CHAPITRE 4

ANALYSE DE LA QUANTIFICATION SOUS SIMULINK

De tous les éléments autour de lui, le seul élément que l'humain ne peut modifier à sa guise, est le temps. À travers les différentes évolutions technologiques, le défi pour appliquer les technologies a toujours été l'aspect temporel. Dans les simulations, l'aspect temps, au départ, est plutôt mis de coté. Les travaux portent plus sur les performances et l'efficacité du modèle. Mais, pour mettre en marché un produit, il est important de tenir compte de l'influence du temps sur le processus. Le temps réel, par définition, c'est l'étude d'un matériel ou d'un logiciel lorsque ce dernier est soumis à une contrainte de temps, c'est-à-dire que l'opération a des limites de temps pour ses sous-opérations ou processus**[21]**. Même si l'on désire souvent une réponse rapide ou un système performant, il n'est pas essentiel dans certains cas.

Il y a deux catégories de système en temps réel : des systèmes dits critiques et des systèmes dits non critiques. Un système non critique a peu de contraintes temporelles ou ces dernières sont relativement larges. Par exemple, un système de gestion des places dans un avion peut réagir à l'intérieur de quelques secondes sans toutefois causer un problème critique dans le système de réservation des places. Dans le cas des systèmes critiques, il est impératif que le processus réponde rapidement à la donnée entrante. Dans cette catégorie, on peut inclure, par exemple, le cas des contrôleurs d'une chaîne de montage dans une usine ou un électrocardiogramme.

Dans notre cas, il est essentiel que notre récepteur réponde dans les plus brefs délais à cause du mouvement des satellites et du mobile. Le récepteur en mouvement doit connaître sa position de manière instantanée, sinon l'information s'avère peu utile et même désuète, ce qui peut entraîner des erreurs graves (dans le cas, par exemple, de l'atterrissage d'un avion). Il s'agit donc d'un système critique.

Pour assurer le passage efficace entre la partie logicielle et la partie matérielle, plusieurs méthodes ont été développées. La toute première, et la plus connue, est la technique MASCOT. Les concepteurs de cette technique ont été à l'encontre des méthodologies existantes à cette époque. Les méthodologies existantes se développaient autour des aspects fonctionnels du projet, en développant un langage rigoureux et structuré. À l'inverse, le principe MASCOT met l'emphase sur l'aspect architectural du logiciel. Ainsi, les efforts se concentraient sur le contrôle en temps réel des processus et des interfaces entre les processus du modèle.

4.1 Principes de base pour l'implémentation en temps réel avec Matlab/Simulink

La méthodologie développée dans le cadre de notre projet se base, non seulement sur les méthodes développées pour le passage en temps réel tel MASCOT, mais aussi sur le concept SDR tel qu'expliqué à la section 2.1. Au CHAPITRE 3, l'architecture du récepteur numérique a été présentée, tel que la méthodologie MASCOT le propose. Pour le passage en temps réel, nous devons tenir compte de certains paramètres, tout comme dans le cas des simulations. La fréquence d'échantillonnage reste le paramètre de base qui contrôle la majorité des fonctions. Dans les librairies de Xilinx, à l'intérieur de Simulink, les blocs définis par les programmeurs ont tous des paramètres communs. Ces paramètres sont :

1. Entrée de remise à zéro
2. Port de contrôle
3. Utilisation des variables « *double* »
4. Fréquence d'échantillonnage

Les deux premiers paramètres sont utilisés pour la commande des blocs, à savoir quand on veut les mettre actifs ou inactifs et avec quelle valeur de départ. L'utilisation des variables de type *double* peut se faire seulement lors de simulation à l'intérieur de Simulink seulement.

À l'inverse des blocs Simulink de base, les blocs des librairies Xilinx discrétisent automatiquement les variables arrivant dans leurs ports d'entrées. Dans le modèle Simulink, des bloqueurs d'ordre 0 avaient dû être incorporés dans le modèle afin d'assurer la discrétisation des variables. Ainsi, la fréquence d'échantillonnage devient un paramètre global. Tous les processus dépendront de cette fréquence, et il nous sera impossible de faire un échantillonnage asynchrone. Aussi, puisque tout dépend de ce paramètre, nos variables paramétrables tels le temps d'intégration des boucles, la commande pour la fréquence centrale des NCO et les filtres numériques sont tous en fonction de cette variable.

4.2 Analyse de l'effet de quantification entre les modèles (Simulink et quantifié)

Le passage en temps réel implique de nombreux changements. Un des changements les plus importants est la numérisation des signaux utilisés. Puisque les signaux sont analogiques et que les signaux à traiter seront numérisés par les convertisseurs numériques analogiques, notre modèle doit être capable de traiter des signaux numériques. Ainsi, dans notre modèle, les données traitées doivent être définies en binaire. Lors de la simulation en Simulink, les variables étaient de type *double*, c'est-à-dire des nombres réels. Cependant, dû au nombre limité de bits pour une variable, une erreur d'approximation intervient dans la quantification de notre signal. La **Figure 44** nous montre un exemple de quantification et les paramètres s'y rattachant.

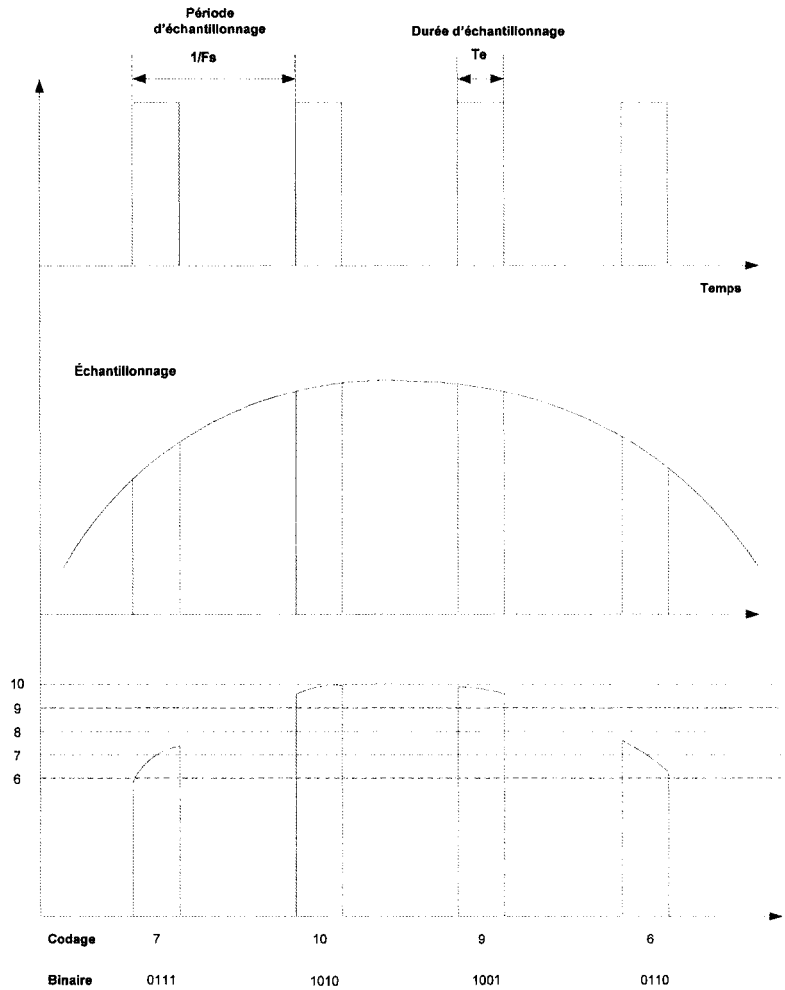

Figure 44 Exemple de quantification d'un signal sinusoïdal

Le fait de quantifier notre signal introduit une erreur d'approximation, appelé erreur de quantification. Lorsqu'on quantifie un signal, on doit prendre un certain nombre d'échantillons, et si la durée de l'échantillonnage (T_e sur la **Figure 44**) n'est pas assez courte, il se peut que le signal change de valeur fréquemment à l'intérieur de cet intervalle. Ce temps de quantification est défini dans les blocs de conversion des variables (*Gateway In* de la libraire de Xilinx) et il est important de choisir ce temps le plus court possible. La période qui correspond au temps le plus court de notre modèle est la période d'échantillonnage, soit l'inverse de la fréquence F_s. De cette façon, nous

arrivons à avoir seulement une approximation du signal. Cependant, il est possible de diminuer cette erreur en introduisant un nombre plus élevé de bits. Le tableau suivant nous montre un exemple de quantification et l'erreur engendrée en fonction du nombre de bits de quantification.

Tableau VII

Exemple d'erreur de quantification

Nombre de bits de quantification	**Valeur à convertir**	**Valeur en binaire**	**Valeur réelle de la quantification**	**Erreur de quantification**
3 bits	1.65	1.11	1.75	0.10
4 bits	1.65	1.101	1.62	0.03

Cette forme d'erreur de quantification survient sur les nombres dont la partie fractionnaire est différente de 0. En analysant les différentes valeurs possibles de nos signaux à l'intérieur de notre récepteur numérique, seule la donnée du satellite GPS a une composante fractionnaire nulle. Les sinusoïdes, les réponses des discriminateurs et les sorties des filtres auront une composante fractionnaire non nulle, impliquant obligatoirement une erreur de quantification.

Pour déterminer le nombre minimum de bits nécessaire, nous devons regarder les limites de nos variables utilisées dans les calculs du récepteur numérique.

Tableau VIII

Valeurs limites des signaux de commandes

Signal	Limite inférieure	Limite supérieure
Récupération de la porteuse	-1	1
Récupération du code	-1	1
Intégrateur	-1	1
Discriminateur PLL (arctan)	$-\pi/2$	$\pi/2$
Discriminateur DLL (AMR)	-1	1

Les valeurs limites des discriminateurs sont démontrées sur les **Figure 33** et **Figure 40**, selon les discriminateurs. Malgré leurs différences d'algorithmes, les valeurs limites des discriminateurs sont calculables. À la sortie des filtres, les valeurs limites dépendent directement de la valeur de la bande de bruit équivalente choisie, tel que montré au **Tableau V**. Cependant, à l'intérieur des processus, certaines valeurs peuvent dépasser ces limites, mais comme elles sont des variables locales, il sera toujours possible de minimiser l'impact.

Le nombre de bits est aussi important dans un autre aspect de notre récepteur numérique, la précision. Le dernier bit de quantification constitue le plus petit pas de discrétisation possible pour notre modèle, déterminant la résolution de notre système. Comme nous avons des nombres fractionnaires, notre résolution est de 2^{-m} où m est le nombre de bits après la virgule. La détermination de m permettra non seulement de réduire les erreurs

de quantification mais aussi de déterminer la plus petite erreur de phase que le modèle pourra calculer. L'importance de ce paramètre est démontrée dans les **Figure 45** et 47.

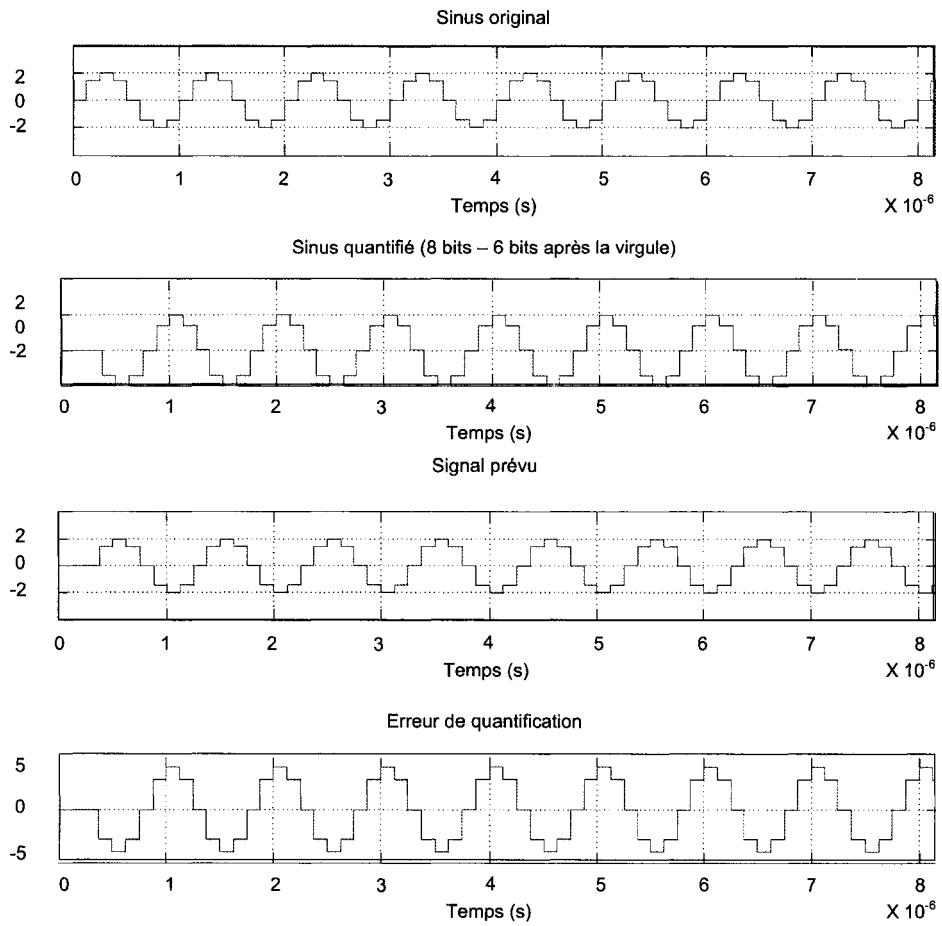

Figure 45 Erreur de quantification avec une quantification de 6 bits

La **Figure 45** permet de montrer l'erreur de quantification lorsque le nombre de bit après la virgule est trop grand par rapport au nombre de bits total, soit 8 bits. Comme nous n'avons que 2 bits avant la virgule dont un est le bit de signe, la précision de notre système est de 0,015625. Cependant, la valeur maximale que nous pouvons atteindre est 1,984375, et non 2. Cette petite erreur, jumelée avec le déphasage engendrée par le bit de signe, fait en sorte que nous avons une erreur de quantification qui s'approche de 5

par moment. Une telle erreur est trop importante pour notre récepteur numérique. Pour diminuer l'erreur de quantification, le nombre de bits après la virgule a été réduit en gardant le même nombre de bits total. Les résultats sont présentés à la **Figure 46**.

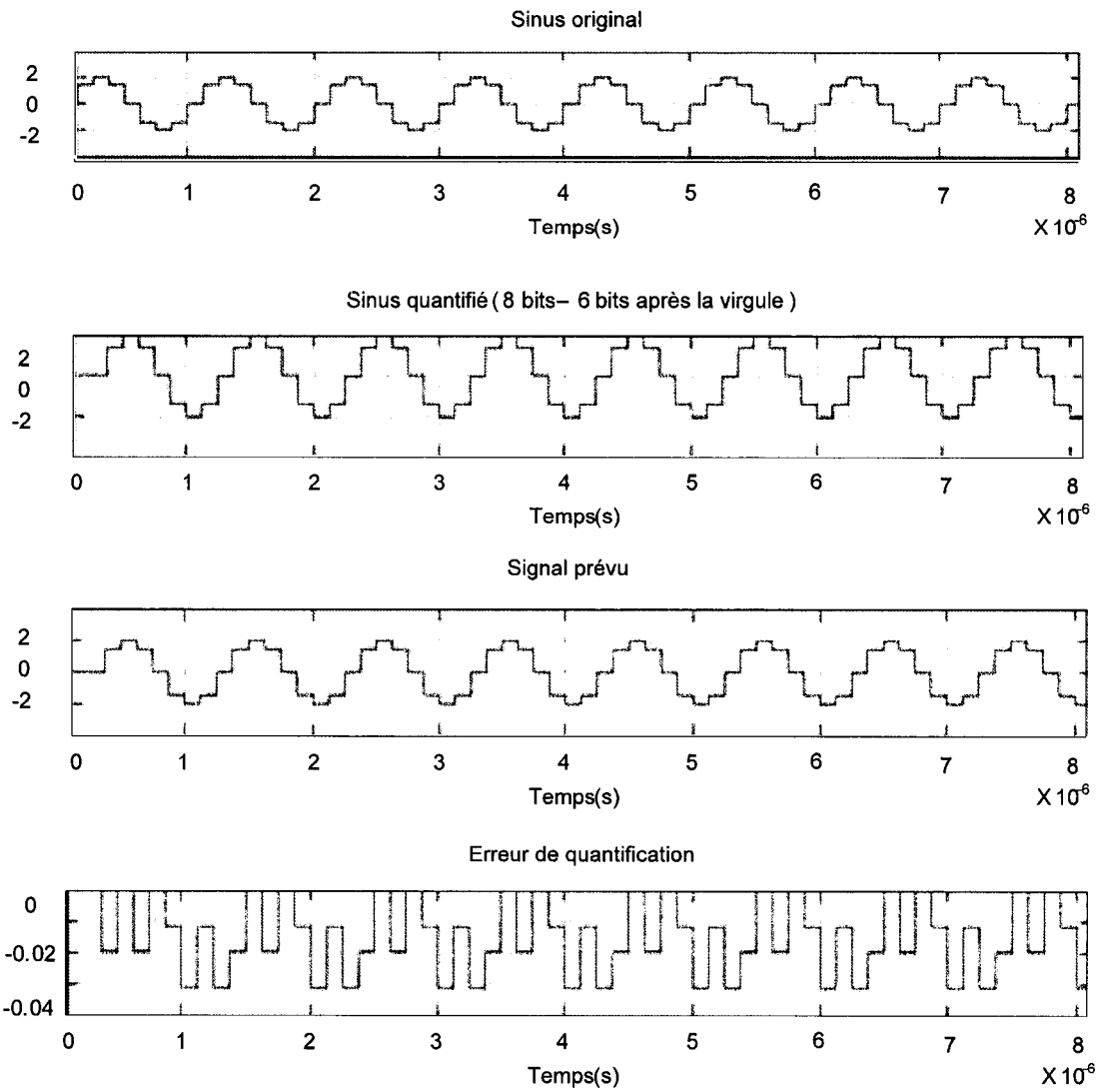

Figure 46 Exemple de quantification avec 4 bits fractionnaires

Avec seulement 4 bits de précision, la résolution est de 0,0625. Par contre, l'erreur de quantification a diminué à 0.03, (au maximum). Une amélioration grandement importante pour la précision de notre système. Si l'on se réfère au **Tableau VIII**, la

valeur maximale est de $\pi/2$, ce qui demande un minimum de 3 bits pour définir la partie entière de notre signal.

Au niveau de la résolution, le Tableau IX

Niveau de précision en fonction du nombre de bits montre le niveau de précision selon le nombre de bits de la partie fractionnaire.

Tableau IX

Niveau de précision en fonction du nombre de bits

Nombre de bits	**Précision minimale**
1	0,5
2	0,25
3	0,125
4	0,0625
5	0,03125
6	0,015625

De par la conception du NCO et des tables de correspondance de génération des sinusoïdes, le nombre total de bits requis est de 9. En étant déjà assuré d'avoir au moins 3 bits pour la partie entière de nos variables, nous pouvons aller jusqu'à 6 bits de précision au niveau de la partie fractionnaire. Cependant, en analysant le **Tableau IX**, le gain de précision après le quatrième bit n'est pas significatif. Une précision de plus du centième n'est pas nécessaire, par contre, celle du dixième est essentielle.

Ces conditions d'utilisations (nombre de bits et précision) de notre récepteur indiquent que notre récepteur doit avoir, au minimum 3 bits pour la partie réelle et 4 pour la partie fractionnaire. Ainsi, pour optimiser l'utilisation des ressources à notre disposition dans le FPGA, le choix de 8 bits (dont quatre pour la partie fractionnaire) s'impose de lui-même.

4.3 Modification du modèle Simulink pour le passage en temps réel

Au CHAPITRE 3, nous avons démontré et expliqué le fonctionnement du simulateur d'une chaîne de communication GPS. Les différentes caractéristiques du modèle doivent maintenant être adaptées afin de le modifier pour effectuer le passage en temps réel. Si l'on reprend notre cheminement montré à la **Figure 13**, le chapitre antérieur établit la base de notre graphique. En comparant le tout à une figure géométrique simple, nous avons établi les fondations de la pyramide. Maintenant, il faut modifier notre modèle Simulink afin de pouvoir générer les bons codes. Les outils utilisés ont été décrits dans le chapitre 2 du présent ouvrage, mais seront expliqués dans le détail à la section 4.4 .

En tenant compte des différentes caractéristiques du modèle ainsi que du matériel utilisé, la première étape est de séparer notre modèle en divers processus et d'analyser ces processus afin de déterminer par quel matériel ils seront exécutés. Il est à noter, qu'à partir de cette section, la partie représentant le canal n'est pas étudié. Le projet a pour but de faire un récepteur numérique, et, dans cette situation, la modélisation du canal n'est pas nécessaire. Le canal simulé génère des interférences et nous permet de tester la robustesse de notre récepteur. Cette partie pourra être faite à l'intérieur de l'ordinateur, ce qui nous permettra de réserver le maximum de ressources disponibles pour nos processus.

Pour revenir à notre modèle, nous pouvons le séparer de la façon qui suit :

Tableau X

Processus à l'intérieur du récepteur numérique

Partie du modèle	Processus
Source	Générateur de fréquence
	Générateur de code
	Générateur de donnée
	Modulation en fréquence
Récepteur	Boucle PLL
	Boucle DLL
	NCO
	Filtre
	Filtre passe-bas/Intégrateur
	Discriminateur PLL
	Discriminateur DLL

Au niveau de la source, on remarque, outre la multiplication en fréquence, que nous n'avons que des oscillateurs. Une caractéristique de ces oscillateurs est qu'ils ne sont pas tous indépendants. Premièrement, la fréquence des données est de 50 Hz, qui est la fréquence de base. Les données sont ensuite modulées par le code qui est à 1,023 MHz. Le message codé est enfin modulé par la porteuse de 1575,42 MHz. Tous ces signaux (données, code et porteuse) sont synchronisés, ce qui nous permet de limiter au minimum les oscillateurs. En utilisant un générateur de fréquence de 1575,42 MHz, nous pouvons, à l'aide d'un diviseur de fréquence, générer la fréquence du code. Comme ils sont synchronisés, la cadence du code est un dénominateur de la fréquence du code. Ainsi, le diviseur de fréquence aura comme valeur 1575,42/1,023 donc 1540. Pour notre modèle, nous n'utiliserons pas la fréquence réelle de la porteuse, tel qu'expliqué à la section 3.5, mais une valeur intermédiaire qui respecte la synchronisation des signaux.

Cette particularité du signal GPS nous permet donc de générer un seul oscillateur, ce qui, nous le verrons plus tard, diminue la quantité de ressources utilisées dans notre matériel, laissant ainsi plus de ressources disponibles au traitement du signal.

Au niveau du récepteur, contrairement à la source, les processus ont des caractéristiques relativement différentes.

Tableau XI

Caractéristiques des processus du récepteur

Processus	**Caractéristiques**
Boucle PLL	Sensible au bruit
Boucle DLL	Résistante au bruit
NCO	Grande résolution, oscillateur local
Filtre	Calcul mathématique, fréquence faible
Filtre passe-bas/Intégrateur	Calcul mathématique, temps d'intégration court
Discriminateur PLL	Algorithme mathématique complexe
Discriminateur DLL	Algorithme mathématique complexe

Le **Tableau XI** nous résume les caractéristiques des processus du récepteur GPS. La grande diversité des processus fait en sorte que nous ne pouvons, comme dans le cas de la source, réutiliser un processus. Pour ajouter à cette difficulté, à cause des caractéristiques différentes, on ne peut même pas utiliser le même matériel pour tout notre récepteur, ce qui augmente la complexité du travail. Si l'on se réfère au CHAPITRE 2, les caractéristiques des DSP et FPGA nous mettent sur la piste pour séparer notre modèle. Ainsi, les processus qui demandent un calcul mathématique complexe seront modifiés pour être traités dans le DSP. Dans le même ordre d'idée, les

algorithmes demandant une fréquence élevée de traitement seront transformés pour le FPGA. L'optimisation des ressources étant un élément important, nous allons devoir surveiller l'implémentation afin de ne pas surcharger un des processeurs.

Cette séparation de notre modèle implique aussi un changement et une difficulté accrue au niveau des communications. L'échange de données devra être rapide afin d'arriver à récupérer le signal GPS. Cet élément est important d'autant plus qu'il s'agit d'interaction à l'intérieur même d'une boucle. Le bus de données choisi devra fonctionner à une vitesse élevée. Comme le récepteur se compose de deux boucles, le temps alloué à l'échange de données augmentera le temps de récupération de la porteuse ou du code, diminuant l'efficacité de nos boucles.

Tableau XII

Répartition des processus du récepteur numérique

Processeur	**Processus**
FPGA	Boucle PLL
	Boucle DLL
	NCO
	Filtre passe-bas/Intégrateur
DSP	Filtre
	Discriminateur PLL
	Discriminateur DLL

Tel qu'indiqué au **Tableau XII**, le FPGA prendra en charge les processus dit rapides, tandis que le DSP prendra en charge les processus dit complexes. Les filtres sont laissés au DSP en raison du minimum d'interaction entre les différents processeurs. En raison

de sa fréquence d'horloge rapide, le FPGA est en mesure d'effectuer des opérations non complexes beaucoup plus rapidement que le DSP. Cet avantage est capital pour les récupérations de porteuse et de code. Par exemple, le désétalement spectral doit être fait rapidement afin que les échantillons de données soient envoyés au filtre passe-bas, remplacé ici par un intégrateur.

Les processus du modèle étant repartis à travers les processeurs cibles (DSP ou FPGA), nous pouvons poursuivre à travers notre implémentation pas-à-pas. Après la simulation de notre modèle, il faut le modifier avec les librairies correspondantes incluses dans les logiciels de Xilinx et de Sundance. Le développement des librairies de transition à l'intérieur de Matlab/Simulink par ces entreprises facilite le passage en temps réel car elle respecte les caractéristiques des processeurs.

Pour le DSP, Sundance en collaboration avec Texas Instruments, a développé une implémentation contenant une librairie (SMT6050) et un logiciel (Code Composer Studio). Code Composer studio permet non seulement de transformer un fichier en code C++ mais aussi de se connecter et de configurer le processeur. De plus, Mathworks offre un compilateur C++ avec Matlab directement. Ces deux aspects combinés ensemble font que les changements pour les processus allant dans le DSP sont minimes. La **Figure 47** nous montre, comme exemple, la source de notre modèle.

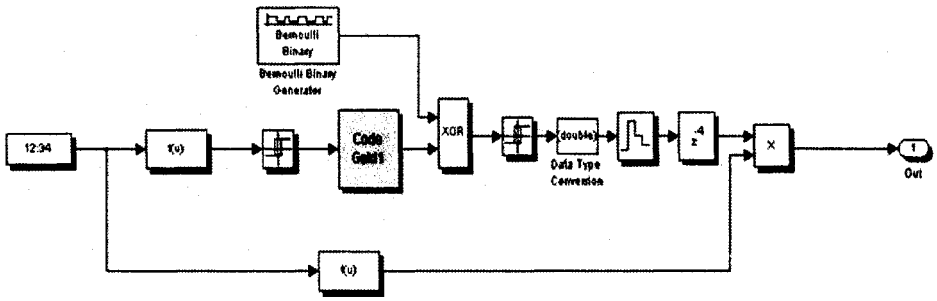

Figure 47 Source GPS en Simulink

On remarque certaines modifications par rapport à la source présentée au chapitre 3. Tout d'abord, il n'y a qu'un seul oscillateur pour générer les fréquences de la porteuse et du code de Gold. À l'aide d'une horloge simple et d'un algorithme mathématique, la génération des fréquences se fait de façon plus simple et utilise moins de ressources. Les expressions mathématiques utilisées pour générer les fréquences sont :

1. Pour la porteuse :

$$\frac{\sqrt{2}}{2} * sin(2 * Fs * u(1)) \qquad \textbf{(4.1)}$$

où fs est la fréquence de la porteuse, et u(1) la valeur de l'horloge d'entrée.

2. Pour le Code de gold :

$$\frac{\sqrt{2}}{2} * sin(2\pi * Fc * u(1)) \qquad \textbf{(4.2)}$$

où fc est la fréquence du générateur de code de Gold, et u(1) la valeur de l'horloge d'entrée.

Comparativement au système de base, qui comprenait au moins 4 oscillateurs différents afin de simuler des perturbations, ce système permet une utilisation maximale des ressources avec des algorithmes mathématiques simples. Ainsi pour simuler l'effet Doppler, nous n'avons qu'à rajouter une composante de phase dans la génération des signaux. Nous pouvons simuler ainsi deux perturbations : perturbation sur la fréquence et perturbation sur la phase. En se basant sur l'expression générale d'une onde radio, nous avons :

$$\cos(\omega * t + \theta) \tag{4.3}$$

où θ est la phase du signal et ω, la fréquence. Ainsi, l'expression mathématique de la porteuse devient :

$$\frac{\sqrt{2}}{2} * sin((2\pi * Fs + \omega_p) * u(1) + \theta_p) \tag{4.4}$$

où ω_p est la déviation en fréquence de la porteuse et θ_p est la déviation en phase de la porteuse. Comme la fréquence F_s est fixe, les paramètres variables à définir sont les déviations du signal. Donc, au lieu de générer 3 fréquences, ce qui occupe beaucoup de place dans le processeur, nous n'avons qu'à passer les valeurs des perturbations.

À travers les libraires de Sundance, la SMT6050 est celle qui s'assure du lien entre Code Composer Studio, le DSP ainsi que le modèle Simulink. En effet, cette librairie est divisée en quelques parties, dont communication, mathématique, etc... Comme Matlab possède un convertisseur en C++ d'un modèle Simulink, les blocs internes Simulink n'ont pas besoin d'être modifiés ou changés. Nous pouvons générer notre code à partir du schéma du modèle lui-même.

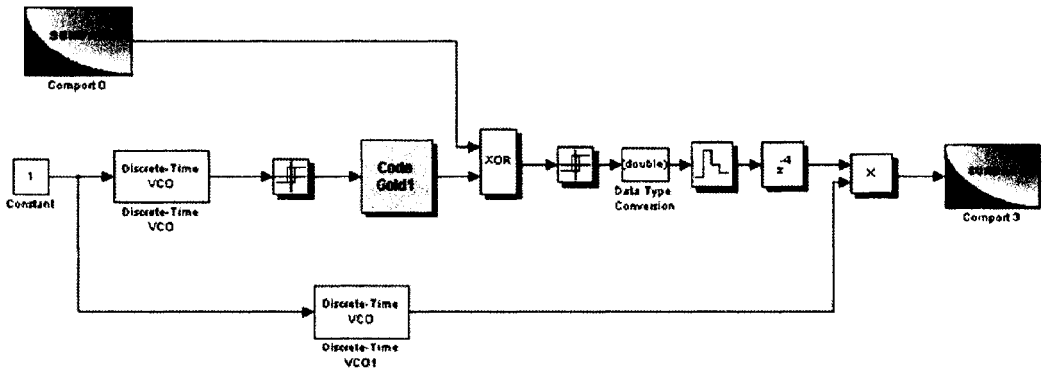

Figure 48 Source GPS modifié pour le DSP

La **Figure 48** nous montre les changements apportés à la source du signal GPS. Puisque nous simulions une perturbation, nous avons décidé de laisser deux(2) oscillateurs (VCO) locaux pour générer les fréquences de Gold et de la porteuse. À la différence du modèle de base, les données de navigation, simulées normalement par un générateur de Bernoulli, sont un paramètre que nous simulerons à l'intérieur de l'ordinateur. Pour ce faire, nous devons indiquer au processeur TMS320C6416 qu'il doit vérifier ses ports de communication afin d'y recevoir une donnée. De plus, avec les libraires Sundance, nous avons le choix du port sur lequel la donnée sera envoyée. Il en est de même pour le signal de la source GPS. Le bloc *Comport* nous permet d'effectuer ses tâches en précisant la fréquence d'échantillonnage que nous voulons. Dans notre modèle, nous avons soumis l'entrée sur le port 0 du processeur et la sortie sur le port 3. Les résultats de cette implémentation seront discutés à la section 5.2.

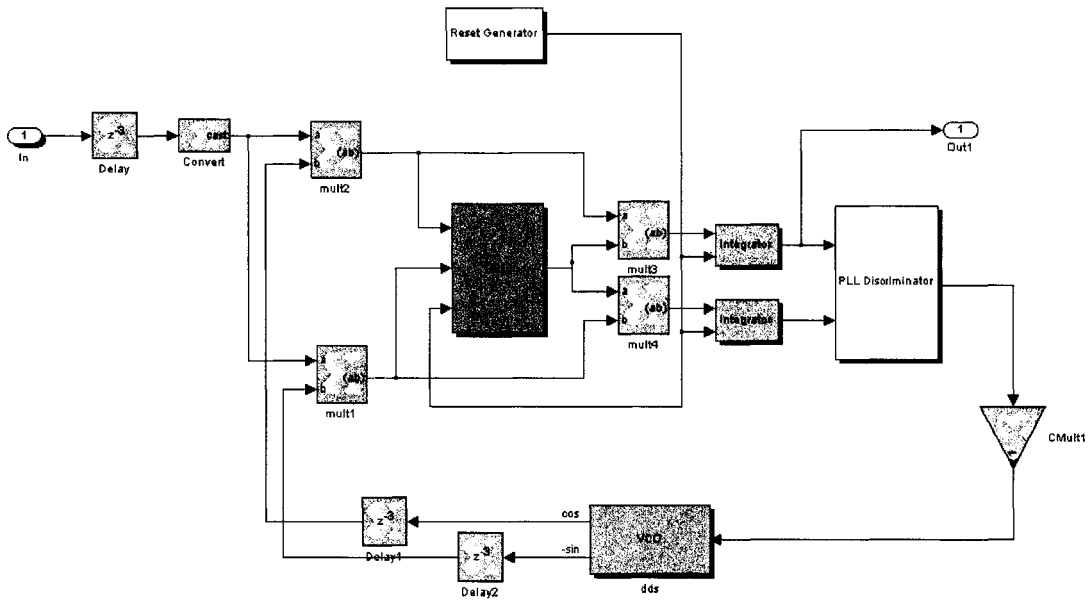

Figure 49 Schéma du récepteur préliminaire pour l'implémentation dans le FPGA

Au niveau du récepteur, comme il est séparé en plusieurs processus, nous avons plusieurs changements à faire. Si l'on se réfère au **Tableau X**, les processus les plus simples à modifier seront les multiplicateurs en bande de base. La multiplication de deux signaux se fait relativement bien en binaire. La **Figure 50** nous montre le désétalement spectral de notre signal en Simulink. Il s'agit tout simplement d'une multiplication entre le code de Gold et le signal démodulé avant de transmettre la voie I et la voie Q dans le filtre passe-bas.

Cette partie de notre récepteur se trouvera dans le FPGA tel que montré au **Tableau XII**. Dans la libraire que XIlinx ont conçu pour les FPGA, il existe plusieurs blocs pouvant faire le désétalement spectral. Par contre, comme nous sommes soucieux de l'optimisation des ressources, le choix d'un multiplicateur simple avec un délai de 2 échantillons reste un choix logique. Ainsi, la démodulation sous Simulink, montré à la

Figure 50 devient le schéma de la **Figure 51** une fois modifié pour le temps réel. L'ajout d'un bloc *Gateway In* et de deux blocs *Gateway Out* est nécessaire pour la numérisation du signal. Ces deux types de blocs sont importants pour la réalisation du modèle en temps réel puisqu'il détermine le code binaire dont nous voulons paramétrer le modèle. Ce sujet, ainsi que ses effets, sera abordé plus loin dans le présent chapitre.

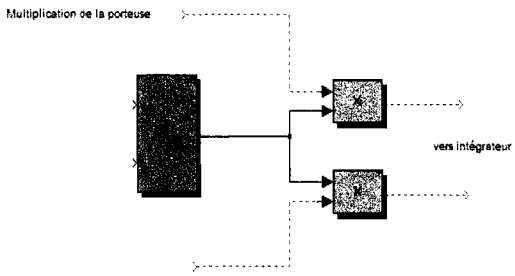

Figure 50 Désétalement spectral du signal en Simulink

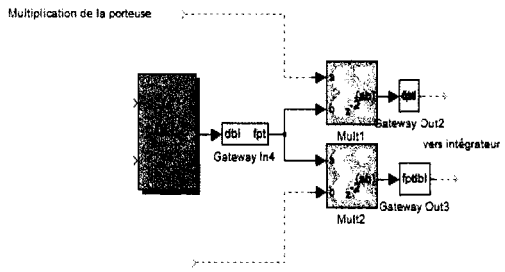

Figure 51 Désétalement spectral modifié pour le temps réel

Ce premier changement nous montre déjà un des effets de la modification pour le passage en temps réel sur notre modèle, la quantité de bloc nécessaire. Si l'on compare les deux figures (**Figure 50** et **Figure 51**), on remarque que pour une opération avec deux blocs Simulink, nous devons mettre cinq blocs des librairies de Xilinx pour le FPGA. Cette particularité nous demande de simplifier au maximum notre système afin d'utiliser le moins de bloc possible, donc le moins de ressources matérielles. Il est bien important de rappeler que nous travaillons avec un nombre limité d'outils, que ce soit de portes logiques, de tables de correspondance, etc. Aussi, un autre aspect démontré par ce changement est le délai qui sera incorporé à cause du bloc multiplicateur. Lorsque nous avons un signal GPS à 1575,42 MHz et deux boucles de Costas pour récupérer la porteuse et le code, l'ajout d'un délai peut causer une désynchronisation importante, ce qui pourrait entraîner un temps plus long dans la récupération des donnés, puisque la boucle mettra plus de temps à s'arrimer sur le signal GPS. Un bloc de délai devra donc

être ajouté au signal pour que la porteuse et le code soient récupérés plus rapidement. Ce bloc devra donc tenir compte du nombre de délais ajoutés lors de la modification de notre système. Considérons la génération de la réplique de la porteuse, il est important qu'elle soit en phase, sur la voie I, par rapport au signal GPS. Or, en incorporant des délais, la réplique sera automatiquement retardée, c'est-à-dire déphasée. Le système prendra plus de temps avant de récupérer les données GPS, diminuant ainsi les performances de notre modèle. Cependant, comme la réplique est un signal sinusoïdal, nous pouvons récupérer notre signal à chaque période, ce qui fait que pour une valeur de délai x, nous avons la possibilité de récupérer notre signal en phase, ainsi que pour nx, où n est un entier. La détermination de x sera expliquée plus loin dans le texte.

Ce petit changement, sur le désétalement spectral, sert à illustrer l'implication du nombre de modifications à apporter afin de bien arrimer le passage entre le temps réel et la simulation. De plus, l'importance de bien optimiser notre modèle est facilement visible par le dernier exemple, puisque nous avons plus que doubler les ressources nécessaires lors de la modification.

En poursuivant la modification de notre récepteur, un cas particulier, le NCO, nous force à effectuer des changements importants dans notre modèle. Le NCO se situe dans les deux boucles, de phase et de code, de notre récepteur et génère la réplique que l'on module avec notre signal pour récupérer, soit la porteuse, soit le code. La théorie, tel que montré dans Kaplan**[1]**, nous annonçait cette modification qui a trait à la sensibilité du NCO. Le générateur de fréquence doit se baser sur une fréquence d'horloge afin de créer le signal qui servira au recouvrement. Mais ce signal doit être numérisé, ce qui en fait, se traduit par la relation suivante :

$$F_{out} = \frac{F_{clk}}{2^L} * S \qquad \textbf{(4.5)}$$

où F_{out} est la fréquence de sortie du NCO, F_{clk} la fréquence de l'horloge, L , le nombre de bits sur la phase et S, valeur de l'ajustement de la phase. On remarque que le nombre de bits a une influence importante sur la fréquence de sortie. Ce nombre détermine en fait le pas avec lequel les changements peuvent être effectués. Par exemple, avec une valeur L de 2, le pas sera du quart de la fréquence d'horloge. Par contre, pour une valeur de L plus grande, soit 8, le pas seront de 1/256ième de la fréquence d'horloge, ce qui rend le NCO beaucoup plus précis. Mais l'ajout du nombre de bits augmente l'imprécision au niveau de la valeur de S, nous devons donc faire un compromis entre ces deux sources d'erreur. Plusieurs possibilités peuvent être envisagées pour améliorer la sensibilité du NCO, mais deux approches retiennent notre attention : soit l'ajustement fin de la phase et le processeur d'arrondi**[30]**.

L'ajustement fin de la phase se veut d'utiliser le nombre maximum de bits sur S. Cette valeur contient l'erreur sur la phase de la boucle. Ainsi, en augmentant la précision de S, sans changer L, nous diminuons l'erreur dû à L sur la valeur d'erreur. Pour contourner cette problématique, nous pouvons séparer l'équation 4.5 de cette façon :

$$F_{out} = \frac{F_{clk}}{2^L} * \left(S' + \frac{B}{A} \right) \tag{4.6}$$

Les valeurs de A et de B doivent être l'erreur en arrondi. Comme nous voulons une fréquence de sortie, en simulation, pour la porteuse de 2,046 MHz, en se basant sur cette relation, nous devons avoir un nombre de bit égal à 11. Cependant, notre signal S a seulement 8 bits. Nous devons donc le séparer afin de diminuer l'erreur.

Une première hypothèse posée dans notre modèle est que nous avons 8 bits, soit 4 bits après la virgule. Pour les nombres binaires, la valeur des chiffres après la virgule $\frac{1}{2^n}$ où n est la position du chiffre. Donc, avec 4 bits comme partie fractionnaire, nous avons

une précision de $\frac{1}{16}$ sur la valeur de *S.*, impliquant un pas minimal de 0,5435 MHz. Le dénominateur, la valeur de A, sera toujours de 16. Le bloc *slice* nous permet d'isoler les bits d'un nombre binaire. Ainsi nous pouvons donc raffiner la valeur de *S* en sachant que le dénominateur aura une valeur constante à 4 bits. Le schéma de la **Figure 63** nous montre l'arrangement du modèle avec les blocs *slice* et des opérateurs logiques utilisés.

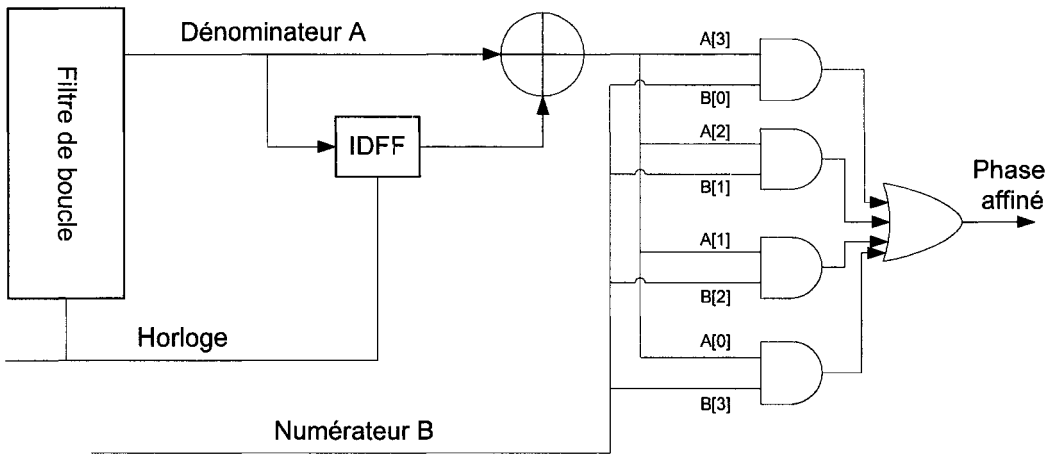

Figure 52 Principe de l'ajustement fin de la phase

Le schéma de la **Figure 53** nous montre le processus qui nous permettra d'arrondir correctement la valeur de *S.* Ce principe permet de limiter la perte des bits moins significatifs sur la précision du NCO. Le principe d'arrondissement utilisé dans notre système est assez simple. La première étape est de séparer le nombre en 2, soit la partie entière et la partie décimale. La partie décimale est alors envoyée dans un registre. La prochaine valeur décimale est additionnée à la précédente. Cette valeur est donc ajoutée à la partie entière. L'équation suivante nous démontre ce calcul :

$$y_M(n) = \left[x_L(n) + \sum_{k=0}^{n} x_{L-M}(k) \right] \qquad (4.7)$$

Cette formule générale peut être adapté à notre système avec les termes suivant, basé sur les équations 4.5 et 4.6.

$$y_{sortie}(n) = \left[S(n) + \sum_{k=0}^{n} LSB_{S(n)}(k) \right] \tag{4.8}$$

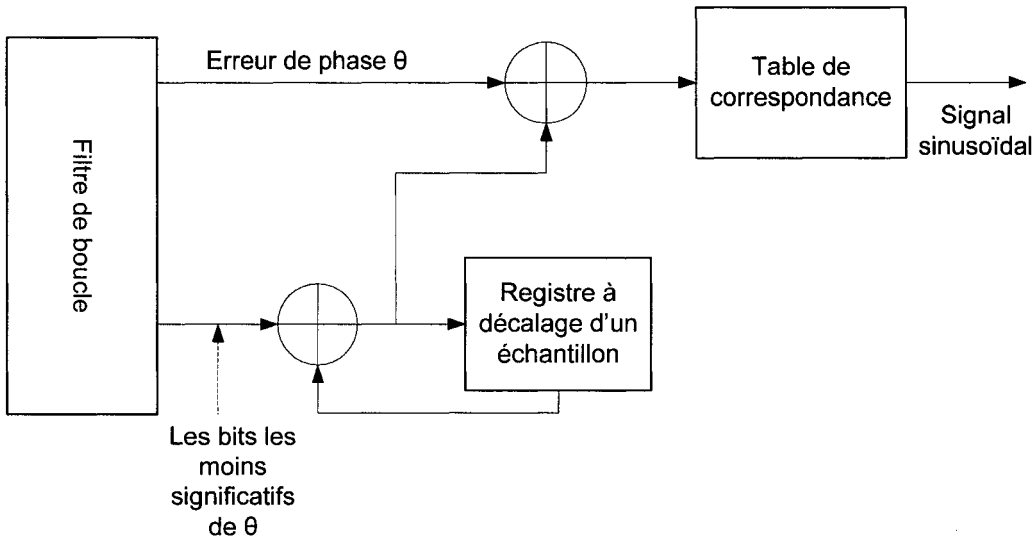

Figure 53 Principe d'arrondissement

Le résultat est donc une valeur beaucoup plus près de la réalité, donc une réponse plus rapide de nos boucles de phases et de code. À l'intérieur du FPGA, le signal sinusoïdal est calculé à partir de LUT (*look-up tables*) ou tables de correspondance, ce qui signifie qu'il y a un pas de discrétisation à respecter. Ce facteur a une influence considérable sur le temps de réaction de notre boucle. À la sortie des discriminateurs, le calcul indique l'erreur de la fréquence de recouvrement, et non la valeur de la fréquence, le NCO doit donc compenser cette erreur avec une augmentation ou diminution de la fréquence de sortie. Donc, par rapport aux tables de correspondance, l'erreur doit être additionnée à la valeur centrale de notre NCO. Le pas de discrétisation intervient ici dans notre modèle. Plus il est petit, plus le raffinement de la fréquence de sortie sera ajusté. Cependant, le nombre limité de ressource fait en sorte qu'un compromis devra être fait entre la

précision du NCO et le temps de réponse de nos boucles. Les tables de correspondance prennent un espace fixe à l'intérieur du FPGA et le travail doit être fait à l'intérieur de ces paramètres.

4.4 Méthodologie de génération des codes sources temps réel

Les modifications et l'analyse de la quantification permettent de prédire la réponse de notre récepteur numérique à l'intérieur des parties matérielles. Avant l'implémentation de notre récepteur numérique dans le DSP et FPGA, l'application doit être encodé pour les exécuter dans les processeurs.

4.4.1 Génération du code C++ pour le DSP

Les différentes ententes entre les compagnies font en sorte que la génération d'un code pour l'implémentation à l'intérieur d'un processeur soit fait de façon le plus conviviale possible. Le DSP choisi, dont les caractéristiques sont à l'ANNEXE 2, a été conçu par Texas Instrument et doit être programmé en assembleur. Cependant, avant de l'implémenter en assembleur, nous devons générer le modèle en C++ et Texas Instruments a développé le logiciel Code Composer Studio qui permet de programmer, à partir d'un code C++, le DSP.

Tel qu'expliqué à la section 4.3, les modifications pour le DSP sont minimes. Cependant, il reste des paramètres à imposer qui auront une influence sur le code. Matlab et Microsoft ont travaillé ensemble afin d'inclure un simulateur à l'intérieur de Matlab, ce qui permet de générer un code C++ directement à l'intérieur de Matlab. Le DSP doit avoir une fréquence d'opération plus élevée que notre fréquence d'échantillonnage. Le fonctionnement du DSP, à l'inverse du FPGA, ne dépend pas d'une horloge interne, mais plutôt de la longueur des instructions. Le langage assembleur, qui sera le langage avec lequel sera programmé le DSP, est construit

d'instruction simple et à chaque pas d'opération, une instruction est faite. Ainsi, c'est le nombre d'instruction pour traiter une donnée qui détermine le temps de traitement de cette donnée, représenté par la relation suivante :

$$t_{traitement} = nb_{opération} * F_{opération} \tag{4.9}$$

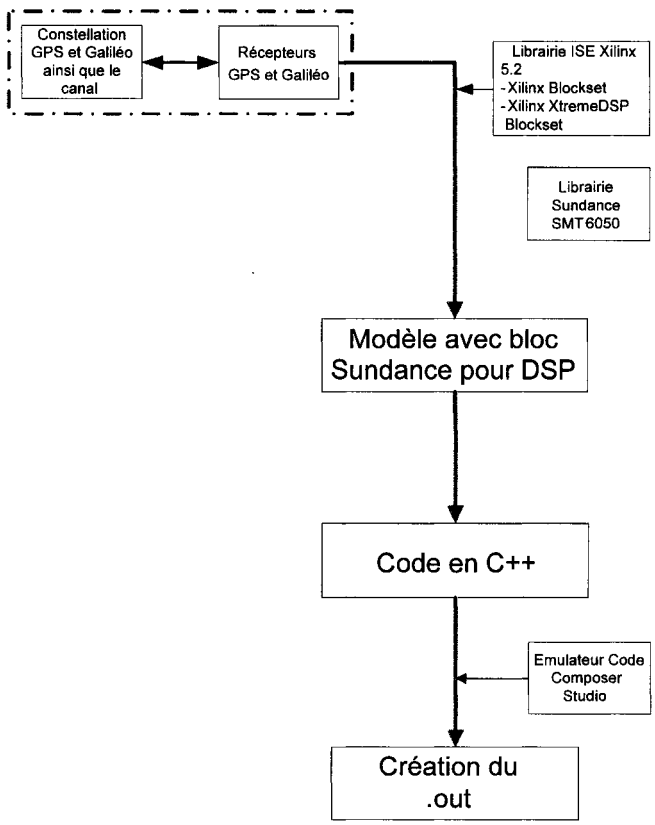

Figure 54 Méthodologie de génération du code pour le DSP

La **Figure 54** montre les étapes pour arriver à générer le code C++ pour le logiciel Code Composer Studio. Tout d'abord, tel que montré précédemment (**Figure 48** et **Figure 49**), des blocs de communication doivent être introduits pour recevoir ou envoyer les données. Ensuite les générateurs de code C compatible avec Mathworks génèrent le code

C++ et automatiquement le logiciel Code Composer Studio nous affiche le code généré. Il est alors possible d'analyser le code afin de l'optimiser.

4.4.2 Génération du code VHDL pour le FPGA

Le VHDL a été conçu pour programmer les différents processeurs utilisés en industrie. Il est la relève du langage assembleur. L'avantage d'utiliser le langage VHDL est qu'il facilite l'incorporation des vecteurs dans les algorithmes. Selon la déclaration des variables faites, le code peut inclure des calculs sur les nombres binaires. En fait, les nombres binaires sont déclarés comme des vecteurs et le langage VHDL permet d'effectué des opérations binaires (tel le déplacement d'un bits vers la gauche ou la multiplication en complément à 2). En considérant que nous utilisons des variables binaires dans notre modèle, le langage VHDL est tout désigné pour l'implémentation dans le FPGA.

Le passage entre le modèle Simulink de notre récepteur GPS numérique et le code VHDL est représenté à la **Figure 55**, qui est un agrandissement d'une partie de la **Figure 13**.

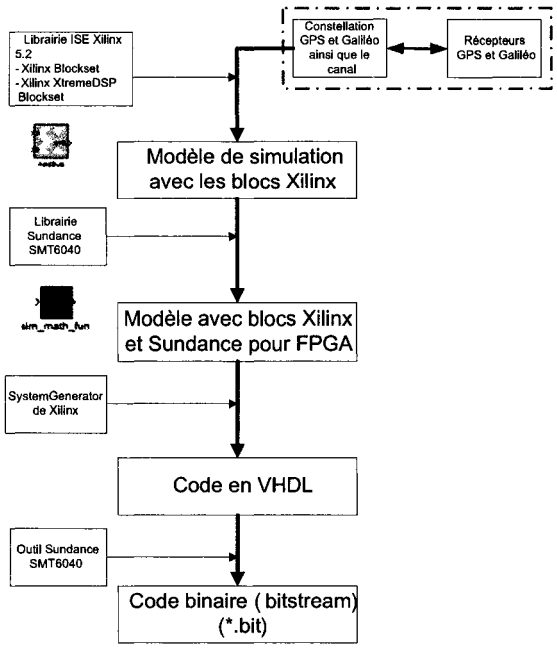

Figure 55 Méthodologie de génération du code binaire

Les changements au récepteur sont nettement plus nombreux que dans le cas du DSP. En effet, Mathworks a développé sont logiciel avec le langage Visual Basic, langage qui s'apparente plus au C++ qu'au VHDL, ce qui explique le nombre plus important de modifications apportées au modèle Simulink.

Tout d'abord, le changement de type de variable entraîne plusieurs modifications. Le fait de travailler en binaire, au lieu des variables de type *double* ou *real*, limite les opérations que nous pouvons effectuer sur les variables, tel qu'expliqué aux sections 4.3 et 4.2. Une fois les modifications de variables et d'algorithmes effectués, les blocs de communications sont ajoutés au modèle pour assurer l'échange des données entre les processeurs. Ces blocs sont inclus dans les librairies des processeurs cibles.

Afin de générer le code en VHDL, le groupe de Xilinx a développé un ajout à sa série ISE, le programme System Generator. Ce logiciel prend les codes des blocs Xilinx qui

compose le modèle à l'intérieur de Simulink, déjà en VHDL, et les intègre à un code global contenant les variables globales et les horloges. La génération de ce code se fait par des processus automatisés dans System Generator, et n'est pas optimal. Le logiciel doit tenir compte de plusieurs paramètres, dont la fréquence d'échantillonnage, le processeur ciblé et l'outil de synthétisation. La **Figure 56** montre l'interface usager du logiciel System Generator. Les paramètres indiqués dans la figure sont les plus importants pour notre modèle. Le choix du processeur permet de déterminer le nombre de portes logiques nécessaires pour la réalisation du modèle. Si le nombre de portes logiques est trop petit, notre modèle ne pourra être implémenté dans le processeur cible. Comme expliqué tout au long de ce chapitre, l'horloge du FPGA est une valeur importante puisqu'elle servira de signal de commande à nos processus. Il est important que la fréquence d'échantillonnage de Simulink soit plus faible que l'horloge du FPGA, afin que le FPGA puisse générer la fréquence d'opération requis dans notre modèle. Le logiciel System Generator fera le rapport entre ces deux horloges et, à l'aide d'un fichier binaire inclus dans le code général, générera la fréquence d'opération du récepteur numérique.

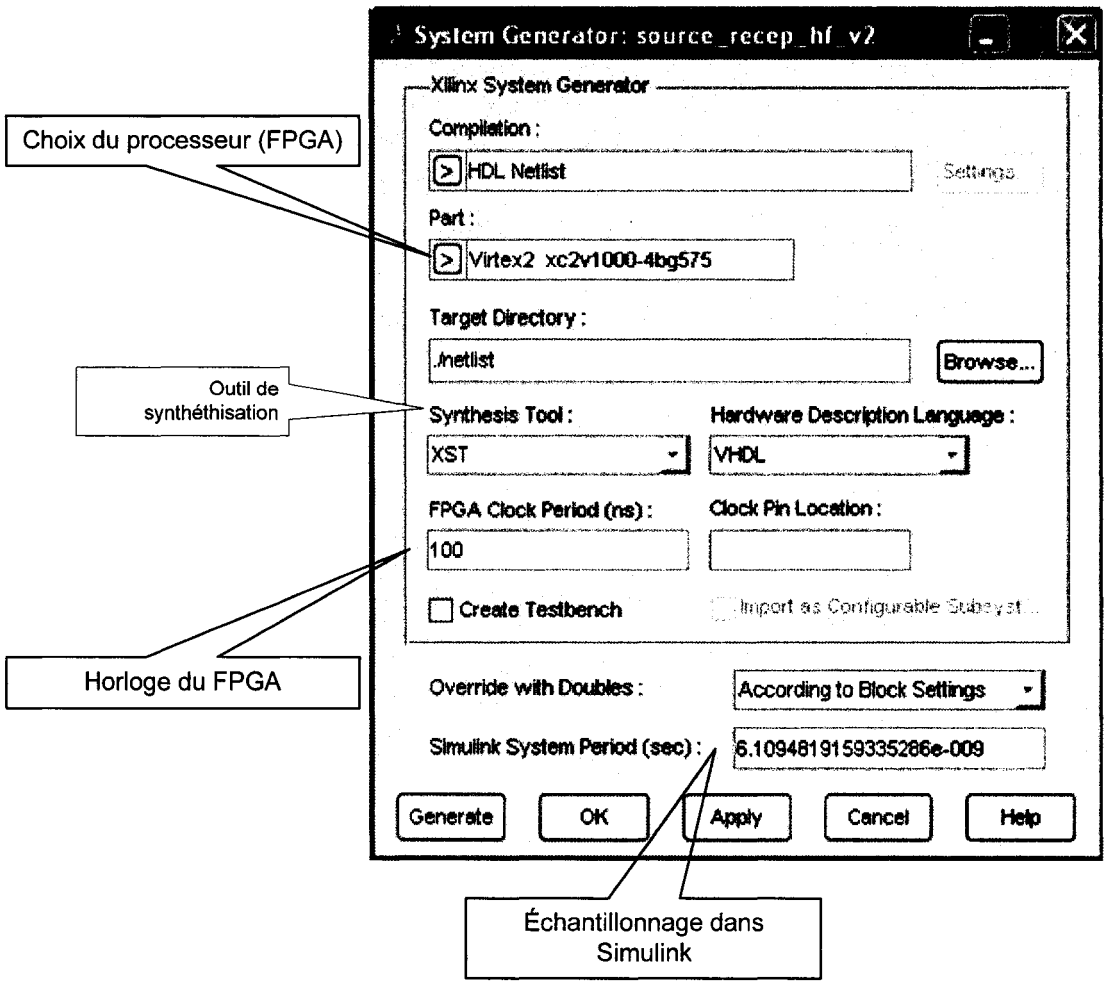

Figure 56 Paramètres de génération du code binaire

La génération automatisée n'est pas optimale mais elle permet de voir si des erreurs se sont glissées à l'intérieur de notre modèle. Le logiciel Project Navigator, un produit de Xilinx, sert à analyser le code VHDL généré. La quantité de ressources utilisées (porte logique, table de correspondance, entrée-sortie,...) est calculée par ce logiciel et une carte de l'emplacement des processus à l'intérieur du FPGA est même fournie par Project Navigator. Cette carte permet, de façon visuelle, d'analyser la répartition des ressources à notre disposition.

Tableau XIII

Utilisations des ressources à l'intérieur du FPGA

Ressources	**Nombre utilisé**	**Nombre total**	**Pourcentage d'utilisation**
Slices	937	5120	18 %
Bascule	817	10240	8 %
Tables de correspondance	2132	10240	20 %
Entrées – Sorties reliées	50	432	12 %
Horloge globale	1	16	6 %

Enfin, la génération du code binaire se fait à l'aide de Project Navigator. Il est aussi possible de générer directement le code binaire à l'aide de System Generator. Cette option cependant empêche de voir la quantité de ressources utilisées, ce qui permet de valider l'implémentation de notre récepteur. Le **Tableau XIII** représente l'espace utilisé par le récepteur numérique à l'intérieur du FPGA. Il est normal que les tables de correspondance représentent un pourcentage élevé puisque les quatre NCO génèrent les sinusoïdes à partir de valeurs pré-calculés à l'intérieur de tables de correspondance. Cette analyse a été faite pour un récepteur avec un seul canal. Il sera donc possible d'ajouter d'autres canaux, puisque nous n'utilisons, au maximum, qu'un cinquième des ressources nécessaires.

4.5 Conclusion

La quantification a plusieurs effets sur le modèle du récepteur. La préparation pour l'implémentation en temps réel nécessite une étude approfondie du changement de variable. Nous avons donc pu établir, par l'étude de ses effets, le nombre exact de bits

dont nous avons besoin, soit un minimum de 8. Le processus inclus à l'intérieur de notre modèle a été modifié pour traiter les nouveaux types de variables. Suite à ces modifications, la table est mise pour générer les différents codes requis pour implémenter les processeurs cibles. Dans le prochain chapitre, nous aborderons les fonctionnalités et verrons les effets aux niveaux des algorithmes des modifications effectuées. Le passage pour le temps réel est un processus qui nécessite plusieurs étapes, pouvant introduire plusieurs erreurs. Cependant, avec l'analyse du présent chapitre, ces erreurs possibles devraient être minimisées.