REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ECOLE NATIONALE POLYTECHNIQUE

DEPARTEMENT: GÉNIE ELECTRIQUE

Mémoire de Projet de Fin d'Etudes en vue de l'Obtention du Diplôme d'Ingénieur d'Etat en Electronique

Implémentation d'un MODULATEUR OFDM sur un circuit FPGA

Proposé et dérigé par :
Mr Zidane TERRA

Etudié par :
§ MERROUCHE Walid
§ MOSSI IDRISSA Moctar

Soutenu le: 26 juin 2007

Devant le jury composé de :
Président : Mr M. HADDADI
Examinateur : Mr M.Taghi
Promoteur: Mr Z. TERRA

Promotion : juin 2007

Remerciement

Nous tenons tout d'abord à remercier notre promoteur pour l'aide qu'il nous a apporté au cours de ce travail et pour ce beau monde de la conception sur FPGA dans le domaine de la communication qu'il nous a fait découvrir.

Nous remercions toute la « famille » de la cité universitaire Bouraoui pour nous avoir supporté, aidé et soutenue pendant tout le temps que nous avons passé ensemble.

MOSSI IDRISSA Moctar

Mes remerciements vont également au peuple algérien pour m'avoir accueilli et donné la chance de faire un grand pas en avant, tout particulièrement à la famille Alma de Azazga, à la famille Mettouchi de Tigizirt pour m'avoir accueilli. ET je tiens à remercier exceptionnellement la famille Belaïdi pour m'avoir accepté au sein de leur famille.

Et enfin je remercie toute ma famille.

Merrouche WALID

Je remercie toute ma petite famille de mes parents jusqu'au mon petit frère de m'avoir soutenu à tout moment. Et un grand remerciement à ma grande famille et surtout mes oncles à Alger et tous les amis de l'ENP et de notre résidence.

الملخّص:

هذا العمل يقدّم مفهوم تقنية النظام المتعدد الحوامل المتعامدة (OFDM) ، مبرمج على مستوى الدارة المبرمجة FPGA. في البداية قمنا بتعريف النظام OFDM، ثم عرجنا بعد ذلك إلى تفصيل الأدوات و البرامج اللازمة لتحقيق هذه البرمجة و هي: ISE وXILINX. في القسم FPGA عرضنا بعض التطبيقات التي تناولت النظام OFDM و برمجته على مستوى الدارة FPGA .أمّا الباب الأخير من هذا البحث فقد تناول عملنا في برمجة النظام OFDM و مراحله التفصيلية.

الكلمات المفاتيح: النظام المتعدد الحوامل المتعامدة (OFDM)، الدارة المبرمجة FPGA، البرنامج ISE و VHDL.

RÉSUMÉ :

Ce travail présente l'implémentation d'un modulateur OFDM implémenté sur FPGA. Nous avons tout d'abord abordé la théorie sur l'OFDM, après nous avons présenté les outils de conception à savoir l'ISE et Modelsim. Et enfin nous avons implémenté notre modulateur OFDM. Nous avons effectué une mise au point de l'architecture du modulateur et les étapes nécessaires à sa validation, telles que la synthèse, la simulation, le Routage et le placement.

Mots clés : OFDM, FPGA, ISE, VHDL, Placement, Routage, Synthèse.

ABSTRACT :

In this work we present a design of an OFDM modulator implemented on FPGA. We first present the background around OFDM, then a presentation on the design on FPGA, so we talk about ISE, and MoedelSim. At the end we present an architecture that we try to implement. In this way we present the architecture validation by synthesis, simulation, placement and routage.

Keys words: OFDM, FPGA, ISE, VHDL, Placement, Routage and Synthesis.

Table de matière

Abréviation

ASIC: Application Specific Integrated Circuit.
DAB: Digital Audio Broadcast
TFD: Transformée de Fourier Discrète.
DSP : Digital Signal processor or Digital Signal Processing.
DVB-T : Digital Video Broadcast Terrestrial
FDM : Frequency Division Multiplexing
FFT: Fast Fourier Transform
FIFO: First In First Out
FPGA: Field Programmable Gate Array
HDL: Hardware Description Language
CI: circuit Intégré
ICI Inter- Carrier Interference
TFDI: Transformée de Fourier Discrète Inverse.
IEEE: Institute of Electric and Electronic Engineers
IF: Intermediate Frequency.
FI: Fréquence Intermédiaire
IFFT: Inverse Fast Fourier Transform
I/O: Input /Output
IQ: In phase and Quadrature
ISI: Inter Symbol Interference
LUT: Look Up Table
MHz: Mega Hertz
OFDM: Orthogonal Frequency Division Multiplexing
PSK: Phase Shift Keying
QAM: Quadrature Amplitude Modulation
RAM: Random Access Memory
ROM: Read Only Memory
UCF: User Constraints File
VHDL: VHSIC Hardware Definition Language
XST: Xilinx Synthesis Tool
WLAN: Wireless LAN (Local Area Network)

Table de figures

Introduction générale

Les communications numériques sont en train d'envahir la quasi-totalité des domaines d'activité et la demande pour les systèmes de transmission assurant des très hauts débits avec une qualité de services importante ne cesse de croître; Ceci a motivé la recherche de nombreux schémas de transmission capable de supporter des transmissions à large bande.

Ces dernières années, les groupes de recherche ont découvert l'intérêt considérable des transmissions multiporteuses orthogonales OFDM (Orthogonal Frequency Division Multiplexing) pour les transmissions à large bande. La motivation principale de cet intérêt est la grande efficacité spectrale de l'OFDM, comparée à celles des schémas traditionnels de transmission à large bande, qui permet de réduire le spectre exploité et donc le coût des communications. En plus, les développements récents de la technologie d'intégration VLSI (Very-Large-Scale-Integration) ont également motivé l'utilisation des systèmes OFDM en rendant leur implémentation commercialement viable **[1]**.

D'autre part, les circuits FPGA (Field Programmable Gate Array), qui sont des circuits programmables standards, et qui peuvent être adaptés à des besoins divers, deviennent incontournables dans les applications nécessitant un temps de développement rapide et une modularité garantie. Ils sont surtout utilisés dans les systèmes embarqués (avionique, automobile, espace, …) et tendent à se généraliser dans le domaine des applications on chip.

Ce projet énumère les diverses approches pour implémenter et réaliser un système OFDM en émission sur circuit FPGA.

Le présent PFE est organisé comme suit: le premier chapitre comporte les principes de base et la constitution d'un modulateur OFDM, le deuxième chapitre est consacré au développement d'un projet sur circuit FPGA et le troisième chapitre est l'implémentation du modulateur OFDM. On terminera par une conclusion générale.

1 CHAPITRE I : Emetteur OFDM

1.1 INTRODUCTION

OFDM (Orthogonal Frequency Division Multiplexing) est un système de transmissions multi porteuses, dont lequel un bloc d'information est transmis sur un nombre de *sous porteuses* synchronisées en temps et en fréquence entre elles. Cette *synchronisation* permet de *conserver* la nature *orthogonale* de l'OFDM **[2].** L'avantage principal d'utilisation de l'OFDM est la réduction de l'interférence dans une bande étroite, ce qui donne une certaine robustesse au signal.

L'idée de l'OFDM est la transmission par multiplexage fréquentiel orthogonal; cette idée a pu être dépistée dans les années 50 **[3]**. Dans le multiplexage fréquentiel classique (FDM), la bande totale est divisé en N sous canaux qui ne se chevauchent pas, alors que sur OFDM la bande est divisée en un certain nombre de sous canaux se superposant mais avec des fréquences orthogonales, l'objectif est un plus grand nombre de sous porteuses sans interférences ; la figure I.1 montre la différence entre l'OFDM et le multiplexage fréquentiel classique "FDM".

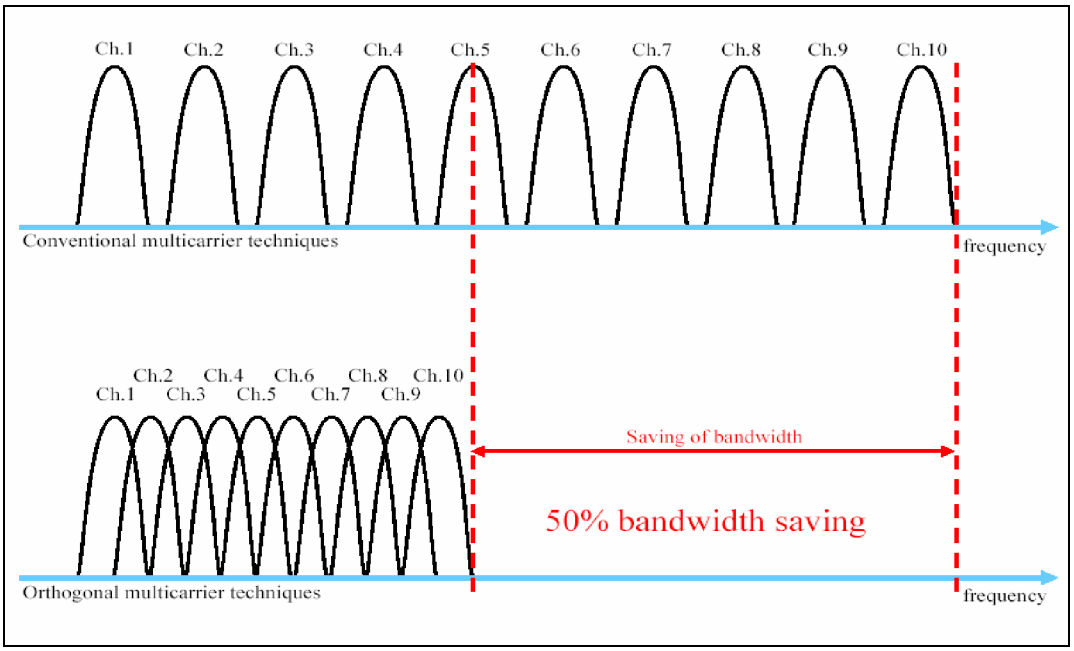


Figure 1.1 Multiplexage de fréquense a) classic FDM b) Orthogonale OFDM

L'ensemble des sous porteuses forme un symbole OFDM. Grâce à l'orthogonalité de l'OFDM, les différentes sous porteuses se chevauchent dans le domaine fréquentiel mais sans causer d'interférence entre sous porteuses ICI (Intercarrier Interference). Cette propriété rend ce système robuste contre le problème des **multi-trajets.** Ceci rend la technique particulièrement intéressante pour la transmission des données à haut débit et offre la possibilité d'utilisation d'un réseau à une seule fréquence SFN

(Single Frequency Network) en diffusion de données.

L'OFDM a été utilisé depuis les années 60 pour les systèmes militaires, pendant les années 80 a été présenté sur les modems à grande vitesse ; dans les années 90, elle a été exploité pour des communications de données à large bande, des lignes numériques élevées d'abonné de débit binaire (HDSL), des lignes numériques d'abonné de très haut vitesse (VDSL), la radiodiffusion numérique d'acoustique (TAPE) et la télévision a haute définition (HDTV) **[3]**. Actuellement, l'OFDM est la technique de multiplexage employée par la norme d'IEEE 802.11a pour l'Ethernet sans fil, comme pour l'IEEE 802.16 ; OFDM est également employé à HIPERLAN/2 **[4].**

1.2 Système OFDM:

1.2.1 Système multi-porteuses

L'OFDM est un système multi-porteuses **[5]** (**figure** 1.2). A la différence des systèmes de porteuses simples, toutes les fréquences sont envoyées simultanément dans le temps (en parallèle).

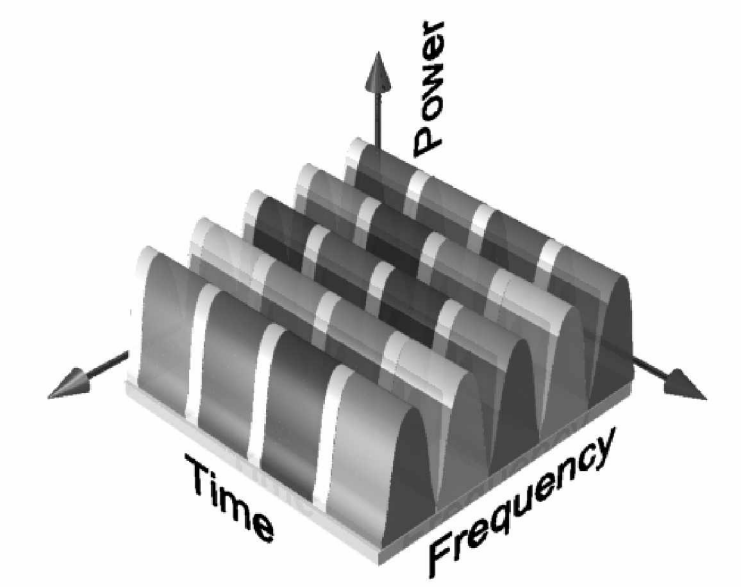


Figure 1.2 La Technique Multi-porteuses (OFDM)

L'OFDM offre plusieurs avantages par rapport au système de porteuses simple, une meilleure immunité pour l'effet de trajets-multiples et une égalisation plus simple de canal et des contraintes de synchronisation.

1.2.2 Orthogonalité des porteuses

Les fréquences utilisées dans le système d'OFDM sont orthogonales. Cette propriété est montrée dans la figure1.3 où *f1*, *f2*, *f3*, *f4* et *f5* sont orthogonales. Ceci a comme conséquence l'utilisation efficace de la largeur de bande. L'OFDM peut donc fournir un débit plus élevé pour une même largeur de bande.

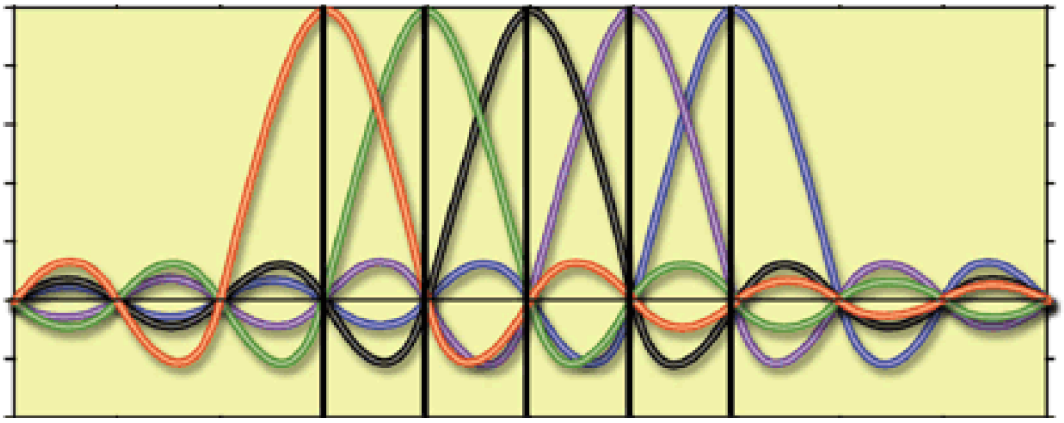


Figure 1.3 Spectre d'un signal OFDM formé de 5 sous porteuses.

En général, les fonctions sont orthogonales entre elles si :

$$\int_0^T s_i(t)s_j(t)dt = \begin{cases} C & i = j \\ 0 & i \neq j \end{cases} \quad \textbf{…….. (1.1)}$$

Où *T* est la durée de symbole.

Dans le cas de l'OFDM, les fonctions orthogonales sinusoïdales représentent les sous porteuses d'un signal OFDM réel (équation 1.2).

$$s_k(t) = \begin{cases} \sin(2\pi k f_0 t) & 0 < t < T_{FFT} \quad k = 1,2,...M \\ 0 & ailleurs \end{cases} \quad \textbf{…… (1.2)}$$

Où f_0 est l'espacement entre les sous porteuses, *M* est le nombre de sous porteuses, T_{FFT} est la durée du symbole OFDM.

Un signal OFDM est réalisé à partir d'une somme de sinusoïdes, chaque sinusoïde corresponde à une sous porteuse. La fréquence, en bande de base, de chaque sous porteuse est un multiple de l'inverse de la durée du symbole OFDM, ce qui implique que chaque sous porteuse a un nombre entier

de période par symbole OFDM. Cette propriété entraîne la vérification de la condition d'orthogonalité (équation 1.1) entre les sous porteuses.

En effet, lorsque le signal OFDM est détecté en utilisant une transformée de Fourier discrète DFT, le spectre vu par le récepteur n'est plus continu, comme indiqué dans la figure1.4 (b), mais échantillonné. Le spectre échantillonné est marqué par des points noirs dans la figure1.4 (a). Les échantillons de la FT correspondent juste aux valeurs maximales des sous porteuses, alors la région de chevauchement fréquentiel n'affect pas le spectre vu par le récepteur, par conséquent elle n'affecte pas la transmission OFDM.

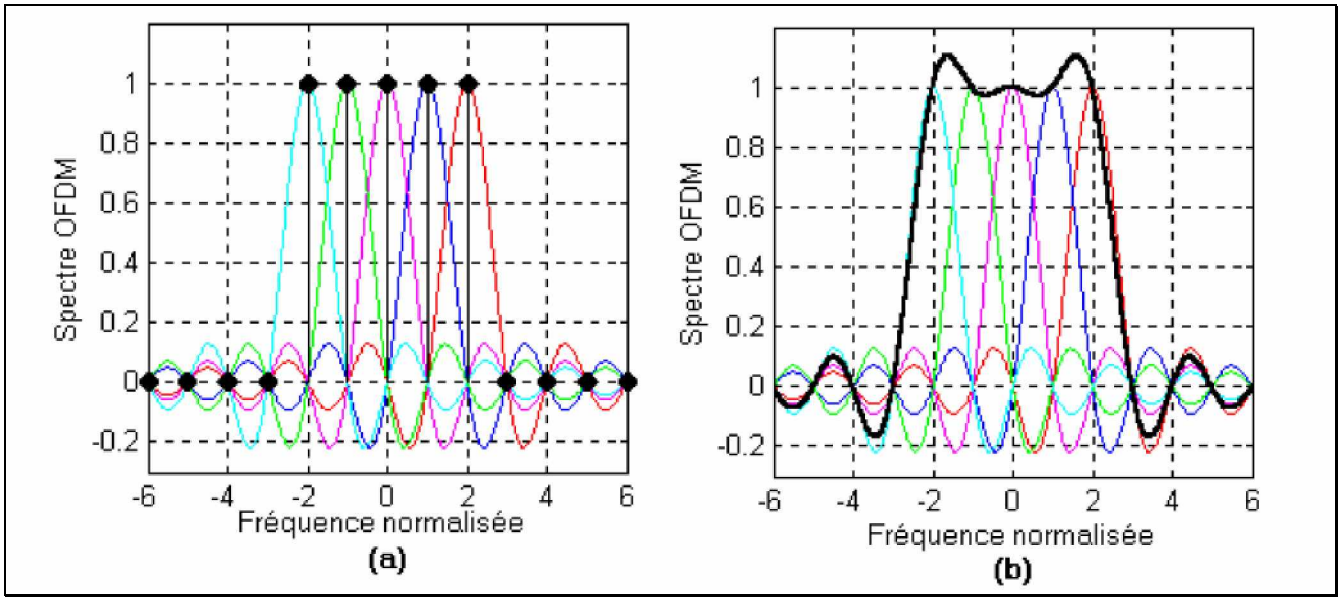


Figure 1.4 Spectre d'un signal OFDM formé par 5 sous porteuses.

1.2.3 Effet ISI et ICI.

Une des raisons pour lesquelles l'OFDM est largement répandu est la manière efficace de son traitement de l'interférence inter-symbole "**ISI**", causée par les trajets multiples. Il réduit la durée du symbole avec un rapport de N_s.

En plus de la protection du signal OFDM contre l'ISI, l'intervalle de garde assure également la protection contre les effets du décalage temporel entre le récepteur et l'émetteur **[2]**, voir figure 1.5.

Toutefois une interférence "ICI" (Intercarrier Interference) peut être provoquée par les défauts de sélectivité des canaux de transmission. Ceci aura l'effet de rendre "moins orthogonales" les porteuses. Pour éliminer cet effet le symbole OFDM est cycliquement prolongé dans l'intervalle de garde, comme il est montré dans la figure1.6 **[6].**

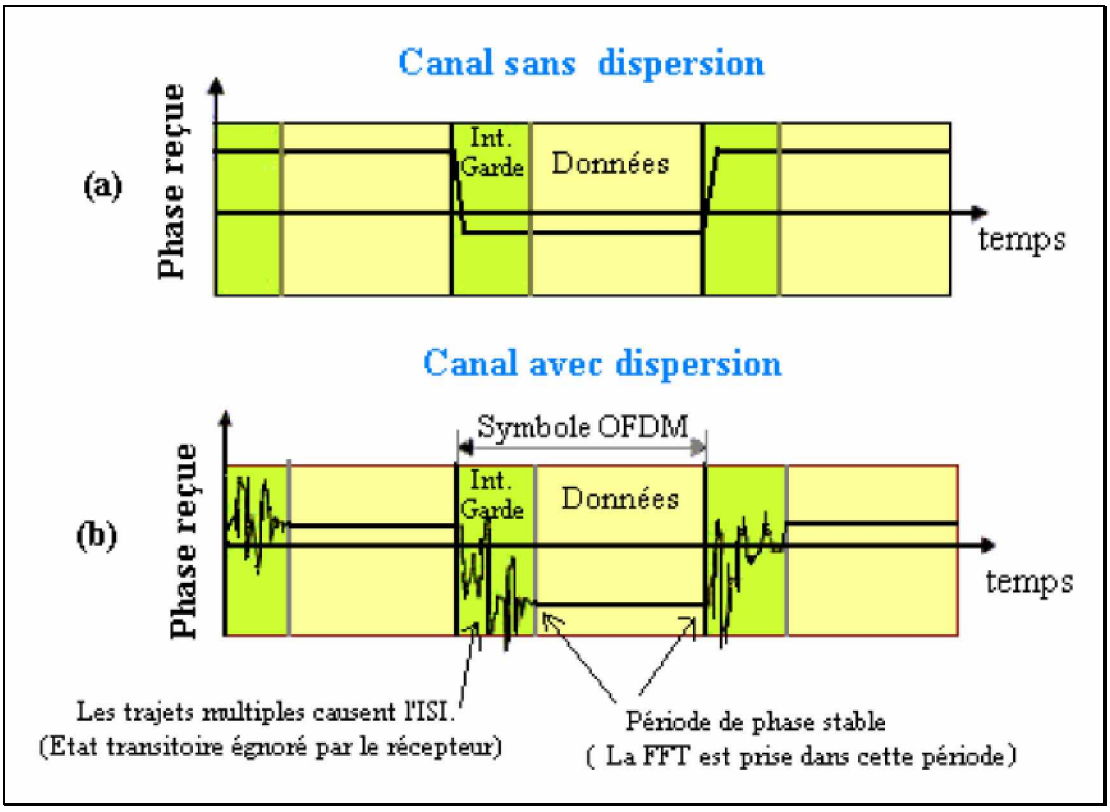


Figure 1.5 La fonction de l'intervalle de garde pour la protection contre ISI

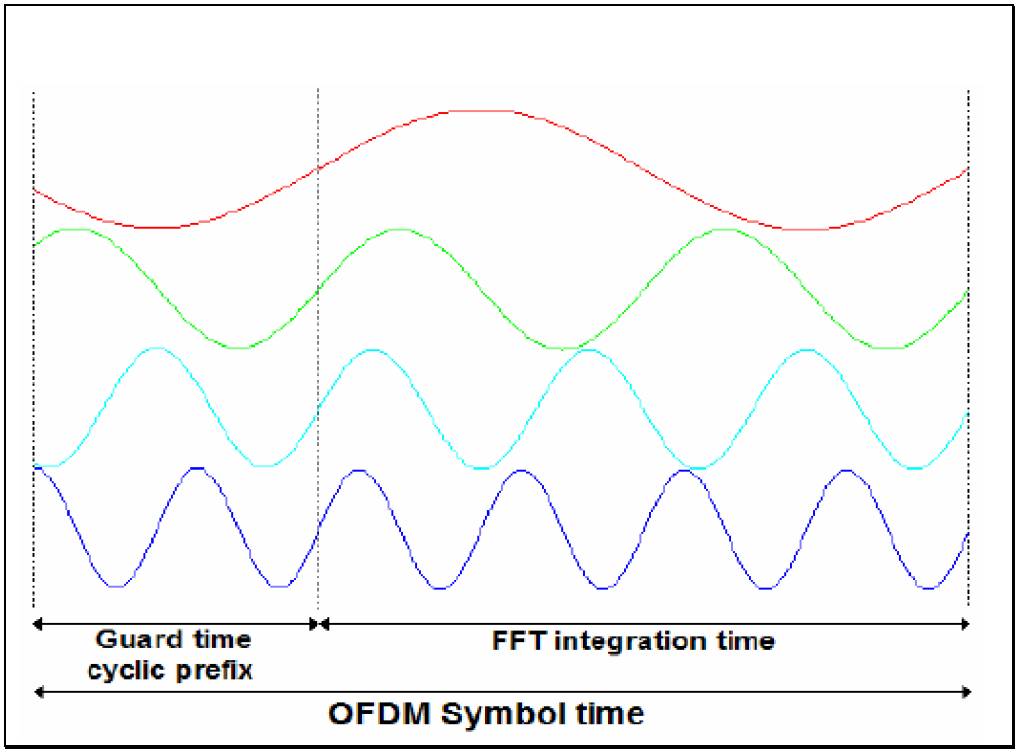


Figure 1.6 Le préfixe cyclique

La figure 1.7 illustre comment un intervalle T_{GI} de garde, est présenté pour prolonger la durée de la période active de symbole, T_{FFT}, pour créer toute la durée de symbole, T.

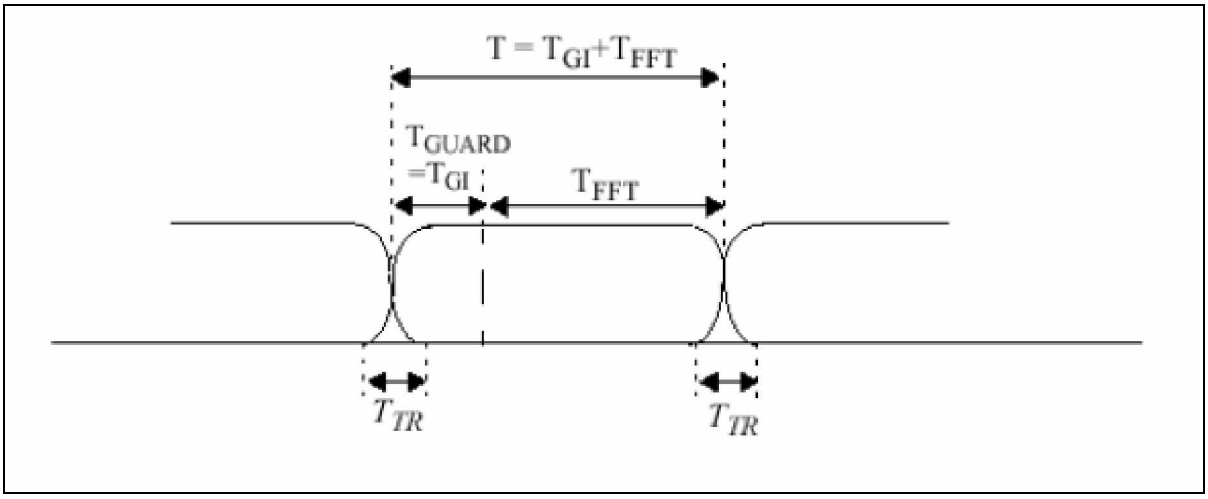


Figure 1.7 l'intervalle T_{GI} de garde

1.3 Etude d'un émetteur OFDM

1.3.1 Schéma bloc OFDM

Chaque sous porteuses dans un système OFDM est modulée en amplitude et en phase par les bits de données. Les techniques de modulation utilisées sont BPSK, QPSK, 16QAM, 64QA…etc. Le processus de combiner différentes sous porteuses pour former un signal dans le domaine temporel est réalisé en utilisant la transformée de Fourier rapide. Différents codes comme le codage par bloc sont employés pour réaliser une meilleure exécution.

- Le schéma fonctionnel d'un émetteur récepteur d'OFDM est représenté par la figure1.8.

À l'émetteur, les données sont codées et intercalées. Au début une conversion série/parallèle des données. Ensuite, une modulation aux M sous porteuses parallèles dans le domaine fréquentiel. Les valeurs complexes résultantes sont converties par une FFT inverse de taille $N \geq M$ du domaine fréquentiel vers le domaine temporel. Le préfixe cyclique (ou l'intervalle de garde) est alors ajouté à chaque symbole avant la conversion numériques-analogique et puis la transmission.

- **Au récepteur**, après conversion analogique-numérique et suppression du préfixe cyclique, une FFT de taille N agit pour traduire le signal reçu à un signal parallèle de $M \leq N$ représentations, de données complexes. Les opérations d'égalisation du canal, le décodage et la détection sont utilisées pour récupérer le flux des données transmises.

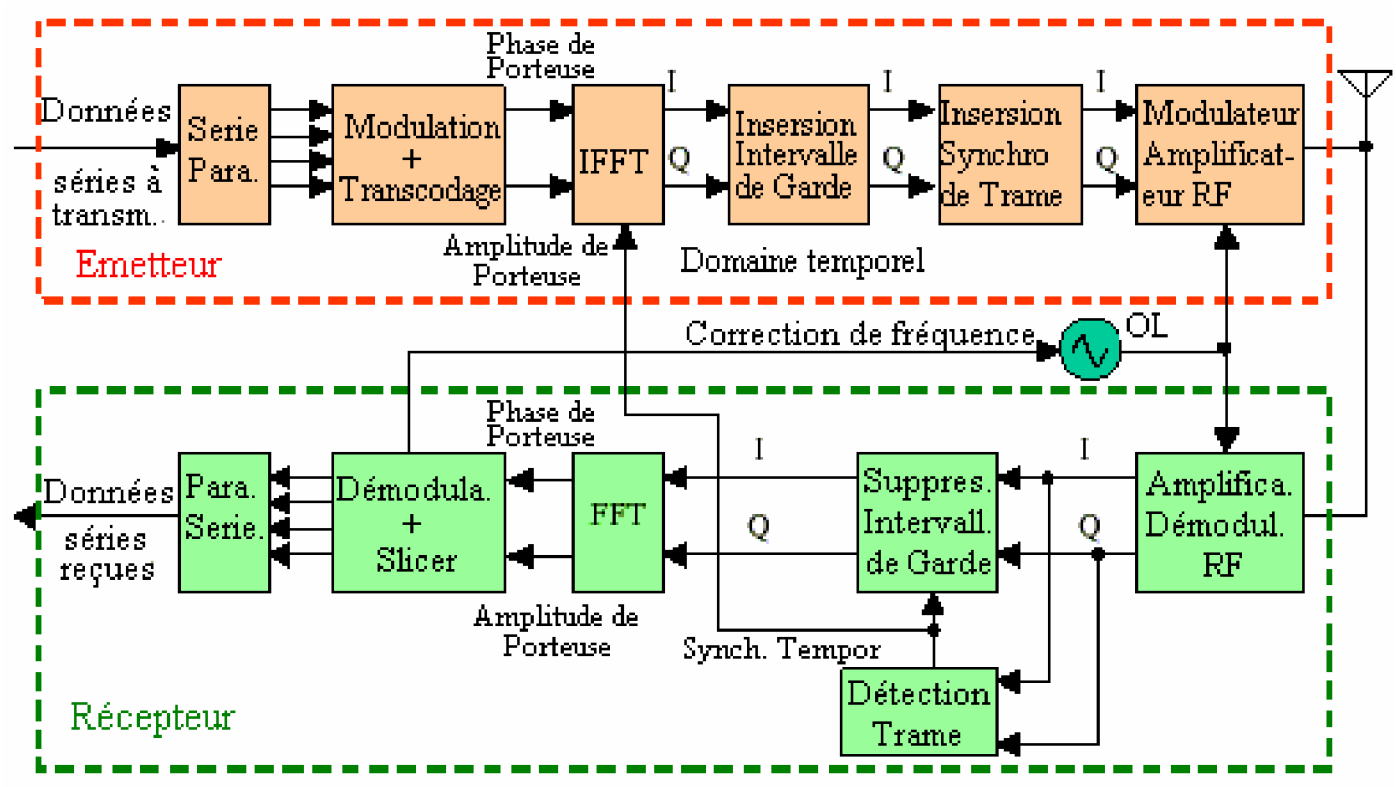


Figure 1.8 Schéma fonctionnel d'un émetteur-récepteur OFDM de base

1.3.2 Conversion série-parallèle

Les données à transmettre sont sous forme d'un flot de données binaire. Chaque symbole OFDM transmit entre 40-4000 bits, alors l'étape de conversion série-parallèle est nécessaire pour transmettre un nombre important de bits par un seul symbole OFDM (voir figure 1.10). Le nombre de bits transmis dans chaque symbole OFDM dépend des schémas de modulation utilisée par les sous porteuses et du nombre de sous porteuses utilisées

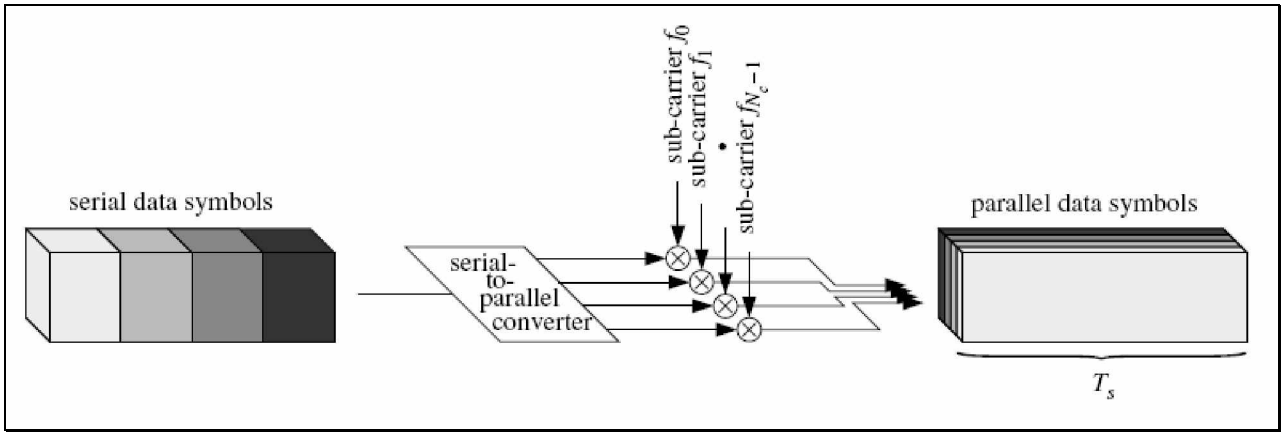


Figure 1.9 Modulation multiporteuses avec N=4 sous canaux

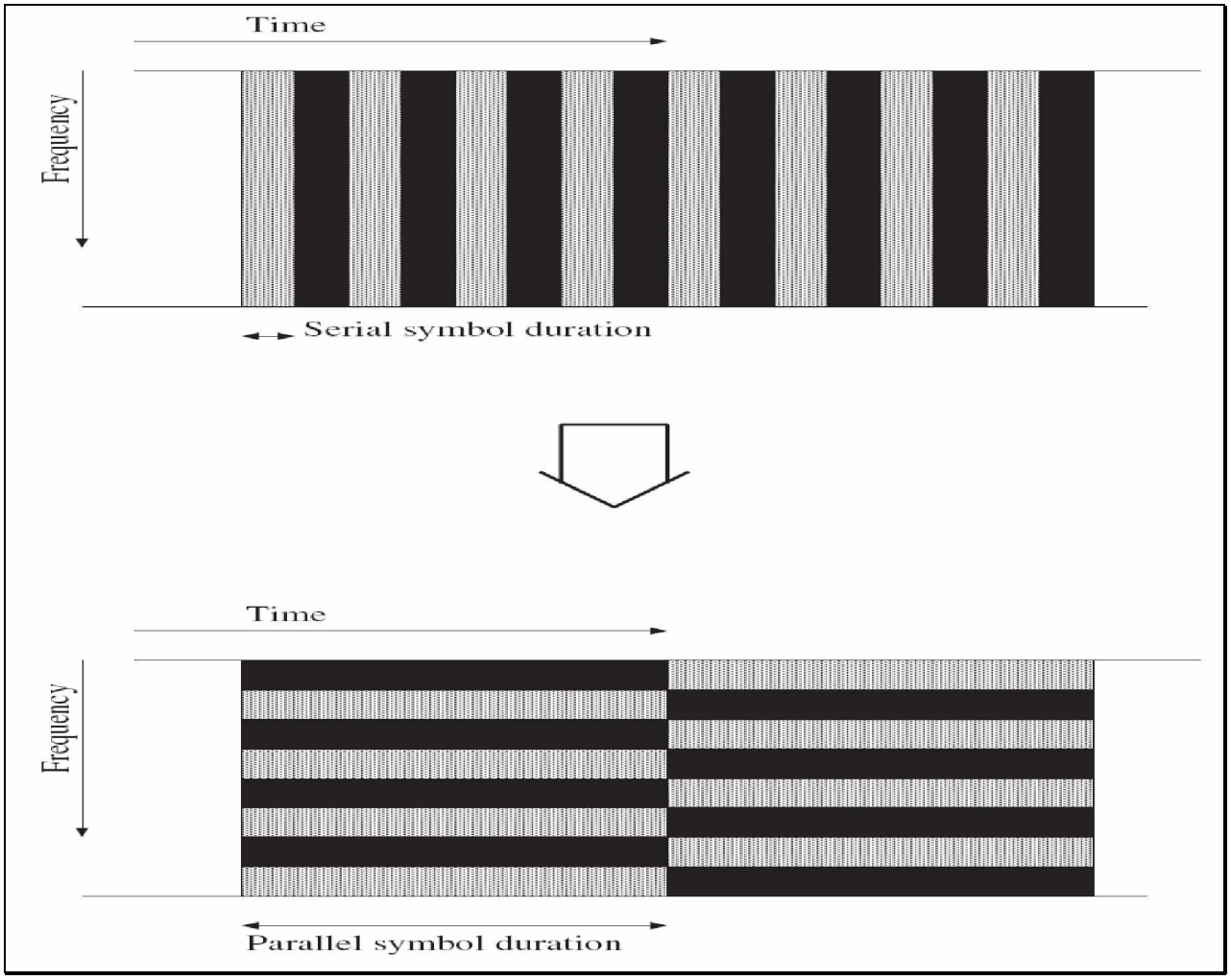


Figure 1.10 Conversion série – paralelle

1.3.3 Modulateur OFDM

L'OFDM se compose d'une somme de sous porteuses. les techniques de modulation utilisées par l'OFDM sont la PSK (Phase Shift Key) et la QAM (Quadrature Amplitude Modulation) (figure1.12). Un modulateur OFDM est représenté par la figure1.13.

Figure 1.11 Différents type de codage numérique

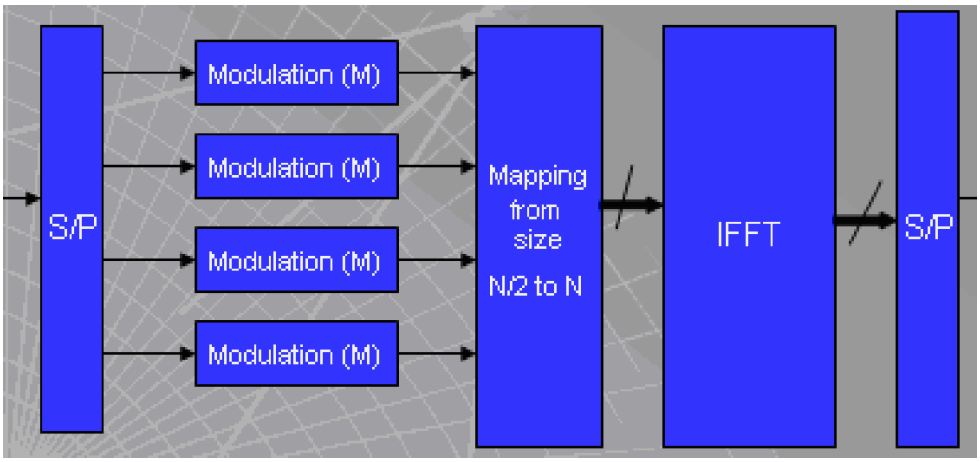


Figure 1.12 Modulateur OFDM

1.3.3.1 Modulation des sous porteuses

Les sous porteuses sont modulées par les transcodes mapping (position des symboles dans la constellation). Chaque transcode est un nombre complexe représenté par un vecteur (vecteur IQ) dans la constellation.

La figure 1.14 montre un exemple de modulation des sous porteuses en utilisant le schéma de modulation QAM. Dans ce cas, chaque sous porteuse porte 2 bits par symbole d'information.

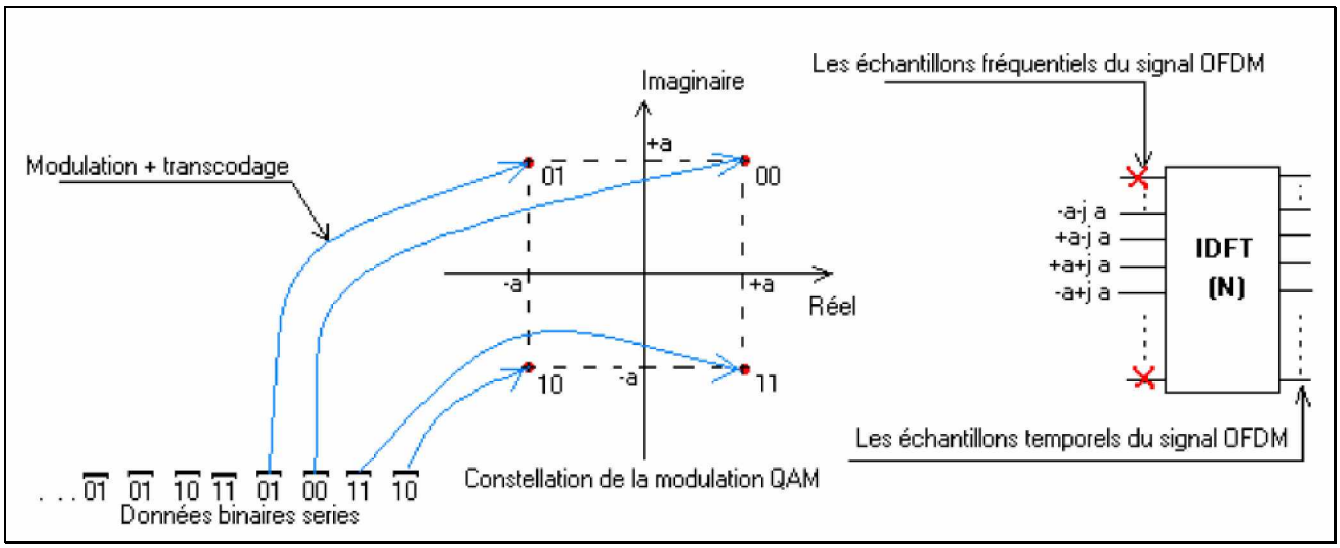


Figure 1.13 Modulation des sous porteuses

La figure 1.15 donne un autre exemple de modulation des sous porteuses. Cet exemple montre le schéma de modulation 16-QAM. Chaque combinaison de 4 bits correspond à un seul vecteur IQ.

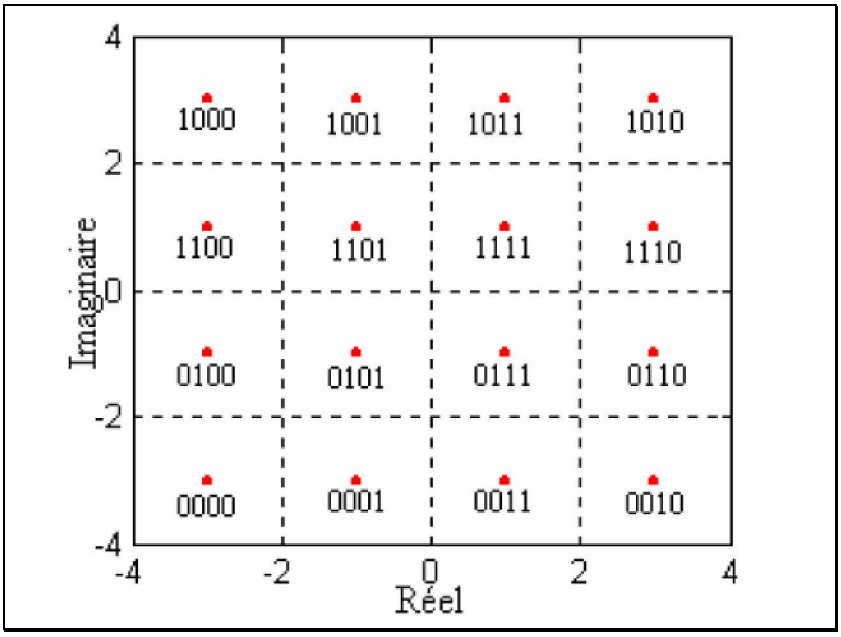


Figure 1.14 Constellation 16-QAM avec codage de GRAY

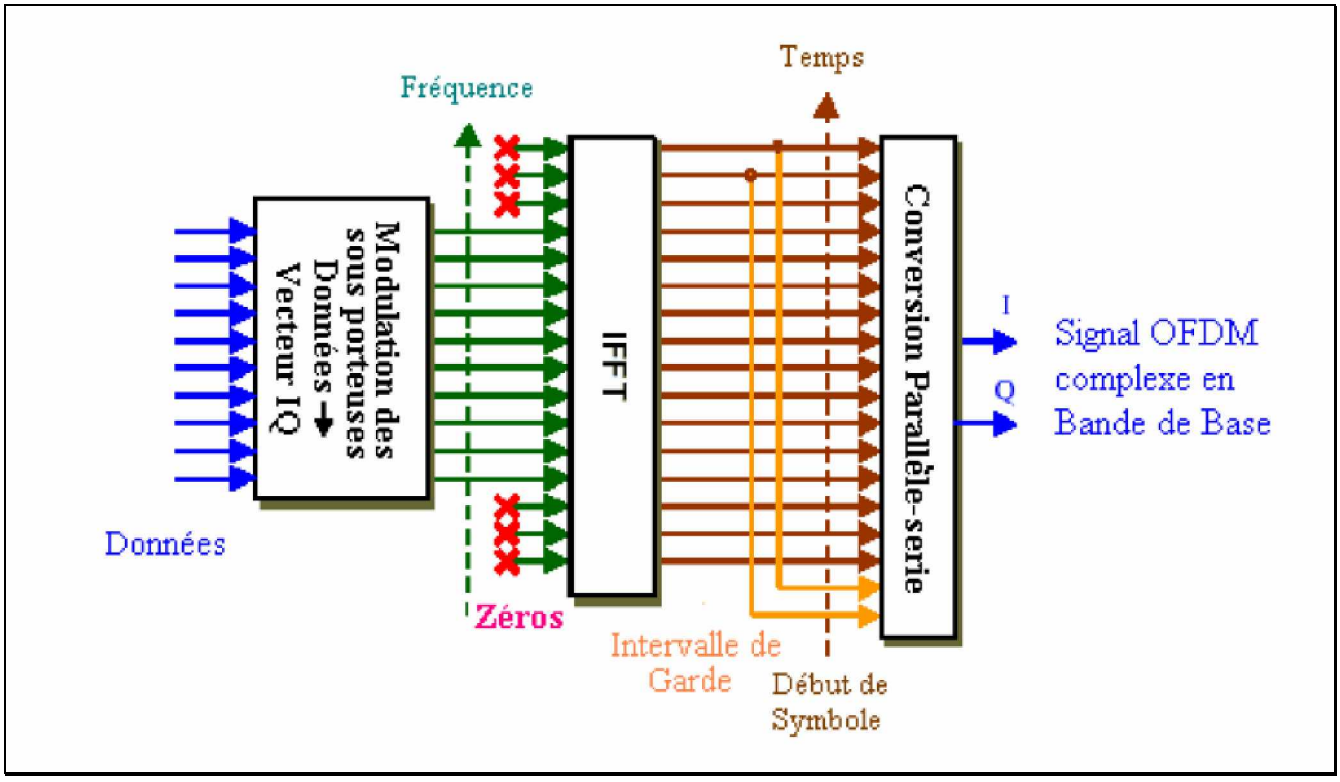


Figure 1.15 Génération de l'OFDM

1.3.4 Conversion du domaine fréquentiel au domaine temporel

Après l'étape du mapping, les sous porteuses sont mis à des amplitudes et phases basées sur les données à transmettre et le schéma de modulation utilisé ; toutes les sous porteuses non utilisées sont

mises à zéro. Le signal OFDM est établi dans le domaine fréquentiel. L'IFFT est utilisée pour convertir ce signal au domaine temporel. La figure 1-16 montre la partie IFFT de l'émetteur OFDM.

Dans le domaine fréquentiel, avant l'application de l'IFFT, chaque échantillon de l'IFFT correspond à une seule sous porteuse. La plupart des sous porteuses sont modulées par les données.

Les sous porteuses périphériques ne sont pas modulées, c'est-à-dire elles sont mises à zéro. Ces sous porteuses nulles permettent l'insertion de l'intervalle de garde fréquentiel avant la fréquence de coupure du filtre de reconstruction passe bas.

1.3.5 Intervalle de garde

L'effet de l'ISI sur le signal OFDM peut être minimisé davantage par l'addition d'un intervalle de garde (guard period) au début de chaque symbole OFDM. Cet intervalle est une copie de la fin du symbole OFDM, qui prolonge la durée du symbole OFDM (figure 1-17). **[1]**

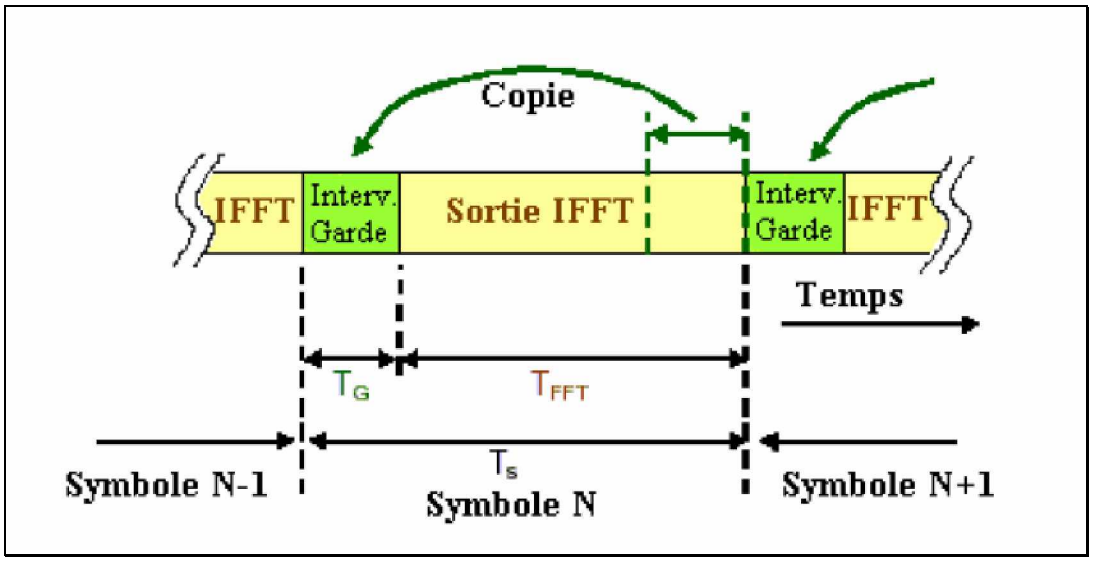


Figure 1.16 Ajout de l'intervalle de garde au signal OFDM

La nouvelle durée totale du symbole OFDM est $T_S = T_G + T_{FFT}$,

Où T_G est la durée de l'intervalle de garde ajouté.

T_{FFT} est la durée initiale du symbole généré par l'IFFT.

L'intervalle de garde doit être plus grand que le retard de la diffusion prévue.

1.3.6 Modulation RF

La Modulation RF utilisée par les systèmes OFDM est la Modulation d'Amplitude en Quadrature. Deux porteuses RF sont modulées par la partie réelle (canal I) et la partie imaginaire (canal Q) du signal OFDM temporel.

Le modulateur RF peut être implémenté en utilisant une technique numérique (Digital Up Converter) comme indiqué dans la figure1.18. La performance de la modulation numérique tend à être meilleure grâce à l'équilibre amélioré entre le traitement des canaux I et Q, et à l'exactitude de la phase du modulateur IQ numérique.

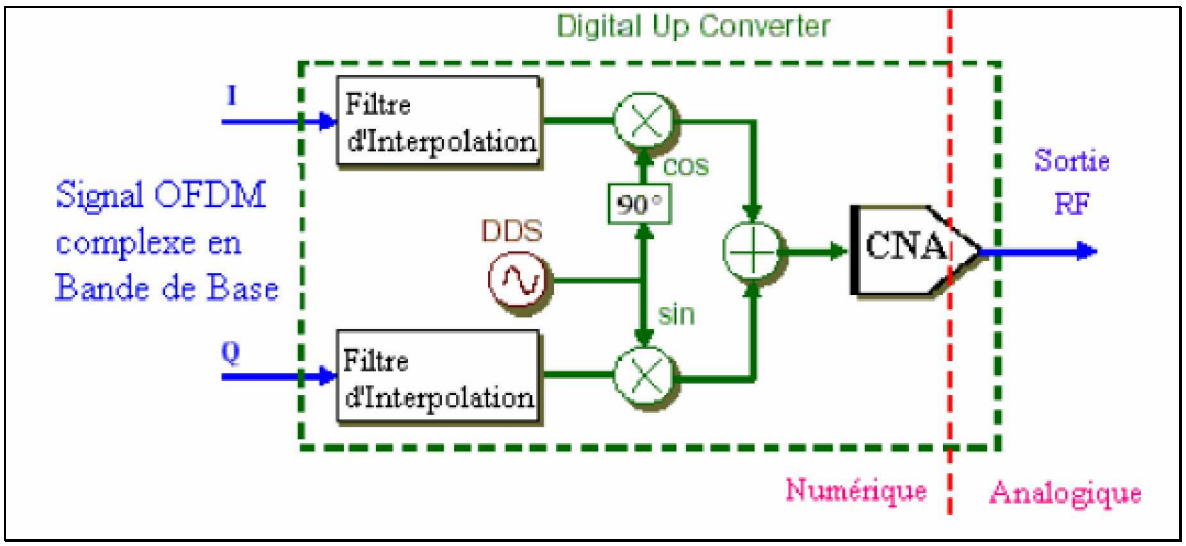


Figure 1-18 Modulation RF technique numérique

1.4 Conclusion

Nous avons présenté la modulation par répartition orthogonale de fréquence, qui pourrait être considérée comme technique de modulation ou d'accès multiple. Les caractéristiques principales qui définissent un système OFDM sont:

- le nombre de sous porteuses,
- le nombre de points du FFT,
- largeur de bande, durée de symbole, l'intervalle de garde et type de modulation.
- D'autres paramètres relatifs qui sont: le nombre et la position des pilotes de porteuses, la durée de FFT et la méthode de codage de FEC.

Notre travail sera basé sur l'implémentation de la partie émettrice de ce système OFDM sur une carte FPGA.

2 CHAPITRE II : Développement d'un projet sur un circuit FPGA

2.1 Introduction

Les FPGAs (Field Programmable Gate Array) sont des circuits à *architecture programmable* qui ont été inventés par la société XILINX en 1985. Les FPGAs sont bien distincts des autres familles de circuits programmables tout en offrant le plus haut niveau d'intégration logique. Ce sont des circuits:

- Ø *Entièrement configurables* par programmation qui permettent d'implanter physiquement, par simple programmation, n'importe quelle fonction logique.
- Ø Ne demandent donc pas de fabrication spéciale en usine, ni de systèmes de développement coûteux.
- Ø Ceci permet de les reprogrammer à volonté afin d'accélérer notablement certaines phases de calculs.
- Ø De plus, ils *ne sont pas limités à un mode de traitement* séquentiel de l'information comme avec les *microprocesseurs* ; et en cas d'erreur, ils sont reprogrammables électriquement *sans avoir à extraire* le composant de son environnement.
- Ø Un autre avantage de ces circuits est leur grande *souplesse* qui permet de les réutiliser à volonté dans des algorithmes différents en un temps très court (quelques millisecondes).
- Ø Ainsi, les FPGAs occupent une position de compromis entre PLDs et ASICs parce que leur fonctionnalité peut être adaptée aux besoins du client dans le labo comme les PLDs, en même temps ils peuvent contenir des millions de portes logiques, et être employé pour une application extrêmement grande et complexe, qu'on ne pourrait réalisés seulement qu'en utilisant des ASICs. Sans oublier le coût inférieur d'une conception FPGA à celui de l'ASIC.

Au début des années 2000, des FPGAs à rendement élevé contenant des millions de portes étaient disponible. Certains de ces dispositifs ont intégré des noyaux de microprocesseur, des interfaces d'entrée/sortie *(I/O)* à grande vitesse. Le résultat final est que le FPGA d'aujourd'hui peut être employé pour n'importe quelle application, y compris des dispositifs de communications, le SDR (Software Defined Radio), les radars, imagerie, et d'autres applications de traitement numérique de signal (DSP).

Actuellement deux Leaders mondiaux se disputent le marché, Altéra et Xillinx. De nombreux autres fabricants de moindre envergure, proposent également leurs propres produits.

L'objet de notre étude est d'implémenter l'émetteur OFDM déjà cité dans le chapitre I sur un circuit FPGA. La programmation a été faite en utilisant le langage de description VHDL. Nous allons

d'abord faire une description des circuits FPGA, ensuite on va introduire les étapes nécessaires au développement d'un projet sur un circuit FPGA, de la programmation jusqu'au chargement sur la carte, et on terminera par le langage VHDL. Nous nous intéresserons aux derniers circuits présents sur le marché à savoir le circuit FPGA de la famille Virtex-II. Cette dernière présente une densité d'intégration de [4 à 10 millions de portes], avec une vitesse dépassant les 420 Mhz.

2.2 Circuits FPGA

2.2.1 FPGA

De nombreuses familles de circuits programmables et reprogrammables sont apparues depuis les années 70 avec des noms très divers suivant les constructeurs. La figure II.1 donne une classification possible des circuits numériques en précisant où se situent les circuits FPGA dans cette classification.

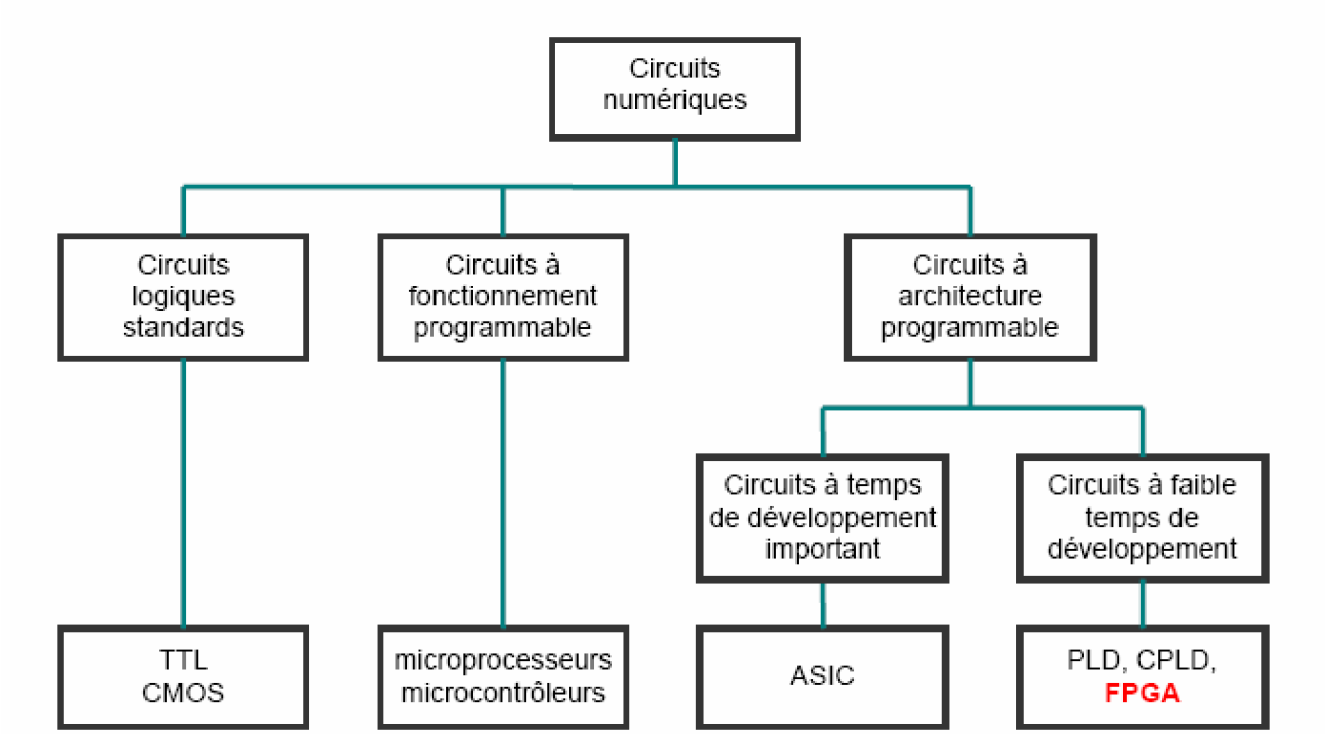


Figure 2.1 Classification des circuits numériques

Les FPGA (Field Programmable Gate Array) sont *des circuits intégrés numériques (ICs)* qui contiennent des blocs de logique et d'interconnections configurables (programmables). Les ingénieurs d'études peuvent les configurer (programmer) de tels façons pour exécuter une variété énorme de tâches. Selon la voie de laquelle ils sont mis en application, certain FPGA peuvent être programmé seulement une seule fois, alors que d'autres peuvent être reprogrammés à plusieurs reprises encore. Pas

étonnamment, qu'un dispositif qui peut être programmé seulement une fois est désigné sous le nom *de programmable jetable (OTP=One-time programmable)*.

La partie "**Field Programmable**" du nom de FPGA se rapporte au fait que sa programmation aura lieu "sur place" (par opposition aux dispositifs dont la fonctionnalité interne est câblée par le constructeur). Ceci peut signifier que les FPGAs peuvent être configurés dans un laboratoire, ou par un utilisateur par un simple système électronique. Si un dispositif est capable d'être programmé *sans l'extraire* de son environnement, il est désigné sous le nom *in-system programmable (ISP)* [11].

2.2.2 Architecture des circuits FPGA

2.2.2.1 Architecture retenue par Xilinx

Les circuits FPGA possèdent une structure matricielle de deux types de blocs (ou cellules). Des blocs d'entrées/sorties et des blocs logiques programmables. Le passage d'un bloc logique à un autre se fait par un routage programmable. Certains circuits FPGA intègrent également des mémoires RAM, des multiplieurs et même des noyaux de processeurs.

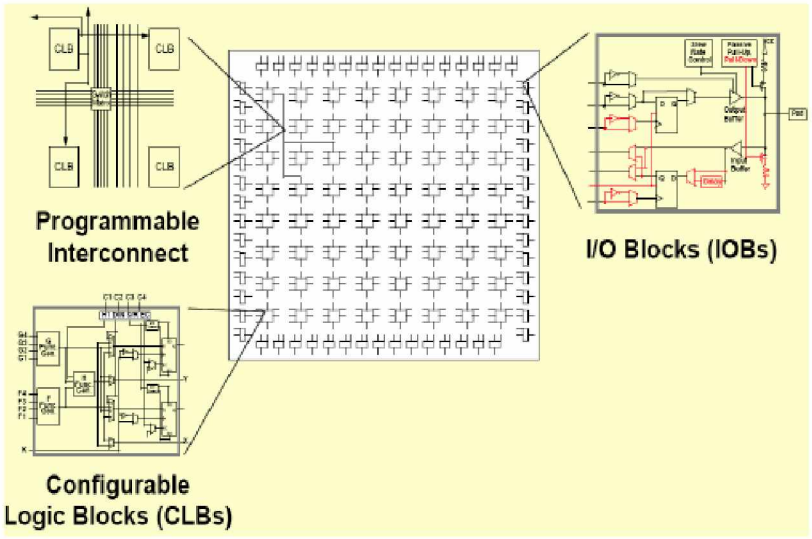


Figure 2.2 Architecture interne du FPGA

Dans ce qui suit, on va faire une description de l'architecture utilisée par Xilinx, car c'est sur des circuits Xilinx qu'on va implémenter nos programmes.

L'architecture retenue par Xilinx se présente sous forme de deux couches :

- Une couche appelée ***Circuit Configurable***,
- Une couche réseau mémoire ***SRAM*** (Static Read Only Memory).

Ø Circuit configurable

La couche dite « circuit configurable » est constituée:

- ü d'une matrice de **blocs logiques configurables** (**CLB**) permettant de réaliser des fonctions combinatoires et des fonctions séquentielles (figure 2.3).
- ü Tout autour de ces blocs logiques configurables, on trouve des **blocs d'entrées/sorties** (**IOB**) dont le rôle est de gérer les entrées-sorties réalisant l'interface avec les modules extérieurs (figure 2.2.).

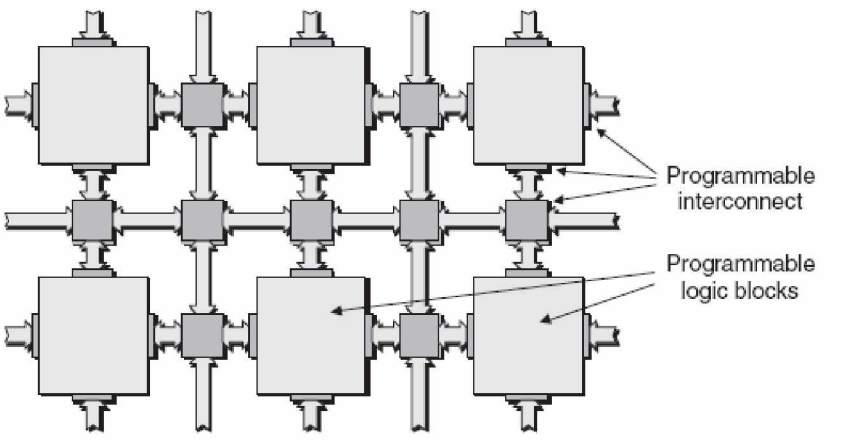


Figure 2.3 Blocs et Interconnections programables

Ø **Réseau mémoire SRAM**

La programmation du circuit FPGA, appelé aussi LCA (Logic Cells Arrays), consistera en l'application d'un potentiel adéquat sur la grille de certains transistors à effet de champ servant à interconnecter les éléments des CLB et des IOB, afin de réaliser les fonctions souhaitées et d'assurer la propagation des signaux. Ces potentiels sont tout simplement mémorisés dans le réseau mémoire SRAM (figure 2.4).

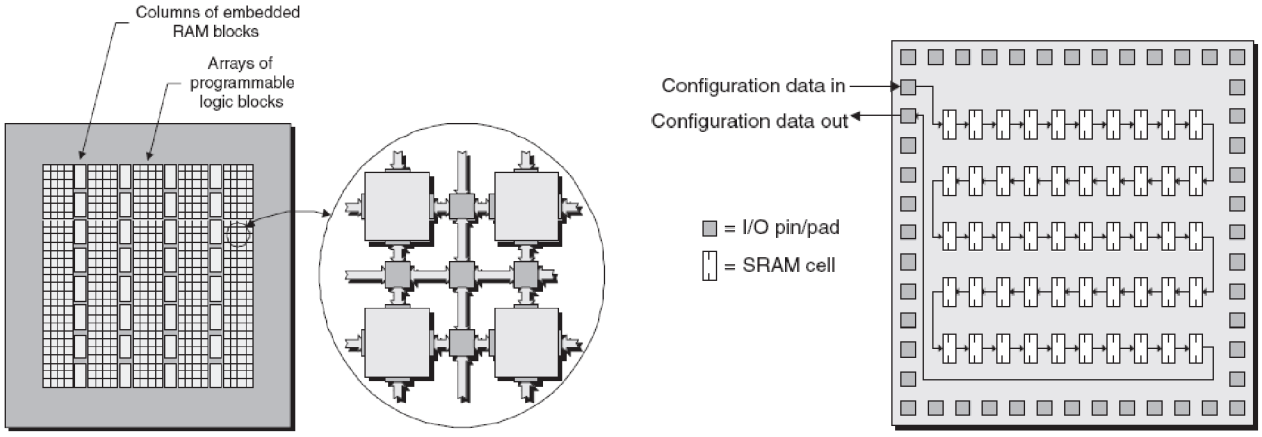


Figure 2.4 Situation du réseau SRAM

La programmation d'un circuit FPGA est volatile, la configuration du circuit est donc mémorisée sur la couche réseau SRAM et stockée dans une ROM externe. Un dispositif interne permet à chaque mise sous tension de charger la SRAM interne (figure 2-5) à partir de la ROM. Ainsi on conçoit

aisément qu'un même circuit puisse être exploité successivement avec des ROM différentes puisque sa programmation interne n'est jamais définitive. On voit ici tout le profit que l'on peut tirer de cette souplesse en particulier lors d'une phase de mise au point. Une erreur n'est pas rédhibitoire, mais peut aisément être réparée.

La mise au point d'une configuration s'effectue en deux temps : Une première étape purement logicielle va consister à dessiner puis simuler logiquement le circuit fini. Dans la seconde étape, on effectuera une simulation matérielle en configurant un circuit réel. On pourra alors vérifier si le fonctionnement réel correspond bien à l'attente du concepteur, et si besoin est identifier les anomalies liées généralement à des temps de transit réels légèrement différents de ceux supposés lors de la simulation logicielle, ce qui peut conduire à des états instables voire même erronés.

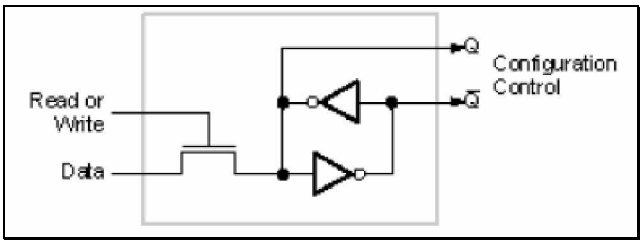


Figure 2.5 Structure d'un cellule SRAM

Les circuits FPGA du fabricant Xilinx utilisent deux types de cellules de base : les cellules d'entrées/sorties appelés IOB (Input Output Bloc), et les cellules logiques appelées CLB (Configurable Logic Bloc). Ces différentes cellules sont reliées entre elles par un réseau d'interconnexions configurable. On décrit dans ce qui suit chacun de ces composants.

2.2.2.2 CLB (Configurable Logic Bloc)

Les blocs logiques configurables sont les éléments déterminants des performances du circuit FPGA (figure 2.6). Chaque CLB est un bloc de logique combinatoire composé de générateurs de fonctions à quatre entrées (LUT) et d'un bloc de mémorisation/synchronisation composé de bascules D. Quatre autres entrées permettent d'effectuer les connexions internes entre les différents éléments du CLB.

La LUT (Look Up Table) est un élément qui dispose de quatre entrées, il existe donc $2^4 = 16$ combinaisons différentes de ces entrées. L'idée consiste à mémoriser la sortie correspondant à chaque combinaison d'entrée dans une petite table de 16 bits, la LUT devient ainsi un petit bloc générateur de fonctions. La figure2.7 montre le schéma simplifié d'un CLB de la famille XC4000 de Xilinx.

Dans cette famille, la cellule de base contient deux LUTs à 4 entrées qui peuvent réaliser deux fonctions quelconques à 4 entrées. Une troisième LUT peut réaliser une fonction quelconque à 3

entrées à partir des sorties des deux premières LUT (F' et G' qui deviennent H2 et H3) et d'une troisième variable d'entrée H1 sortant du bloc « sélecteur ». Le bloc sélecteur contient 4 signaux de contrôle : 3 signaux dédiées pour les registres : une donnée "Din", un signal de validation "Ec" et une remise à un ou à zéro asynchrone "S/R", et le 4ème signal représente l'entrée H1 de la LUT à 3 entrées.

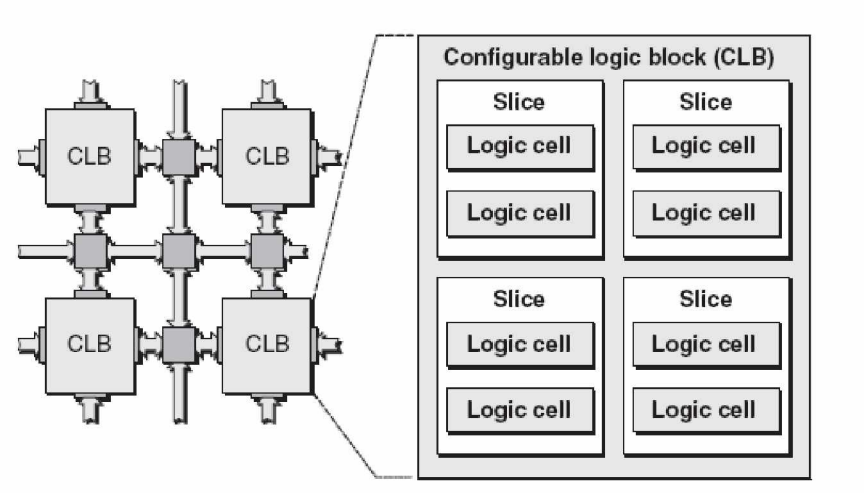


Figure 2.6 Bloc CLB

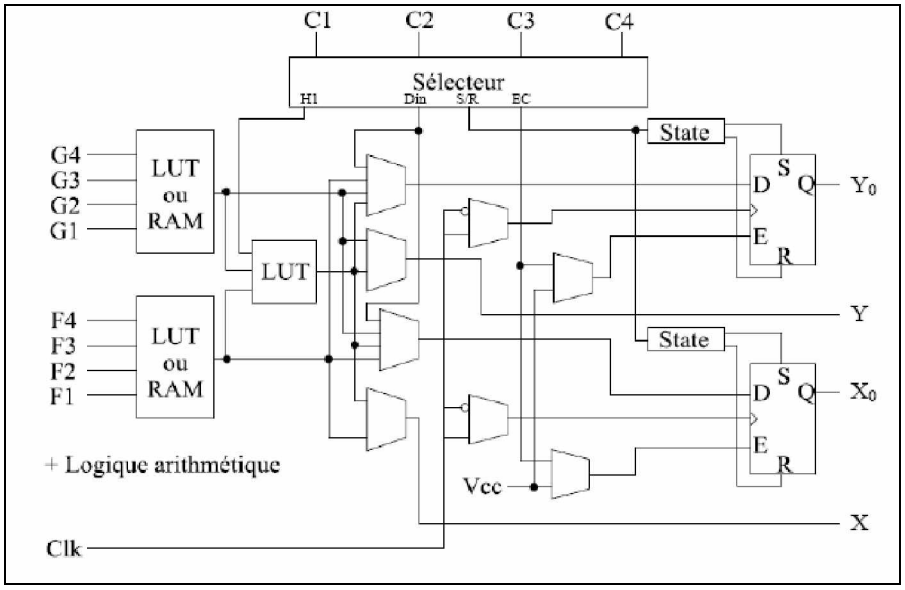


Figure 2.7 Schéma d'une cellule logique (XC4000 de Xilinx)

Les signaux des générateurs de fonction peuvent sortir du CLB, soit par la sortie X pour les fonctions F' et H', soit Y pour les fonctions G' et H'. Ainsi un CLB peut être utilisé pour réaliser:

-deux fonctions indépendantes à 4 entrées indépendantes ;

- ou une seule fonction à 5 variables ;

- ou deux fonctions, une à 4 variables et une autre à 5 variables.

Les sorties de ces blocs logiques peuvent être appliquées à des bascules au nombre de deux ou

directement à la sortie du CLB (sorties X et Y). Chaque bascule présente deux modes de fonctionnement : un mode « flip-flop » avec comme donnée à mémoriser, soit l'une des fonctions F', G', H' ; soit l'entrée directe Din. La donnée peut être mémorisée sur un front montant ou descendant de l'horloge (Clk). Les sorties de ces deux bascules correspondent aux sorties du CLB X_0 et Y_0.

Un mode dit de « verrouillage » exploite l'entrée S/R qui peut être programmée soit en mode SET, mise à 1 de la bascule, soit en Reset, mise à zéro de la bascule. Ces deux entrées coexistent avec une autre entrée, qui n'est pas représentée sur la figure2.7, et qui est appelée le global Set/Reset. Cette entrée initialise le circuit FPGA à chaque mise sous tension, à chaque configuration, en commandant toutes les bascules au même instant soit à '1', soit à '0'. Elle agit également lors d'un niveau actif sur le fil RESET lequel peut être connecté à n'importe quelle entrée du circuit FPGA.

L'idée de cette architecture consiste à pouvoir modifier le contenu des mémoires des LUT en cours de fonctionnement. En effet, les LUT ne sont rien d'autre que des petites RAM qui étaient configurées au démarrage ; on peut alors les utiliser comme des petites mémoires de 16x1 bits. Un mode optionnel des CLB est donc la configuration en mémoire RAM de 16x2 bits ou 32x1 bit. Les entrées F1 à F4 et G1 à G4 deviennent des lignes d'adresses sélectionnant une cellule mémoire particulière. La fonctionnalité des signaux de contrôle est modifiée dans cette configuration : les lignes H1, Din et S/R deviennent respectivement les deux données D_0 et D_1 d'entrée (RAM 16x2bits) et le signal de validation d'écriture WE. Le contenu de la cellule mémoire (D_0 et D_1) est accessible aux sorties des générateurs de fonctions F' et G'. Ces données peuvent sortir du CLB à travers ses sorties X et Y ou alors en passant par les deux bascules.

L'intégration de fonctions à nombreuses variables diminue le nombre de CLB nécessaires et les délais de propagation des signaux ; par conséquent, elle augmente la densité et la vitesse du circuit. Le plus large circuit de cette famille (le circuit XC40250) dispose d'un réseau de 92x92 cellules de base et est équivalent à environ 250.000 portes logiques.

Xilinx propose également des composants à "haute densité" avec les familles Virtex (4 millions de portes) et Virtex II (6 millions de portes). La famille Virtex apporte plusieurs nouveautés par rapport à la famille XC4000 :

• l'adjonction de blocs mémoires de 4 Kbits au cœur de la logique et même de plus larges mémoires dans la famille "Extended Memory".

• l'utilisation de boucles à verrouillage de phase améliorées : DLL (Digital Delay Locked Loop) qui permettent de synchroniser une horloge interne sur une horloge externe, de travailler en quadrature de phase et de multiplier ou diviser la fréquence.

• La présence d'un anneau de connexions autour du circuit pour faciliter le routage des entrées-sorties.

• La compatibilité avec de nombreux standards de transmission de données et de niveaux logiques.

L'architecture des cellules logiques est toujours basée sur des LUT à 4 entrées configurables également en petites mémoires RAM 16 bits. De plus, de la logique a été ajoutée pour permettre la synthèse de plus larges LUT comme une combinaison des LUT existantes. Le plus large circuit de cette famille (le circuit XCV3200E) contient un réseau de 104x156 cellules logiques et est équivalent à environ 4 millions de portes logiques.

Finalement, la famille Virtex II Pro améliore encore un peu le modèle précédent :

• Des blocs de mémoire de 18 Kbits.

• Des multiplieurs signés de 18x18 bits vers 36 bits.

• Des buffers Tri-state en interne pour réaliser des bus.

• Le contrôle des impédances de sortie pour adapter chaque impédance à celle de la piste du circuit imprimée.

• Le DCM (Digital Clock Manager) qui est une évolution du DLL et qui affine encore le déphasage et la synthèse des horloges internes. Le plus large circuit de cette famille (le circuit XC2V10000) contiendra 192 blocs mémoire de 18Kbits, 192 multiplieurs, un réseau de 128x120 cellules logiques et sera équivalent à environ 10 millions de portes logiques (selon Xilinx).

2.2.2.3 IOB (Input Output Bloc)

Ces blocs d'entrée/sortie permettent l'interface entre les broches du composant FPGA et la logique interne développée à l'intérieur du composant.

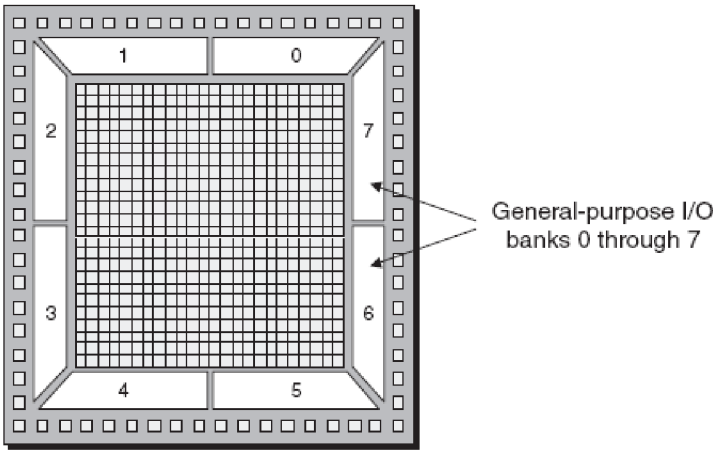


Figure 2.8 Exemple de IOB

Ils sont présents sur toute la périphérie du circuit FPGA. Chaque bloc IOB contrôle une broche du composant et il peut être défini en entrée, en sortie, en signal bidirectionnel ou être inutilisé (état haute impédance). La figure III.5 présente la structure de ces blocs.

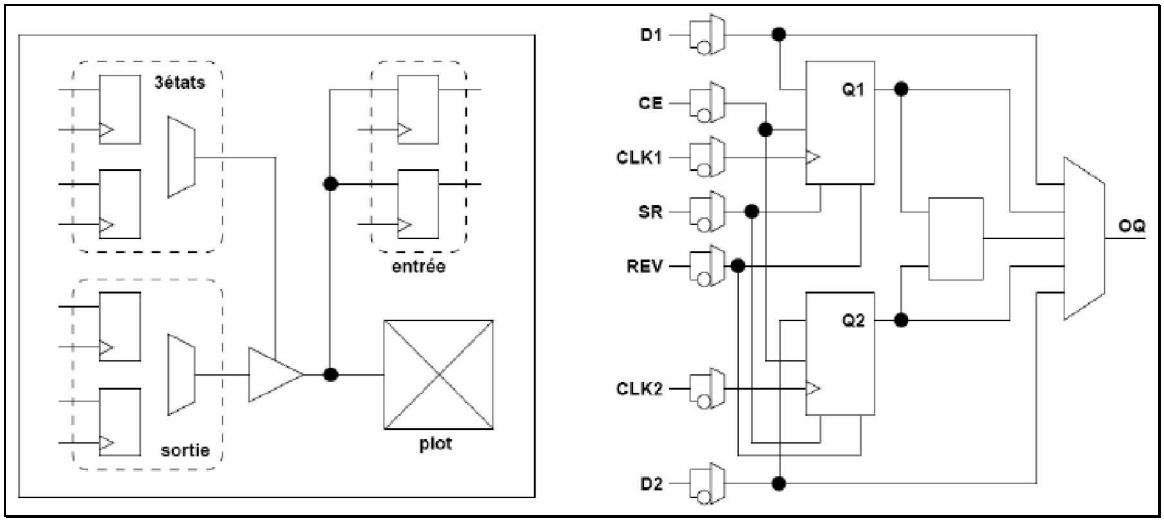


Figure 2.9 Schéma d'un bloc d'entrée/sortie (IOB)

Ø Configuration en entrée

Premièrement, le signal d'entrée traverse un buffer qui, selon sa programmation, peut détecter soit des seuils TTL soit des seuils CMOS. Il peut être routé directement sur une entrée directe de la logique du circuit FPGA ou sur une entrée synchronisée. Cette synchronisation est réalisée à l'aide d'une bascule de type D, le changement d'état peut se faire sur un front montant ou descendant. De plus, cette entrée peut être retardée de quelques nanosecondes pour compenser le retard pris par le signal d'horloge lors de son passage par l'amplificateur. Le choix de la configuration de l'entrée s'effectue grâce à un multiplexeur (program controlled multiplexer). Un bit positionné dans une case mémoire commande ce dernier.

Ø Configuration en sortie

Nous distinguons les possibilités suivantes :

- inversion ou non du signal avant son application à l'IOB ;

- synchronisation du signal sur des fronts montants ou descendants d'horloge ;

- mise en place d'un "pull-up" ou "pull-down" dans le but de limiter la consommation des entrées sorties inutilisées ;

- signaux en logique trois états ou deux états. Le contrôle de mise en haute impédance et la réalisation des lignes bidirectionnelles sont commandés par le signal de commande « Out Enable », lequel peut être inversé ou non.

Chaque sortie peut délivrer un courant de 12mA. Ainsi toutes ces possibilités permettent au concepteur de connecter au mieux une architecture avec les périphériques extérieurs.

2.2.2.4 Différents types d'interconnexions

Les connexions internes dans les circuits FPGA sont composées de segments métallisés. Parallèlement à ces lignes, nous trouvons des matrices programmables réparties sur la totalité du circuit, horizontalement et verticalement entre les divers CLB. Elles permettent les connexions entre les diverses lignes, celles-ci sont assurées par des transistors MOS dont l'état est contrôlé par des cellules de mémoire vive ou RAM (Random Access Memory). Le rôle de ces interconnexions est de relier avec un maximum d'efficacité les blocs logiques et les blocs d'entrées/sorties afin que le taux d'utilisation dans un circuit donné soit le plus élevé possible. Pour parvenir à cet objectif, Xilinx propose trois sortes d'interconnexions selon la longueur et la destination des liaisons. Nous disposons:

- d'interconnexions à usage général,
- d'interconnexions directes,
- de longues lignes.

Ø ***Interconnexions à usage général***

Ce système fonctionne en une grille de cinq segments métalliques verticaux et quatre segments horizontaux positionnés entre les rangées et les colonnes des CLB et des IOB.

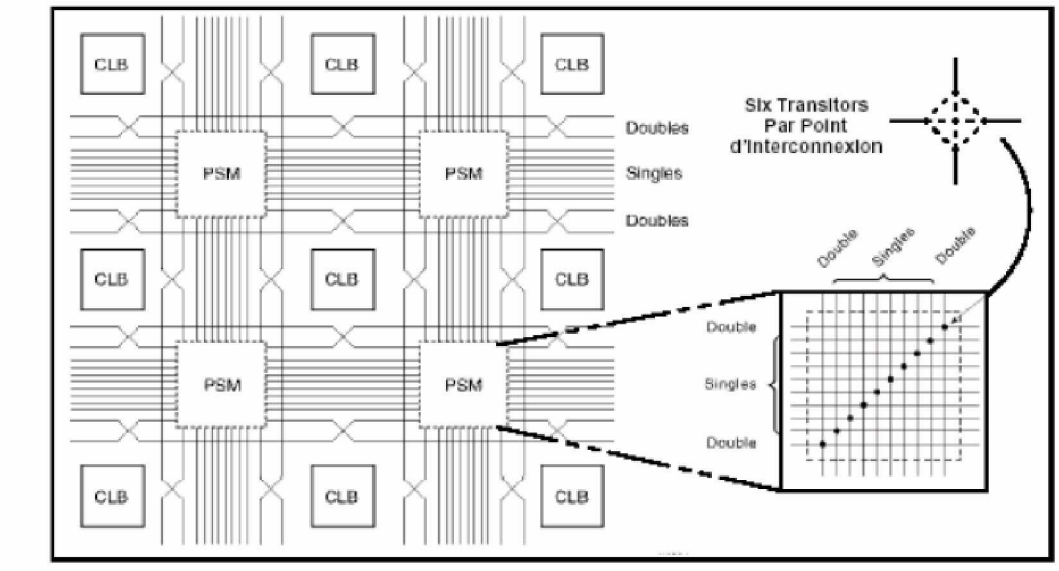


Figure 2.10 Connexions à usage général et détail d'une matrice de commutation

Des aiguilleurs appelés aussi « matrices de commutation » (switch matrix) sont situés à chaque intersection. Leur rôle est de raccorder les segments entre eux selon diverses configurations, ils

assurent ainsi la communication des signaux d'une voie vers l'autre. Ces interconnexions sont utilisées pour relier un CLB à n'importe quel autre CLB. Pour éviter que les signaux traversant les grandes lignes ne soient affaiblis, nous trouvons généralement des buffers implantés en haut et à droite de chaque matrice de commutation.

Ø ***Interconnexions directes***

Ces interconnexions permettent l'établissement de liaisons entre les CLB et les IOB avec un maximum d'efficacité en terme de vitesse et d'occupation du circuit. De plus, il est possible de connecter directement certaines entrées d'un CLB aux sorties d'un autre.

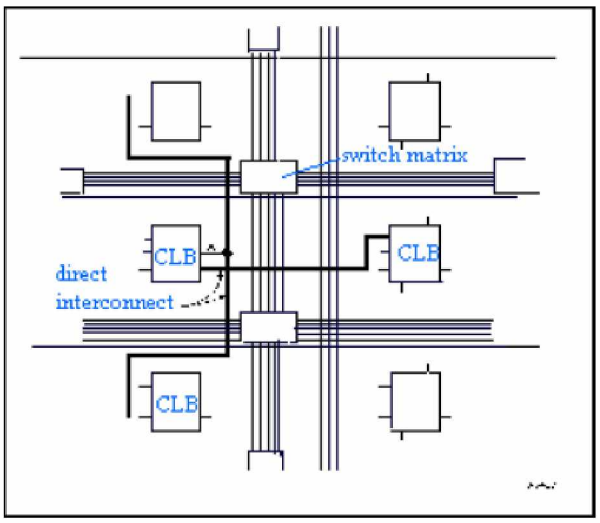


Figure 2.11 Les interconnexions directes

Pour chaque bloc logique configurable, la sortie X peut être connectée directement aux entrées C ou D du CLB situé au-dessus et les entrées A ou B du CLB situé au-dessous. Quant à la sortie Y, elle peut être connectée à l'entrée B du CLB placé immédiatement à sa droite. Pour chaque bloc logique adjacent à un bloc entrée/sortie, les connexions sont possibles avec les entrées I ou les sorties O suivant leur position sur le circuit.

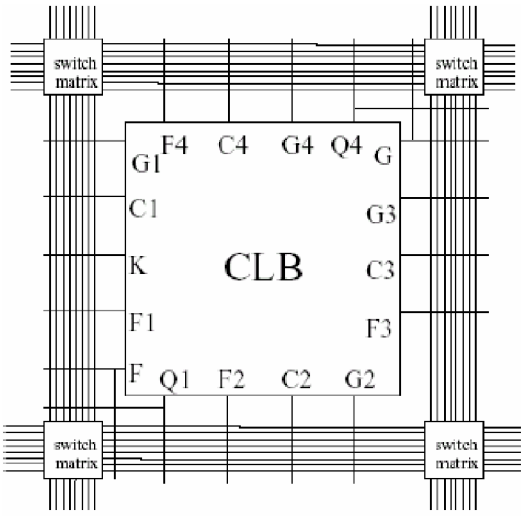


Figure 2.12 Figure : Les longues lignes

Ø ***Longues lignes***

Les longues lignes sont de longs segments métallisés parcourant toute la longueur et la largeur du composant, elles permettent éventuellement de transmettre avec un minimum de retard les signaux entre les différents éléments dans le but d'assurer un synchronisme aussi parfait que possible. De plus, ces longues lignes permettent d'éviter la multiplicité des points d'interconnexion.

Ø ***Performances des interconnexions***

Les performances des interconnexions dépendent du type de connexions utilisées. Pour les interconnexions à usage général, les délais générés dépendent du nombre de segments et de la quantité d'aiguilleurs employés. Le délai de propagation de signaux utilisant les connexions directes est minimum pour une connectique de bloc à bloc. Quant aux segments utilisés pour les longues lignes, ils possèdent une faible résistance mais une capacité importante. De plus, si on utilise un aiguilleur, sa résistance s'ajoute à celle existante

2.2.3 Conclusion

Nous avons vue que le monde de la communication sans fils s'intéresse très particulièrement au FPGA, ceci est du a leur flexibilité en utilisation. Les trois travaux présentés ici on été basé sur différentes normes, mais au cœur de tout ceci on retrouve l'OFDM qui est la technique presque adopté par la nouvelle génération des sans fils.

2.3 Logiciel de développement ISE (Integrated Software Environment) [12] [13]

Dans cette partie nous donnons une description sur notre logiciel de simulation ISE 8.2i. L'objectif visé est de présenter les différents outils disponibles dans l'ISE et la manière de les intégrés ou de les utilisés dans une conception à l'aide d'un dispositif de Xilinx.

ISE contrôle toutes les opérations nécessaires pour la conception. A travers son interface appelé navigateur de projet, on peut accéder à tous les outils susceptibles d'intégré la synthèse, la simulation ou l'implémentation. On peut également accéder à tous les fichiers et documents liés au projet.

2.3.1 Interface du navigateur de projet

L'interface du navigateur de projet est divisée en quatre sous-fenêtres principales, comme le montre la Figure.2.29 Sur la gauche en haut on a la fenêtre de sources qui affiche hiérarchiquement les éléments inclus dans le projet. Sous la fenêtre de sources on a la fenêtre de processus, qui affiche des processus disponibles pour la source actuellement choisie. La troisième fenêtre au bas du navigateur de projet est la fenêtre de transcription qui affiche les messages d'états, d'erreurs, et d'avertissements et contient également un éditeur de commande TCL et la fonction de recherche de fichier. La quatrième fenêtre vers la droite est l'interface d'une fenêtre pour multi document (MDI) référencé comme zone de travail. Elle permet de visualiser des fichiers d'états de type HTML, de textes ASCII, des schémas, et les formes d'onde de la simulation.

- Fenêtre de sources : Cette fenêtre se compose de trois tables qui fournissent des informations pour l'utilisateur.

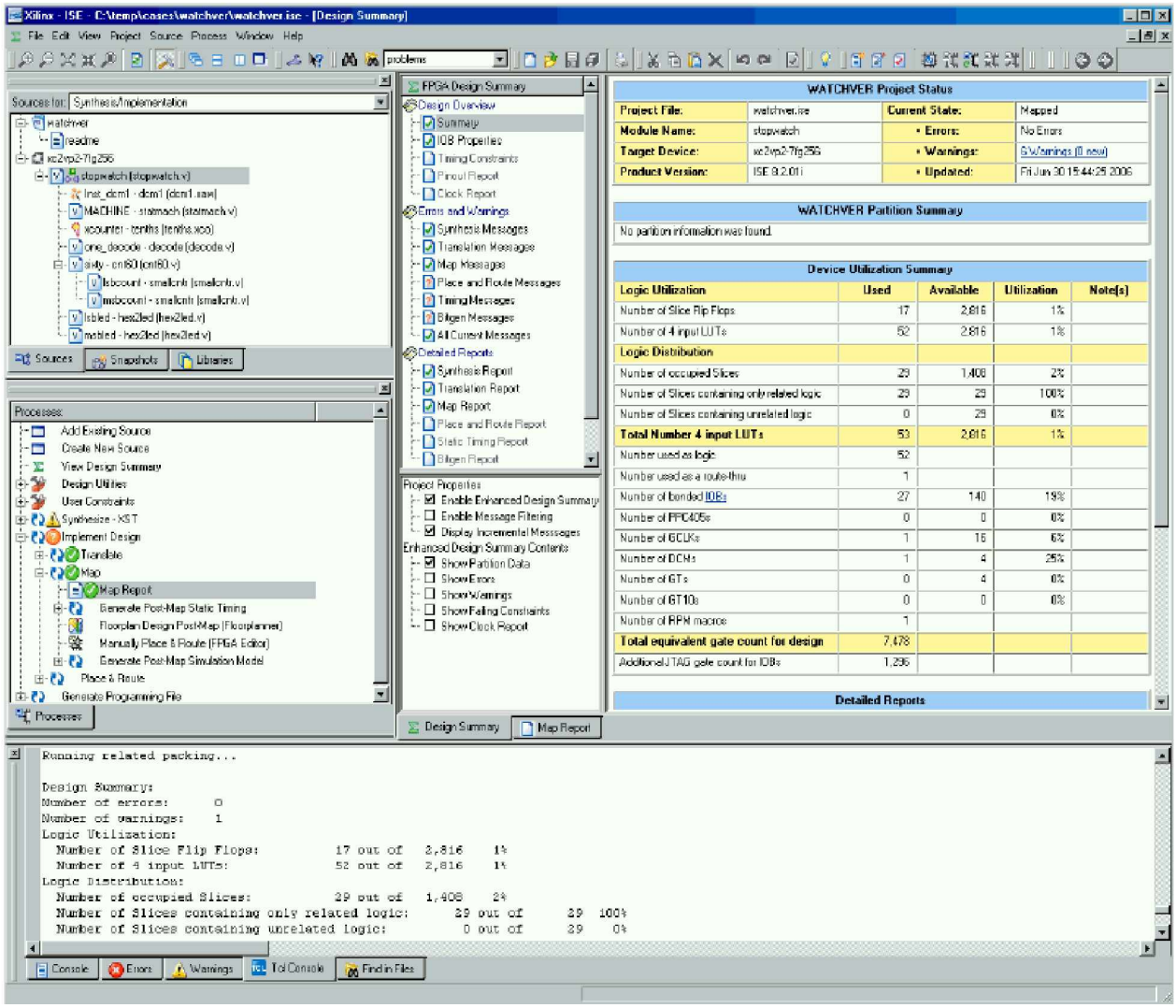


Figure 2.13 Navigateur de projet

ü **Table de Sources** : La table de sources affiche le nom du projet, le dispositif utilisé, les documents utilisés et les fichiers sources de la conception liés à la vue de conception choisie. La vue de conception ("sources for") donne une liste au dessus de la table de sources permettant de visualiser seulement les fichiers sources liés à cette vue, tels que Synthèses/Implémentation, simulation comportementale ou le routage.

Chaque fichier dans une vue de conception a une icône qui lui est associé. L'icône indique le type de fichier (fichier HDL, schéma, noyau, ou fichier texte, par exemple).

ü **Table d'Instantanés** : La table d'instantanés affiche tous les instantanés liés au projet actuellement ouvert dans le navigateur de projet. Un instantané est une copie du projet comprenant tous les fichiers dans le répertoire de fonctionnement, et des sous répertoires de synthèse et de simulation. Un instantané est enregistré avec le projet pour lequel il a été pris, et l'instantané peut être visualisé dans la table d'instantanés. On peut

visualiser les états, les documents utilisés, et les fichiers sources pour tous les instantanés. Toute l'information affichée dans la table d'instantanés est inaltérable. L'utilisation des instantanés fournit un excellent système de commande de version, permettant à des associés de faire un développement simultané sur une même conception.

ü **Table de Bibliothèques** : La table de bibliothèques affiche toutes les bibliothèques liées au projet ouvert dans le navigateur de projet.

Ø Fenêtre de Processus : Cette fenêtre contient une table par défaut appelé la table de processus.

ü **Table de processus** : La table de processus est sensible aux changements basés sur le type de source choisi dans la table de sources et du type du projet. A partir de la table de processus, on peut exécuter les fonctions nécessaires pour définir, exécuter et visualiser la conception. La table de processus permet d'accéder aux fonctions suivantes :

- Ajouter une source existante
- Créer une nouvelle source
- Visualiser le sommaire de la conception
- Les utilitaires d'initialisation des Entrées

Permet d'accéder à la génération de symbole, aux descripteurs d'instanciation, au convertisseur HDL, à la commande du registre de fichiers, et à la compilation de la bibliothèque (pour Modelsim se, le et pe).

- Contraintes utilisateurs : Permet de déterminer le positionnement et le chronométrage des contraintes (contraintes de temps et de paquetage).
- Synthèse : Permet d'accéder au contrôleur de syntaxe, à la synthèse, à la vue RTL ou au schéma de technologie, et tous ce qui se rapportent à la synthèse. Ceci change selon les outils de synthèse qu'on utilise.
- Implémentation de la conception : Permet d'accéder aux outils d'implémentation, aux rapports sur les états de conception, et aux outils associés.
- Génération du fichier programme : Permet d'accéder aux outils de configuration et à la génération de la source binaire.

La table de processus incorpore la technologie « automake ». Ceci permet à l'utilisateur de choisir n'importe quel processus dans l'ensemble de processus et le logiciel exécute automatiquement les processus nécessaires pour l'aboutissement du résultat.

Ø **Fenêtre de transcription** : La fenêtre de transcription contient cinq tables par défaut : Console, erreurs, avertissements, console de TCL, recherche.

- **Console** : Affiche des messages d'erreurs, d'avertissements, et d'information sur le processus en cours. Des erreurs sont signifiées par un (x) rouge à côté du message, alors que les avertissements ont une marque jaune d'exclamation (!).
- **Avertissements** : Permet d'afficher les messages d'avertissement. D'autres messages console sont filtrés dehors.
- **Erreurs** : Affichages des messages d'erreur seulement. D'autres messages console sont filtrés.
- **Console de TCL** : C'est une console interactive d'utilisateur. En plus de l'affichage des messages d'erreurs, d'avertissements et d'autres informations, la console TCL permet à l'utilisateur d'entrer des commandes spécifiques au navigateur de projet TCL.
- **Recherche dans les fichiers** : Affiche les résultats de la recherche quand utilise la commande **edit > find file**.

Ø **Zone de travail**

- **Sommaire de conception** : Le sommaire de conception énumère des informations à un niveau élevé sur le projet, qui comprenant l'information générale, un sommaire sur le dispositif utilisé (nombres de Luts utilisés, nombre de CLBs utilisés et restants…), des données sur la performance obtenue par placement et routage (PAR), l'information sur les contraintes, et l'information récapitulative de tous les rapports liés aux différents états de la conception.
- **Éditeur de Texte** : Des fichiers sources et d'autres documents texte peuvent être ouverts dans un éditeur. L'éditeur par défaut est l'éditeur de texte d'ISE. L'éditeur de texte d'ISE permet d'éditer des fichiers sources et des documents utilisés. On peut accéder aux descripteurs de langage, qui est un catalogue de langage d'ABEL, de Verilog et de VHDL, et descripteurs de contraintes utilisateur classés, que l'on peut utiliser et modifier dans une conception.
- **Simulateur d'ISE /Éditeur de forme d'onde** : Le simulateur d'ISE/éditeur de forme d'onde est un outil de création de banc d'essai et de test intégré au navigateur de projet. L'éditeur de forme d'onde peut être employé pour décrire graphiquement des stimuli, et produire alors un banc d'essai en VHDL.
- **Éditeur Schématique** : L'éditeur schématique est intégré dans le cadre de navigateur de projet. L'éditeur schématique peut être employé graphiquement pour créer et visualiser des conceptions logiques.

2.3.2 Conception du projet à l'aide du VHDL

Lancer le logiciel ISE

Pour lancer ISE :

Double clic l'icône du navigateur de projet d'ISE sur le bureau ou en faisant **Démarrer> tous les programmes > Xilinx ISE 8.2i > navigateur de projet.**

Créer un nouveau projet

1. À partir du navigateur de projet, choisir **file > new project**.

La nouvelle fenêtre de projet apparaît.

2. Entrez le nom du projet.

3. Choisir un répertoire pour le projet.

4. Vérifient que HDL est choisi comme module de niveau supérieurs et cliquer suivant.

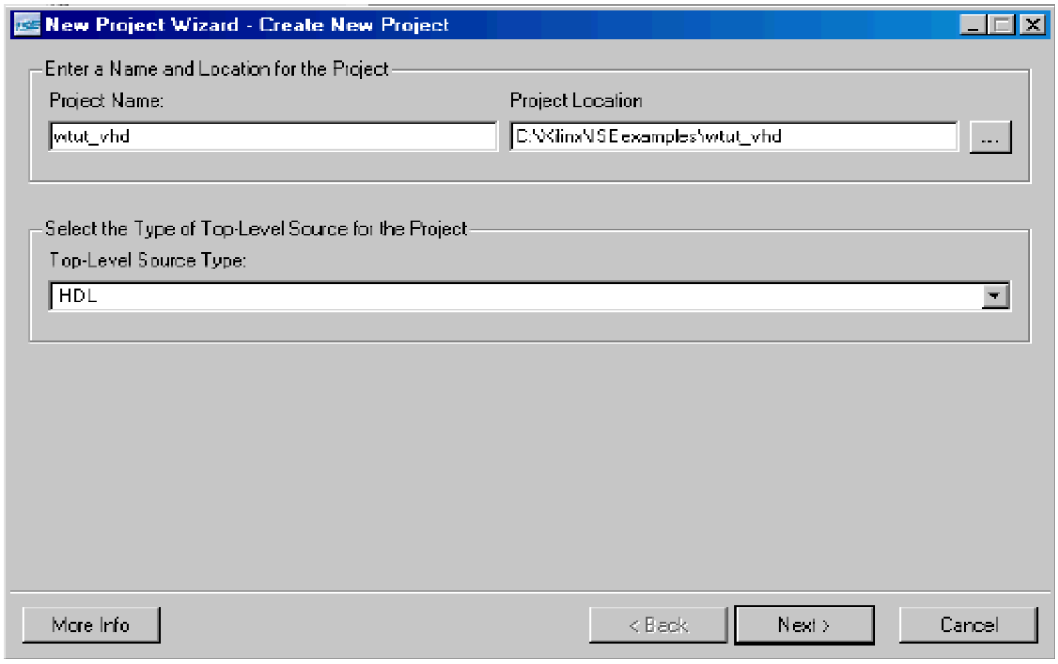


Figure 2.14 Fenêtre de création de projet

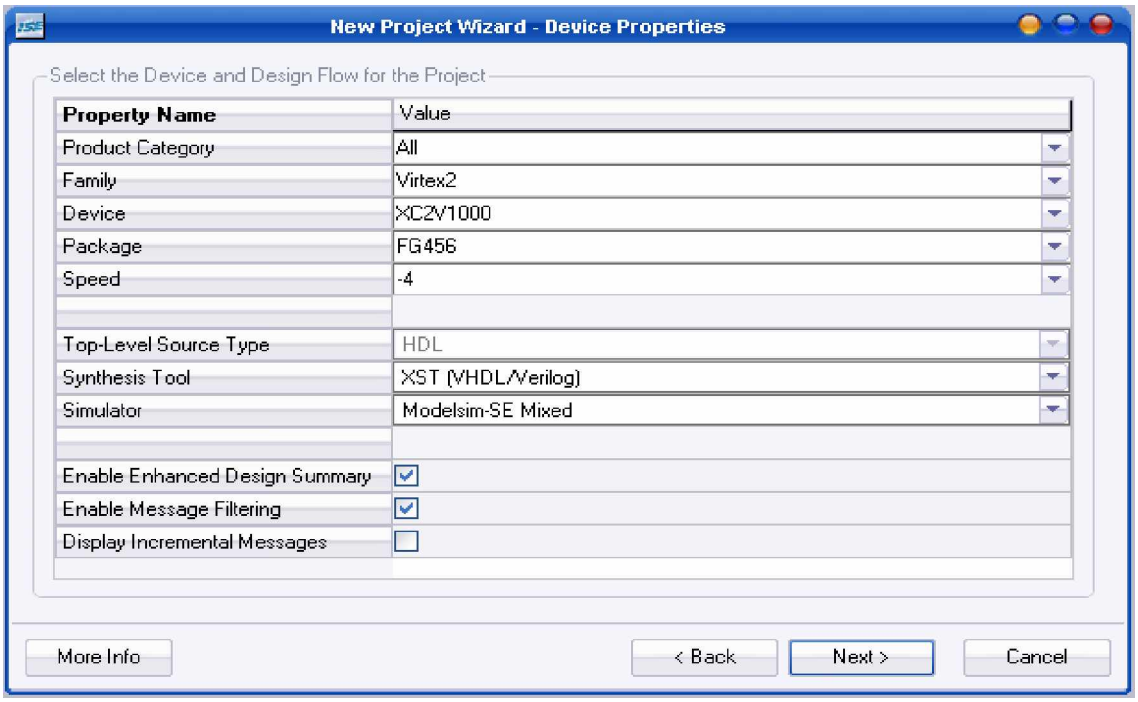


Figure 2.15 les propriétés du dispositif

5. Choisir des valeurs pour le dispositif dans la fenêtre de propriétés de dispositif :

- **Product Category :** Tous
- **Family :** Virtex2
- **Device :** XC2V1000
- **Package :** FG456
- **Speed :** -4
- **outil de Synthesis :** XST (VHDL/Verilog)
- **Simulator :** Modelsim-SE Mixed

Note : Pour l'outil de synthèse on peut si disponible choisir d'autres outils de synthèse qui ne sont pas inclut dans l'ISE mais qui sont prise en charge par l'ISE comme Spectrum synthesis tool, ces outils de synthèse sont à acheter en dehors de l'ISE.

De même pour le simulateur on peut utiliser les simulateurs de mentor graphics qui sont aussi prises en charge par l'ISE et d'ailleurs des versions intérieures de l'ISE ne disposent pas de leur propre simulateur.

6. Cliquer suivant afin de procéder à la création d'une nouvelle source pour le projet.

v Créer une source HDL

Le dossier supérieur de la conception doit être en HDL.

Créer une source VHDL

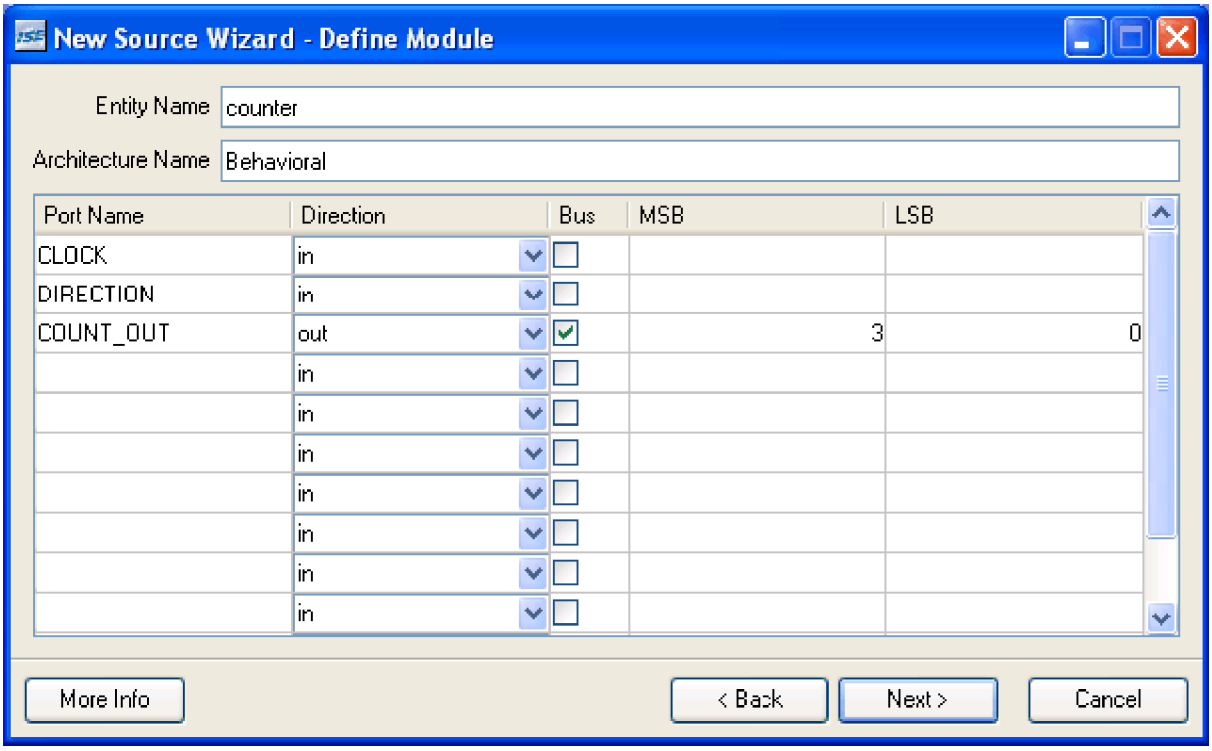


Figure 2.16 l'insertion des ports

Créer un fichier source de VHDL pour le projet comme suit :

1. Cliquer sur **new source** dans la fenêtre.
2. Choisir **Module VHDL** comme type de source.
3. Saisir le nom du fichier.
4. vérifient que **Add to Project** est sélectionné.
5. Cliquer suivant.
6. Déclarer les ports pour la conception en complétant l'information si nécessaire le nombre de bit, MSB (most significant bit) correspond au bit de poids élevé et LSB (least significant bit) au bit de poids faible.
7. Cliquer sur **Next** puis **Finish** pour quitter la fenêtre d'information sur la source.
8. Cliquer sur **Next**, **Next** puis **Finish** pour finir.

On aura le fichier source à compléter.

- **Employer des références de Langue (VHDL)**

Comme dans la plupart des conceptions on utilise généralement une combinaison d'un certains nombre de modules telles que les compteurs synchrones ou asynchrones, les registres, les flips flops, les ROMs et RAMs… C'est dans se sens que Xilinx a développé certaines boîtes qui nous permettent

d'utiliser ces modules comme étant déjà prédéfinis et qu'on aura a adapté à notre projet. Ces modules ont été prédéfinis pour chaque type de langage et pour le type de source. Ainsi on n'a pas besoin de construire des sources pour ces modules, on peut les prendre directement dans l'ISE en utilisant la commande **Template language**.

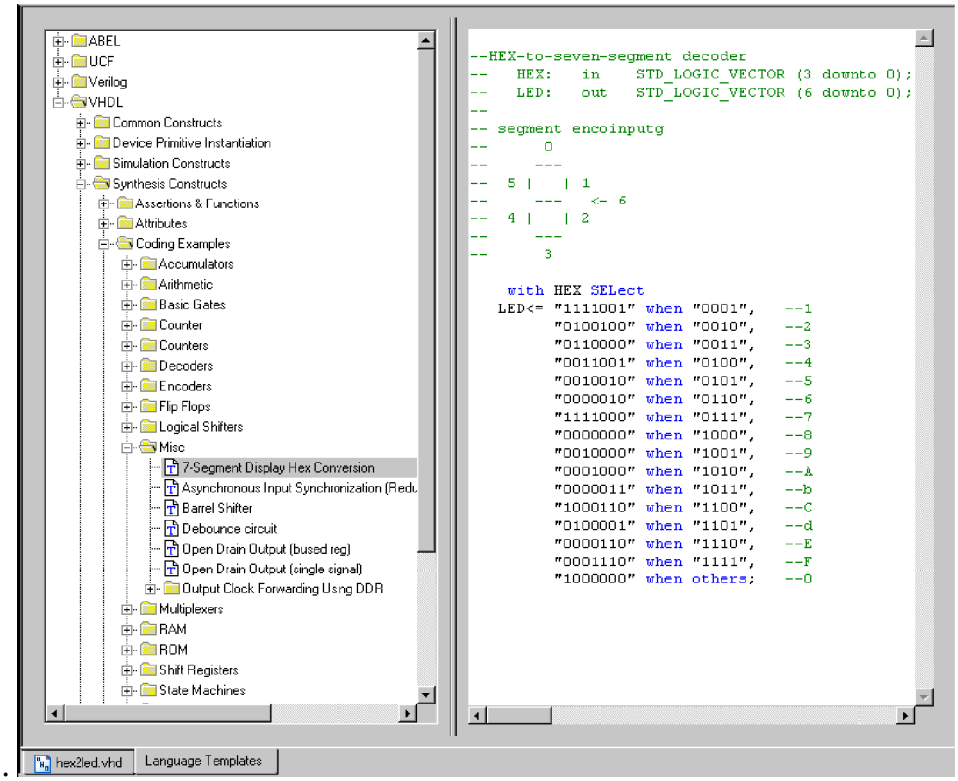


Figure 2.17 L'example d'un codeur Hexadécimal / 7 segments de l'ISE

La prochaine étape explique comment utilisé ces sources

Note : On peut remarquer que les ports sont nommés par défaut HEX et LED.

1. Placer le curseur juste a l'endroit ou l'on veut ajouter le code (généralement en dessous du **Begin** d'**architecture** pour la synthèse).

2. Ouvrir le gabarit des langues en utilisant **Edit** > **Language Templates…**

3. Employer cliquer sur le « +» du langage choisit, et choisir la catégorie de notre code c.-à-d. entité, architecture, attributs…

4. La source sélectionner s'ouvrira et on peut voir le code. Une fois la source sélectionnée faire **Edit>Use in File**, Le code s'affichera directement en dessous de l'endroit ou était placé le curseur.

5. Fermer la fenêtre et revenir au fichier dans le quelle le code a été copié pour l'adapter au reste (Généralement c'est de changé les noms des entrées et des sorties utilisées par défaut par ceux de notre projet).

v Créer un module de générateur de noyau

Le générateur de NOYAU est un outil de conception interactif graphique qui permet de créer les modules à un niveau élevé tels que des compteurs, des registres à décalage, les mémoires, des multiplexeurs et pleins d'autres outils utilisé en traitement de signal comme la FFT. On peut personnaliser et pré-optimaliser les modules pour tirer profit des dispositifs architecturaux inhérents des architectures de Xilinx FPGA, telles que FFT, MÉMOIRE VIVE…

Note : Tous les noyaux ne sont pas adaptés à tous dispositifs de FPGA, il faut donc tenir compte de la famille du dispositif.

Créer un module à l'aide du générateur de noyau.

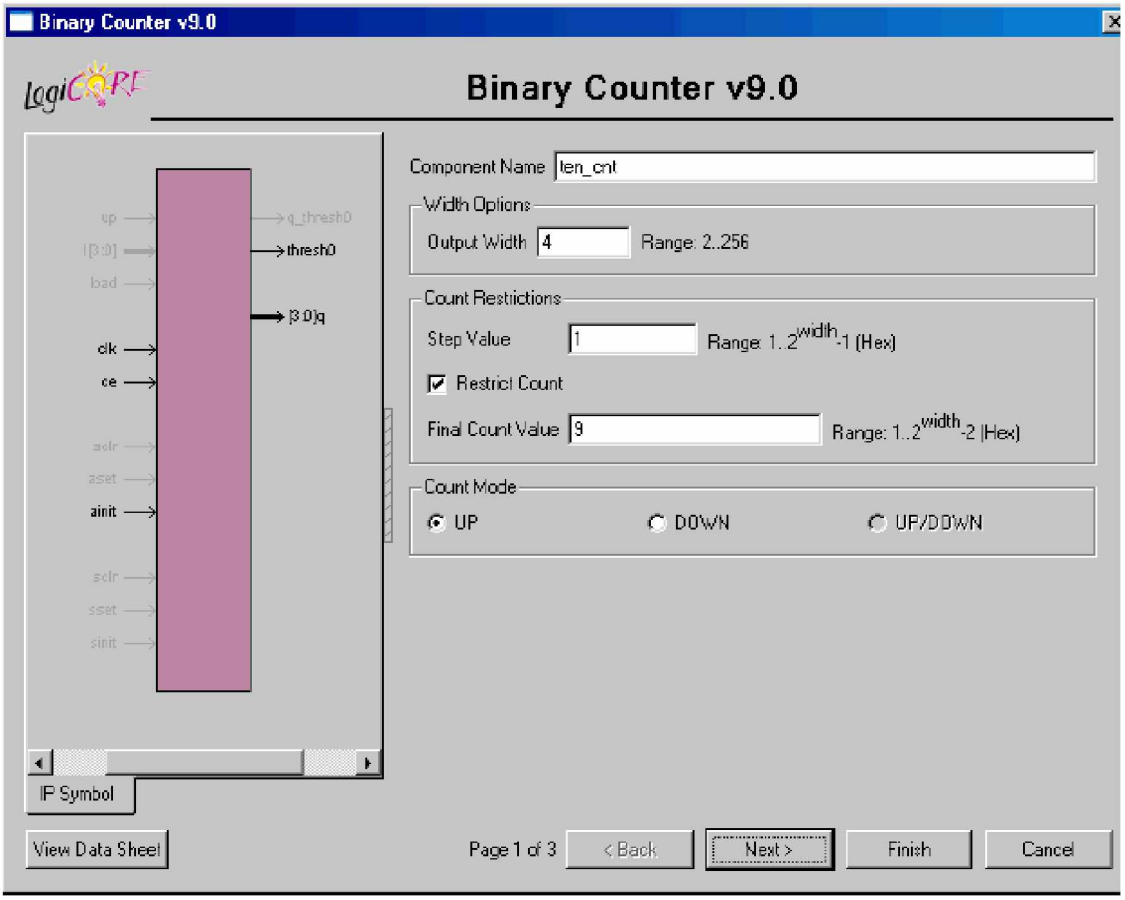


Figure 2.18 UN GENERATEUR DE NOYAU POUR COMPTEUR BINAIRE

Pour créer un module avec le générateur de noyau :

1. Dans le navigateur de projet, choisir **Project > New Source**.
2. Choisir **IP (Coregen & Architecture Wizard)**.
3. Taper le nom du module.
4. Cliquer **Next**.
5. Et choisir le module correspondant à votre choix.

6. Et Définir les différentes propriétés du module. (Pour un compteur par exemple compteur, décompteur ou compteur/décompteur).

Pour bien remplir les spécificités du module il est important de jeter un coup d'œil sur le data sheet pour connaitre les différentes fonctions.

A la fin du processus l'ISE génère un fichier de type nom_du_module.vho. Il reste à copier la différente partie de ce fichier et de les insérer dans le fichier de plus haut niveau du projet. Et une fois insérer on adapte les entrées du corps au Module de haut niveau.

2.3.3 Synthèse de la conception.

La synthèse permet de générer automatiquement à partir du code VHDL un schéma de câblage permettant la programmation du circuit cible. La description du code pour une synthèse est en théorie, indépendante de l'architecture du circuit. En pratique, le style utilisé aura une influence sur le résultat de la synthèse, influence liée au type de circuit et au synthétiseur utilisé. Le code devra être optimisé pour un type circuit. Si le besoin d'optimisation n'est pas très important, cette partie peut se dérouler très rapidement, alors qu'au contraire, si le besoin d'optimisation est grand, les subtilités pour coder de manière adéquate en VHDL obligeront le concepteur à y consacrer beaucoup de temps.

L'outil de synthèse utilise l'extension du HDL de la conception et génère un fichier de type netlist (EDIF ou NGC) pour les outils d'implémentation de Xilinx. L'outil de synthèse exécute trois étapes en générale pour créer le netlist :

- Analyser/ Contrôle de Syntaxe : Contrôle la syntaxe du code source.
- Compiler : Traduit et optimalise le type HDL dans un ensemble de composants que l'outil de synthèse peut reconnaitre.
- Carte : Traduit les composants de l'étape de compilation en composants primitifs de la technologie de cible.

2.3.3.1 Synthèse à l'aide du XST (Xilinx Synthesis Technologie)

Pendant la synthèse, les fichiers de HDL sont traduits en portes et optimalisés à l'architecture de cible.

Les processus disponibles pour la synthèse utilisant XST sont comme suit :

- Rapport sur l'état de la synthèse : Donne un sommaire sur le plan et la synchronisation de la synthèse aussi bien que les optimisations qui ont eu lieu.
- Schéma de la Vue RTL : Génère une vue schématique.
- Schéma de vue de la Technologie : Génère une vue schématique de la technologie.

- Ø Contrôler la Syntaxe : Vérifie que la source HDL est écrite correctement.
- Ø Générer un premier modèle de Simulation : Crée des modèles de simulation de HDL basés sur le netlist de synthèse.
- Ø Contraintes Entrantes

XST supporte une syntaxe de modèle du type fichier de contrainte utilisateur (UCF = User Constraints File) pour définir des contraintes de synthèse et de synchronisation. Ce format s'appelle le fichier de contrainte de Xilinx (XCF = Xilinx Constraints File), et le fichier a une extension de type xcf. XST emploie l'extension xcf pour déterminer si le fichier est un fichier de contraintes.

On crée les fichiers de contraintes comme pour les modules, mais il faut savoir qu'il y'a plusieurs types de contraintes et plusieurs façons de les générés.

2.3.4 Conception du projet à l'aide du schématique

Pour commencer un nouveau projet avec l'ISE pour le schématique on utilise la même procédure vue en (2.1 pour les HDL) à la seule différence il faut prendre pour module de base schématique au lieu de HDL (voir figure1.1) dans la fenêtre de création de projet.

Description de la Conception

La conception basée sur le schématique comme pour les HDL utilise un fichier supérieur de conception qui est une feuille schématique qui peut se rapporter à plusieurs autres macros instructions plus élémentaires. Les macros instructions plus élémentaires sont une variété de différents types de modules, incluant des modules schématiques, un module du générateur de NOYAU, et des modules HDL.

Ici nous allons nous intéressé seulement aux modules schématique, puisque le générateur de noyau et les HDL ont été déjà vue dans la conception à l'aide de HDL.

Entrée de la Conception

Les entrées de la conception correspondent aux différents modules que l'on doit insérer dans une conception, se sont des modules schématiques, HDL, des noyaux, et/ou des machines d'états. Ici nous allons présenter le module schématique.

Créer une macro instruction basée sur le schématique.

Une macro instruction à l'aide de schématique se compose d'un symbole et d'un sous fichier schématique. On peut créer le schéma fondamental ou le symbole d'abord en premier. Le symbole correspondant ou le fichier schématique peut alors être générer automatiquement.

Dans les étapes suivantes, on crée une macro instruction basé sur le schématique en utilisant la boîte de création de source du navigateur de projet. Un fichier schématique vide est alors créé, et on peut définir la logique appropriée. La macro instruction créé alors est automatiquement ajoutée à la bibliothèque du projet.

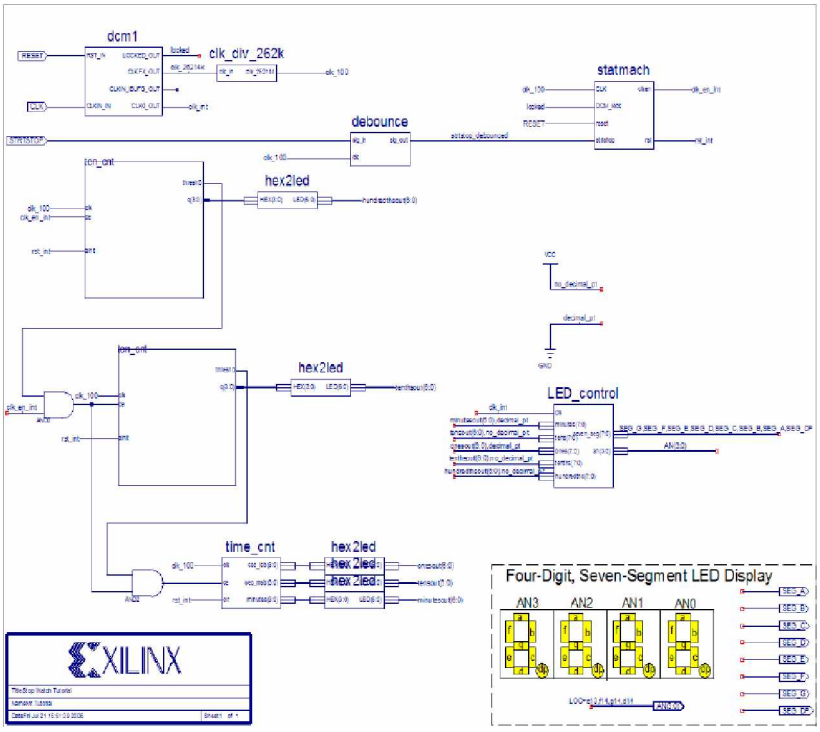


Figure 2.19 Une conception shématique

Pour créer une macro instruction basé sur le schématique:

1. Dans le navigateur de projet, choisir **Project > new source**. La boite de dialogue de source s'ouvre : La boite de dialogue de source affiche une liste de tous les types disponibles de source.

2. Choisir **schematic** comme type de source.

3. Ecrire le nom du fichier.

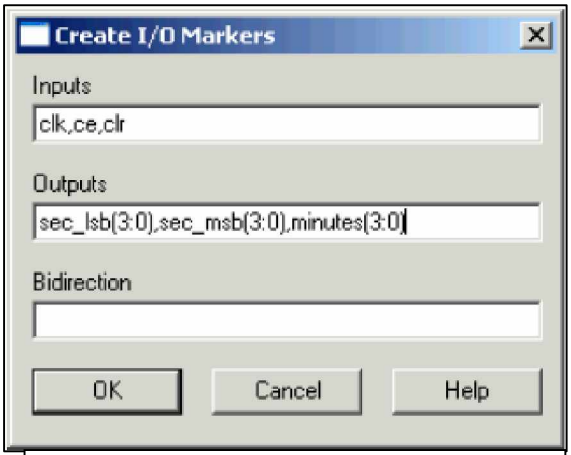
4. Cliquer sur **Next** puis **Finish**.

Un nouveau fichier schématique est créé, ajouté au projet, et ouvert pour être éditer.

Définir le fichier schématique

Maintenant le fichier est créé, mais il est vide. La prochaine étape est d'ajouter les composants que doit contenir la source.

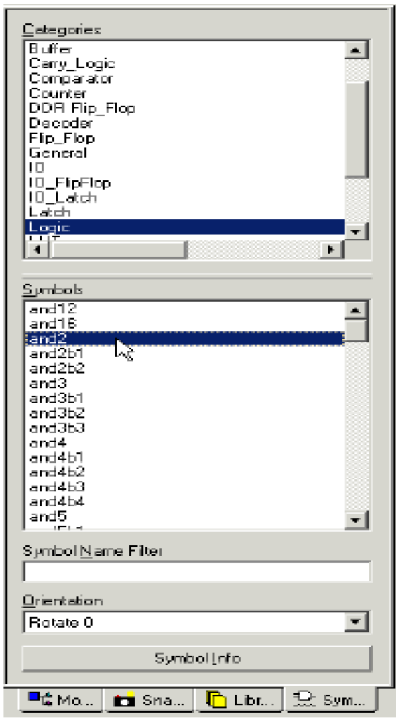
Définir les ports d'entrées/sorties.



Marqueurs d'entrées/sorties

Les marqueurs d'E/S sont employés pour déterminer les ports sur une macro instruction. Le nom de chaque broche sur le symbole doit avoir un connecteur correspondant dans le sous fichier schématique. Ajouter les marqueurs d'E/S au schéma pour déterminer les macros ports.

Pour ajouter les marqueurs d'E/S :



1. Sélectionner **Tools>Create I/O Markers**.

La boite de dialogue pour créer des marqueurs E/S s'ouvre.

2. Dans la zone **Inputs** indiquer les noms des ports d'entrées en les séparant par des virgules.

3. Dans la zone **Outputs** indiquer les noms des ports de sorties en les séparant par des virgules.

4.Cliquer **Ok** et les ports apparaitrons sur le schéma.

Note : La fonction de création de marqueurs pour E/S est disponible seulement pour une feuille schématique vide. Cependant, Des marqueurs d'E/S peuvent être ajoutés aux fils à tout moment en choisissant **add >I/O Marker.**

Ajouter d'autres composants au fichier schématique.

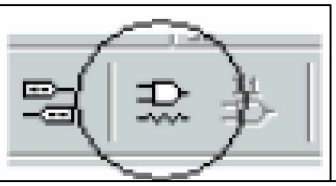
Les composants de la bibliothèque du dispositif pour un projet donné sont disponibles dans la liste des symboles, et le symbole du composant peut être mis sur le schéma. Les composants disponibles énumérés dans la liste de symbole sont classés par ordre alphabétique en se référant à chaque bibliothèque.

1. Dans la barre de menu, choisir **add > symbol** ou cliquer sur l'icône dans la barre d'outils.

Ceci ouvre la liste des symboles à la gauche de l'éditeur schématique, affiche les bibliothèques et leurs composants correspondants.

2. choisir le composant, en utilisant une des deux voies :

- Sélectionner la catégorie (flips flops, éléments logiques, multiplexeurs,..) du composant dans la boite de dialogue de symbole et choisir le composant parmi les symboles énumérés.



Icône d'ajout de symbole.

Ou:

- Choisir tous les symboles et taper le nom dans le filtre de nom de symbole au bas de la fenêtre de symbole.

Note : On peut faire pivoter les composants en choisissant le composant, et puis en appuyant sur Ctrl+R.

Corriger Des Erreurs

Si on fait une erreur en mettant un composant, on peut facilement le déplacer ou le supprimer.

Pour déplacer le composant, cliqué sur le composant et glisser la souris.

Supprimer un composant :

• Clic droit sur le composant et choisir **Delete**.

Mettre en place les fils

Utiliser l'icone d'ajout de fil dans la barre d'outils pour mettre des fils (également appelés les fils) pour connecter les composants du schéma.

Pour mettre un fils entre les composants.

1. choisir **add > wire** ou cliqué sur l'icone **d'ajout de fils** dans la barre d'outils.

2.Clic sur la broche de sortie et cliqué sur la broche d'entrée (de destination) sur le composant, l'éditeur schématique dessine un fils entre les deux broches.

Figure : 2.10 Les symboles

- Clic sur le fils pour créer une courbure de 90 degrés.

Pour dessiner un fils entre un fils déjà existant et une broche, clic une fois sur la broche et une fois sur le fils existant. Un point de jonction est alors dessiné sur le fils existant.

Ajouter Des Bus

Dans l'éditeur schématique, un bus est simplement un fils multi-bit. Pour ajouter un bus, on utilise même la méthode que pour ajouter des fils et puis ajoutent un nom de multi-bit.

Une fois que le bus a été créé, on a l'option "tapping" pour utiliser chaque signal du bus individuellement.

Ajouter des prises de Bus

Employer des fils pour se connecter à un seul bit d'un bus.

Pour lier un fils à un bit simple du bus :

1. choisi **add > bus tap** ou cliqué l'icône **d'ajout de prise de bus** dans la barre d'outils.

Le curseur change, indiquant que tu es maintenant dans la mode de prise de bus.

2. Avec l’option tab à la gauche du schéma, choisir l'orientation correcte pour la prise de bus.

3. Placé la prise sur un des trois bus de sorte que le côté de fil de la prise de bus se dirige vers une broche non liée.

4. répètent l’étape à tous les bits nécessaires du bus.

2.3.5 La simulation

Pour être efficace, une simulation doit être la plus complète possible, sans être redondante. Une bonne simulation épurée de tous les tests doublés permettra une bonne économie tant pour le temps consacré à cette simulation que pour les tests sur circuit ensuite.

Vue d'ensemble de la simulation comportemental

L’ISE de Xilinx permet pour une simulation d’intégré le simulateur ModelSim de mentor et le simulateur de Xilinx ISE qui permet à des simulations d'être exécuter sur le navigateur de projet de Xilinx.

2.3.5.1 Simulation à l’aide du ModelSim

Simulation en utilisant Model Sim.

ISE intègre pleinement le simulateur de ModelSim. ISE permet à ModelSim de créer l'espace de travail, compile les fichiers source, charger la conception, et effectuer la simulation basée sur des propriétés de simulation.

Pour choisir ModelSim en tant que votre simulateur de projet :

1. Dans l'étiquette de sources, clic droit sur la ligne du dispositif.

2. Choisir **Proprieties**.

3. Dans le domaine de simulateur de la zone de dialogue de propriétés de projet, choisir le type de ModelSim avec le langage HDL utilisé.

Localiser les procédés de simulation

Les procédés de simulation dans ISE permettent d’exécuter la simulation sur la conception en utilisant ModelSim.

Pour localiser les processus du simulateur de ModelSim :

1. Dans la table des sources, choisir **simulation comportementale** pour le champ de la vue.

2. choisir le fichier test bench.

3. Dans la fenêtre de processus, cliquer sur **+** près **du simulateur de ModelSim** pour augmenter la hiérarchie de processus.

Si les processus de simulateur de ModelSim n'apparaissent pas, soit ModelSim n'est pas choisi en tant que simulateur dans la zone de dialogue de propriétés de projet, ou le navigateur de projet n'arrive pas à trouver modelsim.exe.

Si ModelSim est installé mais les processus ne sont pas disponibles, les préférences du navigateur de projet peuvent ne pas être correctement placées.

Pour placer ModelSim :

1. Choisir **Edit >Preferences**.

2. Cliquer sur + à côté du général d'ISE pour augmenter les préférences d'ISE

3. Cliquer sur intégré des outils.

4. Dans la partie droite, sous **Model Tech Simulator**, parcourir pour mettre le chemin d'accès à modelsim.exe. Par exemple, c:\modeltech_6.2a\win32\modelsim.exe

Les procédés suivants de simulation sont disponibles :

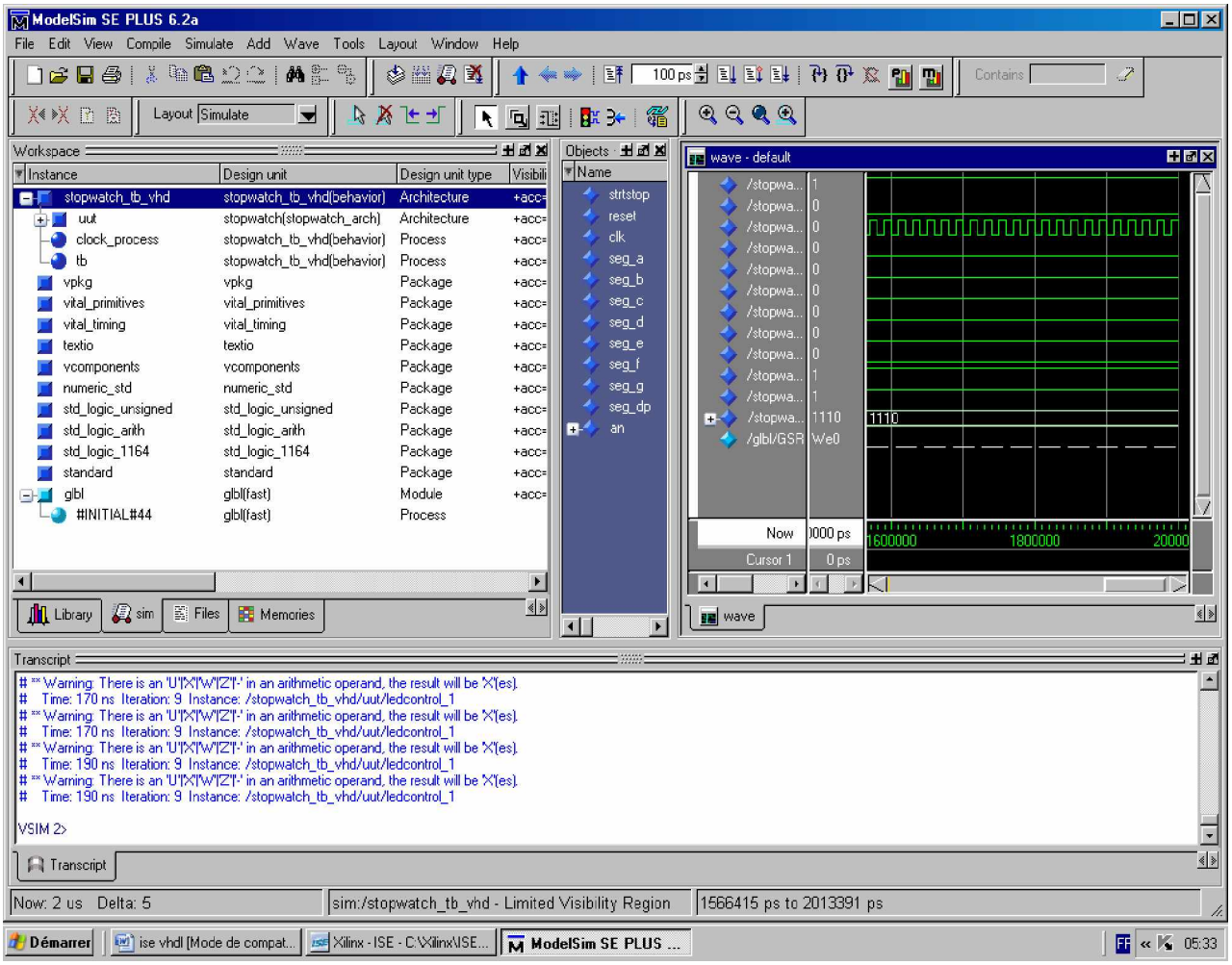


Figure 2.20 L'environnement du ModelSim 6.2a

- **Simuler Le modèle comportemental**

Ce processus commence la simulation de la conception.

- **Générer les Résultats Prévus de la Simulation**

Ce processus est disponible seulement si tu as un fichier d'essai de forme d'onde dans l'ISE. Si on fait un double clic ce processus, ModelSim l'exécute pour générer les résultats dans l'ISE.

Note : Pour certaine conception il est nécessaire de compilé les bibliothèques de l'ISE dans le ModelSim, pour cela il faut cliquer la vue le dispositif dans la table de source et au niveau de la table de processus double clic sur **Compile HDL Simulation Libraries**. Cette action prend un peu de temps donc il va falloir patienter un quart d'heure ou plus.

La simulation avec le simulateur de l'ISE utilise la même procédure et donne le même résultat.

2.3.6 Implémentation

L'exécution de l'implémentation est le processus de la traduction, dresser la carte, pour le placement, routage, et la génération du fichier binaire pour la conception. Les outils d'implémentation de la conception sont dans le logiciel d'ISE pour faciliter l'accès et la gestion du projet.

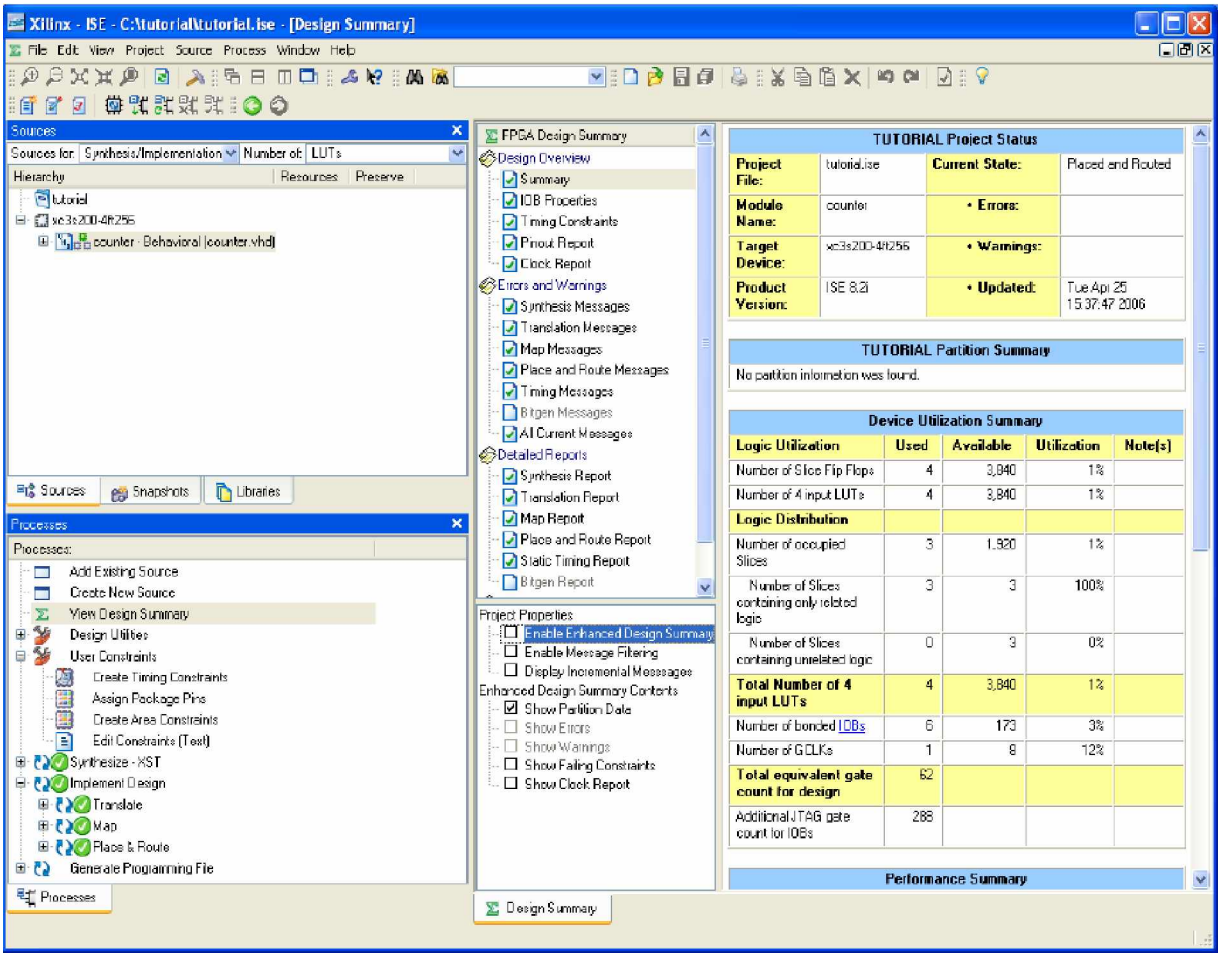


Figure 2.21 l'environnement de travail del'ISE avec les signes au niveau des processus d'Implémentation

Le placement est la phase de mise en place des différents blocs logiques obtenus par synthèse sur le dispositif.

Le routage est la phase de connexion de ses blocs sur le circuit.

1. Choisir le fichier source dans la fenêtre de sources.

2. Ouvrir le sommaire de la conception en cliquant 2 fois sur **View Design Summary** dans la fenêtre de processus.

3. Double clic **Implement Design** dans la fenêtre des processus.

Assigner les contraintes de location de pin

A ce stade il faut bien connaitre le dispositif pour bien mettre les pins.

Indiquer les endroits pour les ports de la conception de sorte qu'elles soient reliées correctement à la carte.

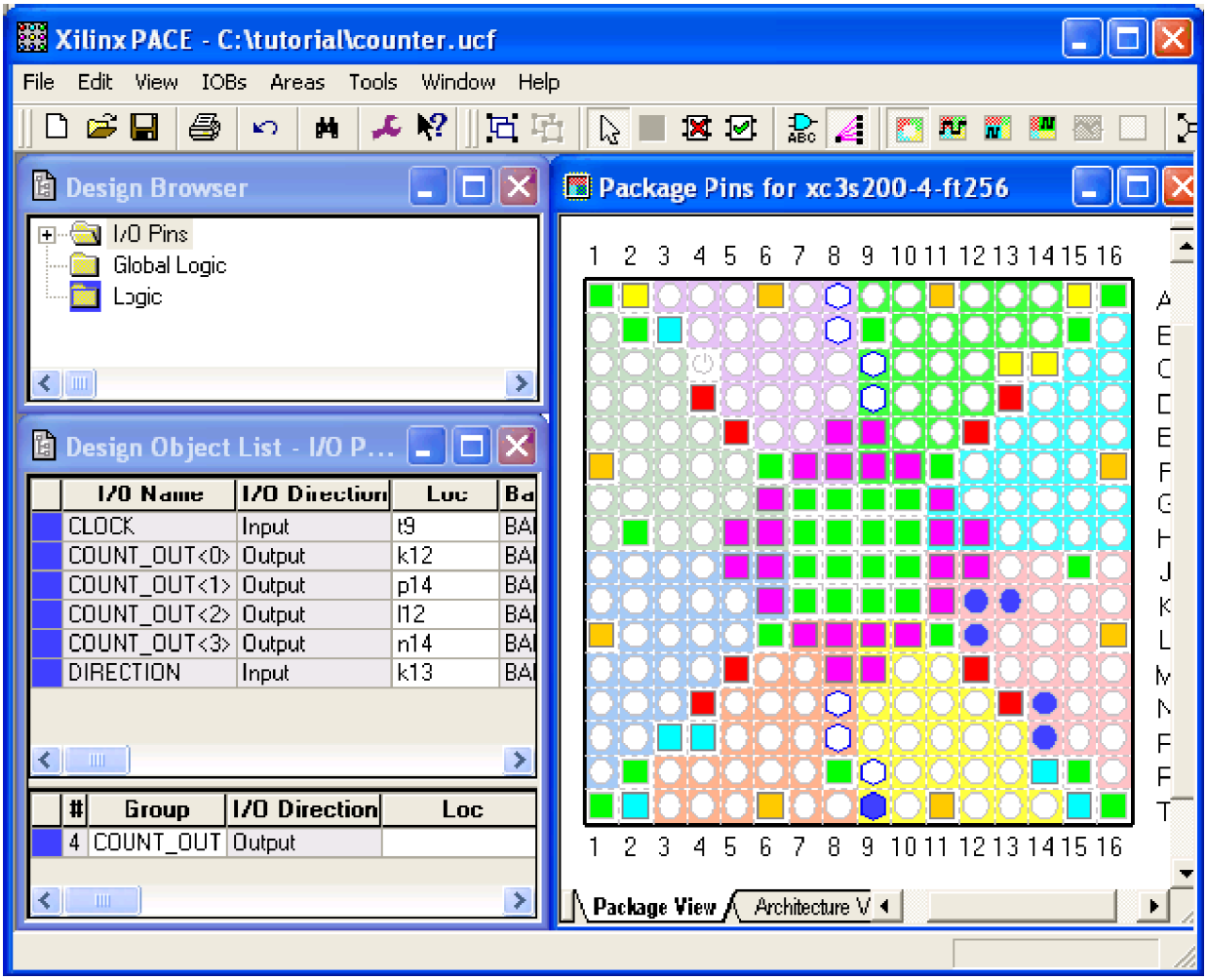


Figure 2.22 Vue de la carte

Pour contraindre les ports de conception pour empaqueter des pins, on fait :

1. Vérifier que la source de niveau haut est choisie dans la fenêtre de sources.

2. Double clic sur **Assign Package Pins** le processus se trouve dans le groupe de processus de contraintes utilisateur. Le Xilinx Pinout and Area Constraints Editor (PACE) s'ouvre.

3. Choisir la table de vue de paquet **Package View.**

4. Dans la fenêtre de liste d'objet de conception, entrer le pin correspondant à chaque port au niveau de **Loc**.

5. On enregistre et on ferme la fenêtre.

6. Ré-implémenter la conception.

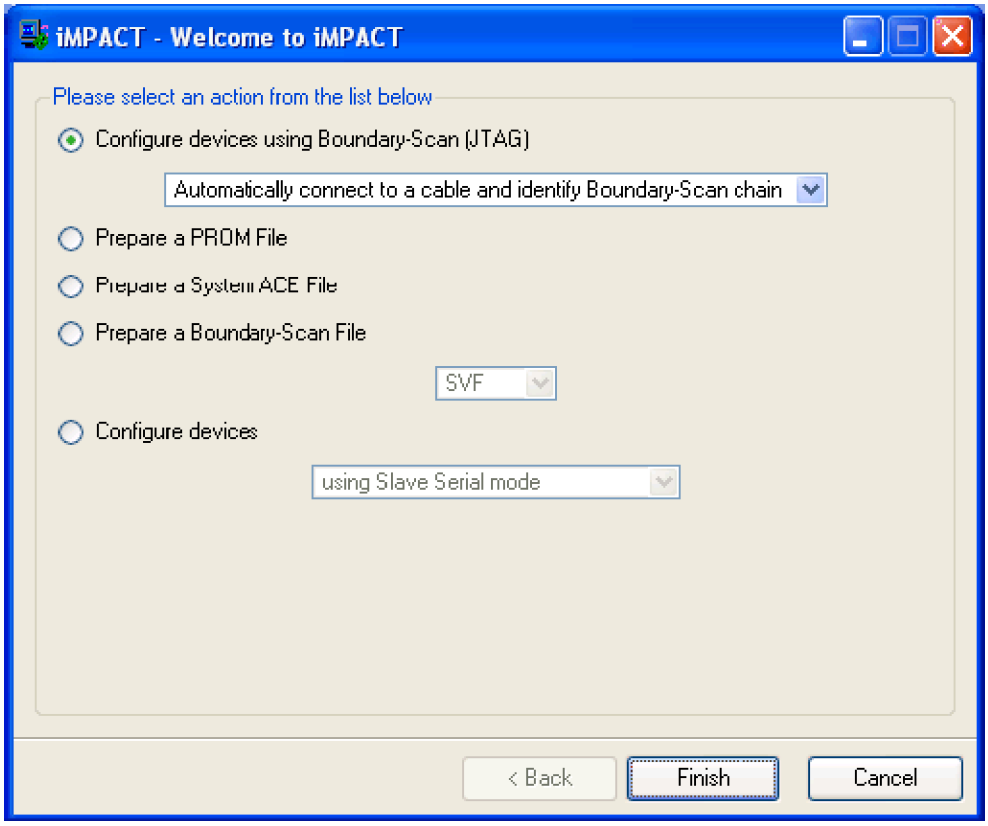


Figure 2.23 La boite de dialogue d'IMPACT

Téléchargement de la conception sur la carte.

C'est la dernière étape dans la procédure de vérification de la conception

1. Connecter le câble d'alimentation de 5V DC à la carte (j4).

2. Connecter le câble de téléchargement entre le PC et la carte (j7).

3. Sélectionner Synthesis/implementation dans la fenêtre de source.

4. Sélectionner le Module de haut niveau dans la fenêtre de source.

5. Dans la fenêtre de processus cliquer sur + à coté de **Generate Programming File.**

6. Double clic sur **Configure Device (iMPACT).**

7. La boîte de dialogue de Xilinx WebTalk s'ouvrira, cliquer sur **Decline**.

8. Sélectionner **Disable the collection of device usage statistics for this project only** et cliquer **OK**.

IMPACT s'ouvre et on a la configuration du dispositif dans la boîte de dialogue.

9. Dan la boite de dialogue de bienvenue, sélectionner **Configure devices using Boundary-Scan (JTAG)**.

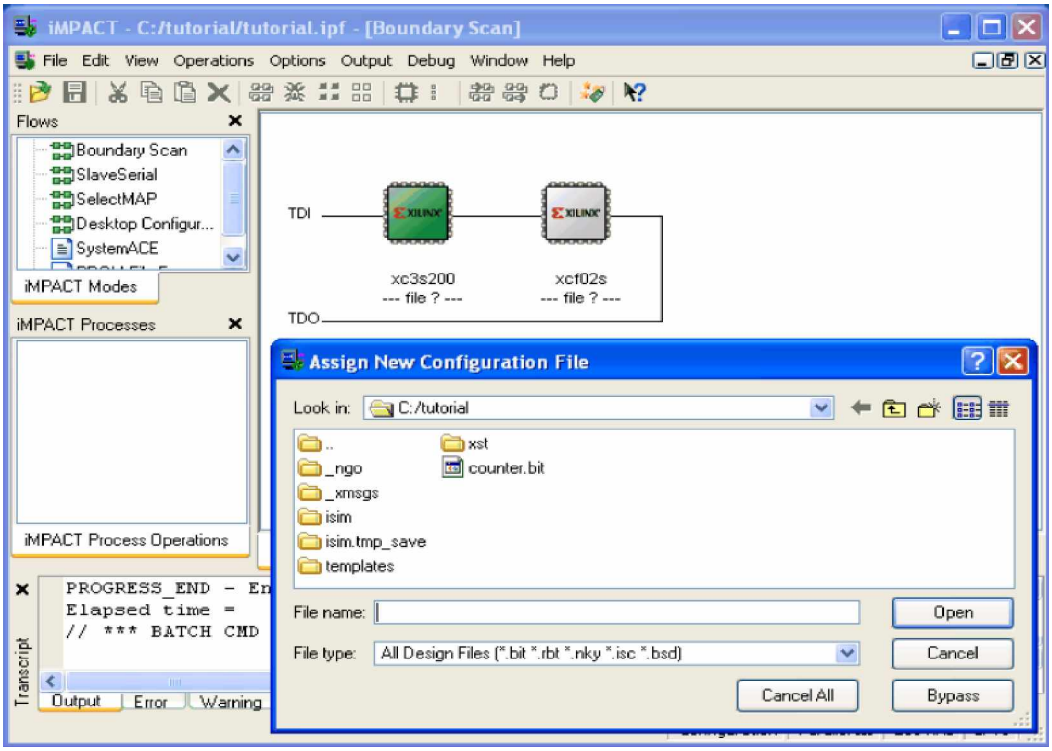


Figure 2.24 Téléchargement du fichier binaire (nom_du_module.bit)

10. Vérifier que **Automatically connect to a cable and identify Boundary-Scan chain** est selectionner.

11. Cliquer sur **Finish**.

12. S'il y'a un message qui dit que 2 dispositifs ont été trouvés cliquer **OK**.

13. La boite de dialogue **Assign New Configuration File** apparait. Pour donner le fichier de la configuration du dispositif dans la chaine du JTAG. Choisir le nom_du_module.bit et cliquer sur **OPEN**.

14. S'il y'a un message d'avertissement cliquer **OK.**

15. Sélectionner **Bypass** pour quitter les autres dispositifs.

16. Clic droit sur l'image du dispositif, et sélectionner **Program…**la boite de dialogue du **Programming Properties** s'ouvre.

17. Cliquer **Ok** dans le programme du dispositif.

Quand le programme sera complètement terminer, et que ça à marcher un message s'affiche.

18. Fermer IMPACT sans sauvegarder.

Conclusion

Nous avons donné un aperçu sur l'utilisation de l'environnement de travail de l'ISE, sans pour autant trop nous approfondir car c'est logiciel très complexe et qui contient énormément d'outils d'aide à la conception. La raison de sa complexité est de vouloir répondre à toutes les étapes nécessaires à la conception, et le plus important c'est qu'il se suffit à lui-même, donc avec le logiciel on peut aborder n'importe quelle conception sur les FPGAs. Certaines parties n'ont pas été abordées comme la création de fichier de contraintes du fait que l'aborder nous emmènerais sur un chemin très long et compliqué à saisir sans certaines connaissance préalable sur le dispositif.

3 CHAPITRE III : Implémentation du modulateur OFDM

Nous allons présenter dans cette partie l'architecture de notre conception et les résultats de la simulation et de l'implémentation. Nous débuterons par une présentation des différents logiciels utilisés pour notre application et ensuite nous passerons au modèle adopter. Nous réaliserons, à partir de ce modèle, les différents programmes permettant d'obtenir un résultat proche de la situation décrite dans l'architecture.

Nous utilisons une méthode qui consiste à diviser notre travail comme suit :

- Analyse,
- Design,
- test et validation.

Notre analyse permet d'établir un schéma à suivre pour notre conception à partir des objectifs que nous nous sommes fixés et des contraintes de notre travail afin de définir les limites du Design. Le Design consiste pour nous de mettre au point un modèle qui correspond à l'architecture d'un modulateur OFDM que nous utiliserons comme base pour la programmation des blocs nécessaires au fonctionnement du modulateur. Et enfin, le teste et la validation du modèle qui conduira à la synthèse, simulation et implémentation du modèle.

3.1 Présentation des logiciels

Pour notre application, Nous nous servirons de trois logiciels dont les deux premiers ont été déjà présentés (2.3):

- L'ISE de Xilinx qui est un outil incontournable pour l'implémentation sur les FPGAs de Xilinx.
- Le ModelSim de Mentor Graphics qui est l'outil par excellence pour la simulation
- Le Matlab de Mathworks qui demeure l'un des outils le plus utilisé dans l'ingénierie de nos jours.

Comme nous l'avons déjà décrit dans le (2.3) l'ISE est utilisé pour l'entrée des flots de la conception (éditeur HDL, vérification de syntaxe) et la synthèse. Une fois la synthèse terminée, nous ferons appel à ModelSim pour la simulation, nous utiliserons des tests benchs de Matlab pour la co-simulation à l'aide du Link for ModelSim et de l'ISE (voir figure3.1). Après nous passerons à l'implémentation, le placement et le routage à l'aide de l'ISE.

3.2 Schéma de conception

Nous allons tout d'abord adopté un schéma de conception. Notre schéma de conception consiste à trouver un modèle pour le modulateur qui va tenir compte de nos contraintes, puis de passer à la réalisation de chaque bloc que l'on testera et en fonction des contraintes qui s'imposerons nous allons revoir notre modèle pour l'adapter. Après avoir simulé chacune des blocs, nous passerons à l'assemblage et jusqu'à ce niveau des modifications si nécessaire seront apportés à notre modèle, pour dire qu'il ne restera jamais statique jusqu'à la réalisation finale (figure3.2).

Un point important a soulevé serai qu'il existe des standards de communication basé sur l'OFDM et nous avons jugé utile de se rapproché d'un des standards pour notre réalisation. Nous essaierons au maximum de s'en approcher tout en gardant à l'esprit notre modulateur et ses contraintes.

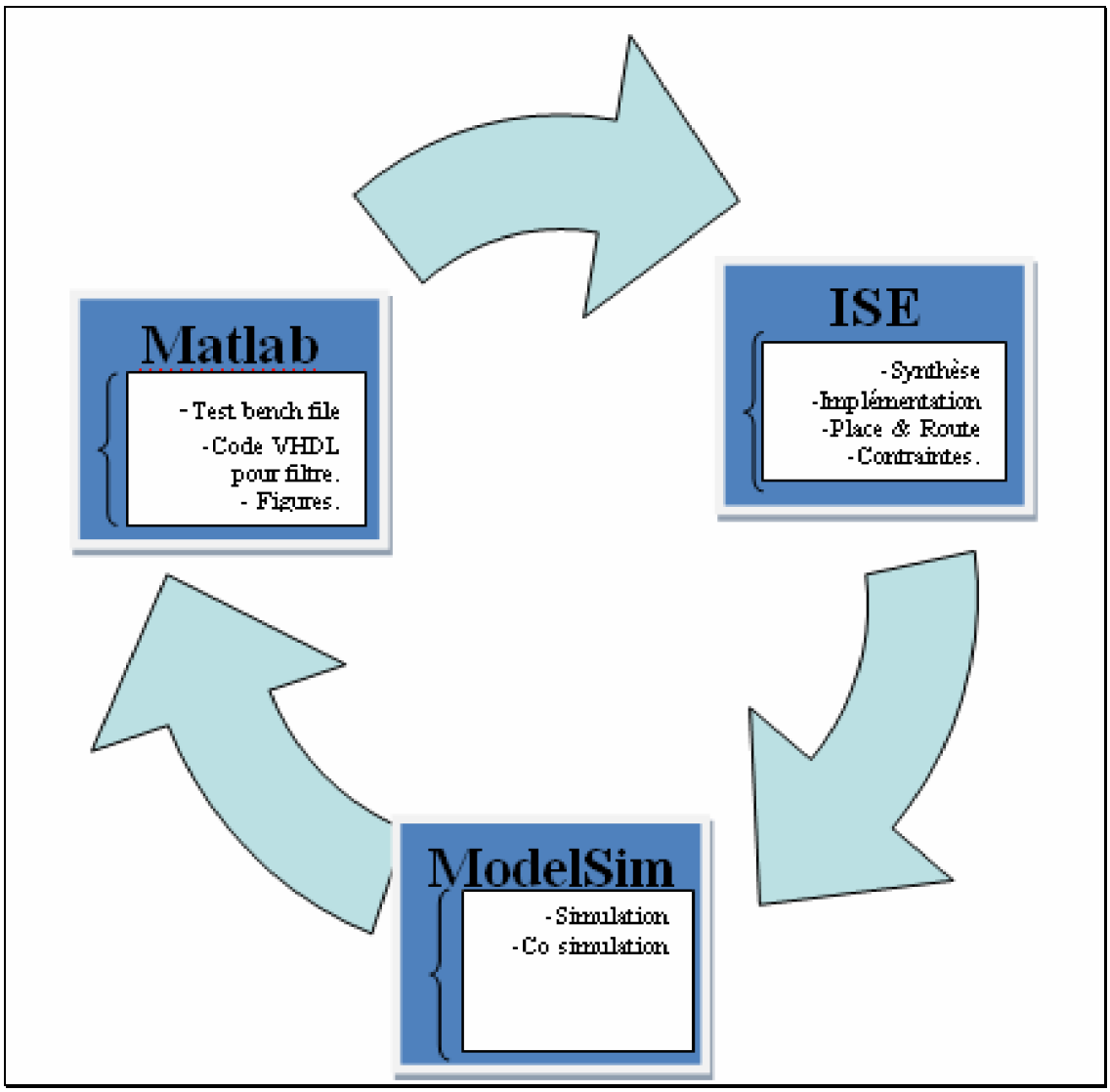


Figure 3.1 Les relations entre logiciel

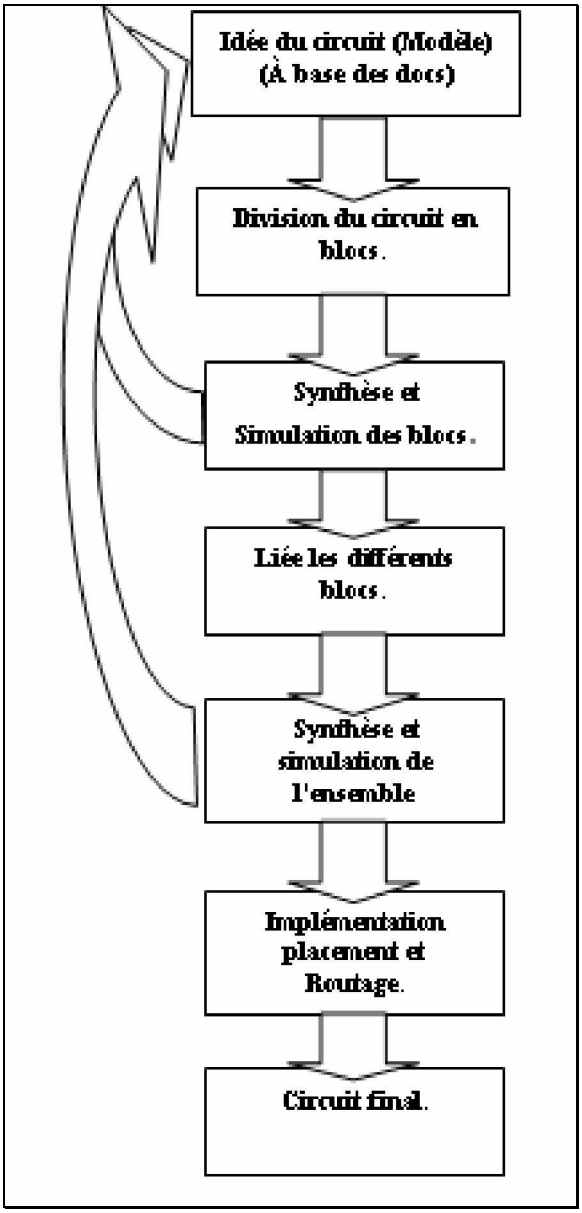


Figure 3.2 Le schéma de conception

Le standard auquel nous nous sommes rapprochés est l'IEEE 802.11a.

3.3 Architecture du modulateur OFDM

Le travail consiste à la réalisation un modulateur OFDM. L'architecture sera mise au point à partir des différents blocs de l'OFDM, nous adopterons un modèle pour chaque bloc qui demande moins de ressources hardware et nous essaierons de se rapprocher du standard IEEE 802.11a.

Apres avoir adopté un modèle nous passerons à la conception VHDL, et nous suivrons le schéma décrit par la figure3.2. Le modulateur utilisé est donné par la figure3.3.

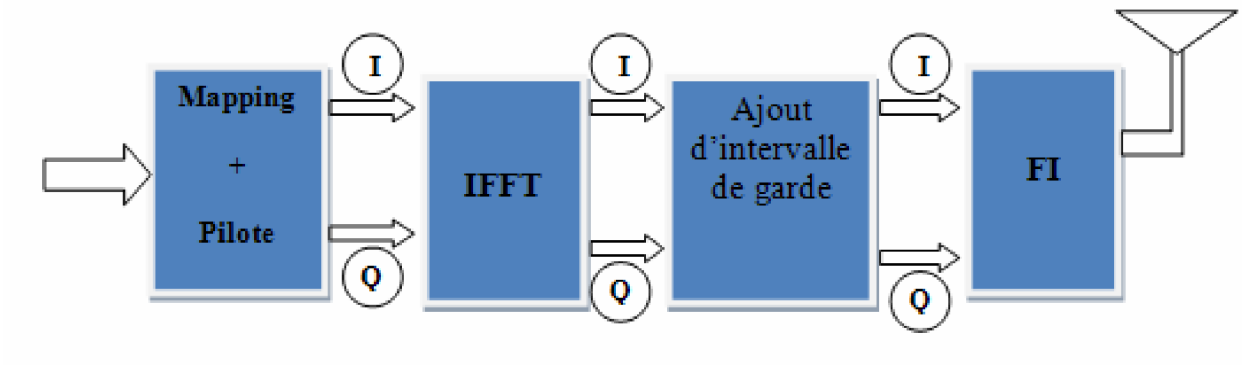


Figure 3.3 modulateur OFDM

3.3.1 Mapping

Pour le mapping, nous utiliserons quatre constellations : QPSK, BPSK, 16-QAM et 64-QAM. Nous réaliserons un modèle qui supporte les 4 constellations et qui pourra être facilement intégré du point de vue liaison avec les autres blocs.

Nous nous référons aux standard IEEE 802.11a [**14**] et [**3**] qui divisent pour chaque constellation le nombre de bits en entrée en 2 parties, la première pour la phase et la seconde pour la quadrature de phase. Ceci est représenté dans la figure 3.4.

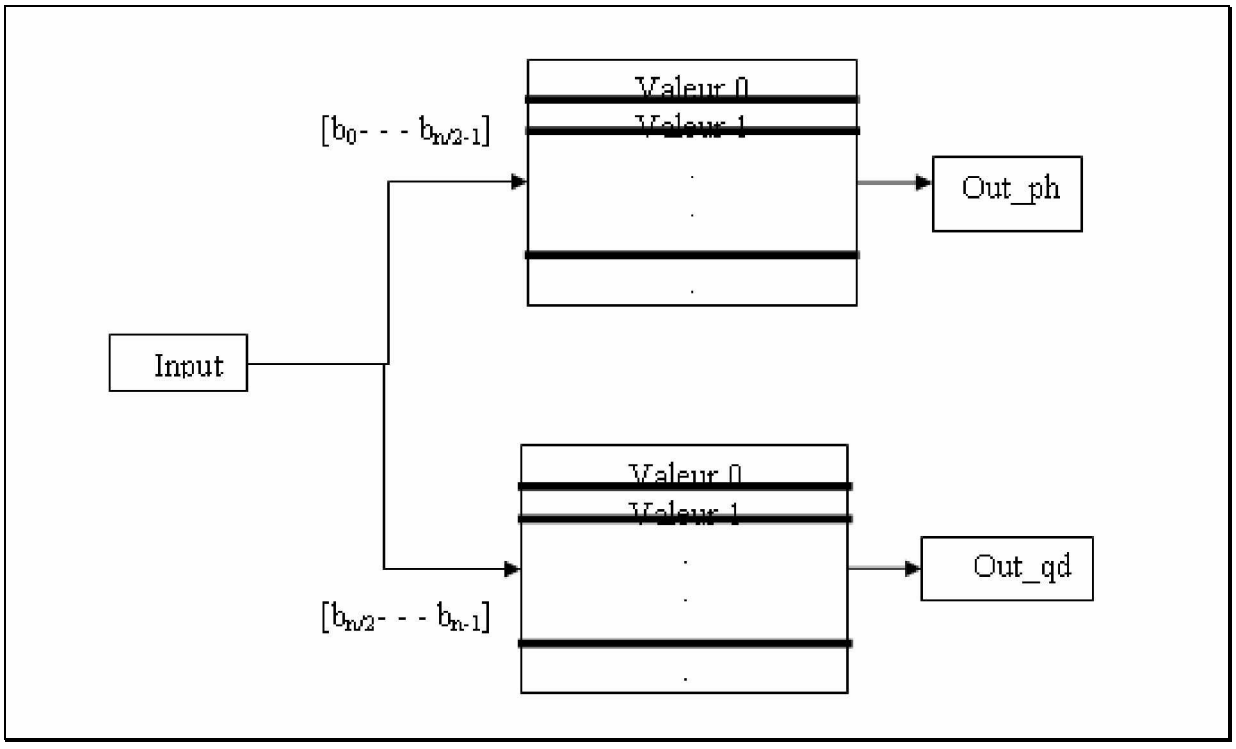


Figure 3.4 Architecture d'une constellation.

Comme on peut le constater, notre schéma à la structure d'un ROM, donc au niveau de la conception. Il nous suffit de créer deux ROMs dont les adresses correspondent à $b_0 \ldots b_{n/2-1}$ et $b_{n/2} \ldots b_{n-1}$

et les données stockés sont les constellations de la phase (**Out_ph**) et de la quadrature de phase (**Out_qd**).

Nous développons maintenant un modèle pour l'ensemble des constellations. Ceci revient à les mettre ensemble et à chaque fois qu'il y'a des données en entrées qu'on puisse choisir une des constellations. Pour cela, nous avons utilisé un démultiplexeur quatre entrées pour une sortie, nous utilisons un seul démultiplexeur au lieu de deux multiplexeurs qui est couramment utilisé, ceci dans le but de diminuer le nombre de ressources à utiliser.

En utilisant un ensemble de constellation, le standard IEEE 802.11a [**14**] prévoit la normalisation des sorties **Out_ph** et **Out_qd**, pour qu'il y ait en sortie un seul niveau de puissance pour toutes les constellations. Cette normalisation impose la multiplication des valeurs en sortie pour :

- le PSK par un facteur de 1,
- le QPSK par un facteur de $1/\sqrt{2}$,
- le 16-QAM par un facteur de $1/\sqrt{10}$,
- le 16-QAM par un facteur de $1/\sqrt{42}$.

Le sélecteur permet de choisir une sortie parmi les 4 constellations, il est formé de 2 bits.

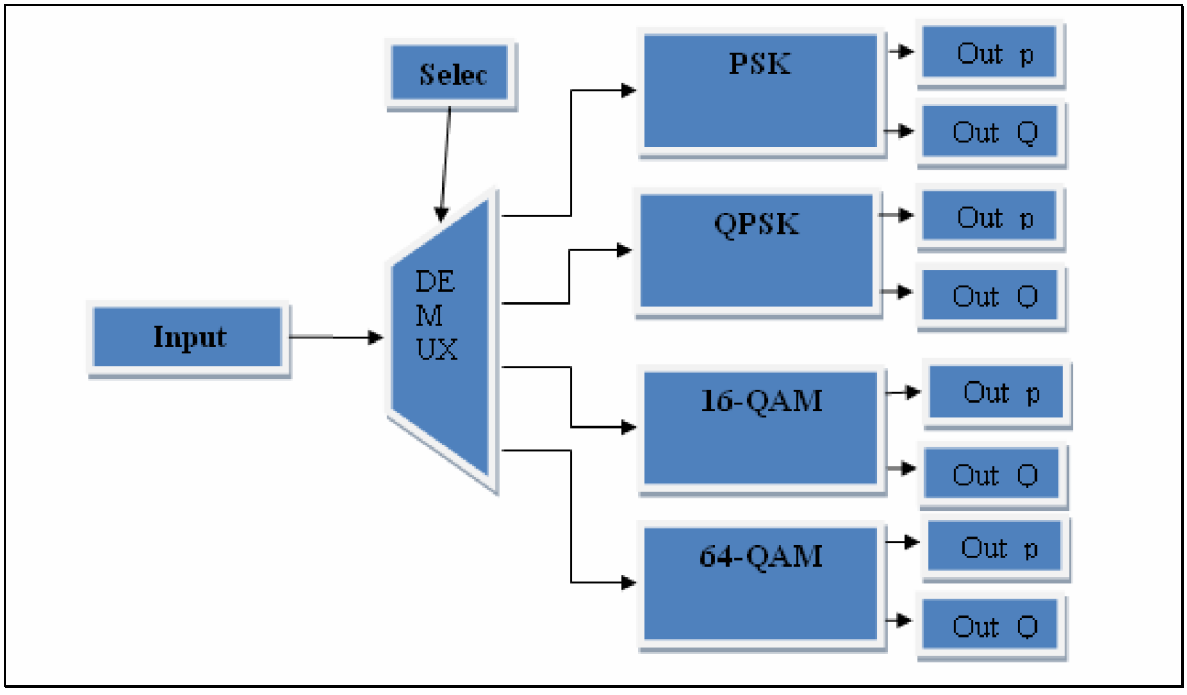


Figure 3.5 Modèle de la mise ensemble des constellations

Input bit b_0 (select=00)	Out_ph	Out_qd
0	-1	0
1	1	0

Tableau : 3.1 pour le PSK

Input bit b_0 (select=01)	Out_ph
0	-1
1	1

Input bit b_1 (select=01)	Out_qd
0	-1
1	1

Tableau : 3.2 Codage du QPSK

Inputs bits (b_0b_1) (select =10)	Out_ph
00	-3
01	-1
10	1
11	3

Inputs bits (b_2b_3) (select =10)	Out_qd
00	-3
01	-1
10	1
11	3

Tableau : 3.3 Codage du 16-QAM

Inputs bits ($b_0b_1b_2$) (select=11)	Out_ph
00	-7
001	-5
010	-3
011	-1
100	1
101	3
110	5
111	7

Inputs bits ($b_3b_4b_5$) (select=11)	Out_qd
000	-7
001	-5
010	-3
011	-1
100	1
101	3
110	5
111	7

Tableau : 3.4 Codage du 64-QAM

Dans ce cas nous stockons en mémoire les valeurs normalisées au lieu de faire la normalisation à la sortie des modulateurs.

En entrée, on a le signal **DATA** qui est le bus de données à transmattre (la taille est définie par la constellation utilisée). **clk** est l'horloge, **selec** permet de sélectionner la constellation. En sortie, on a : **Out_ph** pour la phase et **Out_qd** pour la quadrature de phase.

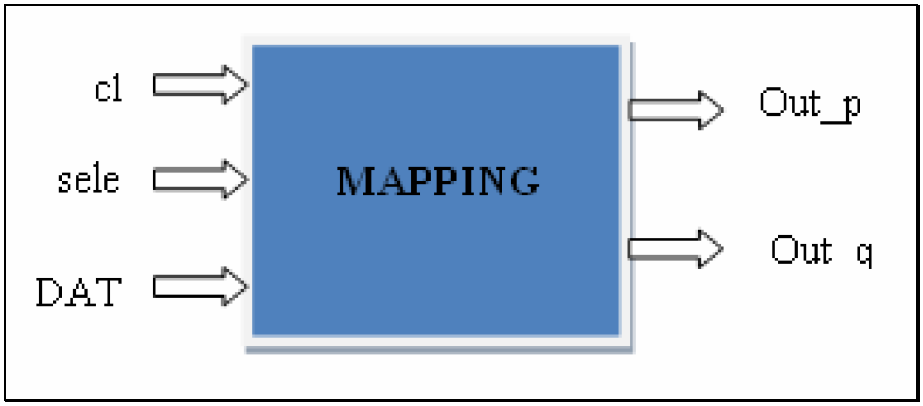


Figure 3.6 Bloc de Mapping

3.3.2 Pilote

En pratique, l'OFDM utilise d'autres sous porteuses, en plus des données, appelées sous porteuses pilotes, pour recueillir l'information sur la qualité du canal et permettre une meilleure prise de décision au niveau du démodulateur.

Les sous porteuses pilotes transportent les symboles pilotes. Ces symboles sont déjà connus par le récepteur qui les utilise pour l'estimation du canal. Sachant que la connaissance du déphasage est important pour toute modulation et de l'amortissement pour toute modulation QAM. Quand un décalage d'amplitude et/ou de phase est mesuré par le récepteur sur les symboles des pilotes, une compensation égale à ce décalage est réalisée au niveau de tous les autres symboles. Les pilotes sont utilisés également pour l'égalisation fréquentielle en cas de sélectivité fréquentielle.

Concernant la norme IEEE 802.11a dans chaque symbole OFDM, 4 sous porteuses sont dédiées aux pilotes pour une meilleure compensation fréquentielle et pour lutter contre l'erreur de déphasage. Ces pilotes sont portés par les sous porteuses -21, -7, 7 et 21(Sachant qu'un symbole OFDM a des sous porteuses numérotés de -26 à +26) modulé en BPSK par une séquence binaire. La contribution du pilote pour la $n^{ème}$ symbole OFDM est donnée par la séquence p.

$P_{-26,+26}$ = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, –1, 0, 0, 0, 0, 0}.

La polarité des pilotes est contrôlée par une séquence p_n cyclique sur 127 éléments donnée par :

$P_{0,126}$ = {1,1,1,1, -1,-1,-1,1, -1,-1,-1,-1, 1,1,-1,1, -1,-1,1,1, -1,1,1,-1, 1,1,1,1, 1,1,-1,1, 1,1,-1,1, 1,-1,-1,1, 1,1,-1,1, -1,-1,-1,1, -1,1,-1,-1, 1,-1,-1,1, 1,1,1,1, -1,-1,1,1, -1,-1,1,-1, 1,-1,1,1, -1,-1,-1,1, 1,-1,-1,-1, -1,1,-1,-1, 1,-1,1,1, 1,1,-1,1, -1,1,-1,1, -1,-1,-1,-1, - 1,1,-1,1, 1,-1,1,-1, 1,1,1,-1, -1,1,-1,-1, -1,1,1,1, -1,-1,-1,-1, -1,-1,-1}[**14**]

Cette séquence peut être réalisée de deux manières, soit en utilisant une mémoire boucle par exemple un FIFO dont la sortie est bouclé sur l'entrée, ou à l'aide des registres. Pour les registres, on utilise des 0 et des 1 et en faisant à la sortie un passage du format unipolaire vers le format bipolaire.

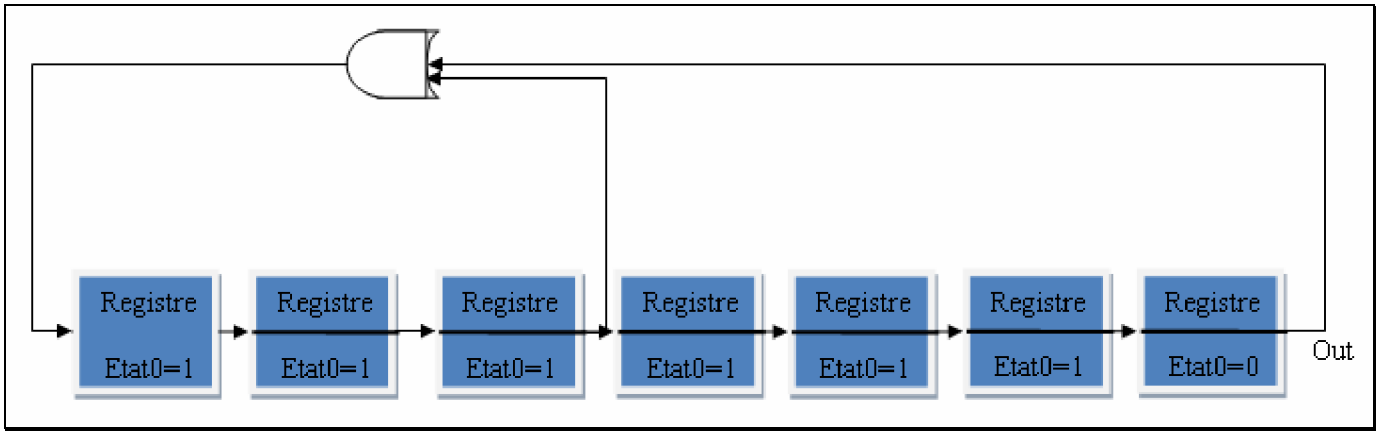


Figure 3.7 Registre générateur de pilote

Pour notre implémentation, nous avons adopté l'utilisation des registres car après la synthèse, nous constatons que cette dernière utilise moins de ressources hardware et en plus il est plus simple de programmer un registre qu'un FIFO.

A l'entrée du générateur de pilote, on a deux signaux l'un pour l'horloge (**clk**) et le second (**CE_pil**) pour valider la demande d'émission du pilote, car le pilote ne génère pas les signaux à la fréquence de l'horloge vue qu'ils ont des positions d'insertions précises.

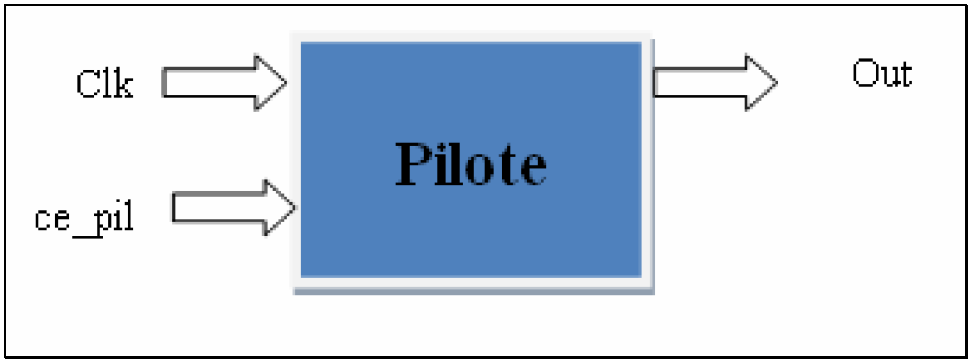


Figure 3.8 Bloc pilote

Ce signal est commandé par un contrôleur que l'on verra dans la section suivante. A la sortie de ce pilote, on a le signal pilote généré.

3.3.3 Entrelacement données, pilotes et zéros

Une fois les pilotes générés, ils doivent être insérer dans les données à des positions bien précises pour qu'ils puissent être identifiés et supprimés à la réception. Mais à part ces pilotes, des zéros sont aussi ajoutés sur les ports restant et eux doivent être aussi identifié à la réception, donc être insérés à des positions précises. Pour cela, on utilise un bloc qui fait le multiplexage entre les données, les pilotes et les zéros. Ce multiplexage tient compte du nombre de données transmises. Nous utiliserons un multiplexeur et un contrôleur pour contrôler la sélection au niveau du multiplexeur.

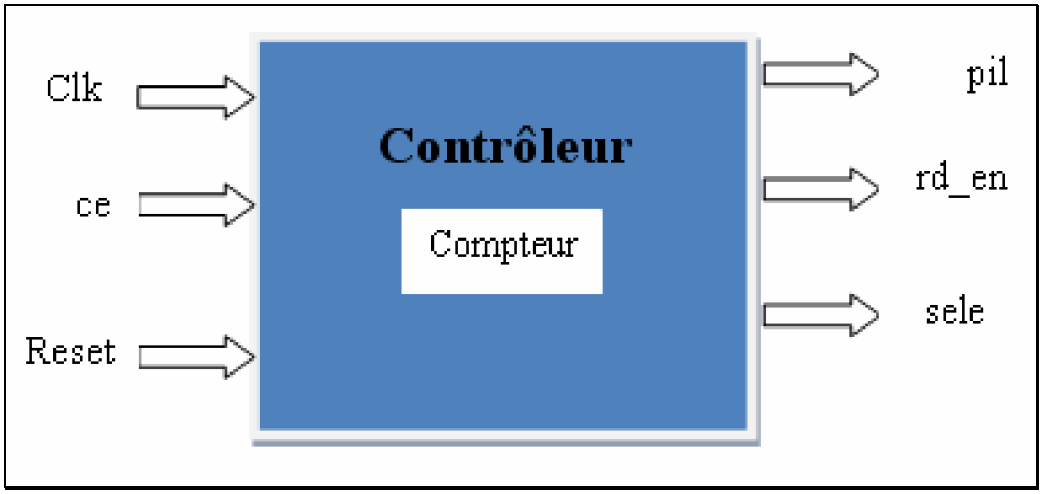


Figure 3.9 Bloc d'entrelacement

Le contrôleur est un bloc doté d'un compteur interne, ce compteur compte le nombre de données envoyées une fois la valeur 7 est atteinte, par exemple, il change le signal de sélection et le

multiplexeur prend en compte ce qui est à l'entrée du pilote, car à cet instant une valeur du pilote doit être insérée.

Le contrôleur a trois signaux d'entrées, un pour l'horloge (**clk**), un deuxième pour la commande (**ce**), car le compteur interne ne compte que lorsque des données sont transmis vers l'IFFT, or ce dernier reçoit les données à des intervalles de temps précis, vue qu'il travail en phase de réception et de calcul. Le troisième (**reset**) est pour la remise à zéro du compteur interne. Cela se produit à chaque fin de transmission d'un symbole OFDM, puisque les positions des pilotes sont les mêmes pour chaque symbole OFDM. Les signaux de sorties sont : le **pil** pour la commande du pilote, le **rd_en** pour le mapping et le **selec** pour la sélection au niveau du multiplexeur.

3.3.4 Génération des sous porteuses (IFFT)

Les sous porteuses sont générées à l'aide d'une IDFT comme expliqué (1.3.4). Comme dans la plupart des applications, on utilise l'algorithme de l'IFFT de Cooley-Tuckey pour implémenter le module de IDFT. La IFFT est plus performante à cause qu'elle nécessite moins de calcul et utilise moins de ressources hardware voir tableau3.5. Notre bloc FFT/IFFT est réalisé à l'aide de l' « intellectual property core » (IP core) de Xilinx. Nous utiliserons ici 64 porteuses dont 48 sous porteuses de données, 4 sous porteuses de pilotes et les autres sont mises à zéro comme pour la standard IEEE 802.11a.

L'IFFT à plusieurs signaux d'entrées, mais nous présenterons les signaux les plus importants figure3.10.

A l'entrée nous avons :

- **clk** : Le signal d'horloge est commun à tous les blocs.
- **xn_re** : reçoit la partie réelle de notre signal c'est-à-dire la phase du mapping.
- **xn_im** reçoit la partie imaginaire de notre signal qui représente la quadrature de phase.

Comme on peut le remarquer, les signaux ne sont pas transmis sur plusieurs bus, mais à chaque cycle d'horloge y'a un couple signal (**xn_re**, **xn_im**) est transmis, après réception des 64 couples, l'IFFT débute la phase de calcul.

- **Unload** : est utilisé pour commencer le chargement. On peut le laissé à un niveau haut pendant toute la durée de transmission.
- **start** est utilisé pour débuter une transmission. Il peut être laissé également à un niveau haut. C'est-à-dire qu'**unload** et **start** ne sont pas contrôlés par une commande externe.
- **fwd_inv** : permet de choisir entre la FFT à l'état haut ou l'IFFT à l'état bas.

A la sortie nous avons :

- **xk_re** : est utilisé pour la partie réelle du signal,
- **xk_im** : est utilisé pour la partie imaginaire.
- **xn_index** et **xk_index** donnent respectivement la position du signal en entrée et en sortie.
- **rfd** nous informe si la FFT est prête à recevoir des données en entrée.
- **busy** indique que la FFT est entraine d'effectuer des calculs.

Les signaux **xn_re** et **xn_im** sont reliées aux sorties des multiplexeurs de la phase et de la quadrature de phase. Les signaux **start** et **unload** sont mis à l'état haut durant toute la transmission. Le signal **fwd_inv** est à l'état bas durant toute la transmission pour le calcul d'IFFT. Le signal **rfd** est relié au contrôleur (CE : du contrôleur) pour permettre l'envoie des données vers la FFT/IFFT.

	FFT (N points)	DFT (N points)
Nombre d'additions	$Log_2(N)*N$	$N*(N-1)$
Nombre de multiplications	$Log_2(N)*N/2$	N^2

Tableau : 3.5 comparaison entre la FFT et la DFT

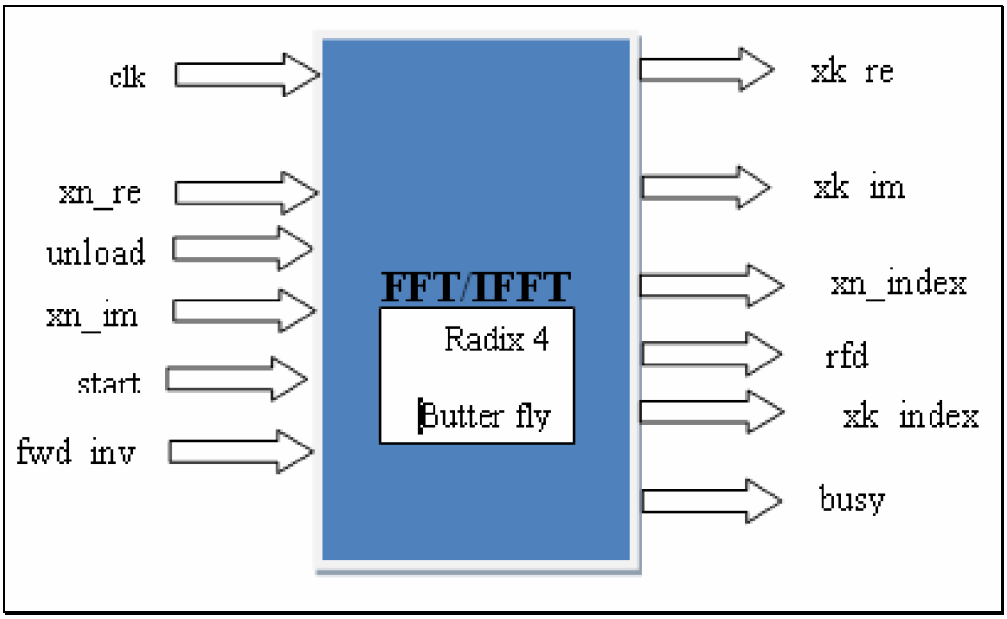


Figure 3.10 Bloc FFT

3.3.5 Intervalle de garde

L'intervalle de garde sert à lutter contre l'ISI. C'est une portion de la fin du symbole OFDM qui est rajouté au début du symbole. Pour réaliser l'intervalle de garde (figure3.11), nous utiliserons deux blocs mémoires :

- L'un pour stocker tout le symbole OFDM (**FIFO1**),
- et l'autre pour le stockage de l'intervalle de garde (**FIFO2**),

C'est le principe utilisé par **[3]**, mais nous apporterons quelques modifications, car au lieu d'utiliser un contrôleur externe pour les deux blocs, nous utiliserons les signaux des deux blocs pour les commander en lecture ou en écriture.

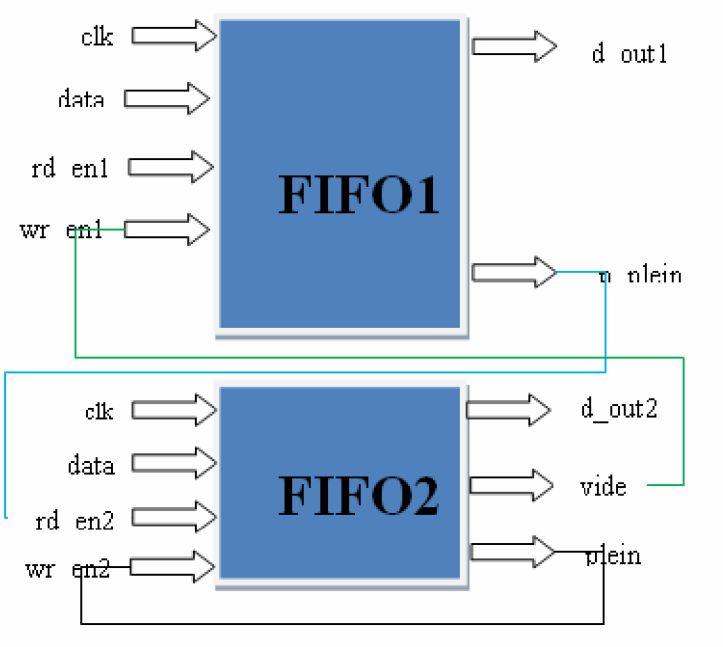


Figure 3.11 Bloc intervalle de garde

Les données arrivent sur le même bus au niveau des deux piles. La pile **fifo2** ne reçoit son signal de lecture qu'au moment ou la pile **fifo1** est presque pleine (**p_plein**).

A la réception du signal **p_plein,** les deux piles stockeront en même temps les derniers symboles envoyés. Une fois que tous les signaux sont reçus, le signal **plein** est mis à l'état « 1 » au niveau de la **fifo2** (ceci veut dire que la taille de cette pile corresponde aux derniers symboles envoyés), qui déclenchera le signal d'écriture **rd_en2** du **fifo2** à l'état '1' et l'envoie des données (les derniers symboles) par **fifo2**.

Après avoir envoyé toutes les symboles stockés, **fifo2** se vide. Le signal **vide** est mis à l'état '1', celui-ci entraine le signal **wr_en1** de **fifo1** à envoyer tous les symboles.

3.3.6 Fréquences intermédiaires

Avant la transmission, les signaux sont mis à une fréquence intermédiaire (RF) voir figure3.12. On utilise une modulation QAM ou PSK. Ceci consiste à moduler les donnée sortie de la constellation par un signal sinus ou cosinus. En OFDM, on module les parties : phase par un cosinus et quadrature de phase par un sinus. Nous utiliserons un générateur de signaux sinus et cosinus pour des angles bien

déterminés. Pour générer l'angle thêta, nous utilisons une mémoire avec les angles : 0, ½*π, π, -½*π, 2π. Il n'est pas nécessaire d'utiliser plusieurs points, vu qu'on aura par la suite un problème de génération de cette fréquence par rapport à celle utilisée par les blocs.

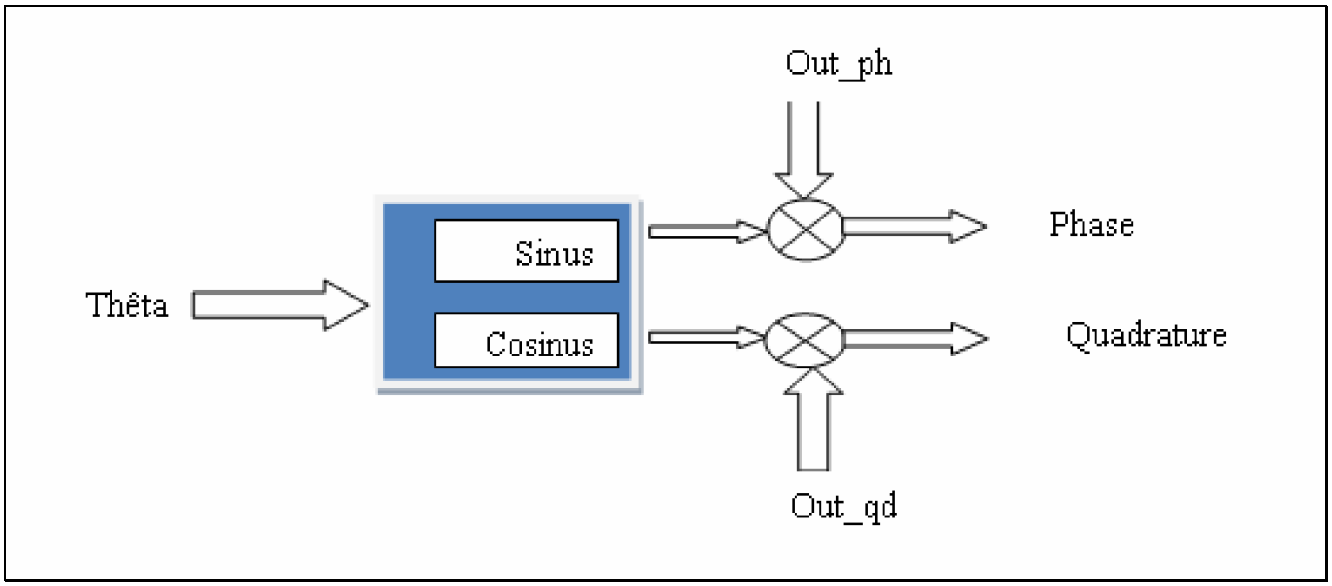


Figure 3.12 Bloc de générateur de FI

3.4 Test et validation de l'architecture

Nous présentons dans cette section :

- les programmes utilisés pour le modèle proposé,
- la synthèse,
- la simulation,
- et l'implémentation.

3.4.1 Programmes

Les programmes réalisés pour notre modèle sont donnés dans l'annexe.

Nous avons utilisé les programmes suivants :

Les chiffres représentent leur position dans le figure3.13.

- Pfe : programme principale faisant appel aux sous programmes des blocs constituant notre modulateur OFDM (3).
- Mapping : sous programme réalisant la fonction de mapping (4).
- BPSK : sous programme réalisant la modulation BPSK (8).

- QPSK4 : sous programme réalisant la modulation QPSK (5).
- 16-QAM : sous programme réalisant la modulation 16-QAM (6).
- 64-QAM : sous programme réalisant la modulation 64-QAM (7).
- Control : Contrôle la transmission des données à l'IFFT (10).
- Mux: permettent l'insertion des pilotes et des zéros (12).
- Fifo1 : Stockent des données de la constellation et permettent la synchronisation des blocs (9).
- IFFT : permet la génération des sous porteuses (13).
- DCM1 et DCM2 : permettent la division de la période de l'horloge par 2 et 4.
- Control2 : Contrôle l'ajout de l'intervalle de garde (15).
- Fifo2 : stocke le symbole OFDM pour l'insertion de l'intervalle de garde (14).
- Fifo3 : stockent l'intervalle de garde (16).
- Mux1 : Transmission de l'intervalle de garde et du symbole OFDM (17).
- Pilote : Génère les pilotes (11).
- Theta : Génère l'angle thêta (18).
- Cos_sin : Génère les signaux cosinus et sinus pour la fréquence intermédiaire (19).

On remarque que tous les programmes n'ont pas la même icône, Ceux qui on l'icône bleu sont des programmes directement écrit en VHDL et ceux qui ont l'icône rouge sont des programmes générés avec le « core generator » de Xilinx. Les blocs qui ont le même rôle utilisent le même programme, par exemple pour les multiplexeurs de phase et de quadrature de phase on a : **mux_ph-mux-Behavioral (mux.vhd)**, **mux_ph** est le nom d'instanciation dans le programme principal, **mux** est le nom de l'entité, **Behavioral** est le nom de l'architecture et **mux.vhd** le nom du fichier programme.

Donc si on prend **mux_ph-mux-Behavioral (mux.vhd)** et **mux_qd-mux-Behavioral (mux.vhd)**, cela veut dire que l'instanciation **mux_ph** et **mux_qd** utilise le même programme.

Nous avons réalisé ces programmes séparément. Un rassemblement de ses programmes se fait à l'aide du programme principal « pfe1 ». C'est certes l'un des avantages de la conception HDL, mais néanmoins il demeure un peu compliqué pour la synchronisation des différents blocs. La figure3.13 represente le programme principal et les liens faits avec les sous programmes.

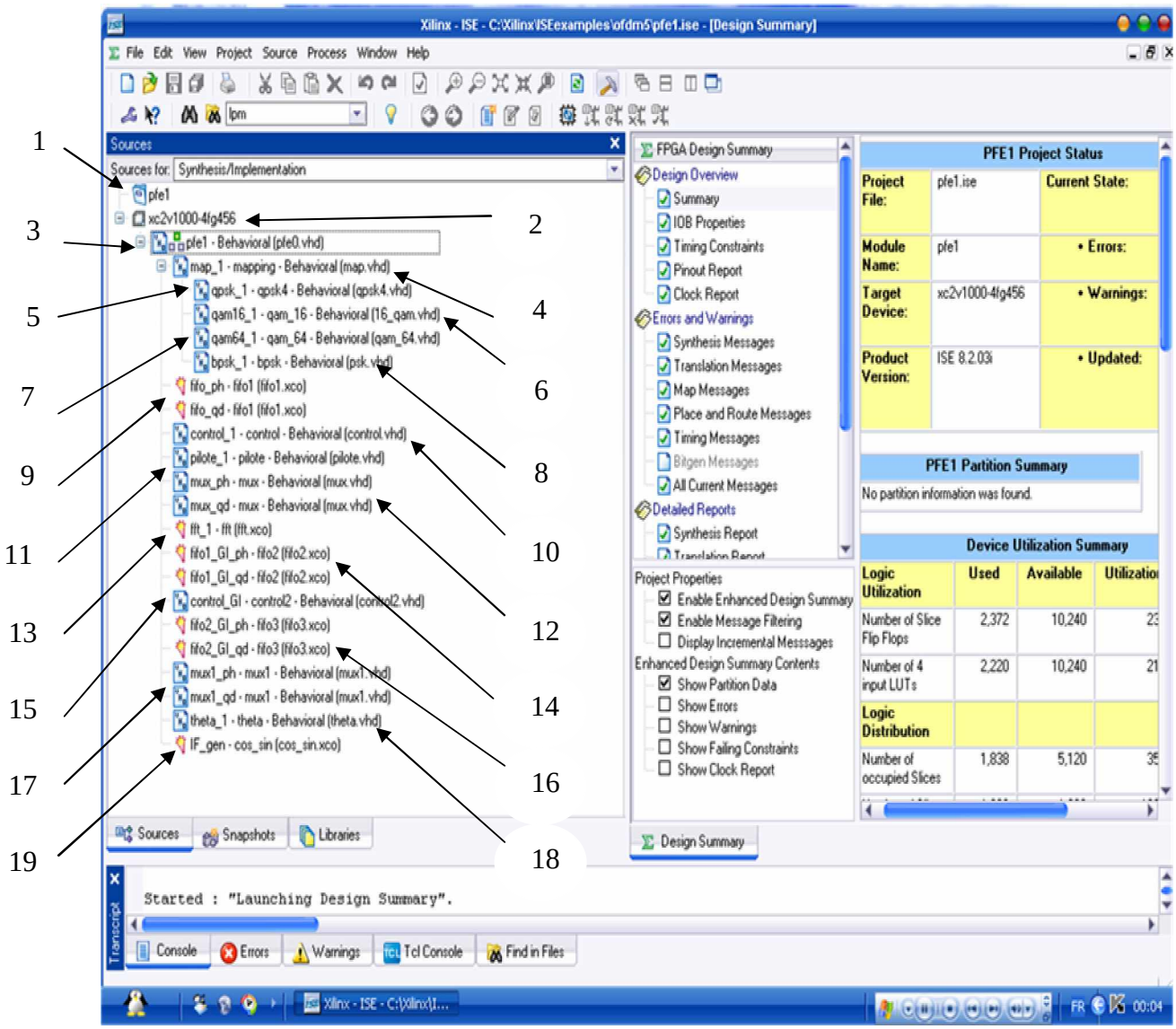


Figure 3.13 Schéma du projet avec les sous programmes

3.4.2 La synthèse

La synthèse de ce modèle a été réalisée avec une option de synthèse qui tient compte plus de la rapidité de l'architecture que des contraintes des ressources hardware. Nous avons remarqué au fil de notre travail, que la carte FPGA Virtex supporte un nombre élevé de module. De ce fait, nous nous sommes intéressés plus aux contraintes de temps que de ressources puisque ce module est dédié à la communication. L'exécution du programme principal a généré le schéma global de notre modulateur (figure3.14).

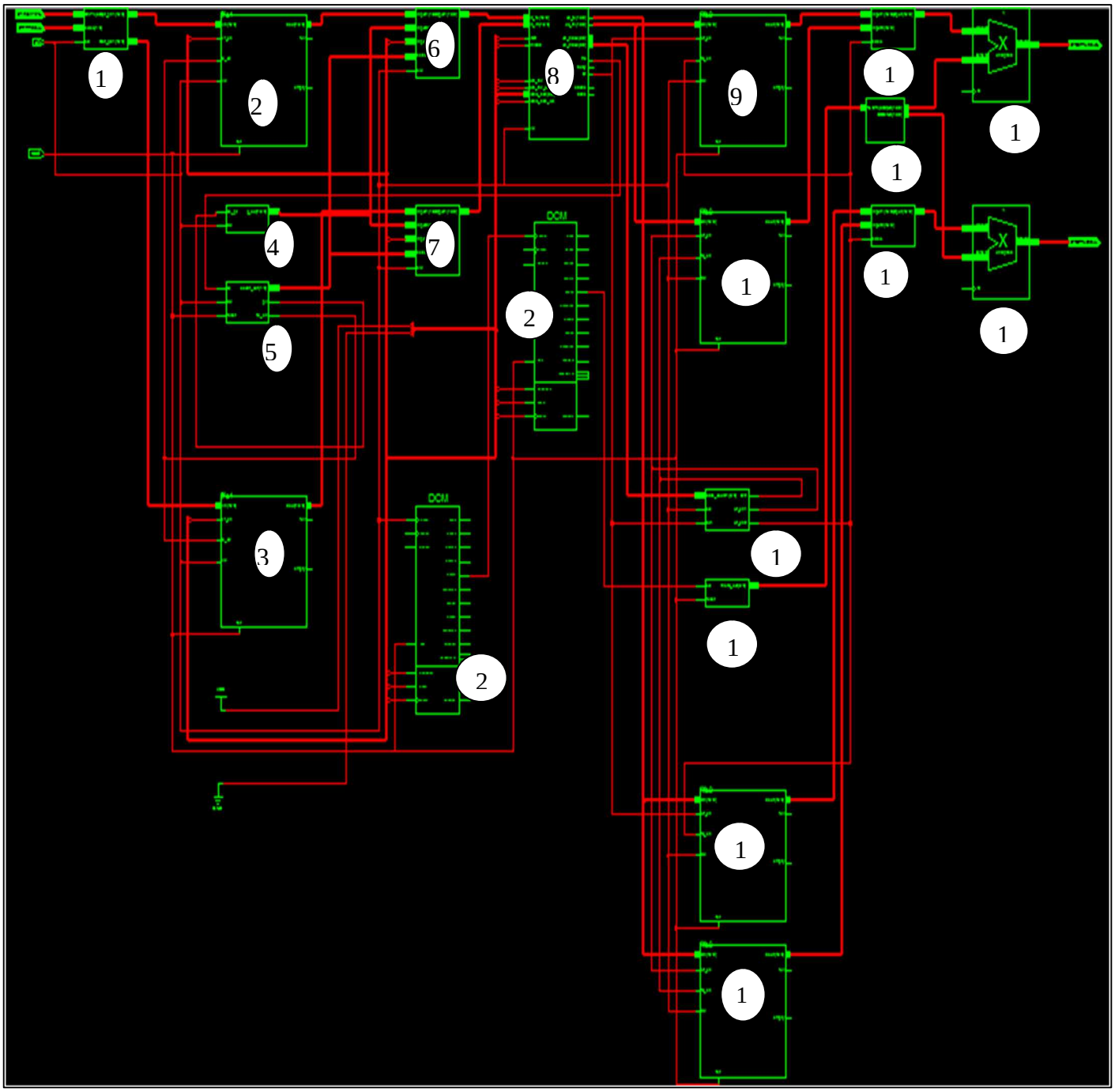


Figure 3.14 Vue de la synthèse de l'architecture

1. Mapping
2. Fifo1 (phase)
3. Fifo1 (quadrature de phase)
4. Pilote
5. Controleur1
6. Multiplexeur1 (phase)
7. Multiplexeur1 (Quadrature de phase)
8. IFFT
9. Fifo2 (phase)
10. Fifo2 (quadrature de phase)
11. Fifo3 (Phase)
12. Fifo3 (quadrature de phase)
13. Theta
14. Contrôleur2
15. Multiplexeur2 (phase)
16. Multiplexeur2 (Quadrature de phase)
17. Générateur de sinus et cosinus.
18. Multiplieur (phase)
19. Multiplieur (quadrature de phase)

On constate de ce schéma global les schémas de chaque sous bloc. Les schémas des sous blocs sont :

1. Schéma du mapping : On remarque les 4 blocs de BPSK, QPSK, 16-QAM, 64-QAM (figure.3.15).

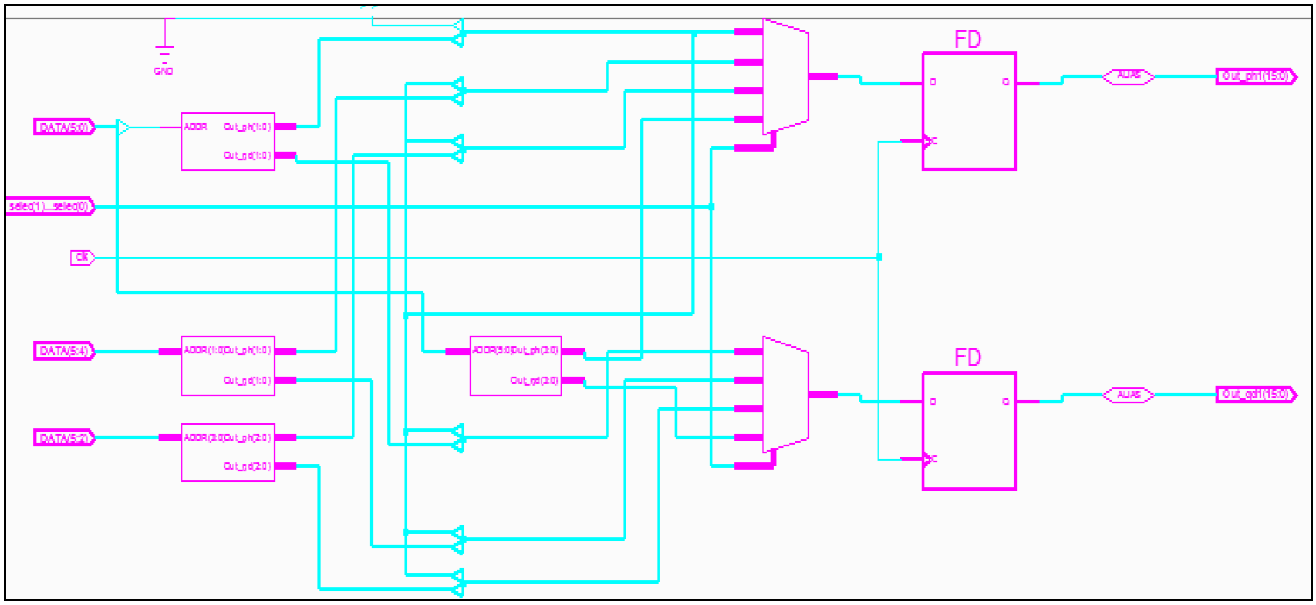


Figure 3-15 Mapping

2. Schéma du pilote : (figure.3.16)

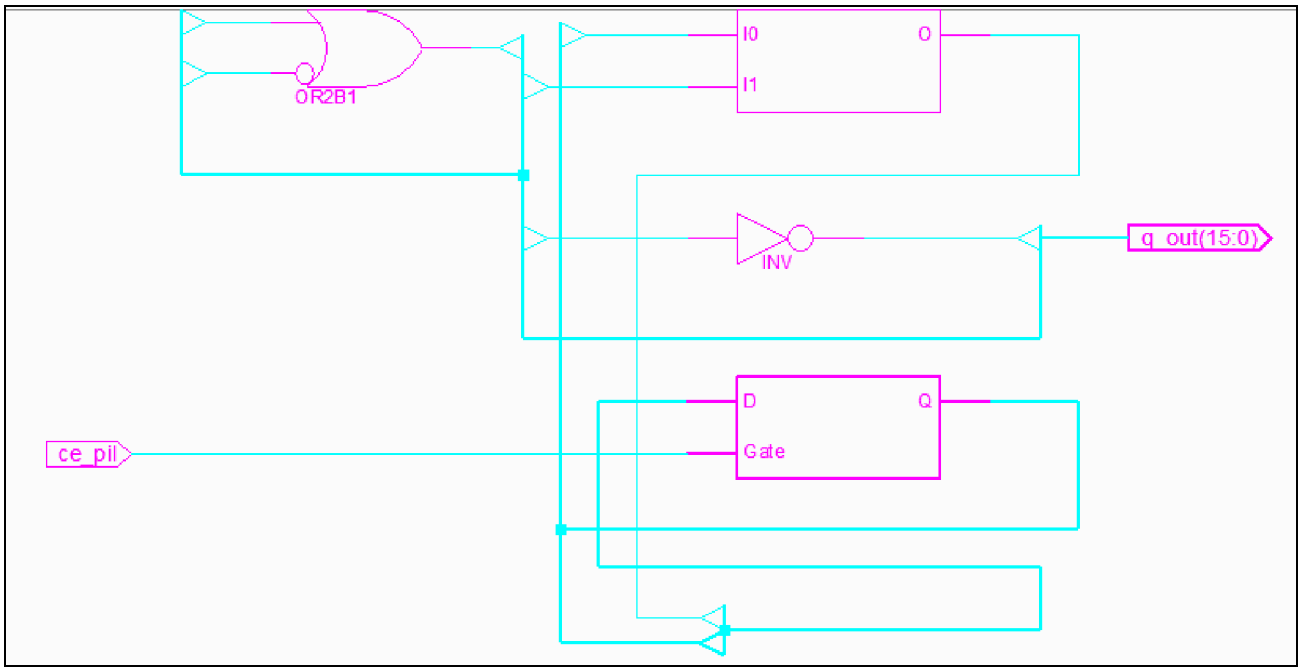


Figure 3-16 Pilote

3. Schéma du controleur1 : (figure3.17)

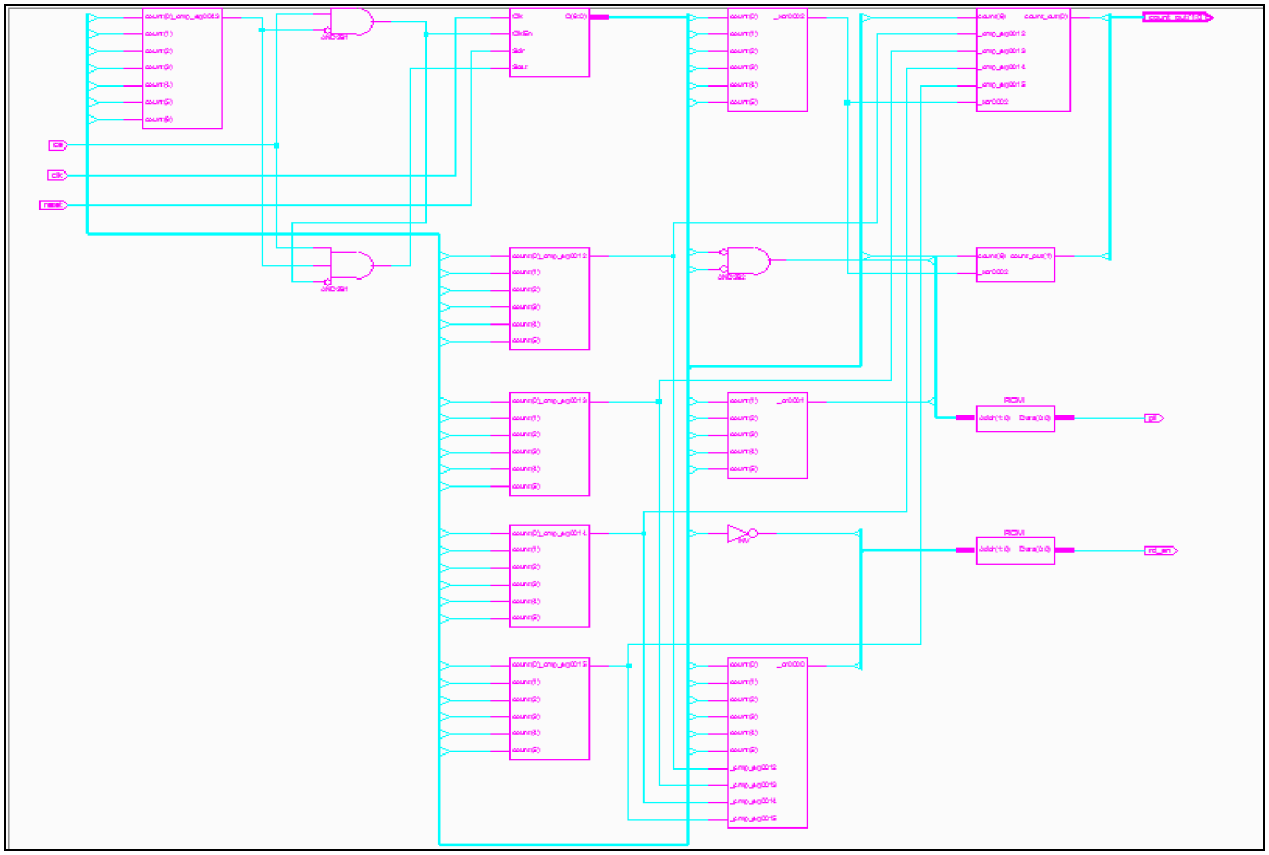


Figure 3-17 Contrôleur1

4. chéma du 64-QAM : Il a la même forme que les autres modulateurs (BPSK, QPSK, 16-QAM), Les ROMs contiennent la constellation (figure3.18).

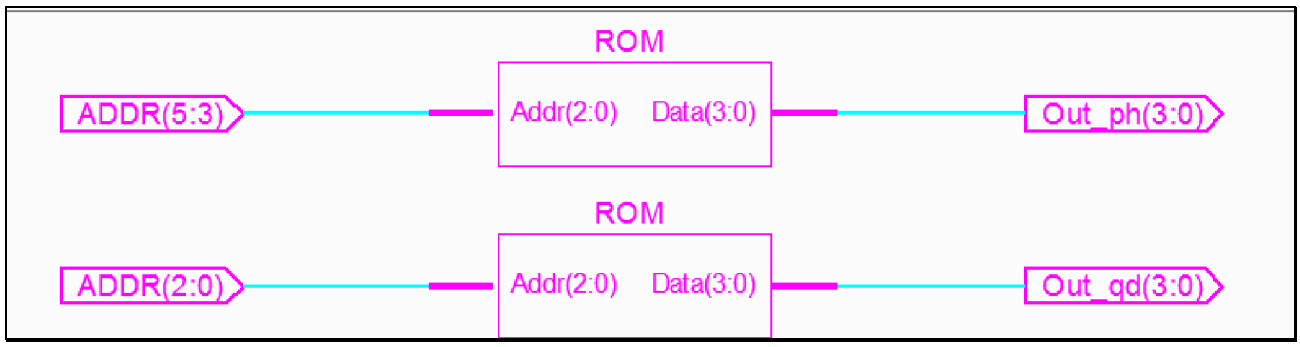


Figure 3-18 64-QAM

5. Schéma du Multiplexeur1 : (figure3.19).

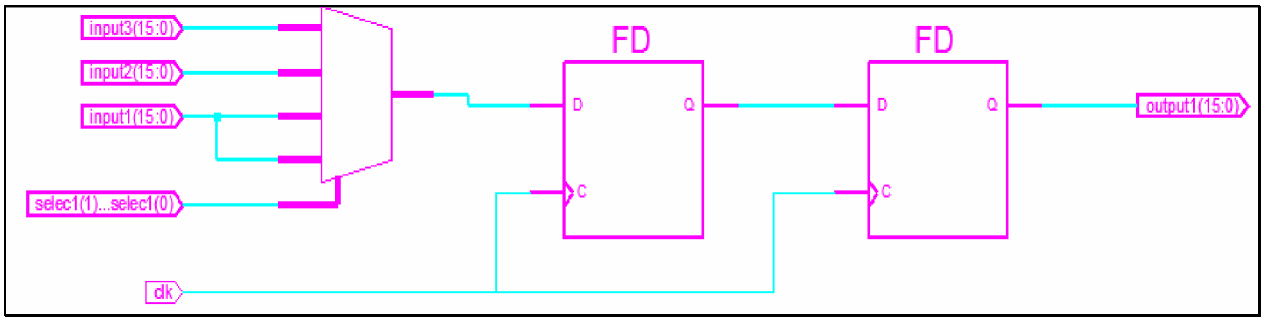


Figure 3-19 Multplexeur1

6. Schéma controleur2 : La valeur 101111=47 correspond au 47ème sous porteuse a partir de cette valeur on commence à prendre l'intervalle de garde. Donc l'intervalle de garde sera formé de 63-47= 16 sous porteuses qui correspondent à 25% du symbole OFDM. (wr_en1 permet de débuter l'écriture dans la FIFO d'intervalle de garde) (figure3.20).

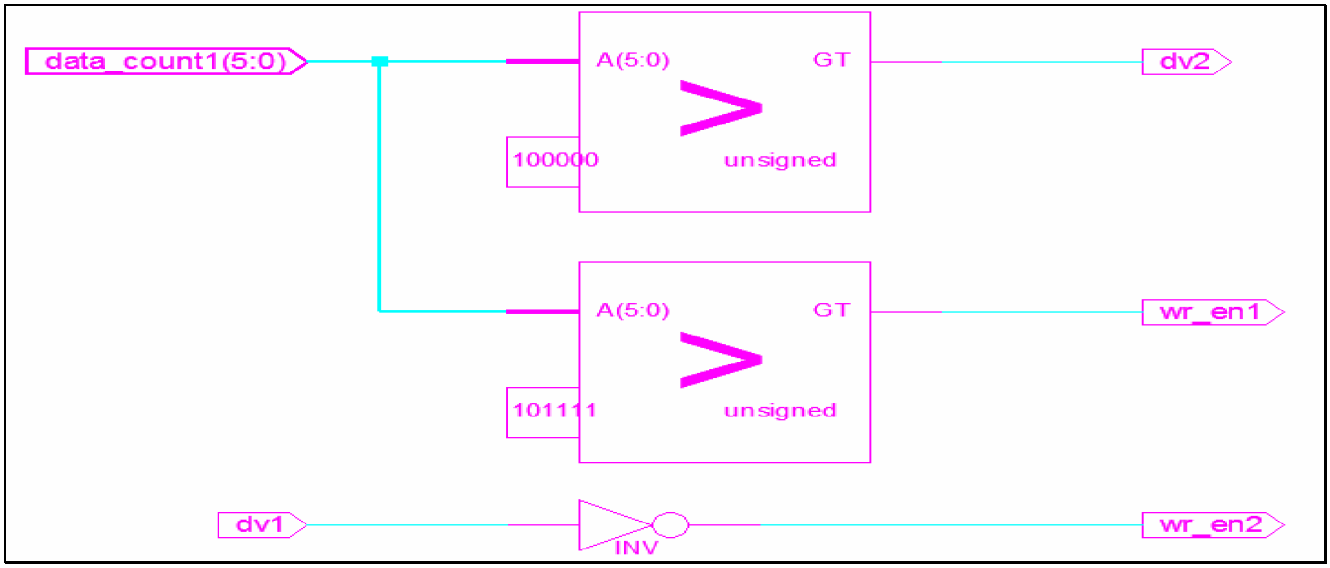


Figure 3-20 Contrôleur2

7. Schéma du thêta : (figure3.21).

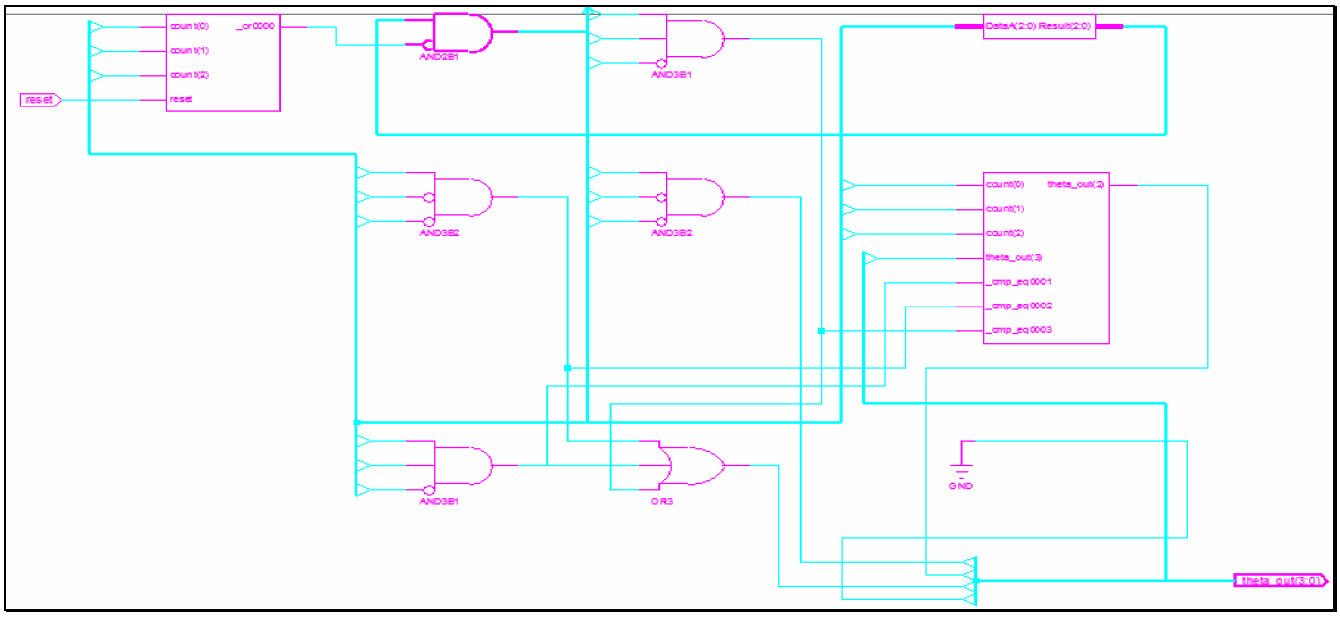


Figure 3-21 thêta

8. Schéma multiplexeur2 : (figure3.22).

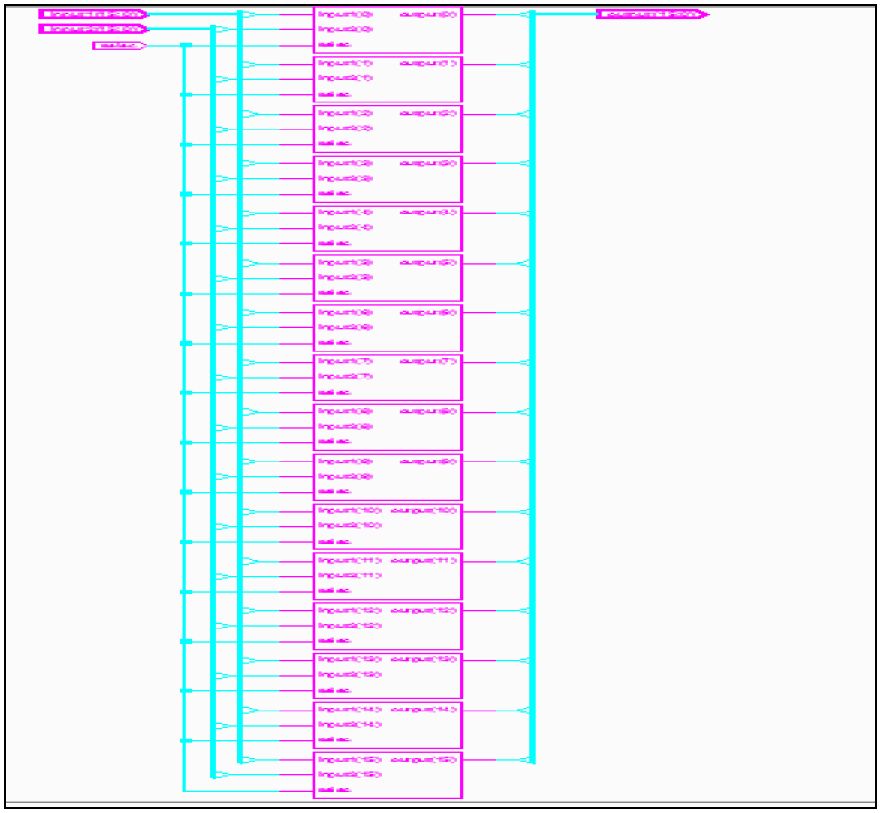


Figure 3-22 Multiplexeur2

9. Schéma de l'IFFT : (figure3.23)

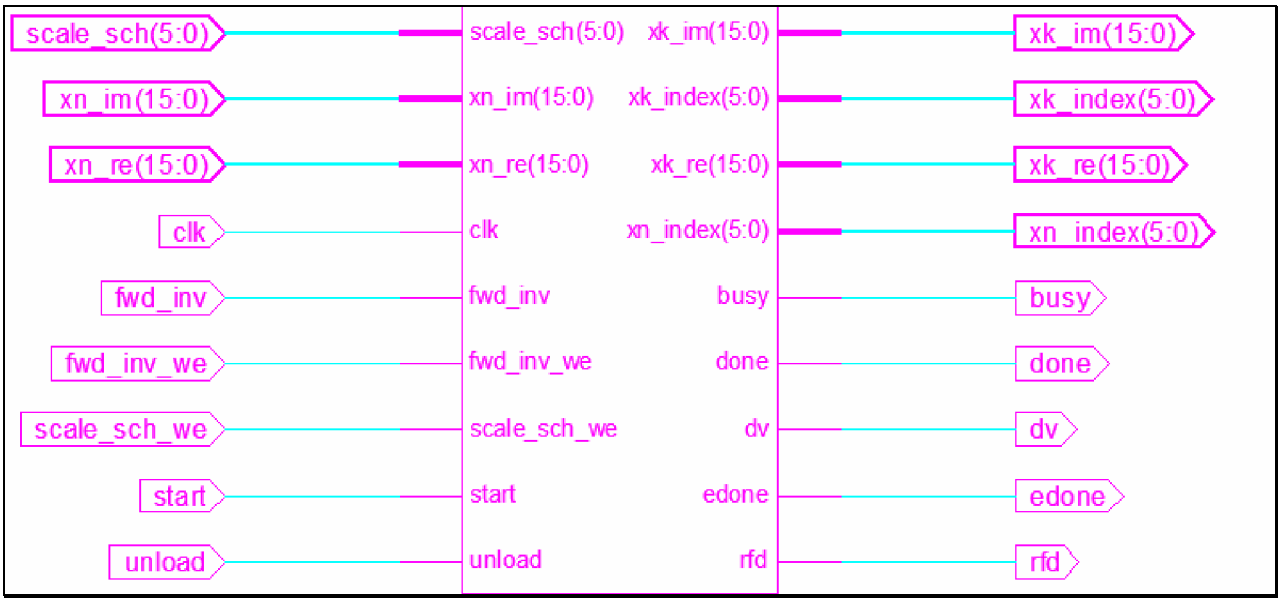


Figure 3-23 IFFT

10. Schéma de FIFO : Les piles ont la même forme (figure3.24).

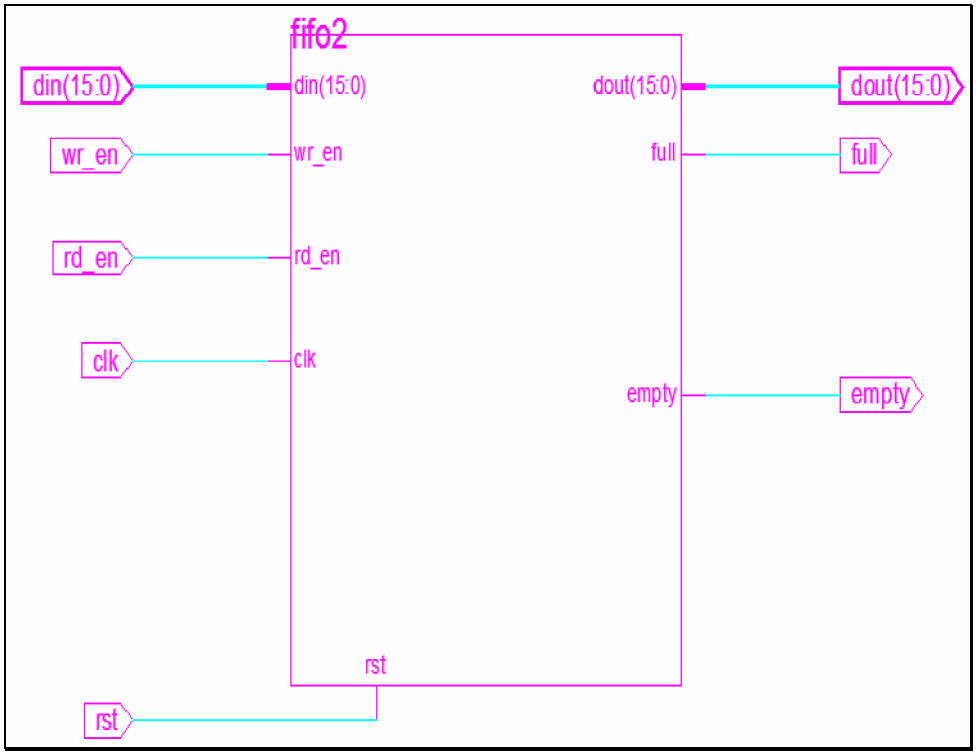
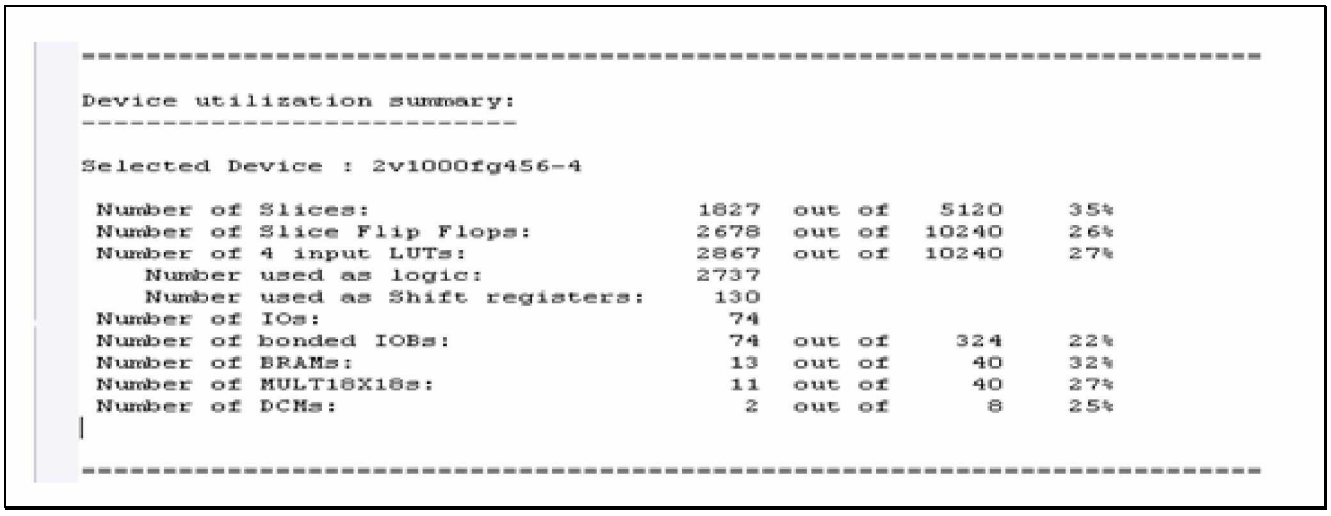


Figure 3-24 FIFO2

On peut remarquer une faible occupation de notre dispositif, nous avons une occupation moyenne de notre dispositif de 27,71% (figure3.25). Donc on il est bien possible de penser à la réalisation d'un démodulateur sur le même dispositif et/ou une augmentation de la performance du dispositif en utilisant plus de ressource. Généralement les implémentations dédiées à la communication font un compromis entre les contraintes de temps et les contraintes de ressources. Un rapprochement a été fait dans le tableau3.6 avec un modèle qui a été réalisé pour le standard 802.11a. On remarque que l'écart de ressources utilisées n'est pas très important.



```
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v1000fg456-4

 Number of Slices:                     1827  out of   5120    35%
 Number of Slice Flip Flops:           2678  out of  10240    26%
 Number of 4 input LUTs:               2867  out of  10240    27%
    Number used as logic:              2737
    Number used as Shift registers:     130
 Number of IOs:                          74
 Number of bonded IOBs:                  74  out of    324    22%
 Number of BRAMs:                        13  out of     40    32%
 Number of MULT18X18s:                   11  out of     40    27%
 Number of DCMs:                          2  out of      8    25%

=========================================================================
```

Figure 3-25 Rapport sur le nombre de blocs utilisés

	Modèle utilisé	Modèle de **[3]**
Nombre de slices	1827	1678
Nombre de slices flip flops	2678	2353
Nombre de 4 inputs Luts	2867	2814
Nombre de bonded IOBs	74	29
Nombre de BRAMs	13	12

Tableau : 3-6 Rapprochement de deux modèles du point de vue ressources.

```
--------------------------------------------------------------------------------

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 9.000ns (Maximum Frequency: 111.117MHz)
   Minimum input arrival time before clock: 4.047ns
   Maximum output required time after clock: 18.736ns
   Maximum combinational path delay: 25.276ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

================================================================================
```

Figure 3-26 Rapport sur le timing

La figure 3.26 donne le rapport du timing de notre modèle. C'est la partie la plus importante vue que notre modèle est dédié à la communication. On peut retenir surtout la fréquence d'utilisation de 111.117 MHz. La fréquence utilisée par la norme IEEE 802.11a est de l'ordre de 80 MHz, ce qui nous permet de dire que le modèle répond aux normes du standard. Dans le tableau3.7 nous avons rapprochement de nos résultats du timing obtenu avec le même modèle utilisé pour les ressources et l'exigence de la norme 802.11a.

	Modèle utilisé	Modèle du **[3]**	802.11a
Période minimale (ns)	9	10,876	12.5
Fréquence maximale (MHz)	111,117	91,948	80
Minimum input arrival time before clock (ns)	4,047	4,374	-
Minimum output required time after clock (ns)	18,736	6,973	-

Tableau 3.7 Rapprochement de deux modèles du point de vue timing

Nous n'avons pas voulu faire une comparaison des deux modèles vue qu'ils sont basés sur des spécifications différentes et le fait qu'on n'a pas respecté toutes les contraintes qu'imposais la standard IEEE 802.11a. Mais ce rapprochement permet de valider notre modèle au niveau du timing.

Nous n'avons pas utilisé de fichier de contrainte par manque de maîtrise de leur utilisation ce qui pourra bien diminuer les ressources utilisés ou augmenter la fréquence d'utilisation. Même si cette fréquence est très suffisante pour l'utilisation. Avec ces résultats il nous reste de vérifier le transfert de données. Pour cela nous avons utilisé le ModelSim c'est ce qui est présenté dans la section suivante.

3.4.3 Simulation

La simulation consiste à la vérification des transferts de données entre les blocs. Nous avons simulé notre modèle sur ModelSim. Nous présentons les niveaux les plus importants du transfert à savoir le mapping, l'insertion des pilotes et des zéros et l'ajout de l'intervalle de garde.

Ø **Mapping** : Sur le schéma de la figure3.27, nous avons les données qui sont en entrées sur **Data.** A la sortie nous avons les constellations du mapping. Le signal **selec** choisit la constellation comme prévu. On remarque bien le changement des valeurs de la constellation en fonction des valeurs de **selec**.

Ø **Insertion Données, pilotes, Zéros** : Pour tester l'insertion des pilotes et des zéros, nous observons la sortie des multiplexeurs de phase et de quadrature de phase. C'est au niveau de ses multiplexeurs que se fait le passage des données, des pilotes et des zéros (figure3.28).

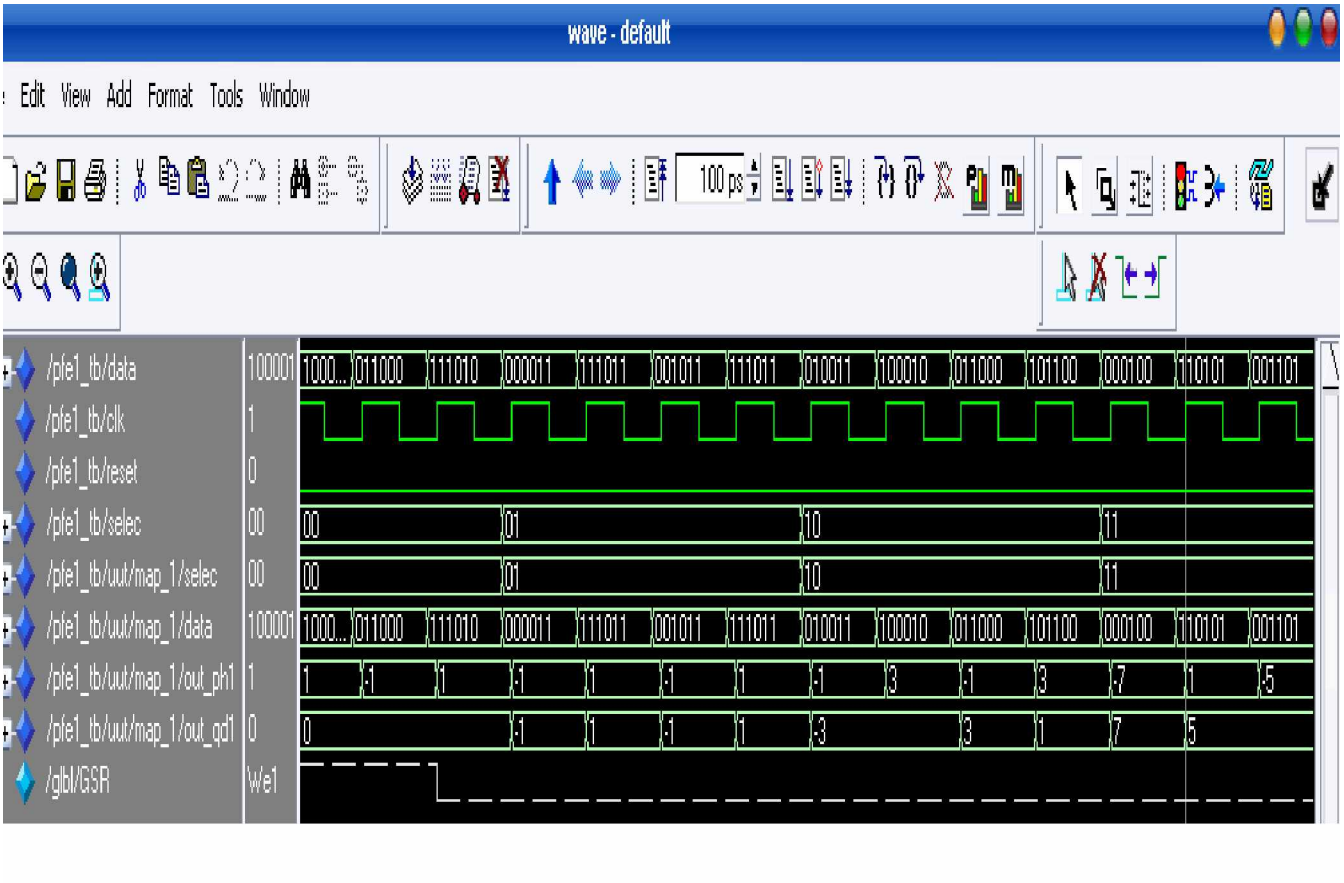


Figure 3-27 Signaux du Mapping

On remarque de la figure3.28, qu'il n'y a pas de perte de donnée après l'insertion des pilotes ou des zéros. L'information en sortie est conservée et transmis au prochain cycle d'horloge. On remarque un retard entre les données en sorties du mapping et les données en sorties du multiplexeur, ceci est causé par le contrôleur. La suppression de ce retard entraîne la perte des données lors de l'insertion des pilotes ou des zéros.

Ø **Intervalle de garde** : La présentation de l'architecture les données (3.3.5) sont stockées dans la pile des données. Si les données stockées atteignent les 75% du symbole, la pile de l'intervalle de garde reçoit en même temps les données restantes du symbole. Apres réception, cette pile commence

l'envoie des données reservées à l'intervalle de garde. Après avoir envoyé ses données la pile de données envoie tout le symbole (figure3.29). Au niveau de l'ajout de l'intervalle de garde, nous nous sommes intéressés à la sortie du multiplexeur, qui fait le passage entre la pile des données et la pile d'intervalle de garde. Le **selec** à l'état '1' de la figure3.19 permet de sélectionner la pile des données.

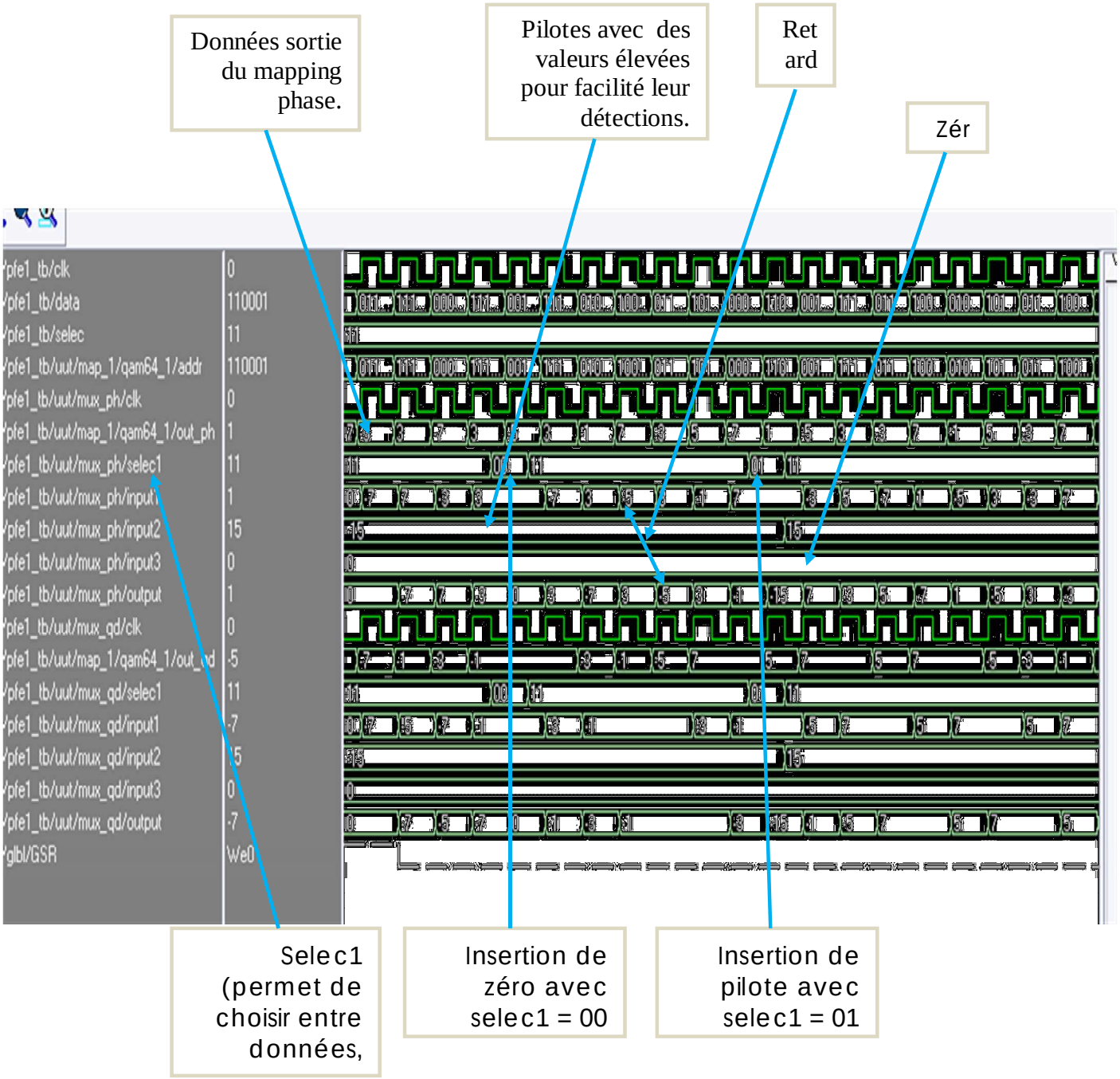


Figure 3-28 Signaux de sortie des multiplexeurs

Ø **Génération de pilotes** : La génération des pilotes est donnée par la figure3.30 au niveau du module pilote, car dans notre programme principal, les pilotes sont émis avec des niveaux beaucoup plus élevés. L'état '1' du pilote est multiplié par un facteur de 25 et l'état '0' est multiplié par un facteur 25 pour permettre la facilité de la détection des pilotes émis avec des niveaux d'énergie élevés, pour les différencier des autres données.

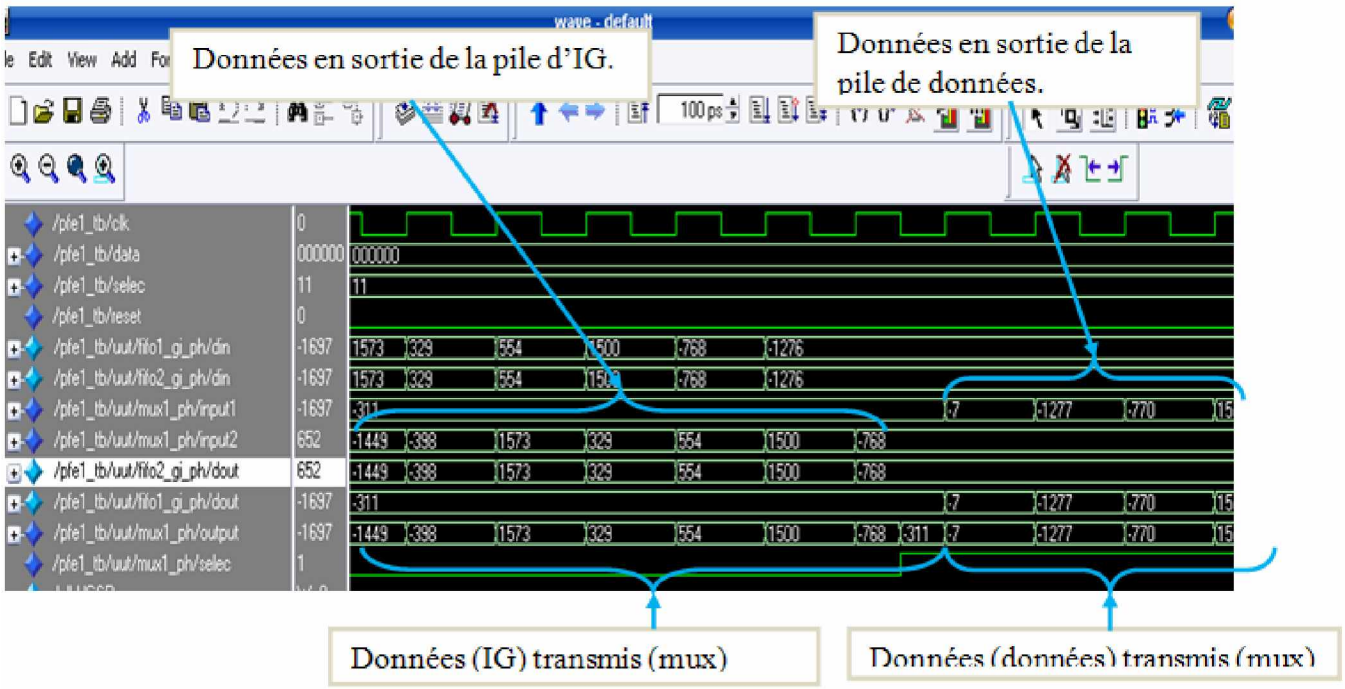


Figure 3-29 Ajout d'intervalle de garde

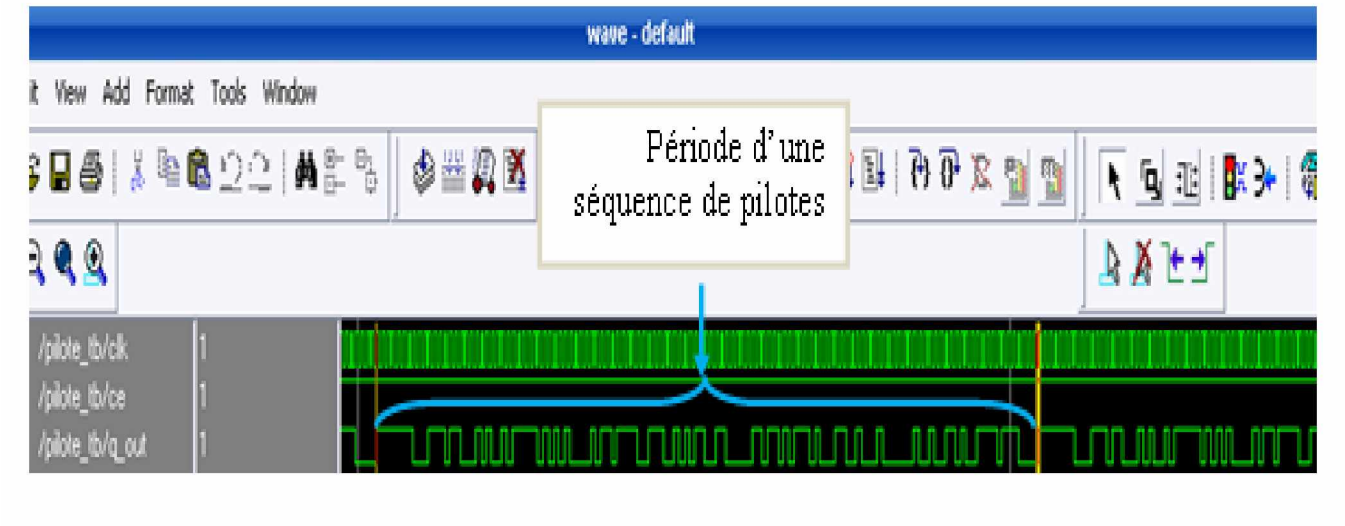


Figure 3-30 Génération de pilotes

Ø **Sortie de phase et de quadrature de phase** : Les signaux de sortie en phase et en quadrature de phase de la figure3.31 sont représentés sous la forme analogique. On peut remarquer 4 changements de ces valeurs analogiques pendant un cycle d'horloge. Ces signaux doivent être des sinusoïdes, mais du fait qu'on a généré seulement 4 points par période de l'horloge au niveau du générateur de thêta, on observe que leurs formes ne sont pas parfaites.

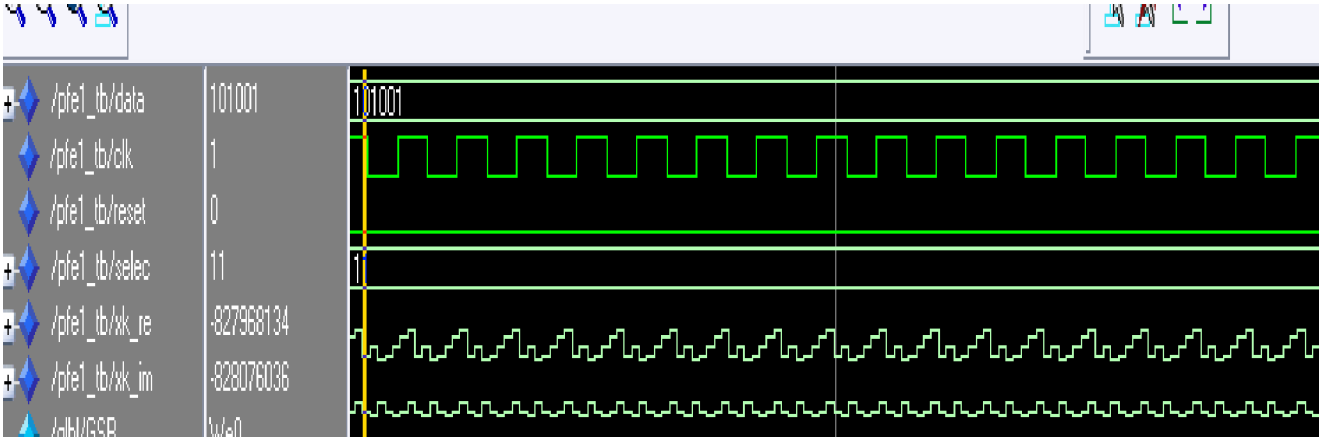


Figure 3-31 Signaux de sortie phase et quadrature

Cette représentation ne pose pas de problème à la réception [**15**], car il suffit d'avoir un détecteur pour ces valeurs. A cause des approximations sur thêta, on ne peut pas générer les valeurs 0, 1 et -1.

La simulation dans une conception sur FPGA est indispensable, mais elle ne constitue pas la finalité du travail. Il va falloir implémenter le module réalisé sur la carte FPGA comme mentionné en (2.3), ceci fera l'objet de la section suivante.

3.4.4 Implémentation

L'implémentation consiste à la mise sur circuit du modulateur OFDM. Elle se fait en trois étapes :

- la traduction (translate),
- le Map,
- et le placement et le routage.

Ces étapes permettent l'obtention des schémas de l'occupation du modulateur sur la carte FPGA.

Une information très utile, concernant la consommation du modulateur, est donnée par le Xpower de l'ISE en estimant la puissance nécessaire. A titre d'exemple, en supposant que la température ambiante est de 25°C, le Xpower donne une puissance de 351 mW. Cette consommation est acceptable.

La figure3.32 donne une vue du modulateur OFDM sur le circuit Virtex2 après le placement. On constate une faible occupation de la carte comme déjà indiqué dans la synthèse (3.4.2). La figure3.23

donne le schéma du modulateur après routage. On remarque sur les figures 3.34, 3.35 et 3.36 certains composants en rouge représentant :

- le control_1 pour l'insertion des pilotes et des zéros (figure3.34),
- le fifo de l'intervalle de garde pour la quadrature de phase (figure3.35),
- et la ligne de Data(5) (figure3.36).

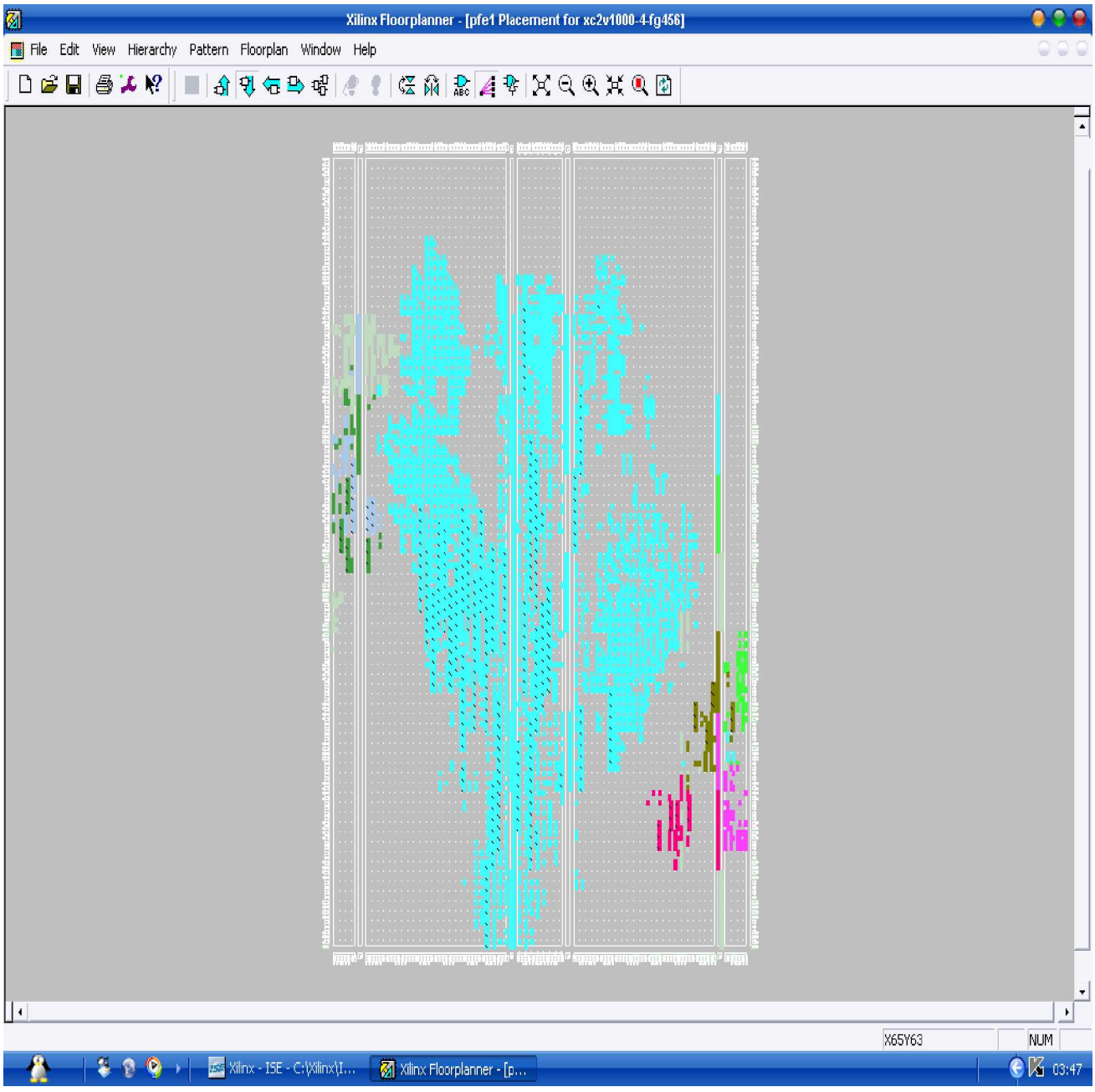


Figure 3-32 Vue du dispositif après placement

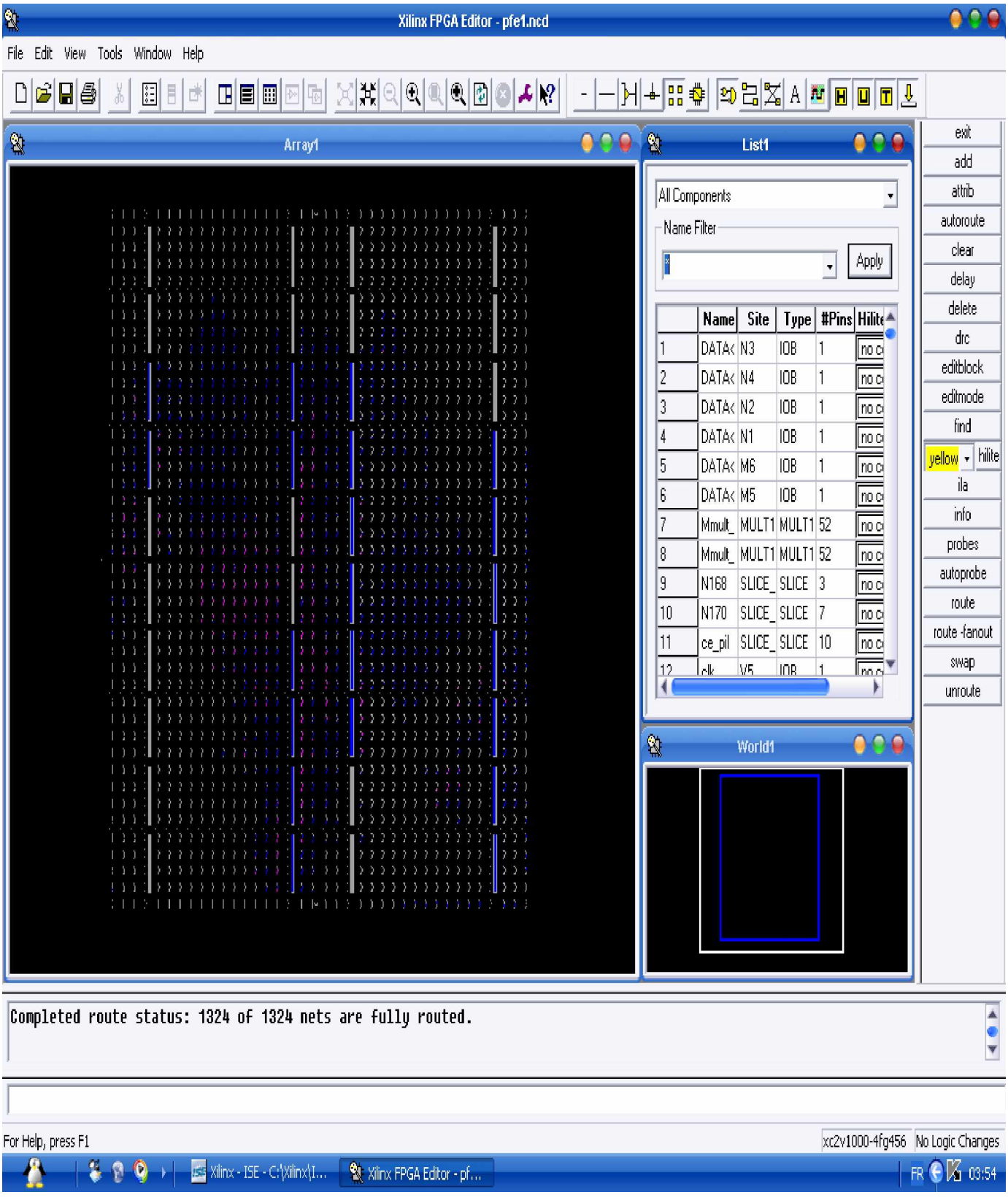


Figure 3-33 Vue du dispositif après routage

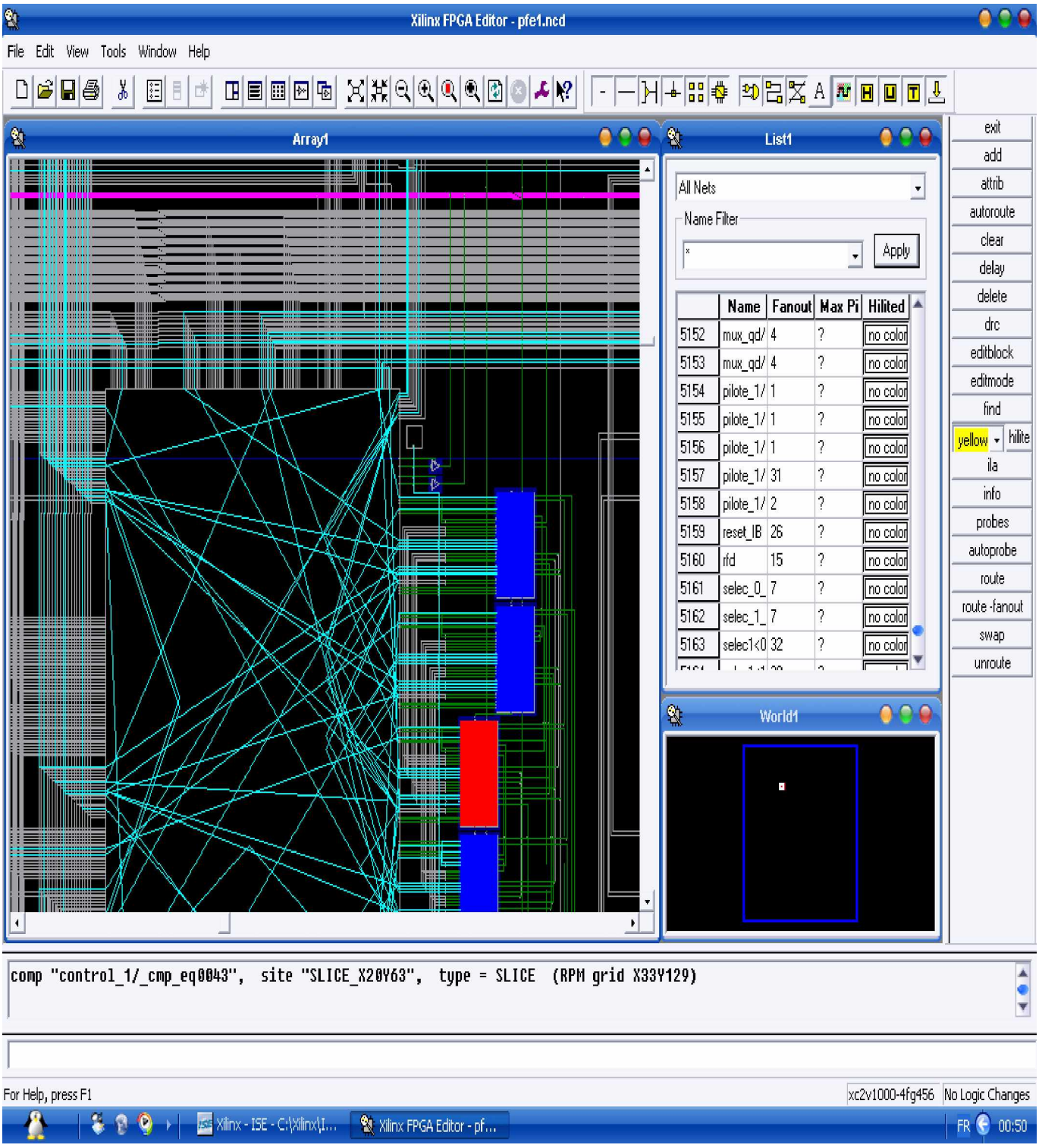


Figure 3-34 Vue du composant control_1 (rouge)

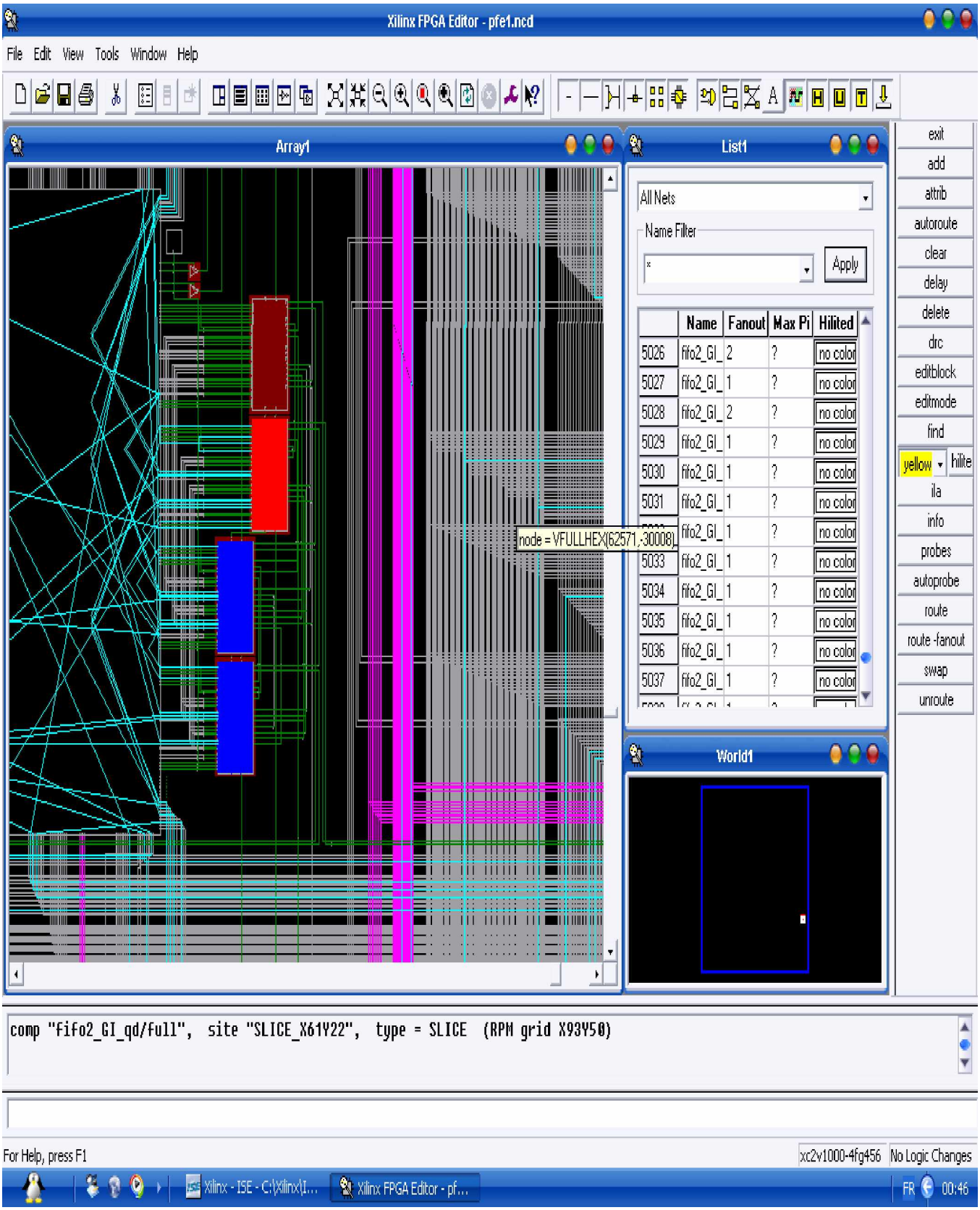


Figure 3-35 Vue du composant fifo2_GI_qd (rouge)

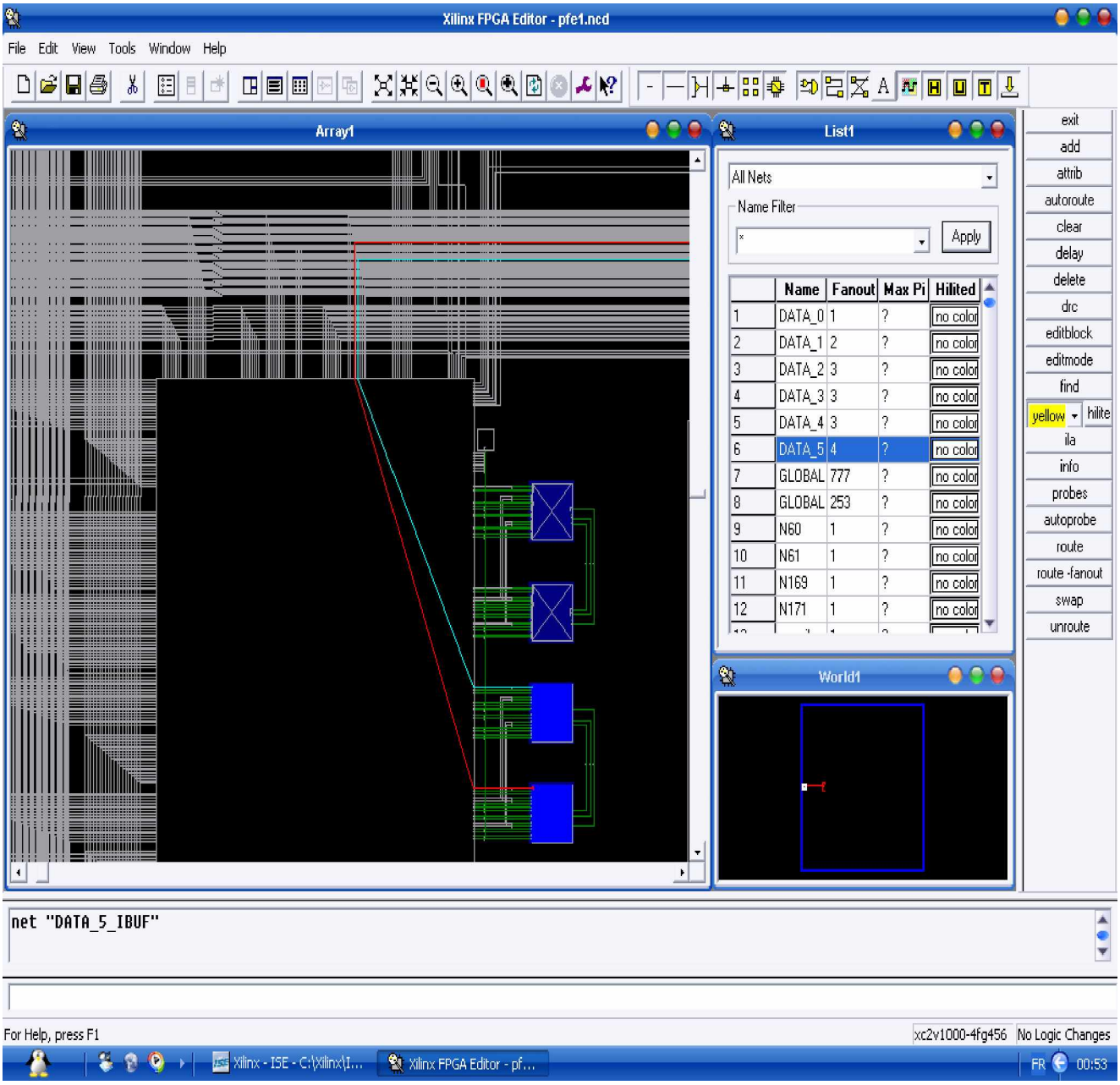


Figure 3-36 Vue de la ligne de donnée Data5 (ligne rouge).

3.5 Conclusion

A travers ce chapitre nous avons présenté l'architecture d'un modulateur OFDM. Cette architecture est surtout caractérisée par sa simplicité, elle répond également au standard IEEE 802.11a. Pour valider l'architecture proposée nous avons réalisé ses différents blocs à l'aide du VHDL, après synthèse du programme, nous avons obtenu un résultat très proche de l'architecture décrit. Nous avons pu constater une occupation de moins de 30% de la carte par notre architecture. Nous avons également une fréquence maximale d'utilisation de 111.117 MHz qui est bien au delà des 80 MHz exigé par le standard IEEE 802.11a. Après la synthèse nous avons simulé le transfert des données de notre

modulateur et nous nous sommes intéressés sur les points sensibles de l'architecture à savoir l'insertion des pilotes et des zéros et l'ajout de l'intervalle de garde. Ceci nous a permis de constater un retard d'un cycle d'horloge qui était nécessaire pour ne pas perdre la donnée à transmettre en cas d'insertion de pilote ou de zéro.

Enfin nous avons implémenté notre circuit sur la carte en passant par le Map, le placement et le routage. Les schémas fournis nous ont permis de remarqué la même faible occupation du modulateur sur la carte, et l'emplacement de certains composants sur la carte.

Les problèmes rencontrés au cours de notre implémentation se résume à un problème de synthèse de certains instructions comme wait for et types comme les nombres à virgule du VHDL par l'XST.

Conclusion générale

L'objectif de ce travail est l'implémentation d'un modulateur ODFM, au delà de cet objectif nous avons tenté de nous rapproché d'un des standard le plus utilisée dans le WLAN qui utilise un modulateur OFDM. Ce standard est le 802.11a.

Pour la réalisation de notre travail nous avons tout d'abord mis au point une architecture à partir des différents documents qu'on a eu à consulter. Avec cette architecture nous avons fait un programme en VHDL que nous avons synthétisé et implémenté à l'aide de l'outil de conception de Xilinx. Le choix du VHDL est basé sur sa flexibilité tant du point de vu architecture que matériel. A partir des résultats de synthèse et d'implémentation, nous avons pu, valider notre architecture en montrant qu'on a surmonté les contraintes de ressources et de timing. D'autre part nous avons présenté la simulation des transferts de données à travers les différents points les plus sensibles de l'architecture, à savoir l'insertion des pilotes et des zéros, et les blocs d'ajout d'intervalle de garde.

Les perspectives qui sont en vue est surement un travail complémentaire qui va soit vers l'adaptation complète de notre modulateur à un standard avec l'insertion d'une couche MAC, soit à l'ajout d'un démodulateur pour avoir sur le même dispositif un modulateur et démodulateur, car les ressources sont assez suffisant puis que nous n'avons pas dépassé la barre des 30% du dispositif. Et surtout qu'il n'y a pas besoin d'un nouveau bloc FFT, car celui que nous avons utilisé peut fonctionner dans les deux sens, il suffit de commander le signal **fwd_in** qui a été présenté dans l'architecture en modulation en état '0' ou en démodulation en état '1'.

Annexe : Les programmes VHDL.

Programme principale.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;
entity pfe1 is
port ( DATA : in std_logic_vector (5 downto 0);
clk, reset : in std_logic;
selec:in std_logic_vector (1 downto 0);
xk_re: OUT std_logic_VECTOR(31 downto 0);
xk_im: OUT std_logic_VECTOR(31 downto 0));
end pfe1;
architecture Behavioral of pfe1 is
signal  rfd, rd_en,dv1,dv2,wr_en1,ce_pil,wr_en2,clkdv1,clkdv2: std_logic;
--signal empty1: std_logic;
signal data_count1: std_logic_vector ( 5 downto 0);
signal selec1 : std_logic_vector ( 1 downto 0);
signal theta_out : std_logic_vector ( 3 downto 0);
signal q_out1,din_ph, din_qd, dout_ph, dout_qd,out_put_ph,
out_put_qd,xk_re1,xk_im1,xk_re12,xk_im12,xk_re2,xk_im2,xk_re3,xk_im3,sine, cosine: std_logic_vector (15 downto 0);
-------------------------------------------------------------
component mapping is
port ( DATA : in std_logic_vector (5 downto 0);
clk : in std_logic;
selec:in std_logic_vector (1 downto 0);
--w_fifo:out std_logic;
Out_ph1, Out_qd1 : out std_logic_vector (15 downto 0));
end component;
-------------------------------------------------------------
component control is
port ( clk, reset, ce :in  std_logic;
pil, rd_en : out std_logic;
       count_out : out std_logic_vector (1 downto 0));
end component;
-------------------------------------------------------------------
component pilote is
port (clk,ce_pil:in std_logic;
     q_out: out std_logic_vector(15 downto 0));
end component;
-------------------------------------------------------------
component mux is
port ( input1,input2, input3:in std_logic_vector (15 downto 0);
     selec1: in std_logic_vector (1 downto 0);
                        clk :std_logic;
                        output1 : out std_logic_vector (15 downto 0));
end component;
-------------------------------------------------------------
component fft
            port (
            xn_re: IN std_logic_VECTOR(15 downto 0);
```

```
        xn_im: IN std_logic_VECTOR(15 downto 0);
        start: IN std_logic;
        unload: IN std_logic;
        fwd_inv: IN std_logic;
        fwd_inv_we: IN std_logic;
        scale_sch: IN std_logic_VECTOR(5 downto 0);
        scale_sch_we: IN std_logic;
        clk: IN std_logic;
        xk_re: OUT std_logic_VECTOR(15 downto 0);
        xk_im: OUT std_logic_VECTOR(15 downto 0);
        xn_index: OUT std_logic_VECTOR(5 downto 0);
        xk_index: OUT std_logic_VECTOR(5 downto 0);
        rfd: OUT std_logic;
        busy: OUT std_logic;
        dv: OUT std_logic;
        edone: OUT std_logic;
        done: OUT std_logic);
end component;

-- FPGA Express Black Box declaration
attribute fpga_dont_touch: string;
attribute fpga_dont_touch of fft: component is "true";

-- Synplicity black box declaration
attribute syn_black_box : boolean;
attribute syn_black_box of fft: component is true;
---------------------------------------------------------------------
component fifo1
        port (
        clk: IN std_logic;
        din: IN std_logic_VECTOR(15 downto 0);
        rd_en: IN std_logic;
        rst: IN std_logic;
        wr_en: IN std_logic;
        dout: OUT std_logic_VECTOR(15 downto 0);
        empty: OUT std_logic;
        full: OUT std_logic);
end component;

-- FPGA Express Black Box declaration
--attribute fpga_dont_touch: string;
attribute fpga_dont_touch of fifo1: component is "true";

-- Synplicity black box declaration
attribute syn_black_box of fifo1: component is true;
---------------------------------------------------------------------
component fifo2
        port (
        clk: IN std_logic;
        din: IN std_logic_VECTOR(15 downto 0);
        rd_en: IN std_logic;
        rst: IN std_logic;
        wr_en: IN std_logic;
        --data_count: OUT std_logic_VECTOR(5 downto 0);
        dout: OUT std_logic_VECTOR(15 downto 0);
        empty: OUT std_logic;
        full: OUT std_logic);
end component;
```

```
-- FPGA Express Black Box declaration
--attribute fpga_dont_touch: string;
attribute fpga_dont_touch of fifo2: component is "true";

-- Synplicity black box declaration
--attribute syn_black_box : boolean;
attribute syn_black_box of fifo2: component is true;
-------------------------------------------------------------------------
component fifo3
            port (
            clk: IN std_logic;
            din: IN std_logic_VECTOR(15 downto 0);
            rd_en: IN std_logic;
            rst: IN std_logic;
            wr_en: IN std_logic;
            dout: OUT std_logic_VECTOR(15 downto 0);
            empty: OUT std_logic;
            full: OUT std_logic);
end component;

-- FPGA Express Black Box declaration
--attribute fpga_dont_touch: string;
attribute fpga_dont_touch of fifo3: component is "true";

-- Synplicity black box declaration
--attribute syn_black_box : boolean;
attribute syn_black_box of fifo3: component is true;
-------------------------------------------------------------------------
component mux1 is
port( input1,input2:in std_logic_vector ( 15 downto 0);
    output: out std_logic_vector ( 15 downto 0);
              selec: in std_logic);
end component;
-----------------------------------------
component control2 is
port (clk,dv1: in std_logic;
data_count1: std_logic_vector ( 5 downto 0);
wr_en1,dv2,wr_en2: out std_logic);
end component;
-----------------------------------------------------------------
component theta is
port ( clk, reset :in  std_logic;
       theta_out : out std_logic_vector (3 downto 0));
end component;
-----------------------------------------------------------------
component cos_sin
            port (
            THETA: IN std_logic_VECTOR(3 downto 0);
            SINE: OUT std_logic_VECTOR(15 downto 0);
            COSINE: OUT std_logic_VECTOR(15 downto 0));
end component;
-- FPGA Express Black Box declaration
--attribute fpga_dont_touch: string;
attribute fpga_dont_touch of cos_sin: component is "true";

-- Synplicity black box declaration
--attribute syn_black_box : boolean;
```

```
attribute syn_black_box of cos_sin: component is true;
-------------------------------------------------------------------------------
begin
-------------------------------------------------------------
map_1: mapping
port map ( DATA => DATA,
clk =>clk,
selec => selec,
--w_fifo => w_f,
Out_ph1 => din_ph,
Out_qd1 => din_qd);
--end mapping;
-------------------------------------------------------------
fifo_ph : fifo1
                port map (
                        clk => clk,
                        din => din_ph,
                        rd_en => rd_en,
                        rst => reset,
                        wr_en => '1',
                        dout => dout_ph,
                        empty => open,
                        full => open);
-------------------------------------------------------------
fifo_qd : fifo1
                port map (
                        clk => clk,
                        din => din_qd,
                        rd_en => rd_en,
                        rst => reset,
                        wr_en => '1',
                        dout => dout_qd,
                        empty => open,
                        full => open);
-------------------------------------------------------------
control_1: control
port map( clk => clk,
reset => reset,
ce => rfd,
rd_en =>rd_en,
pil => ce_pil,
count_out => selec1);
--end control;
-------------------------------------------------------------------
pilote_1: pilote
port map (clk,
ce_pil => ce_pil,
q_out => q_out1);
--end pilote;
-------------------------------------------------------------
mux_ph: mux
port map( clk => clk,
 input1 => dout_ph,
input2 => q_out1 ,
input3 => "0000000000000000",
selec1 => selec1,
output1 => out_put_ph);
--end mux;
```

```
-----------------------------------------------------------------
mux_qd: mux
port map( clk=>clk,
input1 => dout_qd,
input2 => q_out1,----------verifier
input3 => "0000000000000000",
selec1 => selec1,
output1 => out_put_qd);
--end mux;
-----------------------------------------------------------------
fft_1 : fft
                port map (
                        xn_re => out_put_ph,
                        xn_im => out_put_qd,
                        start => '1',
                        unload => '1',
                        fwd_inv => '0',
                        fwd_inv_we => '0',
                        scale_sch => "101011",
                        scale_sch_we => '0',
                        clk => clk,
                        xk_re => xk_re1,
                        xk_im => xk_im1,
                        xn_index => open,
                        xk_index => data_count1,
                        rfd => rfd,
                        busy => open,
                        dv => dv1,
                        edone => open,
                        done => open);
-----------------------------------------------------------------
fifo1_GI_ph : fifo2
                port map (
                        clk => clk,
                        din => xk_re1,
                        rd_en => wr_en2,
                        rst => reset,
                        wr_en => dv1,
                        --data_count=> open,--prog_full => wr_en1,
                        dout => xk_re2,
                        empty => open,
                        full => open);


-----------------------------------------------------------------
fifo1_GI_qd : fifo2
                port map (
                        clk => clk,
                        din => xk_im1,
                        rd_en => wr_en2,
                        rst => reset,
                        wr_en => dv1,
                        --data_count => open,
                        dout => xk_im2,
                        empty => open,
                        full => open);
-----------------------------------------------------------------
control_GI : control2
port map ( clk => clk,
```

```
dv1 =>dv1,
dv2=>dv2,
data_count1 => data_count1,
wr_en2 =>wr_en2,
wr_en1 => wr_en1);
----------------------------------------------------------------
fifo2_GI_ph : fifo3
                port map (
                        clk => clk,
                        din => xk_re1,
                        rd_en => dv2,
                        rst => reset,
                        wr_en => wr_en1,--wr_en1,
                        dout => xk_re12,
                        empty => open,
                        full => open);
----------------------------------------------------------------
fifo2_GI_qd : fifo3
                port map (
                        clk => clk,
                        din => xk_im1,
                        rd_en => dv2,
                        rst => reset,
                        wr_en => wr_en1,-- wr_en1,
                        dout => xk_im12,
                        empty => open,
                        full => open);
----------------------------------------------------------------
mux1_ph: mux1
port map( input1 => xk_re2,
input2 => xk_re12,
    output => xk_re3,
            selec => wr_en2);
-------------------------------------------------------------------
mux1_qd: mux1
port map( input1 => xk_im2,
input2 => xk_im12,
    output => xk_im3,
            selec =>wr_en2);
-------------------------------------------------------------------
DCM_inst : DCM
  generic map (
    CLKDV_DIVIDE => 1.0,
    CLKFX_DIVIDE => 1,
    CLKFX_MULTIPLY => 1,
    CLKIN_DIVIDE_BY_2 => FALSE,
    CLKIN_PERIOD => 0.0,
    CLKOUT_PHASE_SHIFT => "NONE",
    CLK_FEEDBACK => "1X",
    DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS
    DFS_FREQUENCY_MODE => "LOW",
    DLL_FREQUENCY_MODE => "LOW",
    DUTY_CYCLE_CORRECTION => TRUE,
    FACTORY_JF => X"C080",
    PHASE_SHIFT => 0,
    STARTUP_WAIT => FALSE)
  port map (
    CLK0 => open,
```

```
    CLK180 => open,
    CLK270 => open,
    CLK2X => clkdv1,
    CLK2X180 => open,
    CLK90 => open,
    CLKDV => open,
    CLKFX => open,
    CLKFX180 => open,
    LOCKED => open,
    PSDONE => open,
    STATUS => open,
    CLKFB => open,
    CLKIN => clk,
    PSCLK => '0',
    PSEN => '0',
    PSINCDEC => '0',
    RST => reset
  );
--------------------------------------------------------------------
DCM_inst1 : DCM
  generic map (
    CLKDV_DIVIDE => 1.0,

    CLKFX_DIVIDE => 1,
    CLKFX_MULTIPLY => 1,
    CLKIN_DIVIDE_BY_2 => FALSE,
    CLKIN_PERIOD => 0.0,
    CLKOUT_PHASE_SHIFT => "NONE",
    CLK_FEEDBACK => "1X",
    DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",
    DFS_FREQUENCY_MODE => "LOW",
    DLL_FREQUENCY_MODE => "LOW",
    DUTY_CYCLE_CORRECTION => TRUE,
    FACTORY_JF => X"C080",
    PHASE_SHIFT => 0,
    STARTUP_WAIT => FALSE)
  port map (
    CLK0 => open,
    CLK180 => open,
    CLK270 => open,
    CLK2X => clkdv2,
    CLK2X180 => open,
    CLK90 => open,
    CLKDV => open,
    CLKFX => open,
    CLKFX180 => open,
    LOCKED => open,
    PSDONE => open,
    STATUS => open,
    CLKFB => open,
    CLKIN => clkdv1,
    PSCLK => '0',
    PSEN => '0',
    PSINCDEC => '0',
    RST => reset          );
--------------------------------------------------------------------
theta_1: theta
port map ( clk => clkdv2,
```

```
reset=> reset,
theta_out=> theta_out);
-------------------------------------------------------------------------
IF_gen : cos_sin
                port map (
                        THETA => theta_out,
                        SINE => SINE,
                        COSINE => COSINE);
-------------------------------------------------------------------
xk_re <= xk_re3 * cosine;
xk_im <= xk_im3 * sine;
-------------------------------------------------------------------
end Behavioral;
```

Programme mapping

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mapping is
port ( DATA : in std_logic_vector (5 downto 0);
clk : in std_logic;
selec:in std_logic_vector (1 downto 0);
--w_fifo: out std_logic:= '0';
Out_ph1, Out_qd1 : out std_logic_vector (15 downto 0) );
end mapping;

architecture Behavioral of mapping is
signal out_ph, out_qd : integer range -7 to 7;
-----------------------------------------------------------------
component qpsk4 is
port (ADDR: in std_logic_vector (1 downto 0);
--clk : in std_logic;
--selec:in std_logic_vector (1 downto 0);
Out_ph, Out_qd : out integer range -1 to 1);--STD_LOGIC_VECTOR (3 downto 0));
end component;
-----------------------------------------------------------
component qam_16 is
port (ADDR: in std_logic_vector (3 downto 0);
--clk : in std_logic;
--selec:in std_logic_vector (1 downto 0);
Out_ph, Out_qd : out integer range -3 to 3);
end component;
-------------------------------------------------------------
component qam_64 is
port (ADDR: in std_logic_vector (5 downto 0);
--clk : in std_logic;
--selec:in std_logic_vector (1 downto 0);
Out_ph, Out_qd : out integer range -7 to 7);
end component;
-----------------------------------------------------------
component bpsk is
port (ADDR: in std_logic;
--clk : in std_logic;
```

```
--selec:in std_logic_vector (1 downto 0);
Out_ph, Out_qd : out integer range -1 to 1);
end component;
-----------------------------------------------------------
signal Out_ph_1,Out_qd_1,Out_ph_2,Out_qd_2,Out_ph_3,Out_qd_3,Out_ph_4,Out_qd_4 :integer range -7 to 7 :=0;
begin
-------------------------------------------------------------------
qpsk_1 : qpsk4 port map (ADDR => DATA(5 downto 4),
--clk => clk,
--selec => selec,
Out_ph => Out_ph_1,
Out_qd => Out_qd_1);
-------------------------------------------------------------------
qam16_1 : qam_16 port map (ADDR => DATA(5 downto 2),
--clk => clk,
--selec => selec,
Out_ph => Out_ph_2,
Out_qd => Out_qd_2);
---------------------------------------------------------------------------
qam64_1 : qam_64 port map (ADDR => DATA,
--clk => clk,
--selec => selec,
Out_ph => Out_ph_3,
Out_qd => Out_qd_3);
-----------------------------------------------------------
bpsk_1 : bpsk port map (ADDR => DATA(5),
--clk => clk,
--selec => selec,
Out_ph => Out_ph_4,
Out_qd => Out_qd_4);
-----------------------------------------------------------
process
begin
wait until clk'event and clk ='1';
case selec is
when "00" => Out_ph <= Out_ph_4; Out_qd <= Out_qd_4;--bpsk
when "01" => Out_ph <= Out_ph_1; Out_qd <= Out_qd_1;-- qpsk
when "10" => Out_ph <= Out_ph_2; Out_qd <= Out_qd_2;--qam16
when others => Out_ph <= Out_ph_3; Out_qd <= out_qd_3;--qam64
end case;
end process;
out_ph1 <= conv_std_logic_vector(out_ph, 16);
out_qd1 <= conv_std_logic_vector(out_qd, 16);
end Behavioral;
```

Programme BPSK

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity bpsk is
port (ADDR: in std_logic;
--clk : in std_logic;
--selec:in std_logic_vector (1 downto 0);
```

```
Out_ph, Out_qd : out integer range -1 to 1);
end bpsk;

architecture Behavioral of bpsk is
subtype ROM_WORD is integer;
type ROM_TABLE is array (0 to 1) of ROM_WORD;
constant ROM: ROM_TABLE := ROM_TABLE'(
ROM_WORD'(-1),
ROM_WORD'(1));
--signal Out_ph1, Out_qd1: integer :=0;
begin
--process
--begin
--wait until clk'event and clk='1';
--if selec = "00" then
Out_ph <= ROM(conv_integer(ADDR)); -- Read from the ROM
Out_qd <= 0;
--else
--Out_ph1 <=0;
--out_qd1 <=0;
--end if;
--end process;
--Out_ph <= Out_ph1;
--Out_qd <= Out_qd1;

end Behavioral;
```

Programme de QPSK

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity qpsk4 is
port (ADDR: in std_logic_vector (1 downto 0);
Out_ph, Out_qd : out integer range -1 to 1);
end qpsk4;
architecture Behavioral of qpsk4 is
subtype ROM_WORD is integer;
type ROM_TABLE is array (0 to 1) of ROM_WORD;
constant ROM: ROM_TABLE := ROM_TABLE'(
ROM_WORD'(-1),
ROM_WORD'(1));
begin
Out_ph <= ROM(conv_integer(ADDR(1))); -- Read from the ROM
Out_qd <= ROM(conv_integer(ADDR(0)));
end Behavioral;
```

Programme 16-QAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity qam_16 is
port (ADDR: in std_logic_vector (3 downto 0);
```

```
Out_ph, Out_qd : out integer range -3 to 3);
end qam_16;

architecture Behavioral of qam_16 is
subtype ROM_WORD is integer;
type ROM_TABLE is array (0 to 3) of ROM_WORD;
constant ROM: ROM_TABLE := ROM_TABLE'(
ROM_WORD'(-3),
ROM_WORD'(-1),
ROM_WORD'(3),
ROM_WORD'(1));
--signal Out_ph1, Out_qd1: integer :=0;
begin
Out_ph <= ROM(conv_integer(ADDR(3 downto 2)));
Out_qd <= ROM(conv_integer(ADDR(1 downto 0)));
end Behavioral;
```

Programme 64-QAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity qam_64 is
port (ADDR: in std_logic_vector (5 downto 0);
Out_ph, Out_qd : out integer range -7 to 7);
end qam_64;

architecture Behavioral of qam_64 is
subtype ROM_WORD is integer;
type ROM_TABLE is array (0 to 7) of ROM_WORD;
constant ROM: ROM_TABLE := ROM_TABLE'(
ROM_WORD'(-7),
ROM_WORD'(-5),
ROM_WORD'(-1),
ROM_WORD'(-3),
ROM_WORD'(7),
ROM_WORD'(5),
ROM_WORD'(1),
ROM_WORD'(3));
begin
Out_ph <= ROM(conv_integer(ADDR(5 downto 3)));
Out_qd <= ROM(conv_integer(ADDR(2 downto 0)));
end Behavioral;
```

Programme Pilote

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pilote is
port (clk,ce_pil:in std_logic;
     q_out: out std_logic_vector (15 downto 0):="0111111111111111" );
end pilote;
```

```
architecture Behavioral of pilote is
signal q: std_logic_vector (6 downto 0):="1111110";
begin
process (clk)
begin
if ce_pil ='1' then
q(6)<= q(0) xor q(4);
q(0) <= q(1);
q(1) <= q(2);
q(2) <= q(3);
q(3) <= q(4);
q(4) <= q(5);
q(5) <= q(6);
end if;
if q(0) = '1' then
q_out <= "0111111111111111";
else q_out <= "1000000000000001";
end if;
end process;
end Behavioral;
```

Programme Controleur

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity control is
port ( clk, reset, ce :in  std_logic;
pil, rd_en : out std_logic := '0';
       count_out : out std_logic_vector (1 downto 0));
end control;
architecture Behavioral of control is
signal count : std_logic_vector (6 downto 0):= "0000000";
begin

process (clk)
begin
  if clk='1' and clk'event then
    if reset='1' then
      count <= (others => '0');
                              elsif ce='1' and count /="1000000" then
      count <= count + 1;
                              elsif count = "1000000" and ce /= '0' then
                              count <= "0000001";
    end if;
                    end if;
case count is
    when "0000001" => rd_en <= '0';pil <= '0';count_out <= "00";
    when "0011010" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0011011" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0011100" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0011101" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0011110" => rd_en <= '0';pil <= '0';count_out <= "00";
```

```
                    when "0011111" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0100000" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0100001" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0100010" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0100011" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0100100" => rd_en <= '0';pil <= '0';count_out <= "00";
                    when "0001000" => rd_en <= '0';pil <= '1'; count_out <="01";
                    when "0010110" => rd_en <= '0';pil <= '1';count_out <= "01";
                    when "0111010" => rd_en <= '0';pil <= '1';count_out <= "01";
    when "0101100" => rd_en <= '0';pil <= '1';count_out <= "01";            when others => rd_en <= '1';pil <=
'0';count_out <= "11";
  end case;
end process;
end Behavioral;
```

Programme Multiplexeur1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux is
port ( input1,input2, input3:in std_logic_vector (15 downto 0);
    selec1: in std_logic_vector (1 downto 0);
                    clk: in std_logic;
                    output1 : out std_logic_vector (15 downto 0));
end mux;

architecture Behavioral of mux is
signal output: std_logic_vector (15 downto 0);
begin
process (clk,selec1,input1,input2,input3)
begin
if clk'event and clk='1' then
  case selec1 is
    when "00" => output <= input3;
    when "01" => output <= input2;
    when "10" => output <= input1;
    when "11" => output <= input1;
    when others => output <= input1;
  end case;
          output1 <= output;
          end if;
end process;
 end Behavioral;
```

Programme theta

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity theta is
port ( clk, reset:in  std_logic;
```

```
      theta_out : out std_logic_vector (3 downto 0));
end theta;
architecture Behavioral of theta is
signal count : std_logic_vector(2 downto 0):= "000";
begin

process (clk)
begin
       if reset='1' or count= "110" then
       count <= (others => '0');
                              else
       count <= count + 1;
                              end if;
                  case count is
     when "000" => theta_out <= "0000";
                  when "001" => theta_out <= "0010";
                  when "010" => theta_out <= "0100";
                  when "011" => theta_out <= "0110";
                  when "100" => theta_out <= "1000";
                  when "101" => theta_out <= "0010";
                  when  others => theta_out <= "0100";

  end case;
end process;
end Behavioral;
```

j

Programme cos_sin (IP core)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
Library XilinxCoreLib;
ENTITY cos_sin IS
          port (
          THETA: IN std_logic_VECTOR(3 downto 0);
          SINE: OUT std_logic_VECTOR(15 downto 0);
          COSINE: OUT std_logic_VECTOR(15 downto 0));
END cos_sin;

ARCHITECTURE cos_sin_a OF cos_sin IS
component wrapped_cos_sin
          port (
          THETA: IN std_logic_VECTOR(3 downto 0);
          SINE: OUT std_logic_VECTOR(15 downto 0);
          COSINE: OUT std_logic_VECTOR(15 downto 0));
end component;

          for all : wrapped_cos_sin use entity XilinxCoreLib.C_SIN_COS_V5_0(behavioral)
                    generic map(
                              c_has_clk => 0,
                              c_reg_input => 0,
                              c_has_rdy => 0,
                              c_has_sclr => 0,
                              c_symmetric => 1,
                              c_reg_output => 0,
                              c_has_nd => 0,
                              c_enable_rlocs => 0,
```

```
                        c_has_rfd => 0,
                        c_negative_sine => 1,
                        c_latency => 0,
                        c_pipe_stages => 0,
                        c_has_ce => 0,
                        c_has_aclr => 0,
                        c_outputs_required => 2,
                        c_theta_width => 4,
                        c_mem_type => 0,
                        c_negative_cosine => 1,
                        c_output_width => 16);
-- synopsys translate_on
BEGIN
U0 : wrapped_cos_sin
                port map (
                        THETA => THETA,
                        SINE => SINE,
                        COSINE => COSINE);

END cos_sin_a;
```

Programme FFT/IFFT (IP core)

Programme FIFO (IP core)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
Library XilinxCoreLib;
ENTITY fifo1 IS
        port (
        clk: IN std_logic;
        din: IN std_logic_VECTOR(15 downto 0);
        rd_en: IN std_logic;
        rst: IN std_logic;
        wr_en: IN std_logic;
        dout: OUT std_logic_VECTOR(15 downto 0);
        empty: OUT std_logic;
        full: OUT std_logic);
END fifo1;

ARCHITECTURE fifo1_a OF fifo1 IS
component wrapped_fifo1
        port (
        clk: IN std_logic;
        din: IN std_logic_VECTOR(15 downto 0);
        rd_en: IN std_logic;
        rst: IN std_logic;
        wr_en: IN std_logic;
        dout: OUT std_logic_VECTOR(15 downto 0);
        empty: OUT std_logic;
        full: OUT std_logic);
end component;

        for all : wrapped_fifo1 use entity XilinxCoreLib.fifo_generator_v3_2(behavioral)
                generic map(
                        c_rd_freq => 100,
                        c_wr_response_latency => 1,
                        c_has_srst => 0,
```

```
                c_has_rd_data_count => 0,
                c_din_width => 16,
                c_has_wr_data_count => 0,
                c_implementation_type => 0,
                c_family => "virtex2",
                c_has_wr_rst => 0,
                c_wr_freq => 100,
                c_underflow_low => 0,
                c_has_meminit_file => 0,
                c_has_overflow => 0,
                c_preload_latency => 1,
                c_dout_width => 16,
                c_rd_depth => 128,
                c_default_value => "BlankString",
                c_mif_file_name => "BlankString",
                c_has_underflow => 0,
                c_has_rd_rst => 0,
                c_has_almost_full => 0,
                c_has_rst => 1,
                c_data_count_width => 7,
                c_has_wr_ack => 0,
                c_wr_ack_low => 0,
                c_common_clock => 1,
                c_rd_pntr_width => 7,
                c_has_almost_empty => 0,
                c_rd_data_count_width => 7,
                c_enable_rlocs => 0,
                c_wr_pntr_width => 7,
                c_overflow_low => 0,
                c_prog_empty_type => 0,
                c_optimization_mode => 0,
                c_wr_data_count_width => 7,
                c_preload_regs => 0,
                c_dout_rst_val => "0",
                c_has_data_count => 0,
                c_prog_full_thresh_negate_val => 125,
                c_wr_depth => 128,
                c_prog_empty_thresh_negate_val => 3,
                c_prog_empty_thresh_assert_val => 2,
                c_has_valid => 0,
                c_init_wr_pntr_val => 0,
                c_prog_full_thresh_assert_val => 126,
                c_use_fifo16_flags => 0,
                c_has_backup => 0,
                c_valid_low => 0,
                c_prim_fifo_type => "1kx18",
                c_count_type => 0,
                c_prog_full_type => 0,
                c_memory_type => 1);
BEGIN
U0 : wrapped_fifo1
        port map (
                clk => clk,
                din => din,
                rd_en => rd_en,
```

```
                    rst => rst,
                    wr_en => wr_en,
                    dout => dout,
                    empty => empty,
                    full => full);

END fifo1_a;
```

Bibliographie

[1] Theodore S. Rappaport, "Wireless Communications: Past Events and a Future Perspective". *Virginia Tech. IEEE Communications Magazine. 50th Anniversary Commemorative Issue/May 2002.*

[2] Mr Mohamed Seddik TOUHAMI, Mémoire d'ingeniorat "Evaluation des Systèmes OFDM et Estimation du Décalage Fréquentiel de la Porteuse". *ECOLE NATIONALE POLYTECHNIQUE DEPARTEMENT D'ELECTRONIQUE. Alger, juin 2004.*

[3] Joaquín García Ramírez. These: "FPGA-Based Hardware Implementations of OFDM Modules for IEEE 802 standards". *Département Science Informatique, Institut National d'Astrophysique, Optique et Electronique. Puebla. MEXIC. Septembre, 2005*

[4] Aseem Pandey, "VLSI implementation of OFDM modem". *Article de: Wipro Technologies. Santa Clara, CA 95050, USA, 2002*

[5] K. Fazel et S. Kaiser, "Multi-Carrier and Spread Spectrum Systems" , *Edition John Wiley & Sons Ltd, Chichester, West Sussex PO19 8SQ, England, 2003.*

[6] Van Nee, Richard. "OFDM For Wireless Multimedia Communications". *Artech House, 2000.*

[7] S. B. Weinstein and P. M. Ebert, "Data transmission by Frequency Division Multiplexing using Discrete Fourier Transform" *IEEE Trans. Commun., vol. COM-19, no. 5, Oct. 1971.*

[8] Theodore S. Rappaport, A. Annamalai, R.M. Buehrer, and William H. "Tranter Wireless Communications: Past Events and a Future Perspective..". *Virginia Tech. IEEE Communications Magazine. 50th Anniversary Commemorative Issue/May 2002.*

[9] R. Mosier, R. Clabaugh, "Kineplex: A Bandwidth-Efficient Binary Transmission System" *AIEE Trans, vol. 76, jan. 1958, pp. 723-28.*

[10] Henrik Schulze ."Theory and Applications of OFDM and CDMA". *Edition John Wiley & Sons Ltd, Chichester, West Sussex PO19 8SQ, England 2005.*

[11] Clive "Max" Maxfield. "The Design Warrior's Guide to FPGAs" *Newnes .USA Linacre House, Jordan Hill, Oxford .2004*

[12] Xilinx. "Synthesis and simulation design guide". *Xilinx Inc .USA 2000.*

[13] Xilinx." ISE 8.2 In depth tutorial" *Xilinx Inc .2006*

[14] IEEE. IEEE standard 802.11a supplement. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. High-speed Physical Layer in the 5 GHz Band, *1999.*

[15] Mª J. Canet, F. Vicedo, V.Almenar, J. Valls. "FPGA IMPLEMENTATION OF AN IF TRANSCEIVER FOR OFDM-BASED WLAN" *IEEE 2004 SPIS.*