الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne démocratique et populaire

وزارة التعليم العالي و البحث العلمي
Ministère de l'enseignement supérieur et de la recherche scientifique

جامعة سعد دحلب البليدة
Université SAAD DAHLAB de BLIDA

كلية التكنولوجيا
Faculté de Technologie

قسم الإلكترونيك
Département d'Électronique

Mémoire de Projet de Fin d'Études

présenté par

BADJI Issam

&

GUERRACHE Bouchra

pour l'obtention du diplôme d'un **Master en Électronique** option **Système de vision et Robotique**

Thème

Conception d'un système embarqué autour du processeur MicroBlaze à base d'une carte FPGA

Proposé par : **M GUETATFI Zakaria**

Co promoteur : **Dr MAAMOUN Motassir**

Année Universitaire 2013-2014

Remerciements

Avant tout :

Nous remercions notre Dieu (الله) tout puisant qui, sans lui, rien ne peut être fait.

Nous remercions ***GUETTAFI Zakaria et MAAMOUN Montassir*** *qui nous ont proposé le sujet et pour leur conseils qui ont été utiles durant notre étude.*

Nous remercions nos deux familles respectives pour leur encouragement et leur soutient tout au long de notre cursus d'étude.

Nos vifs remerciements à tous nous enseignants qui ont contribué à notre formation, et tous ceux qui ont contribué de prés ou de loin à la réalisation de ce mémoire.

Enfin, nous tenons à remercier tous ceux qui ont contribué de prés ou de loin à l'élaboration de ce travail.

Table des matières

Liste des tableaux

Liste des figures

Chapitre I Généralité sur les FPGAs

Chapitre II MicroBlaze et environnement EDK

Chapitre III Implantation des applications

CONCLUSION GÉNÉRALE

Dans ce projet nous avons commencé par avoir un aperçu général sur les systèmes embarqués et leurs champs d'applications, on a vu qu'il existait plusieurs approches pour les créer, on a pu établir un tableau récapitulatif des différentes approches possible ainsi que leurs avantages et inconvénients.

Le choix a été fait sur les FPGA qui sont reprogrammable ; le plus important c'est la flexibilité qu'offre ces dernières et la possibilité de les paramétrer avec la tâche à accomplir et le matériel disponible dans le système ce qui va engendrer une optimisation qui va se manifester par l'accomplissement de la tâche demandé avec un minimum de ressources possible ce qui est un avantage considérable, et aussi la consommation d'énergie très réduite qui caractérise les circuits FPGAs.

Notre projet s'est effectué sur FPGA nous avons commencé par étudier l'architecture de ces circuits. Le travail s'est fait sur une carte SPARTAN 3AN starter KIT, nous avons vu les périphériques disponibles et surtout les ressources de son processeur XC3S700AN.

La seconde partie du travail a été de passer à la configuration du MicroBlaze pour cela il a fallu bien comprendre les environnements de XILINX tel que XPS et SDK, on a essayé d'expliquer chaque démarche et le passage d'une étape à une autre. Parmi toutes celles qu'on a vues la plus importante est celle du paramétrage du processeur : l'ajout ou la suppression des périphériques. Ce qui permet de mieux comprendre la flexibilité et l'optimisation et de passer des connaissances théorique à la constatation pratique. Une fois le bagage est acquis nous avons commencé la programmation des fonctions qui seront utile pour un éventuel système embarqué.

Ce projet nous a permis de faire notre entrée dans le mode des systèmes embarqués, et plus spécialement ceux à base d'FPGA, d'apprendre comment configurer un MicroBlaze ainsi mieux manipuler les environnements de XILINX et développer des applications en langage C.

I.1 Introduction

Il est fort à parier que les systèmes embarques ne perdrons plus leur place première dans notre quotidien. Simplement parce que leur évolution est la réalisation d'un rêve longtemps attendu !!

Réfléchissez un instant à la journée qui vient de s'écouler. Combien de systèmes embarqués avez-vous touchés ou utilisés aujourd'hui ? Pensez aux appareils électroménagers, aux télévisions, aux tablettes, aux téléphones, à l'éclairage des rues, aux claviers sans fil, aux lecteurs mp3 aux équipements sportifs... On ajoute de l'intelligence à de plus en plus de systèmes chaque jour. Les matériels intelligents de mesure de consommation, de domotique et médicaux à domicile sont quelques exemples qui seront omniprésents dans notre quotidien. Non seulement les systèmes embarqués se multiplient et se diversifient, mais ils deviennent aussi de plus en plus complexes.

L'un des premiers challenges à l'heure actuelle est sans aucun doute la définition d'un modèle de calcul distribue pour les systèmes embarques et connectes en réseau. Le but ultime étant de les faire coopérer pour combiner ou récolter leurs fonctionnalités ou ressources. Leurs nombres et leurs types est si grands, que les modèles distribues traditionnels ne peuvent tout simplement pas être appliques sans poser des pénalités de programmation trop fortes.

L'interopérabilité est un second challenge. La diversité des dispositifs embarques sur laquelle nous sommes dépendants rend notre vie plus complexe, si nous ne pouvons pas faire coopérer silencieusement ces dispositifs pour leur faire échanger des données et des taches.

Nous proposons dans ce premier chapitre d'établir les idées primordiales de ce que sont les systèmes embarqués. Nous allons voir dans un premiers lieu des généralités sur les systèmes embarques Ainsi que les différent familles ou approches pour les réaliser ensuite nous expliquerons les choix qui nous ont conduit a utilisé les FPGA enfin nous allons donner des généralités sur ces derniers et détailler leurs architectures.

I.2 Systèmes embarqués

Un système embarqué peut être défini comme un système électronique conçu pour réaliser une ou un nombre limité de fonctions particulières, souvent avec des contraintes temps réel. Il est intégré à l'intérieur d'un système complet, souvent avec des parties mécaniques et des interfaces avec le monde extérieur (capteurs et actionneurs).

Le terme anglo-saxon est *embedded systems*, qui peut recouvrir en français deux notions distinctes :

– les systèmes embarqués qui sont perçus comme des systèmes mobiles, avec des contraintes temps réel, comme les systèmes informatiques intégrés dans les automobiles, les TGV ou les avions.

– les systèmes enfouis qui sont intégrés dans des systèmes fixes ou dont la mobilité n'est pas liée à des contraintes temps réel.

Si le domaine d'applications des systèmes embarqués est très large, on peut cependant distinguer deux grandes classes :

1/ Les systèmes embarqués orientés contrôle avec les systèmes utilisés dans les transports (automobile, ferroviaire, avionique) et les systèmes de contrôle des gros équipements industriels, comme par exemple le contrôle des centrales nucléaires. Ces systèmes seront appelés « systèmes critiques » ou « systèmes temps réel dur » ;

2/ Les systèmes embarqués orientés calcul ou traitement du signal, avec notamment les systèmes utilisés dans les télécommunications, le multimédia, la télévision numérique, la radio logicielle, etc.

Les systèmes embarqués présentent typiquement les caractéristiques suivantes

- Dédié pour une application spécifique
- Coût réduit
- Espace restreint (volume, capacité mémoire)
- Capacité de calcul appropriée et adaptée
- Exécution temps réel
- Fiabilité et sureté de fonctionnement
- Consommation d'énergie maitrisée, voir très faible en cas d'utilisation sur batterie

La grande majorité des conception de systèmes embarques commencent par un système basé processeur, en utilisant un microcontrôleur ou un microprocesseur comme élément central pour prévoir et traiter les taches élémentaires de contrôle et de surveillance , communiquer avec des interfaces utilisateur et superviser tous les autre aspect de la conception pour les systèmes embarques traditionnels cette architecture fournissait une puissance de traitement suffisante pour réaliser tous les boucles de contrôle et enregistrer des données. Pour des systèmes plus complexe qui intègrent des taches avancées de contrôle et de traitement de signaux les équipes sont obligées d'utiliser des composants de traitement supplémentaires comme des FPGA (field programmable gate arrays) des DSP (digital signal processors et des GPU (graphics processing unis) afin d'atteindre un traitement des données très haute vitesse ainsi qu'un contrôle plus déterministe [1].

Le tableau suivant résume les différentes technologies pour réaliser un système embarqué mettant en évidence les avantages et les inconvénients :

Technologie	Avantages	Contreparties
Microcontrôleurs	Economiques, faible encombrement, simples à programmer	Pas assez de puissance pour les applications haute performance
Microprocesseurs	Vitesses d'horloge élevées pour les applications haute performance, simples à programmer	Consommation élevée, architecture de traitement séquentiel
DSP	Composants dédiés au traitement de signaux, calcul en virgule flottante	Traitement séquentiel par nature
GPU	Moteurs de traitement parallèle pour l'accélération du processeur	Consommation assez élevée, nécessite la présence d'un processeur
FPGA	Matériel flexible via une circuiterie reprogrammable et définie par logiciel, avec traitement parallèle par nature, consommation réduite	Complexité de programmation avec les langages de description matérielle
ASIC	Circuits complètement personnalisés, optimisés en un seul et unique package pour une application unique	Investissements initiaux importants, uniquement possibles en grands volumes

Tableau 1- Les différentes technologies dans les systèmes embarqués

Vu les nombreux avantages qu'offrent les circuits FPGA spécialement la flexibilité et la consommation réduite de l'énergie qui est un facteur très important qu'il faut sérieusement prendre en compte lors de la conception d'un système embarque on préfère travailler avec les circuits FPGA

I.3Définition des PLD's

Un circuit programmable est un assemblage d'opérateurs combinatoires (les opérateurs combinatoires génériques qui interviennent dans les circuits programmables proviennent soit des mémoires (réseaux logiques) soit des fonctions standard (multiplexeurs et OU exclusif) et de bascules dans lequel la fonction réalisée n'est pas fixée lors de la fabrication. Il contient potentiellement la possibilité de réaliser toute une classe de fonctions, plus ou moins large suivant son architecture. La programmation du circuit consiste à définir une fonction parmi toutes celles qui sont potentiellement réalisables. Comme dans toute réalisation en logique câblée, une fonction logique est définie par les interconnexions entre des opérateurs combinatoires et des bascules, et par les équations des opérateurs combinatoires. Ce qui est programmable dans un circuit concerne donc les interconnexions et les opérateurs combinatoires [2].

I.3.1 Structure d'un PLD

La plupart des circuits PLD's suivent la structure suivante:

1- Un bloc d'entrées qui permet de fournir au bloc combinatoire l'état de chaque entrée et de son complément.
2- Un ensemble d'opérateur « ET » sur lesquels viennent se connecter les variables d'entrées et leurs compléments.
3- Un ensemble d'opérateurs « OU » sur lesquels les sorties des opérateurs «ET » sont connectées.
4- Un bloc de sorties.
5- Un bloc d'entrées sorties.
6- Un bloc d'entrées/sorties, qui comporte une porte à 3 états et une broche d'entrée/sortie.

Le bloc combinatoire programmable est formé de matrices « ET » et de matrices « OU » que l'on appelle aussi somme de produits (toute fonction logique combinatoire peut être écrite comme somme de produit) , les interconnexions de ces matrices doivent être programmables, ceci est réalisé par des fusibles qui sont grillés lors de la programmation.

L'autre approche, radicalement opposée, « cellule universelles interconnectées » est de renoncer à la réduction en première forme normale des équations logiques. On divise le circuit en blocs logiques indépendants, interconnectés par des chemins de routage. Une fonction logique est récursivement décomposée en opérateurs simples, jusqu'à ce que les opérations élémentaires rentrent dans une cellule.

Le bloc de sortie est souvent appelé macro cellule, OLMC1 (Output logic macro cell, macro cellule logique de sortie). La macro cellule fut l'élément clé dans le développement des circuits logiques programmables. En effet, la macro cellule procure au composant la flexibilité de configuration : entré, sortie, entré/sortie ou haute impédance. Plus le composant est performant plus celui-ci présente des options sur sa macro cellule. Celle-ci comporte :

- Une porte « OU » exclusif, une bascule D.
- Des multiplexeurs, qui permettent de définir différentes configurations et un dispositif de re-bouclage sur la matrice « ET ».
- Des fusibles de configuration (pour les FPGAs, il est utilisé des cellules de commande des ponts de connexions).

Placement et routage : consiste à attacher des blocs de calcul aux opérateurs logiques d'une fonction et à choisir les broches d'entrées/sorties. Le routage consiste à créer les interconnexions nécessaires. Pour les PLDs simples, le placement est relativement trivial et le routage inexistant. Les compilateurs génériques (i.e. indépendants du fondeur) effectuent très bien ces deux opérations. Pour les CPLDs, et plus encore les FPGAs, ces deux opérations deviennent plus complexes et nécessitent un outil spécifique du fondeur [3].

I.3.2 Différentes familles de PLD's :

La classification des PLDs peut se révéler délicate et difficile, les différences de technologie se doublent de différences d'architectures. La classification suivante n'a que pour objectif de mettre en lumière de grands points de repère2. Néanmoins, on peut les classer suivant leurs structures internes à savoir : le nombre d'entrées, de sorties, de connexions programmables et le niveau d'intégration [4].

Type	Nombre de porte intégré	Matrice ET	Matrice OU	Effaçable
PAL	**10 à 100**	**Programmable**	**Fixe**	**Non**
GAL	**10 à 100**	**Programmable**	**Fixe**	**Electriquement**
EPLD	**100 à 3000**	**Programmable**	**Fixe**	**Par UV**
FPLA	**2000 à 3000**	**Programmable**	**Programmable**	**Electriquement**
FPGA	**Plus de 50 000**	**Programmable**	**Programmable**	**Electriquement**

Tableau2- Classification des différents PLD

I.3.3 PAL's (Programmable Array Logique):

Les PALs ont eu un grand succès dès leur première parution sur le marché, ce fut les premiers circuits logiques programmables3.

Un PAL est un composant relativement simple, dérivé des PROM (Programmable Read Only Mémoire, mémoire morte à lecture seule programmable une fois). Les ingénieurs de MMI ont combiné la technologie à fusibles (utilisée pour les mémoires PROM) avec des portes ET, OU pour réaliser des fonctions logiques. La compréhension de la cellule de base d'un PAL suffit car c'est la même qui se répète sur l'étendue de la capacité de celui-ci. La cellule de base se compose d'un buffer d'entrée qui dispose de l'information et de son complément (tous les PALs sans exception disposent d'un certain nombre d'entrées qui aboutissent toutes, sous forme directe et inversé, sur la matrice de fusible de programmation), suivit de la matrice à fusibles puis de portes ET (considérées en entrée) puis suivit de portes OU en sortie.

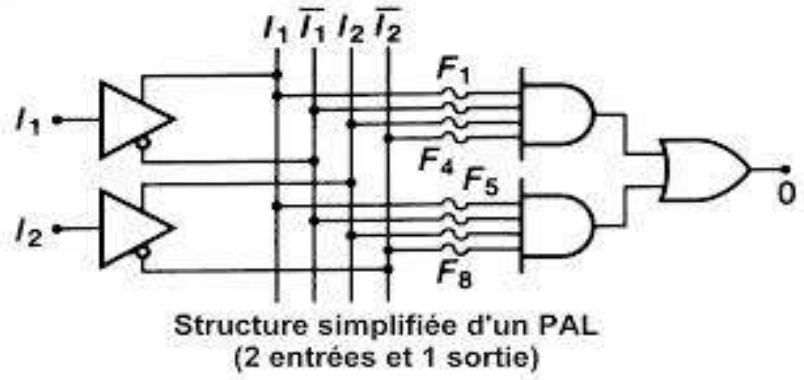


Figure. I.1 Structure d'un PAL

La programmation d'un PAL s'effectue par la destruction de fusibles, à l'aide d'un programmateur dédié en appliquant des tensions de programmation requises.

L'inconvenant majeur des PALs, c'est qu'une fois programmée, ils ne sont plus effaçables (fusible détruit) c'est contraignant en cas d'erreur de programmation ou de mise à jour.

Les PALs existent sous 4 d'architectures (dans l'ordre de leur évolution) :

I.3.3.1 PAL's combinatoires

Possède l'architecture la plus simple. Comme dans tout type de PALs, certaines broches sont dédiées uniquement aux entrées. D'autres bidirectionnelles, sont associés à un buffer de sortie trois états.

I.3.3.2 PAL's à registres

Ils disposent en sortie d'une bascule D. Tout signal de sortie passe obligatoirement par cette bascule. Pour cette raison, certains boîtiers sont proposés avec un certain nombre de sorties à registre (synchrone) et d'autres de type combinatoire.

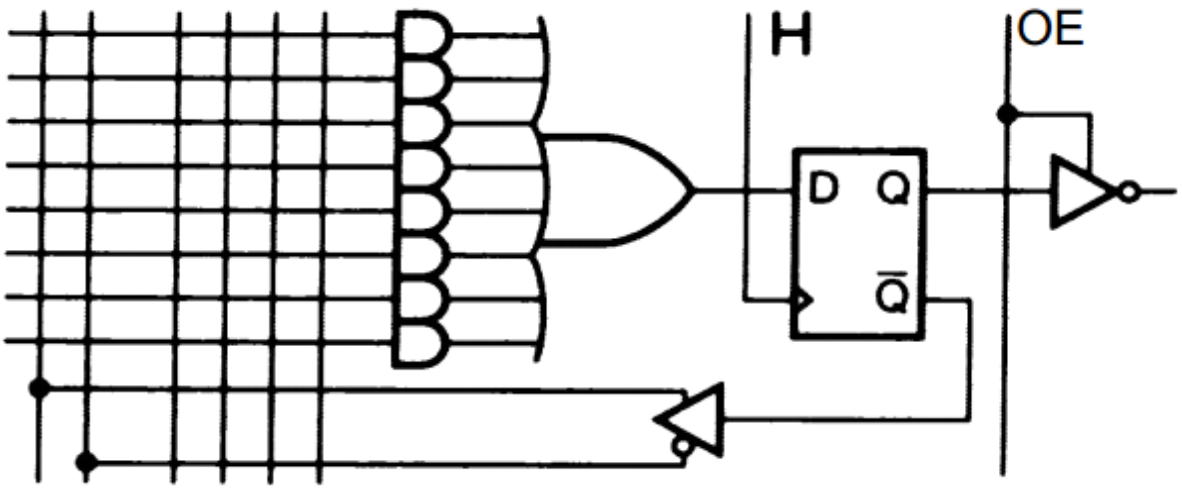


Figure I.2 Structure d'un PAL à registre.

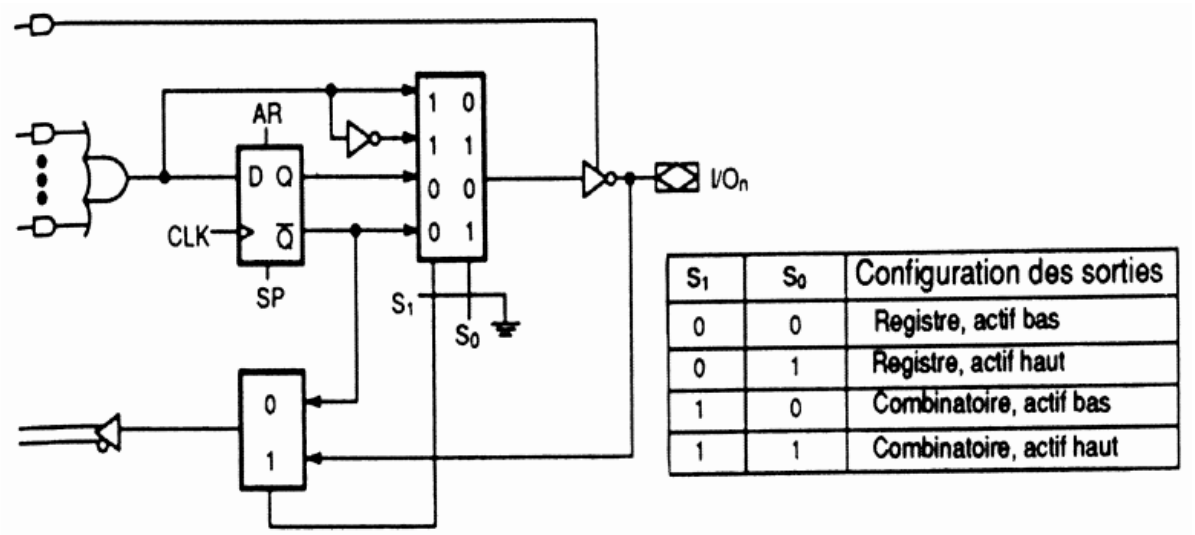
I.3.3.3 PAL's asynchrones à registres

Ce type de PALs constitue une variante du type évoqué précédemment. Une première différence réside dans la méthode de distribution du signal d'horloge des bascules. Contrairement aux PALs à registres, un multiplexeur permet de shunter le registre. Les deux entrées AP (PRESET Asynchrone) et AR

(RESET Asynchrone) sont simultanément utilisées pour piloter ce multiplexeur. A noter la présence d'un XOR se comportant comme un inverseur programmable. Le signal de commande de ce XOR a une valeur statique, il ne s'agit ni d'un signal global ni d'un signal issu de la matrice. C'est à partir de là que le concept de la « macro cellule » programmable a fait son apparition. Cette ressource d'inversion permet décoder une fonction sans tenir compte de la polarité du signal de sortie.

I.3.3.4 PAL's versatiles(VPAL)

Ils constituent une évolution des PALs très significative. Ils proposent des macros cellules très évoluées, du type de cellules rencontrées au sein de composants plus complexes (CPLD, FPGA). Le mode de fonctionnement est obtenu par l'utilisation de deux multiplexeurs qui utilisent comme signal de sélection les points de programmation S0 et S1. En combinaison de ces deux points de programmation, on obtient différentes configurations de la sortie.



S_1	S_0	Configuration des sorties
0	0	Registre, actif bas
0	1	Registre, actif haut
1	0	Combinatoire, actif bas
1	1	Combinatoire, actif haut

Figure I.3 Structure d'un PAL versatile

I.3.3.5GAL's (Generic Array Logic)

Les GALs ne sont rien d'autre (d'un point de vue architectural) que des PALs reprogrammables. D'un point de vue technologique au lieu d'utiliser des transistors bipolaires, ils ont utilisé des transistors MOS FET pouvant être régénérés. Cette possibilité de régénération des fusibles sans pour autant restreindre la durée de vie du composant. Les principaux avantages des GAL's sont :

1. Offrir des produits ayant des vitesses de travail comparable à celle des PALs bipolaires, tout en étant testable a 100%.

2. Permettre un remplacement, au moins fonctionnel, mais idéalement broche pour broche, des PALs bipolaires dans n'importe quelle application.

3. Offrir une consommation beaucoup plus faible que les PALs bipolaires d'une complexité équivalente.

4. Proposer une plus grande souplesse de configuration des entrées/ sorties que les PALs bipolaires.

I.3.3.6 EPLD's(Erasable Programmable Logique Device)

Les EPLDs sont des circuits réalisés en technologie CMOS, présentant l'avantage de faible consommation électrique, mais qui augmente en fonction de la fréquence. Ils disposent d'une macro cellule plus évolué que celles des PALs, en plus ils sont effaçables.

L'introduction des EPLDs telle que l'a voulu ALTERA visait deux buts distincts:

1. Permettre une densité d'intégration nettement supérieur à celle offerte par les PALs et aussi proche que possible que celle permise par les réseaux de portes programmable.

2. Fonctionner a une vitesse, si non égal, du moins comparable à celle des PAL bipolaires et en tout cas nettement supérieur à celle des portes traditionnels.

I.3.3.7 CPLD's:

L'architecture typique d'un CPLD se présente comme un ensemble de fonctions de type PAL pouvant être interconnectées à l'aide d'une matrice. La physionomie est généralement très structurée. Un certain nombre de macros cellules de base sont regroupées pour former des blocs logiques. La complexité, le nombre de macros cellules dans un bloc ainsi le nombre de blocs varie d'un composant à l'autre. On peut considérer deux niveaux d'interconnexion : une matrice globale et un système de distribution des signaux intégrés à chaque bloc logique.

I.4 FPGA (Field Programmable GateArray ou réseau logique programmable sur site)

Un FPGA est un circuit logique reprogrammable. À l'aide de blocs logiques préconstruits et de ressources de routage programmables, c'est un circuit configurable afin de mettre en œuvre des fonctionnalités matérielles personnalisées, sans avoir jamais besoin d'utiliser une maquette ou un fer à souder. Il suffit de développer des tâches de traitement numérique par logiciel et de les compiler sous forme de fichier de configuration ou de flux de bits (bitstream) contenant des informations sur la manière dont les composants doivent être reliés. En outre, les FPGAs sont totalement reconfigurables et peuvent adopter instantanément une nouvelle circuiterie si une nouvelle configuration du circuit est recompilée.

I.4.1 Étude des FPGAs

L'évolution des circuits logiques programmable, depuis la création des PALs qui présentaient l'avantage de réduire l'encombrement et de créer des fonctions logiques personnalisé. Puis vient l'étape des EPLDs qui présentent l'avantage de l'écriture électrique mais en ayant recours à un programmateur (un appareil qui permet d'injecter le routage du circuit via un fichier de programmation) mais effaçable à l'UV (ultraviolet) pour évoluer vers les CPLDs qui sont effaçable électriquement. Puis l'avènement des FPGAs qui représentent une technologie qui permet de reprogrammer le circuit in situ (c'est-à-dire sur circuit ou sur site).
L'avantage majeur que présentent les FPGAs est leur grande flexibilité. En effet, la structure interne du circuit FPGA peut être changée sans avoir à modifier la structure globale de la maquette. Cet avantage est très apprécié par les concepteurs de cartes électroniques vu que ça leurs permet de faire des prototypage rapide et de moindre coût en comparaison aux ASICs pour lesquels il faut des mois pour réaliser un prototype sans avoir de certitude qu'il puisse être opérationnel en plus de ça la moindre erreur nécessite de refaire le travail depuis le début.
Un coût de reviens important et une durée de développement étendu ce qui doit être minimisé dans les milieux industriels. Il y aussi l'avantage de la mise à jour du circuit face à des bugs ou l'ajout de nouvelles fonctionnalités.
Xilinx, Altera et Quicklogic sont les pionniers dans le domaine des FPGAs, et plusieurs autres compagnies produisent les FPGAs. Toutes ces compagnies se partagent le même concept architectural. Il se divise en trois parties : Interfaces d'entrées/Sorties (I/O interface), les blocs logiques de base (Basic Logic Building Blocks) et les interconnexions [5].

I.4.2 Présentation et généralités sur les FPGA's

Un FPGA est un circuit intégré dont la fonctionnalité peut être imprimée dans le matériel grâce à une configuration chargée dans une mémoire.
Cette mémoire, qu'elle soit de type volatile ou non-volatile, peut être programmée directement par l'utilisateur (d'où le terme *Field Programmable)* sans nécessiter un passage coûteux en fonderie.
D'une façon générale, un FPGA peut être vu comme un tableau d'éléments logiques élémentaires (d'où le terme *Gate Array*) dont la fonctionnalité d'un élément et sa connexion avec un sous-ensemble d'éléments, voisins ou distants, dépendent de la configuration

chargée dans le plan mémoire. La figure si dessous donne une vue générique de l'architecture d'un FPGA. On y voit une zone homogène prenant la forme d'une large matrice connectant :

– des **blocs logiques (BL)** ;

– des **blocs d'entrée/sortie (E/S)** ;

– des **blocs de routage (R)** selon des motifs variant selon les familles et les fabricants.

La taille de cette matrice dépasse maintenant facilement les 10^5 BL. À cette architecture très régulière s'ajoutent depuis quelques années différents éléments permettant de rendre le système implanté dans le FPGA beaucoup plus efficace en termes de performance ou d'énergie consommée. On trouve par exemple :

- Des blocs mémoire (RAM) de tailles diverses ;
- Des blocs arithmétiques optimisés pour les principales fonctions du traitement du signal ou de l'image(DSP).
- Des blocs de génération d'horloges programmables par l'utilisateur.
- Des blocs contenant un périphérique spécifique (PERI).
- De plus en plus de processeurs enfouis dans le FPGA (PROC).

Ces derniers peuvent être présents en plusieurs exemplaires dans la puce et permettent de contrôler, de façon très efficace, le système complet. Il est donc maintenant possible de réaliser de véritables systèmes multiprocesseurs au sein de certains gros FPGA.

La reconfiguration matérielle du FPGA a cependant un coût puisqu'elle ajoute de nombreux composants d'interconnexion (R dans la figure I.4) qui implique des pertes importantes en termes de surface et de performances (fréquence d'horloge, consommation d'énergie). On estime par exemple à plus de 90 % de la surface totale celle occupée par les éléments servant aux interconnexions età mémoriser la configuration. De même, les fréquences de fonctionnement typiques des FPGA n'atteignent que quelques centaines de MHz tandis que les microprocesseurs peuvent fonctionner à quelques GHz. Une opération arithmétique implantée en dur dans un microprocesseur peut donc être environ 10 fois plus efficace que sur un FPGA. Cependant, grâce au parallélisme massif disponible dans les FPGA récents, ceux-ci peuvent contrebalancer ces inconvénients au niveau de l'exécution de l'application

complète et dépasse ainsi souvent les performances des microprocesseurs pour de nombreuses applications [6].

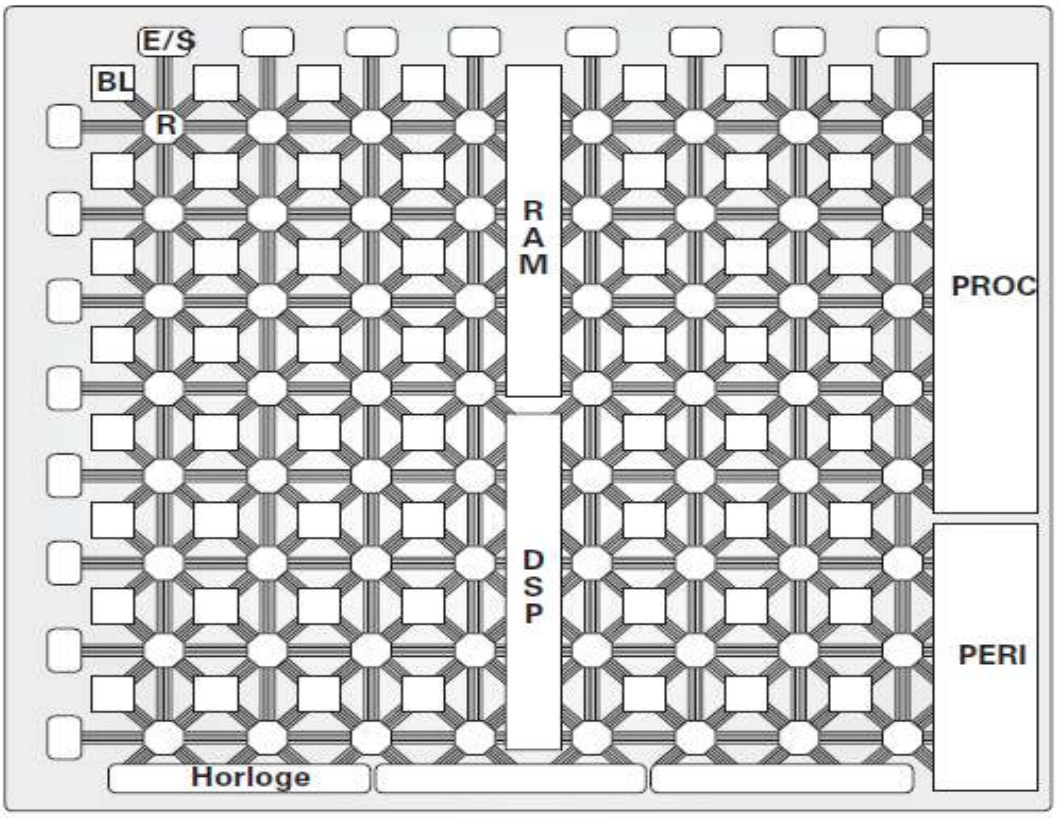


Figure I.4 Vue générique d'un FPGA

I.4.2.1 Blocs logiques configurables

L'élément de calcul de base présent depuis les premiers FPGA est un bloc logique (BL) programmable. La figure **5**montre un exemple représentatif de BL. Celui-ci comprend en général une petite mémoire programmable que l'on appelle une **LUT** (*Look-UpTable)* associée à une **bascule (FF)**. Celle-ci peut être utilisée ou non selon la position (programmable par la configuration) du multiplexeur en sortie de la cellule. La LUT (ici une

LUT à quatre entrées) permet de réaliser n'importe quelle fonction logique à quatre entrées et une sortie.

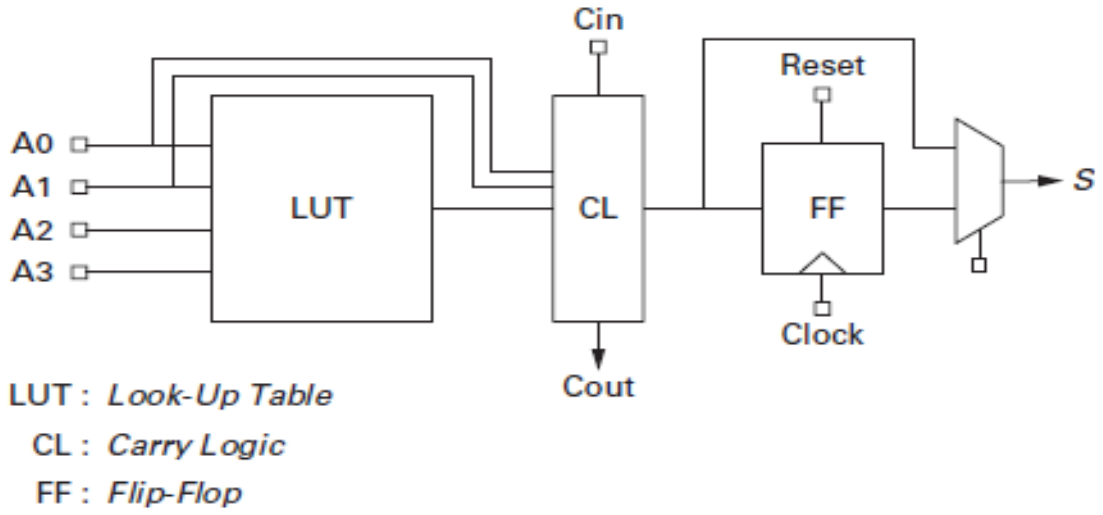


Figure I.5 Vue générique d'un bloc logique (BL) dans un FPGA

Comme les fonctions arithmétiques sont souvent critiques dans les applications ciblées, un chemin spécifique pour la propagation de la retenue (CL) est prévu dans le BL afin de calculer la retenue sortante *Cout* en fonction des entrées *Ai* et de la retenue entrante *Cin*. Cela permet donc de réaliser efficacement des structures d'additions à propagation de la retenue **CPA** (*Carry Propagate Adder)* car celle-ci est propagée directement aux BL voisins sans passer par des blocs de routage généraux, ce qui serait pénalisant en termes de performance. Un additionneur 16 bits à propagation de la retenue peut donc être réalisé à l'aide de 16 BL de façon assez efficace et très régulière dans une colonne de la matrice de BL.
La configuration d'un BL est donc constituée d'un ensemble débits permettant de spécifier :
– la fonction de la LUT ;
– la présence ou non de la bascule ;
– la provenance des signaux de *reset* et d'horloge (*Clock*) ;
– l'utilisation ou non du bloc logique de propagation de la retenue
(CL) ;
– et bien d'autres fonctionnalités pour les BL des FPGA récents.
On voit donc que plus de 20 bits sont nécessaires pour configurer un BL très simple, ce qui représente une taille mémoire importante pour les gros FPGA de plus de 105 BL.

I.4.2.2 Interconnexions programmables

Afin de programmer une fonctionnalité spécifique dans un FPGA, il est nécessaire de connecter les BL entre eux et les BL avec d'autres blocs du FPGA (par exemple, des blocs DSP, des mémoires ou des entrées/sorties du composant). Pour cela, un FPGA s'appuie sur une structure d'interconnexions programmables dont l'objectif est de créer un chemin qui soit le plus court possible entre deux points du FPGA (par exemple, la sortie d'un BL avec l'entrée d'un autre BL). La figure **I.6**représente une structure possible d'un bloc d'interconnexion programmable d'un FPGA.

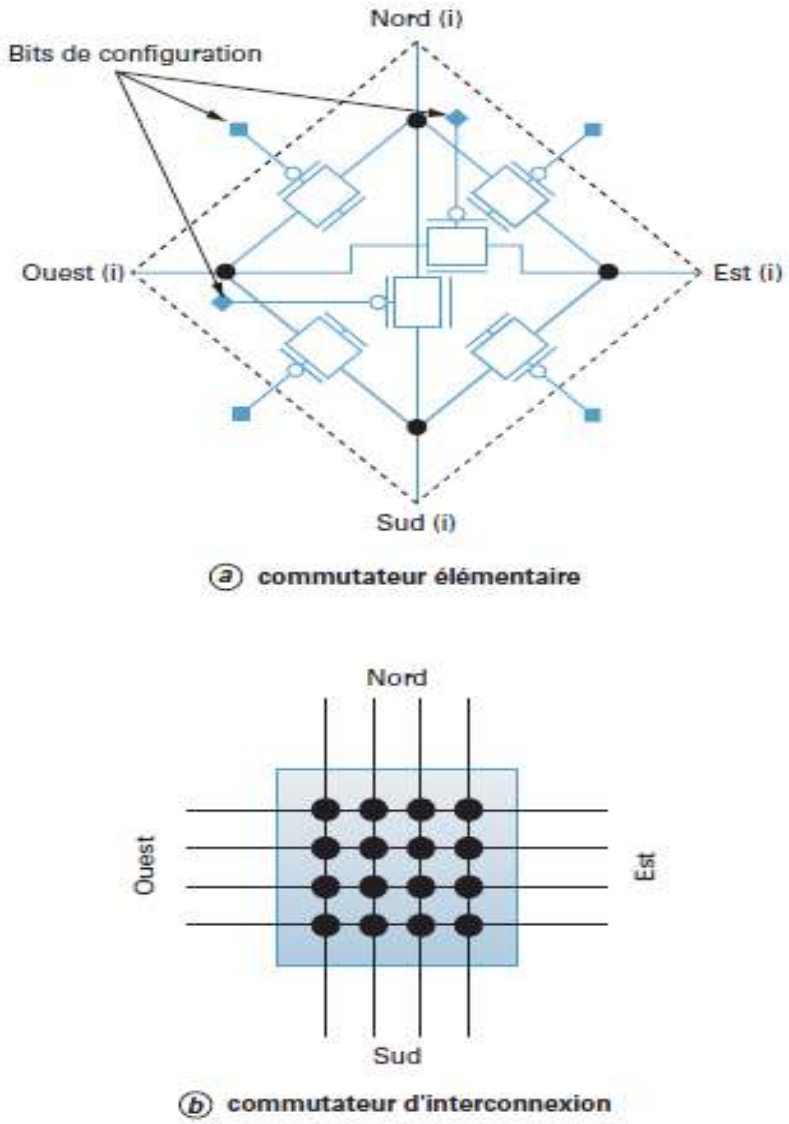


Figure I.6 Bloc d'interconnexion programmable FPGA

La figure I.6(***a)***représente un exemple de structure élémentaire permettant de connecter quatre signaux (notés nord, sud, est et ouest) entre eux. Les losanges sont des bits de configuration qui permettent de configurer le schéma de connexions entre les quatre entrées/sorties. Grâce aux interrupteurs représentés sur la figure par deux transistors NMOS et PMOS en parallèle, il est donc possible de connecter n'importe quel signal à un, deux ou trois de ses voisins (par exemple, le signal issu de l'entrée nord peut être connecté vers les sorties sud et est). Il est important de noter que chaque signal peut être considéré soit comme une entrée, soit comme une sortie. La figure ***b*** prolonge la structure élémentaire (représentée ici par un point noir) afin de connecter quatre blocs de quatre signaux entre eux. Cette structure, appelée **SB** (**Switch Box*)*** dans la littérature, peut prendre diverses formes selon la complexité du routage envisagé. On voit sur cet exemple que 6 bits sont nécessaires pour configurer un commutateur élémentaire et donc 96 bits pour le commutateur 4*4 de la figure ***a***. À nouveau, cela représente une taille mémoire très importante pour les gros FPGA actuels de plus de 105 blocs logiques.

La structure pour les interconnexions programmables dans les circuits FPGA est donc principalement composée de SwitchBox(SB) réparties sur la totalité du circuit. Celles-ci sont interconnectées entre elles et connectées aux différents BL à l'aide de fils métallisés.

Parallèlement à ces lignes, nous trouvons des matrices programmables horizontalement et verticalement entre les divers BL. Le rôle de cette structure d'interconnexions est de relier avec un maximum d'efficacité les blocs logiques et les entrées/sorties afin que le taux d'utilisation dans un circuit donné soit le plus élevé possible. La plupart du temps, cette structure n'est pas homogène mais hiérarchique et contient différents motifs de connexion.

– Tout d'abord, un ensemble limité de BL peut être regroupé au sein d'un **cluster** à l'aide d'un réseau d'interconnexion totalement connecté, ce qui permet ainsi de réaliser des fonctions logiques de type table à un plus grand nombre d'entrées sans utiliser des ressources de routage qui pénalisent les performances.

– Les connexions les plus classiques sont les interconnexions directes entre une SB et une autre SB voisine, un BL voisin ou un bloc d'entrée/sortie, selon une **topologie matricielle** régulière proche de celle de la figure **2**. Grâce à cette structure, il est par exemple possible de connecter la sortie d'un BL à l'entrée d'un autre BL plus ou moins proche en passant à travers un nombre faible de SB.

– Enfin, il existe des lignes métallisées plus longues qui parcourent toute la longueur et la largeur du FPGA et qui permettent de transmettre, avec un temps de propagation plus faible, les signaux entre différents éléments plus éloignés. On peut citer par exemple le routage des horloges, mais aussi les connexions aux mémoires, aux blocs DSP ou aux entrées/sorties. Ces longues lignes, bien que plus performantes, sont en nombre limité et doivent donc être utilisées en priorité pour les signaux critiques par les outils de routage du circuit FPGA.

I.4.2.3 Blocs matériels dédiés

La réalisation d'éléments de mémoire est primordiale dans bon nombre d'applications soit pour stocker temporairement des petites quantités données, soit pour augmenter le débit en utilisant des techniques de pipeline. Dans les FPGA des premières générations, les mémoires pouvaient être réalisées à l'aide de deux éléments : les bascules et les LUT des blocs logiques. L'utilisation massive des **bascules** permet de réaliser des mémoires synchrones tels que des registres, des bancs de registres, ou encore des FIFO (*First-In First-Out)*, tandis que les **LUT** peuvent être configurées en structures de type SRAM. L'utilisation des bascules ou des LUT pose deux problèmes majeurs pour faire des blocs mémoires : leur nombre limité et l'important délai engendré par le passage dans le routage programmable. Ainsi, la plupart des fabricants de FPGA a intégré des petites mémoires câblées directement dans la matrice de blocs logiques de leurs circuits. Ces blocs sont des mémoires très denses en termes de nombres de bits par unité de surface de silicium. Cet ajout de blocs dédiés à la mémorisation permet à la fois d'augmenter significativement les performances et la quantité de mémoire utilisable. Des petits blocs de mémoires, regroupés en colonnes, sont insérés régulièrement dans la matrice. L'architecture du FPGA est alors un peu moins homogène, ce qui peut compliquer un peu la tâche des outils de CAO.

Dans la plupart des cas, ces blocs mémoire sont configurables selon plusieurs tailles (par exemple, des blocs de 1 024*8 bits ou de 256*32 bits pour un bloc RAM de 8 Ko).

La très grande flexibilité permise par les blocs logiques configurables des FPGA se paye en termes de vitesse. En effet, le passage des signaux électriques de l'utilisateur à travers la logique de configuration (même une fois celle-ci configurée) engendre des délais supplémentaires (par rapport à des simples fils de routage dans le cas de circuits ASIC). Très

rapidement, les fabricants de FPGA ont proposé des **blocs dédiés directement câblés « en dur »** pour augmenter les performances de certaines fonctionnalités très souvent utilisées. En premier, ce sont des petits blocs de mémoires (quelques Ko au maximum) qui ont été intégrés, ce qui permet d'éviter l'utilisation de nombreuses LUT des blocs logiques et du routage configurable (assez lent) entre ces blocs. Ensuite, des blocs dédiés pour effectuer des petites multiplications (par exemple, 9*9 bits ou 18*18 bits) ont été intégrés tant cette opération est présente dans de très nombreuses applications. Des blocs spécifiques, et particulièrement performants, pour certains types d'entrées/sorties et de communications sont maintenant courants dans de très nombreux FPGA. Dans certaines familles de FPGA, on a vu aussi arriver des blocs plus spécifiques, mais impossibles à réaliser avec les structures configurables classiques, comme des convertisseurs analogique/numérique.

I.4.2.4 Blocs entrées/sorties IOB et horloges programmables

Les **entrées/sorties(IOB)** sont d'autres éléments importants pour une utilisation efficace des circuits FPGA. Les blocs d'entrée/sortie se doivent d'être très performants et flexibles. Le nombre de ces blocs dépend de la taille du circuit et du type de boîtier utilisé. De nombreuses configurations sont possibles pour s'adapter aux caractéristiques physiques (plages de tensions et de fréquences, adaptation d'impédance, limitations en courant, etc.) et aux caractéristiques logiques (codage, primitives de certains protocoles, etc.). Ici aussi, les FPGA fournissent des solutions très performantes et très facilement utilisables en pratique (par simple configuration). Obtenir les mêmes performances pour ces entrées/sorties dans un circuit ASIC demande, là aussi, une grande expertise technique. Il existe différents types d'entrées/sorties. Un grand nombre sont des blocs assez universels, mais d'autres sont dédiés à certaines fonctions ou dispositifs particuliers. On trouve, par exemple :

- Des blocs spécifiques pour les entrées/sorties à très haut débit (sur des liens sériels) ;
- D'autres dédiés pour les interfaces vers des mémoires externes (DDR, DDR-2, DDR-3, QDR, RLDRAM, etc.) ;
- D'autres pour des interfaces telles que PCI Express ou Ethernet.

- Les **blocs émetteurs/récepteurs dédiés *Transceivers*** (pour la contraction de *Trans-mitters*et de *Re-ceivers*) sont devenus des éléments importants des circuits FPGA récents. Ils sont notamment utiles dès lors que l'on veut interfacer le FPGA avec le

monde « physique », par exemple grâce à des convertisseurs analogique- numérique (ADC) ou numérique-analogique (DAC) rapides.

Les *Transceivers* disponibles dans les dernières générations de FPGA offrent des débits très importants (jusqu'à 28 GHz pour les Stratix V). Leur nombre est un critère de comparaison entre différents FPGA pour beaucoup d'applications. Certaines variantes d'une famille ont été optimisées pour les communications en intégrant plus de *Transceivers*.

- La distribution d'une grande variété de signaux d'**horloges** de bonne qualité est probablement l'un des points les plus importants pour l'utilisateur de FPGA. Dans un circuit ASIC, gérer des horloges avec des hautes fréquences est très complexe, et demande une solide expertise technique. Dans un circuit FPGA, l'utilisateur peut configurer plusieurs horloges avec des fréquences jusqu'à 500 ou 700 MHz pour les meilleurs circuits très simplement, grâce à l'utilisation de structures programmables pour la distribution des signaux d'horloge.

I.5 Conclusion :

Les systèmes embarqués deviennent de plus en plus répandus dans la vie courante, on constate qu'ils prennent de plus en plus de place dans notre quotidien et dans différents domaines.

Dans ce chapitre nous avons fait le point sur ces systèmes tout en détaillant les différentes architectures qui composent ces derniers.

Le plus important était de voir les différentes approches avec lesquelles on peut construire un système embarqué tout en citant les avantages ainsi que les inconvénients ; on a aussi justifié le choix de l' FPGA.

Au final on a expliqué la composition d'un circuit FPGA et son architecture interne.

II.1 Introduction

Dans le chapitre précédant on a acquis des informations sur les systèmes embarqués ainsi que sur les circuits FPGA, on a vu l'architecture et les différentes familles des circuits programmables on a aussi expliqué les raisons de l'utilisation des circuits FPGA pour les systèmes embarqués.

Dans ce chapitre, nous allons voir des généralités sur les microprocesseurs et leurs architectures, on abordera les processeurs hardcore et détailler les processeurs softcore on expliquera par la suite en détails les différents environnements qu'offre XILINX qui permettent la configuration du MICROBLAZE pour finir avec un exemple pratique qui expliquera les différentes étapes.

II.2 Architecture de Von Neuman et de Harvard

Dans l'architecture de la machine de *Von Neuman,* le programme et les données sont enregistrés sur la même mémoire. Chaque instruction contient la commande de l'opération à effectuer et l'adresse de la donnée à utiliser, il faut donc souvent plusieurs cycles d'horloge pour exécuter une instruction. La Figure indique une architecture simple de Von Neuman, constituée d'un bus de données et de programme et d'un bus d'adresses

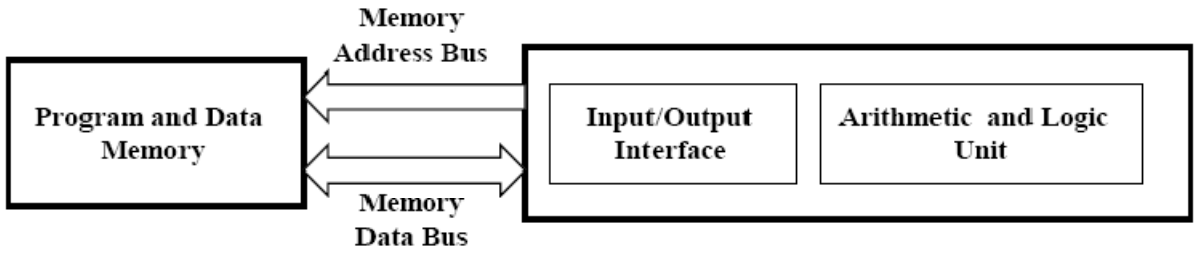


Figure II.1 Architecture de Von Neuman

On voit que les échanges s'effectuent de manière simple entre l'unité arithmétique et logique (ALU), c'est-à-dire l'unité centrale et la mémoire unique, par un bus transitant les codes de programme et les données. On a ainsi des données « collées » aux instructions. Les microprocesseurs et beaucoup de microcontrôleurs utilisent cette architecture car elle est très souple pour la programmation [7].

Dans l'architecture dite *de Harvard* (car mise au point dans cette université américaine en 1930), on sépare systématiquement la mémoire de programme de la mémoire des données : l'adressage de ces mémoires est indépendant. La Figure indique une architecture simple de Harvard, constituée d'un bus de données, d'un bus de programme et de deux bus d'adresse.

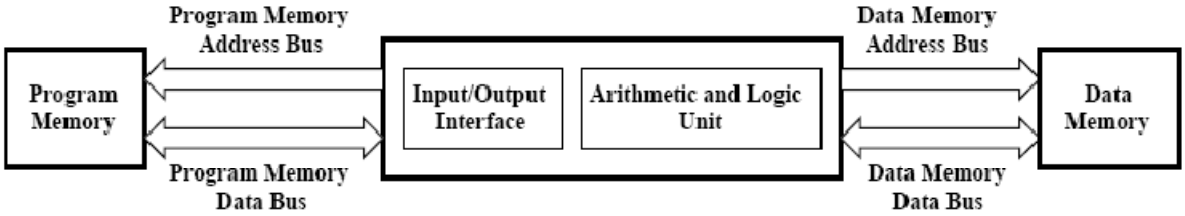


Figure II.2 Architecture de Harvard

On voit que les échanges s'effectuent de manière double entre l'unité centrale et les deux mémoires, ce qui permet une grande souplesse pour l'enregistrement et l'utilisation des données. D'ailleurs, la mémoire de programme est également utilisée en partie comme mémoire de données pour obtenir encore plus de possibilités de traitement avec des algorithmes complexes. L'architecture généralement utilisée par les microprocesseurs est la structure Von Neuman (exemples : la famille Motorola 68XXX, la famille Intel 80X86). L'architecture Harvard est plutôt utilisée dans des microprocesseurs spécialisés pour des applications temps réels, comme les DSP.

II.3 Différence entre architecture de Harvard et d'architecture de Von Neuman

La principale différence entre l'architecture de Harvard et de l'architecture de Von Neuman est dans les données de Von Neumann et l'architecture des programmes sont stockés dans la mémoire même et géré par les mêmes informations système de manutention. Considérant que les données stocke l'architecture de Harvard et de programmes dans les dispositifs de mémoire distincts et ils sont traités par différents sous-systèmes. Il ya bien sûr d'autres différences, mais ce qui précède est la différence la plus profonde qui les différencie en termes de capacités et d'utilisations.

II.4 Processeurs dans les FPGA

Dans la plupart des applications intégrées sur FPGA, il faut un processeur pour la gestion de haut niveau. Deux types de bloc de processeur existent dans les FPGA : les processeurs matériels et les processeurs logiciels.

Les processeurs matériels sont des blocs totalement câblés par le fabricant du FPGA. Ces processeurs offrent des très bonnes performances car ils sont optimisés au niveau transistor. Les fabricants de FPGA utilisent des processeurs connus dans leurs blocs câblés afin de bénéficier des outils de développement et des bases applicatives.

Il semble que les prochaines générations de FPGA vont intégrer des processeurs câblés de plus en plus fréquemment. En effet, des partenariats ont été signés entre la société ARM et certains fabricants de FPGA comme Altera et Xilinx.

Si les blocs de processeurs matériels offrent de très bonnes performances,ils compliquent aussi beaucoup la structure du FPGA.

Ils « brisent » en effet la structure très régulière du FPGA, ce qui augmente le coût de conception et de validation du FPGA (et donc le coût à la vente). De plus, dans bon nombre d'applications, la puissance de calcul d'un processeur évolué, comme un PowerPC, n'est pas nécessaire. Ainsi, très rapidement, les fabricants de FPGA ont proposé des solutions purement logicielles pour faire des blocs de processeur dans leur FPGA.

Les processeurs logiciels SCP, Soft-Core Processors, sont des descriptions de processeurs qui peuvent être totalement réalisées dans la structure configurable du FPGA. Les différents éléments du processeur logiciel sont alors réalisés en utilisant des LUT, des bascules, des blocs mémoires et le routage programmable du FPGA [8].

II.4.1 Processeur MicroBlaze de XILINX

Spécialement pour la conception des SoC, Xilinx a proposé le MicroBlaze, un processeur RISC "soft IP" à 3 étages avec une architecture Harvard et avec 32 registres internes de 32 bits (Figure II.3). Il dispose d'un bus d'instructions et de données internes et externes ((ILMB, DLMB, IOPB et DOPB). Le Processeur MicroBlaze, tout comme le NIOS, est très facilement configurable occupant selon le choix des options de 900 à 2600 éléments logiques et pouvant fonctionner sur une fourchette de fréquences à partir de 80MHz (exemple : en

utilisant une Virtex5-LX50, avec un pipeline à 5 niveaux et sans MMU, le MicroBlaze occupant 1027 LUTs et fonctionne à 235 MHz).

En plus, de nombreux périphériques sont fournis avec le MicroBlaze, afin de constituer un microcontrôleur complet et personnalisable. Il y a, entre autres : contrôleur mémoire (SRAM, Flash), contrôleur mémoire SDRAM, UART lite, Timer/compteur avec fonction PWM, interface SPI, contrôleur d'interruptions, GPIO (entrées-sorties génériques), convertisseurs A/N et N/A Delta-Sigma, DMA, etc..

MicroBlaze peut fonctionner en utilisant plusieurs systèmes d'exploitation (XilinxMicroKernel, uClinux, FreeRTOS, etc.). Ces systèmes d'exploitation sont constitués généralement d'un ensemble de bibliothèques permettant d'obtenir des fonctions basiques telles que : pilotes de périphériques, séquencement de tâches, système de fichiers FAT, pile TCP/IP (avec le logiciel libre lwip), etc [9].

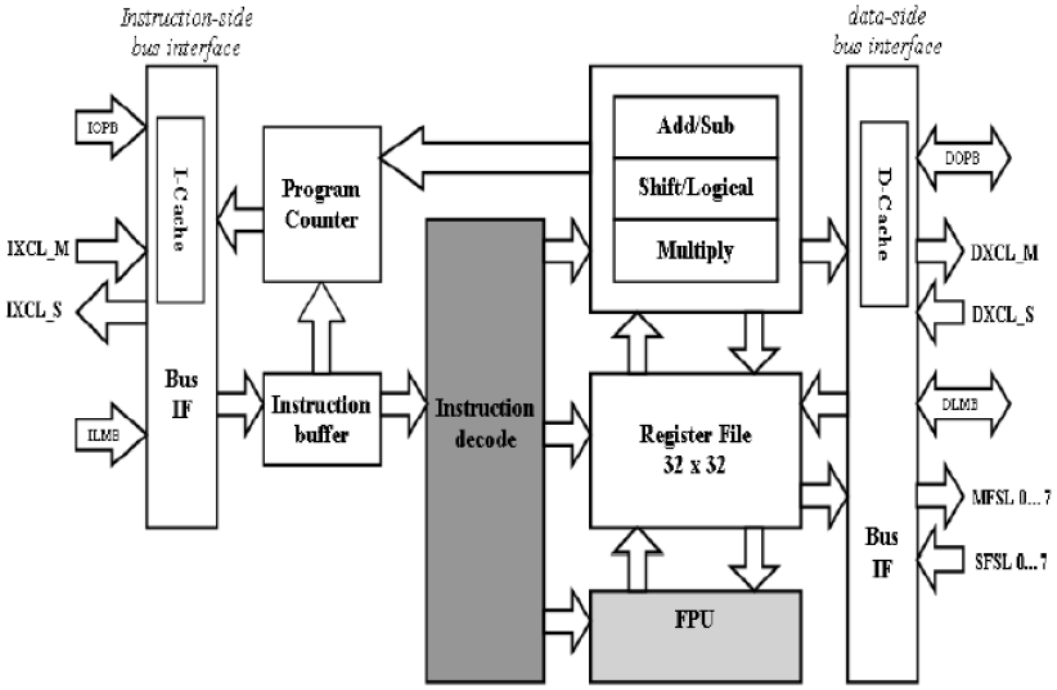


Figure II.3 Architecture interne du MicroBlaze

II.5 Flot de conception

La conception d'un système embarqué se réalise grâce à une série d'outils compris dans la suite logicielle Xilinx ISE Design Suite 12.4. Parmi ceux-ci, deux retiennent notre attention : XPS (Xilinx Platform Studio) et SDK (Software Development Kit). À eux seuls, ils forment la partie EDK (Embedded Development Kit) de la suite et permettent le design complet d'une architecture matérielle munie d'un ou plusieurs processeurs, puis le développement et la compilation d'un logiciel pour ces derniers.

Dans cette partie nous allons voir :

- expliquer la différence entre les outils XPS et SDK;
- expliquer le flot de conception d'un système embarqué avec ces outils;
- expliquer le BSB de XPS et montrer comment l'utiliser afin de créer un système de base
- montrer comment exporter les caractéristiques d'un design matériel terminé vers SDK et couvrir les notions importantes de SDK (Hardware Platform Spécification, Board Support Package, projets C et C++).
- Expliquer comment se fait le debugage et la génération d'un fichier .elf

La conception d'un système embarqué suit un tracé linéaire particulier et inévitable. Le schéma suivant résume celui-ci:

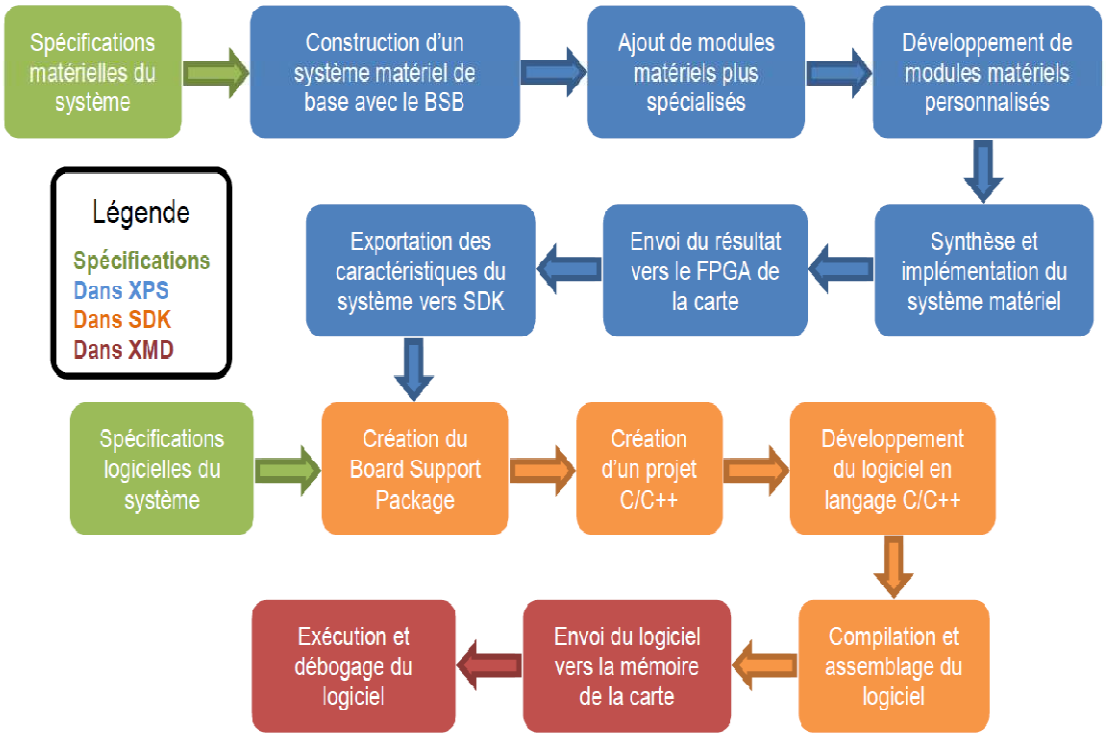


Figure II.4 Schéma de conception d'un système embarqué

Au départ, tout se passe dans XPS. Ce logiciel permet de créer et de paramétrer tout le côté matériel du système. Pensez, par exemple, à une maison : avant de la décorer et de la remplir de meubles, elle doit contenir des planchers, des murs, des entrées et des sorties, de la plomberie, un réseau électrique, etc., lesquels ont tous été déterminés au préalable par l'architecte. Une fois le design matériel terminé et implémenté dans le FPGA de la carte, l'exportation de ses caractéristiques permet de passer à SDK.

SDK : c'est la décoration. Si la maçonnerie d'une maison peut prendre des semaines avant d'être complétées, changer un cadre d'un mur à l'autre ou déplacer un meuble se fait en un tour de main. On parle d'une quinzaine de minutes pour synthétiser et implémenter le design matériel contre une dizaine de secondes pour la compilation du logiciel.

Sans oublier notre analogie, il faut comprendre que modifier un aspect du design matériel demande souvent de recommencer une bonne partie de la synthèse et l'implémentation, tout comme l'ajout d'un étage à une maison demande d'abord une déconstruction partielle.

À la lumière de ce fait, il est important de vérifier tous les paramètres et les modules matériels correctement avant de lancer la synthèse et l'implémentation.

II.5.1 XPS: Xilinx Platform Studio

Tel que mentionné ci-haut, XPS permet la conception matérielle du système embarqué. Celui-ci comprendra la plupart du temps un ou plusieurs processeurs, ainsi que plusieurs modules matériels, nommés *cores*ou *IP cores*, interconnectés par un ou plusieurs bus.

II.5.2 BSB : Base System Builder

Un système matériel dans XPS est décrit à l'aide de plusieurs fichiers de spécifications textuels. Comme leurs formats sont plutôt complexes, surtout en début de cours, il est préférable d'utiliser l'outil BSB dans XPS. Celui-ci présente un guide intelligent muni d'une interface graphique et écrit, selon les options choisies, les fichiers de spécifications automatiquement. Ceux-ci peuvent par la suite être modifiés pour atteindre des objectifs plus précis, mais le BSB à lui seul permet de créer un système matériel de base complet et fonctionnel. Afin d'utiliser le BSB, il suffit de lancer XPS afin de se faire proposer ce choix : En s'assurant que la première option est sélectionnée, cliquer sur « OK » lance le BSB. La fenêtre suivante demande où enregistrer les fichiers du projet et où se trouvent les périphériques additionnels

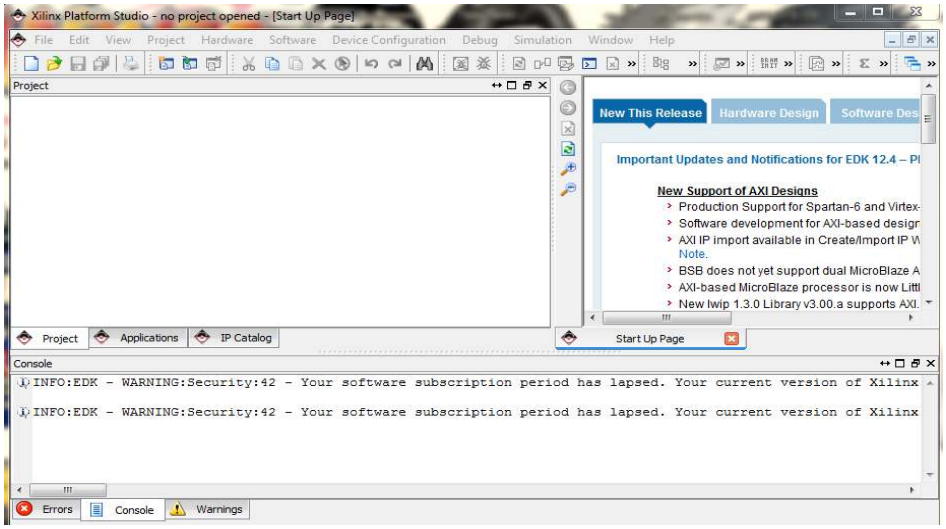


Figure II.5 Capture de la plateforme XPS

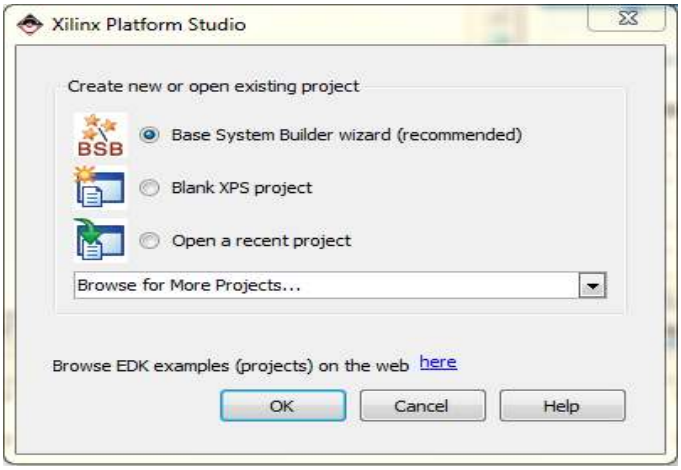


Figure II.6 Capture de la base système builder BSB

Le répertoire de recherche des périphériques additionnels devrait être C:\Xilinx\13.1\ISE_DS\edk_user_repository.

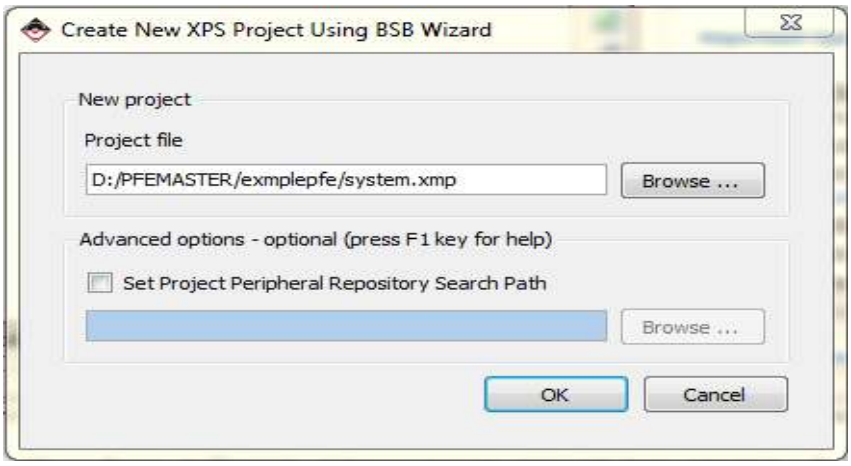


Figure II.7 Création d'un projet XPS

S'en suit alors le choix entre une architecture PLB ou AXI. Le projet utilisera une architecture PLB

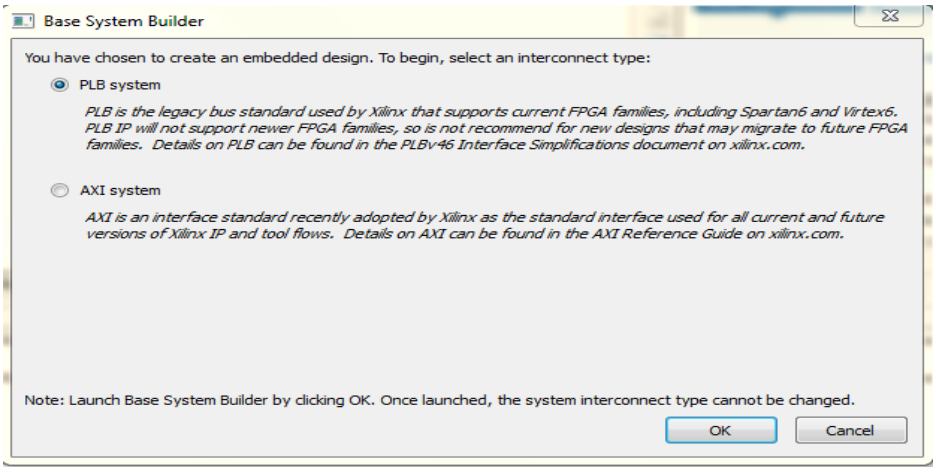


Figure II.8 Le choix de l'architecture.

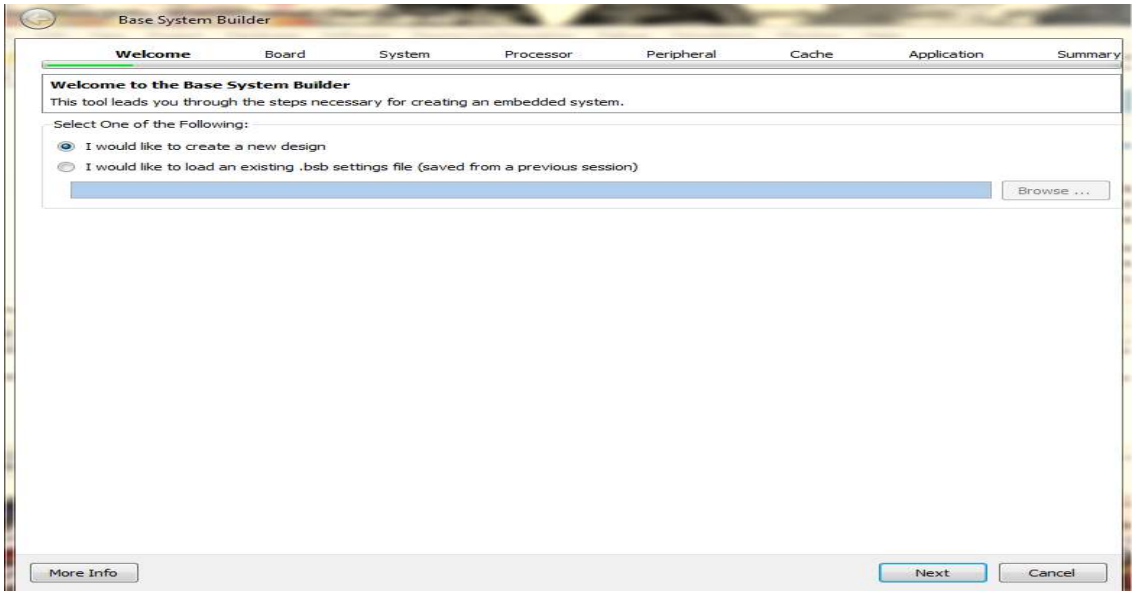


Figure II.9 Début de constitution de l'architecture

Une fois le bouton « OK » appuyé, la première page du BSB est affichée :

Comme nous désirons créer un nouveau design, nous laissons l'option « I wouldlike to create a new design » sélectionnée et appuyons sur « Next> ».

La page suivante donne des informations sur la carte.

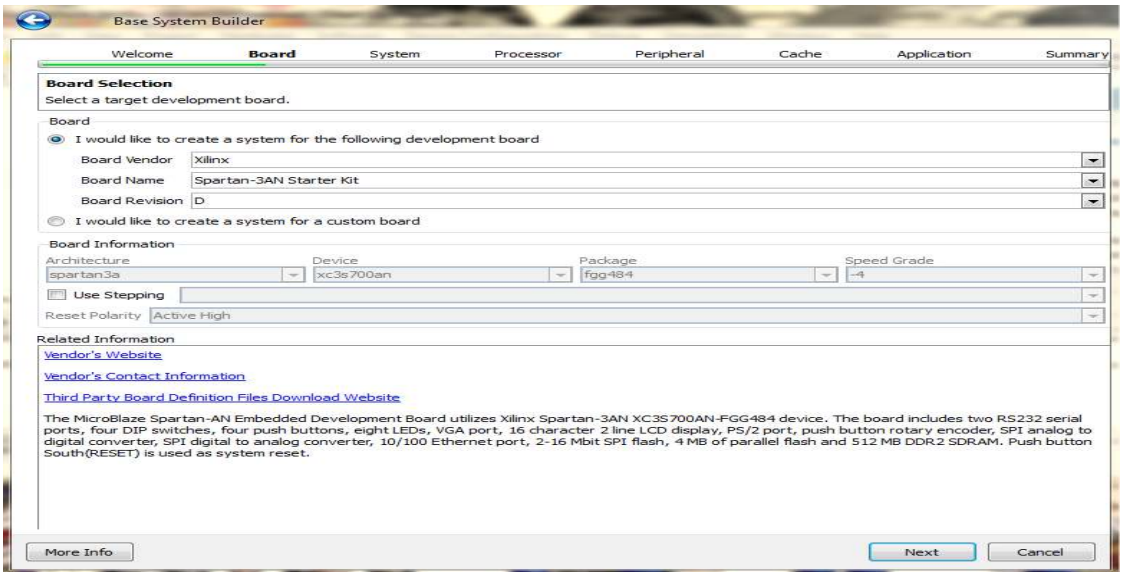


Figure II.10 le choix de la carte FPGA

Nous accédons ensuite à la page où un choix doit être fait entre un système à un ou à plusieurs processeurs. Pour cette introduction simple, nous choisissons un seul processeur :

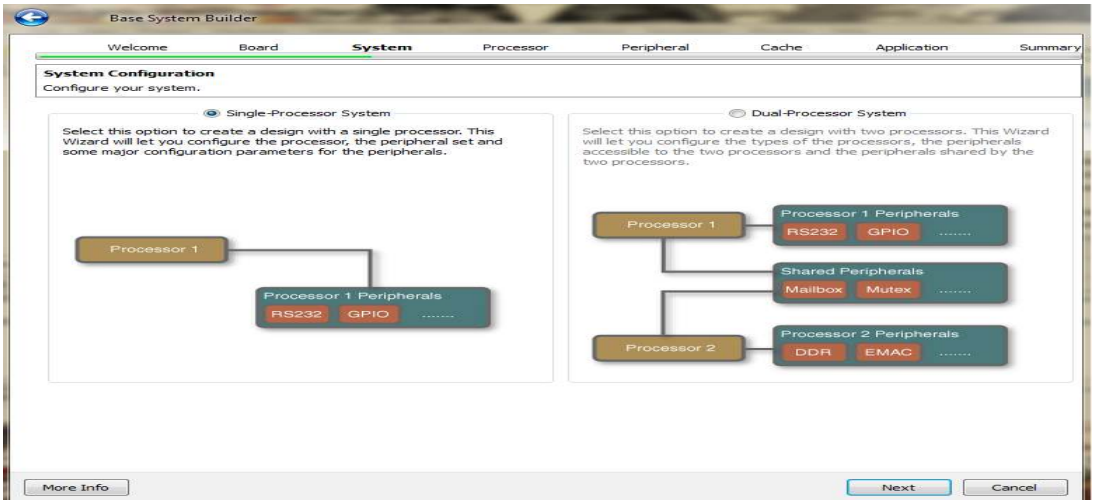


Figure II.11 choix du processeur systeme

Il est ensuite possible de configurer le processeur choisi. Ici, nous désirons que celui-ci soit un MicroBlaze (qui devrait d'ailleurs être le seul choix), avec une fréquence d'horloge de 100 MHz et 64 kiB de mémoire locale (mémoire contenant éventuellement notre logiciel compilé et assemblé) :

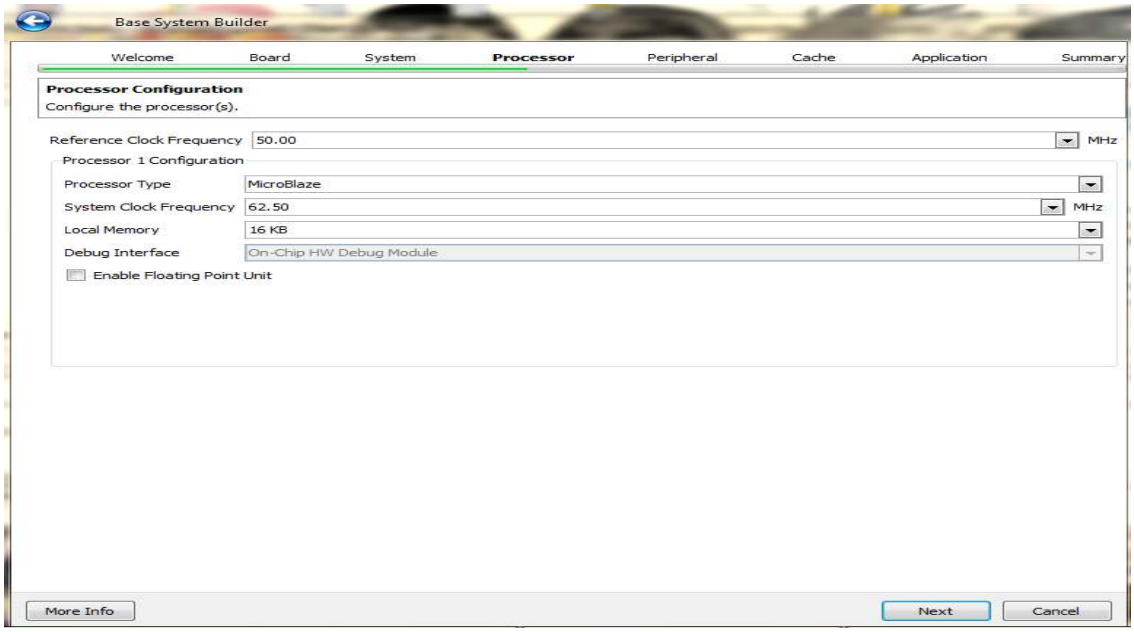


Figure II.12 configuration des caractéristiques de la carte

La page suivante est probablement la plus importante. Elle permet d'inclure ou d'exclure des périphériques au système. Ces périphériques sont liés par un seul et même bus. Ils ne représentent pas la totalité des périphériques, ou *cores*, disponibles, mais plutôt un sous-ensemble déjà configuré pour nous. Dans cette introduction, nous n'allons inclure que les 8 LED, les 8 boutons poussoirs et l'UART (pour communication par RS-232) dans le système :

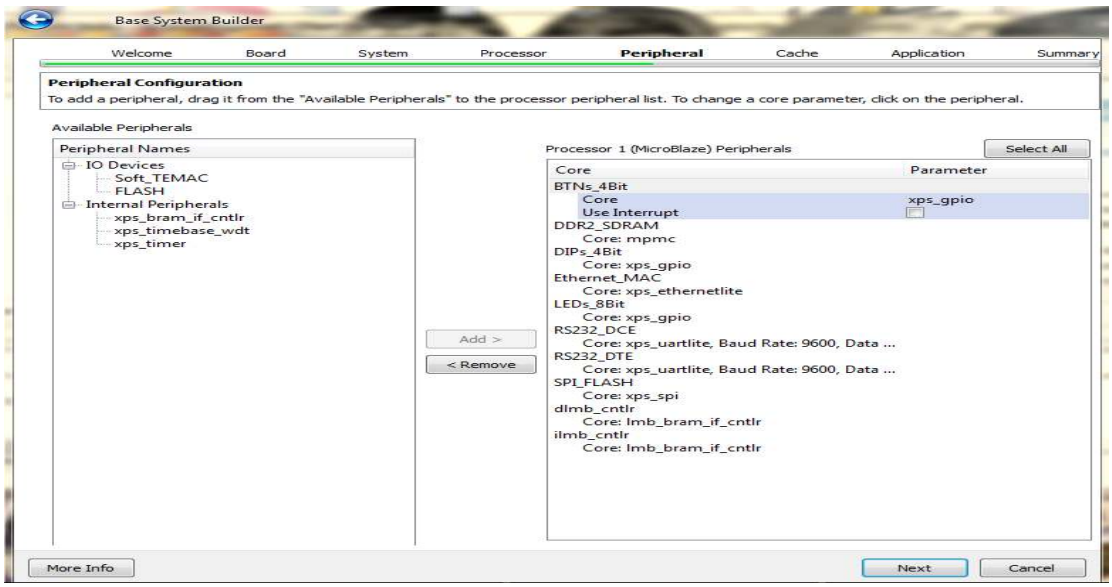


Figure II.13 configuration des périphériques

Le reste devra pour l'instant être retiré grâce au bouton «Remove ». Les étapes suivantes, la cache et le résumé, peuvent être traversées rapidement. Pour la cache, aucune option ne sera disponible puisque nous n'avons pas inclus de mémoire externe dans notre design. Cliquer sur « Finish » termine le BSB, crée le système et ouvre le projet dans XPS :

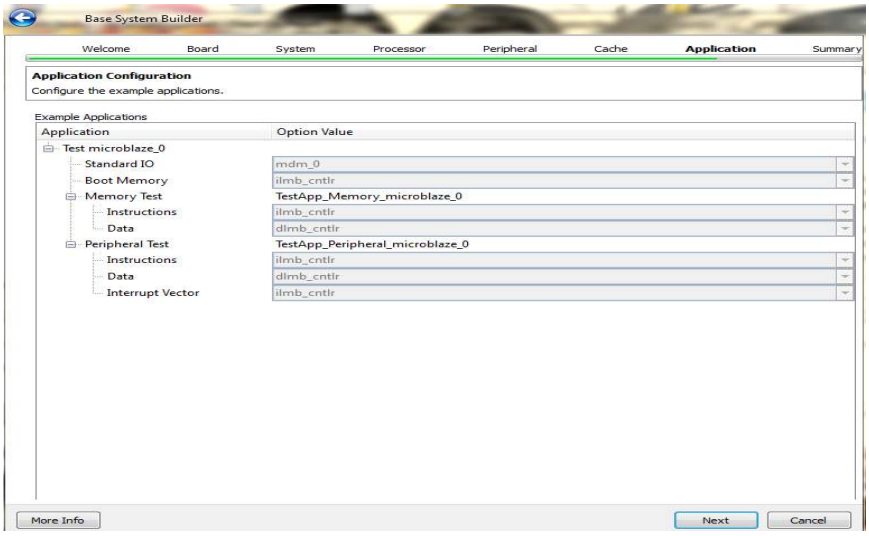


Figure II.14 configuration des mémoires

II.5.3 Interface régulière de XPS

L'interface régulière de XPS permet de modifier les paramètres des périphériques configurés par le BSB, d'en ajouter de nouveaux, de modifier les plages d'adresses des bus et de poursuivre de façon générale le design matériel du système.

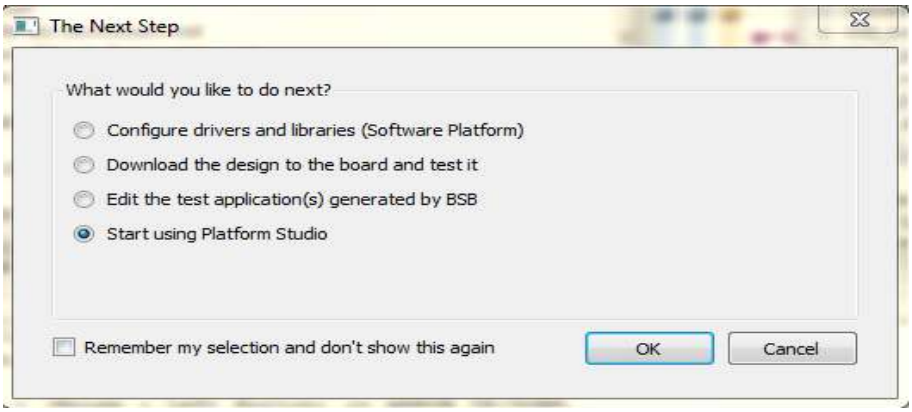


Figure II.15 Début d'utilisation de la plateforme

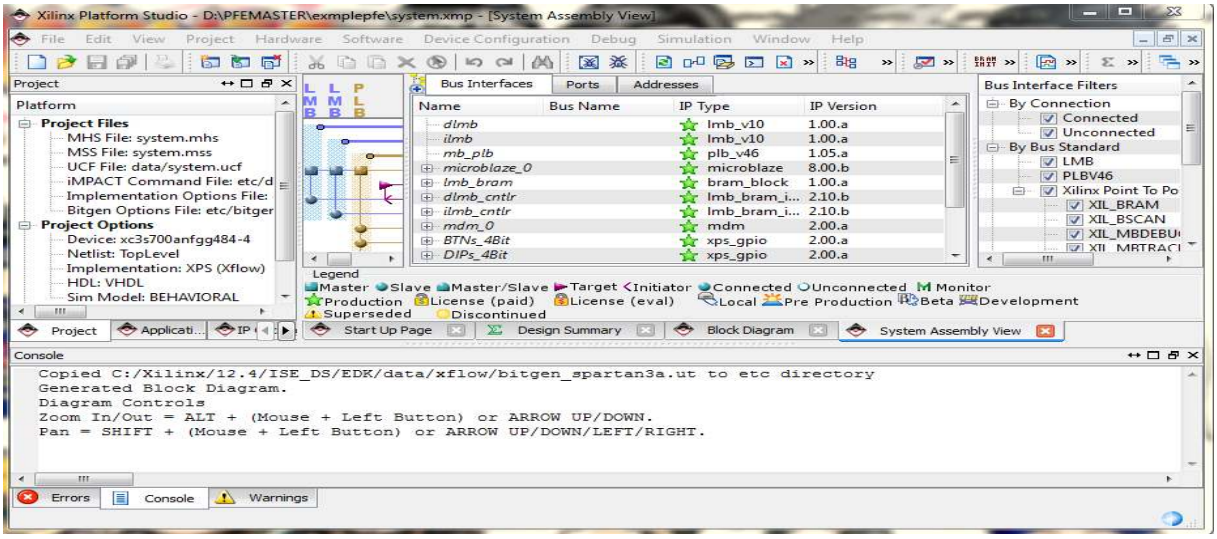


Figure II.16 Capture récapitulative de l'architecture finale

Pour l'instant, nous nous contentons de synthétiser et d'implémenter le système en appuyant sur le bouton « GenrateNetlist» puis par la suite sur le « generate bitstream » montré dans la figure suivante :

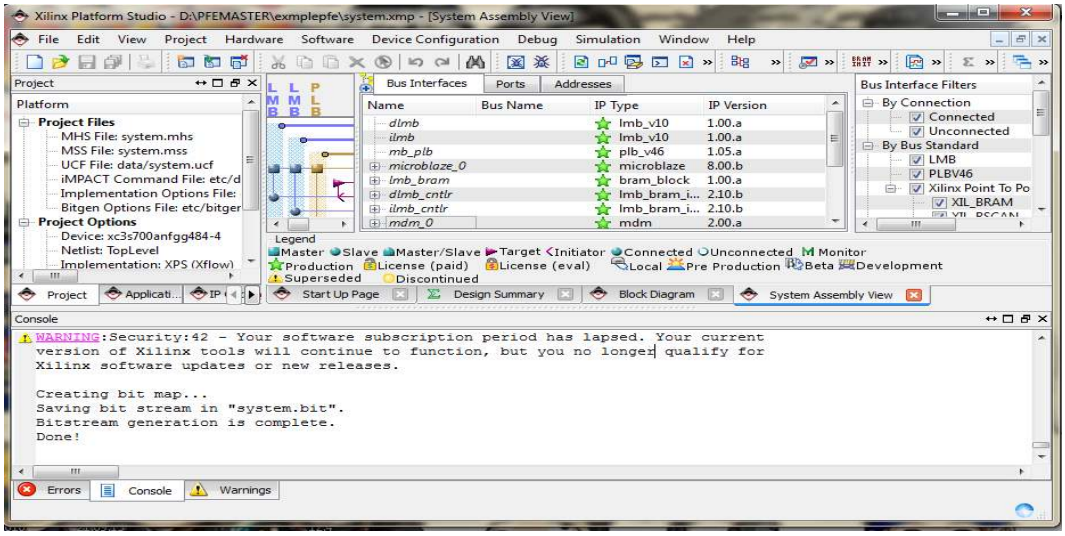


Figure II.17 Génération du « Bitstream » et « Netlist »

Il faudra attendre quelques minutes avant de voir apparaitre la ligne « Done! » dans la console au bas de l'interface.

Que s'est-il alors passé? Le schéma suivant résume de façon générale le processus effectué et ses produits dérivés :

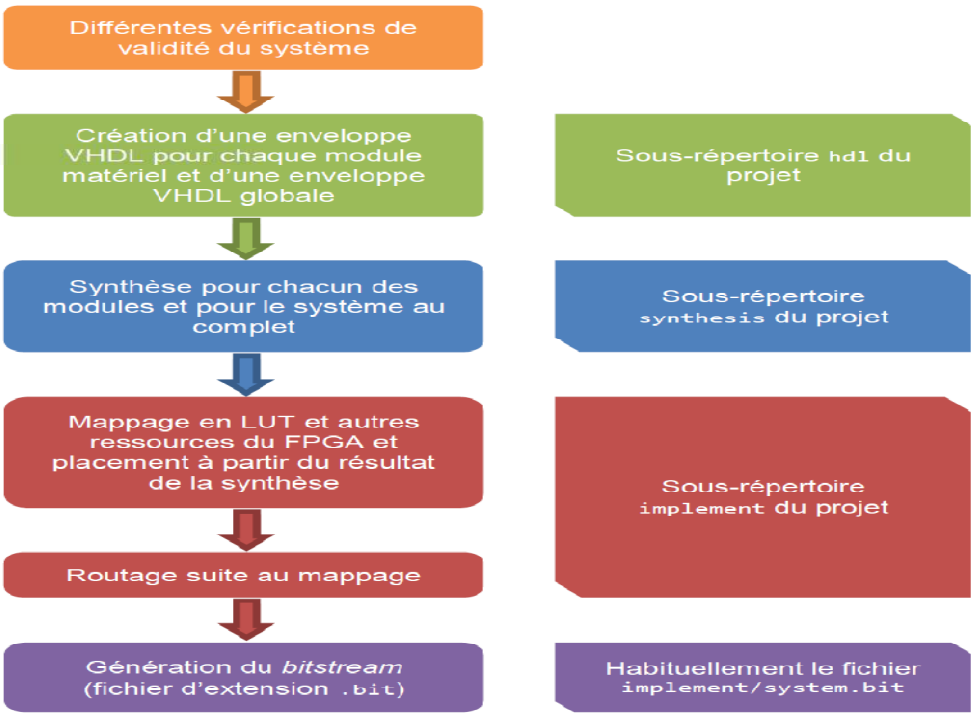


Figure II.18 Les étapes de la généralisation du processus

Suite à la génération du fichier system.bit, le FPGA de la carte de développement peut être programmé avec notre design en appuyant sur le bouton export hardware design to SDK :

Cette action lancera iMPACT et configurera automatiquement le FPGA. Pour passer au développement logiciel, il suffit d'exporter les caractéristiques du système matériel vers un dossier de travail, ou *workspace*, où SDK basera ses projets. Pour ce faire, il faut sélectionner le menu « Project » puis l'item « Export Hardware Design to SDK… ».

Ensuite, le bouton « Export &Launch SDK » peut être appuyé pour lancer automatiquement SDK suite à l'exportation. SDK demandera alors où placer le dossier de travail : il est conseillé de le placer à l'extérieur du répertoire de projet XPS.

II.5.4 SDK: Software Development Kit

La partie matérielle du système est créée, le FPGA est programmé et les caractéristiques du design sont exportées dans un dossier de travail pour SDK : il est maintenant temps de développer un logiciel pouvant s'exécuter sur notre plateforme personnalisée.

Une fois dans SDK, l'onglet « Project Explorer » à gauche ne liste qu'un item nommé « hw_platform_0 ». Il s'agit d'une référence aux caractéristiques de notre design matériel; c'est en fait le produit de l'exportation réalisée précédemment.

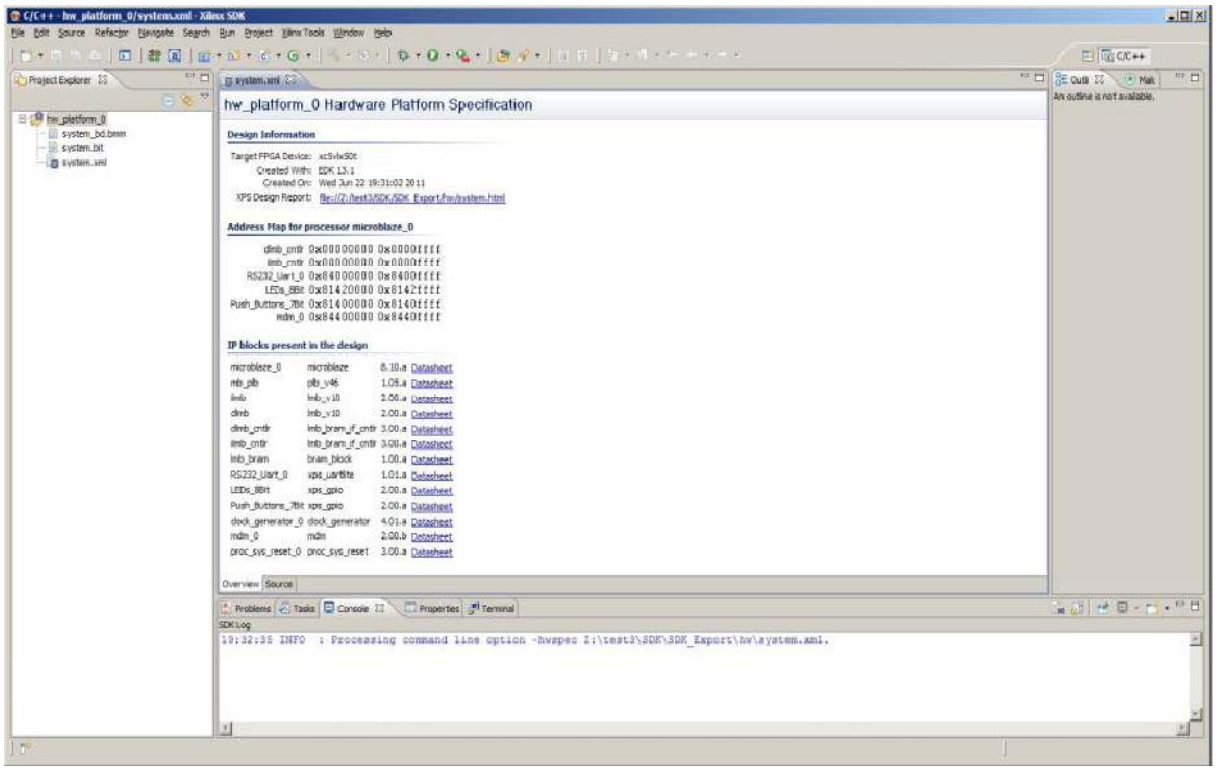


Figure II.19 Plateforme logicielle

II.5.5 Création d'un projet C

Il est maintenant temps de créer un projet C. Via « File ⟶New ⟶Xilinx C Project », une fenêtre présente un choix de modèles pour le nouveau projet; nous choisissons ici un projet vide, « Empty Application ».

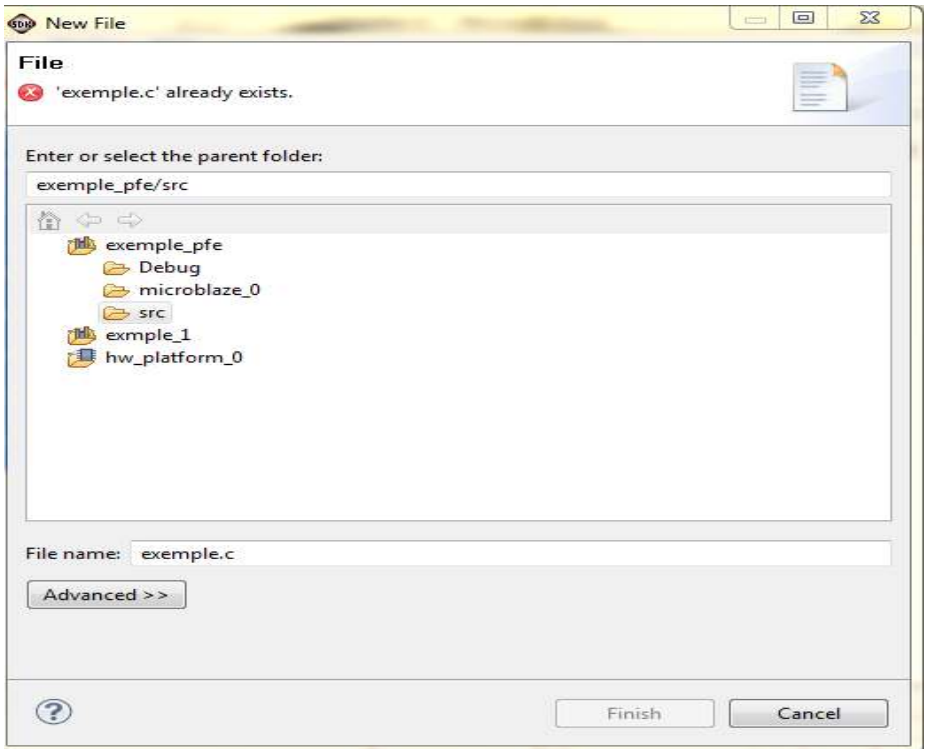


Figure II.20 Création d'un nouveau projet C

En appuyant sur « Next », nous avons l'option de créer un nouveau Board Support Package ou d'en associer un déjà existant au nouveau projet, ce que nous faisons (le seul existant devrait être celui que nous venons de créer) :

Appuyer sur « Finish » termine l'assistant et crée le nouveau projet C.

Nous pouvons désormais créer un nouveau fichier source en cliquant droit sur notre nouveau projet dans l'onglet « Project Explorer », puis en sélectionnant « New → Source file ». Nous le nommons ici main.c, mais le nom du fichier est sans importance. Comme le montre la figure suivante :

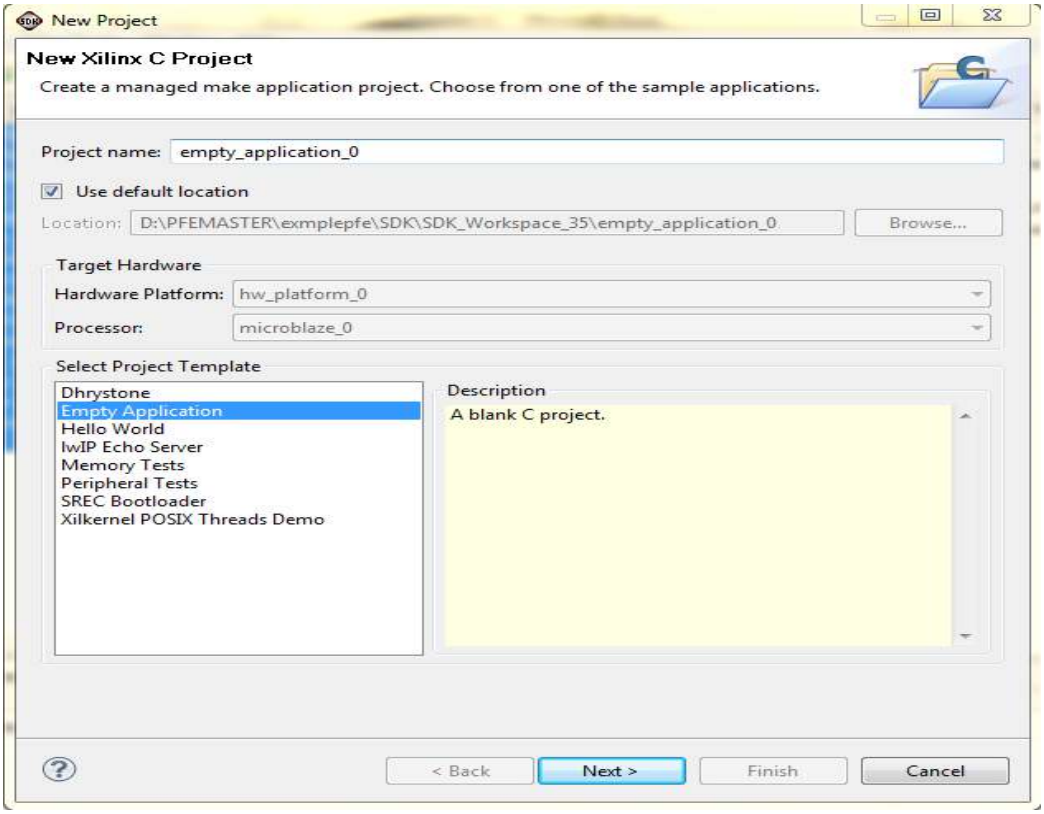


Figure II.21 Création du fichier « .C »

II.5.5.1 Exemple d'application

Suite aux acquits précédemment expliqué on va exposer un exemple pour programmer en C d'un compteur à 8 étages :

Algorithme 1 : **Pseudo-Code Compteur à 8 étages**

1-Charger les Bibliothèques/Drivers du Matériel

2-Définir un Délai LED_DELAY ⟵1000000

3-Définir un Périphérique (Port) gpio_leds

4-Déclaration d'une Valeur Mémoire de 32 bits leds

5-Déclaration d'un Compteur Delay

6-Initialisation du Port gpio_leds

7-Définie la Direction du Port gpio_leds ⟶

8-leds ⟵ 0x01

9- ledsgpio_ ⟶ leds (Envoyer la valeur leds vers le port gpio_leds)

10- While infinie do

11- For Delay = 0 to LED_DELAY do

12- LED_DELAY ⟵ LED_DELAY + 1

13- End for

14- Ledsleds ⟵ + 1

15- Ledsgpio_ ⟶ leds (Envoyer la valeur leds vers le port gpio_leds)

16- End while.

Le programme en C:

```
#include"xparameters.h"
#include"xgpio.h"
#define LED_DELAY 1000000
intmain( void ){
Xuint32leds ;
XGpiogpio_leds ;
volatileint Delay ;
XGpio_Initialize (&gpio_leds , XPAR_LEDS_8BIT_DEVICE_ID) ;
XGpio_SetDataDirection(&gpio_leds,1,0x0 ) ;
leds = 0x01 ;
XGpio_DiscreteWrite(&gpio_leds,1,leds ) ;
while ( 1 ) {
for(Delay = 0 ; Delay <LED_DELAY;Delay++);
leds = leds + 1 ;
XGpio_DiscreteWrite(&gpio_leds,1,leds) ;
} ;
} ;
```

Aussitôt le fichier enregistré, SDK compile automatiquement notre projet et crée l'exécutable ELF. Celui-ci se trouve par défaut dans le répertoire Debug du projet.

II.5.6 Envoi de l'exécutable vers la carte

Le fichier ELF compilé et assemblé peut maintenant être envoyé vers la carte. Pour ce faire, nous utilisons XMD, le débogueur de Xilinx qui est couvert en détails par une autre partie du cours.

Le débogueur peut être lancé dans XPS ou dans SDK. Pour cette introduction, nous utilisons SDK. Pour le démarrer, il suffit de sélectionner le menu « Xilinx ⟶ Tools XMD Console » puis de taper la commande

connect mb mdm : cette instruction permet de se connecter au processeur

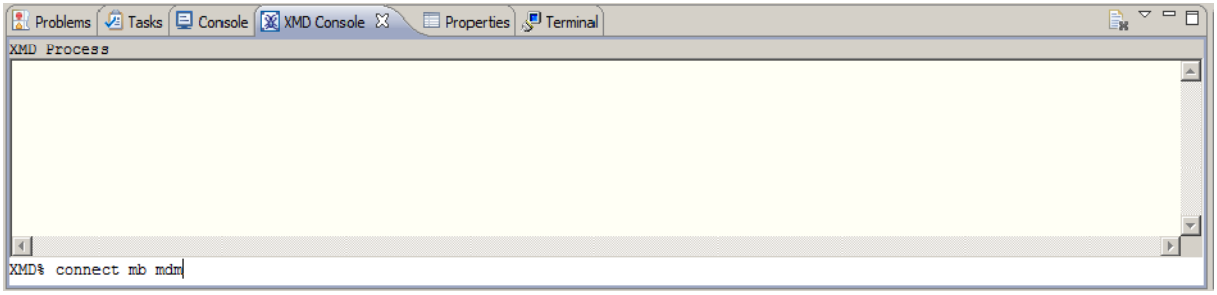


Figure II.22 Console XMD pour debugage

Il faut ensuite aller retrouver l'exécutable (fichier ELF) en naviguant dans les dossiers avec les commandes cd (changer de répertoire), ls (liste des fichiers du répertoire) et pwd (chemin complet en cours). Dans notre cas, nous tapons

cd D :/exemple_pfe_sdk/exmple_1/Debug

Pour envoyer le fichier exemple_1.elf, nous utilisons la commande dow :

Le terminal affichera alors l'information suivante aux dernières lignes :

Setting PC with Program Start Address 0x00000000 System Reset.... DONE

Ceci est signe que l'exécutable est désormais dans la mémoire programmable de la carte et que le logiciel est prêt à être lancé.

II.6 Conclusion:

Dans ce chapitre on a pu avoir une idée sur l'architecture ainsi le fonctionnement du processeur. Dans un premier lieu on a vu qu'il y'a deux types de processeurs (soft et hard). Le plus important est de connaitre les environnements de XILINX qui nous ont permis de créer et de configurer notre processeur.

Grâce à un exemple pratique on a réussi à connaitre les étapes nécessaires pour créer et configurer le processeur MicroBlaze, nous avons essayé d'expliquer chaque parties pour une bonne maitrise de la plus importante des parties, à savoir le paramétrage de notre processeur c'est-à-dire ajouter ou enlever des périphériques, selon la tâche que le processeur est censé accomplir. Ce qui reflète la principale raison pour laquelle on a choisi de travailler avec le MICROBLAZE sur FPGA. Cette raison ou cette caractéristique est la flexibilité (car elle donnera au système embarqué l'optimisation de la tâche qu'il va accomplir tout en consommant le moins possible les ressources disponible.

III.1 Introduction

Apres avoir acquis des notions générales sur les FPGA's, le processeur softcore et les environnements de XILINX qui nous permettent de configurer le processeur nous pouvons passer maintenant a la configuration du processeur pour les différents périphériques disponible sur la carte.

Dans ce chapitre nous allons présenter notre travail qui a pour but de faire cinq programmes qui consistent a créer des applications développées autour du processeur MicroBlaze implanté dans la carte FPGA en utilisant ces différents périphériques le premiers aura comme tache de faire une commande d'un jeux de lumière par des Switches et des boutons le deuxième et le troisième fera l'émission UART et la réception UART ; le quatrième va faire l'affichage sur l'écran LCD disponible sur la carte en dernier on établira l'émission d'une trame Ethernet.

III.2 Présentation de la carte SPARTAN 3AN

La carte SPARTAN 3-AN EST UNE CARTE hautement intégré sa densité d'intégration est de 90nm son volume relativement petit et son design simple lui permet d'etre utiliser dans plusieurs domaine : médicale, télécom,…. Elle offre aussi l'avantage d'être a faible cout ce qui permet une carte a la porté des particuliers (amateurs).

La Spartan 3AN incluent plusieurs périphériques : port VGA ; port série RS232, un afficheur LCD, port PS/2, 4 Switches, 4 boutons, 8 LEDs, fiche audio (jack), port RJ45, port USB et des ports d'extension pour pouvoir connecter la carte a d'autre modules

Elle comporte aussi une mémoire : 512Mb DDR2 SDRAM, un convertisseur numérique analogique et trois sources d'horloges : la première a 50MHz, une deuxième auxiliaire a 133MHz et une troisième qui peut être connectée en mode SMA pour générer des signaux a tres hautes fréquences.

Notre carte SPARTAN 3AN starter kit contient le processeur XC3S700AN, il offre les caractéristiques suivantes :

processeur	Système gates	Portes logiques équivalentes	CLB	slices	DCM	Block RAM (Bits)	Max IN/OUT	Taille du Bitstream	In système Flash bits
XC3S700AN	700K	13 248	1 472	5 888	8	360K	165	2 669K	8M

TABLEAU III.1 Caractéristiques du processeur XC3S700AN

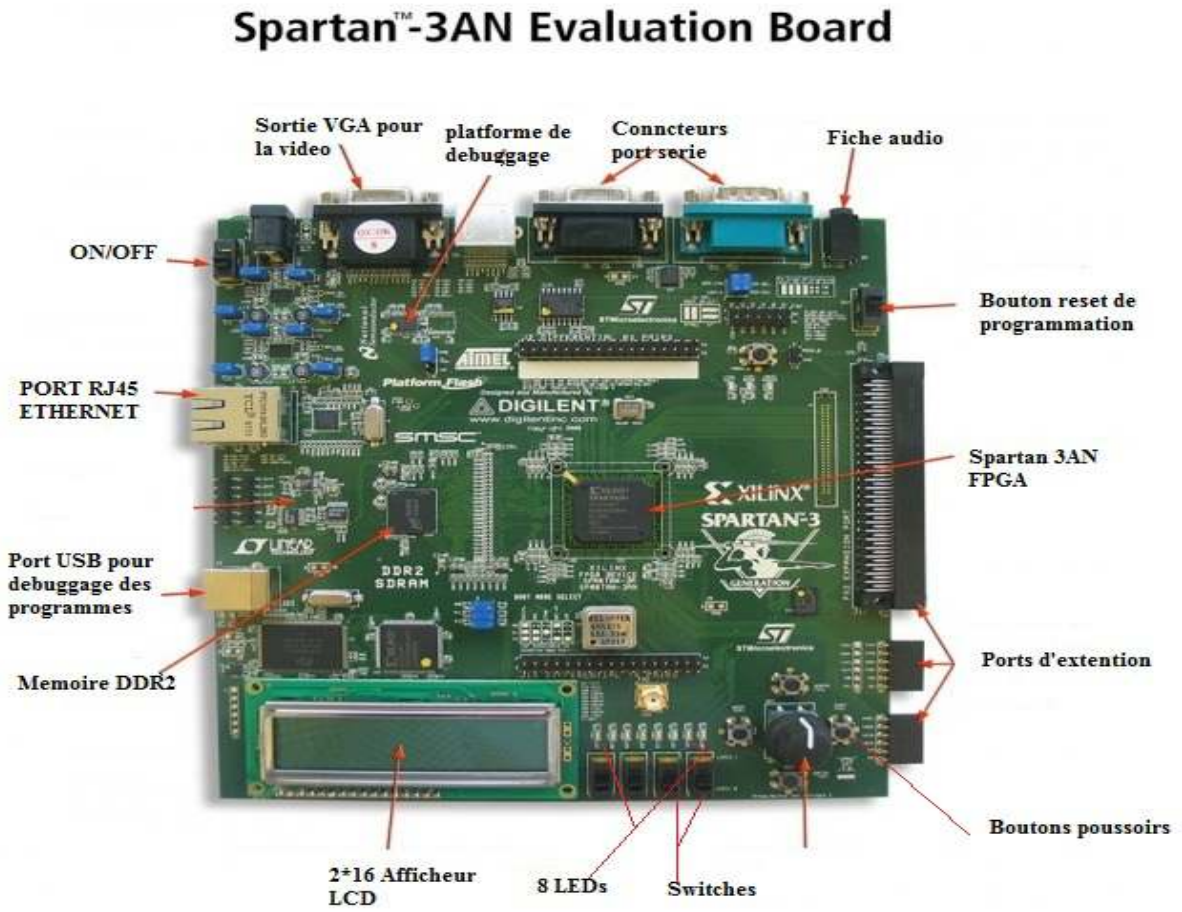


Figure III.1 Carte SPARTAN 3AN avec ces différents périphériques

III.3 Programme de commande d'un jeu de lumière via Switches et boutons

Parmi les périphériques disponibles sur la carte on a parlé de Switch de boutons et de Leds.

Ce programme va nous permettre de travailler avec ces trois périphériques simultanément. Il consiste à avoir plusieurs séquences de jeu de lumière (leds) ; le choix sur la séquence qui va être joué sera fera selon les conditions sur les Switches et les boutons.
Le but de ce programme c'est de faire la commande d'un jeu de lumière (LEDs) a travers des boutons et des switches on va commencer par définir nos périphériques.

III.3.1 Les Switches

La carte SPARTAN 3AN contient quatre switches localisés au milieu du bas de la carte come le montre la figure suivante

Quand le switch et sur la position ON (haut) il sera connecté au pin de 3.3 V et quand il est sur la position OFF (bas) il est connecté a la masse.

Le switch a besoin généralement de 2ms pour changer d'état après le basculement mécanique

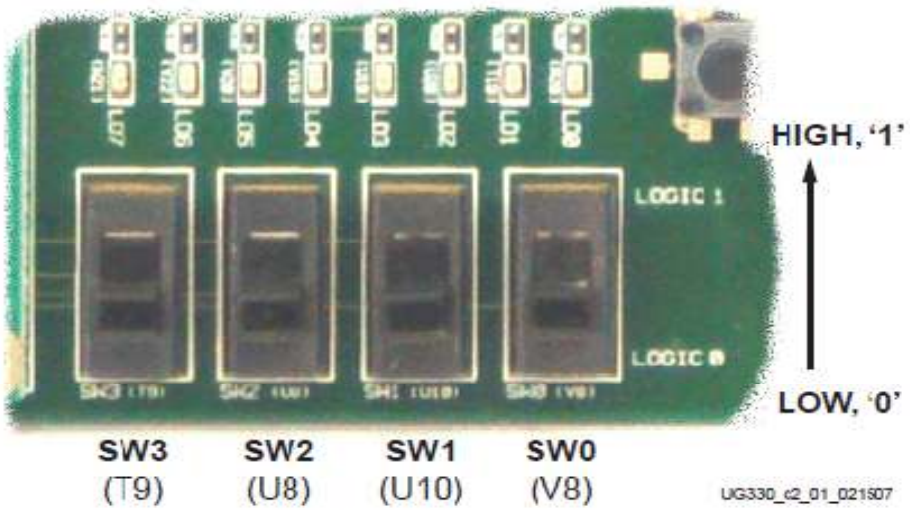


Figure III.2 Positionnement des Switches sur la carte

III.3.2 Les boutons

La carte SPARTAN 3AN contient quatre boutons poussoir ils sont situés dans le coin droit de la carte ils sont nommés boutons nord, est, ouest, sud et est comme le montre la figure suivante

Quand on appuis sur le bouton il sera connecté a une tension de 3.3 V et quand il n'est pas appuyai c'est un circuit ouvert donc c'est 0 V.

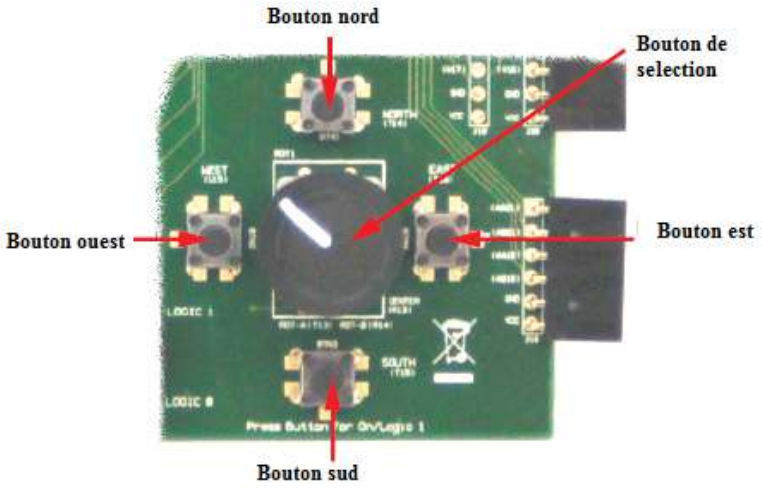


Figure III.3 Positionnement des boutons sur la carte

III.3.3 Les LEDs

Dans la carte on a aussi 8 LEDs nommé LED0 jusqu'à LED 7 en partant de la droite

Chaque LED a une broche connecté a la masse et une broche connecté a une résistance de protection de 390 Ω.

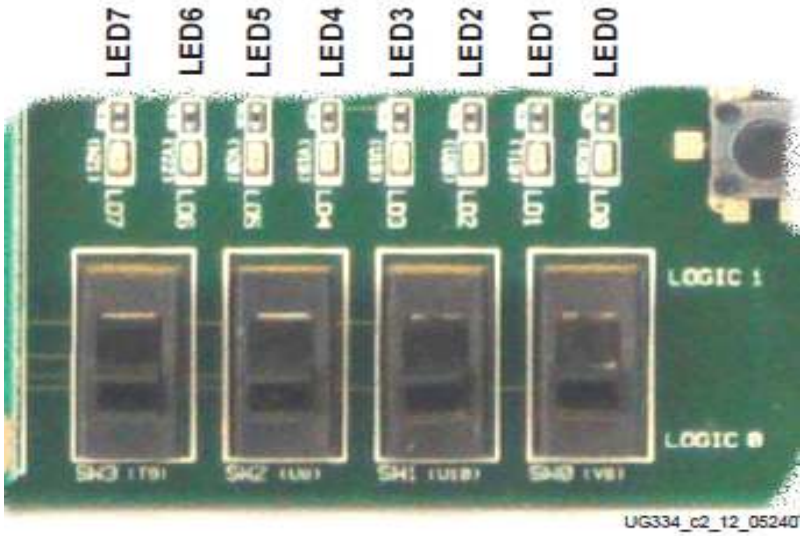


Figure III.3 Position des LEDs sur la carte

III.3.4 Fonctionnement de l'interface

Le fonctionnement du port est géré par le pilote (librairies) xgpio.h, il existe cinq routines (fonctions) permettant d'accéder au port et faire la lecture et/ou écriture.

III.3.5 Monter le pilote de l'interface E/S

Charger ou monter le pilote XGpio, cela nous permet d'utiliser les commandes du pilote afin d'accéder au périphérique.#include "xgpio.h"

III.3.5.1 Déclaration du port

La déclaration se fait par l'instruction XGpio, elle permet d'allouer une variable (type structure) pour le dispositif GPIO et le nommé, afin d'utiliser les fonctions du pilote.

Exemple :XGpiogpio_leds;

III.3.5.2 Initialisation du port

L'initialisation permet de monter le port à l'application, d'associer le nom déclaré avec l'ID du port (périphérique), et de charger les données du port vers la variable.

Exemple : XGpio_Initialize(&gpio_leds, XPAR_LEDS_8BIT_DEVICE_ID);

III.3.5.3 Configuration du port

La configuration est de définir la direction du port soit en entrée, soit en sortie, on peut aussi de configurer chaque pin séparément

Exemple : XGpio_SetDataDirection(&gpio_leds, 1, 0x0);

III.3.5.4 Lecture/Écriture du port

À ce niveau-là, on peut utiliser le port c'est à dire faire la lecture, s'il s'agit d'un port configuré en entrée, l'écriture s'il s'agit d'un port configuré en sortie

Exemple

Envoyer la valeur de Val vers le port gpio_leds

XGpio_DiscreteWrite(&gpio_leds, 1, Val);

Lecture du port Switches

Val=XGpio_DiscreteRead(&Switches, 1);

III.3.6 Le Timer

C'est un compteur de 32 bits qui peut fonctionner selon trois modes : mode Timer, mode générateur et mode MLI (PWM), ce périphérique est géré par le pilote xtmrctr. Dans la suite des applications on nécessite un délai, la réalisation de ce dernier ce fait comme suite :

III.3.6.1Monter le pilote du Timer

Charger ou monter le pilote xtmrctr.h, cela vous permet d'utiliser les commandes du pilote.

Exemple :#include "xtmrctr.h"

III.3.6.2 Déclaration du Timer

La déclaration se fait par l'instruction XTmrCtr, elle permet d'allouer une variable (type structure) pour le Timer et le nommé, afin d'utiliser les fonctions du pilote.

Exemple : XTmrCtrMyTimer;

III.3.6.3 Initialisation du Timer

L'initialisation permet de monter le Timer à l'application, d'associer le nom déclaré avec l'ID du périphérique, et de charger les données du Timer vers la variable déclaré.

Exemple : XTmrCtr_Initialize(&MyTimer,XPAR_XPS_TIMER_0_DEVICE_ID);

III.3.6.4 Configuration du Timer

La configuration est de définir le mode de fonctionnement, il s'agit :

Option 1 : XTC_ENABLE_ALL_OPTION (Permet à tous les Timers de compter à la fois).

Option 2 : XTC_DOWN_COUNT_OPTION (Configure le Timer de décompter à partir d'une valeur).

Option 3 : XTC_CAPTURE_MODE_OPTION (Configure le Timer pour faire le comptage d'un signal externe).

Option 4 : XTC_AUTO_RELOAD_OPTION

Option 5 : XTC_EXT_COMPARE_OPTION

Exemple :XTmrCtr_SetOptions(&MyTimer, 1, XTC_DOWN_COUNT_OPTION);

III.3.7 Sous-programme Délai

Dans la suite de nos programmes, on aura besoin à chaque fois d'un délai, donc nous avons réalisé un sous-programme delay_ms, selon le programme suivant :

```
voiddelay_ms(Xuint32 time)
{
XTmrCtr_SetResetValue(&MyTimer, 1, time * 50000);
XTmrCtr_Start(&MyTimer, 1);
while(!(XTmrCtr_IsExpired(&MyTimer, 1))){}
```

```
XTmrCtr_Stop(&MyTimer, 1);

}
```

Instruction 1

```
XTmrCtr_SetResetValue(&MyTimer, 1, time * 50000);
```

Dans cette instruction on déclare la valeur ou le timer se réinitialise à 0, c'est à dire nous avons 50000 cycle d'horloge par milliseconde par exemple si on a besoin que le timer atteigne une seconde 1sec = 1000ms ← 1000*50000 =50000000cycles

Instruction 2

```
XTmrCtr_Start(&MyTimer, 1);
```

Cette instruction permet de déclencher le timer

Instruction 3

```
while(!(XTmrCtr_IsExpired(&MyTimer, 1))){};
```

Cette instruction est une boucle qui permet de vérifier si le timer a atteint la valeur de reset

Instruction 4

```
XTmrCtr_Stop(&MyTimer, 1);
```

Si le timer atteint la valeur de reset, cette instruction permet de stopper le timer

Dans cette manipulation nous réalisons les opérations de lecture et d'écriture, la lecture des entrées numérique se fait via des switches (Port Switches configuré en entrée), l'écriture des données se fait vers des LEDs (Port gpio_leds configuré en sortie).

N.B : Il faut monter ou inclure le pilote « xparameters.h » dans le programme de l'application, ce dernier regroupe les paramètres des périphériques existant dans la plateforme

III.3.8 Algorithme 1 Pseudo code de commande d'un jeu de lumière via Switches et boutons

Cette manipulation consiste à réaliser un jeu de lumière programmable sur gpio_leds selon les séquences présentées comme le montre la figure **III.5**

La commande de la séquence qui va être joué parmi les sept disponible se fera à partir des cartes switches et trois boutons.

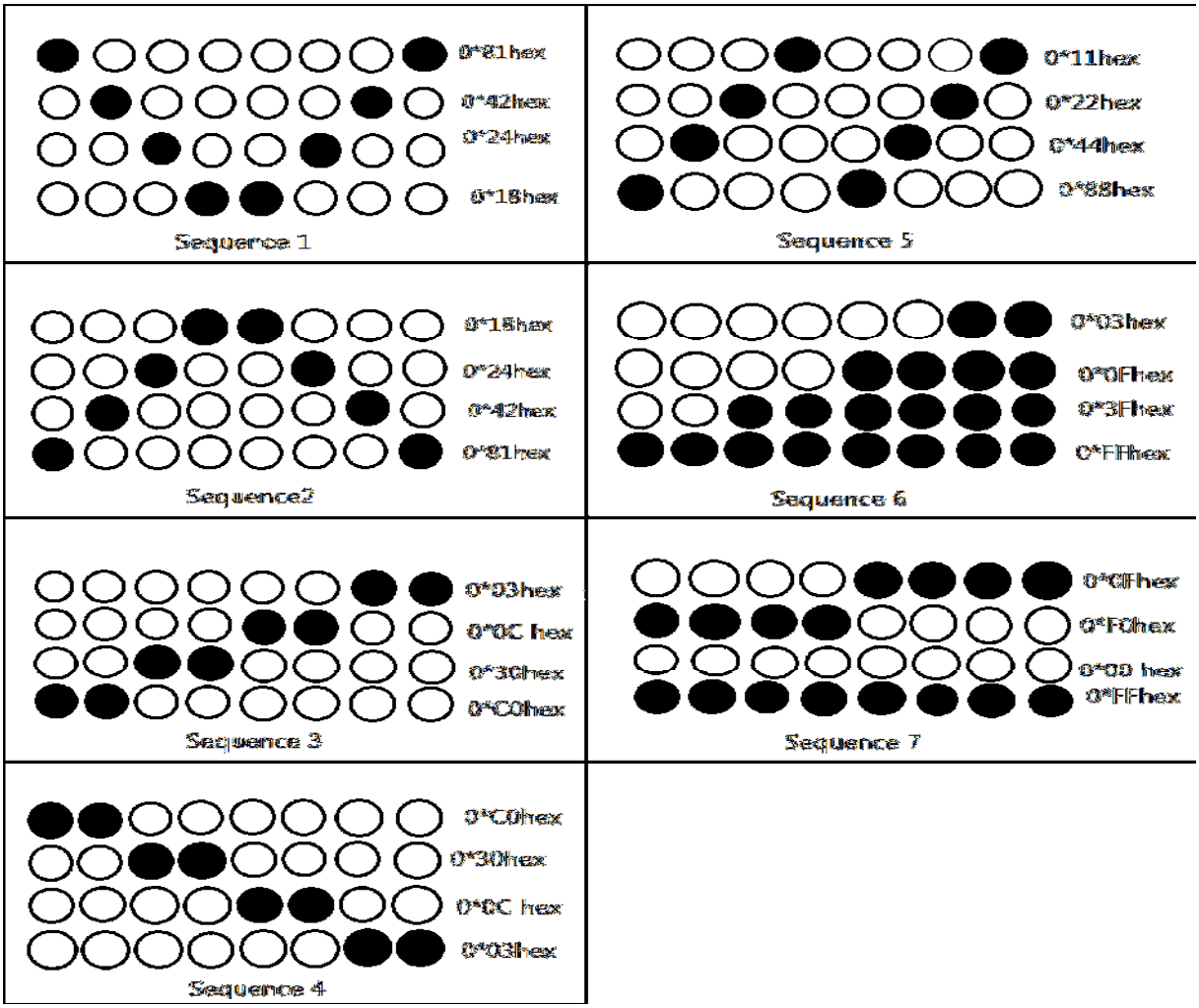


Figure III.5 Séquences de jeux de lumière programmées

L'algorithme sera donc :

Charger les Bibliothèques/Drivers du Matériel

Définir un Périphérique (Port) gpio_leds

Définir un Périphérique (Port) gpio_switch

Définir un Périphérique (Port) gpio_bouttons

Définir un Délai DELAY ← 1000

Initialisation du Port gpio_leds

Définir la Direction du Port gpio_leds →

Déclaration des vecteur des séquences désirésleds

While infinie do

if switch [i] est actif

for i = 0 to 7do

leds[i] ← gpio_leds (Envoyer la valeur leds[i] vers le port gpio_leds)

if buttons [i] est activé

for i = 0 to 7 do

leds[i] ← gpio_leds (Envoyer la valeur leds[i] vers le port gpio_leds)

Appel du Sous-programme Délai : delay_ms(DELAY)

End for

End while

III.3.8.1 Programme en C

```
#include"xparameters.h"
#include"xgpio.h"
#include"xtmrctr.h"
XTmrCtrMyTimer ;
voiddelay_ms( Xuint32 time);
intmain (void){
Xuint32delay ;
Xuint32 led2;
Xuint32 led3;

staticu32 leds1[8]={0x81,0x42,0x24,0x18} ;
staticu32 leds2[8]={0x18,0x24,0x42,0x81};
staticu32 leds3[8]={0x03,0x0C,0x30,0xC0};
staticu32 leds4[8]={0XC0,0x30,0x0C,0x03};
staticu32 leds5[8]={0x11,0x22,0x44,0x88};
```

```
staticu32 leds6[8]={0x03,0x0F,0x3F,0xFF};
staticu32 leds7[8]={0x0F,0XF0,0x00,0xFF} ;
XGpiogpio_leds;
XGpio Switches;
XGpiobuttons;

int i;
XGpio_Initialize(&gpio_leds , XPAR_LEDS_8BIT_DEVICE_ID);
XGpio_SetDataDirection(&gpio_leds , 1, 0x0 );
XTmrCtr_Initialize (&MyTimer ,XPAR_XPS_TIMER_0_DEVICE_ID);
XTmrCtr_SetOptions(&MyTimer ,1, XTC_DOWN_COUNT_OPTION);
XGpio_Initialize(&Switches,XPAR_DIPS_4BIT_DEVICE_ID);
XGpio_SetDataDirection(&Switches , 1 , 0xFFFFFFFF);
XGpio_Initialize (&Buttons, XPAR_BUTTONS_4BIT_DEVICE_ID);
XGpio_SetDataDirection(&Buttons,1,0xFFFFFFFF) ;
delay= 500;
while(1){
XGpio_DiscreteWrite(&gpio_leds,1, leds1[1]);
led2=XGpio_DiscreteRead(&Switches,1);
led3=XGpio_DiscreteRead(&Buttons,1);
if(led2==0x00000001){
for (i=0; i <4; i++){
XGpio_DiscreteWrite(&gpio_leds,1, leds1[i]);
delay_ms(delay) ;
}
}
if(led2==0x00000002){
for (i=0; i <4; i++){
XGpio_DiscreteWrite(&gpio_leds,1, leds2[i]);
delay_ms(delay);
}
if(led2==0x00000004){
for (i=0; i<4; i++){
XGpio_DiscreteWrite(&gpio_leds,1, leds3[i]);
delay_ms(delay);
 }
}
if(led2==0x00000008){
for(i=0;i<4; i++){
XGpio_DiscreteWrite(&gpio_leds,1, leds4[i]);
delay_ms(delay);
}
}
if(led3==0x00000001){
for (i=0; i <4; i++){
XGpio_DiscreteWrite(&gpio_leds,1, leds5[i]) ;
```

```
delay_ms(delay) ;
}
}
if(led3==0x00000002){
for (i=0; i <4; i++){
XGpio_DiscreteWrite(&gpio_leds,1, leds6[i]) ;
delay_ms(delay) ;
}
}
if(led3==0x00000004){
for (i=0; i <4; i++){
XGpio_DiscreteWrite(&gpio_leds,1, leds7[i]) ;
delay_ms(delay) ;
}
}
}
}
voiddelay_ms( Xuint32 time )
{
XTmrCtr_SetResetValue(&MyTimer,1,time * 50000) ;
XTmrCtr_Start(&MyTimer , 1 ) ;
while (!(XTmrCtr_IsExpired(&MyTimer,1))){}
XTmrCtr_Stop(&MyTimer,1) ;
}
```

III.3.8.3 Bilan de ressources :

On peut trouver les informations détaillé sur toutes les ressources utilisées par notre plateforme dans la fenêtre **Design Summary**

On remarque par exemple que le nombre des Flip Flop utilisés soit au nombre 1656 sur les 11776 a savoir 14% des ressources totale existant dans le FPGA ont était utilisé même pour les LUTs 22% (2607 sur les 11776) et pour les slice c'est 32% des 5888 disponible.

La figure suivante donne en détails toutes les ressources utilisées

De façon générale on peut dire que la plateforme choisis pour notre commande à minimiser le nombre de ressources utilisés ce qui a optimiser la performance de l'FPGA

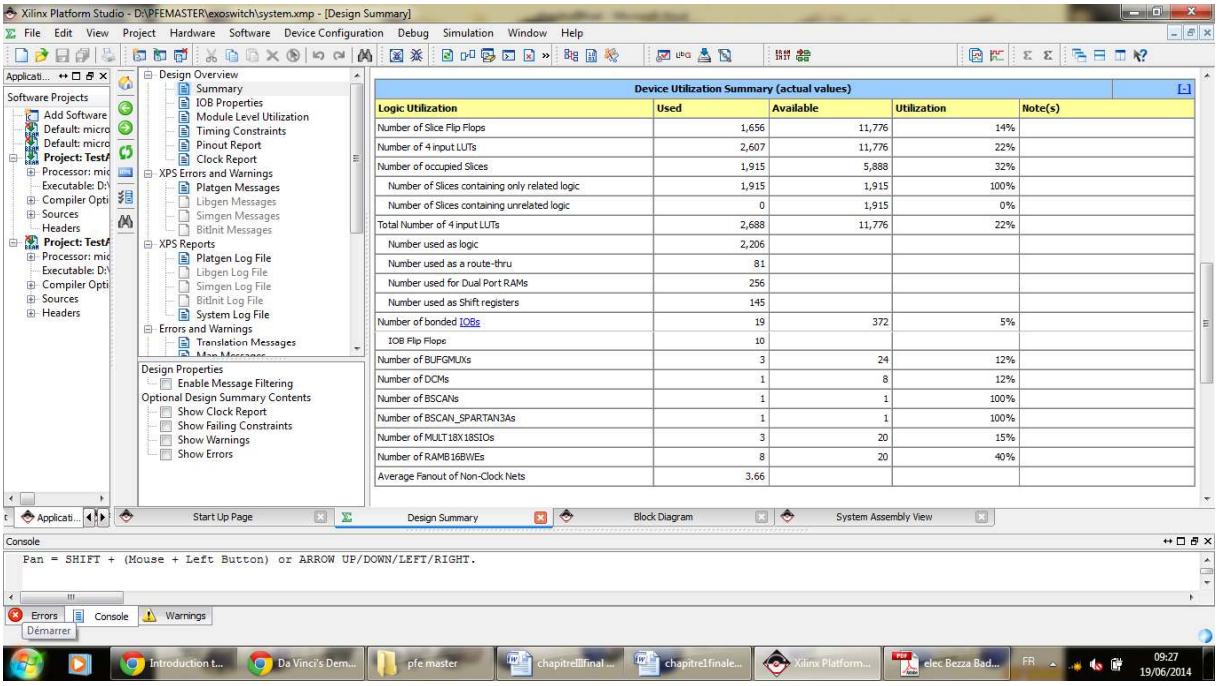


Figure III.6 Bilan des ressources

III.3.8.2 Bilan d'énergie

Pour obtenir le bilan d'énergie on doit aller dans la Tools box de **XILINX** et puis sur **XPower Analyzer.**

La figure **(III.7)** présente le bilan d'énergie que consomme la plateforme choisis, sous la forme d'un tableau qui explique l'énergie consommée par chaque élément de l'architecture.

On cite par exemple l'énergie consommé par les horloges utilisées est de 30mW ; l'énergie totale consommé par la plateforme de notre commande est donc de 115mW On remarque que l'énergie total très faible. Ce qui appuie l'un des avantages très important de l'utilisation de FPGA.

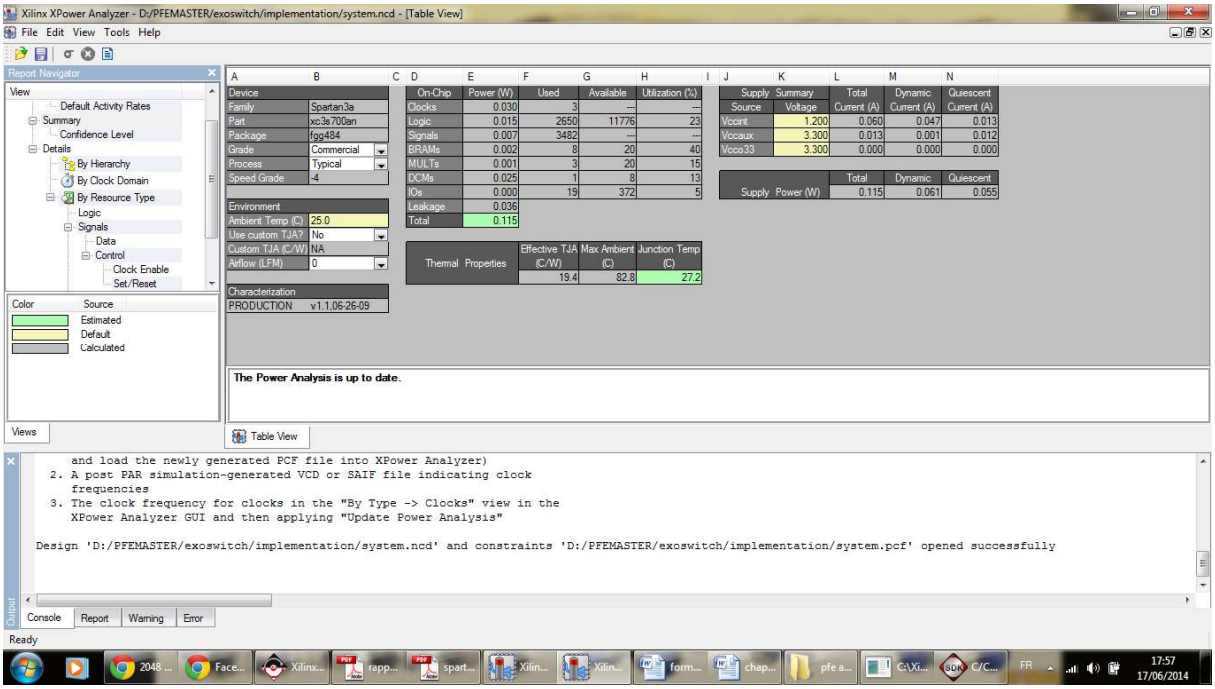


Figure III.7 Bilan d'énergie de l'architecture des switches boutons et LEDs

III.4 Transmission d'une donnée numérique via UART (RS232) :

Le but de cette manipulation est d'effectuer les opérations de transmission de données numériques via le port série RS232.

III.4.1 Le port série UART

La communication RS232 permet le transfert de données point à point. Il est couramment utilisé dans les applications d'acquisition de données pour transférer des données du et vers le PC.

III.4.2 Fonctionnement du port série

Le fonctionnement du port est géré par le pilote XUartLite, ce pilote permet d'accéder au port et faire l'envoie et/ou recevoir les données.

III.4.3 Monter le pilote du port série

Charger et/ou monter le pilote « xuartlite.h », cela vous permet d'utiliser les commandes du pilote afin d'accéder au périphérique/interface.

#include "xuartlite.h"

III.4.4 Déclaration du port série

La déclaration se fait par l'instruction XUartLite, elle permet d'allouer une variable (type structure) pour le dispositif GPIO et le nommé, afin d'utiliser les fonctions du pilote.

Exemple : XUartLiteUartLite;

III.4.5 Initialisation du port série

L'initialisation permet de monter le port à l'application, d'associer le nom déclaré avec l'ID du port (périphérique), et de charger les données du port vers la variable.

Exemple : XUartLite_Initialize(&UartLite, UARTLITE_DEVICE_ID);

III.4.6 Configuration du port série

La configuration est de définir les paramètres de communication du port, elle est accessible à l'entité « xparameters.h »

Exemple :

```
/* Definitions for peripheral RS232_DCE */
#define XPAR_RS232_DCE_BASEADDR 0x84000000
#define XPAR_RS232_DCE_HIGHADDR 0x8400FFFF
#define XPAR_RS232_DCE_DEVICE_ID 1
#define XPAR_RS232_DCE_BAUDRATE 9600
#define XPAR_RS232_DCE_USE_PARITY 0
#define XPAR_RS232_DCE_ODD_PARITY 0
#define XPAR_RS232_DCE_DATA_BITS 8
```

III.4.7 Emission/Réception via le port série

À ce niveau-là, on peut utiliser le port pour faire l'échange des données numériques

Exemple :

Envoyer d'une trame de donnée via UART :

```
XUartLite_Send(&UartLite, SendBuffer, BUFFER_SIZE);
```

Réception d'une trame de donnée via UART :

```
XUartLite_Recv(&UartLite, RecvBuffer, BUFFER_SIZE);
```

III.4.8 Assurance du transfert

Afin d'assurer que la trame a été envoyée, il s'agit d'une commande permet de tester si le module à encours d'envoyer, avant d'entamer d'autres commandes.

XUartLite_IsSending(&UartLite)

III.4.9 Transmission d'une Trame

Cette manipulation consiste à transmettre un message (RS232-UART dont le code ASCII [82 83 50 51 50 45 85 65 82 84]) du MicroBlaze (FPGA) vers le PC.

Ce programme va générer une trame en code ASCII ; cette trame sera afficher sur PC a l'aide d'un logiciel appeler Téra Tram.

Nous pouvons aussi envoyer une trame qui contient l'état des switchers, boutons ou même les leds, c'est-à-dire que le processeur va lire et générer une trame qui contient l'information sur l'état de ces derniers (voir annexe).

III.4.9.1 Algorthme2 Pseudo-Code transmission d'une trame via UART

Charger les Bibliothèques/Drivers du Matériel

Définir le Port Série UartLite

Déclaration de la taille du buffer BUFFER_SIZE = 16

Déclaration d'un buffer d'une taille BUFFER_SIZE SendBuffer

Initialisation du Port RS232_DCE

Charger les données à transmettre au buffer

SendBuffer[8] ⟵ [82, 83, 50, 51, 50, 45, 85, 65, 82, 84]

SendBuffer ⟶ RS232_DCE (Envoyer la trame via UART)

III.4.9.2 Programme en C

```
#include"xparameters.h"
#include"xgpio.h"
#include"xstatus.h"
#include"xuartlite.h"
#define TEST_BUFFER_SIZE 15
XUartLiteUartlite;
Xuint8SendBuffer [TEST_BUFFER_SIZE] ;
```

```
intmain(void)
{ XUartLite_Initialize (&UartLite,XPAR_RS232_DCE_DEVICE_ID);
SendBuffer [1]=0x51  ;
SendBuffer [2]= 0x70 ;
SendBuffer [3]= 0x70 ;
SendBuffer [4]= 0x6C ;
SendBuffer [5]= 0x69 ;
SendBuffer [6]= 0x62 ;
SendBuffer [7]= 0x74 ;
SendBuffer [8]= 0x69 ;
SendBuffer [9]= 0x6F ;
SendBuffer [10]= 0x6E;
SendBuffer [11]= 0x75;
SendBuffer [12]= 0x61;
SendBuffer [13]= 0x72;
SendBuffer [14]= 0x74;
XUartLite_Send(&UartLite,SendBuffer,TEST_BUFFER_SIZE);
}
```

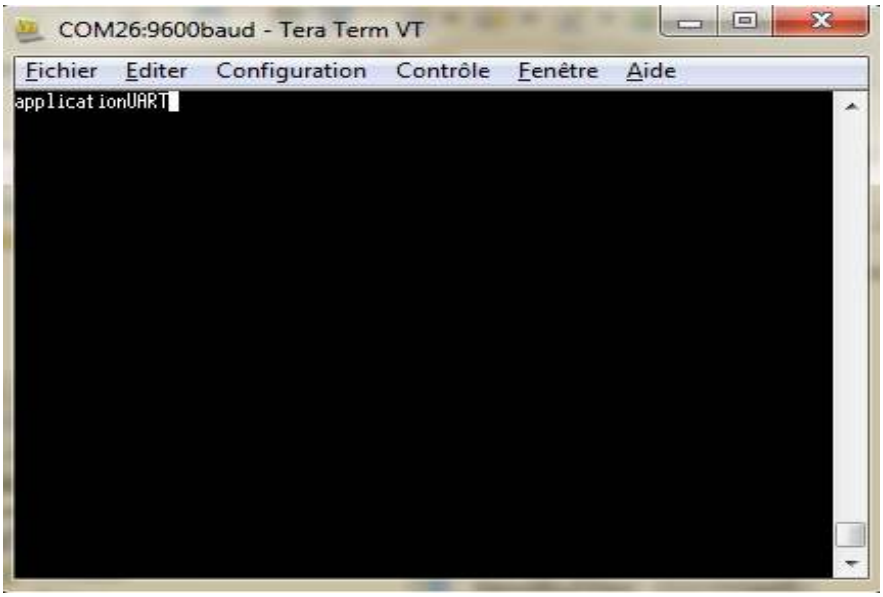


Figure III.8 Transmission d'une donnée via le port série

III.5 Réception via UART

Cette manipulation consiste à envoyer des caractères et/ou des chiffres du PC vers la carte FPGA (MicroBlaze), à son tour le FPGA envoie le caractère et/ou chiffre, selon l'algorithme 7 et le programme en C

III.5.1 Algorithme 3 Pseudo-Code réception d'une trame via UART

Charger les Bibliothèques/Drivers du Matériel

Définir le Port Série UartLite

Déclaration de la taille du buffer BUFFER_SIZE = 1

Déclaration deux buffers SendBuffer et RecvBuffer

Initialisation du Port RS232_DCE

while infinie do

RecvBuffer RS232_DCE (Réception d'un octet via UART)

Appel du Sous-Programme Délai : delay_us(1000)

Charger les données à transmettre au buffer

SendBuffer[3] ⟶ [RecvBuffer[0], 10, 13]

SendBuffer ⟵ RS232_DCE (Envoyer la trame via UART)

while (Trame en cours de Transfert) do

endwhile

Appel du Sous-programme Délai : delay_us(100000)

end while

III.5.2 Programme en C:

```
#include"xparameters.h"
#include"xgpio.h"
#include"xstatus.h"
#include"xuartlite.h"
#include"xtmrctr.h"
#define BUFFER_SIZE 5
XUartLiteUartLite ;
XTmrCtrMyTimer ;
Xuint8SendBuffer [BUFFER_SIZE] ;
Xuint8RecvBuffer [BUFFER_SIZE] ;
```

```
voiddelay_us ( Xuint32 time ) ;
intmain( void )
 {
XTmrCtr_Initialize (&MyTimer ,XPAR_XPS_TIMER_0_DEVICE_ID) ;
XTmrCtr_SetOptions(&MyTimer , 1 , XTC_DOWN_COUNT_OPTION) ;
XUartLite_Initialize (&UartLite , XPAR_RS232_DCE_DEVICE_ID) ;
while(1){
XUartLite_Recv(&UartLite , RecvBuffer , BUFFER_SIZE) ;
delay_us (1000);
SendBuffer [0]=RecvBuffer[0];
SendBuffer [1]=RecvBuffer[1];
SendBuffer [2]=RecvBuffer[2];
SendBuffer [3] = 10;
SendBuffer [4] = 13;
delay_us (1000) ;
 }
XUartLite_Send(&UartLite , SendBuffer,3) ;
while( XUartLite_IsSending(&UartLite ) )
 {
}
```

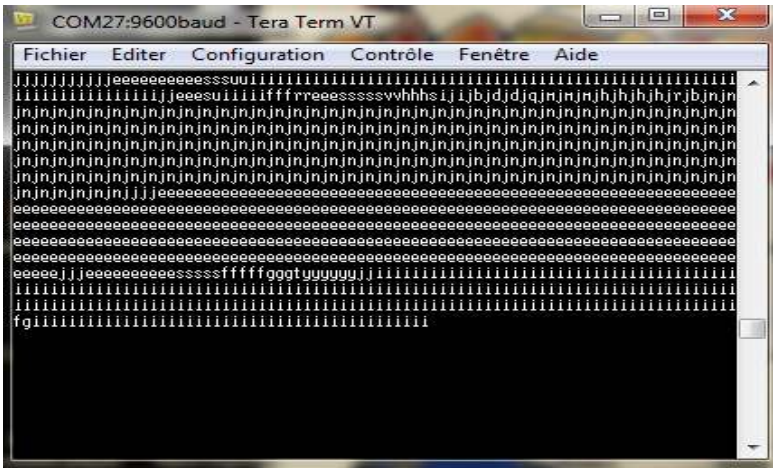


Figure III.9 Réception bien établie

III.5.3 Bilan de ressources :

Pour le bilan des ressources, le même constat : une architecture qui n'utilise pas beaucoup de ressources (14 % des **Flip flop** 22% de **LUTs** 34% des **slices**) (voir figure III.)

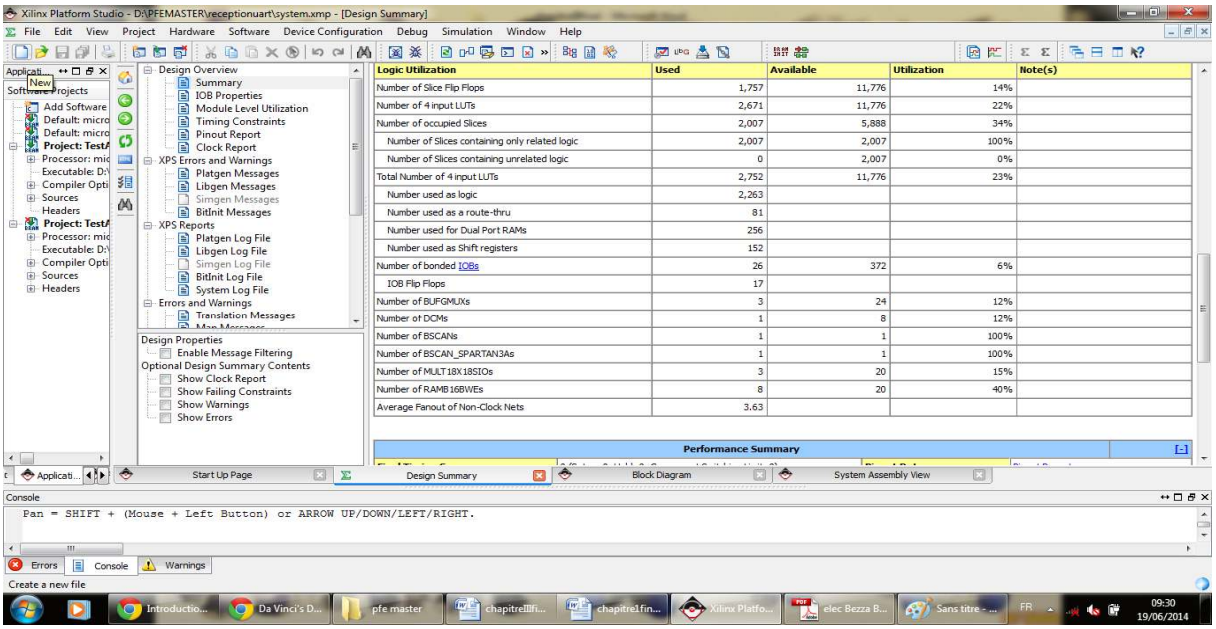


Figure III.10 Bilan des ressources

III5.4 Bilan d'énergie :

Figure III.11

III.6 Transmission de données numériques par Ethernet

Une large gamme de fonctionnalités de communication à distance est possible lorsque la connexion Ethernet est ajoutée aux conceptions embarquées. Par exemple, il peut permettre aux utilisateurs de surveiller à distance les systèmes, le but de cette manipulation est d'effectuer des opérations de transmission des trames de données via Ethernet

III.6.1Ethernet

Ethernet est le nom de la technologie de commutation de paquets LAN développée par la société Xerox au début des années 1970, puis standardisé par Xerox, Intel et Digital Equipment en 1978, L'IEEE publie peur après un standard compatible avec Ethernet sous le label 802.3 Depuis, la technologie Ethernet, qui présente de nombreuses variantes, est devenue la technologie LAN par excellence, à la base de la plupart des réseaux d'entreprise et domestiques.

III.6.2 Format des trames Ethernet

Ethernet est considéré comme un protocole de niveau liaison de données. Les paquets transmis au sein de tels réseaux sont appelés trames. Leur taille varie entre 64 et 1518 octets, incluant l'en-tête, les données et le contrôle par code de redondance cyclique (CRC – cyclyc Redundancy Check). Les trames Ethernet incluent un champ contenant l'adresse du destinataire (voir le Tableau III.1).

Préambule	Adresse destination	Adresse source	Type de la trame	Données de la trame	CRC
8 octets	6 octets	6 octets	2 octets	46-1500octets	4 octets

Table III.1 Format d'une trame Ethernet.

En plus de fournir l'adresse de la source et de la destination, toute trame transmise à travers d'un réseau Ethernet contient un préambule, un champ type, un champ de données et un CRC, le préambule se compose de 8 bits constitués de 0 et de1 en alternance pour permettre la synchronisation des interfaces réceptrices au niveau bit.

Le CRC de 32 bits sert quant à lui à détecter les erreurs de transmission : l'expéditeur le calcul. Il dépend des données contenues dans le paquet, le destinataire le recalcule pour vérifier l'intégrité des paquets entrants. Le récepteur accepte le paquet si le CRC est correct, sinon il le détruit.

Le champ de type contient un nombre entier codé sur 16 bits, il identifie le type des données que la trame transporte, pour le cas trame Ethernet décrire la longueur de la donnée.

III.6.3 Montage du Module Ethernet à la plateforme

La plateforme initiale ne contient pas un module Ethernet, afin d'ajouter ce dernier on consulte le volet IP Catalog dans l'environnement XPS, on le trouve sur le nom XPS 10/100 Ethernet MAC Lite dans la catégorie Communication High-Speed, ce dernier est commandé par le pilotexps_ethernetlite, il faut suivre les étapes suivant puis que le module sera fonctionner:

- Ajouter le module (IP core) XPS 10/100 Ethernet MAC Lite dans la plateforme et le nommé _xps_ethernetlite_0_

- Connecter l'IP avec le processeur via PLB bus
- Générer à nouveau les adresses puis que l'IP prendre leur espace d'adresse
- Écrire le brochage de l'IP dans le fichier UCF selon le datasheet de la carte SPARTAN 3AN
- Générer le Netlist et le Bitstream
- Réexportation des caractéristiques du système vers SDK, afin de faire la mise à jour

III.6.4 Fonctionnement du port Ethernet

Le fonctionnement du port est géré par le pilote xemaclite, ce pilote permet d'accéder au port et faire l'envoie et/ou recevoir les données.

III.6.5 Monter le pilote du port Ethernet

Charger et/ou monter le pilote xemaclite.h, cela vous permet d'utiliser les commandes du pilote afin d'accéder au périphérique/interface.

#include "xemaclite.h"

III.6.6 Déclaration du port Ethernet

La déclaration se fait par l'instruction XEmacLite, elle permet d'allouer une variable (type structure) pour le dispositif GPIO et le nommé, afin d'utiliser les fonctions du pilote.
Exemple: static XEmacLite EmacLiteInstance;

III.6.7 Initialisation du port Ethernet

L'initialisation permet de monter le port Ethernet à l'application, d'associer le nom déclaré avec l'ID du port, et de charger les données du port vers la variable.

Exemple :

```
ConfigPtr = XEmacLite_LookupConfig(DeviceId);
XEmacLite_CfgInitialize(EmacLitePtr, ConfigPtr,
ConfigPtr->BaseAddress);
```

III.6.8 Définir l'adresse MAC de la carte

Puis qu'il s'agit d'une communication Ethernet, il faut définir une adresse physique (MAC)

Exemple :

```
static u8 LocalAddress[XEL_MAC_ADDR_SIZE] ={0x88, 0x55, 0x04, 0x03, 0x02, 0x01};
```

```
XEmacLite_SetMacAddress(EmacLitePtr, LocalAddress);
```

III.6.9 Préparation d'envoi

Avant d'entamer la procédure d'envoi il faut faire deux choses :

- Effacer les trames de réception existante
- Assurer que le Buffer d'émission est libre pour le chargement des données à transmettre (Teste de disponibilité

```
XEmacLite_FlushReceive(EmacLitePtr);
if (XEmacLite_TxBufferAvailable(EmacLitePtr) != TRUE) {
return XST_FAILURE;
}
```

III.6.10 Emission/Réception via le port

À ce niveau-là, on peut utiliser le port pour faire l'échange des données numériques, il faut juste définir les données à transmettre et l'adresse MAC du destinataire.

Exemple : Envoyer d'une trame de donnée

```
SendFrame(EmacLitePtr, TxLength, RemoteAddress, Data, Size);
```

Réception d'une trame de donnée

```
RecvFrame(Data, Size);
```

III.6.11 Envoie une trame de données vers le PC via Ethernet

Cette manipulation consiste à envoyer une trame contient des données vers le PC possède l'adresse MAC suivant **[0x10,0x1F, 0x74, 0xEA, 0x9E, 0xCF],** la trame envoyée est le message **'master 2014'**qui a pour code ASCII **[0x6D, 0x41, 0x53, 0X54, 0x45, 0x52, 0x20, 0x32, 0x30, 0x31, 0x34, 0x4D**], et les résultats sont affichés sur le logiciel de visualisation des trames Ethernet (Wireshark) (voir figure 31 et la table 2).On lira le résultat sur la sous couche LLC.

III.6.11.1 LLC: Logical Link Control

La sous-couche LLC repose sur la sous-couche MAC. Lorsqu'une station a gagné son droit de parole, la sous-couche LLC contrôle la transmission des données.

III.6.11.2 Sous-couche MAC : Medium Access Control

Champs adresses destinataire et source.

On parle d'adresse MAC ou adresse physique : 3octets numéro du constructeur 3octets numéro de série.

L'adresse physique de la carte réseau est une adresse unique. Avec les trois premiers octets de l'adresse, on retrouve le constructeur de la carte.

III.6.12 Algorithme4 Pseudo-code envoie d'une trame via Ethernet

Charger les Bibliothèques/Drivers du Matériel

Définir l'adresse MAC de la carte LocalAddress

Définir l'adresse MAC du destinataire RemoteAddress

Définir le port Ethernet EmacLiteInstance

Déclaration du buffer d'émessionSendBuffer

Définir la taille de la trame TxLength MIN = 46,MAX = 1500

Chargement des données de la trame DATA

Définir une variable structure de configuration ConfigPtr

Chargement de la configuration du module Ethernet ConfigPtr

Intialisation du module Ethernet

Chargement de l'adresse MAC

Effacer les trames de recéption existant

while (Buffer d'émission vide 6= Vrai) do

end while

Appel du Sous-programme SendFrame (NomEmac,TxLength,RemoteAddress,DATA, Size) ;

III.6.13 Programme en C:

```
#include"xparameters.h"
#include"xstatus.h"
#include"xemaclite.h"


/************************** Constant Definitions *****************************/

/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#define EMAC_DEVICE_ID          XPAR_EMACLITE_0_DEVICE_ID
#define INTC_DEVICE_ID          XPAR_INTC_0_DEVICE_ID
#define INTC_EMACLITE_ID      XPAR_INTC_0_EMACLITE_0_VEC_ID
```

```
#define EMACLITE_TEST_FRAME_SMALL_SIZE       46   /* Size of a small Test Frame */
#define EMACLITE_TEST_FRAME_LARGE_SIZE   1500 /* Size of a long Test Frame */

/************************** Type Definitions *****************************/


/***************** Macros (Inline Functions) Definitions *********************/


/************************** Function Prototypes *****************************/

intEMACLiteConfig_Send(u16 DeviceId,u32 TxLength,u8 *Data,u8 Size);

staticintSendFrame(XEmacLite *InstancePtr, u32 PayloadSize, u8 *DestAddress, u8 *Data, u8
DataSize);
/************************** Variable Definitions *****************************/

/*
 * Set up valid local and remote MAC addresses. The loopback tests will
 * use the LocalAddress both as source and destination, while the network
 * tests will use both RemoteAddress and LocalAddress
 */
static u8 RemoteAddress[XEL_MAC_ADDR_SIZE] =
{
  0x10, 0x1F, 0x74, 0xEA, 0x9E, 0xCF

};
static u8 LocalAddress[XEL_MAC_ADDR_SIZE] =
{
   0x88, 0x55, 0x04, 0x03, 0x02, 0x01
};

staticXEmacLiteEmacLiteInstance;    /* Instance of the EmacLite */

/*
 * Buffers used for Transmission and Reception of Packets
*/
static u8 TxFrame[XEL_MAX_FRAME_SIZE];
//static u8 RxFrame[XEL_MAX_FRAME_SIZE];

staticvolatile u32 RecvFrameLength;
staticvolatileintTransmitComplete;

/*****************************************************************************
*/
```

```
/**
*
* This function is the main function of the XEmaclite Level 1 example.
*
* @paramNone
*
* @return       XST_SUCCESS to indicate success, otherwise XST_FAILURE.
*
* @note             None
*
*****************************************************************************
**/
int main()
{
int Status;
u32TxLength;
/*
       * Run the EmacLite L1 example , specify the the Device ID that is
       * generated in xparameters.h
       */
TxLength = EMACLITE_TEST_FRAME_SMALL_SIZE;
u8 Data[12] =
                        {
                  0x6D, 0x41, 0x53, 0X54, 0x45, 0x52, 0x20, 0x32, 0x30, 0x31, 0x34, 0x4D
};
u8 Size=12;
      Status = EMACLiteConfig_Send(EMAC_DEVICE_ID,TxLength, Data, Size);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
}
return XST_SUCCESS;

}
/*****************************************************************************
**/
/**
*
* The main entry point for the EmacLite driver in interrupt mode example.
*
* @paramDeviceId is device ID of the XEmacLiteDevice , typically
*            XPAR_<EMAC_instance>_DEVICE_ID value from xparameters.h
*
* @return       XST_SUCCESS to indicate success, otherwise XST_FAILURE
*
* @note             None.
*
```

```
*****************************************************************************
***/
intEMACLiteConfig_Send(u16 DeviceId,u32 TxLength,u8 *Data,u8 Size)
{
int Status;
//XIntc *IntcPtr;
XEmacLite *EmacLitePtr;
XEmacLite_Config *ConfigPtr;

RecvFrameLength = 0;
//IntcPtr = &IntcInstance;
EmacLitePtr =&EmacLiteInstance;
/*u8 myData=Data;
u32myLength=TxLength;
u8mySize=Size;*/

/*
	* Initialize the EmacLite device.
	*/
ConfigPtr = XEmacLite_LookupConfig(DeviceId);
if (ConfigPtr == NULL) {
return XST_FAILURE;
	}
	Status = XEmacLite_CfgInitialize(EmacLitePtr,
ConfigPtr,
ConfigPtr->BaseAddress);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
}
/*
	* Set the MAC address.
	*/
XEmacLite_SetMacAddress(EmacLitePtr, LocalAddress);
/*
	* Set up the interrupt infrastructure
	*/
/*Status = SetUpInterruptSystem(IntcPtr);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
}*/
/*
	* Empty any existing receive frames.
	*/
XEmacLite_FlushReceive(EmacLitePtr);
/*
	* Start the EMACLite controller
```

```
*/
//XEmacLite_EnableInterrupts(EmacLitePtr);

/*
	* Check if there is a tx buffer available.
*/
if (XEmacLite_TxBufferAvailable(EmacLitePtr) != TRUE) {
return XST_FAILURE;
	}

/*****************Envoyer message ***************/

	Status = SendFrame(EmacLitePtr, TxLength,
RemoteAddress, Data, Size);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
	}
/*
	* Wait for the frame to be transmitted
	*/
while (TransmitComplete == FALSE);

return XST_SUCCESS;
}
/*****************************************************************************
***/
/**
*
* This function sends a frame of given size. This function assumes interrupt
* mode and sends the frame.
*
* @paramXEmacInstancePtr is a pointer to the XEmacLite instance to be
*		worked on.
* @paramPayloadSize is the size of the frame to create. The size only
*		reflects the payload size, it does not include the Ethernet
*		header size (14 bytes) nor the Ethernet CRC size (4 bytes).
* @paramDestAddress if the address of the remote hardware the frame is
*		to be sent to.
*
* @return	XST_SUCCESS if successful, a driver-specific return code if not.
*
* @note		None.
*
*****************************************************************************
***/
staticintSendFrame(XEmacLite *XEmacInstancePtr, u32 PayloadSize,
```

```
                u8 *DestAddress, u8 *Data, u8 DataSize)
{
      u8 *FramePtr;
      u8 *AddrPtr = DestAddress;
u8 Index;
int Status;

/*
       * Set the Complete flag to false
       */
TransmitComplete = FALSE;
/*
       * Assemble the frame with a destination address and the source address
       */
FramePtr = (u8 *)TxFrame;

      *FramePtr++ = *AddrPtr++;
      *FramePtr++ = *AddrPtr++;
      *FramePtr++ = *AddrPtr++;
      *FramePtr++ = *AddrPtr++;
      *FramePtr++ = *AddrPtr++;
      *FramePtr++ = *AddrPtr++;
/*
       * Fill in the source MAC address
       */
      *FramePtr++ = LocalAddress[0];
      *FramePtr++ = LocalAddress[1];
      *FramePtr++ = LocalAddress[2];
      *FramePtr++ = LocalAddress[3];
      *FramePtr++ = LocalAddress[4];
      *FramePtr++ = LocalAddress[5];

/*
       * Set up the type/length field - be sure its in network order
       */
      *((u16 *)FramePtr) = PayloadSize;
FramePtr++;
FramePtr++;
/*
       * Now fill in the data field with known values so we can verify them
       * on receive.
       */
/*for (Index = 0; Index <PayloadSize; Index++) {
            *FramePtr++ = (u8)Index;
      }*/
for (Index = 0; Index < DataSize+1; Index++) {
```

```
                *FramePtr++ = Data[Index];
            }
/*
     * Now send the frame
*/
Status = XEmacLite_Send(XEmacInstancePtr, (u8 *)TxFrame,
PayloadSize + XEL_HEADER_SIZE);

return  Status;
}
```

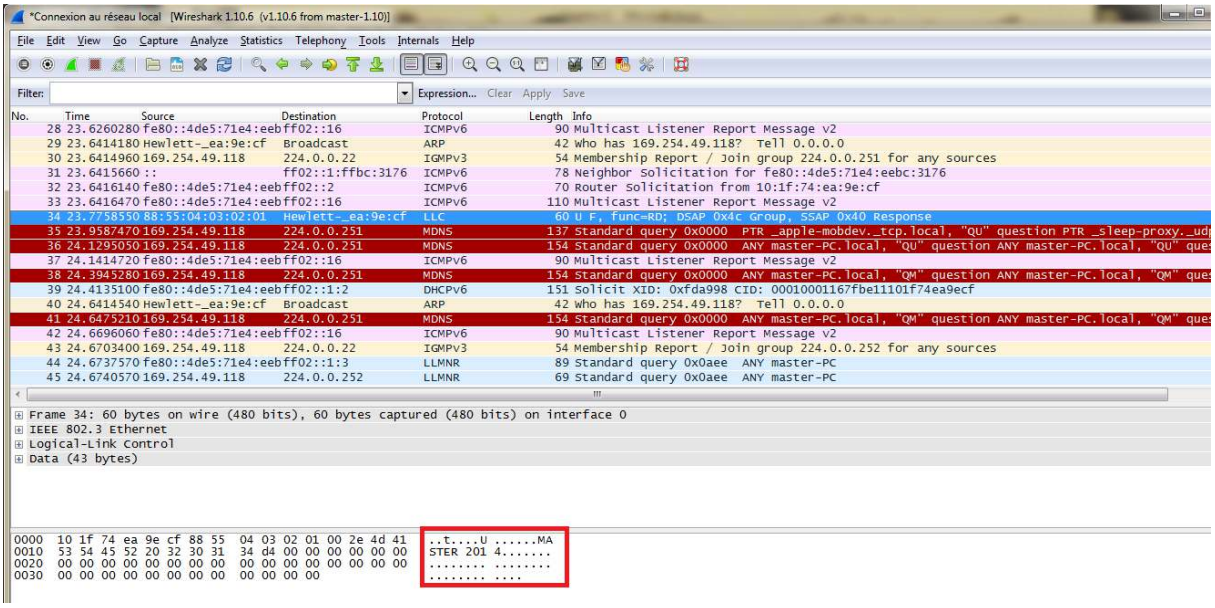


Figure III.12 Transmission de la trame établie

III.14 Bilan des ressources

Pour la plateforme de notre commande Ethernet on remarque le nombre de ressources a augmenter cela est du a la complexicité de notre design. (19% pour les **Flip Flop**, 28% pour les **LUT's**, 43% **slices**).

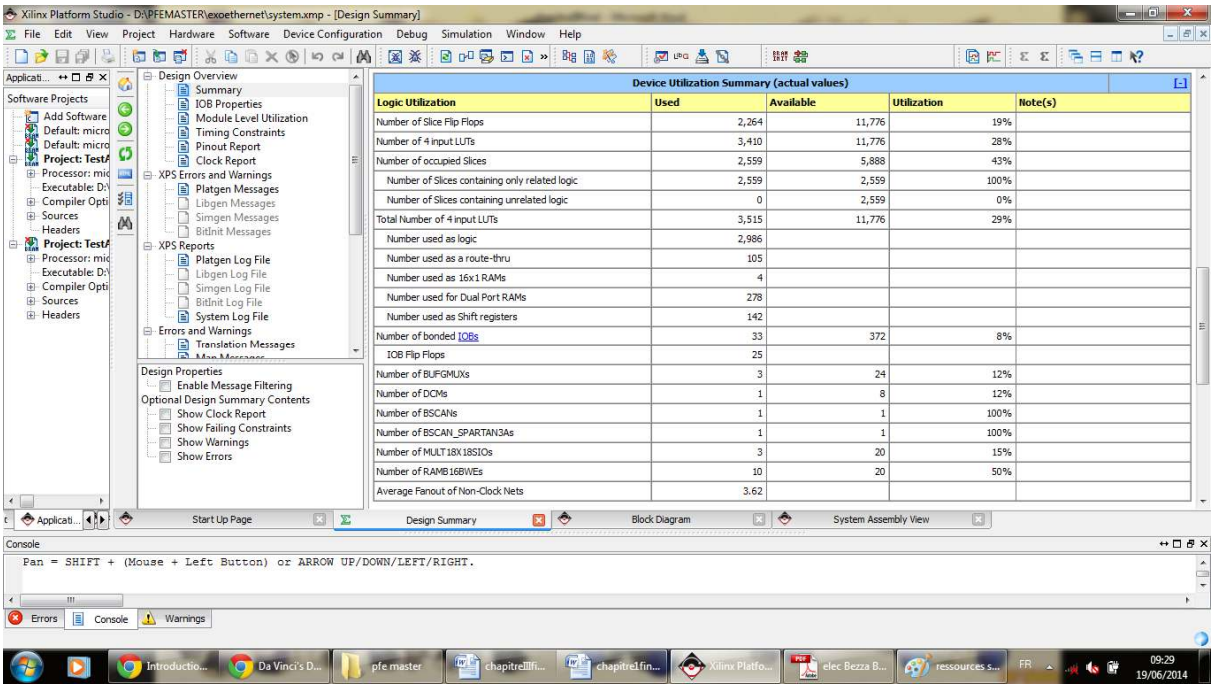


Figure III.13 Bilan des ressources

III.6.15 Bilan d'énergie

On remarque que l'énergie totale à augmenter et cela est du au fait que le nombre de ressources a augmenté.

La figure III.10 montre que l'énergie consommée par l'architecture reste assez optimale avec une puissance consommer de 0.130W.

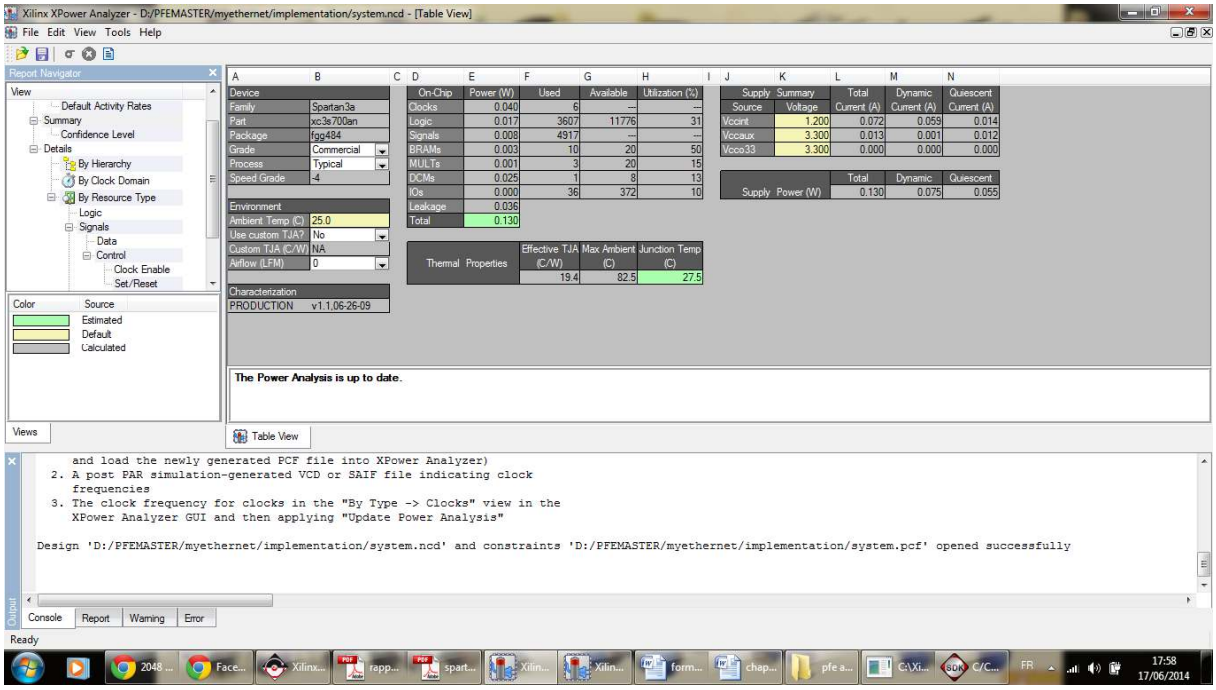


Figure III.14 Bilan d'énergie de l'architecture Ethernet

III.7 Afficheur LCD

On trouve aussi dans la carte SPARTAN3 an un afficheur LCD a deux lignes de 16 caractères il est contrôlé par le FPGA via un registre a 8 bit comme le montre la figure III.6

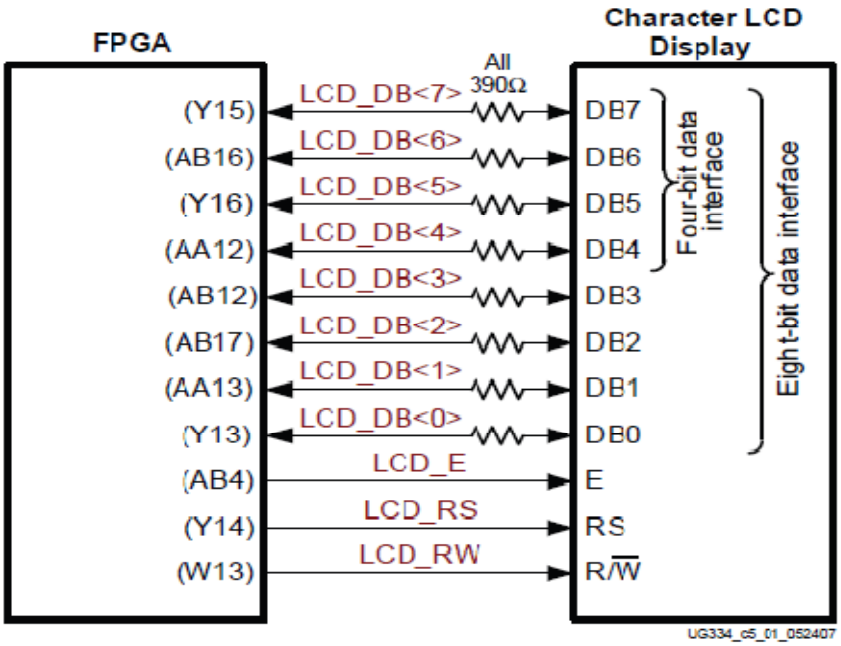


Figure III.15 Interface d'affichage du LCD

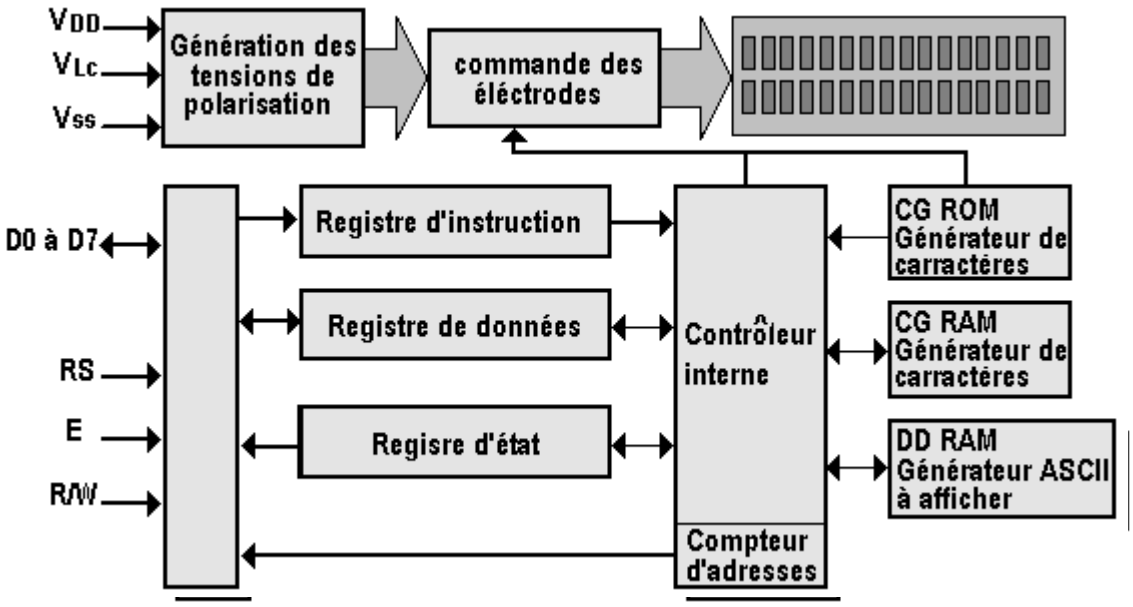


Figure III.16 Schéma fonctionnel interne de l'afficheur LCD.

Le schéma fonctionnel interne de l'afficheur LCD présenté à la figure **III.12** contient les blocs suivants :

- **Registre de données :** bidirectionnel, il reçoit les codes ASCII des caractères à afficher. Les codes seront ensuite stockés dans la DD-RAM. Le registre de données permet aussi de lire le contenu de la DD-RAM.
- **Registre de contrôle :** destiné à recevoir les consignes de contrôle, comme effacer l'afficheur, déplacer le curseur, etc. (accessible uniquement en écriture).
- **Registre d'état :** destiné à indiquer à l'utilisateur si le processeur interne est prêt à recevoir une instruction (accès en lecture seule).

- **Le contrôleur :** l'élément de base auquel toutes les fonctions souhaitées réalisées par l'afficheur sont orchestrées par ce dernier.
- **La CG-ROM** : C'est une ROM génératrice de caractères qui fournit 192 motifs de caractères différents en matrices de 5 x 7 points. La relation entre le code à transmettre et le motif du caractère est donné en figure 02. Il faut signaler que la correspondance ASCII est respectée uniquement entre les codes $20 et $7D, c'est à dire pour les chiffres de 0 a 9, les lettres majuscules et minuscules, et enfin quelques-uns couramment utilisés.

➢ **La CG-RAM** : c'est une RAM génératrice de caractères à programmer qui fournit 8 motifs. On peut ainsi obtenir un jeu de 8 caractères personnalisés.

- **La DD-RAM** : reçoit les codes des caractères à afficher. L'adresse à laquelle est placé un code dans la DD RAM définit la position du caractère sur le panneau d'affichage.
- **Le compteur d'adresse** : Le pointage d'un élément dans la DD-RAM ou la CG-RAM est déterminé par un compteur d'adresses interne, accessible en lecture par l'utilisateur.
- **Lignes de données D0 à D7 :** Les données peuvent être transmises sur 8 ou 4 bits. Dans ce dernier cas on utilise les lignes D4 à D7. On envoie d'abord le poids fort et ensuite le poids faible.
- **Ligne R/W :** Unidirectionnelle, lors de l'écriture d'un registre on la positionne à zéro, pour une lecture on la positionne à 1.
- **Ligne RS :** Unidirectionnelle, lors de l'envoie d'un code instruction on positionne la ligne a 0 lors de l'envoie d'un code caractère on positionne la ligne a 1.
- **Ligne E :** Unidirectionnelle, elle permet de valider les données sur front

III.7.1 Brochage de l'afficheur LCD

Le tableau suivant illustre une description sur les broches de l'afficheur LCD

Numéro de broche du connecteur	Symbole	Type	Description
1	GND	alimentation	masse (0V)
2	Vdd	alimentation	+ 5 V (alimentation du contrôleur interne)
3	V0	alimentation	alimentation du panneau LCD
4	RS (Register Select)	entrée	• RS = 1 : sélection du registre de données • RS = 0 : ○ en mode écriture : sélection du registre d'instruction ○ en mode lecture : sélection du drapeau BUSY et du compteur d'adresse
5	R/W (Read / Write)	entrée	• R/W = 1 : mode lecture • R/W = 0 : mode écriture
6	E (Enable)	entrée	Entrée de validation

7	DB0 (Data bit 0)	entrée/sortie	Bus de données (8 bits) • DB0 : bit de poids faible (LSB) • DB7 : bit de poids fort (MSB) • **En mode écriture : bus configuré en entrée** • **En mode lecture : bus configuré en sortie**
8	DB1 (Data bit 1)	entrée/sortie	
9	DB2 (Data bit 2)	entrée/sortie	
10	DB3 (Data bit 3)	entrée/sortie	
11	DB4 (Data bit 4)	entrée/sortie	
12	DB5 (Data bit 5)	entrée/sortie	
13	DB6 (Data bit 6)	entrée/sortie	
14	DB7 (Data bit 7)	entrée/sortie	
15	A (Anode)	alimentation	alimentation du système de rétro éclairage
16	K (Cathode)	alimentation	

Tableau III.3 Les broches de l'afficheur LCD.

II.7.2 Instructions de contrôle et d'affichage

Un afficheur LCD est capable d'afficher tous les caractères alphanumériques usuels et quelques Symboles supplémentaires. Pour certains afficheurs, il est même possible de créer ses propres Caractères.

Chaque caractère est identifié par son code ASCII qu'il faut envoyer sur les lignes D0 à D7 broches 7 A 14. Ces lignes sont aussi utilisées pour la gestion de l'affichage avec l'envoi d'instructions telles Que l'effacement de l'écran, l'écriture en ligne 1 ou en ligne 2, le sens de défilement du curseur.

III.7.3 Montage du module LCD :

Apres avoir pris le temps de comprendre le fonctionnement interne d'un afficheur a cristaux liquide LCD nous allons pouvoir expliquer comment monter le module LCD dans une architecture pour pouvoir le commander a l'aide du processeur MicroBlaze

Tout d'abord on doit savoir que l'on a utilisé une architecture déjà existante la plateforme initiale ne contenait pas le module LCD on v donc expliquer comment pouvoir l'ajouter

On consulte le **volet IP CATALOG** dans l'environnement **XPS**, on le trouve sous le nom de **GENERAL PURPOSE I/O**, on joute alors le module **(IP)** Dans la plateforme et le nommé « XPS_gpio_0 » on connecte alors le **IP** via **PLB bus** et on génère ensuite les adresse puisque le IP va prendre des espaces mémoires on va ensuite sur le fichier **UCF** pour pouvoir écrire le brochage de l'**IP** selon le Datasheet de la carte (voir annexe)

Net xps_gpio_0_GPIO_IO_O_pin<E> LOC = "AB4" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;

Net xps_gpio_0_GPIO_IO_O_pin<RS> LOC = "Y14" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;

Net xps_gpio_0_GPIO_IO_O_pin<RW> LOC = "W13" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;

Net xps_gpio_0_GPIO_IO_O_pin<7> LOC = "Y15" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;

Net xps_gpio_0_GPIO_IO_O_pin<6> LOC = "AB7" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;

Net xps_gpio_0_GPIO_IO_O_pin<5> LOC = "Y16" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;

Net xps_gpio_0_GPIO_IO_O_pin<4> LOC = "AA12" | IOSTANDARD = LVCMOS33 | DRIVE = 8 | SLEW = SLOW ;

Le LCD est maintenant implanter dans la plateforme et on peut donc procédé à la programmation de notre commande.

III.7.6 le programme en C

```
#include"xparameters.h"
#include"xbasic_types.h"
#include"xgpio.h"
#include"xstatus.h"
#include"xtmrctr.h"

// Masks to the pins on the GPIO port

#define LCD_DB4     0x01
#define LCD_DB5     0x02
#define LCD_DB6     0x04
#define LCD_DB7     0x08
#define LCD_RW      0x10
#define LCD_RS      0x20
#define LCD_E       0x40
#define LCD_TEST    0x80

// Global variables

XGpio GpioOutput;
XTmrCtr DelayTimer;

// Function prototypes

voiddelay_us(Xuint32 time);
voiddelay_ms(Xuint32 time);
voidgpio_write(Xuint32 c);
Xuint32gpio_read(void);
voidlcd_clk(void);
voidlcd_set_test(void);
voidlcd_reset_test(void);
voidlcd_set_rs(void);
voidlcd_reset_rs(void);
voidlcd_set_rw(void);
voidlcd_reset_rw(void);
voidlcd_write(Xuint32 c);
voidlcd_clear(void);
voidlcd_puts(constchar * s);
voidlcd_putch(Xuint32 c);
voidlcd_goto(Xuint32 line,Xuint32 pos);
voidlcd_init(void);

// Main function
```

```
intmain (void)
{
// Initialize the Timer
  XTmrCtr_Initialize(&DelayTimer,XPAR_XPS_TIMER_0_DEVICE_ID);
  XTmrCtr_SetOptions(&DelayTimer, 1, XTC_DOWN_COUNT_OPTION);
// Initialize the GPIO driver for the LCD
  XGpio_Initialize(&GpioOutput,XPAR_XPS_GPIO_0_DEVICE_ID);
// Set the direction for all signals to be outputs
  XGpio_SetDataDirection(&GpioOutput, 1, 0x00);
// Initialize the LCD
  lcd_init();
// Example write to the LCD
  lcd_puts("Eq. Concept");
  lcd_goto(1,1);
  lcd_puts("Syst.Embarques2");
while(1){
  }
}
// Delay function (microseconds)
voiddelay_us(Xuint32 time)
{
  XTmrCtr_SetResetValue(&DelayTimer, 1, time * 125);
  XTmrCtr_Start(&DelayTimer, 1);
while(!(XTmrCtr_IsExpired(&DelayTimer, 1))){}
  XTmrCtr_Stop(&DelayTimer, 1);
}
// Delay function (milliseconds)
voiddelay_ms(Xuint32 time)
{
  XTmrCtr_SetResetValue(&DelayTimer, 1, time * 125000);
  XTmrCtr_Start(&DelayTimer, 1);
while(!(XTmrCtr_IsExpired(&DelayTimer, 1))){}
  XTmrCtr_Stop(&DelayTimer, 1);
}
// Write to GPIO outputs
voidgpio_write(Xuint32 c)
{
// Write to the GPIOs
XGpio_DiscreteWrite(&GpioOutput, 1, c & 0x0FF);
}
// Read the GPIO outputs
Xuint32gpio_read()
{
// Read from the GPIOs
return(XGpio_DiscreteRead(&GpioOutput, 1));
}
```

```
// Clock the LCD (toggles E)
voidlcd_clk()
{
Xuint32 c;
// Get existing outputs
  c = gpio_read();
  delay_us(1);
// Assert clock signal
  gpio_write(c | LCD_E);
  delay_us(1);
// Deassert the clock signal
  gpio_write(c & (~LCD_E));
  delay_us(1);
}
// Assert the RS signal
voidlcd_set_rs()
{
Xuint32 c;
// Get existing outputs
  c = gpio_read();
// Assert RS
  gpio_write(c | LCD_RS);
  delay_us(1);
}
// Deassert the RS signal
voidlcd_reset_rs()
{
Xuint32 c;
// Get existing outputs
  c = gpio_read();
// Assert RS
  gpio_write(c & (~LCD_RS));
  delay_us(1);
}
// Assert the RW signal
voidlcd_set_rw()
{
Xuint32 c;
// Get existing outputs
  c = gpio_read();
// Assert RS
  gpio_write(c | LCD_RW);
  delay_us(1);
}
// Deassert the RW signal
voidlcd_reset_rw()
```

```
{
Xuint32 c;
// Get existing outputs
  c = gpio_read();
// Assert RS
  gpio_write(c & (~LCD_RW));
  delay_us(1);
}
// Write a byte to LCD (4 bit mode)
voidlcd_write(Xuint32 c)
{
Xuint32 temp;
// Get existing outputs
  temp = gpio_read();
  temp = temp & 0xF0;
// Set the high nibble
  temp = temp | ((c >> 4) & 0x0F);
  gpio_write(temp);
// Clock
  lcd_clk();
// Delay for "Write data into internal RAM 43us"
  delay_us(100);
// Set the low nibble
  temp = temp & 0xF0;
  temp = temp | (c & 0x0F);
  gpio_write(temp);
// Clock
  lcd_clk();
// Delay for "Write data into internal RAM 43us"
  delay_us(100);
}
// Clear LCD
voidlcd_clear(void)
{
  lcd_reset_rs();
// Clear LCD
  lcd_write(0x01);
// Delay for "Clear display 1.53ms"
  delay_ms(2);
}
// Write a string to the LCD
voidlcd_puts(constchar * s)
{
  lcd_set_rs();
while(*s)
    lcd_write(*s++);
```

```
}
// Write character to the LCD
voidlcd_putch(Xuint32 c)
{
  lcd_set_rs();
  lcd_write(c);
}
// Change cursor position
// (line = 0 or 1, pos = 0 to 15)
voidlcd_goto(Xuint32 line, Xuint32 pos)
{
  lcd_reset_rs();
  pos = pos & 0x3F;
if(line == 0)
    lcd_write(0x80 | pos);
else
    lcd_write(0xC0 | pos);
}
// Initialize the LCD
voidlcd_init(void)
{
Xuint32 temp;
// Write mode (always)
  lcd_reset_rw();
// Write control bytes
  lcd_reset_rs();
// Delay 15ms
  delay_ms(15);
// Initialize
  temp = gpio_read();
  temp = temp | LCD_DB5;
  gpio_write(temp);
  lcd_clk();
  lcd_clk();
  lcd_clk();
// Delay 15ms
  delay_ms(15);
// Function Set: 4 bit mode, 1/16 duty, 5x8 font, 2 lines
  lcd_write(0x28);
// Display ON/OFF Control: ON
  lcd_write(0x0C);
// Entry Mode Set: Increment (cursor moves forward)
  lcd_write(0x06);
// Clear the display
lcd_clear();
}
```

III.7.8 Bilan d'énergie

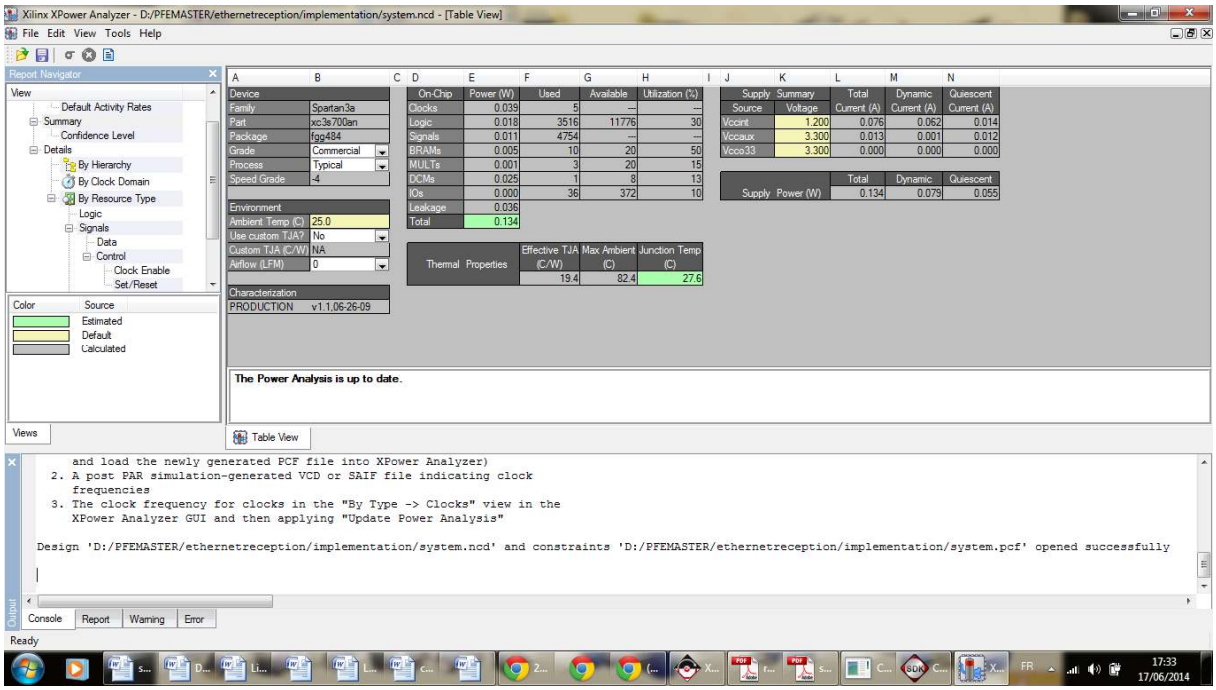


Figure III.17 Bilan d'énergie d'une architecture qui comprend un LCD

III. Bilan des ressources (figureIII.18)

III.8 Conclusion

Dans ce chapitre nous avons utilisé les acquis illustrés dans le chapitre II concernant la configuration et le paramétrage du MicroBlaze, afin de développer des applications et constater pratiquement la caractéristique la plus importante qui rend le travail sur un système embarqué plus intéressant : la flexibilité.

Nous pouvons reprogrammer ajouter ou enlever des périphériques selon les besoins de la tache ce qui va engendrer une optimisation constater sur les ressources.

Un autre avantage de l'utilisation des FPGAs, confirmé durant l'implémentation des commandes c'est la faible consommation d'énergie, de l'ordre de mW et cela malgré la complexité de la fonction implémentée.

Vu l'importance d'avoir une consommation d'énergie réduite dans la conception d'un système embarqué on peut conclure que les FPGAs sont un moyen très pratique.

INTRODUCTION GÉNÉRALE

Les systèmes embarqués nous entourent et nous sommes littéralement envahis par eux, fidèles au poste et prêts à nous rendre service. On en croise des dizaines par jour sans le savoir. Ils sont donc partout, discrets, efficaces et dédiés à ce à quoi ils sont destinés. Et ils seront de plus en plus présents. On parle en effet d'informatique et d'électronique: Ils sont saturés d'électronique plus ou moins complexe et d'informatique plus ou moins évoluée.

Un nombre toujours croissant de nouveaux projets de conception et une complexité grandissante obligent les équipes de conception de systèmes embarqués à être plus efficaces et à mieux déterminer la technologie à utiliser. Pour satisfaire les besoins du marché de la conception des systèmes embarqués et aider les équipes à mettre les produits sur le marché plus rapidement, les fournisseurs de technologie sont en train de développer des composants, des modules ou même des plates-formes embarquées complètes avec des niveaux d'intégration plus élevés et des fonctionnalités accrues. L'utilisation des circuits reconfigurables, pour la conception des systèmes embarqués, offre des possibilités intéressantes d'augmentation des performances et de réduction de la consommation d'énergie et des ressources En effet, ces circuit sont utilisés en complément d'un ou de plusieurs processeurs pour permettre de décharger les calculs intensifs et des traitements de flots de données.

Les circuits FPGA, permettent d'envisager des systèmes beaucoup plus flexibles en offrant notamment la possibilité de l'exécution de blocs de calcul sur la même surface et d'intégrer un grand nombre de ressources matérielles allant de plusieurs microprocesseurs, d'opérateurs mathématiques jusqu'aux portes logiques élémentaires en très grand nombre. Ceci amène à se poser la question sur la façon de concevoir les systèmes pour couvrir l'ensemble ou une partie des besoins une meilleure utilisation des ressources, une puissance réduite, et une bonne vitesse d'exécution. Une possibilité est de partitionner une application en tâches matérielles et logicielles avant la synthèse.

Dans notre projet, et pour notre entrée dans le monde des systèmes embarqués on va développer des applications séparément mais qui serons utile plus tard pour être utilisé dans un système embarqué ces applications utiliserons des périphériques disponibles dans la carte FPGA SPARTAN 3AN STARTER KIT.

PERSPECTIVES

En perspective on peut en premier lieu développer des applications pour les autres périphériques disponibles dans la carte, nous pensons au port VGA pour les applications vidéo et à la fiche audio (Jack) pour les applications audio.

Nous pouvons aussi faire une application sur le port Ethernet, qui va nous permettre de faire la communication avec d'autre équipements sous Protocol TCP/IP, elle nous permettra de mieux exploiter le port Ethernet ca serai intéressant aussi d'envisager un système qui travaille avec toutes ces applications simultanément, ou encore pour embarquer la carte sur un robot pour pouvoir le contrôler, tout ça permettra d'acquérir plus de compétence en matière de système embarqué, mais pour cela il faut envisager de travailler avec une carte plus puissante et plus riche en ressources que la SPARTAN 3AN.

Nous pouvons aussi essayer de développer ou utiliser un module hardware microprocesseur par exemple et l'intégrer à la carte pour permettre d'alléger les calculs gagner en rapidité et surtout économiser les ressources pour d'autre application.

Bibliographie

[1] 'JAVA dans les Systèmes Embarqués et temps réel', techniques de l'ingénieur, Xavier CORNU, S8086-18, 10/06/2004.

[2] [3] [4] Badraddine BEZZA : thèse de magistère intitulé 'Micro processeur à instruction variable', département d'électronique université de Batna, 2010.

[5] [6] ' Architectures reconfigurables FPGA', techniques de l'ingénieur, Olivier SENTIEYS et Arnaud TISSERAND, H1196-20,10/08/2012.

[7]'DSP et temps réel', cours de la haute école spécialisé de Suisse occidentale, M. Correvon.

[8] Messaoudi Kamel : thèse de doctorat intitulé 'traitement des signaux et images en temps réel', université de Bourgogne spécialité Instrumentation et informatique de l'image, 19/12/2012.