitation du LHC. Cependant un incident peut par exemple se produire sur l'une d'elles et notre système sera alors en première ligne pour définir où se situe la défaillance. Ceci permettra d'étendre le nombre de cas de notre générateur de codes d'erreur à d'autres erreurs que celles internes à notre FPGA.

Un moyen d'y parvenir sera par exemple la stimulation de la haute tension par une suite binaire pseudo aléatoire. L'étude du système se fera alors par une analyse de Fourier, ou par une fonction d'intercorrélation afin de trouver quel paramètre de la chaîne produit une influence majeure sur le signal et ainsi trouver de quel élément provient la panne.

CONCLUSION

L'objectif de ce rapport de stage était d'offrir une restitution fidèle du travail effectué durant ma période de stage au sein de l'Organisation Européenne de Recherche Nucléaire (CERN). Nous avons en effet abordé l'ensemble des activités exercées durant ces quelques mois. Une description détaillée de tout le travail n'étant cependant pas possible, nous nous sommes efforcés d'attirer l'attention sur quelques points clés permettant d'apprécier le travail de conception qu'il est demandé dans le cadre de la formation de **Conception de Systèmes intègres Numérique et Analogiques**. D'autres points moins significatifs ont donc volontairement étés omis afin de rendre la lecture de ce rapport plus agréable. La méthodologie par contre a été scrupuleusement restituée, puisqu'elle consiste à mon sens le point fort du travail réalisé durant ce stage. En effet, l'acquisition d'une organisation méthodique de travail a permis de repousser la plupart des difficultés rencontrées.

Comme j'en ai fait part à mon enseignant tuteur **Olivier Rossetto** au cours d'une communication ayant pour but le suivie du déroulement du stage, cette expérience au CERN n'a pas seulement eu pour résultat de conforter mes connaissances en conception de systèmes intégrés. En effet, comme le lecteur a pu l'apprécier tout au long de ce rapport, une large quantité de notions ont été mises à profit pour atteindre l'objectif. Ces notions dépassent le cadre de la formation de CSINA, et ont également faites appel à des connaissances acquises durant l'ensemble des 5 années d'études que compte à présent mon cursus universitaire.

C'est justement sur ce point que ce stage au CERN est une double réussite. Il a non seulement permis de mettre en pratique les connaissance de cette dernière année de spécialisation en microélectronique, mais a également permis de passer en revue toutes les connaissances acquises durant 5 ans, et de prendre conscience à titre personnel de la polyvalence qu'elles m'ont apportées. Cette expérience m'aura donc permis de cibler mes attentes futures sur le plan professionnel et faire un état de mes compétences actuelles à mettre en avant. Pour finir, les domaines de recherche abordées au CERN sont des plus captivantes, ce qui a permis un enrichissement personnel de mes connaissances à propos de la physique fondamentale.

Erik Verhagen
Master 2 Pro CSINA
Université Joseph Fourier / INP Grenoble

ANNEXES

INFORMATIONS ADMINISTRATIVES

Organisme :

CERN – Organisation Européenne pour la Recherche Nucléaire –

Activité :

Laboratoire de recherche fournissant des outils aux scientifiques du monde entier pour la recherche en physique des particules et des hautes énergies.

Service :

Beam Loss Monitoring : Supervision des pertes du faisceau. Equipe en charge de la conception des systèmes de contrôle de dépassement de niveau des pertes de faisceau à l'extérieur du tube, et en charge du déclenchement de l'extraction en cas perte de contrôle.

Mes coordonnées au CERN :

Adresse : Erik Verhagen division : AB – BI – BL
CH 1211 Geneva 23
Switzerland

Telephone : +41 22 767 0239

Mail : Erik.Verhagen@cern.ch

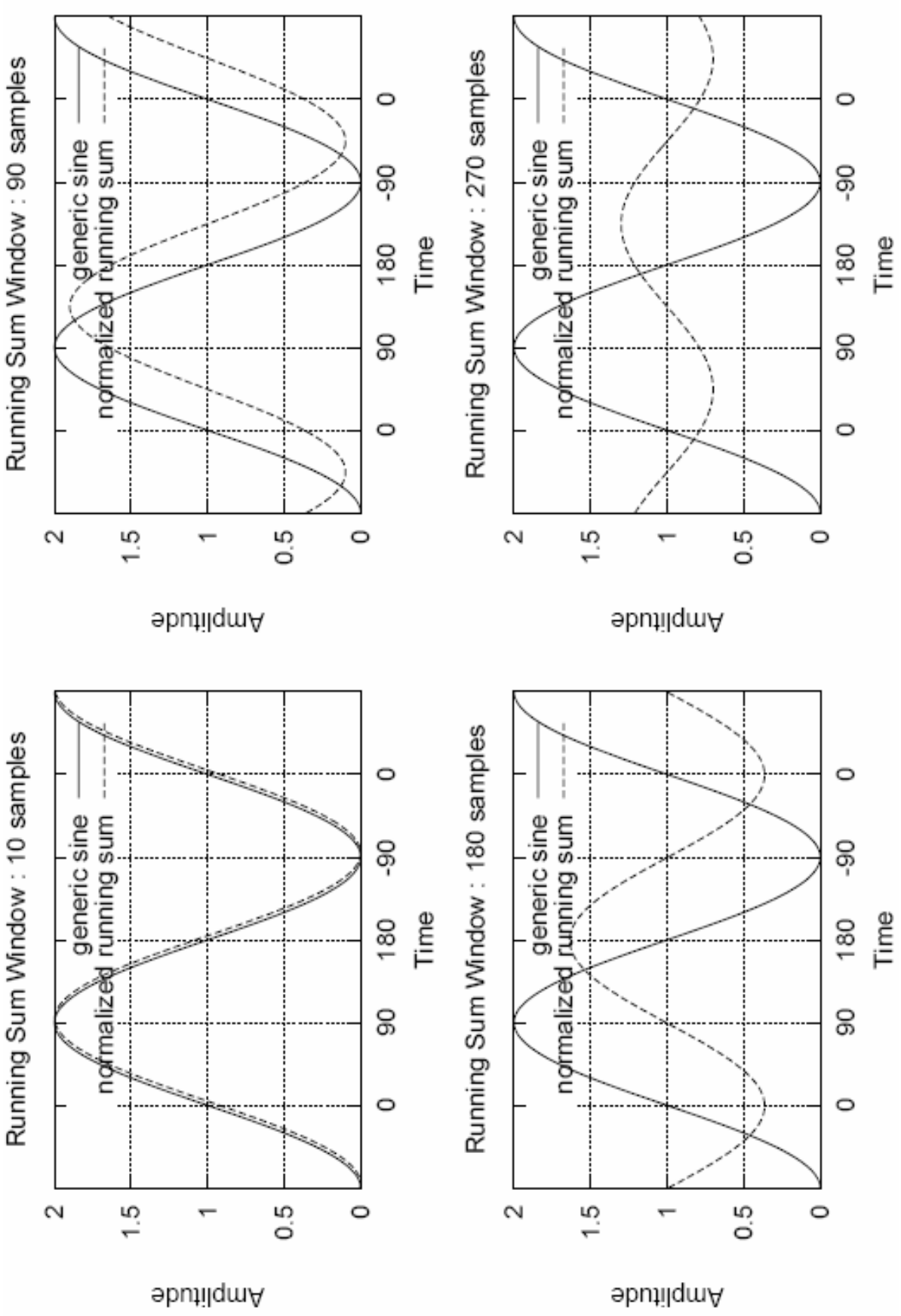
Coordonnées de mon superviseur :

Adresse : Bernd Derning division : AB – BI – BL
CH 1211 Geneva 23
Switzerland

Telephone : +41 22 767 5541
Fax : +41 22 767 9560

Mail : Bernd.Dehning@cern.ch

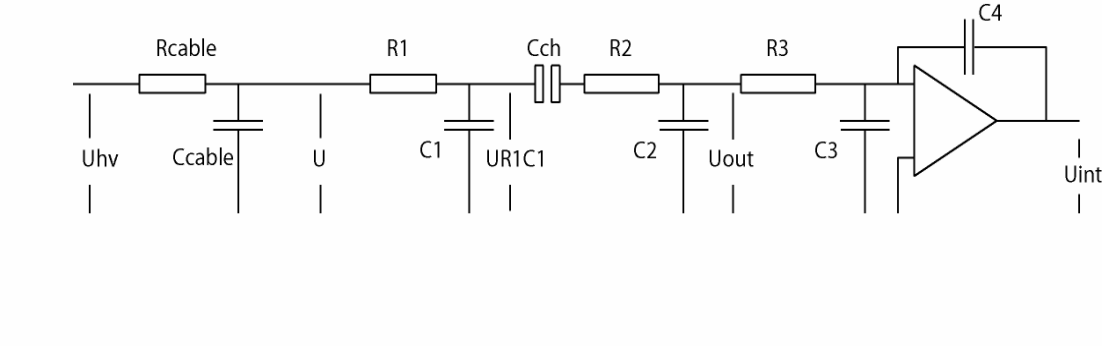
ETUDE DE L'INFLUENCE DES SOMMES A FENETRE GLISSANTE



FONCTION DE TRANSFERT DE LA CHAINE D'ACQUISITION

Date: 08.02.06

BLM: noise source and CFC input circuit



$U_{hv} := 1$ $\omega := 0.001, 0.004 .. 100$

$R_{cable} := 110$ $R_1 := 10 \cdot 10^6$ $R_2 := 470$ $R_3 := 2200$ $C_3 := 0.47 \cdot 10^{-9}$

$C_{cable} := 125 \cdot 10^{-9} \cdot 0.5$ $C_1 := 470 \cdot 10^{-9}$ $C_{ch} := 312 \cdot 10^{-12}$ $C_2 := 4.7 \cdot 10^{-9}$ $C_4 := 100 \times 10^{-12}$

$$Z_{C2R3}(\omega) := \frac{1}{\frac{1}{R_3} + \frac{1}{\frac{-i}{\omega \cdot C_2}}}$$

$$Z_{CchR2}(\omega) := \frac{-i}{\omega \cdot C_{ch}} + R_2 + Z_{C2R3}(\omega)$$

$$U(\omega) := \frac{U_{hv}}{R_{cable} + Z_{cable}(\omega)} \cdot Z_{cable}(\omega)$$

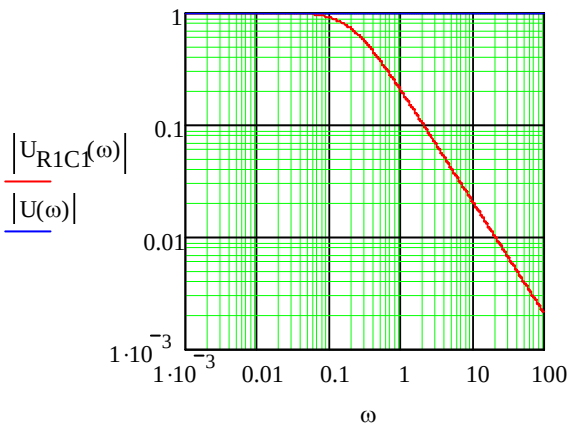
$$Z_{C1Cch}(\omega) := \frac{1}{\frac{1}{\frac{-i}{\omega \cdot C_1}} + \frac{1}{Z_{CchR2}(\omega)}}$$

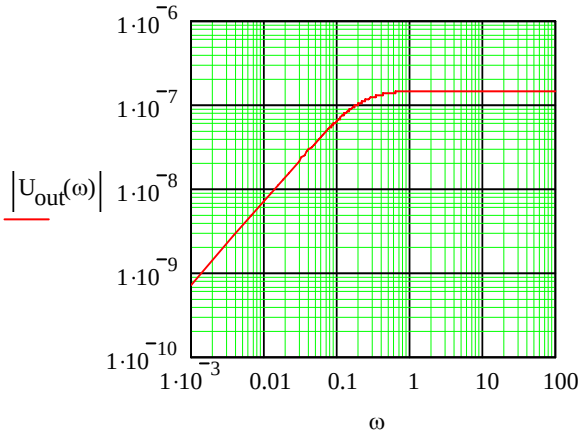
$$Z_{cable}(\omega) := \frac{1}{\frac{1}{\frac{-i}{\omega \cdot C_{cable}}} + \frac{1}{Z_{R1}(\omega)}}$$

$$Z_{R1}(\omega) := Z_{C1Cch}(\omega) + R_1$$

$$U_{R1C1}(\omega) := \frac{U(\omega)}{R_1 + Z_{C1Cch}(\omega)} \cdot Z_{C1Cch}(\omega)$$

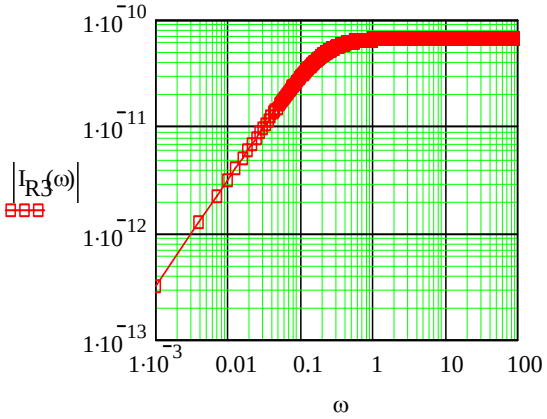
$$U_{out}(\omega) := \frac{U_{R1C1}(\omega)}{Z_{CchR2}(\omega)} \cdot Z_{C2R3}(\omega)$$

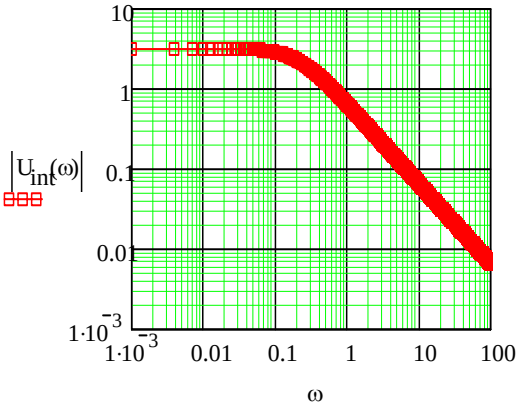




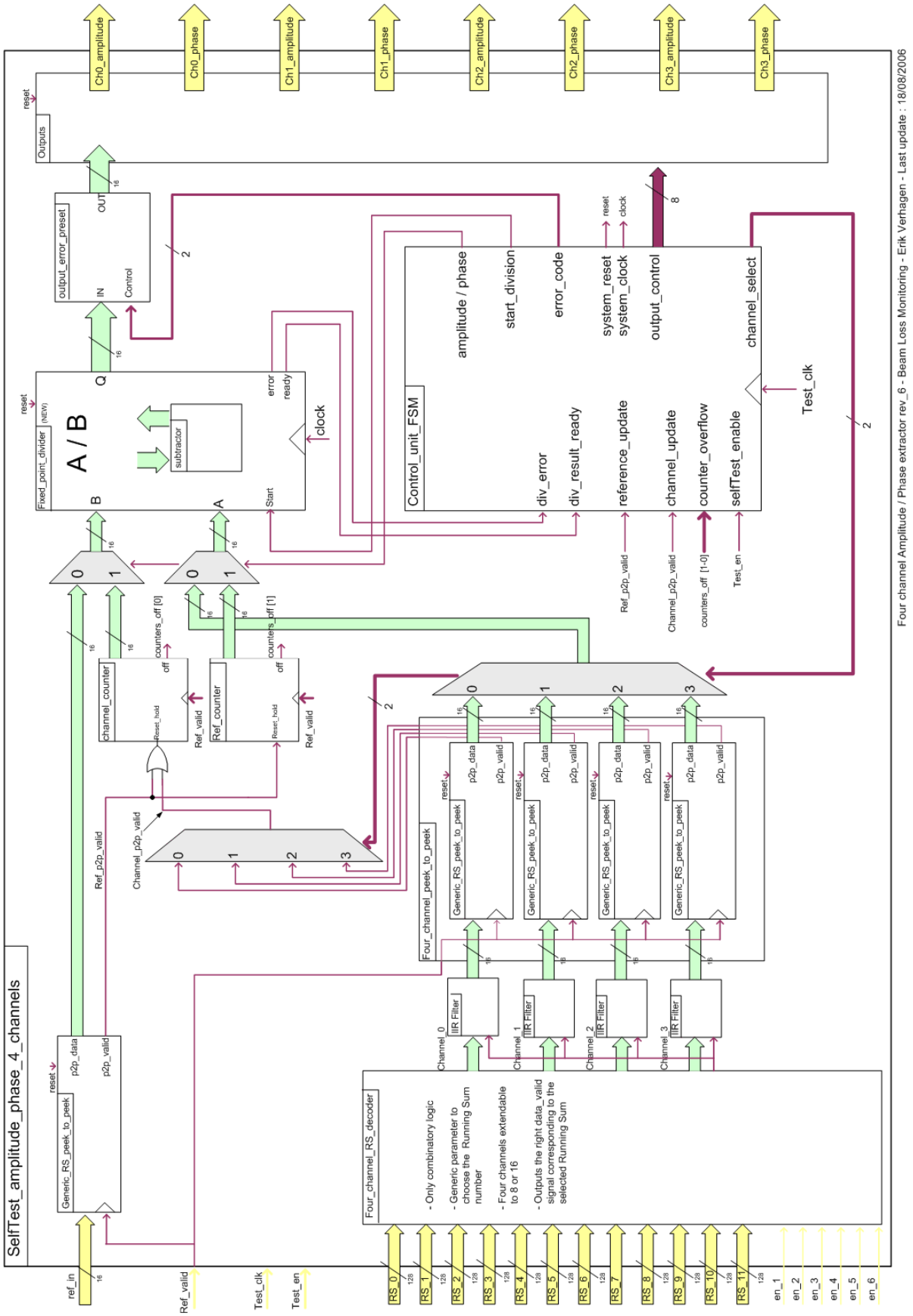
$$I_{R3}(\omega) := \frac{U_{out}(\omega)}{R_3}$$

$$U_{int}(\omega) := I_{R3}(\omega) \cdot \frac{1}{C_4 \cdot \omega}$$





SCHEMA RTL DU DESIGN



CODE VHDL DU L'UNITE DE DIVISION

```
--------------------------------------------------------------------------------
-- File :      divider_unit.vhd
-- Description:  arithmetical divider
-- Project :    Beam Loss Monitoring Self-test functionality
-- Author :     Erik Verhagen
-- For :       Beam Loss Monitoring section, AB-BI-BL
-- Updated :    31 August 2006
--------------------------------------------------------------------------------
-- NOTES:
--
--
--
--
--
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164. all ;
use IEEE.std_logic_arith. all ;
use IEEE.std_logic_unsigned. all ;

entity divider_unit is
    generic (       width              : natural range 0 to 40;
                    nb_decimals : natural range 0 to 40);
    port     (      clock              : in std_logic ;
                    dividend           : in std_logic_vector (width - 1 downto 0);
                    divider            : in std_logic_vector (width - 1 downto 0);
                    division_start     : in std_logic ;
                    quotient           : out std_logic_vector (width - 1 downto 0);
                    result_ready       : out std_logic ;
                    overflow           : out std_logic
                );
end entity ;


architecture testing of divider_unit is
    component sub
        generic (width : natural range 0 to 40);
        port (
                    a:          in std_logic_vector (width - 1 downto 0);
                    b:          in std_logic_vector (width - 1 downto 0);
                    result: out std_logic_vector (width downto 0)
            );
    end component ;

    signal dividend_register : std_logic_vector (2*width - 1 downto 0);
    signal sub_operand_1 : std_logic_vector (width - 1 downto 0);
    signal subtract_result_register : std_logic_vector (width downto 0);
    signal quotient_shift_register : std_logic_vector (width - 1 downto 0);
begin

    SUBTRACTOR : sub
    generic map (width => width)
    port map ( a => sub_operand_1,
                        b => divider,
                        result => subtract_result_register
                );

    process (clock, divider, dividend, division_start)
        variable cpt : natural range 0 to 40;
    begin
        if clock'event AND clock ='1' then
            if division_start = '1' then
                quotient_shift_register (width - 1 downto 1) <= shl(quotient_shift_register(width - 2 downto 0),"0");
                quotient_shift_register (0) <= not subtract_result_register (width);
                sub_operand_1 (0) <= dividend_register(width - 1);
                dividend_register <= shl(dividend_register, "1");
                if subtract_result_register(width) = '0' then                                   -- subtract success
                    sub_operand_1 (width - 1 downto 1) <= subtract_result_register (width - 2 downto 0);
                else                                                                                    -- subtract fail
                    sub_operand_1 (width - 1 downto 1) <= shl(sub_operand_1 (width - 2 downto 0),"0");
                end if ;

                if cpt = width + nb_decimals then
                    result_ready <= '1';
                    cpt := 0;
                else
                    result_ready <= '0';
```

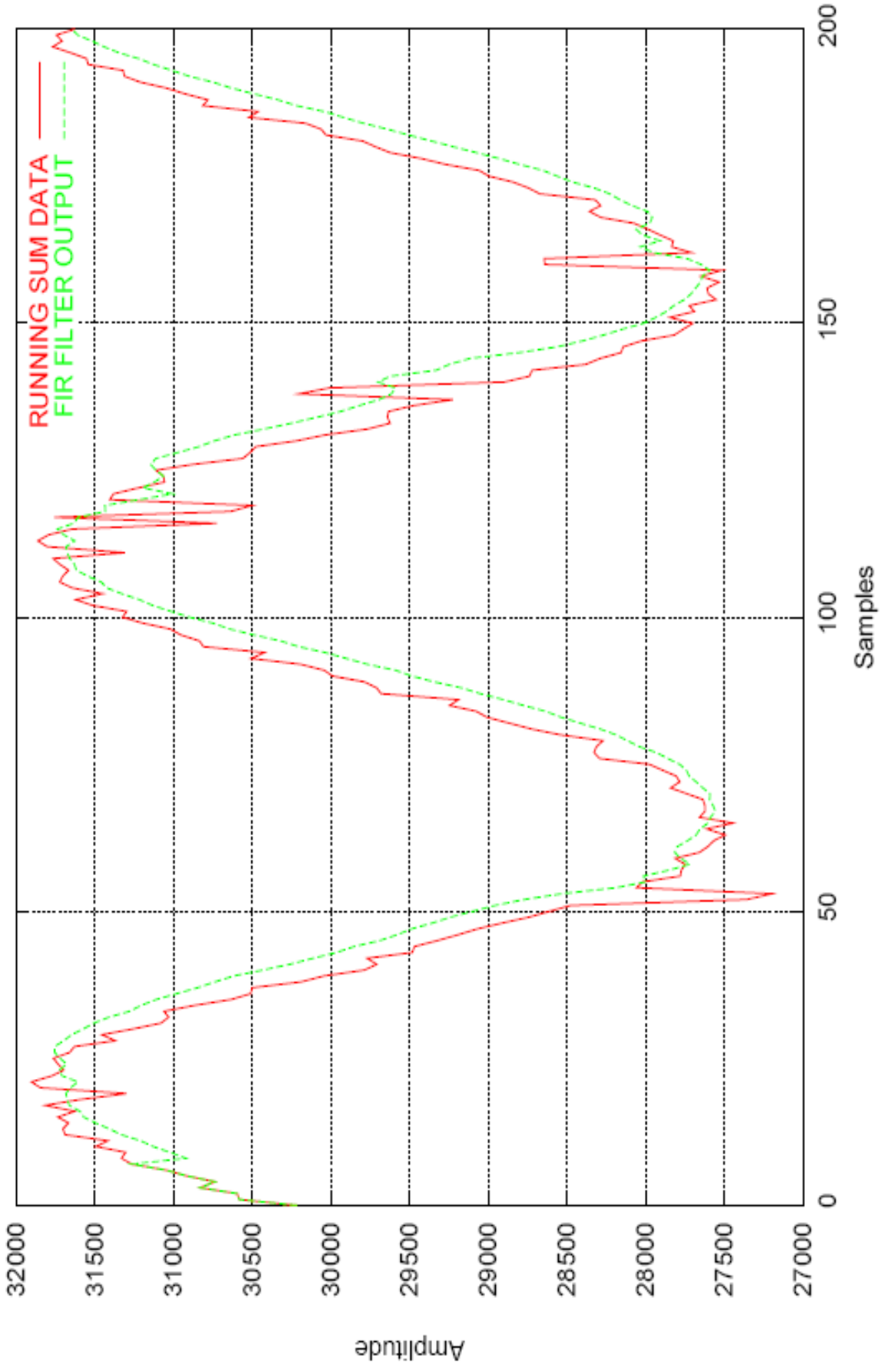
```
                cpt := cpt + 1;
            end if ;

            else -- asynchronus reset
                result_ready <= '0';
                dividend_register <= conv_std_logic_vector(0, width) & dividend;
                sub_operand_1 <= conv_std_logic_vector(0, width);
                quotient_shift_register <= conv_std_logic_vector(0, width);
                cpt := 0;
            end if ;
        end if ;
    end process ;


    test_overflow : process (quotient_shift_register)
    begin
        for i in 1 to (width - 1) loop
            overflow <= quotient_shift_register(i-1) AND quotient_shift_register(i);
        end loop ;
    end process ;

    quotient <= quotient_shift_register;
end architecture ;
```

CODE VHDL DE L'UNITE DE CONTROLE

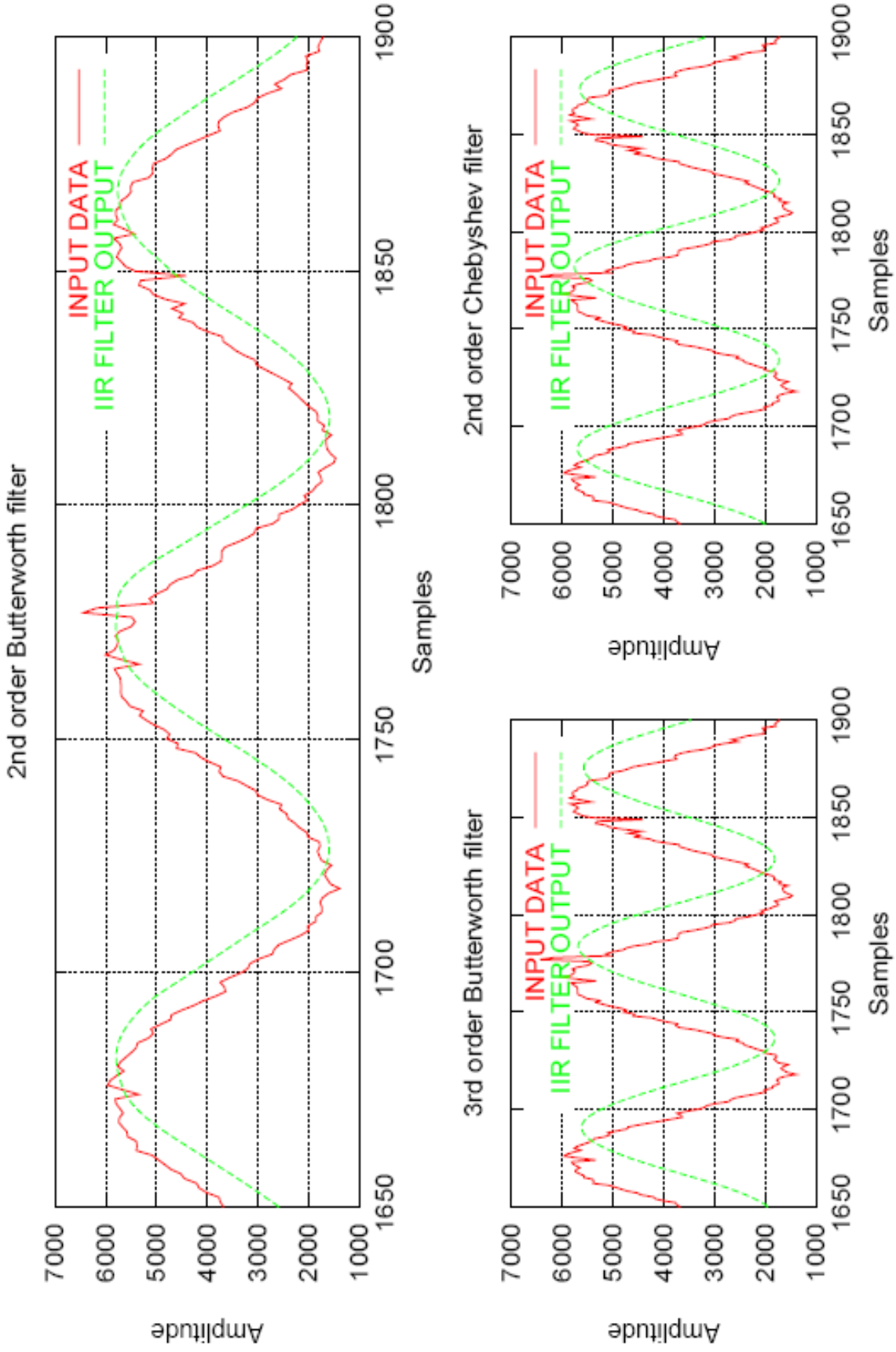
```
--------------------------------------------------------------------------------
-- File :      control_unit_FSM.vhd
-- Description:  Main control unit
-- Project :    Beam Loss Monitoring Self-test functionality
-- Author :     Erik Verhagen
-- For :       Beam Loss Monitoring section, AB-BI-BL
-- Updated :    30 August 2006
--------------------------------------------------------------------------------
-- NOTES:
--
--
--
--
--
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164. all ;
use IEEE.std_logic_arith. all ;

entity control_unit_FSM is
    generic (       nb_channels             : natural range 1 to 16 := 4 );
    port      (     clock                   : in std_logic ;
                    selfTest_enable         : in std_logic ;
                    ref_p2p_valid           : in std_logic ;
                    channel_p2p_valid       : in std_logic ;
                    counter_overflow_flag   : in std_logic_vector (1 downto 0);
                    div_error               : in std_logic ;
                    div_result_ready        : in std_logic ;
                    system_clock            : out std_logic ;
                    system_reset            : out std_logic ;
                    start_division          : out std_logic ;
                    phase_amplitude         : out std_logic ;
                    error_code              : out std_logic_vector (1 downto 0) ;
                    channel_select          : out std_logic_vector (3 downto 0) ;
                    output_control          : out std_logic_vector ((2 * nb_channels) - 1      downto 0)
                );
end entity ;

architecture valid of control_unit_FSM is

 type t_state is (init, waiting_channel_data, waiting_reference_data, A_operation, A_store, P_operation, P_store, division_error_amplitude, divisi
on_error_phase, counter_error);
    signal current_state, next_state      : t_state := init;
    signal channel_out                    : bit_vector (nb_channels - 1 downto 0);
    signal line_out                       : bit_vector (2 * nb_channels - 1 downto 0);

begin

    transitions : process (current_state, ref_p2p_valid, counter_overflow_flag, channel_p2p_valid, div_error, div_result_ready)

    begin
        case current_state is
            when init =>
                if channel_p2p_valid = '1' then
                    next_state <= waiting_channel_data;
                else
                    next_state <= init;
                end if ;

            when waiting_channel_data =>
                if channel_p2p_valid = '1' then
                    next_state <= waiting_reference_data;
                elsif counter_overflow_flag(0) = '1' then
                    next_state <= counter_error;
                else
                    next_state <= waiting_channel_data;
                end if;

            when waiting_reference_data =>
                if ref_p2p_valid = '1' then
                    next_state <= A_operation;
                elsif counter_overflow_flag(1) = '1' then
                    next_state <= counter_error;
                else
                    next_state <= waiting_reference_data;
                end if ;
```

```
            when counter_error =>
                next_state <= waiting_channel_data;

            when A_operation =>
                if div_result_ready = '1' then
                    next_state <= A_store;
                elsif div_error = '1' then
                    next_state <= division_error_amplitude;
                else
                    next_state <= A_operation;
                end if ;

            when division_error_amplitude =>
                next_state <= A_store;

            when A_store =>
                next_state <= P_operation;

            when P_operation =>
                if div_result_ready = '1' then
                    next_state <= P_store;
                elsif div_error = '1' then
                    next_state <= division_error_phase;
                else
                    next_state <= P_operation;
                end if ;

            when division_error_phase =>
                next_state <= P_store;

            when P_store =>
                next_state <= waiting_channel_data;


        end case ;
    end process ;


    PROCEED_STATE :  process (clock, selfTest_enable)
    begin
        if selfTest_enable = '0' then
            current_state <= init;
        elsif clock'event AND clock = '1' then
            current_state <= next_state;
        end if ;
    end process ;


    CHANNEL_SELECTION : for i in 0 to (nb_channels - 1) generate
        channel_out(i) <= line_out(2 * i);
    end generate ;


    INCREMENT_LINE : process (current_state, line_out, clock)
    begin
        if clock'event AND clock = '1' then
            if current_state = A_store OR current_state = P_store then
                line_out <= line_out rol 1;
            elsif current_state = counter_error then
                line_out <= line_out rol 2;
            elsif current_state = init then
                line_out <= to_bitvector(conv_std_logic_vector(1, 2*nb_channels));
            end if ;
        end if ;
    end process ;

    system_clock <= clock AND selfTest_enable;
    system_reset <= not selfTest_enable;

    start_division <= '1' when current_state = A_operation OR current_state = P_operation else '0';
    phase_amplitude <= '1' when current_state = A_store OR current_state = P_operation OR current_state = P_store else '0';

error_code <= "01" when current_state = counter_error else "10" when current_state = division_error_amplitude OR current_state = division_e
rror_phase else "00";

channel_select <= To_stdlogicvector(channel_out);

output_control <= To_stdlogicvector(line_out) when next_state = A_store OR next_state = P_store OR current_state = counter_error OR curren
t_state = division_error_amplitude OR current_state = division_error_phase else conv_std_logic_vector(0, 2 * nb_channels);

end architecture ;
```

SIMULATION D'UN FILTRE RIF



COMPARAISON DE 3 FILTRES IIR



SYNTHESE D'UN FILTRE BUTTERWORTH D'ORDRE 2

- Forme canonique :

$H(p) = [B(p)]^{-1}$ avec $B(p)$ le polynôme générateur de Butterworth

Dans notre cas il s'agit d'un second ordre, la fonction de transfert analogique s'écrit donc :

$$\boxed{H(p) = \frac{1}{1 + \sqrt{2}.\frac{p}{\omega_c} + \left(\frac{p}{\omega_c}\right)^2}}$$

- Pôles et zéros :
  - Ce filtre ne possède pas de zéros analogique
  - Ce filtre possède 2 pôles analogiques :

$$\Delta = \frac{2}{{\omega_c}^2} - \frac{4}{{\omega_c}^2} = -\frac{2}{{\omega_c}^2} = \left(j.\frac{\sqrt{2}}{\omega_c}\right)^2 \quad \Rightarrow \quad \boxed{p_{1,2} = \frac{\frac{\sqrt{2}}{\omega_c} \pm j.\frac{\sqrt{2}}{\omega_c}}{2} = -\frac{\sqrt{2}}{2.\omega_c}.(1 \pm j)}$$

- Stabilité :

$\Re(p_{1,2}) = -\frac{\sqrt{2}}{2.\omega_c} < 0, \quad \forall \omega_c \Rightarrow$ Le filtre est toujours stable

- Transformation bilinéaire :

$$\boxed{p = \frac{2}{Ts}.\frac{1 - z^{-1}}{1 + z^{-1}}}$$

$$H(z) = \frac{1}{\left[\frac{2}{Ts.\omega_c}.\frac{1-z^{-1}}{1+z^{-1}}\right]^2 + \frac{\sqrt{2}}{\omega_c}.\left[\frac{2}{Ts}.\frac{1-z^{-1}}{1+z^{-1}}\right] + 1} = \frac{\left(1+z^{-1}\right)^2}{\left[\frac{2}{Ts.\omega_c}.\left(1-z^{-1}\right)\right]^2 + \frac{\sqrt{2}}{\omega_c}.\left[\frac{2}{Ts}.\left(1-z^{-1}\right).\left(1+z^{-1}\right)\right] + \left(1+z^{-1}\right)^2}$$

$$H(z) = \frac{1 + 2.z^{-1} + z^{-2}}{\left[\frac{2}{Ts.\omega_c}\right]^2.\left(1 - 2.z^{-1} + z^{-2}\right) + \left[\frac{2.\sqrt{2}}{Ts.\omega_c}\right].\left(1 - z^{-2}\right) + \left(1 + 2.z^{-1} + z^{-2}\right)} = \frac{z^{-2} + 2.z^{-1} + 1}{b1.z^{-2} + b2.z^{-1} + C} = \frac{Y(z)}{X(z)}$$

avec $\boxed{b1 = \left(\frac{2}{Ts.\omega_c}\right)^2 - \frac{2.\sqrt{2}}{Ts.\omega_c} + 1}$ $\boxed{b2 = -\left(\frac{2.\sqrt{2}}{Ts.\omega_c}\right)^2 + 2}$ $\boxed{C = \left(\frac{2}{Ts.\omega_c}\right)^2 + \frac{2.\sqrt{2}}{Ts.\omega_c} + 1}$

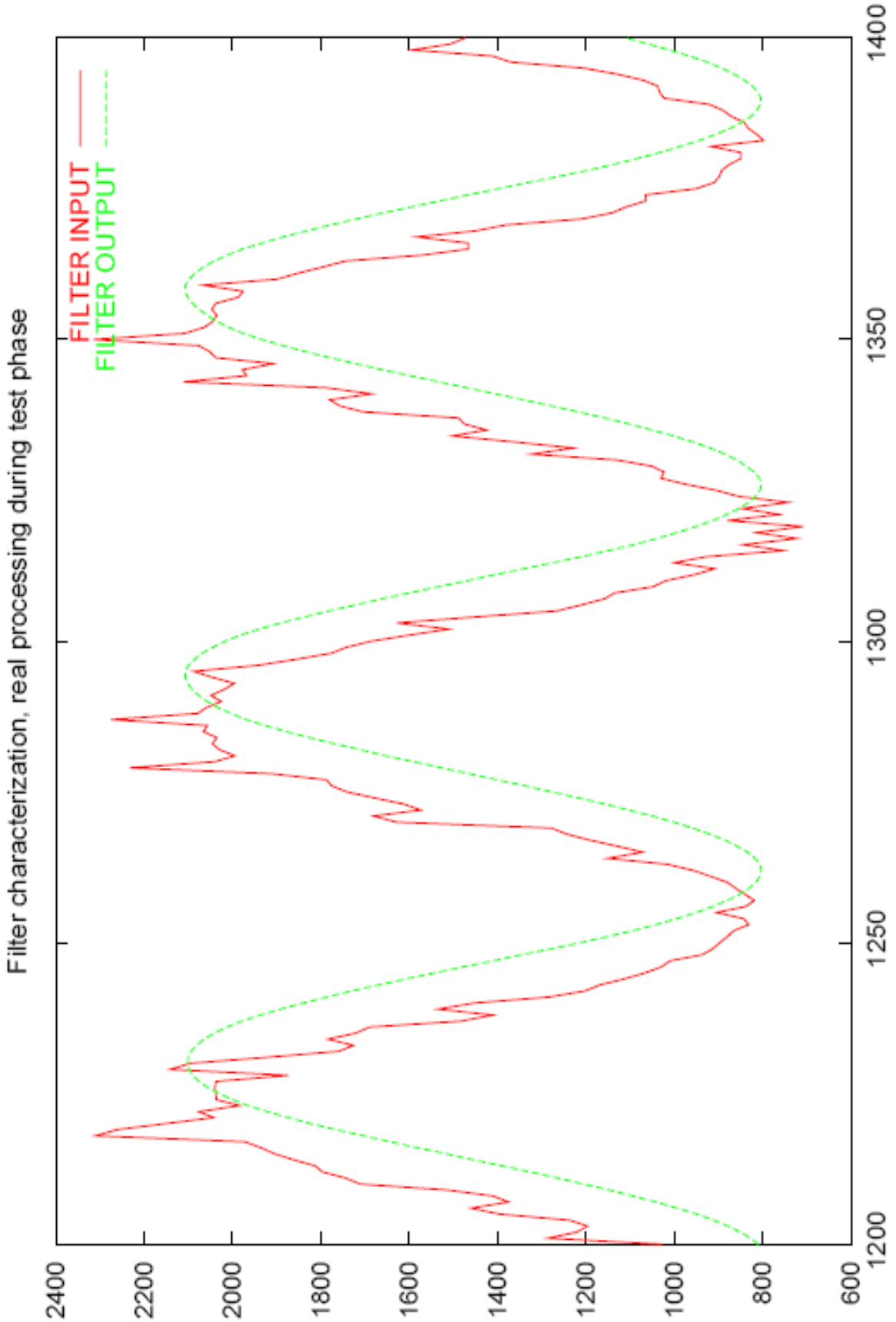
- Equivalence numérique :

$$X(z).\left[z^{-2} + 2.z^{-1} + 1\right] = Y(z).\left[b1.z^{-2} + b2.z^{-1} + C\right]$$

$\Rightarrow$

$$\boxed{y(n) = \frac{x(n) + 2.x(n-1) + x(n-2) - b1.y(n-1) - b2.y(n-2)}{C}}$$

TEST DU FILTRE IIR, DONNEES ISSUES D'UN SIGNAL TAP



RAPPORT DE SYNTHESE DE LA CONFIGURATION DE TEST IMPLEMENTEE

Flow Status	Successful - Mon Sep 04 09:15:20 2006
Quartus II Version	5.1 Build 216 03/06/2006 SP 2 SJ Full Version
Revision Name	COMBINER_TEST
Top-level Entity Name	BLMTC_FULL
Family	Stratix
Device	EP1S40F780C7
Timing Models	Final
Met timing requirements	No
Total logic elements	5,922 / 41,250 (14 %)
Total pins	559 / 616 (91 %)
Total virtual pins	0
Total memory bits	98,180 / 3,423,744 (3 %)
DSP block 9-bit elements	17 / 112 (15 %)
Total PLLs	2 / 6 (33 %)
Total DLLs	0 / 2 (0 %)