République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ECOLE NATIONALE POLYTECHNIQUE
Département de Génie Electrique

Projet de fin d'études

Pour l'obtention du titre
d'Ingénieur d'Etat en automatique

THEME :

SYSTEME TEMPS REEL EMBARQUE POUR COMMANDE D'UN ROBOT MOBILE

Présenté par :

Melle : Farida BOUNKAR

Melle : Sabrina DJELLOULI

Proposé par :

Mr R.SADOUN

Promotion : juin 2005

Ecole Nationale Polytechnique
10, Avenue Hacen Badi, El-Harrach, Alger.

A tout ceux que nous aimons

Remerciements

Nous tenons à remercier vivement Monsieur R.SADOUN chargé de cours à l'Ecole Nationale Polytechnique. En lui exprimant ici toute notre reconnaissance et qu'il veuille accepter l'expression de notre profonde et respectueuse gratitude pour son aide, son soutien, ses directives et conseils judicieux et son suivi régulier, ce qui a permis l'accomplissement de ce modeste travail.

Nous tenons à remercier tout particulièrement, Monsieur STIHI, Professeur à l'Ecole Nationale Polytechnique, d'avoir accepté de présider le jury de notre mémoire.

Nous adressons également nos plus vifs remerciements à Monsieur HADDADI, professeur à l'Ecole Nationale Polytechnique, pour l'honneur qu'il nous a fait en acceptant de juger ce travail.

Nos remerciements vont aussi à tout le corps enseignant de l'Ecole Nationale Polytechnique spécialement les enseignants du département automatique pour leur apport de connaissance durant nos cinq années d'études.

Nos remerciements vont à tout le personnel du Centre de Calcul de l'Ecole Nationale Polytechnique, pour leur aide et encouragements.

Enfin, tous nos remerciements à toutes les personnes qui ont contribué de prés ou de loin à l'aboutissement de ce travail. Nous leurs somme très reconnaissantes.

****

****

**

Résumé

Notre projet consiste à commander un robot mobile en utilisant un système d'exploitation temps réel embarqué sur un processeur virtuel MicroBlaze qui a été configuré sur un circuit FPGA en utilisant l'environnement de développement EDK. Une fois la configuration du circuit est effectuée, nous avons adapté notre carte pour pouvoir la relier au robot après avoir développé le programme en langage C qui gère les différentes tâches tout en satisfaisant le concept du temps réel.

Mots clefs : robot mobile -système embarqué –système temps réel –processeur virtuel – circuit FPGA.

Abstract

This work consists on the use of an embedded real time operating system to control a mobile robot. The OS is ported on an FPGA circuit which is configured with a virtual processor MicroBlaze using the Embedded Development Kit EDK. Once the circuit is configured, we have adapted our card appropriately to be linked with the robot and we have developed the program using the C language to manage the tasks of the robot satisfying the concept of real time.

Key words: mobile robot –embedded system –real time operating system –virtual processor – FPGA circuit.

ملخص:

إن هذا المشروع يتعلق بالتحكم بإنسان آلي باستخدام نظام تشغيل آلي معبأ من فوق الحاسوب الخيالي ميكرو بلا ز المبرمج فوق بطاقة تحمل دارة من نوع FPGA

لقد قمنا بأقلمة البطاقة لكي تربطها بالإنسان الآلي و قمنا بكتابة البرنامج الذي يقوم بإدارة مختلف الأشغال التي يقوم بها الإنسان الآلي المتحرك مع إحترام مبدأ الا نية.

كلمات مفتاح: إنسان آلي متحرك – نظام تشغيل آلي - حاسوب خيالي -دارة FPGA

Sommaire

Introduction générale

I- Introduction générale

Depuis plusieurs décennies, la robotique mobile n'a cessé de connaître des progrès spectaculaires. En effet, les générations de robots mobiles se sont vite succédées offrant à chaque fois plus de performances en matière de perception, d'autonomie et de pouvoir décisionnel. Ces progrès étaient un résultat inévitable des développements qu'ont connu la mécanique, la microélectronique et l'informatique.

L'apparition du concept des systèmes embarqués était à lui seul un événement très important pour les concepteurs de robots mobiles, du moment où il leur donne la possibilité de réduire la taille et l'encombrement de leurs robots et d'en enfouir un système d'exploitation qui se chargera de gérer les différentes tâches du robot mobile lui donnant une certaine autonomie. Cette possibilité d'utiliser un système d'exploitation offre de très grands avantages, notamment la simplicité de programmation pour les développeurs de robots mobiles, puisqu'elle leur permet d'utiliser un langage évolué au lieu de programmer en langage assembleur.

D'un autre point de vue, réaliser un robot mobile « obéissant » implique des exigences en matière de temps de réponse pour lui permettre d'interagir avec son environnement en temps réel. Cependant, l'utilisation d'un système d'exploitation traditionnel temps partagé risque de pénaliser les performances du robot en matière de temps de réponse. Pour cela, il est recommandé d'utiliser un système d'exploitation temps réel qui garantira la réactivité du robot.

Sur le plan matériel, l'apparition des systèmes on chip à savoir les circuits FPGA qui permettent l'utilisation d'un processeur virtuel a révolutionné le monde de la microélectronique et en peu de temps, les SoCs ont réussi à entrer en concurrence avec les microcontrôleurs et à leur en tenir tête en un temps record.

L'utilisation d'un processeur virtuel est l'un des avantages le plus important des circuits FPGA. Ces processeurs nous permettent d'ajouter ou d'enlever des périphériques et des ports d'entrées/sorties selon nos besoins sans avoir à modifier ou à ajouter du matériel qui pourra coûter une fortune.

Le principe « co design » qui caractérise les SoCs nous permet de développer le software en parallèle avec le hardware du moment ou celui-ci est virtuel. Cet atout majeur que les microcontrôleurs ne pourront jamais avoir puisqu'ils sont matériellement figés, est

d'une grande importance pour toutes les conceptions en général et pour la robotique mobile en particulier.

L'objectif principal de notre travail est de remplacer la carte de commande d'un robot mobile existant (***KIFPLOO***) par une carte à base d'un circuit FPGA, d'utiliser le processeur soft MicroBlaze, de réaliser un système embarqué avec un noyau temps réel et de programmer le robot mobile afin d'éviter tout obstacle qui se présentera sur son chemin.

Pour atteindre cet objectif, il était important d'étudier les flots des conceptions matérielle et logicielle de l'environnement de développement EDK, le processeur virtuel MicroBlaze et la compilation d'un noyau temps réel soit sur linux (dans le cas de uClinux), soit sur EDK (dans le cas de Xilkernel).

Ce travail est alors organisé en sept chapitres de la manière suivante :

Le premier chapitre présente certaines généralités concernant la robotique mobile, et en particulier il présente le robot mobile ciblé par notre travail ***KIFPLOO***.

Le second chapitre introduit les systèmes temps réel, il inclut diverses notions de base portantes sur les ordonnanceurs, les priorités des tâches, le concept multitâche, les ressources partagées pour ne citer que les plus importantes

Dans le troisième chapitre, un tour d'horizon des différentes approches qu'on pouvait adopter est donné : la solution microcontrôleur ou la solution SoC, le choix VHDL ou le choix d'un noyau soft, et finalement le choix de MicroBlaze ou de Power Pc.

Le quatrième chapitre traite de la partie développement. Nous y présenterons l'environnement EDK qui permet la configuration et la compilation du matériel parallèlement au logiciel. Nous présenterons aussi le microprocesseur virtuel MicroBlaze et la carte de commande Spartan 3 qui porte le circuit FPGA et qui sera configuré avec MicroBlaze. La dernière partie de ce chapitre a été consacrée à la présentation de deux systèmes d'exploitation qui ont attirés notre attention et qui peuvent être supportés par le processeur MicroBlaze à savoir uClinux et Xilkernel.

Dans le dernier chapitre nous citons les différentes adaptations qu'on a effectué sur la Spartan 3 et sur MicroBlaze, la compilation des deux noyaux temps réels Xilkernel et uClinux, le choix de celui qui sera chargé sur la carte et finalement le développement des différentes tâches du robot mobile et la programmation de ce dernier.

Chapitre I

Generalites sur la robotique

II-1- Introduction à la robotique

Malgré son aspect récent, la robotique tire ses origines des civilisations les plus antiques. Il est peut être même vrai que toutes les autres sciences ont existé juste pour permettre à la robotique de se développer afin d'aider l'Homme à créer cet « esclave » qui appliquera ses ordres au doigt et à l'oeil et qui le libérera à jamais du travail.

De l'Homme préhistorique qui s'empressa d'inventer des extensions technologiques de ses membres afin de faciliter son travail, comme la massue par exemple, jusqu'au robot mobile autonome le plus performant, la robotique n'a cessé de progresser faisant le bonheur des industriels, des scientifiques mais surtout de nombreux fans et rêveurs. En 1997, elle représentait déjà plus de 5.3 milliards de dollars du marché mondial.

Depuis déjà les années soixante, la robotique est devenue une science à part entière, sollicitée énormément par les industriels qui n'hésitent pas à débourser des sommes colossales pour assurer son développement. C'est pour cela qu'en très peu de temps, les générations de robots se succèdent pour voir naître une branche de la robotique visant de plus amples horizons : c'est l'ère de la robotique mobile.

II-1-1-Les robots dans l'histoire [1], [2]

Dans l'imaginaire de l'humanité, les premiers robots, c'est nous. Les tablettes mésopotamiennes déterrées par les archéologues et qui remontent à des millénaires, nous racontent comment Les dieux vivaient entre eux, dans une société inégalitaire.

Le dieu de la technique se proposa alors de fabriquer des pantins mortels : ce sont les ancêtres de l'Homme.

Dans la mythologie grecque, dans son Iliade, Homère (VIIIe siècle av. J.-C.) nous raconte qu'Héphaïstos, dieu de la forge, fut le premier fabriquant de créatures artificielles. Il s'était construit deux servantes en or pour l'assister dans ses travaux. Ces servantes étaient douées d'intelligence et de la parole. Il construisit aussi Talos une statue géante en bronze représentant un guerrier en armure, elle était douée de vie et elle avait pour fonction de défendre le roi Minos.

L'origine de la robotique contemporaine remonte à l'Alexandrie Ptolémaïque. En effet, à partir du VI^{e} siècle av. J.-C, apparaissent des systèmes hydrauliques et pneumatiques, ancêtres des systèmes de régulation d'aujourd'hui.

Parmi les premiers constructeurs d'automates, on peut citer Léonard de Vinci ou encore le mécanicien français Jacques de Vaucanson, dont les tentatives de reproduction des fonctions vitales des êtres humains (circulation, respiration et digestion) au moyen d'automates, notamment le Joueur de flûte traversière (1737) qui a fait sensation au XVIIIe siècle. Nous citons aussi le Tigre automate Tippu's Tiger datant du XVIIIe siècle et retrouvé en Inde.

En 1818, Mary Shelley dans son roman *Frankenstein*, reprit l'idée d'une créature façonnée par l'homme et dotée de vie. C'est l'histoire de Frankenstein, jeune étudiant passionné d'alchimie, qui découvre le secret de la vie. Il assemble les organes qui composeront le corps d'un être humain. Mais lorsque enfin son rêve s'exauce, que l'affreux géant prend vie, son enthousiasme se mue en horreur. Le monstre n'arrivant pas à s'adapter à la société finit par se retourner contre son créateur. Il assassina les parents du jeune scientifique avant de tuer ce dernier. Ce roman est l'un des premiers du genre fantastique. Il met en évidence le thème des dangers mortels de la science et celui de l'orgueil humain qui tente d'égaler Dieu en créant la vie.

II-1-2-Origines du mot « robot » [1], [3]

Étymologiquement, le mot « robot » tire sa racine du bulgare *robu* qui signifie «serviteur» et qui a donné naissance, entre autres, au russe *rabota* qui signifie «travail» et au tchèque *robota* qui se traduit par « travail forcé ». C'est justement l'écrivain tchèque Karel Capek qui a popularisé le terme vers 1920, au travers d'une pièce de théâtre intitulée «Rossum's Universal Robots » qui met en scène des petits automates de forme humaine qui finissent par se rebeller contre leurs inventeurs et dominer le monde.

Le russe Isaac Asimov (1920-1992) père de la science fiction en tant que genre littéraire inventa le mot « robotique ». Il définit en 1942 « les trois lois fondamentales de la robotique » qui visent la protection de l'humanité des risques que pourrait engendrer une très forte « robotisation ».

II-1-3-Définition [4]

Il existe diverses définitions du terme robot, mais elles tournent en général autour de celle-ci :

Un robot est une machine équipée de capacités de perception, de décision et d'action qui lui permettent d'agir de manière autonome dans son environnement en fonction de la perception qu'il en a.

II-2- Les robots modernes [1]

Ce n'est qu'en 1961 que voit le jour le premier robot de l'histoire. Nommé Unimate (universal mate : compagnon universel) c'est le premier bras articulé programmé.

En fait, c'est l'apparition de l'informatique qui a basculé à jamais le monde de la robotique. Jusque là, la mécanique s'était occupée de l'aspect « corps » du robot, l'électronique de l'aspect commande, l'informatique aura pour tache l'aspect « cerveau » du robot.

II-2-1-Première génération de robots [4]

La première génération des robots est les robots programmables et asservis à trajectoire continue ou point à point, dont le cycle de travail se répète sans modification, ils étaient dédiés à une seule tâche ce qui les rendait fortement limités, en plus , ils n'avaient aucune perception de leurs environnements .

II-2-2-Deuxième génération de robots [4]

Les robots de seconde génération ont la caractéristique principale de « la perception de l'état et de la tâche ». Ce sont des manipulateurs automatiques programmables capables d'analyser les modifications de leurs environnements et de réagir en conséquence. Il peut en résulter une modification partielle du cycle opératoire (exemples : manipulation avec reconnaissance de forme, assemblage avec contrôle d'effort, soudage avec suivi de joint...).

II-2-3-Troisième génération de robots [4]

Ce sont des robots utilisant des ressources comme celles de l'intelligence artificielle pour assimiler des instructions globales proches du langage naturel, capables d'une interprétation exhaustive de leur environnement et de prendre des décisions d'action en

conséquence (exemples : robots d'intervention en milieux hostiles, robots autonomes multiservices...).

Les progrès spectaculaires réalisés par la microélectronique et la microinformatique permettent de plus en plus de concrétiser le « fantasme » collectif d'un robot doté de la capacité de raisonner , d'accomplir des tâches tout en mettant an œuvre des relations intelligentes de la perception et de l'action.

II-3- Généralités sur la robotique mobile [4], [5]

De part leurs applications, la plupart des robots sont fixes et leurs environnements parfaitement définis à l'avance. Grâce à la chute des prix et à la fiabilité des composants électroniques, il est maintenant permis de concevoir des robots très différents pouvant réaliser d'autres tâches: c'est l'ère de la robotique mobile.

La robotique mobile regroupe l'ensemble des techniques qui permettent de créer des systèmes dotés de moyens d'acquisition et de traitement d'information, ainsi que de capacités décisionnelles, pouvant accomplir, sous contrôle humain réduit, un certain nombre de tâches, parfois non connues à l'avance, et susceptibles d'évoluer dans le temps. Les robots mobiles fonctionnent en mode télé opéré, commandés à distance par un opérateur, ou en mode autonome, où le robot prend seul ses décisions face à un environnement donné.

II-3-1- Le robot mobile [4], [5]

Le robot mobile est d'abord une base mobile ; à ce titre il peut être utilisé pour le transport. Mais, dés lors qu'il est équipé d'un bras manipulateur ou d'une caméra, ses applications sont plus nombreuses (surveillance, détection d'incendie, etc.).

Les déplacements représentent l'aspect le plus important d'un robot mobile, les outils dont il peut être équipé doivent être considérés comme des robots de type fixe et indépendant dans un premier lieu.

Si l'on souhaite réaliser une étude pour optimiser la trajectoire d'un robot, il est nécessaire de connaître parfaitement à l'avance l'environnement de travail du robot. Le sol, les obstacles dans un milieu restreint doivent être modélisés pour réaliser une planification de la trajectoire. Mais son programme doit inclure une part d'apprentissage. En effet l'analyse

des informations issues des capteurs va lui permettre de créer une base de données qui sera comparée au modèle programmé. Cela permet de corriger les erreurs de lecture des capteurs. Pour tout cela on privilégie d'abord les roues.

Le robot mobile doit donc, obligatoirement pouvoir se repérer dans son mouvement par rapport à son environnement et à l'état de la tâche en cours au moyen de capteur. Ces derniers sont dans la majorité des cas des capteurs de contact ultrasoniques, électromagnétiques ou optoélectroniques.

II-3-2-Les organes sensoriels des robots [5]

Les capteurs sont les organes sensoriels d'un robot. Certains sont fragiles et doivent être protégés, d'autres au contraire doivent être capables d'absorber des chocs. Les plus simples peuvent être directement connectés au centre de contrôle, comme les interrupteurs. Les autres nécessitent une petite interface d'adaptation, comme les capteurs infrarouges ou ultrasons. D'autres plus sophistiqués, nécessitent une carte spéciale et un traitement logiciel qui dépassent les capacités d'une carte à processeur classique. C'est le cas des caméras, couleurs ou pas, qui nécessite un ordinateur pour traiter les informations.

Le choix et l'utilisation des capteurs pour un robot sont un défi à part entière. Les premiers capteurs utilisés par les débutants sont souvent des capteurs simples (interrupteurs faisant office de détecteurs de chocs ou photorésistances pour suivre une source de lumière). Leur principe de fonctionnement est très facile à comprendre. Mais il ne faut pas croire qu'ils peuvent aisément servir d'organe sensoriel au robot.

Pour adapter un capteur à un robot programmable, il faut mesurer les valeurs qu'il fournit, analogiques ou numériques, les améliorer si nécessaire par une interface classique puis convertir l'information par un programme. Le résultat est une information logique sur l'état du capteur ; par exemple, la valeur est inférieure ou supérieure à un seuil choisi. Alors, seulement, on peut décider d'activer le moteur qui peut être associé au capteur.

II-3-3-Différents types de robots mobiles [5]

Il est possible de classifier les robots mobiles de nombreuses manières et selon de nombreux critères tels : le domaine d'application, le degré d'autonomie, le degré d'intelligence. Puisque leur mobilité est l'une de leurs caractéristiques essentielles, nous allons nous intéresser à leur classification selon leur mode de locomotion.

Les robots mobiles peuvent être classifiés selon leur mode de locomotion en cinq catégories :

1. les robots mobiles à roues
2. les robots mobiles à pattes
3. les robots mobiles à chenilles
4. les robots mobiles volants
5. les robots mobiles navigants

II-3-4-Domaine d'application de la robotique mobile

Aujourd'hui, le marché commercial de la robotique mobile est toujours relativement restreint, mais il existe de nombreuses perspectives de développement qui en feront probablement un domaine important dans le futur. Les applications des robots peuvent se trouver dans de nombreuses activités "ennuyeuses, salissantes ou dangereuses" (3 D's en anglais pour Dull, Dirty, Dangerous), mais également pour des applications ludiques ou de service, comme l'assistance aux personnes âgées ou handicapées. Parmi les domaines concernés, citons :

– La robotique de service (hôpital, bureaux)
– La robotique de loisir et la robotique domestique
– La robotique industrielle ou agricole (entrepôts, récolte de productions agricoles, mines)
– La robotique en environnement dangereux (spatial, industriel, militaire)

A cela, s'ajoute à l'heure actuelle des nombreuses plates-formes conçues essentiellement pour les laboratoires de recherche.

II-4- Cas particulier : le robot mobile à commander (KIFPLOO[1])

II-4-1- Présentation générale du robot [5]

Le robot pour lequel on veut développer une carte de contrôle, est un petit robot réalisé en 2004 ayant pour simple fonction d'éviter les différents obstacles qui peuvent se présenter sur son chemin. Pour ce faire, le robot est doté de deux moteurs lui permettant de se mouvoir, d'une structure mécanique qui lui sert de support, de dispositifs de perception et bien évidement d'une carte de contrôle réalisée à base d'un microcontrôleur 68HC11.

Le remplacement de cette dernière avec une autre carte plus performante, assurant au robot le moyen d'effectuer ses tâches plus facilement, est le principal but de notre travail.

II-4-2- La structure mécanique du robot [5]

Notre robot est un robot mobile à roues, sa structure mécanique est constituée principalement d'un châssis circulaire de 20 centimètres de diamètre, servant de support du système de commande et de tout les autres éléments du robot, il est encoché des deux cotés pour permettre aux roues et aux moteurs de s'imbriquer dans la structure circulaire, cette dernière permet au robot de faire des rotations et des semi rotations sur lui-même dans le cas de blocage. Sur les deux cotés du châssis, sont montées deux roues motrices parallèles se faisant face sur l'axe de mouvement, elles sont placées directement sur les arbres des moteurs, pour pouvoir commander très précisément les positions du robot, et elles sont imbriquées dans la structure circulaire afin de protéger le robot d'obstacles non détectés par le système de perception. En addition à ces deux roues, deux autres roues dites folles multidirectionnelles sont placées à l'avant et à l'arrière du robot, ces deux roues n'ont aucun rôle directeur ni aucune influence sur le mouvement du robot, elles ont pour seule fonction d'apporter deux points de contact supplémentaires nécessaires à l'équilibrage de la structure mécanique. Pour fixer les moteurs sur le châssis, deux essieux sont utilisés, ils sont fixés au bas du châssis perpendiculairement à la ligne médiane. Ces essieux permettent une rigidité et une stabilité primordiale des moteurs et ce afin d'assurer que les moteurs et par conséquent les roues soient parallèles et que de ce fait, les mesures de distances parcourues soient précises.

[1] Le nom du robot mobile qu'on va commander

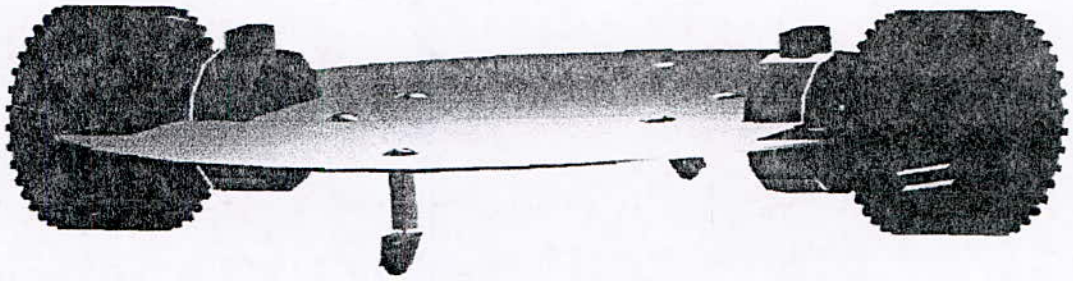


Fig .II-1 : La structure mécanique du robot mobile

II-4-3- Les moteurs utilisés [5]

Afin de mettre en mouvement le robot, deux actionneurs électriques (moteurs électriques) sont placés dans les deux cotés du robot. Les moteurs utilisés sont des moteurs pas à pas à 96 pas par tour et pouvant fonctionner en mode bipolaire ou unipolaire selon le type de commande utilisé, ils offrent en fonctionnement optimal (vitesse de 250 P/S) un couple atteignant 0.045 N.m.

Les moteurs pas à pas sont très utilisés dans la robotique. Ils permettent de convertir une information numérique en un positionnement angulaire ou linéaire. Leur caractère synchrone leur permet de fonctionner sans boucle de retour, contrairement au moteur à courant continu qui nécessite un codeur optique sur leur axe.

Leur prix est plus avantageux qu'un moteur à courant continu avec sa dynamo tachymétrique ou son codeur optique. On en trouve dans les tables traçantes, les imprimantes, les lecteurs de disquettes, et bien sûr dans les robots.

A la différence des moteurs à courant continu, synchrones ou asynchrones qui ont des mouvements où l'angle de rotation varie continûment, le moteur pas à pas est mis en rotation par incréments angulaires. C'est donc un moteur qui tourne en fonction des impulsions électriques reçues dans ses bobinages. Les impulsions électriques sont du type tout ou rien c'est à dire passage de courant ou pas de passage de courant.

II-4-3-1- La commande des moteurs pas à pas [6]

Les moteurs pas à pas peuvent être commandés soit en utilisant une alimentation en tension constante qui est très utilisée pour les faibles vitesses, mais dissipe une puissance assez grande, soit une alimentation en courant constant qui est répondue pour la commande des moteurs bipolaire et permet d'obtenir un grand couple.

Les moteurs utilisés étant des moteurs bipolaires, les deux types de commande sont permis, mais le choix a été porté pour l'utilisation d'une alimentation à courant constant pour minimiser la puissance.

Pour commander les moteurs utilisés dans notre robot, un circuit de puissance basé sur les deux circuits L297/L298 a été réalisé, ces circuits sont les composants les plus utilisés lorsqu'il s'agit de commander les moteurs pas à pas, ils peuvent délivrer un courant allant de quelques dizaines de milliampères à plusieurs ampères.

II-4-3-2- Présentation générale du couple (L297, L298) [6]

Le L297 représente l'étage de commande du moteur pas à pas. Il est utilisé pour la commande des moteurs bipolaires à deux phases et unipolaires à quatre phases. Il est dédié aux applications commandées par un processeur.

Le moteur peut être piloté pour les trois modes de fonctionnement (demi pas, alimentation d'une seule phase à la fois en pas complet, alimentation des deux phases au même temps en pas complet), le hacheur qui y est intégré permet de gérer les commutations du courant dans les enroulement et de le réguler pour ne pas excéder un seuil prédéfini.

L'une des caractéristiques essentielles de ce dispositif est qu'il demande seulement l'injection des signaux suivants : l'horloge, le sens de rotation, le mode de fonctionnement du moteur (demi pas, une seule phase alimentée, deux phases alimentées). Parmi ces avantages nous pouvons aussi citer son encombrement réduit, et le nombre limité des composantes que son assemblage nécessite.

Le L297 peut fonctionner avec d'autres circuits comme le L298N ou le L293E, cela dépend essentiellement du type de moteur et de la valeur du courant traversant les enroulements du moteur.

Compte tenu des caractéristiques du moteur pas à pas utilisé, le L297 de notre carte de commande ne peut être utilisé qu'avec un L298N.

Le L298 est essentiellement composé d'un double pont en H, dont les entrées proviennent du L297. Il possède 15 broches, il accepte les niveaux logiques TTL standards, et peut être utilisé pour piloter des charges inductives comme les relais, les solénoïdes, les moteurs à CC et les moteurs pas à pas. Deux entrées sont fournies pour valider ou non le dispositif indépendamment des signaux d'entrée provenant du L297.

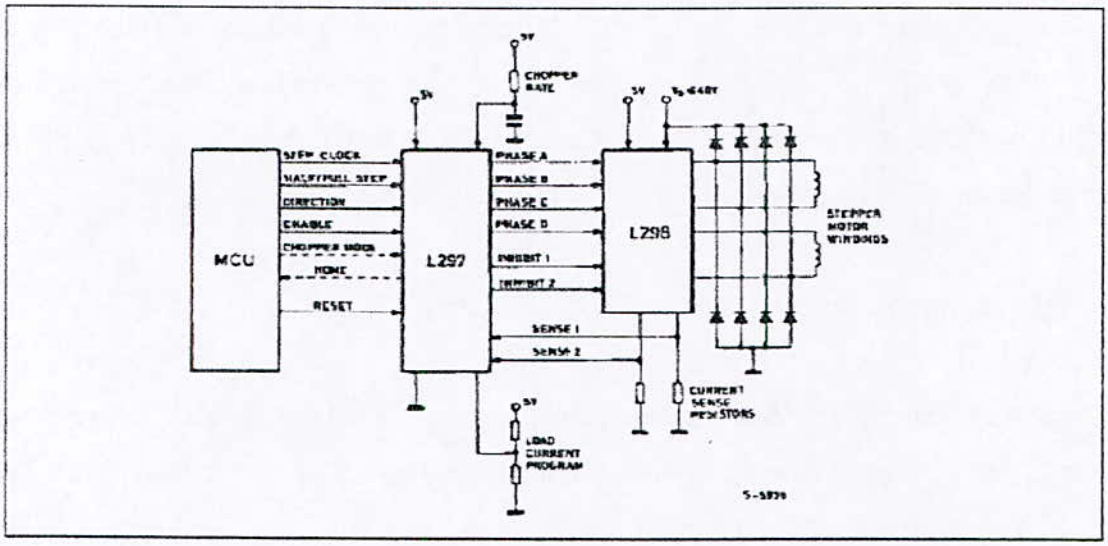


Fig.II-2 : Carte de puissance (association du L297 et du L298)

Voici finalement comment sont réalisées les deux cartes de puissance des deux moteurs :

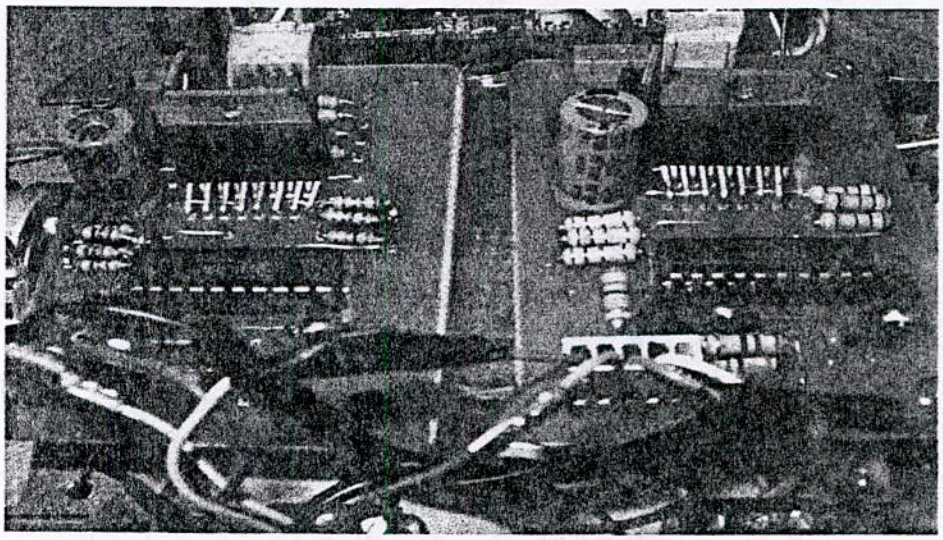


Fig.II-3 : Les deux cartes de puissance montées sur le robot mobile

II-4-4- La perception [5]

Le robot est doté d'un système de perception lui permettant de détecter d'éventuels obstacles et de les éviter. Cette fonction se base sur un dispositif d'émetteurs/récepteurs infrarouges.

Les émetteurs sont deux LEDs infrarouges qui sont disposées de chaque coté du robot et orientées vers l'avant légèrement à l'extérieur. Le récepteur infrarouge a pour caractéristique de ne détecter que les faisceaux infrarouges émis à une fréquence de 38kHz. Un oscillateur 38kHz a été donc ajouté à la conception pour générer un signal à la fréquence désirée. Ce signal est ensuite introduit dans une porte AND où l'on introduit un signal de commande des LEDs.

Les LEDs sont activées l'une après l'autre pour pouvoir déterminer plus précisément la position de l'obstacle. Si une détection se produit alors que la LED à droite est activée, on désactive cette dernière et on active la LED gauche. Si cette dernière rencontre un obstacle, on saura que l'obstacle se trouve droit devant et il prendra les mesures nécessaires pour l'éviter. On agit de la même manière pour détecter des obstacles se trouvant en avant, à droite ou à gauche.

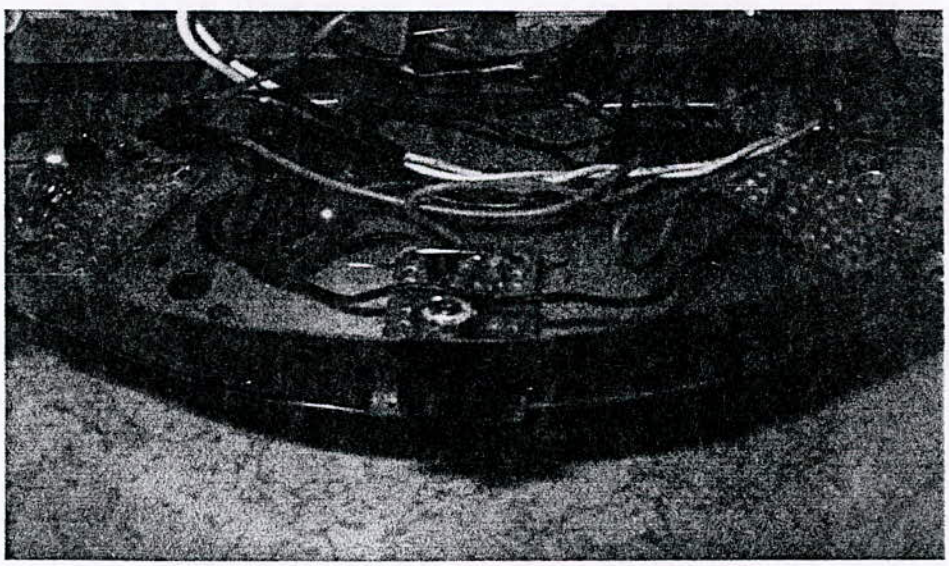


Fig II-4 : Le module de détection IR.

II-5- Conclusion

L'objet de notre étude s'inscrit dans le cadre de la robotique mobile et consiste principalement à l'étude d'une carte de commande pour le robot mobile KIFPLOO. La carte de commande a véritablement constitué un handicap du fait des limitations : espace mémoire réduit et temps de cycle trop grand. L'approche de développement de l'application intégrée était élémentaire et a conduit à une programmation de bas niveau nécessitant une interaction directe avec le matériel. Ce qui limitait le développement d'une application plus complète pour la navigation du robot.

Il nous a été donc proposé de la remplacer par une carte de commande plus adaptée et on a pensé à remplacer la dite carte de commande par une autre basée sur un circuit FPGA Ce qui nous offrira de plus amples possibilités comme l'utilisation d'un système d'exploitation temps réel qui gérera l'interactivité de notre robot avec son environnement.

Chapitre II

Les systèmes temps réel

III-1- Introduction

La gestion du temps est un des problèmes majeurs des systèmes d'exploitations. La raison en est simple : les systèmes d'exploitations modernes sont multitâches, or ils utilisent du matériel basés sur des processeurs qui ne le sont pas, ce qui oblige le système à partager le temps du processeur entre les différentes tâches et à en favoriser certaines d'entre elles selon un ordre prédéfini. Cette notion de partage implique une gestion du passage d'une tâche à une autre, ce qui est effectué par un ensemble d'algorithmes appelé *Ordonnanceur (ou scheduler)*.

En effet, pour de nombreux systèmes, un temps de réponse trop long est fatal. Il est devenu alors impératif d'œuvrer pour incorporer aux systèmes d'exploitations qui les gèrent des algorithmes d'ordonnancement capables de céder la main à la tâche voulue au moment voulu.

Pour ce qui est de notre cas, il est important de pouvoir commander le robot mobile de manière à ce que les tâches les plus prioritaires soient traitées à temps, sinon on risque d'obtenir un robot qui, même ayant reçu un signal des capteurs infra- rouges détectant un obstacle droit devant, continuera à aller dans la même direction. Un tel comportement annulera toutes les possibilités de commande car il rend le robot inobéissant.

III-2- Le temps partagé [9]

U système d'exploitation classique utilise la notion de *temps partagé*, par opposition au *temps réel*. Dans ce type de système, le but de l'Ordonnanceur est de donner à l'utilisateur une impression de confort d'utilisation toute en assurant que toutes les tâches demandées sont finalement exécutées. Ce type d'approche entraîne une grande complexité dans la structure même de l'Ordonnanceur qui doit tenir compte de quelques notions comme la régulation de la charge du système, ou la date depuis laquelle une tâche donnée est en cours d'exécution. De ce fait on peut citer plusieurs limitations par rapport à la gestion du temps :

Tout d'abord, la notion de priorité entre les tâches est peu prise en compte, car l'Ordonnanceur a pour but premier le partage équitable du temps entre les différentes tâches du système (on parle du *quantum du temps*).

Ensuite, les différentes tâches doivent accéder à des ressources dites *partagées*, ce qui entraîne des incertitudes temporelles. Si une des tâches effectue une écriture sur le disque dur,

ce dernier n'est plus disponible pour les autres tâches à un instant donné et le délai de disponibilité du périphérique n'est pas prévisible.

En outre la gestion des entrées /sorites peut générer des temps morts, car une tâche peut être bloquée en attente à un élément d'entrée/sortie.

Enfin, la gestion des *interruptions* reçues par une tâche n'est pas optimisée. Le temps de latence, soit le temps écoulé entre la réception de l'interruption et son traitement, n'est pas garanti par le système.

Toutes ces limitations font que le temps de réponse d'un système classique ne soit pas garanti.

III-3- Le temps réel et les systèmes temps réel [7], [8], [9]

Le temps réel est un concept un peu vague et chacun y va de sa définition. On pourrait le définir comme :

« La possibilité d'exécuter une commande ou une instruction et d'avoir une réponse à un laps de temps prédit relatif à la tâche. Il est aussi définit comme le temps durant lequel une tâche s'exécute ou un événement arrive ».

Un système est dit Temps Réel lorsque:

« L'information après acquisition et traitement reste encore pertinente ».

Cela veut dire que dans le cas d'une information arrivant de façon périodique (sous forme d'une interruption périodique du système), les temps d'acquisition et de traitement doivent rester inférieur à la période de rafraîchissement de cette information.

Comme on peut le définir comme suit :

« *Un système temps réel est une association logiciel/matériel où le logiciel permet, entre autre, une gestion adéquate des ressources matérielles en vu de remplir certaines tâches ou fonctions dans des limites temporelles bien précises ».*

Une confusion classique est de mélanger Temps Réel et rapidité de calcul du système donc puissance du processeur. On entend souvent : " *Etre temps Réel, c'est avoir beaucoup de puissance* ". Dans ce qui va suivre, nous allons voir que ceci n'est pas toujours vrai.

III-4- Les types de systèmes temps réel [9]

On pourra diviser les systèmes temps réel en deux catégories :

- **Les systèmes à contraintes temporelles souples (Soft Real Time)**

  Ces systèmes acceptent des variations dans le traitement des données de l'ordre de la demi seconde ou de la seconde. Dans de tels systèmes, la tâche doit être exécutée le plus vite possible mais elle n'a pas un temps d'exécution fini.

  Le Soft Real Time n'est ni préemptif[1] ni déterministe[2], mais c'est une bonne solution pour plusieurs besoins en temps réel.

  On peut citer l'exemple des systèmes multimédias : si quelques images ne sont pas affichées, cela ne met pas en péril le fonctionnement correct de l'ensemble du système. Ces systèmes se rapprochent fortement des systèmes d'exploitations à temps partagé.

- **Les systèmes à contraintes temporelles fortes (Hard Real Time)**

  Pour ces systèmes, une gestion stricte du temps est nécessaire. Un tel système doit être prévisible. La tâche doit donc être exécutée non seulement correctement mais dans un temps fini bien spécifique. On citera en guise d'exemple les contrôleurs de processus industriels sensibles comme la régulation des centrales nucléaires ou les systèmes embarqués utilisés dans l'aéronautique.

III-5- Fonctionnement d'un système d'exploitation temps réel [10]

Il se base sur les interactions entre le procédé et le système. Tout d'abord, le système d'exploitation temps réel doit réagir aux variations d'état constatées du procédé en lui appliquant les commandes appropriées, ou en signalant cette variation à l'opérateur.

---

[1] *Préemptif :* l'ordonnancement permet à une tâche prioritaire d'anticiper une tâche active moins prioritaire en suspendant son traitement pour libérer une ressource.

[2] *Déterministe* : se réfère à la possibilité de prédire le moment précis de production d'un événement spécifique.

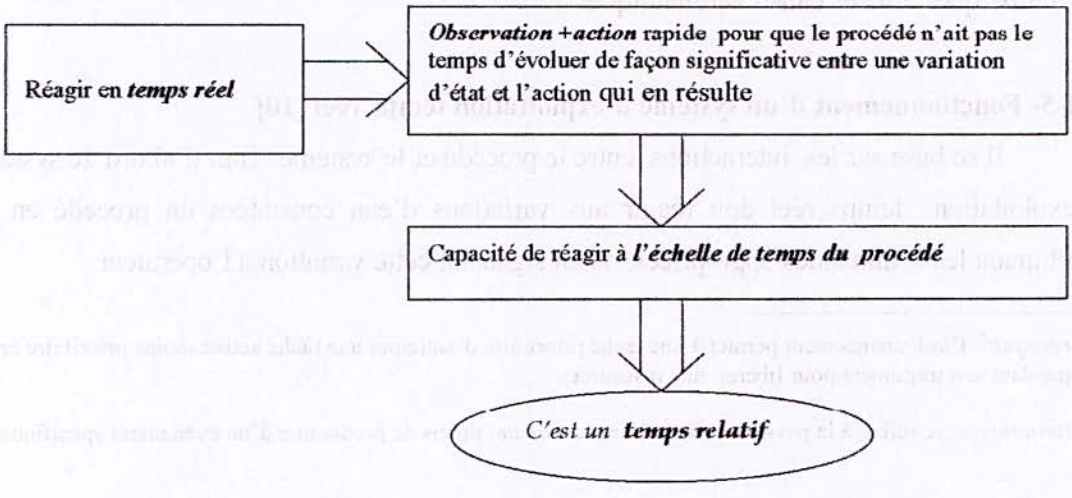
Une forme de «dialogue» s'établit donc entre le procédé et le système informatique de pilotage, et ce via trois types de grandeurs :

Les mesures, les événements que le procédé signale au système de pilotage, mais qui sont souvent intrinsèquement fugaces (c'est-à-dire qui disparaissent sitôt émis), et *la commande.*

Il se base ensuite sur «le temps »; les interactions entre procédé et système doivent être «instantanées ». Fondamentalement, un système temps réel doit être à l'écoute permanente des variations du procédé, et agir en retour aussitôt que nécessaire sur le procédé afin de le contraindre à adopter le comportement souhaité. Ces interactions ne prennent en fait le sens de « temps réel » que si le procédé est actif, c'est-à-dire s'il continue à évoluer en absence de nouvelles commandes. Ce concept s'applique donc essentiellement aux procédés qui possèdent des dynamiques internes.

Or, par principe, le fonctionnement d'un système informatique ne permet qu'une observation échantillonnée de son environnement. Par ailleurs, émettre une nouvelle commande est une opération du système informatique qui ne peut être instantanée car elle requiert un temps de calcul qui peut être conséquent. Or, par principe, le fonctionnement d'un système informatique ne permet qu'une observation échantillonnée de son environnement ; par ailleurs, émettre une nouvelle commande est une opération du système informatique qui ne peut être instantanée car elle requiert un temps de calcul qui peut être conséquent. Pour un système de pilotage, réagir « dans les temps » implique donc une observation et une action suffisamment rapides pour que le procédé n'ait pas le temps d'évoluer de façon significative entre une variation d'état et l'action qui en résulte. Le temps réel reflète donc une capacité de réaction à l'échelle de temps du procédé : c'est un temps relatif.

Ceci peut être résumé de la manière suivante :



III-6- Domaines d'application des systèmes d'exploitations temps réel [10]

Les domaines industriels d'application qui s'ouvrent à l'informatique « temps réel » peuvent être sommairement distingués ainsi :

- Les systèmes embarqués dans des équipements de haute technologie.
- Les systèmes embarqués produits en très grande quantité
- Les systèmes à l'architecture matérielle ou logicielle dynamique.

III-7- Concepts relatifs aux systèmes temps réel [8], [11]

III-7- 1-La section de code critique (région critique)

C'est un code indivisible, c'est-à-dire qu'une fois son exécution lancée, elle ne doit pas être interrompue (avant de lancer l'exécution du code critique il faut mettre les interruptions invalides et puis les valider une fois le code critique est terminé).

III-7-2-Les ressources

Une ressource est toute entité utilisée par une tâche, elle peut donc être des dispositifs d'entrée sortie, tel qu'une imprimante, un clavier, etc.

III-7-3-Ressources partagées

Ce sont des ressources qui peuvent être utilisées par plusieurs tâches. Chaque tâche peut avoir un accès exclusif à la ressource pour éviter la corruption des données, ce qui est appelé *exclusion mutuelle.*

III-7- 4-Le multitâche

C'est le processus de l'ordonnancement de la CPU[3]. La CPU se partage entre les tâches. L'aspect le plus important d'un système multitâches est qu'il permet aux programmeurs d'application de gérer les applications en temps réel.

[3] CPU central processing unit

III-7- 5-L'ordonnanceur de tâches (scheduler)

III-7- 5-1-Définition d'une tâche

Du point de vue du microprocesseur, une tâche est une activité qui consomme des ressources de la machine informatique. Ces ressources sont :

- De la mémoire (une tâche a besoin de RAM pour tourner)
- Du temps CPU (le temps que la tâche s'exécute les autres n'ont pas le temps CPU)

A tout instant une tâche à un état. On dit que la tâche est :

- *Active* (running) c'est le cas de la tâche qui est entrain de s'exécuter. Quand il n'y a qu'un processeur, une seule tâche est active à un moment donné, pas forcement la plus prioritaire.
- *Prête* (ready) c'est le cas des tâches qui sont en attente de pouvoir s'exécuter. La priorité de ces tâches est inférieure ou égale à celle de la tâche active.
- *Endormie* (asleep) : c'est le cas des tâches qui sont en attente d'événements qui provoqueront leurs réveils (interruptions, réception de messages ou fin du décompte d'un timer). Une tâche plus prioritaire que la tâche active peut être dans cet état.

Les tâches sont parfois appelées process ou processus.

III-7- 5-2-Définition d'un Ordonnanceur de tâches

Un Ordonnanceur de tâches, scheduler, ou superviseur a pour rôle de faire basculer le traitement d'une activité sur une autre. C'est lui qui gère le temps CPU, en faisant la distinction entre le *temps absolu*, référence à une origine connue de toute les tâches et le *temps relatif* que l'on peut assimiler à un délai. C'est l'Ordonnanceur de tâches qui prend la décision d'allouer la ressource « temps CPU » à une activité plutôt qu'à une autre (d'activer une tâche ou non). Cela revient à faire changer l'état de deux tâches :

- Faire passer l'état de la tache d' « active » à « prête » ou « endormie ».
- Faire passer l'état de la tache de « prête » ou « endormie » à « active ».

Le temps de basculement de contexte (latency time) est un facteur de qualité du scheduler.

III-7- 5-3-Mode de fonctionnement d'un Ordonnanceur de tâches

Il existe deux possibilités :

- Le système est mono tâche; dans ce cas, le temps CPU est alloué en permanence à une seule activité, ce cas revient au fonctionnement sans scheduler
- Le système est multitâche; dans ce cas il y a plusieurs modes de fonctionnement :

a)-Fonctionnement en temps partagé (time sharing)

Dans ce cas, le scheduler est capable de gérer plusieurs activités et alloue le temps CPU à une tâche pour un temps limité appelé *quantum de temps*. Au bout de ce temps le scheduler reprend la main et lance l'activité de la tâche suivante. On peut donc dire que dans ce mode de fonctionnement chaque tâche à un temps d'allocation limité qui est uniforme pour toute les tâches, et que ces dernières sont exécutées à tour de rôle.

b)-Fonctionnement basé sur la priorité (priority based)

Dans ce cas, le scheduler est capable de gérer plusieurs activités et alloue le temps CPU a une tâche pour un temps infini (hors interruption plus prioritaire). Par contre, à chaque appel système effectué par l'activité en cours, la maîtrise du temps est reprise par le scheduler. Celui ci décide alors selon les priorités qui ont été données par l'utilisateur aux différentes activités, laquelle des activités va reprendre le temps CPU.

c)-Fonctionnement en tourniquet (round robbin)

Dans ce cas, le scheduler est capable de gérer plusieurs activités et alloue le temps CPU à une tâche pour un temps infini, tant qu'une tâche de priorité plus grande ne devient pas prête, comme dans un fonctionnement basé sur la priorité mais fonctionne en temps partagé à l'intérieur d'un même niveau de priorité. L'Ordonnanceur considère deux classes de niveau de priorités : une classe où le fonctionnement est basé sur la priorité et une autre classe où le fonctionnement est en temps partagé.

Il est évident que seul le fonctionnement basé sur la priorité pourra assurer le temps réel, puisque c'est toujours la tâche la plus prioritaire qui tourne, hormis celles qui sont en attente d'un événement qui n'est pas arrivé.

III-7- 6-Basculement d'une tâche à l'autre (context switch)

Quand un événement active une tâche plus prioritaire que celle qui s'exécute, le noyau sauvegarde cette dernière dans la pile et exécute la nouvelle tâche.

Le temps nécessaire pour exécuter un changement de tâche dépend du nombre des registres CPU devant être sauvegardés et restitués, mais la performance du noyau temps réel n'est pas jugée par le nombre des changements de tâches qu'il peut faire par seconde.

III-7- 7-Le noyau

Le noyau est la partie principale d'un système d'exploitation. C'est la partie du système multitâches responsable de la gestion des tâches (gestion du temps de la CPU) et de la communication entre les tâches. Le service fondamental fourni par le noyau est le context switch.

L'utilisation d'un noyau temps réel généralement simplifie la conception des systèmes en divisant les applications en plusieurs tâches gérées par le noyau.

Chaque tâche nécessite un espace propre à elle dans la pile, ce qui prend trop d'espace mémoire. Ainsi un noyau à temps réel a besoin d'une RAM et d'une ROM. Par conséquent un microcontrôleur ayant une petite RAM ne peut pas travailler avec un noyau temps réel.

Le noyau permet de mieux utiliser la CPU en fournissant de nombreux services tels que gérer le sémaphore, les délais, etc.

III-7-7-1-Le noyau non préemptif

Dans le noyau non préemptif, toute tâche doit être complètement effectuée avant de céder le temps CPU. Même si une interruption met une tâche prioritaire dans son état prête, le noyau doit toujours retourner à la tâche active jusqu'à ce qu'elle soit achevée, puis il exécute la tâche la plus prioritaire en attente.

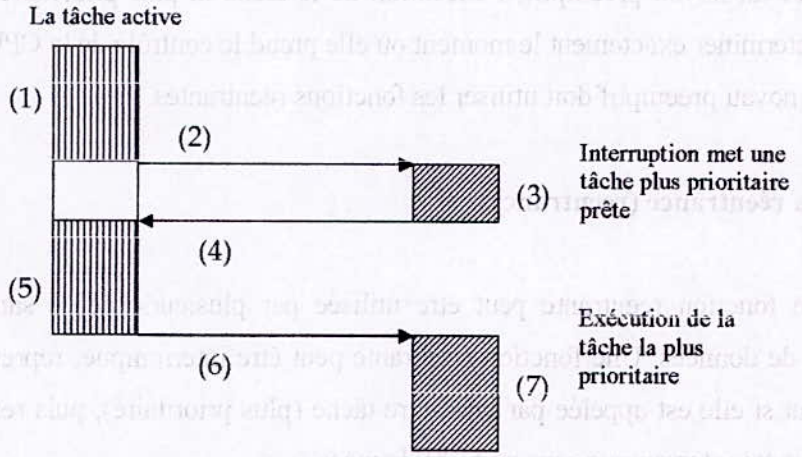


Fig.III-1 : Fonctionnement d'un noyau non préemptif

Le temps de réponse d'un noyau non préemptif n'est pas déterministe, même si une tâche est prioritaire, on ne peut pas savoir le moment exacte de son exécution.

III-7-7-2-Le noyau préemptif

La plus prioritaire tâche prête à être exécutée prend le contrôle de la CPU dans un noyau préemptif. Quand une interruption met une tâche plus prioritaire dans son état prête, l'exécution de la tâche active est suspendue et c'est la tâche prioritaire qui prend le contrôle de la CPU.

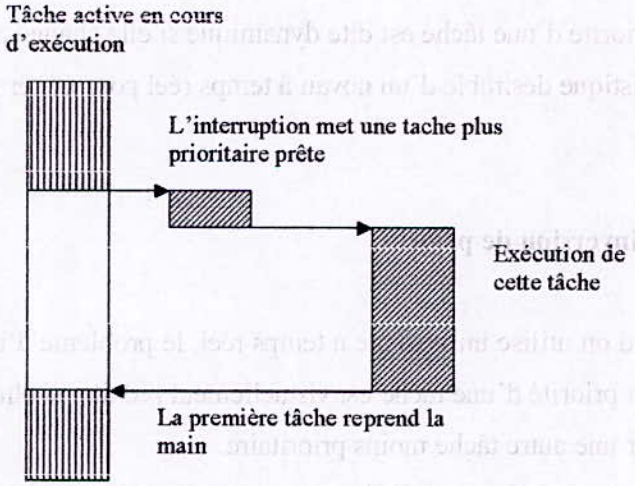


Fig.III-2 : fonctionnement d'un noyau préemptif

Avec un noyau préemptif, l'exécution de la tâche la plus prioritaire est déterministe (on peut déterminer exactement le moment où elle prend le contrôle de la CPU.

Un noyau préemptif doit utiliser les fonctions réentrantes.

III-7- 8-La réentrance (reentrancy)

Une fonction réentrante peut être utilisée par plusieurs tâches sans avoir peur de corruption de données. Une fonction réentrante peut être interrompue, reprend son exécution dés le début si elle est appelée par une autre tâche (plus prioritaire), puis revenir à l'endroit où elle avait été interrompue sans perte de données.

Les variables d'une fonction réentrante sont systématiquement localisées sur la pile, si non, si des variables globales sont utilisées on doit utiliser des variables protégées.
Dans un noyau préemptif, il faut éviter d'utiliser des fonctions non réentrantes.
On peut rendre une fonction non réentrante, réentrante en utilisant l'une des techniques suivantes :

- Déclarer des variables locales.
- Invalider les interruptions avant la fonction et les valider après.
- Utiliser les sémaphores.

III-7- 9-Les priorités dynamiques

La priorité d'une tâche est dite dynamique si elle change au cours de l'exécution. C'est une caractéristique désirable d'un noyau à temps réel pour éviter les inversions de priorités.

III-7- 10-L'inversion de priorité

Quand on utilise un système à temps réel, le problème d'inversion de priorité peut être rencontré. La priorité d'une tâche est virtuellement réduite, si elle attend une ressource qui est déjà prise par une autre tâche moins prioritaire.

On peut résoudre ce problème en augmentant la priorité de la tâche moins prioritaire possédant la ressource jusqu'à ce qu'elle la libère, puis la faire revenir a son premier niveau de priorité. Permettant ainsi à la tâche utilisant la ressource d'avoir une priorité plus grande que celles de toutes les autres tâches concourantes pour avoir cette même ressource.

Pour pouvoir faire ces changements de priorités, un noyau qui change automatiquement ces priorités est nécessaire, ce changement automatique est appelé : priority inheretance.

III-7-11-Assigner des priorités aux tâches

Assigner un ordre de priorité aux tâches n'est pas une opération triviale à cause de la nature complexe des systèmes temps réel. En effet, dans la majorité des systèmes, toutes les tâches ne sont pas critiques, les tâches non critiques ont une priorité basse. Ajouté à cela la majorité des systèmes sont une combinaison des exigences Soft et Hard.

Une des techniques les plus intéressantes pour assigner un ordre de priorité aux tâches est la méthode « RMS » (rate monotonic scheduling), cette méthode est basée sur le nombre de fois que la tâche s'exécute : les tâches à grande ration d'exécution ont la plus grande priorité.

L'une des critiques les plus importantes de cette méthode est que la tâche la plus répétée n'est pas toujours la plus importante. Dans ce cas c'est l'application qui dicte comment ordonnancer les tâches. Mais n'empêche que la méthode RMS reste un bon début.

III-7-12-L'exclusion mutuelle

Le moyen le plus facile de communiquer entre les tâches est l'utilisation des structures de données partagées, mais il faut s'assurer que chaque donnée a un accès exclusif aux données partagées pour éviter la corruption de ces dernières.

Les méthodes les plus courantes pour obtenir un accès exclusif aux ressources partagées sont :

- Invalider les interruptions.
- Exécution des opérations TEST et SET.
- Annuler l'ordonnancement.
- Utiliser les sémaphores.

III-7- 12-1-Invalider les interruptions

C'est le moyen le plus rapide pour avoir accès à une ressource partagée, cependant il ne faut pas invalider les interruptions très longtemps pour éviter d'écraser des demandes

d'interruptions vitales pour le système (une demande d'interruption relative à la sécurité du système par exemple). Pour ce, il existe des noyaux qui permettent de connaître combien de temps les interruptions sont annulées.

III-7- 12-2-L'exécution des opérations TEST et SET

Une variable globale à la valeur 0 quand aucune tâche n'utilise la ressource, et elle se met à 1 quand une tâche l'utilise du coup les autres tâches n'en ont plus l'accès.

III-7- 12-3-Invalider l'Ordonnanceur

Cette méthode est utilisée lorsque la tâche ne partage pas des données avec l' ISR[4]. Notons, toutefois que lorsque l'Ordonnanceur est invalide les interruptions restent toujours valides : si une interruption surgit, elle est aussitôt exécutée, à la fin, le noyau revient à la tâche interrompue même si une autre tâche plus prioritaire est prête.

III-7- 13-Les sémaphores

Les sémaphores sont utilisés pour :

- Contrôler l'accès aux ressources partagées.
- Signaler l'arrivée d'un événement.
- Permettre à deux taches de synchroniser leurs activités.

Le sémaphore peut être défini comme étant :
«Une clef que le code donne à la tâche afin de continuer son exécution. »
Si un sémaphore est déjà pris, la demande de la tâche est suspendue jusqu'à ce qu'il soit libre.

Il y a deux types sémaphores :

- Binaires [5]
- Compteurs (counting)[6]

[4] Interrupt servive request : service de demande d'interruption
[5] Valeur logique : 0 ou 1.
[6] Utilisée lorsque la ressource peut être utilisée par plus d'une tache à la fois , sa valeur est entre 0 et 255 dans le cas d'un noyau à 8 bit .

Une tâche désirant un sémaphore doit exécuter une demande d'accès au sémaphore. Si Ce dernier est disponible (sa valeur est supérieur à zéro), l'accès lui est accordé, la valeur du sémaphore est alors décrémentée, et la tâche continue son exécution.

Si lors de la demande d'accès au sémaphore la valeur de celui-ci est égale à zéro (sémaphore utilisé par une autre tâche), la tâche qui exécute la demande sera placée dans une liste d'attente.

Ils existe des noyaux qui spécifient un timeout : si le sémaphore n'est pas disponible, au bout d'un certain temps, la tâche va continuer son exécution et afficher à la fin un code d'erreur.

Lorsqu'une tâche libère un sémaphore, et que la liste d'attente est vide, la valeur du sémaphore est incrémentée. Cependant, si une tâche attend l'accès au sémaphore, sa valeur n'est pas incrémentée. Le sémaphore est donné directement à l'une des tâches l'attendant. Dépendamment du noyau utilisé, la tâche qui recevra le sémaphore est soit :

- La tâche de plus haute priorité qui attend l'accès.
- La première tâche à avoir demander l'accès (FIFO[7])

De ce fait L'importance de la connaissance de l'existence d'un sémaphore est accrue. Dans certain cas, une tâche passe un sémaphore sans se rendre compte, dans ce cas le sémaphore est encapsulé dans une fonction qui a l'aspect normal.

III-8- Conclusion

Ainsi, compte tenu de toutes ces notions, nous pouvons d'ores et déjà prévoir quelques unes des caractéristiques du système temps réel pour lequel on optera. Notons tout de même la diversité des choix qui nous seront proposés, vu le nombre croissant de développeurs qui ne cessent de faire le bonheur des industriels et des amateurs des systèmes temps réels.

Dans notre cas, l'utilisation d'un système d'exploitation temps réel embarqué pourrait à la fois assurer l'autonomie du robot mobile et améliorer ses performances en matière de temps de réponse.

[7] First in first out.

Chapitre III

Choix de la plateforme materielle

IV-1- Introduction

Les progrès technologiques continus dans le domaine de la microéléctonique ont permis de nous offrir plusieurs alternatives concernant le choix de la plateforme matérielle qui accueillera le système d'exploitation temps réel pour lequel on aurait opté.

En effet, depuis l'apparition des circuits logiques programmables vers la fin des années 80, et leur évolution spectaculaire en matière de performances, de capacités d'intégration, et de logiciels de programmation, ils n'ont cessé d'être sollicités pour de nombreuses applications dans divers domaines.

Ces circuits, et plus spécialement les circuits de type FPGA sont devenus alors, la solution matérielle idéale pour de nombreux systèmes embarqués, qui souffraient jusque la des coûts de réalisation élevés et de conception bien figée que leur infligeaient les microcontrôleurs.

Malgré le fait que la solution dite *On Chip* (ASIC et circuits programmables) soit de plus en plus ''tendance'', les microcontrôleurs restent les plus convenables pour les petits systèmes embarqués statiques ne nécessitant pas l'ajout d'autres modules ou périphériques à la conception de base.

Dans ce qui suit, nous allons étudier plusieurs solutions technologiques qui peuvent être utilisées, afin d'opter pour celle qui nous convient le plus.

IV-2- Différentes approches possibles pour un système logique [12]

Elles consistent à choisir entre l'utilisation de la logique standard (le choix microcontrôleur), et l'utilisation de la solution On Chip qui elle-même offre la possibilité de choisir entre les ASICs (spécifiques à l'application pour lequel ils ont été conçu), et les PLDs (logique programmable) et plus précisément les FPGAs.

IV-2-1- La Logique standard [12], [13]

Le microcontrôleur est un circuit intégré dédié aux applications embarquées. A la différence du microprocesseur, sa puce comporte un certain nombre d'interfaces, une petite RAM et PROM, des timers et des ports d'entrée/sortie.

En général, la logique standard se caractérise par : une disponibilité immédiate, un faible coût, une fonction figée par le constructeur, et une faible intégration.

Il est vrai que l'aspect Standard des microcontrôleurs favorise leur utilisation dans tout type d'application, mais il peut être très pénalisant pour ce qui est des performances. Ce qui limite leur utilisation aux réalisations embarquées amateurs et aux conceptions de moindre qualité et coût.

IV-2-2-La solution On Chip [13]

Un system en SOC[1] est un dispositif caractérisé par l'intégration de toutes les parties du système informatique en un seul circuit. En effet, cela permet de réduire :

- La dissipation de l'énergie.
- Les interconnections.
- La taille du dispositif.

Un system en SOC typique contient un ou plusieurs noyaux de processeurs, un nombre arbitraire de périphériques, une mémoire on chip, et un bus qui interconnecte tout ces dispositifs.

Le but de la conception d'un system on chip est d'utiliser uniquement un seul circuit pour l'application.

En pratique un système on chip peut aussi contenir une multitude d'interfaces entrées/sorties vers d'autres circuits (périphériques Off Chip).

Un système On Chip travaille généralement à une fréquence d'horloge beaucoup plus faible qu'une CPU standard. En effet, cette dernière nécessite une fréquence entre 500 MHZ et 3GHZ, tandis qu'une CPU SOC nécessite uniquement quelque mégahertz.

Une CPU SOC idéale travaille à la fréquence d'horloge minimale pour réaliser la tâche désirée.

Par l'utilisation d'une faible fréquence, la consommation de l'énergie et la température du circuit sont réduites, Ce qui réduit les besoins en refroidissement et en énergie.

[1] Solution on chip.

IV-2-2-a-Les ASICs (Application Specific Integrated Circuit) [13]

Par définition, les circuits ASIC regroupent tous les circuits dont la fonction peut être ***personnalisée*** d'une manière ou d'une autre en vue d'une application spécifique, par opposition aux circuits standards dont la fonction est définie et parfaitement décrite dans le catalogue des composants.

Les ASIC peuvent être classés en plusieurs catégories selon leur niveau d'intégration, en fait un ASIC est défini par sa structure de base (réseau programmable, cellule de base, matrice, etc.). Sous le terme ASIC deux familles sont regroupées, les semi personnalisés (avec des réseaux prédéfinis) et les personnalisés (non préfabriqué, qu'on optimise pour créer son propre composant).

IV-2-2-a-1- Avantages et inconvénients de l'utilisation d'ASIC

D'une manière générale l'utilisation d'un ASIC conduit à de nombreux avantages provenant essentiellement de la réduction de la taille des systèmes. Il en ressort :

- Réduction du nombre des composants sur le circuit imprimé. La consommation et l'encombrement s'en trouvent considérablement réduits.
- Le concept ASIC par définition assure une optimisation maximale du circuit à réaliser. Nous disposons alors d'un circuit intégré correspondant réellement à nos propres besoins.
- La personnalisation du circuit donne une confidentialité au concepteur et une protection industrielle.
- Enfin, ce type de composant augmente la complexité du circuit, sa vitesse de fonctionnement et sa fiabilité.

Dans l'approche des circuits du type ASIC, l'inconvénient majeur réside dans le fait du *passage obligatoire chez le fondeur* ce qui implique des frais de développement élevés du circuit.

Les ASICs sont généralement plus convenables pour les productions en série de conceptions déjà vérifiées et non pour les prototypes.

IV-2-2-b-Les PLD (Programmable Logic Device) [13]

Ce sont des chips qui peuvent être programmés pour se comporter comme une conception arbitraire. Un PLD peut être programmé pour une implémentation aussi simple qu'une opération en logique combinatoire comme il pourrait l'être pour des conceptions beaucoup plus importantes.

IV-2-2-c- Les FPGAs (Field programmable gate array) [13]

Ce sont des circuits de type PLD. Un FPGA possède une architecture générique qui consiste en un ensemble de blocs logiques et séquentiels configurables et d'un ensemble d'interconnections programmables.

Les circuits FPGAs ne sont pas optimisés pour une application bien déterminée, par conséquent ils consomment plus d'énergie que les ASICs. Par contre ils sont beaucoup plus simples à programmer et à reprogrammer, ce qui raccourcit les cycles de conception et permet de suivre l'évolution de l'application pour laquelle, ils ont été conçus.

Les FPGAs sont plus convenables pour les prototypes et pour les productions en série limitées qui ne sont pas de la qualité des ASICs.

IV-3- Notre choix

Dans le cas de notre robot mobile, l'expérience d'utiliser un microcontrôleur programmé en assembleur, a donné des résultats acceptables, mais la possibilité de réaliser un système embarqué à base de cette même carte était impossible à cause de l'insuffisance de l'espace mémoire.

Utiliser une autre carte à base d'un microcontrôleur plus performant pourrait être une solution aux problèmes de mémoire, mais cette solution ne nous permet guère d'avoir la liberté de modifier le hardware, et d'ajouter d'autres périphériques sans passer par la soudure.

Séduits par les diverses possibilités qu'offre le choix d'une carte à base de circuit de type FPGA, nous avons tout de suite opté pour cette solution.

Le circuit FPGA que nous avons utilisé est un FPGA du constructeur Xilinx, implanté sur une carte Spartan 3.

IV-4- Les FPGA de Xilinx

IV-4-1- Architecture générale

Il s'agit d'une matrice carrée de cellules configurables pouvant être connectées entre elles par un réseau d'interconnexions ; la liaison vers l'extérieur du circuit se fait par des blocs d'entrées/sorties configurables en niveau logique, en impédance et en direction.
L'architecture, retenue par Xilinx, se présente sous forme de deux couches :

- une couche appelée circuit configurable,
- une couche réseau mémoire SRAM[2].

La couche dite 'circuit configurable' est constituée d'une matrice de blocs logiques configurables **CLB** permettant de réaliser des fonctions combinatoires et des fonctions séquentielles. Tout autour de ces blocs logiques configurables, nous trouvons des blocs entrées/sorties **IOB** dont le rôle est de gérer les entrées-sorties réalisant l'interface avec les modules extérieurs.
La figure (IV-1) présente l'architecture générale des circuits FPGA.

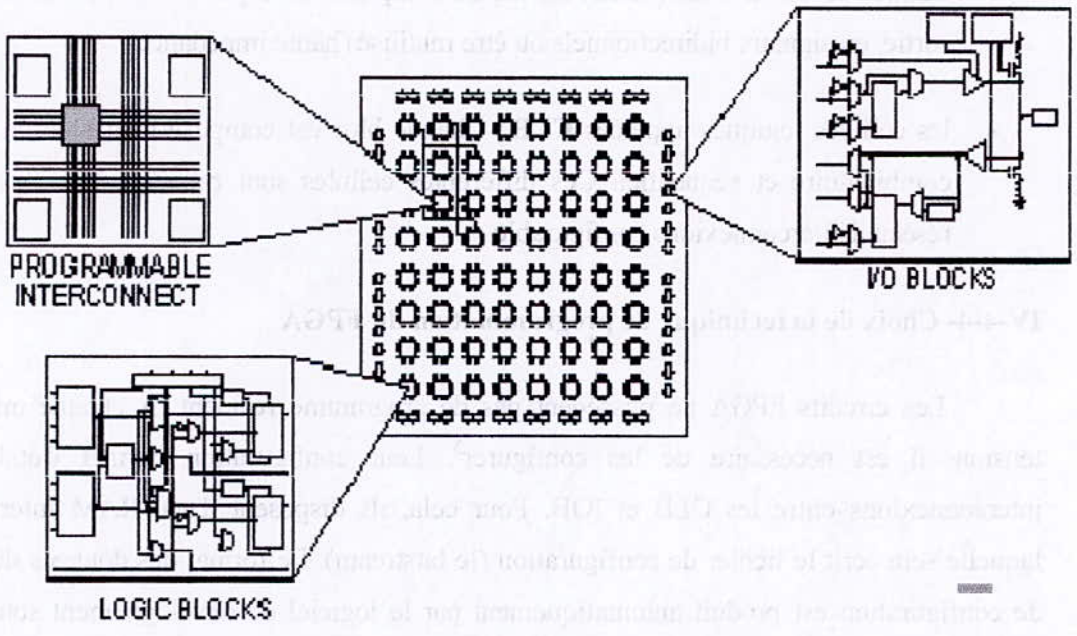


Fig.IV-1 : Architecture interne du FPGA

[2] ***static random access memory***
C'est un type de mémoire qui est plus rapide et plus fiable que la DRAM (dynamic random acsess memory).

IV-4-2- Configuration du circuit FPGA

La configuration du circuit est mémorisée sur la couche réseau SRAM et stockée dans une ROM externe. Un dispositif interne permet à chaque mise sous tension de charger la SRAM interne à partir de la ROM. Ainsi on conçoit aisément qu'un même circuit peut être exploité successivement avec des contenus différents de la ROM, puisque sa programmation interne n'est jamais définitive. On voit tout le parti que l'on peut tirer de cette souplesse en particulier lors d'une phase de mise au point. Une erreur n'est pas inacceptable, puisqu'elle peut aisément être réparée.

IV-4-3-Les composants de base du circuit FPGA

Les circuits FPGA du fabricant Xilinx utilisent deux types de cellules de base :

- Les cellules d'entrées/sorties appelés **IOB** ; Ces blocs entrée/sortie permettent l'interfaçage entre les broches du composant FPGA et la logique interne développée à l'intérieur du composant. Ils sont présents sur toute la périphérie du circuit FPGA. Chaque bloc IOB contrôle une broche du composant et il peut être défini en entrée, en sortie, en signaux bidirectionnels ou être inutilisé (haute impédance).

- les cellules logiques appelées **CLB** ; Chaque bloc est composé d'un bloc de logique combinatoire et séquentiel. Ces différentes cellules sont reliées entre elles par un réseau d'interconnexions configurable.

IV-4-4- Choix de la technique de programmation du FPGA

Les circuits FPGA ne possèdent pas de programme résident. A chaque mise sous tension, il est nécessaire de les configurer[3]. Leur configuration permet d'établir des interconnexions entre les CLB et IOB. Pour cela, ils disposent d'une RAM interne dans laquelle sera écrit le fichier de configuration (le bitstream). Le format des données du fichier de configuration est produit automatiquement par le logiciel de développement sous forme d'un ensemble de bits organisés en champs de données.

[3] Cela n'empêche nullement la possibilité de sauvegarder le fichier de configuration sur une mémoire ROM (une Flash par exemple) qui résiderait on chip.

Il existe diverses approches pour programmer un circuit FPGA, nous avons opté pour la programmation en mode dit esclave dans lequel, le programme de configuration est envoyé à partir d'un PC, d'une station de travail ou à partir d'un autre circuit FPGA. Pour notre cas, il s'agit d'envoyer le fichier de configuration à partir d'un PC et via un câble JTAG[4].

IV-4-5- Choix des outils de développement des FPGA

Xilinx a développé des logiciels performants capables de configurer le circuit FPGA et de simuler son fonctionnement.

Les outils de développement Xilinx qu'on possède nous offrent la possibilité de choisir entre deux approches différentes pour générer notre fichier de configuration (bitstream) :

- Choisir d'utiliser un langage de description de matériel (le VHDL par exemple), cela impliquerait l'utilisation de l'outil de développement ISE de Xilinx.
- Ou bien choisir d'utiliser un processeur soft prédéfinit dans l'environnement EDK de Xilinx.

IV-4-5-1-La synthèse en VHDL [13]

Le langage VHDL *(Very High Speed Integrated Circuit Hardware Description Language (VHDL))* est un langage de description du matériel qui permet de synthétiser des fonctions logiques complexes. Un usage commun du VHDL est de décrire comment un matériel doit être implémenté. De telles descriptions peuvent être traduites en une Netlist qui décrit comment sont interconnectés et configurés les blocs formant le circuit FPGA.

IV-4-5-2-L'utilisation d'un noyau soft [13]

Selon un jargon utilisé par les fabricants d'ASICs, les noyaux sont classés en trois catégories :

- Les noyaux ***Hard***, qui ont une disposition physique (*layout*) fixe, et qui sont incorporé à la conception en tant qu'une puce standard.

[4] Acronyme anglais pour *Joint Test Advisory Group*. La norme JTAG permet de tester les parties numériques des systèmes, cartes électroniques et ASICs.

- **Les noyaux Firm,** qui sont délivrés en tant qu'élément de librairie.
- **Les noyaux Soft**, qui sont fournis sous forme de Netlist, ou bien d'un code source en HDL.

Les noyaux Soft ont récemment gagnés beaucoup de popularité spécialement parmi les développeurs des FPGAs. Cela est du à divers facteurs qui sont essentiellement :

- Les performances croissantes de ces circuits (les noyaux Soft utilisent maintenant mieux les FPGAs et les ASICs).
- Augmentation de la ration Performances/prix des FPGAs.
- Les noyaux soft sont préférés aux noyaux hard du fait que le software ajoute plus de flexibilité, raccourcit les cycles de développement, permet de plus rapides reconfigurations et simplifie le debuggage.

Pour toutes ces raisons, et en plus de la complexité relative de programmer en VHDL, nous avons opté pour cette solution.

L'environnement EDK de Xilinx nous permet de configurer notre FPGA avec deux noyaux Soft différents : MicroBlaze et Power Pc.

Notre choix a porté sur l'utilisation de MicroBlaze du fait que Power Pc ne soit pas compatible avec la carte que nous possédons (Spartan 3).

IV-5-Conclusion

Ainsi, l'étude des différentes approches que nous pouvions adopter pour le choix de la plateforme matérielle de notre système embarqué, nous a emmené à écarter plusieurs possibilités en raison d'incompatibilité, d'indisponibilité, ou de complexité.

Finalement nous avons donc opté pour l'utilisation d'un circuit programmable de type FPGA, c'est le XC3S200. Ce circuit est fourni sur la SPARTAN 3 Starter Board[5] et sera chargée par un noyau soft (MicroBlaze).

Le tout nous donne une plateforme matérielle prête à accueillir et le système d'exploitation à embarquer et les applications software.

[5] L'étude détaillée de cette carte sera l'objet de la partie « Hardware » du chapitre consacré à l'implémentation.

Chapitre IV

Développement

V-1- Introduction

Le présent chapitre porte sur les différents outils qui nous permettent de réaliser notre système embarqué à savoir le noyau temps réel lui-même et la plateforme matérielle qui l'accueillera.

Pour ce qui est de la plateforme matérielle, le choix d'utiliser MicroBlaze implique l'utilisation de l'environnement de développement EDK, et l'implémentation sur une carte à base de circuit FPGA (Spartan 3).

Le noyau temps réel quant à lui, doit répondre à un critère important qui est la portabilité sur le processeur MicroBlaze.

Dans la première partie de ce chapitre nous vous proposons de découvrir les principales fonctionnalités de cet environnement de développement. La deuxième partie étudiera le processeur virtuel MicroBlaze et ses principales caractéristiques ainsi qu'une présentation détaillée de la carte Spartan, puis dans la dernière partie nous étudierons en détail les différent systèmes d'exploitation temps réel qui peuvent être utilisés pour réaliser notre but et nous exposerons la démarche suivie pour embarquer notre système d'exploitation temps réel à savoir la compilation et le chargement du noyau adéquat sur notre carte.

V-2- La première partie : l'environnement de développement EDK

V-2-1- Introduction

L'environnement de développement fourni par Xilinx et qui permet la configuration des circuits de type FPGA est appelé EDK (Embedded Development Kit). Grâce au principe du Co-Design, cet environnement nous permet d'une part de développer le hardware (configuration de MicroBlaze et de ces périphériques) et d'une autre part, de développer le software (les applications) puis de générer un même Bitstream qui contient à la fois le matériel et le logiciel et qui sera chargé sur le circuit FPGA

EDK contient un environnement de développement intégré (IDE) appelé Xilinx Platform Studio (XPS) qui est une interface graphique d'EDK qui permet de définir le matériel et le logiciel qui seront implémentés sur le circuit FPGA.

EDK contient aussi la plateforme SDK qui peut être utilisée pour développer les applications softwares en langage C ou C++.

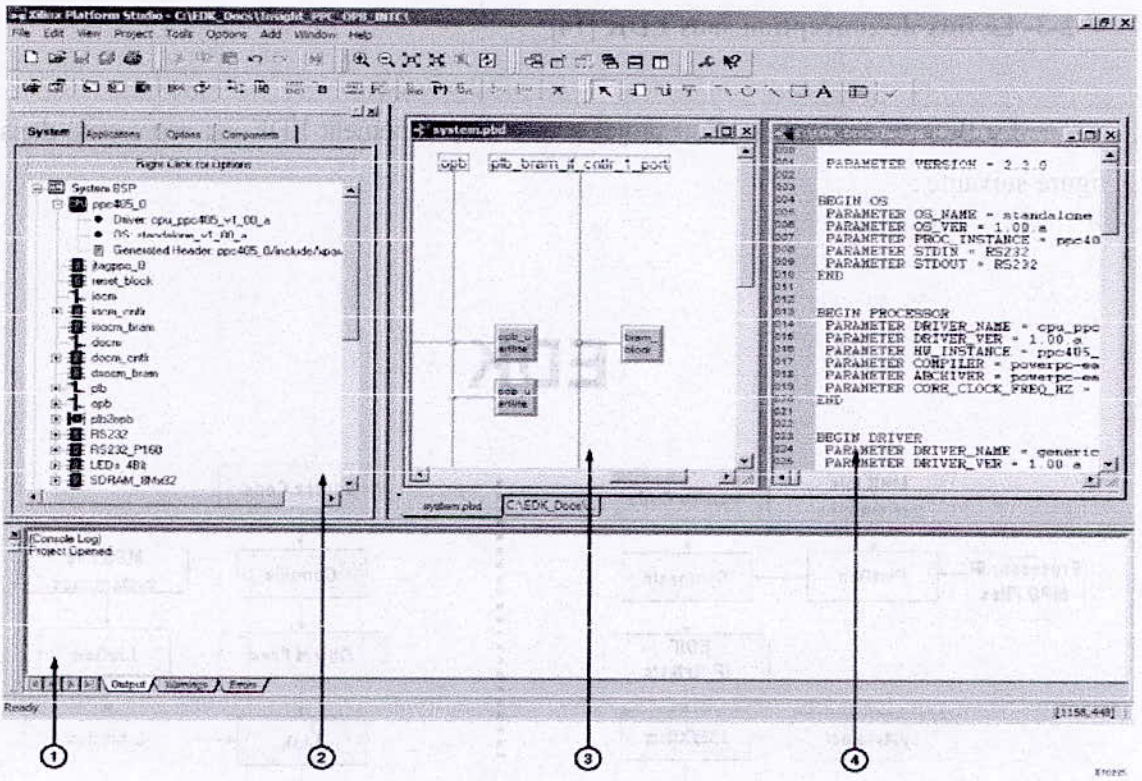
L'environnement de développement EDK contient les outils suivants

a. Xilinx Platform Studio – XPS
b. Base System Builder – BSB
c. Creating/Importing IP Wizard
d. Hardware generation tool – PlatGen
e. Library generation tool – LibGen
f. Simulation generation tool – SimGen
g. GNU software development tools
h. System verification tool – XMD
i. Processor IP
j. Drivers for IP
k. Documentation

V-2-2- Xilinx Platform Studio (XPS) [14]

La fenêtre XPS se divise en plusieurs parties :

1. une console de transcription qui permet de visualiser les erreurs, les commentaires et les warning présentés lors de la compilation du projet.
2. la fenêtre des détails du système (tree view) qui montre les différents composants et fichiers du système.
3. la fenêtre du diagramme (system diagram view) qui montre schématiquement les composants et les connections utilisés dans le système.
4. l'éditeur de code source qui permet d'éditer le code source des applications et de lire les différents fichiers représentés dans la fenêtre tree view.



En plus de EDK, XPS dépend d'un autre environnement de logiciel intégré (ISE) fourni aussi par Xilinx.

ISE est un produit de Xilinx nécessaire pour implémenter une conception sur le circuit FPGA. L'accès aux différents composants supportés par EDK est impossible sans cet environnement, car de nombreux outils présents dans ISE sont appelés à partir de EDK pour effectuer diverses taches (par exemple la génération de la Netlist grâce au flux ISE).

V-2-3- Le flux de conception sous EDK [14]

Le flux de conception d'un projet sous l'environnement EDK est représenté dans la figure suivante :

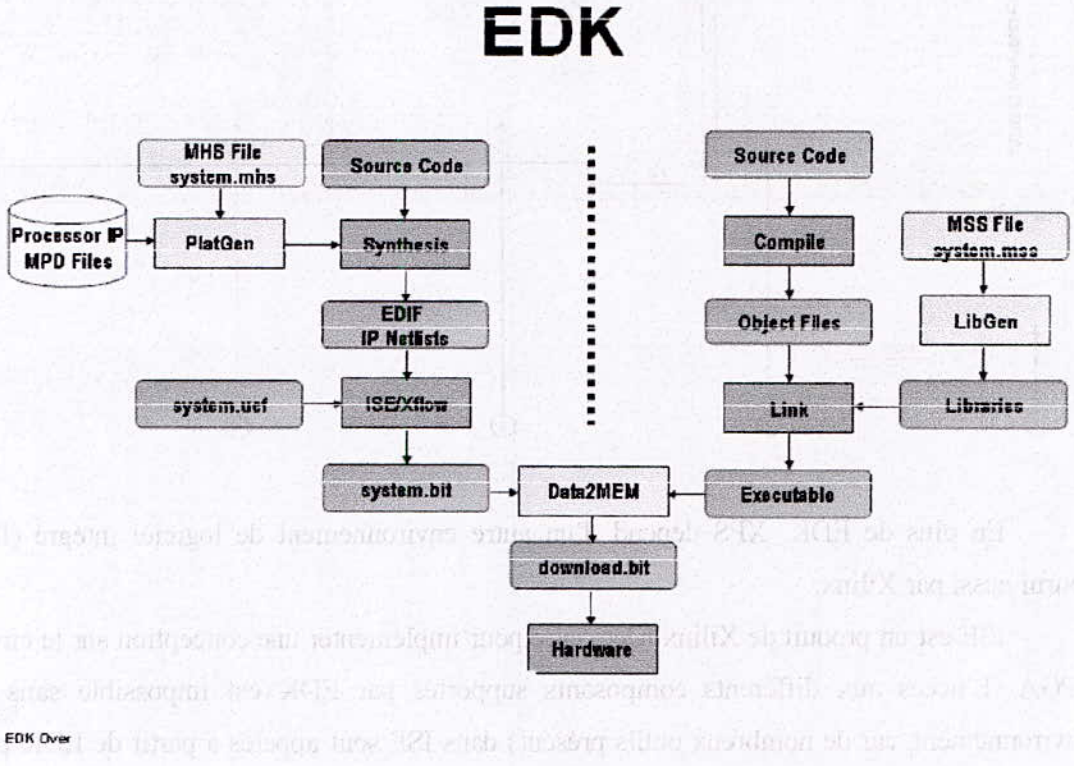


fig.V-1 : le flux de conception sous EDK

L'aspect *co-design* du flux de conception sous EDK nous permet de développer la plateforme matérielle en parallèle avec la plateforme logicielle.

Dans le flux de conception de la partie Hardware, on choisi un processeur IP et les différents périphériques qui lui seront connectés puis on exécute l'outil PlatGen de EDK qui les configure en se basant sur les caractéristiques spécifiées dans le fichier MHS. Une fois la plateforme matérielle spécifiée, une Netlist[1] est synthétisée à partir des codes sources des IP cores propriétaires fournis par l'environnement ISE en langage VHDL. La configuration de la carte et les connexions des différents périphériques avec le circuit FPGA spécifiés dans le

[1] Netlist est une forme synthétisée du matériel qui contient les données logiques de la conception et des contraintes

fichier de contrainte UCF sera ensuite lié avec la Netlist pour former un fichier *.bit* qui est le fichier de configuration de toute la plateforme matérielle d'un projet sous EDK.

Dans le flux de conception logiciel, le programme de l'application doit être écrit soit en langage assembleur soit en langage C ou C++. Le code source obtenu doit être compilé pour avoir des fichiers en code objet qui seront liés avec des librairies spécifiques. Ces dernières sont générées lors de l'exécution de l'outil PlatGen de EDK pour les spécifications du software qui sont données dans le fichier MSS. On obtient à la fin du flux software, un fichier exécutable.

Le fichier *.bit* obtenu à la fin du flux matériel et le fichier exécutable de l'application obtenu à la fin du flux software sont liés par l'outil DATA2MEM pour former un seul fichier *download.bit* qui configure tout le système en entier et qui sera chargé sur le circuit FPGA.

V-2-4- Les fichiers créés par EDK [14]

L'environnement de développement EDK crée plusieurs types de fichiers pour un projet donné, ces fichiers sont généralement édités selon les besoins spécifiés par notre conception voir le type de carte utilisée, le processeur enfoui spécifié ainsi que les périphériques et même les applications et les options choisies. Ces fichiers créés peuvent se résumer dans la liste suivante :

- **XMP (Xilinx Microprocessor Project)** : c'est le fichier Top_level du projet pour la conception de EDK
- **BMM (Block Memory Map)** : le fichier BMM est un simple fichier texte qui décrit comment un bloc RAM constitue un espace mémoire pour une donnée. L'outil qui met a jour le fichier de configuration (Bitstream) utilise les fichiers BMM pour transférer les données dans l'espace d'initialisation.
- **BSB (Base System Builder)** : outil prenant en charge la création d'un nouveau projet. En effet, XPS fournit un Wizard qui nous permet de construire pas à pas la plateforme matérielle.
- **EDIF (Electronic Data Interchange Format)** : c'est un format standard pour les spécifications d'une conception Netlist
- **ELF (Executable Linked Format)** : le format ELF est un format commun standard pour les fichiers exécutables. Généralement ils contiennent la représentation binaire

d'une application pour les instructions machine d'un processeur spécifique, Mais ils peuvent aussi contenir un format intermédiaire qui fournit des services pour l'exécution d'un interpréteur. Le fichier elf est l'exécutable associé à une application.

- **MDD (Microprocessor Driver Description) :** un fichier MDD contient des directives pour personnaliser des pilotes softwares.
- **MHS (Microprocessor Hardware Specifications) :** le fichier MHS définit les composantes matérielles du système. Il est utilisé comme entrée pour l'outil PlatGen. Il définit la configuration du système enfoui et contient :
  - L'architecture des bus
  - Les périphériques
  - Le processeur
  - Les connections du système
  - Les espaces d'adressage
- **MLD (Microprocessor Library Definition) :** un fichier MLD contient les directives pour les librairies softwares personnalisées et les BSP générés pour les systèmes d'exploitation.
- **MPD (Microprocessor peripheral Definition) :** le fichier MPD définit les interfaces des périphériques, il a les caractéristiques suivantes :
  - Liste les différentes connectivités pour les interfaces de bus
  - Liste les paramètres et les valeurs par défaut
  - Chaque paramètre MPD est autrement écris par les assignements équivalents dans le fichier MHS.
- **MSS (Microprocessor Software Specifications) :** le fichier MSS est utilisé comme entrée pour l'outil de génération de librairies LibGen, il contient des directives pour les librairies, les pilotes et les systèmes d'exploitation.
- **NGC :** le fichier NGC est une Netlist qui contient la conception de données logiques et des contraintes. Ce fichier remplace les fichiers EDIF et les fichiers NCF.
- **PAO (Peripheral Analyse Order) :** le fichier PAO contient les fichiers HDL nécessaires pour la synthèse et définie un ordre d'analyse pour la compilation.
- **PBD (Processor Block Diagram) :** c'est un fichier éditable qui peut être ouvert à partir de XPS. L'éditeur PBD permet de lire, écrire, modifier et enregistrer une description du circuit FPGA qui représente les composants matériels utilisés. L'éditeur PBD utilise les informations de la plateforme matérielle fournis par le fichier MHS.

- **UCF (User Constraints File) :** le fichier UCF est utilisé pour définir les pins du circuit FPGA qui sont utilisée dans la conception.

V-2-5- Les répertoires crées lors du lancement d'un nouveau projet [14]

Les fichiers créés lors de la création d'un projet sont inclus dans des répertoires spécifiques, ces répertoires sont :

- **Bootloops :** il contient le fichier exécutable Bootloop qui est générer automatiquement à chaque création d'un nouveau projet sur XPS.

Un Bootloop est un programme qui permet de garder le processeur dans un état définit jusqu'à ce que les applications soient chargées sur le circuit. Il consiste en une simple instruction de branchement qui maintient le PC[2]dans une boucle infinie. Cette boucle ne sera interrompue qu'une fois l'application chargée et son exécution lancée.

- **BSP :** c'est l'acronyme de *Board Support Package* ce répertoire est utilisé pour inclure les fonctions du BSP ou modifier un BSP existant.

Le BSP est la couche la plus basse du software permettant d'accéder à des fonctions spécifiques au processeur. C'est une librairie de fonctions très proches du matériel car elles prennent en charge les caches, les exceptions, les interruptions, l'écriture et la lecture du port FSL …etc.

Lorsque aucun système d'exploitation n'est utilisé, XPS sélectionne par défaut le BSP de Standalone. Ce même BSP est indispensable dans le cas ou le noyau Xilkernel est utilisé. En effet, il est souvent assimilé à « l'arête principale » du noyau.

- **Data :** ce répertoire est indispensable pour chaque projet, il contient le fichier UCF qui définit les contrainte du système et les différentes connections des pins du FPGA au périphériques utilisés et spécifiés sur la carte.
- **Drivers :** c'est un répertoire optionnel, il est utilisé pour ajouter des pilotes.
- **Etc :** XPS crée automatiquement ce répertoire quand un nouveau projet est crée. Il contient des fichiers nécessaires pour l'utilisation de l'outil BitGen relatif aux spécifications de la carte utilisée. Quelques fichiers de ce répertoire peuvent être modifiés pour correspondre à la carte sélectionnée.

[2] Program Counter

- **Hdl :** quand on sélectionne **Generate Netlist** des outils fournis par XPS, PlatGen crée ce répertoire qui contient tout le code HDL représentant le système.
- **Implementation :** XPS crée ce fichier automatiquement quand on sélectionne **Generate Bitstram :** il contient une copie du fichier UCF, du fichier BMM et les résultats de l'implémentation contenant le fichier .bit nécessaire pour tout chargement sur la carte.
- **MicroBlaze_ :** ce répertoire est automatiquement crée par LibGen dés qu'on exécute le processus **Generate libraries and BSPs.** Il contient les sous répertoires suivants :
  - **Include :** il contient les fichiers C headers nécessaires pour les pilotes. Le fichier Xparameter.h est aussi crée par libgen dans ce répertoire. Ce fichier définit les adresses de base des périphériques utilisés dans le système, les différentes fonctions (#defines) nécessaires pour les pilotes, les systèmes d'exploitation, les librairies, les programmes utilisateurs ainsi que des fonctions prototypes.
  - **Lib :** il contient les librairies *libc.a, libm.a* et *libxil.a.* la librairie *libxil* contient les fonctions des pilotes.
  - **Libsrc :** il contient des fichiers Makefile intermédiaires nécessaires pour compiler les systèmes d'exploitation, les librairies et les pilotes. Il contient aussi les fichiers des pilotes de périphériques spécifiques, le fichier BSP du système d'exploitation, et les fichiers des librairies.
  - **code :** un répertoire pour les exécutable de EDK
- **pcores :** c'est un répertoire optionnel qui contient les périphériques matériels utilisés dans le système.
- **Simulation :** XPS crée automatiquement ce répertoire quand on exécute le processus de simulation par **tools>sim model generation.** SimGen produit tous les fichiers de simulation pour chaque type de simulation (behavioral, structural et timing) dans le sous répertoire approprié.
- **Sw_services :** c'est un fichier optionnel utilisé pour inclure les librairies personnalisées.
- **Synthesis :** PlatGen crée ce répertoire automatiquement quand on sélectionne **Generate Netlist**, il contient tout les scripts de synthèse et les fichiers .log qui crée la Netlist utilisée pour l'implémentation du système.

- **Testapp :** Généré par BSB pour contenir le fichier test application qui est un simple application préconisée pour tester le bon fonctionnement du système.

V-2-6- La création d'un projet [14]

XPS permet à l'utilisateur de créer son système d'une manière très simple en fournissant un Wizard BSB (Base System Builder). Ce Wizard automatise la configuration de plusieurs matériels standards et des plateformes logicielles communes pour la majorité des conceptions. Il est très efficace pour la création d'un système sur une carte à base de FPGA supportée par EDK, mais si l'utilisateur désire utiliser une autre carte, certaines modifications doivent être ajoutées. L'utilisateur n'est pas obligé d'utiliser ce Wizard mais dans ce cas il doit avoir une connaissance assez profonde du matériel utilisé.

Les étapes du flux de BSB pour créer un projet sont :

- appeler le Wizard BSB
- sélectionner un point de départ, créer un nouveau projet ou ouvrir un projet existant.
- sélectionner une carte de développement qui sera utilisée.
- Sélectionner un processeur
- Configurer caractéristiques et propriétés du processeur et du système.
- sélectionner les mémoires externes et les dispositifs d'entrée/sortie
- ajouter des périphériques
- configurer les caractéristiques du software
- générer le système et la configuration mémoire
- les fichiers de sorties
- quitter le BSB

Un projet typique de conception d'un système embarqué est développé en suivant les étapes suivantes :

- la création d'une plateforme matérielle,
- la création d'une plateforme logicielle
- la création d'une application logicielle
- la vérification du logiciel (déboguage)

V-2-6-1- Création de la plateforme matérielle [14]

La plateforme matérielle est crée en utilisant l'outil de génération de plateforme PlatGen. Cet outil a pour entrée le fichier MHS qui définit l'architecture du système, les périphériques, les processeurs enfouis, les connections du système, la configuration des adresses de chaque périphérique dans le système et les options de configuration de chaque périphérique. PlatGen crée donc une Netlist qui décrit le matériel utilisé en plusieurs formes (NGC, EDIF). Après l'exécution de cet outil, les outils d'implémentation sur FPGA (ISE) s'exécutent pour compléter l'implémentation du matériel. A la fin du flux ISE, un fichier de configuration (Bitstream) est généré pour configurer le circuit FPGA. Le Bitstream contient l'information pour l'initialisation de la mémoire BRAM sur le circuit FPGA.

Plusieurs périphériques utilisés souvent, sont fournis par Xilinx avec l'outil EDK, cependant on peut définir nos propres périphériques et les inclure dans le fichier MHS.

Il est possible d'éditer son propre fichier MHS, et d'y inclure d'autre périphériques mis à part ceux définis sur EDK, mais cela implique l'écriture (sur ISE) de codes VHDL descriptifs du matériel ajouté.

Le flux de configuration du matériel est donné dans la figure suivante :

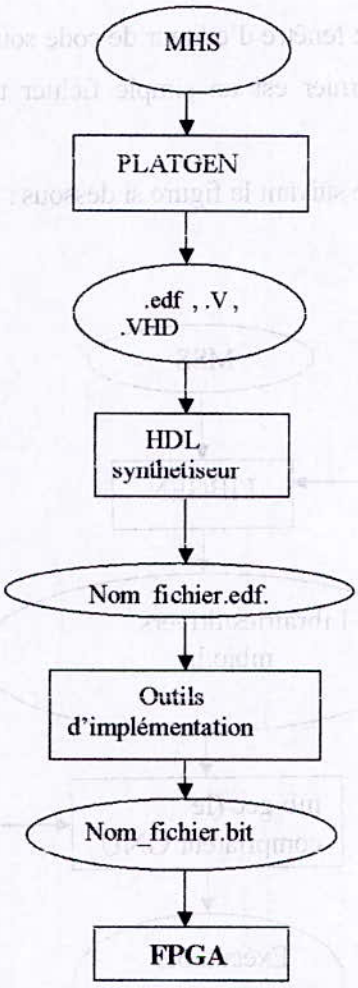


Fig. V-2 : Flux matériel de EDK

V-2-6-2- La création de la plateforme logicielle [14]

La plateforme logicielle est spécifiée par le fichier MSS (Microprocessor Software Specification) qui définit les librairies, les pilotes, les paramètres de personnalisation du processeur, les dispositifs d'entrées/sorties, les routines de traitement d'interruptions et d'autres caractéristiques du software. Le fichier MSS est utilisé comme entrée pour l'outil de génération de librairies (LibGen). Cet outil nous permet de configurer les librairies et les pilotes avec les adresses des périphériques du processeur enfoui.

Les librairies et les pilotes configurés et les programmes sources (d'extension .c ou bien .h) seront compilés grâce à mb-gcc qui est un compilateur GNU adapté à MicroBlaze.

Le résultat est un fichier exécutable qui sera chargé dans la BRAM ou la SRAM de MicroBlaze pour être exécuté.

L'interface XPS fournit une fenêtre d'éditeur de code source dans lequel on peut créer le fichier MSS. Cependant ce dernier est un simple fichier texte qu'on peut écrire dans n'importe quel éditeur de texte.

La génération des librairies est faite suivant la figure si dessous :

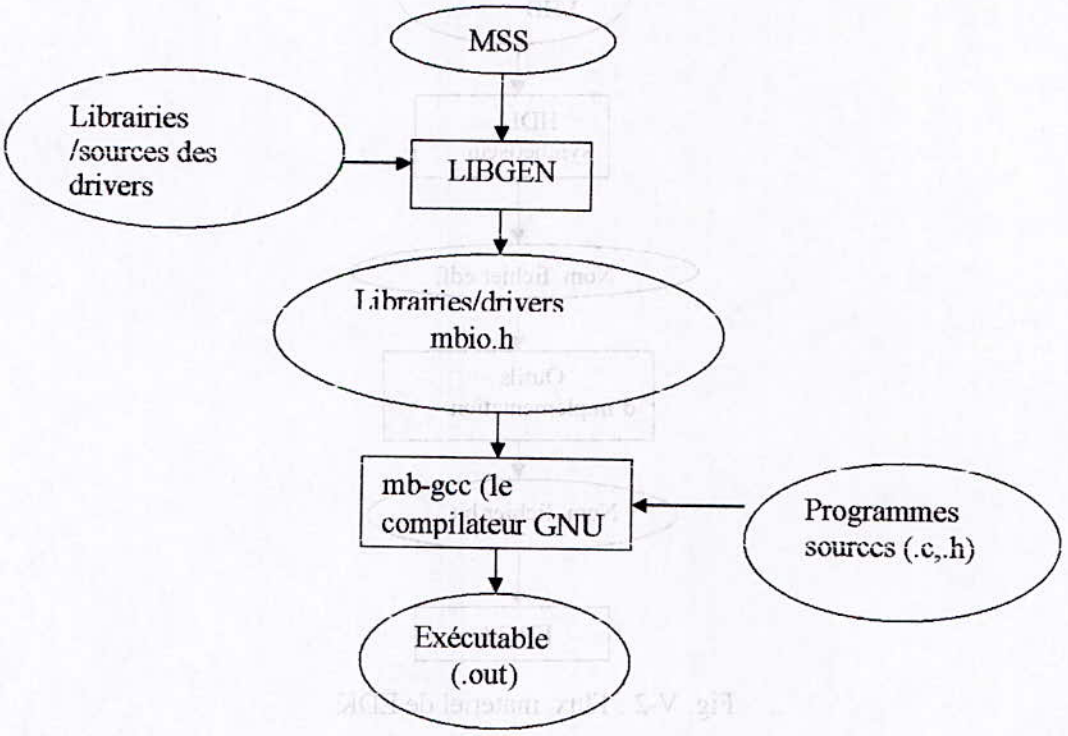


Fig. V-3 : Flux logiciel de EDK

V-2-6-3- La création d'une application software [14]

Une application software nécessite

- la création d'une nouvelle application software
- la compilation de l'application et la création du fichier ELF.
- L'initialisation des BRAMs avec les informations ELF.
- Chargement du bitstream

- Le déboguage de l'application

Une fois le fichier code source crée, compilé et après création des liens vers les librairies, un fichier exécutable d'extension .ELF est généré. Ce fichier doit être inclut dans le Bitstream si on veut qu'il soit chargé sur le circuit FPGA au même temps que le matériel. Pour ce faire un outil de ISE est utilisé pour permettre à l'utilisateur de mettre à jour le *Bitstream* avec une information code/donnée obtenue à partir du fichier executable.elf qui est généré à la fin du flux.

Le flux de génération d'une application software est illustré dans la figure suivante :

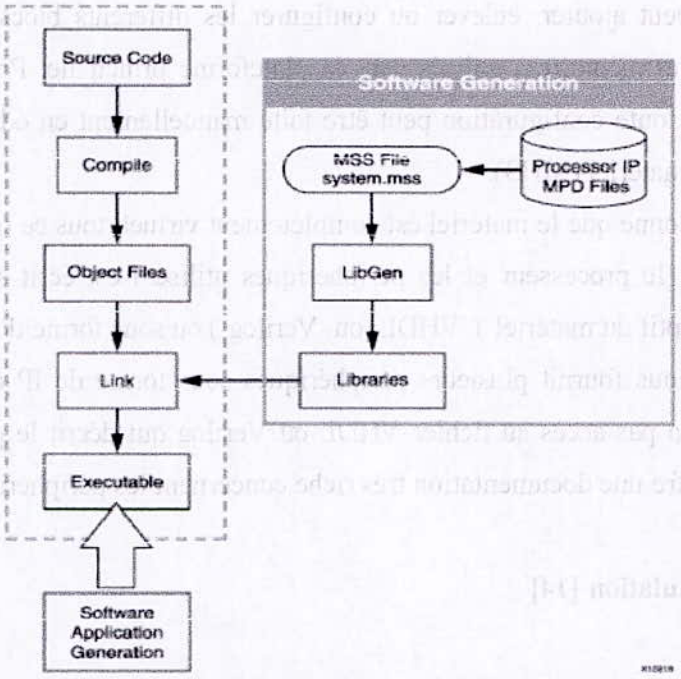


Fig. V-4 : Le flux de génération d'une application software

V-2-6-4- Le déboguage du logiciel [14]

Pour le déboguage du logiciel, un débuggeur appelé XMD (Xilinx Microprocessor Debugger) est inclus, il permet de charger l'application logicielle dans la mémoire de MicroBlaze, contrôler le flux d'exécution et inspecter les registres.

Ce débuggeur peut être connecté à MicroBlaze ou PowerPC implémentés sur le circuit FPGA ou exécuter une instruction de simulation. Un autre débuggeur appelé GDB de GNU

est aussi nécessaire pour compléter le déboguage du logiciel, il permet aux utilisateurs de commencer l'exécution du programme pour poser des conditions initiales, les points d'arrêt et pour examiner l'état du processeur et les composantes des mémoires. Lors de l'exécution du système sur la carte FPGA, XMD peut être connecté à MicroBlaze via un module de déboguage en utilisant un câble JTAG.

V-2-7- La configuration du matériel [14]

La configuration du matériel relatif à MicroBlaze peut être faite à l'aide d'une fenêtre de dialogue « Add/Edit hardware Platform specification » fournie dans l'environnement XPS. L'utilisateur peut ajouter, enlever ou configurer les différents blocs matériels (processeur, périphériques et mémoires) utilisés dans la plateforme principale. Pour des utilisateurs plus expérimentés, toute configuration peut être faite manuellement en éditant des fichiers textes définissant le matériel (MHS).

Etant donné que le matériel est complètement virtuel, tous ce qui sera configuré sur le circuit FPGA (le processeur et les périphériques utilisés) est écrit sous forme de code en langage descriptif du matériel (VHDL, ou Verilog) ou sous forme d'une Netlist synthétisée.

EDK nous fournit plusieurs périphériques sous forme de IP cores[3]. C'est-à-dire que l'utilisateur n'a pas accès au fichier VHDL ou Verilog qui décrit le périphérique. En contre partie EDK offre une documentation très riche concernant les périphériques qu'elle fournit.

V-2-8- La simulation [14]

Le logiciel et el matériel spécifiés dans un projet peuvent être simulés par un simulateur d'instruction ISS (Instruction Set Simulation) débutée à partir du débuggeur XMD. Deux simulateurs HDL (ModelSim et NcSim) sont supportés par XPS et permettent de vérifier l'application générée dans le projet. Malheureusement quelques librairies et models doivent être compilés avant de générer la simulation ce qui est plus au mois délicat. Il existe trois modes de simulation *behavioral simulation, structural simulation et timing simulation*, ces trois modes doivent être sélectionnés et compilés avant de lancer la simulation.

[3] Intellectual property : codes protégés

V-3- La deuxième partie : La plateforme matérielle

Comme nous l'avons déjà évoqué, concernant le hardware, l'approche sur laquelle notre choix a été porté est l'utilisation d'un circuit programmable du type FPGA et d'un processeur soft (MicroBlaze).

La taille réduite d'un circuit FPGA exige son implémentation sur un support matériel doté de périphériques et de pins à l'échelle des fils électriques utilisés pour la commande des conceptions matérielle tel un robot. Le support matériel de notre implémentation (notre carte de contrôle) est la SPARTAN 3 qui inclut le circuit FPGA ainsi que d'autres périphériques (switches, leds, afficheurs sept segments, mémoire flash,...etc.).

V-3-1- Le processeur MicroBlaze [15]

Le processeur soft MicroBlaze embarqué est un processeur à jeu d'instruction réduit (RISC[4]) optimisé pour l'implémentation de Xilinx sur des circuits FPGAs. Développé par Xilinx, le processeur soft MicroBlaze est disponible dans l'environnement EDK de Xilinx sous la forme d'une description faite en HDL (langage de description du matériel : Hardware Description Language). De nombreux périphériques peuvent lui être connectés via le bus OPB tel un contrôleur réseau Ethernet ou un module permettant la gestion du port série.

Conçu selon une architecture dite HARVARD séparant le chemin des instructions de celui des données et permettant un parallélisme dans le transfert des données et des instructions, MicroBlaze réalise l'exécution des instructions en un seul cycle d'horloge grâce à l'utilisation du pipeline. La figure (V-5) nous donne un aperçu général sur l'architecture de MicroBlaze.

[4] C'est un jeu d'instruction formé par le instructions les plus simples et les plus fréquemment utilisées. Un processeur RISC se caractérise par : une exécution simple (en un cycle d'horloge) des instructions, une longueur d'instruction uniforme, des modes d'adressage simples, et une couche micro programmée inexistante.

V-3-1-1- Caractéristiques de MicroBlaze [15]

MicroBlaze se caractérise par :

- 32 registres banalisés à 32 bits.
- Un mot d'instruction à 32 bits avec trois opérandes et deux modes d'adressage.
- des bus de données et d'instructions séparés à 32 bits conforment aux spécifications des bus OPB d'IBM.
- Des bus de données et d'instructions séparés de 32 bits connectant le processeur au bloques mémoire internes via le bus LMB.
- Des bus d'adresse à 32 bits.
- Un pipeline qui assure l'exécution d'une instruction à chaque cycle d'horloge.
- Un cache d'instructions et de données qui permet d'augmenter la bonde passante entre le processeur et la mémoire.
- Un debugger logique du matériel.
- Un support FSL (Fast Simplex link) qui permet une seule fonction à la fois soit la lecture soit l'écriture

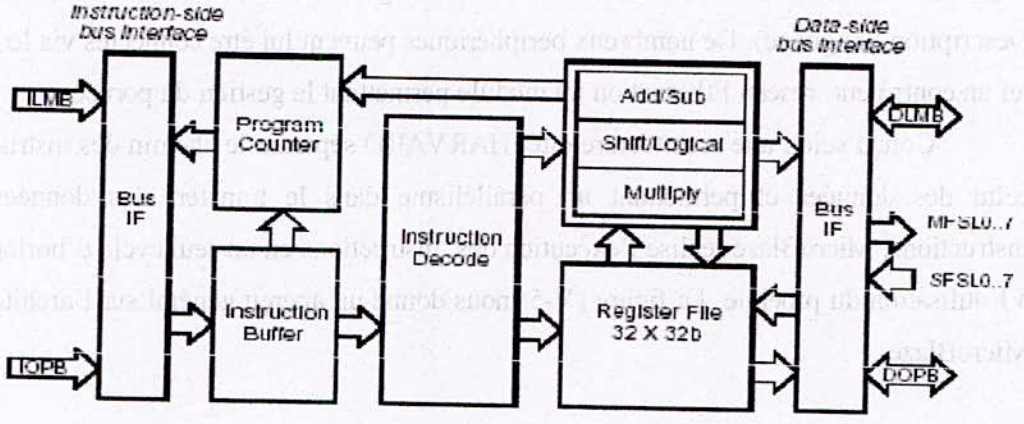


Fig. V-5 : l'architecture de MicroBlaze

V-3-1-2-Les registres de MicroBlaze [15]

Les registres du processeur constituent un élément que l'on retrouve dans la plupart des architectures à l'heure actuelle, en effet MicroBlaze se présente sous un modèle de programmation en machine à registre. Les caractéristiques essentielles des registres sont :

- Ils constituent un espace d'adressage distinct ;
- Leur nombre est peu élevé ;
- Ils sont de taille fixe ;
- Leur accès est rapide ;
- Ils sont spécialisés ou banalisés selon le type d'architecture ;

MicroBlaze comporte 32 registres à 32 bits banalisés, et 2 registres à 32 bits spécialisés.

V-3-1-2-1-Les registres banalisés :

Varient de R0 à R31. R0 est défini de manière à avoir toujours la valeur 0. Tous ce qui sera écrit dans R0 sera écrasé.

V-3-1-2-2- Les registres spécialisés :

- **Le PC (program counter : compteur programme) :**

Ses 32 bits contiennent l'adresse de l'instruction suivante qui doit être ramenée pour la décoder et l'exécuter. Le PC peut être lu par l'instruction MFS mais on ne peut y écrire par l'instruction MTS (voir annexe A).

- **Le MSR (machine status register : registre d'état de la machine) :**

Il contient le drapeau du Carry, les validations des interruptions, du verrouillage du bus, et des caches, mais aussi un signalement des erreurs qui se produisent sur le FSL (fast simplex lin : lien rapide simplexe).Le MSR peut être lu en utilisant l'instruction MFS. Lors de la lecture du MSR, le bit 29 (qui contient le carry) est copié dans le bit 0 du MSR comme une copie du carry. On peut aussi écrire dans le MSR et ce en utilisant l'instruction MTS, la valeur écrite prendra place un cycle d'horloge après l'exécution de l'instruction MTS.
Toute valeur écrite dans le bit 0 du MSR sera écrasée.

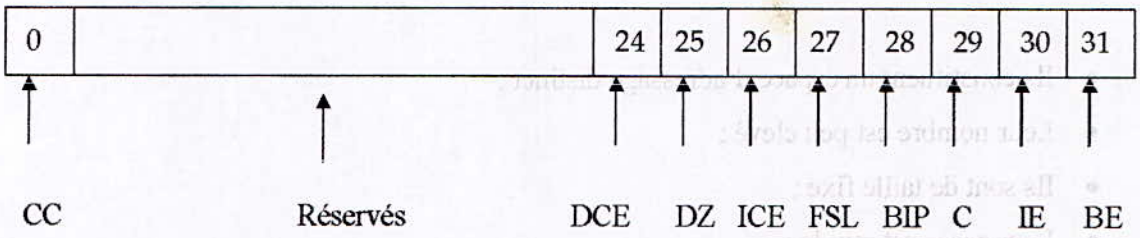


Fig. VI-2 : Le MSR

Voici un tableau descriptif du MSR :

Bit	Nom	Description	Valeur au RESET
0	CC	*Arithmetic carry copy* Copie de la carry arithmétique Ce bit peut seulement être lu	000.....00000
1-23	Réservé		
24	DCE	*Data cache enable* 0 : le cache de données est non valide. 1 : le cache de données est validé	000.....00000
25	DZ	*Division by Zero* 0 : il ne s'est pas produit de division par zéro 1 : division par zéro	000.....00000
26	ICE	*Instruction cache enable* 0 : le cache d'instruction est non validé. 1 : le cache d'instruction est validé.	000.....00000
27	FSL	Erreur sur le *Fast simplex link* (lien simplexe rapide) 0 : pas d'erreur sur le FSL get \put 1 : le FSL get \put s'est trompé dans le type d'instruction et dans le type de la valeur.	000.....00000

28	BIP	*Break in progress* (rupture du déroulement) 0: pas de rupture. 1: rupture La source du break peut être logicielle (contenu dans le programme) ou bien matérielle (break extérieur)	000.....00000
29	C	*Arithmetic Carry* (retenue arithmétique) 0 : pas de retenue (emprunt) 1 : il y a une retenue (pas d'emprunt)	000.....00000
30	IE	*Interrupt enable* (validation des interruptions) 0 : mode non interruptif 1 : mode interruptif	000.....00000
31	BE	*Buslock enable* (validation du verrouillage du bus) 0 : le verrouillage non valide du bus OPB coté données(DOPB). 1 : verrouillage valide du bus OPB coté données (DOPB).Le verrouillage de bus n'affecte pas les opérations s'effectuant sur :ILMB, DLMB, IOPB.	000.....00000

V-3-1-3-Le jeu d'instruction de MicroBlaze [15]

MicroBlaze compte 62 instructions distinctes d'une longueur de 32 bits . Elles sont regroupées sous l'ordre fonctionnel suivant :

- Arithmétiques ;
- Logiques ;
- De branchement ;
- De chargement et de sauvegarde ;
- Et les instructions spéciales.

Microblaze offre deux formats d'instructions et deux modes d'adressages.

V-3-1-4- Les modes d'adressages

MicroBlaze comporte deux modes d'adressage :

- **Immédiat :**

On donne la valeur immédiate de l'opérande .sans spécifier le registre qui la contient.

- **Implicite :**

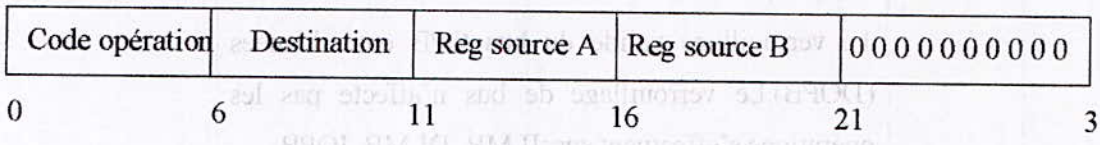
Les opérandes manipulés sont les registres internes de MicroBlaze (R0 à R31, PC et MSR) .

V3-1-5- Les formats des instructions

Les instructions de MicroBlaze sont soit de type soit de type B :

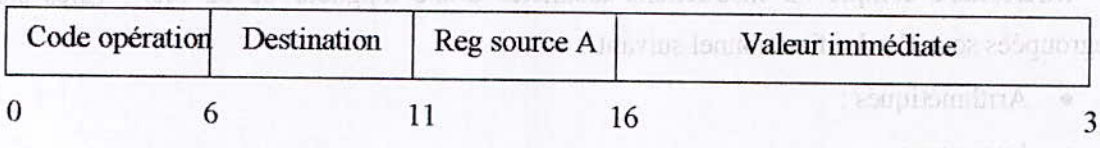
- **Le type A :**

Le type A est utilisé pour les instructions *registre à registre* (tout est en adressage implicite). Il contient le code opération, une seule destination et deux registres sources.



- **Le type B :**

Pour le type B les instructions contiennent un code opération, un registre destination et un registre source, et une source en valeur immédiate sur 16 bits .



V-3-1-6- L'architecture chargement \sauvegarde [15]

Les données sur MicroBlaze ont les trois formats suivants:

- Byte (sur 8 bits)
- Halfword (sur 16 bits)
- Word (sur 32 bits)

V-3-1-7- Les entrées /sorties [15]

L'accès à la mémoire peut se faire via trois bus différents :

LMB

C'est le local Memory Bus, il permet l'accès à la Block RAM. Il utilise un nombre minimum de signaux de control et un protocole simple pour assurer que la BRAM est accédée en un cycle d'horloge.

La structure Harvard de MicroBlaze divise le LMB en ILMB pour le transport des instructions, et le DLMB pour les données.

OPB

C'est le On chip Peripheral Bus, il permet d'accéder au périphériques on chip. L'OPB est aussi divisé en DOPB (pour les données allant de ou vers les périphériques on chip), et en IOPB (pour les instructions) .

V-3-1-8-Le pipeline de MicroBlaze [15]

L'architecture HARVARD de MicroBlaze lui permet d'exécuter une instruction en un cycle d'horloge, ceci est possible grâce au pipeline parallèle de MicroBlaze qui est divisé en trois stages :

- Ramener l'instruction
- La décoder
- L'exécuter

En effet, en général chaque stage prend un cycle d'horloge pour être exécuté, par conséquent, pour que l'instruction soit exécutée, il nous faut au moins trois cycles d'horloges (sans compter les retards).

cycle 1	cycle 2	cycle 3
Fetch	Decode	Execute

Dans le pipeline parallèle de MicroBlaze on peut exécuter trois instructions simultanément, chacune dans un des trois stages du pipeline même si chaque instruction prend toujours trois cycles pour être exécuter .En réalité, pendant un cycle d'horloge, une nouvelle instruction est ramenée, une autre est décodée et une troisième est exécutée. Par conséquent, Le pipeline exécute effectivement une instruction par cycle d'horloge. Ceci n'aurait pas été possible sans l'architecture *HARVARD,* qui permet d'accéder aux données et aux instructions de deux manières différentes, permettant ainsi un parallélisme du pipeline.

	cycle 1	cycle 2	cycle 3	cycle4	cycle5
instruction 1	Fetch	Decode	Execute		
instruction 2		Fetch	Decode	Execute	
instruction 3			Fetch	Decode	Execute

V-3-1-9- Les interruptions [15]

Quand une interruption se produit, MicroBlaze stoppe l'exécution en cours , pour prendre en charge l'interruption .MicroBlaze se branche à l'adresse 00....00010 et utilise l'adresse 14 pour sauvegarder l'adresse de l'instruction qui devait s'exécuter quand l'interruption est survenue .MicroBlaze invalide les interruptions futures en mettant à zéro le drapeau d'interruption IE (interrupt Enable) dans le MSR (mise à zéro du bit 30 du MSR), l'instruction qui se trouve dans l'adresse ou se pointe le PC courant n'est pas exécutée , mais elle le sera une fois l'interruption passée.

L'interruption ne peut avoir lieu si le bit BIP du MSR est active (BIP=1, validation du break).

V-3-1-10-Les exceptions [15]

Quand une exception se produit, MicroBlaze stoppe l'exécution en cours pour prendre en charge l'exception .MicroBlaze se branche à l'adresse 0....0008, et utilise le registre banalisé R17 pour sauvegarder l'adresse de l'instruction qui devait s'exécuter quand l'exception s'est produite.

V-3-1-11- Les caches [15]

En général, les caches sont utilisés pour faciliter et accélérer l'accès aux données et aux instructions. Les caches varient selon les informations qu'ils mémorisent :

- Données ;
- Instructions ;
- Données plus instructions.

En remarque que dans l'ensemble l'efficacité d'un cache d'instruction est plus importante que celle d'un cache de données .en effet, la localité d'un code est plus importante que celle d'une donnée.

MicroBlaze peut être utilisé avec un cache d'instructions pour améliorer les performances lors de l'exécution de codes résidant sur un périphérique (en dehors des instructions ramenées via le bus ILMB.

Il peut aussi être utilisé avec un cache de données qui permet un accès rapide aux données provenant via le bus DOPB.

V-3-1-12- L'interface simplex link (lien simplexe) [15]

MicroBlaze comprend huit entrées et huit sorties d'interfaces de lien rapide simplexe (FSL : fast simplex link).

Les canaux FSL sont des interfaces unidirectionnelles dédiées au transport de données ou d'instructions .Sur MicroBlaze les interfaces FSL ont une largeurs de 32 bits .De plus , un même canal FSL peut être utilisé pour transmettre ou recevoir des données ou des instructions, un des 32 bits est un drapeau qui nous indique s'il s'agit d'une données ou d'une instruction.

V-3-2- La carte de contrôle: La *Spartan3 Starter Board* [16]

V-3-2-1-Présentation générale

La carte de contrôle est fournie sous forme d'un kit comprenant la carte, les câbles de programmations ainsi qu'un module d'alimentation. Ce kit fait office de plateforme de développement "tout en un" universelle spécialement conçue pour l'apprentissage rapide des techniques de conception numérique.

De part la présence de son FPGA très largement dimensionné (près de 200 K portes) et de ses dispositifs de commandes et de visualisation divers (boutons-poussoirs, afficheurs, Leds, port PS2, Port VGA...), cette platine convient tout aussi bien pour la réalisation d'applications de décodage logique très simple comme pour la mise au point de réalisations extrêmement complexes et puissantes. L'ensemble est livré avec un bloc d'alimentation, un câble de programmation "JTAG" permettant de programmer votre application en mémoire Flash non volatile.

V-3-2-2-Caractéristiques principales du kit [16]

- Base de développement FPGA complète avec câble de programmation "JTAG" livré
- Equipé d'un FPGA Spartan-3 avec 216 Kbits de bloc RAM et horloge interne jusqu'à 500 MHz
- Oscillateur 50 MHz inclus - support pour second oscillateur
- Plate-forme flash 2 Mbits (XCF02S) intégrée à la carte
- Mémoire SRAM (256 Kb x 32) intégrée à la carte
- 3 connecteurs d'extensions inclus sur la carte
- 4 afficheurs 7 segments à Leds
- 9 Leds
- 8 interrupteurs et 4 boutons-poussoirs
- Port série, VGA et PS2
- 3 régulateurs de tension (3,3 V / 2,5 V et 1,2 V)
- Livré avec câble de programmation (prévoir une alimentation de : +5 V)

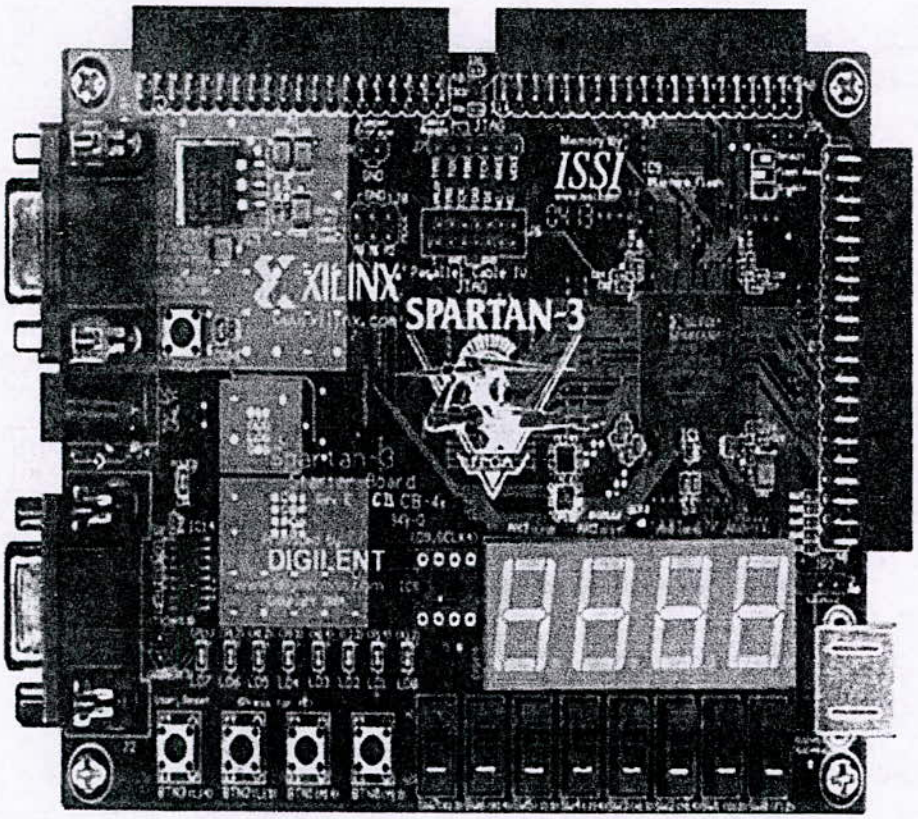


Fig.V-6: Spartan 3 starter board de DIGILENT

V-3-2-3- Composants essentiels de la Spartan 3 Starter board [16]

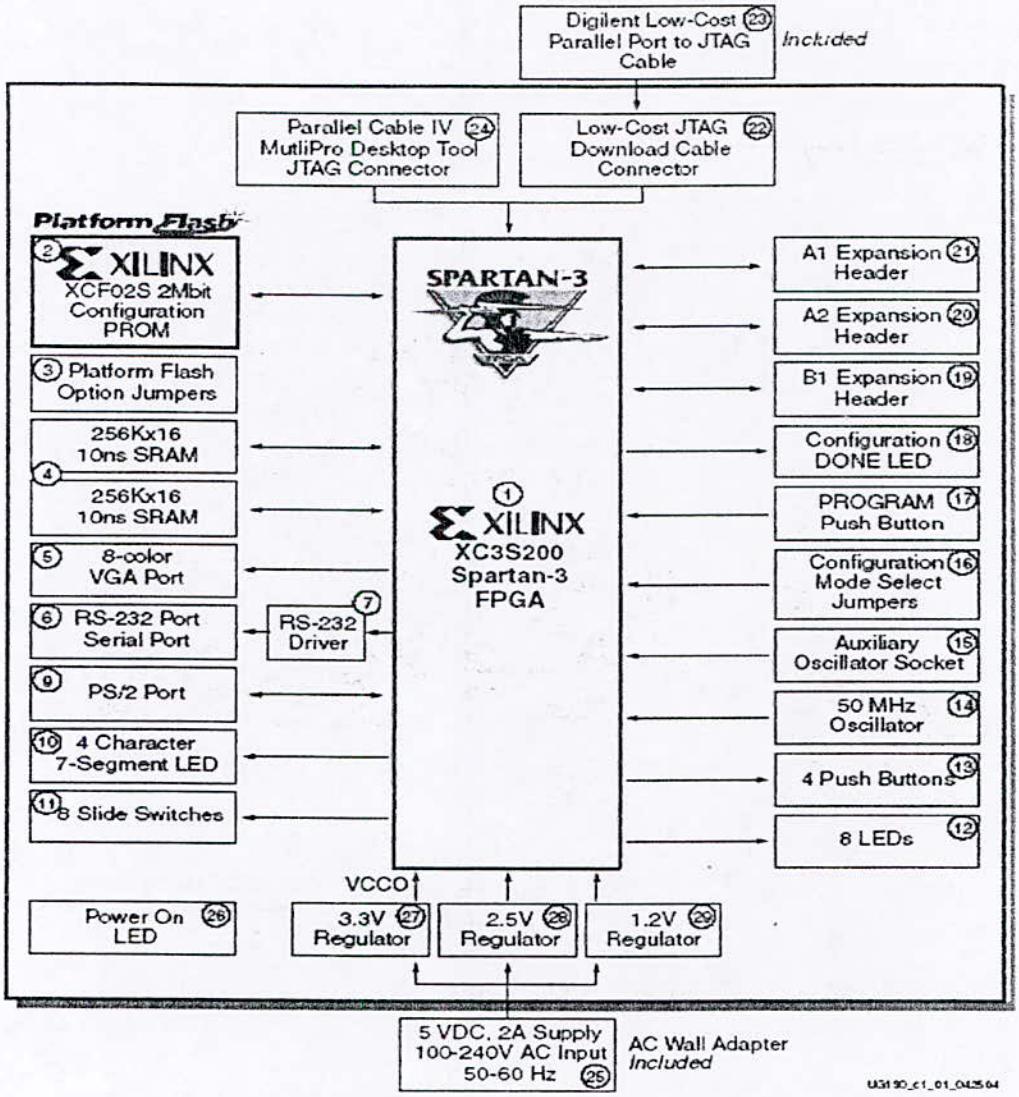


Fig.V-6 : Diagramme block de la Spartan 3

1. Circuit programmable de type FPGA (constructeur : Xilinx) avec une capacité d'intégration de 200 000 portes logiques. Le circuit FPGA est fournit suivant le package XC3S200FT256. ①
2. Flash mémoire PROM d'une capacité de 2Mbit ② :

- 1Mbit de mémoire non volatile comme support de stockage de données et de programmes disponibles après la configuration du circuit FPGA.
- Les options du cavalier permettent au FPGA de lire des données PROM ou une configuration du FPGA à partir d'autres sources ③ .

3. 1M byte de SRAM④.
4. Port à 3 bits pour afficheur VGAà 8 couleurs⑤.
5. Port série RS-232 à 9 pins⑥.
6. Pilote pour le RS-232 ⑦.
7. Port PS/2 pour sourie/clavier⑨.
8. Afficheur sept segments à quatre caractères⑩.
9. huit swithes⑪.
10. Huit Leds individuelles⑫.
11. quatre boutons poussoirs⑬.
12. Cristal oscillateur à 50 MHZ⑭.
13. Socket pour une horologe supplémentaire ⑮ .
14. Mode de configuration du FPGA sélectionné selon la position de cavaliers ⑯ .
15. Bouton poussoir servant à forcer la reconfiguration du FPGA (une fois le bouton appuyé l'ancienne configuration est écrasée) ⑰ .
16. Led indiquant que le FPGA a été correctement configurée ⑱ .
17. 3 ports de connection à 40 pins chacun ⑲ ⑳ ㉑ .
18. Port JTAG㉒ pour câble de chargement ㉓ .
19. Câble JTAG Digilent pour chargement et debuggage connecté au port parallèle du PC㉓.
20. Port JTAG de chargement /debuggage compatible avec le câble parallèle de Xilinx ㉔.
21. Entrée pour adaptateur AC ㉕ .
22. Led indicatrice (s'allume lors de la mise sous tension de la carte) ㉖ .
23. Régulateurs de tension 3.3 V ㉗, 2.5 V ㉘ et 1.2 V㉙.

V-3-2-4-Localisation des composants [16]

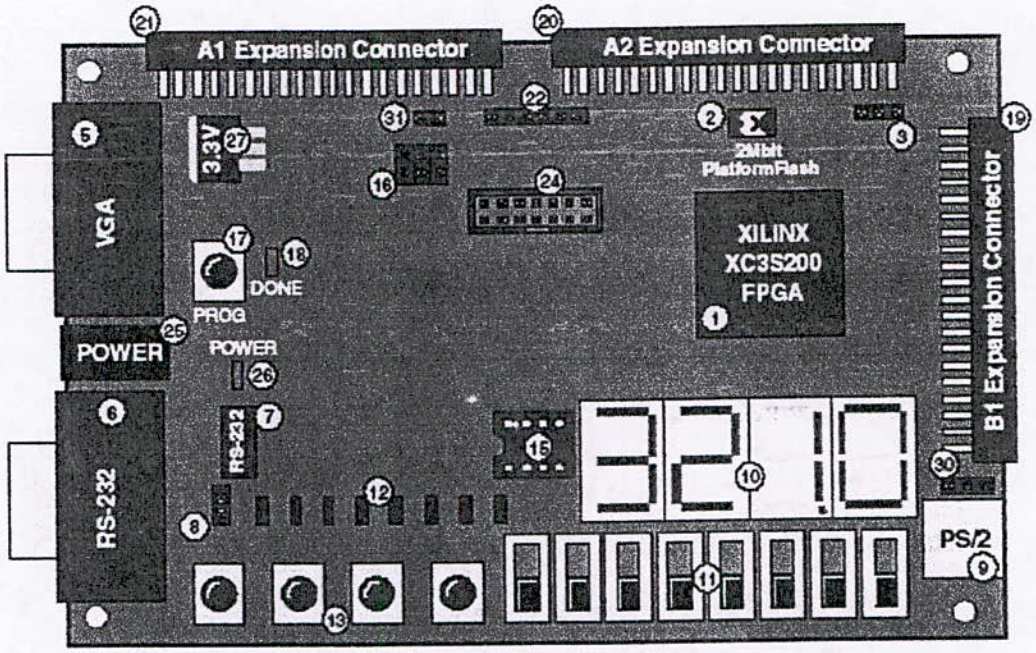


Fig.V-7 : Le Kit Spartan 3 vu d'en haut

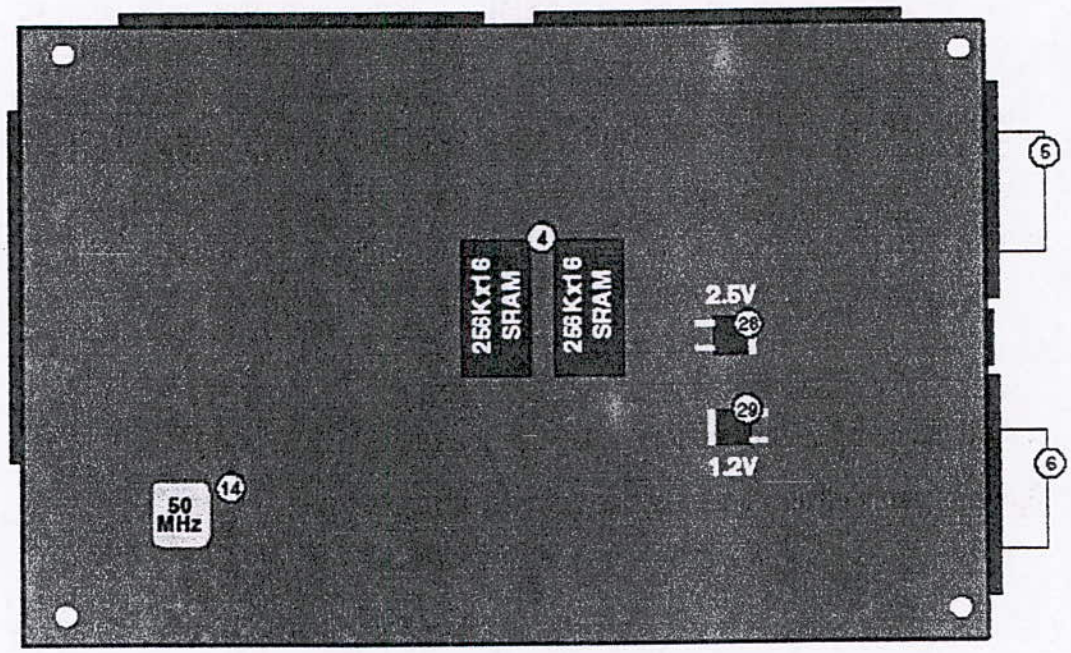


Fig.V-8: le kit Spartan 3 vu d'en bas

V-4 – Troisième partie : La plateforme logicielle

Dans cette partie nous exposerons quelques notions concernant les systèmes embarqués. On présentera ensuite les deux systèmes temps réel qui ont particulièrement attiré notre attention : Ce sont uClinux, qui est sans aucun doute l'un des systèmes Linux open source embarqué le plus populaire, et Xilkernel, le noyau temps réel disponible dans l'environnement EDK.

V-4-1- Généralités sur les systèmes embarqués [8]

Un système embarqué peut être vu comme un système électronique et informatique autonome ne possédant pas des entrées/sorties standards comme un clavier ou un écran d'ordinateur. Le système matériel et l'application sont intimement liés et noyés dans le matériel et ne sont pas aussi facilement discernables comme dans un environnement de travail classique de type PC.

On peut citer comme exemples de systèmes embarqués :

- Un four à micro ondes.
- Une télécommande de TV.
- Une fusée.
- Un missile.
- ...

V-4-1-1- Définition d'un système d'exploitation [8]

De manière générale le système d'exploitation cache les détails du hardware fondamental. Il fournit les appels systèmes en étant une interface avec les couches supérieures. Autrement dit un système d'exploitation est un ensemble de programmes permettant :

1. de gérer les ressources matérielles en assurant leur partage entre un ensemble plus au moins grand d'utilisateurs.
2. d'assurer un ensemble de services en présentant aux utilisateurs une interface mieux adaptée à leurs besoins que celle de la machine physique.

V-4-1-2- Définition d'un système embarqué [7], [8]

Il est fréquent d'entendre la terminologie de *système* embarqué : cette terminologie désigne le plus souvent un *système d'exploitation*, version complexe et multi usage du logiciel. En général, un logiciel embarqué (*embedded software* en anglais) est un programme utilisé dans un équipement industriel ou bien de consommation. La différence essentielle avec un logiciel classique tient à la complète intégration du logiciel embarqué dans cet équipement : il n'a pas de raison d'être fonctionnel en dehors de l'équipement pour lequel il a été conçu. On parle également de logiciel *intégré* ou *dédié*. L'équipement est valorisé uniquement par son aspect *fonctionnel* et un bon logiciel intégré le sera à un tel point qu'on finira par l'oublier.

Un système embarqué est dit *temps réel* lorsque on y trouve enfoui un système d'exploitation ou un noyau Temps Réel (*Real Time Operating System*, RTOS).

V-4-1-3-Caractéristiques d'un logiciel embarqué [7]

Les principales caractéristiques que doit posséder un logiciel embarqué sont :

- **ciblé :** son domaine d'activité est limité aux fonctions pour lesquelles il a été créé.
- **Fiable et sécurisé :** le logiciel nécessite une grande fiabilité car il est destiné à un fonctionnement complètement autonome.
- **maintenable dans le temps :** il est indispensable que le logiciel embarqué soit maintenable durant toute la durée de vie du produit en cas de découverte de problème de fonctionnement majeur.
- **Spécifique :** l'interface de dialogue avec l'utilisateur est spécifique : dans la majorité des cas un tel logiciel n'utilise pas les interfaces classiques clavier/souris propres à la micro-informatique.
- **Optimisé :** le plus souvent, mais ce n'est pas obligatoire, ce logiciel est de petite taille si on le compare aux volumes démesurés atteint par les logiciels classiques multi usages comme en bureautique. La raison en est double :
    - La nécessité de fiabilité citée précédemment cohabite mal avec un volume démesuré : plus on écrit de lignes de codes, plus on a des chances que celles si contiennent des bogues.

- Le logiciel est souvent embarqué dans des équipements produits à grande échelle sur lesquels le moindre écart de coût dû à une taille imprévue du logiciel peut avoir de fortes répercussions.

V-4-1-4- Le but d'utilisation d'un noyau [17]

Plusieurs facteurs nous mènent à choisir d'utiliser un noyau temps réel pour une conception donnée en particulier pour les systèmes embarqués, parmi ces facteurs, on peut citer les suivants :

- Les boucles de contrôle dans un système embarqué sont très difficiles à développer quand le nombre de tâches nécessaires est assez grand et les performances de telles applications diminuent dramatiquement quand les complexités du système augmentent.
- Diviser les tâches en plusieurs applications individuelles et les implémenter dans un système d'exploitation est plus intuitif, spécialement quand la complexité des applications augmente.
- Plusieurs applications communes dépendent des services d'un système d'exploitation tel que le management du temps (l'ordonnancement).

V-4-2- Le choix uClinux [7]

V-4-2-1- Linux comme système embarqué [7]

Linux depuis presque 3 ans est en train de conquérir un domaine où on ne l'attendait pas vraiment : l'univers des systèmes embarqués.

Les raisons pour lesquelles on trouve Linux dans le monde de l'embarqué sont diverses. Tout d'abord pour ses qualités qu'on lui reconnaît maintenant dans l'environnement plus standard du PC grand public :

- Libre, disponible gratuitement au niveau source : pas de royalties à reverser.
- Ouvert.
- Différentes distributions proposées pour coller au mieux à un type d'application.
- Stable et efficace.
- Aide rapide en cas de problèmes par la communauté Internet des développeurs Linux.

- Nombre de plus en plus important de logiciels disponibles.
- Connectivité IP en standard.

Linux a aussi d'autres atouts très importants dans le domaine plus feutré des systèmes embarqués :

- Portage sur processeurs autres que x86 : PowerPC, ARM, MIPS, 68K, ColdFire...
- Taille du noyau modeste compatible avec les tailles de mémoires utilisées dans un système embarqué.
- Différentes distributions proposées suivant le domaine : routeur IP, PDA, téléphone...
- Support du chargement dynamique de modules qui permet d'optimiser la taille du noyau.
- Migration rapide et en douceur pour un spécialiste Linux à Linux embarqué ; ce qui réduit les temps de formation (et les coûts...).

On retrouve généralement aussi un certain nombre de suppressions de fonctionnalités dans les distributions Linux embarqué :

- Pas de gestion de la MMU (*Memory Management Unit*) pour ne pas pénaliser les performances globales du système.
- Utilisation de systèmes de fichiers en mémoire RAM (*RAM Disk*) ou en mémoire FLASH *(FLASH Disk)*.

Il existe cependant des cas dans lesquels Linux sera inadapté. Si le système à embarquer nécessite seulement des fonctions de base n'incluant pas de support réseau ni de multitâche et si cet équipement n'est pas destiné à évoluer, il n'est pas nécessaire d'utiliser un système aussi riche que Linux.

V-4-2-2- Linux et le temps réel [7]

Linux standard n'est pas un système d'exploitation Temps Réel (dur) car :

- Le noyau Linux possède de longues sections de code où tous les événements extérieurs sont masqués (sections non interruptibles).
- Le noyau Linux n'est pas préemptible durant le service d'une interruption (ISR), d'ou un temps de latence important et non borné fatal à un système Temps Réel.

- L'Ordonnanceur de Linux essaye d'attribuer de façon équitable le CPU à l'ensemble des processus (ordonnancement de type *old aging* mis en oeuvre pour favoriser l'accès CPU aux processus récents). C'est une approche égalitaire. Un Ordonnanceur Temps Réel donnera toujours la main à la tâche de plus forte priorité prête.

V-4-2-3- Extensions temps réel pour Linux [7]

Il existe deux solutions permettant d'améliorer les performances temps réel de Linux.

a)- La première solution :

Elle consiste à modifier le noyau Linux standard par application de patchs.
Les développeurs ont mis au point deux principaux patchs permettant d'améliorer la réactivité du noyau Linux :

- ***Patch Preempt Kernel***

Il permet d'améliorer le comportement du noyau Linux en réduisant les temps de latence de ce dernier. A chaque fois qu'un événement apparaît et rend un processus de plus forte priorité prêt, le noyau préempte le processus courant et exécute le processus de plus forte priorité.

- ***Patch Low Latency***

Le principe est un peu différent car au lieu d'opter pour une stratégie systématique du noyau tout préemptif, les développeurs du patch ont préféré effectuer une analyse du code source du noyau afin d'ajouter des points de préemption obligatoire (appel de schedule ()) subtilement placés dans les sources du noyau afin de casser des boucles non préemptibles trop longues.

Ces modifications ne transforment pas Linux en noyau temps réel « dur » mais permettent d'obtenir des résultats satisfaisants dans le cas de contraintes temps réel « molles ».

b)- La deuxième solution :

C'est le noyau temps réel auxiliaire. Les promoteurs de cette technologie considèrent que le noyau Linux ne sera jamais véritablement temps réel sans l'ajout d'un Ordonnanceur temps réel à priorités fixes. Ce noyau auxiliaire traite directement les tâches temps réel et délègue les autres tâches au noyau Linux, considéré comme la tâche de fond de plus faible

priorité. Cette technique permet de mettre en place des systèmes temps réel « durs ». Cette technologie est utilisée par RTLinux et RTAI.

V-4-2-4- Les systèmes embarqués basés sur Linux [7]

Il existe d'ores et déjà un grand nombre de distributions embarquées basées sur Linux dont la majorité est constituée de projets Open Source.

PeeWee Linux

Utilise une version standard du noyau 2.2 et n'inclut pas de fonctionnalités temps réel.

RTLinux et RTAI

RTLinux est projet open source qui a pour but d'ajouter à un noyau Linux standard un noyau temps réel préemptif et à priorités fixes .Ce petit noyau temps réel, chargeable dynamiquement dans le système, considère le noyau Linux comme la tache de plus faible priorité. RTAI est basé sur le même principe que RTLinux mais a ajouté quelques modifications concernant l'interface de programmation.

TUXIA

Comme son nom l'indique, TUXIA développe une distribution embarquée adaptée aux applications de type *Internet Appliances* (équipements d'accès à internet).

Embedix

C'est un produit commercial initialement non destiné aux applications temps réel dures, il est aujourd'hui complété pour cela.

Hard Hat Linux

Présenté comme principalement bien adapté aux applications temps réel dures. A la différence de RTLinux et RTAI, le noyau Linux est modifié afin de le rendre plus préemptif. Comme tous les dérivés de Linux, c'est un produit Open Source.

uClinux

C'est le portage du noyau Linux pour micro contrôleurs et processeurs dépourvus de MMU. Nous développerons ultérieurement avec plus de détails cette distribution embarquée basée sur Linux.

Tableau récapitulatif des systèmes embarqués [7]

Nom	Editeur	Open Source	Temps réel	Remarques
VxWorks	WindRiver	non	oui	C'est le leader du marché mondial de l'embarqué
pSOS	WindRiver	non	oui	Ancien mais très répandu
QNX	QNX	partiellement	oui	Basé sur Unix, assez répandu, bonnes performances
uC/OS	Micrium	non	oui	Pour les microcontrôleurs
Windows CE	Microsoft	non	oui	
LynxOS	LynuxWorks	non	oui	
Nucleus	Accelerated technology	oui	oui	
ECOS	Red Hat	oui	oui	Utilisable pour de très faibles empreintes mémoire, environnement de développement croisé[5] Linux disponible
PeeWeeLinux	aucun	oui	non	Fondé sur Red Hat 6.2, Noyau 2.2
RTLinux	FSMLabs	oui	oui	Temps réel dur

[5] Un environnement de développement croisé permet de développer pour le système cible sur une autre machine dans un environnement plus confortable.

RTAI	aucun	oui	oui	Proche de RTLinux mais non commerciale
Red Hat Embedded Linux	Red HAT	oui	non	
uClinux	aucun	oui	oui	Pour microcontrôleur sans MMU, existe en noyau 2.0 ,2.4 et 2.6
Embedix	Lineo	oui	non	Utilisé dans le PDA[6] de SHARP
TUXIA	TUXIA	oui	non	Dédié « internet appliance »
Montavista Linux(Hard Hat)	Montavista	oui	oui	Fondé sur des patchs préemptif du Noyau Linux, pas de temps réel dur

V-4-2-5- Le projet uClinux [7]

Parmi l'ensemble des distributions Linux embarqué, notre attention particulière s'est basée sur uClinux qui est de plus en plus sollicité de nos jours. Ce choix s'est basé essentiellement sur les performances temps réels offerte par uClinux, sa faible empreinte mémoire, sa portabilité sur le processeur utilisé dans notre application (Microblaze) en plus de beaucoup d'autres critères que nous citerons plus bas.

UClinux (prononcer " *you see Linux* ") est l'acronyme de *Micro Controller Linux.* C'est le portage du noyau Linux pour microcontrôleurs et processeurs dépourvus de MMU[7].De ce fait il n'y a pas de protection mémoire d'un programme à l'autre et un programme peut « planter » un autre programme ou le noyau lui-même.

Le projet uClinux lancé en janvier 1998 est un portage de Linux version 2.0.x originellement sur des processeurs ne possédant d'unité de gestion mémoire MMU. C'est en fait à la base un portage de Linux sur microcontrôleur Motorola 68328. Outre la famille

[6] *Personnal digital assistant.*
[7] *Memory managment unit*, le concept MMU sera développé plus loin.

Motorola 683xx, il existe maintenant des portages uClinux pour processeurs Motorola ColdFire, i960 d'Intel, ARM7TDMI et depuis peu pour MicroBlaze.

V-4-2-6- Le concept MMU [7]

Linux a été initialement développé sur la base du mécanisme de mémoire protégé du processeur Intel 80386. Ce mécanisme, qui repose sur un composant matériel appelé MMU (*memory management unit*), permet à un processus de ne jamais écraser l'espace mémoire d'un autre processus. La MMU autorise la conversion entre les adresses physiques (adresses effectivement utilisées dans la machine) et les adresses logiques (adresses vues par le processus et allouées par le système d'exploitation. Si un système tente de sortir par erreur de l'espace mémoire qui lui est accordé , la MMU détecte l'erreur est arrête le programme .De ce fait un programme tournant sous Linux dans l'espace dit « utilisateur » (par opposition à l'espace « noyau ») ne peut jamais « planter » le système.

Pour les systèmes embarqués, il est important de réduire l'empreinte mémoire, pour cela la version uClinux prévu pour être embarquée sur des microcontrôleurs n'utilise pas le module MMU.

L'absence de MMU impose quelques limitations d'usage par rapport à l'environnement Linux. En effet, la mémoire virtuelle n'existe pas, et la taille de la pile est fixe pour chaque processus.

V-4-2-7- Avantages de uClinux par rapport à ces concurrents

Le plus grand avantage que possède uClinux par rapport à ses concurrents est la compatibilité de l'interface de programmation API[8] avec les systèmes linux standard. Il dispose également de toutes les fonctionnalités réseau TCP/IP disponibles sur le noyau Linux, ce qui est aussi un grand avantage dans le cas de développement d'un produit communicant. Le système uClinux est également très raisonnable au niveau de l'empreinte mémoire.

Pour ce qui est des performances temps réel, il existe aussi un portage de RTLinux pour le noyau uClinux 2.0 pour avoir finalement un système embarqué Temps Réel dur.

[8] *Acronyme de « application programming interface »*

V-4-3- Le choix Xilkernel [17]

V4-3-1-Présentation du noyau Xilkernel

Le kit de développement embarqué EDK fournit des librairies, des systèmes d'exploitation complets en plus des pilotes de périphériques pour aider l'utilisateur à développer ses applications.

V-4-3-2- Définition du BSP [17]

Le BSP (Board Support Package) est la couche la plus basse des modules software utilisé pour accéder aux fonctions spécifiques du processeur il forme avec les pilotes Libxil, l'abstraction matériel de plus bas niveau. Le BSP de Standalone est au dessous de la couche système d'exploitation, et il est utilisé quand une application peut avoir l'accès directement aux caractéristiques du processeur.

Quand le système utilisateur contient un processeur MicroBlaze et aucun système d'exploitation n'est spécifié, le générateur de librairies LibGen construit le BSP de Standalone dans la librairie Libxil.a.

Le noyau Xilkernel se base sur le BSP de Standalone fournit sur l'environnement EDK. En effet, Ce BSP est indispensable au noyau Xilkernel car il représente « l'arête principale » du noyau.

V-4-3-3- Le noyau Xilkernel [17]

Xilkernel, est un noyau embarqué qui peut être personnalisé pour une large gamme de systèmes. Il possède les caractéristiques clef d'un noyau temps réel comme le multitâche, un Ordonnanceur préemptif piloté par les priorités, une communication inter processus, une facilité de synchronisation, une prise en charge des interruptions, etc.

Xilkernel est un petit noyau robuste, modulaire, personnalisé par l'utilisateur et peut être utilisé dans différentes configurations du système. C'est une librairie software libre qu'on obtient avec EDK contenant des fonctions prédéfinies telle une fonction de validation d'interruption, il permet un degré de personnalisation assez important permettant à l'utilisateur d'adapter le noyau à un niveau optimal en terme de taille et de fonctionnalités. Il

supporte les caractéristiques d'un noyau temps réel nécessaire pour les systèmes embarqués, et fournit à l'Ordonnanceur des primitives de synchronisation et de communication *inter-process* pour permettre à la conception une exécution (co-opérative) des tâches.

V-4-3-4- L'organisation de Xilkernel [17]

Xilkernel est un noyau modulaire, on peut sélectionner et personnaliser les modules du noyau nécessaire pour une application donnée. La figure suivante montre les différents modules de Xilkernel.

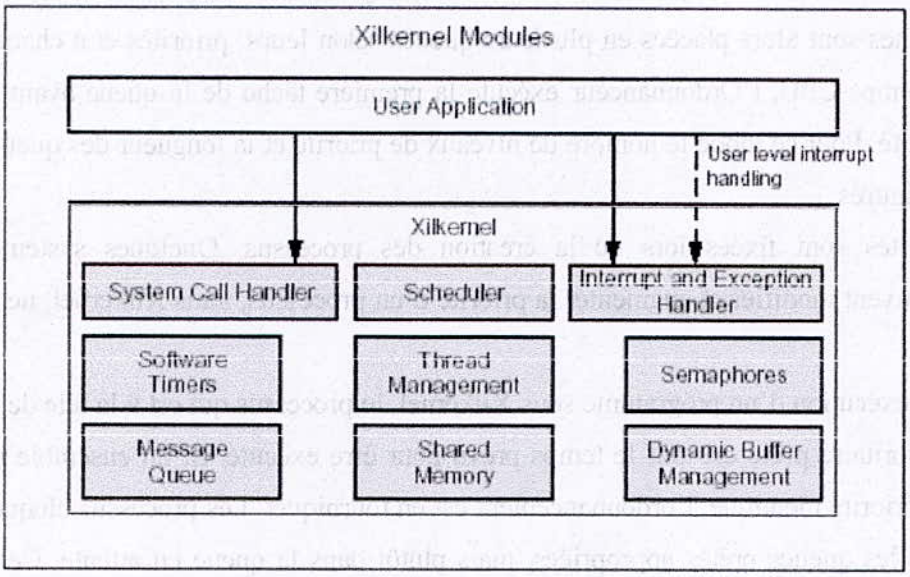


Fig.V-9 : organisation de Xilkernel

V-4-3-5- Gestion des processus [17]

Les systèmes d'exploitation exécutent plusieurs applications utilisateurs comme des processus indépendants. Un *timer* matériel interrompe périodiquement le processeur et fait appel à l'Ordonnanceur qui détermine quel processus doit être exécuté. L'intervalle de temps entre ces appels est appelé *time slice* et a chaque *time slice,* l'Ordonnanceur sélectionne un processus à exécuter.

Sans un système d'exploitation, les applications doivent être combinées dans une seule fonction énorme, difficile à mettre en œuvre. Cependant, avec un noyau, chaque application

devient un processus indépendant et peut être développé et testé indépendamment de l'autre.

Xilkernel supporte deux modes d'ordonnancement, l'ordonnancement basé sur la priorité *(priority based)* et l'ordonnancement en tourniquet *(round robbin).* Ceci doit être configuré statiquement au moment de la génération du noyau.

L'algorithme d'ordonnancement le plus simple pour Xilkernel est l'Ordonnanceur en tourniquet, qui met tous les processus dans une seule queue et accorde le *time slice* pour la première tâche sans prendre en compte aucune priorité. Dans ce mode, une seule queue est prêtre à être exécuté et chaque processus s'exécute durant un *time slice* avant de louer l'exécution au processus suivant.

Dans le mode basé sur les priorités, on a autant de queues prêtes que de niveaux de priorités. Les tâches sont alors placées en plusieurs queues selon leurs priorités et à chaque interruption du temps CPU, l'Ordonnanceur exécute la première tâche de la queue ayant la plus grande priorité. Pour ce mode le nombre de niveaux de priorité et la longueur des queues doivent être configurés.

Les priorités sont fixées lors de la création des processus. Quelques systèmes d'exploitation peuvent modifier et augmenter la priorité d'un processus, mais Xilkernel, ne le fait pas.

Lors de l'exécution d'un programme sous Xilkernel, le processus qui est à la tête de la queue la plus prioritaire prête est tout le temps prévu pour être exécuté. Si un ensemble de processus a une priorité identique, l'ordonnancement est en tourniquet. Les processus bloqués ne sont pas dans les queues prêtes appropriées, mais plutôt dans la queue en attente. Cette dernière est configurée comme une queue de priorité si l'ordonnancement est basé sur les priorités, autrement, elle est configurée comme une queue FIFO.

Chaque processus peut être dans l'un des états suivants :

- PROC_NEW : un nouveau processus crée
- PROC_READY : un processus prêt à être exécuté
- PROC_RUN : un processus en exécution (actif)
- PROC_WAIT : un processus bloqué dans une ressource
- PROC_DELAY : un processus en attente d'un *timeout*
- PROC_TIMED_WAIT : un processus bloqué dans une ressource et ayant un *timeout.*

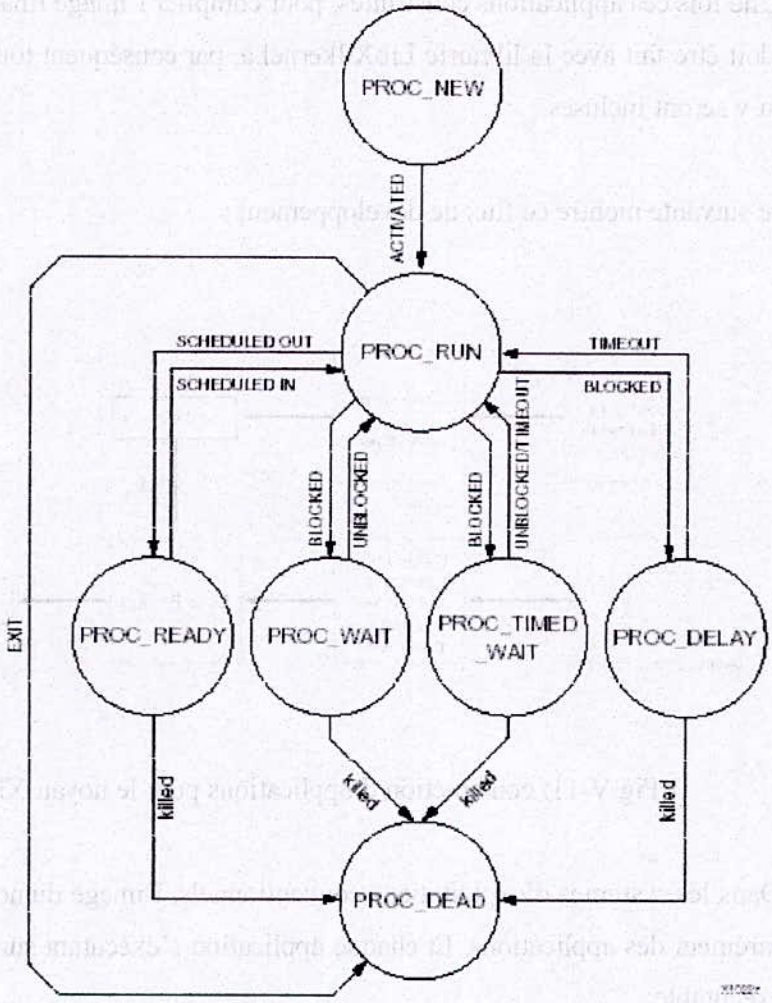


Fig.V-10 : les états des processus

V-4-3-6- Flux de conception sous Xilkernel

V-4-3-6-1- Construction des applications sous Xilkernel [18]

Xilkernel est organisé sous la forme d'une librairie de fonctions (voir l'annexe N°). Pour construire l'image du noyau, Xilkernel doit être inclus dans la plateforme software, il doit être configuré convenablement, pour exécuter le générateur de librairies de EDK LIBGEN et obtenir la librairie LibXilkernel.a.

Les applications peuvent être développées séparément ou bien au sein même de la plateforme software XPS.

Une fois ces applications construites, pour compiler l'image finale du noyau Xilkernel, un lien doit être fait avec la librairie LibXilkernel.a, par conséquent toutes les fonctionnalités du noyau y seront incluses.

La figure suivante montre ce flux de développement :

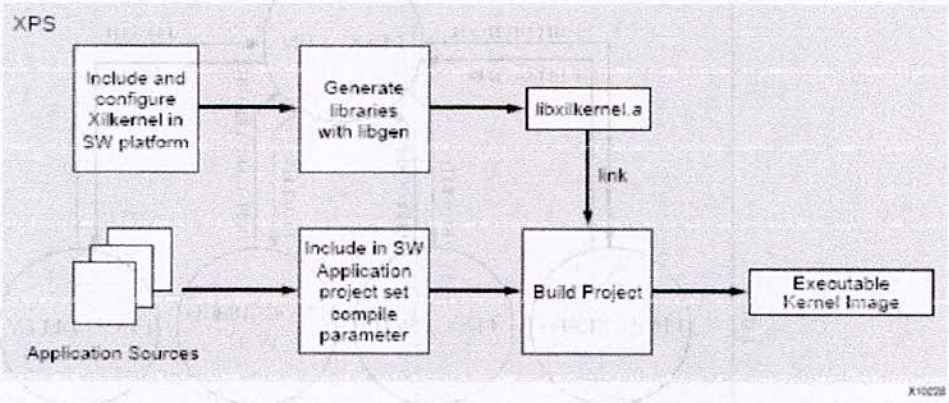


Fig.V-11: construction d'applications pour le noyau Xilkernel

Dans les systèmes d'exploitations conventionnels, l'image du noyau est un exécutable crée séparément des applications. Et chaque application s'exécutant sur le noyau possède son propre exécutable.

Cependant pour faciliter le debuggage, le chargement sur la carte et le chargement sur la SRAM, XPS nous permet de créer un même exécutable contenant et l'image du noyau et les exécutables des applications.

De cette manière, une fois la librairie LibXilkernel.a créée, et les paramètres de compilation des applications ajustés, le lien entre LibXilkernel.a et les applications est créé pour finalement compiler le tout est obtenir un seul exécutable.

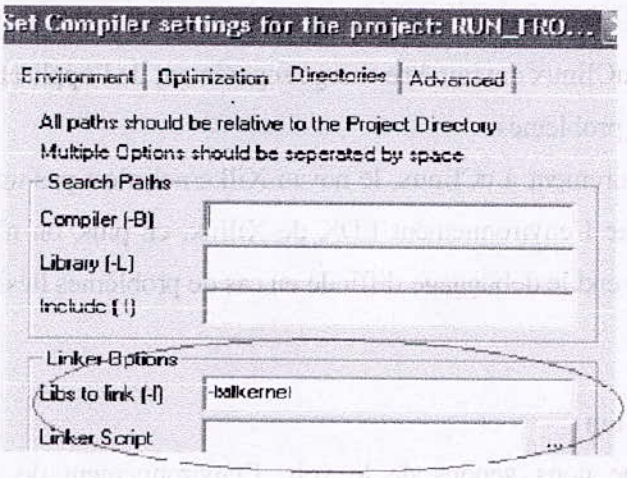
V-4-3-6-2- Etapes essentielles au développement d'une application sous Xilkernel [18]

- Les fichiers sources C des applications doivent inclure la librairie xmk.h. Cette librairie doit être mentionnée avant toutes les autres.

Par exemple : #include "xmk.h"

Définir ce drapeau rend disponible certaines définitions et déclarations exigées par l'utilisation du noyau Xilkernel.

- un lien entre l'application software et la librairie LibXilkernel.a doit être établi, et ce en incluant « Xilkernel » dans la liste des librairie avec lesquelles des lien sont crées. Ceci est traduit par l'ajout de l'option *–lxilkernel* dans les paramètres du compilateur comme suit :



- Xilkernel est responsable de toutes les prises en charge des interruptions et des exceptions du premier niveau pour MicroBlaze et Power PC. Par conséquent, les méthodes de prise en charge des interruptions et des exceptions documentées pour programmer Standalone ne sont plus valides.

- L'application doit fournir une fonction *main()* qui est le point de départ de l'exécution de l'image du noyau .Toutes les initialisations et les configurations voulues doivent être faites au sein de cette fonction. Le noyau reste *dormant*, jusqu'à ce que l'on veuille le lancer et ce par l'utilisation de la fonction *xilkernel_main()* qui active le noyau et valide les interruptions.

V-4-3-7- Avantages et inconvénients de Xilkernel par rapport à uClinux

L'avantage principal que représente le noyau Xilkernel par rapport à uClinux est sa parfaite optimisation pour être embarqué sur le processeur virtuel MicroBlaze.

En effet, ce noyau est disponible sur l'environnement EDK, et la compilation de son image se fait aussi sur EDK, ce qui nous permet d'obtenir une image sur mesure pour notre processeur.

A titre d'exemple, la taille d'un noyau Xilkernel fonctionnel est de l'ordre de quelques Kilo-octets, alors que pour un noyau uClinux il est difficile d'obtenir une image fonctionnelle de taille inférieure à une centaine de Kilo-octet. Ces détails deviennent importants lorsque l'espace mémoire disponible sur la carte est réduit.

Ajouté à cela, notons la facilité avec laquelle Xilkernel est chargé sur la carte du moment ou il est contenu dans le même fichier exécutable que l'application software, alors que le noyau uClinux devrait être chargé séparément de l'application software, ce qui peut par fois créer des problèmes.

Contrairement à uClinux, le noyau Xilkernel n'est pas un noyau open source, il n'est fournit qu'avec l'environnement EDK de Xilinx, en plus, on n'a pas accès aux sources du noyau ce qui rend le debuggage difficile en cas de problèmes liés au noyau lui-même.

V-5- Conclusion

Comme nous venons de le voir, l'environnement de développement EDK prend complètement en charge et la conception matérielle et la conception logicielle. Ce qui en fait un outil indispensable pour le co-design.

Malgré tous ses avantages, l'une des plus grandes critiques de EDK est le fait qu'il n'inclut pas le code source du processeur virtuel MicroBlaze, puisque ce dernier est propriétaire. Ce qui rend la simulation et le déboguage plus difficiles car il est impossible de savoir ce qui se passe exactement au niveau du processeur.

L'utilisation d'un processeur virtuel fournit plusieurs avantages par rapport aux processeurs matériels. Contrairement à ces derniers qui ont une conception figée, les processeurs virtuels peuvent être configurés selon les besoins du développeur. On pourra alors configurer notre processeur MicroBlaze avec les périphériques et les ports nécessaires pour notre application.

Chapitre V

Mise en oeuvre de l'application

VI-1- Introduction

Le but essentiel de notre travail est d'utiliser un système d'exploitation temps réel enfoui afin de commander le robot mobile. Pour ce, on doit configurer l'architecture générale du processeur virtuel qu'on a choisi d'utiliser à savoir ses bus de données et d'instructions, ses ports d'entrée/sortie, ainsi que ses périphériques *on chip* et *off chip* d'une manière appropriée pour accueillir le noyau temps réel et assurer la commande du robot mobile. Ensuite, on doit compiler le noyau temps réel pour pouvoir le charger sur la mémoire du processeur, ce qui nous permettra de programmer notre application de telle manière que le robot exécute ses tâches en satisfaisant le concept du temps réel.

Pour pouvoir commander le robot, on doit le connecter à la carte Spartan 3 qui contient le circuit FPGA lui-même contenant le processeur virtuel MicroBlaze. Ce dernier est gouverné par un système d'exploitation temps réel qui permet de gérer l'application écrite en langage C. Celle-ci se compose de plusieurs tâches indépendantes qui doivent être exécutées en alternance. De cette manière, on aura obtenu le modèle concentrique représenté par la figure suivante :

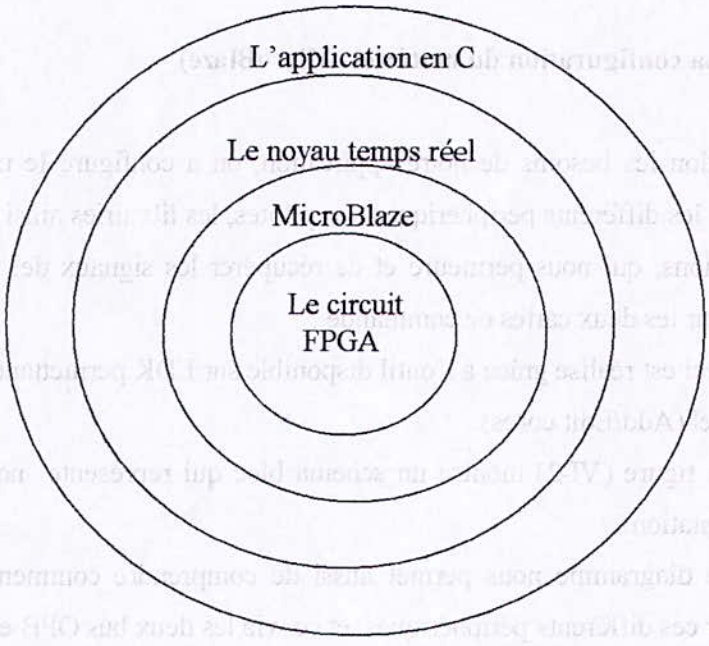


fig.VI-1 Le modèle concentrique

VI-2- Adaptation de la carte

La stratégie générale utilisée pour la commande de notre robot mobile consiste à envoyer deux signaux de validation vers les deux émetteurs pour choisir entre détecter les obstacles qui se trouvent sur le coté droit ou bien ceux qui se trouvent sur le coté gauche, de récupérer le signal provenant du récepteur détectant un obstacle, et finalement envoyer vers les deux cartes de commande des moteurs deux signaux : un signal commandant la marche et l'arrêt, et un deuxième commandant le sens de rotation du moteur.

Il en résulte un besoin immédiat d'adapter notre carte pour qu'elle soit apte à générer et à interpréter tous ces signaux correctement.

Sur le plan matériel, cette adaptation se traduit par :

- La configuration du matériel (MicroBlaze) par le choix des périphériques adéquats.
- La modification du fichier UCF afin de nous permettre d'inclure dans notre conception les signaux d'entrée et de sortie pour commander le robot.

VI-2-1- La configuration du matériel (MicroBlaze)

Selon les besoins de notre application, on a configuré le processeur MicroBlaze en spécifiant les différents périphériques, les pilotes, les librairies ainsi que les bus de données et d'instructions, qui nous permettre et de récupérer les signaux des capteurs et d'en envoyer d'autres sur les deux cartes de commande .

Ceci est réalisé grâce à l'outil disponible sur EDK permettant l'ajout et la modification du matériel (Add/Edit cores).

La figure (VI-2) montre un schéma bloc qui représente notre plateforme matérielle après adaptation.

Ce diagramme nous permet aussi de comprendre comment agit MicroBlaze pour gouverner ces différents périphériques, et ce via les deux bus OPB et LMB.

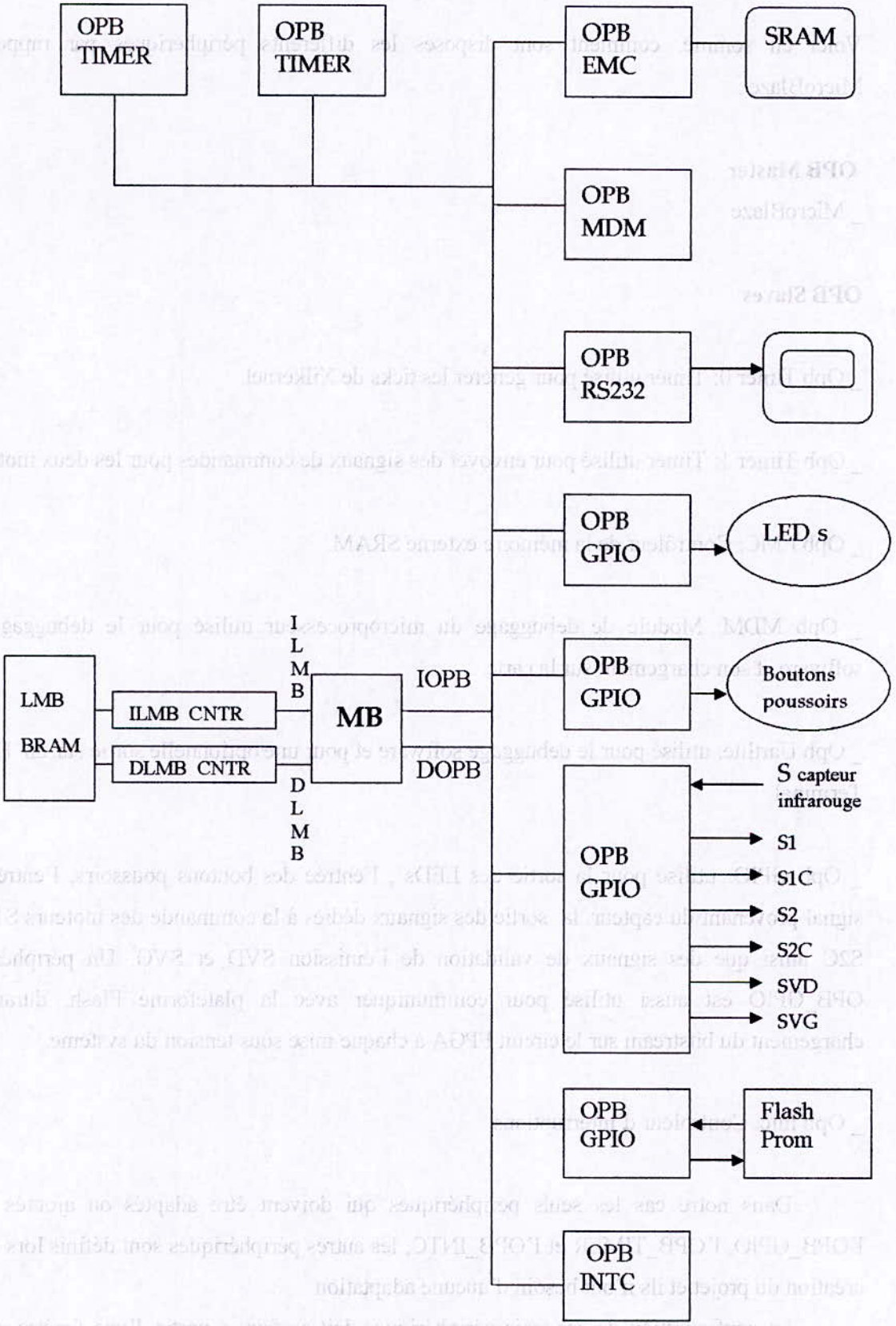


Fig. VI-2 : Bloc Diagramme

Voici en somme, comment sont disposés les différents périphériques par rapport à MicroBlaze :

OPB Master

_ MicroBlaze

OPB Slaves

_ Opb Timer 0: Timer utilisé pour générer les ticks de Xilkernel.

_ Opb Timer 1: Timer utilisé pour envoyer des signaux de commandes pour les deux moteurs.

_ Opb EMC: Contrôleur de la mémoire externe SRAM.

_ Opb MDM: Module de debuggage du microprocesseur utilisé pour le debuggage du software et son chargement sur la carte.

_ Opb Uartlite: utilisé pour le debuggage software et pour une optionnelle sortie sur un Hyper Terminal.

_ Opb GPIO: utilisé pour la sortie des LEDs , l'entrée des boutons poussoirs, l'entrée du signal provenant du capteur, la sortie des signaux dédiés à la commande des moteurs S1C et S2C ainsi que des signaux de validation de l'émission SVD et SVG. Un périphérique OPB_GPIO est aussi utilisé pour communiquer avec la plateforme Flash, durant le chargement du bitstream sur le circuit FPGA à chaque mise sous tension du système.

_ Opb Intc: Contrôleur d'interruptions.

Dans notre cas les seuls périphériques qui doivent être adaptés ou ajoutés sont l'OPB_GPIO, l'OPB_TIMER et l'OPB_INTC, les autres périphériques sont définis lors de la création du projet et ils n'ont besoin d'aucune adaptation.

La configuration de ces trois périphériques doit se faire à partir d'une fenêtre qu'on obtient lors de l'appel de l'outil Add/Edit cores du flux de configuration de EDK. Grâce à cet

outil, La fréquence de commutation du TIMER, les différentes pins ou périphériques *off_chip* qui doivent être connectés aux ports du processeurs MicroBlaze via des OPB_GPIOs ainsi que l'ordre de priorité des interruptions sont spécifiés.

VII-2-1-1- L'OPB_timer :

Le timer est un périphérique on chip indispensable dans le cas de l'utilisation d'un noyau car il permet de générer les ticks du système d'exploitation. L'intervalle entre chaque deux ticks étant le time slice que le noyau alloue à chaque tache lorsqu'il ordonnance en mode round robbin.

Dans le cas de MicroBlaze, les timers ainsi que tous les autre périphériques on chip, sont fournis sous forme de IP cores virtuels propriétaires, qui sont décrits par un code VHDL, et qui ne peuvent être accessibles par l'utilisateur.

Le Timer/compteur est organisé en deux modules timers identiques. A chaque module timer est associé un registre de chargement dédié à contenir soit la valeur initiale du compteur dans le cas du mode *generate*, soit la valeur à capturer dans le cas du mode *capture*. Le diagramme bloc du Timer/compteur est le suivant :

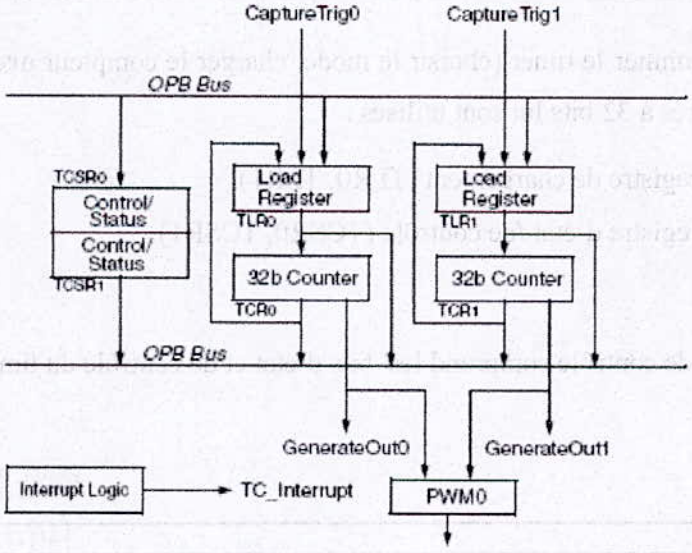


Fig. VI-3 : Organisation interne du Timer/compteur

MicroBlaze nous permet d'utiliser autant de timers qu'on désire avec possibilité d'utiliser deux modules timers à la fois dans chacun et ce pour trois modes de fonctionnement différents :

- Mode *capture* : qui permet de mesurer le temps que met un événement extérieur pour se produire.
- Mode *generate* permet de générer des interruptions à des intervalles réguliers.
- Mode *PWM* utilise les deux modules timers pour générer un signal modulé en fréquence.

Le mode generate :

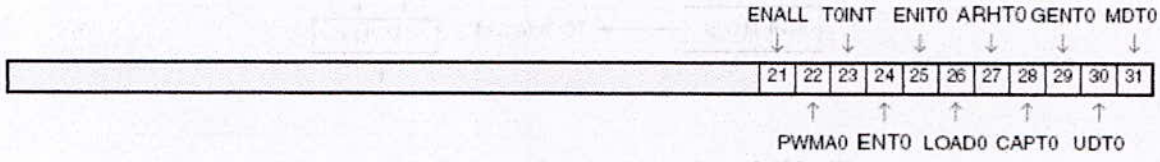
Dans ce mode, la valeur du registre de chargement est placée dans le compteur. Une fois que celui ci est validé il commence aussitôt à compter ou à décompter. Le compteur s'arrête automatiquement à la transition de son bit carry, recharge la valeur qui se trouve dans le registre de chargement et recommence à compter ou à décompter.

Si le signal d'interruption est validé, ce mode est utilisé pour générer une interruption vers l'extérieur à chaque fois que le bit carry du compteur subit une transition. Il est utile lorsqu'on veut générer des interruptions répétitives vers l'extérieur.

Pour programmer le timer (choisir le mode, charger le compteur avec une valeur donnée,...) deux registres à 32 bits lui sont utilisés :

- Le registre de chargement (TLR0, TLR1).
- Le registre d'état /de contrôle (TCSR0, TCSR1)

Le registre de contrôle comprend les bits d'état et de contrôle du timer :



MDT : désigne le mode du timer (1: generate, 0: capture)

UDT: 0: Compteur ,1: décompteur

GENT: valide la génération de signaux vers l'extérieur.

CAPT : valide la capture d'un signal de l'extérieur

ARHT valide l'auto rechargement du timer

LOAD: chargement du compteur par la valeur du registre de chargement

ENIT: Valide le mode interruptible

ENT: Valide le timer.

TINT: indique que la condition que le timer attendait pour envoyer une interruption s'est produite

PWMA: valide le mode PWM

ENALL: valide tout les timers.

Notre programmation écrite en langage évolué et non en assembleur, nous n'avons pas accès à ces registres la, mais l'environnement EDK fournit une librairie de fonction permettant de charger le registre de chargement et le registre de contrôle (cette librairie est nommé xtmtctr_l.h).

Parmi ces fonctions nous citons :

XTmrCtr_mWriteReg : qui nous permet de charger le registre de chargement par la valeur voulue.

XTmrCtr_mSetControlStatusReg : qui nous permet de charger le registre de contrôle par une valeur correspondante au choix du mode du timer, à la validation des interruptions, … etc.

Pour notre conception, en plus du timer qui sera dédié à la génération des ticks pour le noyau et pour qui on ne doit que spécifier la durée du *time slice* qui sera allouée pour chaque tâche, on a besoin d'utiliser un autre Timer qui permettra de générer plusieurs signaux périodiques. Ces signaux sont :

- Un signal de 1 KHz qui servira de porteuse pour le signal de 38 kHz (généré par l'oscillateur) émis par les deux leds IR des capteurs. En effet, L'architecture du récepteur utilisé ne lui permet pas de détecter des signaux de fréquence 38Khz continus. Les deux signaux superposés forment un signal en créneau *burst* d'une

largeur de 0.5 ms. En branchant la sortie du récepteur sur un oscilloscope, on obtient après détection le même signal que celui émis sur les LEDs.

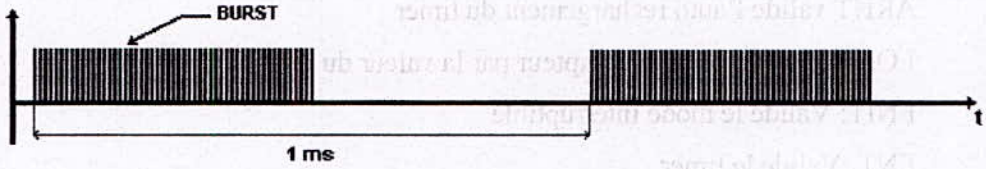


Fig.VI-4 : Signal de détection IR

- Deux signaux d'horloge pour les deux moteurs.

Ce Timer sera utilisé en mode interruptif. Il générera une interruption chaque 0.5 ms . Une fonction de prise en charge de l'interruption (handler) doit être comprise dans le programme afin de créer un signal carré d'une fréquence de 1KHz. La même interruption est utilisée pour générer les signaux d'horloge des deux moteurs.

VII-2-1-2- L'OPB_GPIO

GPIO est l'acronyme de *General Purpose Input/Output (GPIO)* qui est un périphérique on chip de MicroBlaze.

Ce périphérique est relié à MicroBlaze via le bus OPB est assure le pilotage des différentes entrée/ sorties standards et périphériques comme les leds, les boutons poussoirs et les autre périphériques présents sur la carte. Le GPIO peut aussi être utilisé pour piloter une entrée/sortie externe à la carte afin d'émettre ou de recevoir des signaux de l'extérieur.

Pour notre conception, l'utilisation des OPB_GPIO est très importante car elle nous permet de piloter les sept signaux qui proviennent des capteurs et ceux qui vont vers les deux cartes de commande du robot mobile.

VI-2-2- L'adaptation du fichier UCF

Le fichier UCF est le fichier de configuration des pins, il nous permet de raccorder les périphériques aux différentes pins du circuit FPGA.

Le fichier UCF est généré automatiquement pour les périphériques configurés lors de la création du projet (les LEDs et les boutons poussoirs), pour les périphériques ajoutés on est obligé de modifier le fichier UCF manuellement. En assignant à chaque pin sa localisation sur le circuit FPGA. Pour notre cas, on utilise sept pins du connecteur d'expansion A2 de la Spartan 3.

VI-3- Embarquer le système d'exploitation uClinux sur la carte

- Dans cette partie, on va citer les différentes étapes qu'on a suivi pour charger le noyau temps réel uClinux sur le circuit FPGA configuré avec le processeur MicroBlaze. Malheureusement, lors du chargement de l'image obtenue après compilation aucun résultat n'apparaissait sur l'écran.

  Cependant, on juge bon de citer les différentes étapes qu'on a suivi pour compiler puis charger l'image du noyau uClinux sur la carte Spartan 3.

VI-3-1- le choix de uClinux

Le noyau uClinux a été porté à une large gamme de microcontrôleurs et processeurs. Sa principale fonctionnalité est de pouvoir se passer d'un gestionnaire de mémoire, et donc de fonctionner sur des systèmes embarqués relativement simples ne comprenant qu'un processeur et quelques périphériques servant à le faire fonctionner.

Séduits par toutes ses caractéristiques, notre première intention était de compiler le noyau uClinux et de l'embarquer sur notre MicroBlaze.

VI-3-1-1- Environnement de compilation croisée (Cross Compilation)

Un environnement croisé permet de développer pour le système cible sur une autre machine dans un environnement plus confortable. On peut alors citer la disponibilité sous

linux d'un environnement de développement pour le processeur MicroBlaze, basé sur le compilateur GNU gcc.

Notre noyau doit être compilé dans un environnement de compilation croisé. Dans notre cas, il s'agit de compiler le noyau uClinux pour le processeur MicroBlaze sur une machine linux ayant pour processeur x86 de Intel.

On doit optimiser le noyau aux applications prévues, pour obtenir une image de taille minimale car notre carte ne contient que 1MO de SRAM.

VI-3-1-2- Outils indispensables à la compilation de uClinux

Nous avons tout d'abord récupéré la chaîne de développement de MicroBlaze MicroBlaze_elf_tools. Cette chaîne de développement est indispensable à notre compilation car elle contient des outils très importants comme mb-gcc, le compilateur gcc propre à MicroBlaze.

Une fois assurés de la fonctionnalité de tous ces outils, nous avons récupéré la distribution du noyau uClinux qui contient les programmes utilisateur ainsi que les librairies C, et nous avons procédé à sa compilation.

VI-3-1-3- Compiler notre propre noyau

La distribution uClinux comprend les noyaux 2.4 et 2.6 adaptés aux systèmes sans MMU. La configuration se fait comme pour une compilation classique d'un noyau Linux en utilisant d'abords la commande ***make menuconfig.***

La différence vient ensuite au niveau du menu, au lieu d'arriver directement sur le menu de configuration du noyau, l'utilisateur sélectionne le type de carte sur laquelle se fait le développement et choisit s'il veut modifier les paramètres du noyau et/ou les applications compilées dans l'image de disque à charger en RAM. Nous utiliserons comme librairie C compacte uClibc. En quittant ce premier menu les menus suivants (configuration noyau et utilitaires) apparaissent. La compilation est complétée par ***make dep*** et ***make*** cette dernière instruction place dans le répertoire **images,** une image disque prête à être transférée en RAM du système embarqué ***image.bin*** et dans le répertoire ***romfs*** elle place le contenu de cette image. Notons que le noyau s'exécute toujours à partir de la RAM pour des raisons de temps d'accès aux différents types de mémoires : même un noyau destiné à être stocké en mémoire

flash doit être défini (menu **processor type and features** de la configuration du noyau) avec ***kernel executes from RAM.***

Il existe une autre manière de configurer le noyau uClinux. Cette méthode présente le privilège de ne pas avoir à configurer manuellement les différents paramètres du noyau.
En effet, il suffit de récupérer les BSP de uClinux et de les placer à coté du répertoire de l'installation de EDK, puis créer un projet sous XPS en prenant le soin de sélectionner uClinux comme système d'exploitation, ensuite il faut exécuter la commande *update Bitstream* qui crée le fichier auto_config.in. Ce fichier interprète parfaitement les besoins du processeur et présente l'avantage d'être compréhensible par l'environnement de compilation croisé.

Une fois le fichier auto_config.in obtenu il faut le placer dans un des répertoires de uClinux_dist, par conséquent, lors de la configuration du noyau, ce fichier est pris en compte et aucune configuration manuelle ne sera demandée. Le résultat est ''une image sur mesure'' pour notre processeur.

Le premier problème qu'on a rencontré était de compiler une image d'une taille qui ne dépasse pas les 1 Mo de la SRAM de notre carte. Il était donc impératif de ne garder que les fonctionnalités essentielles. Après plusieurs essais on a pu obtenir une image d'une taille ne dépassant pas les 600 Ko.

VI-3-1-4- Chargement de l'image sur la carte

Une fois que l'image du noyau adaptée à notre processeur est obtenue, on doit la charger sur la SRAM de la carte. Pour ce faire, l'environnement EDK nous fournit le débuggeur XMD qui permet de charger des fichiers exécutables ayant l'extension .elf ainsi que des fichiers binaires fichier.bin.

L'image qui doit être chargée sur la carte est celle qui a l'extension .bin. Pour ce, il suffit de se connecter au module de déboguage via le câble JTAG et charger le noyau dans la mémoire SRAM de la Spartan 3. Mais avant de pouvoir charger la mémoire externe, on doit configurer le circuit FPGA avec le projet Bootloop pour spécifier le processeur MicroBlaze et les périphériques utilisés.

Le flux de chargement en utilisant XMD est très intéressant et il est souvent utilisé pour le déboguage, car on peut charger et recharger la mémoire en quelques secondes. Cependant, le contenu de la mémoire dans ce cas la est volatile.

VI-3-1-5- Résultat du chargement de uClinux sur la carte

Nous avons suivi cette procédure mais malheureusement on n'a obtenu aucun résultat. En effet, l'image paraissait être chargée en mémoire mais en essayant de l'exécuter on n'obtenait rien. Ceci peut être dû à la limitation en matière de mémoire imposée par la carte Spartan3 qui comprend uniquement 1Méga Octets de SRAM alors que l'image du noyau à elle seule dépasse les 500 Kilos octets pour un minimum de fonctionnalités.
Devant cette situation, il ne nous restait plus qu'embarquer Xilkernel au lieu de uClinux.

VI-4- L'utilisation du noyau temps réel Xilkernel

Contrairement à uClinux la compilation du noyau Xilkernel se fait sur l'environnement EDK lui-même, aucun besoin de relier les deux machines (machine sur laquelle se fait la compilation du noyau et la machine sur laquelle EDK est installé) n'est donc ressenti.

Lors de la configuration du noyau Xilkernel, on choisit le type de l'Ordonnanceur (basé sur la priorité ou Round Robbin), la largeur du time slice, les taches et leur ordre de priorité, la longueur des queues, l'utilisation des sémaphores,..etc.

Dans le cas de Xilkernel le noyau et l'application ont le même exécutable, ce qui réduit les problèmes de chargement de l'image sur la carte.

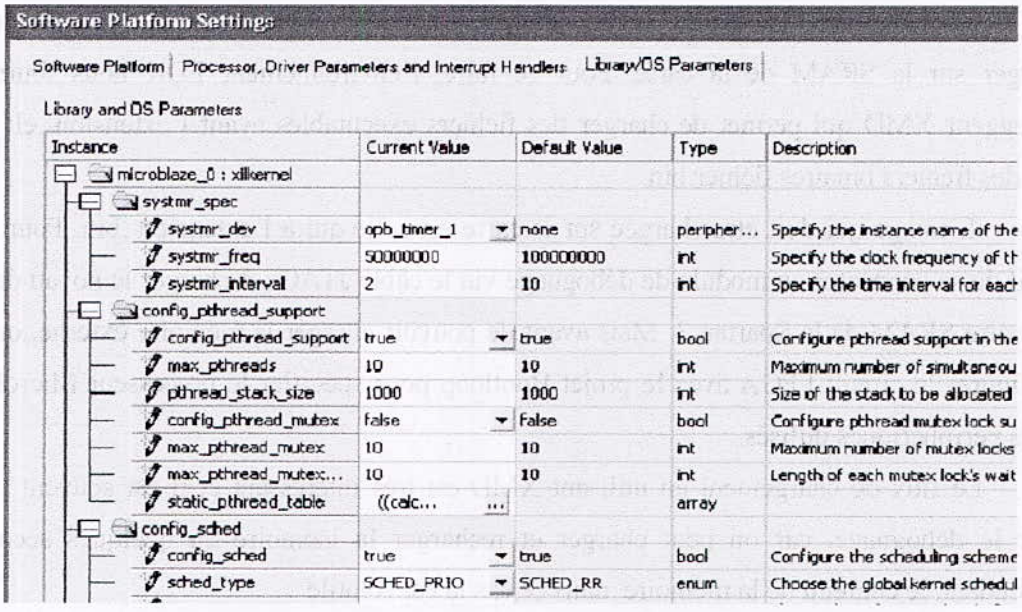


Fig. VI-5 : Configuration du noyau Xilkernel sous EDK

VI-5- L'algorithme de fonctionnement du robot mobile

L'algorithme de fonctionnement du robot est assez simple, il doit avancer droit devant (donc les deux moteurs doivent se mettre en marche simultanément et dans le même sens) en émettant le signal de fréquence 38 Khz modulé par une fréquence de 1 Khz sur les deux LEDs gauche et droite en alternance. Si un obstacle est détecté sur la droite (au moment le signal de l'émission sur le coté gauche), on laisse le moteur gauche tourner dans le sens horaire et on commande le moteur droit pour qu'il tourne dans le sens inverse de manière à faire tourner le robot sur lui même, on garde aussi l'émission sur le coté gauche jusqu'à ce qu'on ait plus de détection d'obstacle. A ce moment, on récupère le fonctionnement normal du robot en faisant tourner les deux moteurs dans le même sens et en émettant sur les deux cotés alternativement. Si le robot tourne plus de deux minutes sur lui-même en essayant de trouver une issue, il se met en mode veille. Ce fonctionnement est le même quand on détecte un obstacle sur le coté droit.

L'algorithme de base de fonctionnement du robot est représenté dans l'organigramme suivant :

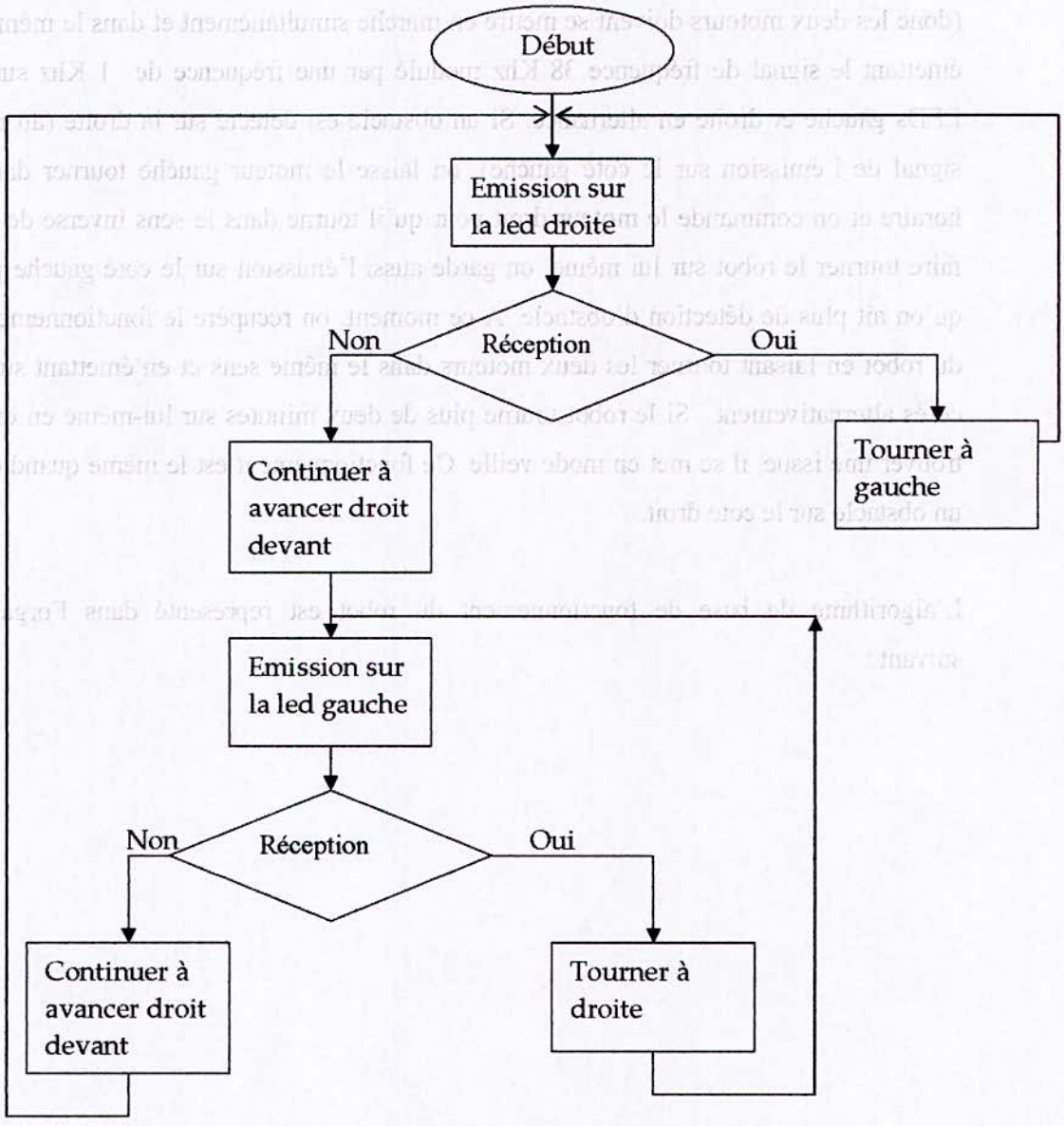


Fig. VI.6 : L'algorithme de détection des obstacles

Notons toute fois, que plusieurs autres stratégies peuvent être mise en œuvre permettant chacune au robot d'éviter les obstacles, mais le principe général reste le même.

VI-6- Ecriture du programme final

Après avoir adapté la carte, compilé le noyau temps réel et tracé la stratégie générale de mouvement du robot mobile, il faut assembler toutes ces pièces pour écrire le programme final.

Pour commencer, il nous faut un bootloader qui nous permet de copier l'image exécutable finale de la flash prom à la SRAM. externe. Ensuite une routine main () est executée pour appeler le système d'exploitation Xilkernel.

Une fois le noyau appelé et initialisé, il appelle son Ordonnanceur pour ordonnancer les différentes tâches du programme. On a configuré le noyau Xilkernel pour avoir un Ordonnanceur basé sur les priorités. Ainsi la tâche la plus prioritaire non bloquée sera exécutée. Tandis que les tâches de même priorité seront exécutées en round robin basé sur les *time slice* du processeur.

La première tâche qui sera exécuté dans notre programme est la tâche **robot_main_TASK** qui a la priorité la plus grande et qui sera donc exécutée dés qu'en charge notre projet. Le but de cette tâche est d'enregistrer les fonctions de prise en charge des interruptions (Handlers), valider les interruptions et initialiser le timer. Une fois l'exécution de cette tâche achevée, elle est quittée et ne sera plus exécutée une autre fois.

Les interruptions sont au nombre de trois :

- emission_led_int: le timer génère cette interruption à chaque 0.5 ms pour nous permettre de générer un signal carré d'une fréquence de 1Khz.qui servira de porteuse pour le signal de 38 KHz que les leds émettrons pour détecter la présence d'obstacles.
- reception_int : cette interruption surgit lorsqu'il y a réception sur l'un des deux cotés (droit ou gauche).
- arret_robot_int: lorsqu'on appuie sur le bouton poussoir 3, cette interruption permet d'arrêter le robot.

L'ordre de priorité de ces interruptions de la plus prioritaire à la moins prioritaire est le suivant : arret_robot_int, reception_int, puis emission_led_int.

Les deux tâches qui seront exécutées ensuite sont les tâches **moteur_droit_TASK** et **moteur_gauche_TASK** qui ont la même priorité. Ces deux taches permettent de tester la réception sur le coté correspondant et d'envoyer les signaux de commande correspondants.

Afin de s'assurer que les deux taches ne s'exécutent pas au même temps on réalise une communication entre les deux en utilisant deux queues.

A la première exécution c'est la tâche **moteur_droit_TASK** qui est exécutée la deuxième tâche est alors bloquée. Pendant 10 ms la led se trouvant sur le coté droit émet son signal de détection, au bout de ce temps on teste s'il y a eu réception, si c'est le cas le on envoie sur le moteur droit un signal de commande pour tourner dans le sens inverse puis on teste la réception jusqu'à ce qu'on s'assure que l'obstacle ne gène plus. A ce moment la tâche **moteur_droit_TASK** se bloque et envoie un message pour réveiller la tache **moteur_gauche_TASK** qui répétera exactement la même chose mais cette fois ci du coté gauche. Si jamais aucun signal de réception n'est pas détecté la tâche se bloque et cède la main à la deuxième tache.

En plus de ça un time out de 2 mn est prévu pour arrêter le moteur au cas ou celui-ci tourne autour de lui-même pendant deux minutes sans pouvoir trouver une issue.

Lorsque l'interruption arret_robot_int surgit (après appui sur le troisième bouton poussoir) la tâche la plus prioritaire (**arret_robot_TASK)** devient prête et s'exécute immédiatement. Cette tâche permet l'arrêt du robot s'il tourne autour de lui sans pouvoir trouver une issue.

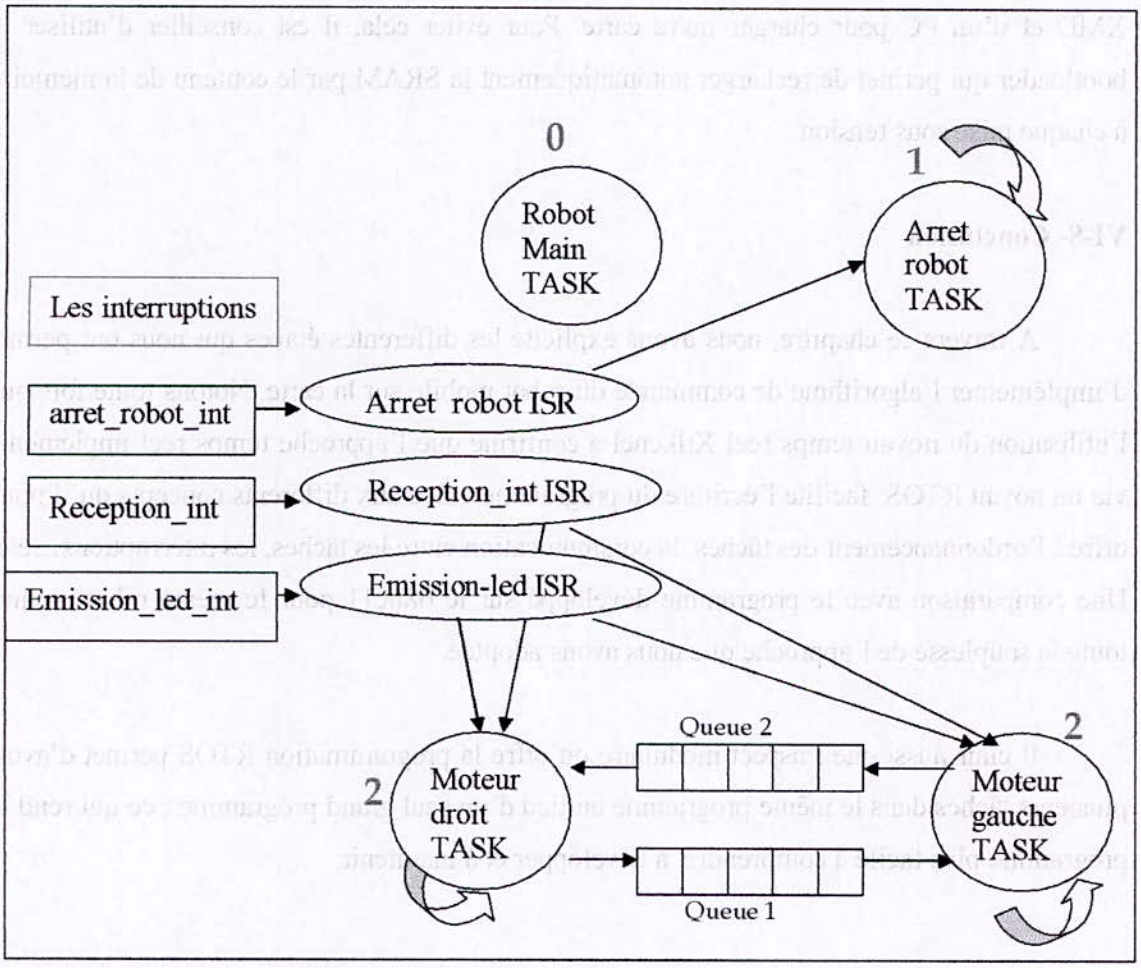


Fig. VI-7 : Structure générale du programme de commande du robot mobile

VI-7- Chargement de la conception sur la carte

Cette application contient trois projets software : **boot_loop SRAM_boot** et **RUN_FROM_SRAM.**

Le projet **boot_loop** est utilisé seulement comme une application boot_loop pour initialiser la BRAM du circuit FPGA. **RUN_FROM_SRAM** qui contient le programme qui sera exécuté pour faire marcher le robot, elle sera chargée sur la mémoire SRAM via la fenêtre XMD. Tandis que le projet **SRAM_boot** contient l'application bootloader pour charger le noyau Xilkenel sur la mémoire SRAM.

Pour charger le circuit FPGA avec notre projet, l'application boot_loop configure le circuit FPGA avec le processeur MicroBlaze, puis la fenêtre XMD est utilisée pour charger le fichier exécutable de l'application dans la SRAM. Cependant, une fois la carte est débranchée, le fichier *.elf* doit être rechargé sur la SRAM, ainsi on dépend toujours de l'outil

XMD et d'un PC pour charger notre carte. Pour éviter cela, il est conseiller d'utiliser le bootloader qui permet de recharger automatiquement la SRAM par le contenu de la mémoire à chaque mise sous tension.

VI-8- Conclusion

A travers ce chapitre, nous avons explicité les différentes étapes qui nous ont permis d'implémenter l'algorithme de commande du robot mobile sur la carte. Notons toute fois que l'utilisation du noyau temps réel Xilkenel a confirmé que l'approche temps réel implémenté via un noyau RTOS facilite l'écriture du programme grâce aux différents concepts qu'il nous offre : l'ordonnancement des tâches, la communication entre les tâches, les interruptions…etc. Une comparaison avec le programme développé sur le 68hc11 pour le même robot montre toute la souplesse de l'approche que nous avons adoptée.

Il clair aussi que l'aspect modulaire qu'offre la programmation RTOS permet d'avoir plusieurs tâches dans le même programme au lieu d'un seul grand programme ; ce qui rend le programme plus facile à comprendre, à développer et à maintenir.

Conclusion génerale

VII- Conclusion générale

Nous avons donc pu réaliser l'objectif de notre projet qui est l'utilisation d'un système d'exploitation temps réel pour commander le robot mobile ***kifploo***. Nous avons étudié tout d'abord les principales caractéristiques d'un robot mobile et en particulier le robot cible. Dans cette partie, on s'est beaucoup plus intéressé à la commande des moteurs et au système de perception utilisés dans le robot pour mieux comprendre son fonctionnement et pouvoir ensuite générer les signaux de commandes qui seront envoyés à partir de la carte de commande.

On a ensuite essayé de configurer le noyau uClinux et le charger sur la carte FPGA, mais on n'a pas pu achever cette approche pour des contraintes liées au matériel utilisé ainsi qu'au temps alloué à notre projet. On a alors choisit d'utiliser un autre système d'exploitation temps réel (Xilkernel) qui nous fournit toutes les caractéristiques nécessaires d'un système temps réel.

Pour pouvoir utiliser le noyau Xilkernel et programmer le robot ***kifploo***, on a créé un projet sous l'environnement EDK dont on a configuré le processeur MicroBlaze et les différents périphériques utilisés. On a ensuite écrit notre programme en langage C dans ce même environnement en exploitant les caractéristiques fournies par le système temps réel, à savoir l'utilisation d'un ordonnanceur en round robin pour gérer les différentes tâches du robot et l'utilisation d'interruptions. Ce qui nous a permis d'avoir un temps de réponse du robot de l'ordre de quelques microsecondes.

La programmation du robot ***kifploo*** n'est qu'une simple application servant à montrer l'intérêt d'utiliser un système temps réel embarqué et de choisir en plus un processeur virtuel pour le porter, afin de mener à mieux le fonctionnement d'un robot mobile. Il y apparaît important de souligner que l'application que nous avons implémentée n'est qu'une démonstration de la validité de l'approche que nous avons adoptée. Elle reste extensible dans le sens d'une application plus complète voire plus complexe

Ce projet pourra plutôt servir de base pour commander d'autres systèmes pour lesquels la réponse en temps réel est une exigence primordiale.

Bibliographie

La bibliographie :

[1] Y. Nguyen Minh, R. Audemard, « *Histoire de la robotique* », CNAM, 1997.

[2] : Encyclopédie Encarta 2004

[4] : quid 2000

[5] : Ziat Mehdi, « *Etude et réalisation d'un robot mobile* », Projet de fin d'études à l'ENP, Septembre 2004.

[6]: SGS-THOMSON Microelectronics, "*Application note of the L297 and the L298*", 2000.

[7] : Pierre Ficheux, « *LINUX embarqué* », édition EYROLLES 2002.

[8]: jean j.Labbros,"*MicroC/OS-II the real-time kernel*", CMP books Lawrence, kansas66046.

[9] : Dipnarayan Guha and Jun Kyun Choi, «*Embedded Real Time Linux Kernel Design using Game Theory and Control Logic*», Broadband Network Laboratory, Information and Communications University of KOREA ,2000.

[10] : Pruski, « Robot mobiles autonomes », Techniques de l'ingénieur, R7850, 1998.

[11] : Torstein Wroldsen Ståle Tveitane, *"Real Time Operating System for embedded platforms"*; thèse de magister en technologie de communication et d'information. Grimstad, Mai 2004.

[12] : N. Julien, « Circuits logiques programmables », maitrise EEA, *Université de bretagne sud lorient, septembre 1999.*

[13]: Peter Magnusson, "Evaluating Xilinx MicroBlaze for Network SoC solutions", *thèse de magister en the Computer Engineering* , 10th January 2004.

[14]: EDK OS and libraries Reference Manual, "*Embedded Development Kit 6.3i.*", UG 114(V3.0), 20 aout 2004.

[15]: "Microblaze processor reference guide", *Embedded Development 6. 3Kit.* UG081(V4.0), 24 aout 2004.

[16] : "Spartan 3 starter Kit Board reference guide"; UG130(V1.0.3), le 15 octobre 2004.

[17] : "Embedded System Tools Reference Manual"; *Embedded Development Kit 6.3i.* UG 111(V3.0), 20 aout 2004.

[18]: Platform Studio User Guide XPS, *Embedded Development Kit 6.3i.* UG 113(V3.0), 20 aout 2004.

La webographie :

http://perso.wanadoo.fr/michel.hubin/physique/microp/chap_mp4.htm

http://www.rennes.supelec.fr/ren/fi/elec

http://www.edanews.com/index.html

http://www.abcelectronique.com/acquier/Sources/L297.pdf

http://www.uclinux.org/index.html

http://www.itee.uq.edu.au/~jwilliams/

annexes

ANNEXE A : Les instructions de MicroBlaze

Table 1-4: **Instruction Set Nomenclature**

Symbol	Description
Ra	R0 - R31, General Purpose Register, source operand a
Rb	R0 - R31, General Purpose Register, source operand b
Rd	R0 - R31, General Purpose Register, destination operand
Imm	16 bit immediate value
Immx	*x* bit immediate value
FSLx	3 bit Fast Simplex Link (FSL) port designator where *x* is the port number
C	Carry flag, MSR[29]
Sa	Special Purpose Register, source operand
Sd	Special Purpose Register, destination operand
s(x)	Sign extend argument *x* to 32-bit value
*Addr	Memory contents at location Addr (data-size aligned)
&	Concatenate. E.g. "0000100 & Imm7" is the concatenation of the fixed field "0000100" and a 7 bit immediate value.

Table 1-5: **MicroBlaze Instruction Set Summary**

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
ADD Rd,Ra,Rb	000000	Rd	Ra	Rb	00000000000	Rd := Rb + Ra
RSUB Rd,Ra,Rb	000001	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + 1
ADDC Rd,Ra,Rb	000010	Rd	Ra	Rb	00000000000	Rd := Rb + Ra + C
RSUBC Rd,Ra,Rb	000011	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + C
ADDK Rd,Ra,Rb	000100	Rd	Ra	Rb	00000000000	Rd := Rb + Ra
RSUBK Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + 1
ADDKC Rd,Ra,Rb	000110	Rd	Ra	Rb	00000000000	Rd := Rb + Ra + C
RSUBKC Rd,Ra,Rb	000111	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + C
CMP Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000001	Rd := Rb + $\overline{Ra}$ + 1(signed)
CMPU Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000011	Rd := Rb + $\overline{Ra}$ + 1(unsigned)
ADDI Rd,Ra,Imm	001000	Rd	Ra	Imm		Rd := s(Imm) + Ra
RSUBI Rd,Ra,Imm	001001	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + 1
ADDIC Rd,Ra,Imm	001010	Rd	Ra	Imm		Rd := s(Imm) + Ra + C
RSUBIC Rd,Ra,Imm	001011	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + C
ADDIK Rd,Ra,Imm	001100	Rd	Ra	Imm		Rd := s(Imm) + Ra
RSUBIK Rd,Ra,Imm	001101	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + 1
ADDIKC Rd,Ra,Imm	001110	Rd	Ra	Imm		Rd := s(Imm) + Ra + C
RSUBIKC Rd,Ra,Imm	001111	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + C
MUL Rd,Ra,Rb	010000	Rd	Ra	Rb	00000000000	Rd := Ra * Rb
BSRL Rd,Ra,Rb	010001	Rd	Ra	Rb	00000000000	Rd : = Ra >> Rb
BSRA Rd,Ra,Rb	010001	Rd	Ra	Rb	01000000000	Rd := Ra[0], (Ra >> Rb)
BSLL Rd,Ra,Rb	010001	Rd	Ra	Rb	10000000000	Rd := Ra << Rb
MULI Rd,Ra,Imm	011000	Rd	Ra	Imm		Rd := Ra * s(Imm)
BSRLI Rd,Ra,Imm	011001	Rd	Ra	00000000000 & Imm5		Rd : = Ra >> Imm5
BSRAI Rd,Ra,Imm	011001	Rd	Ra	00000010000 & Imm5		Rd := Ra[0], (Ra >> Imm5)
BSLLI Rd,Ra,Imm	011001	Rd	Ra	00000100000 & Imm5		Rd := Ra << Imm5
IDIV Rd,Ra,Rb	010010	Rd	Ra	Rb	00000000000	Rd := Rb/Ra, signed
IDIVU Rd,Ra,Rb	010010	Rd	Ra	Rb	00000000001	Rd := Rb/Ra, unsigned
GET Rd,FSLx	011011	Rd	00000	0000000000000 & FSLx		Rd := FSLx (blocking data read) MSR[FSL] := FSLx_S_Control

Table 1-5: **MicroBlaze Instruction Set Summary** ***(Continued)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
BRLD Rd,Rb	100110	Rd	10100	Rb	00000000000	PC := PC + Rb; Rd := PC
BRA Rb	100110	00000	01000	Rb	00000000000	PC := Rb
BRAD Rb	100110	00000	11000	Rb	00000000000	PC := Rb
BRALD Rd,Rb	100110	Rd	11100	Rb	00000000000	PC := Rb; Rd := PC
BRK Rd,Rb	100110	Rd	01100	Rb	00000000000	PC := Rb; Rd := PC; MSR[BIP] := 1
BEQ Ra,Rb	100111	00000	Ra	Rb	00000000000	if Ra = 0; PC := PC + Rb
BNE Ra,Rb	100111	00001	Ra	Rb	00000000000	if Ra /= 0; PC := PC + Rb
BLT Ra,Rb	100111	00010	Ra	Rb	00000000000	if Ra < 0; PC := PC + Rb
BLE Ra,Rb	100111	00011	Ra	Rb	00000000000	if Ra <= 0; PC := PC + Rb
BGT Ra,Rb	100111	00100	Ra	Rb	00000000000	if Ra > 0; PC := PC + Rb
BGE Ra,Rb	100111	00101	Ra	Rb	00000000000	if Ra >= 0; PC := PC + Rb
BEQD Ra,Rb	100111	10000	Ra	Rb	00000000000	if Ra = 0; PC := PC + Rb
BNED Ra,Rb	100111	10001	Ra	Rb	00000000000	if Ra /= 0; PC := PC + Rb
BLTD Ra,Rb	100111	10010	Ra	Rb	00000000000	if Ra < 0; PC := PC + Rb
BLED Ra,Rb	100111	10011	Ra	Rb	00000000000	if Ra <= 0; PC := PC + Rb
BGTD Ra,Rb	100111	10100	Ra	Rb	00000000000	if Ra > 0; PC := PC + Rb
BGED Ra,Rb	100111	10101	Ra	Rb	00000000000	if Ra >= 0; PC := PC + Rb
ORI Rd,Ra,Imm	101000	Rd	Ra	Imm		Rd := Ra or s(Imm)
ANDI Rd,Ra,Imm	101001	Rd	Ra	Imm		Rd := Ra and s(Imm)
XORI Rd,Ra,Imm	101010	Rd	Ra	Imm		Rd := Ra xor s(Imm)
ANDNI Rd,Ra,Imm	101011	Rd	Ra	Imm		Rd := Ra and $\overline{\text{s(Imm)}}$
IMM Imm	101100	00000	00000	Imm		Imm[0:15] := Imm
RTSD Ra,Imm	101101	10000	Ra	Imm		PC := Ra + s(Imm)
RTID Ra,Imm	101101	10001	Ra	Imm		PC := Ra + s(Imm); MSR[IE] := 1
RTED Ra,Imm	101101	10010	Ra	Imm		PC := Ra + s(Imm); MSR[EE] := 1, MSR[EIP]:=0
RTBD Ra,Imm	101101	10010	Ra	Imm		PC := Ra + s(Imm); MSR[BIP] := 0
BRI Imm	101110	00000	00000	Imm		PC := PC + s(Imm)
BRID Imm	101110	00000	10000	Imm		PC := PC + s(Imm)
BRLID Rd,Imm	101110	Rd	10100	Imm		PC := PC + s(Imm); Rd := PC
BRAI Imm	101110	00000	01000	Imm		PC := s(Imm)
BRAID Imm	101110	00000	11000	Imm		PC := s(Imm)
BRALID Rd,Imm	101110	Rd	11100	Imm		PC := s(Imm); Rd := PC

Table 1-5: **MicroBlaze Instruction Set Summary** *(Continued)*

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
PUT Ra,FSLx	011011	00000	Ra	1000000000000 & FSLx		FSLx := Ra (blocking data write)
NGET Rd,FSLx	011011	Rd	00000	0100000000000 & FSLx		Rd := FSLx (non-blocking data read) MSR[FSL] := FSLx_S_Control MSR[C] := not FSLx_S_Exists
NPUT Ra,FSLx	011011	00000	Ra	1100000000000 & FSLx		FSLx := Ra (non-blocking data write) MSR[C] := FSLx_M_Full
CGET Rd,FSLx	011011	Rd	00000	0010000000000 & FSLx		Rd := FSLx (blocking control read) MSR[FSL] := not FSLx_S_Control
CPUT Ra,FSLx	011011	00000	Ra	1010000000000 & FSLx		FSLx := Ra (blocking control write)
NCGET Rd,FSLx	011011	Rd	00000	0110000000000 & FSLx		Rd := FSLx (non-blocking control read) MSR[FSL] := not FSLx_S_Control MSR[C] := not FSLx_S_Exists
NCPUT Ra,FSLx	011011	00000	Ra	1110000000000 & FSLx		FSLx := Ra (non-blocking control write) MSR[C] := FSLx_M_Full
OR Rd,Ra,Rb	100000	Rd	Ra	Rb	00000000000	Rd := Ra or Rb
AND Rd,Ra,Rb	100001	Rd	Ra	Rb	00000000000	Rd := Ra and Rb
XOR Rd,Ra,Rb	100010	Rd	Ra	Rb	00000000000	Rd := Ra xor Rb
ANDN Rd,Ra,Rb	100011	Rd	Ra	Rb	00000000000	Rd := Ra and $\overline{Rb}$
SRA Rd,Ra	100100	Rd	Ra	0000000000000001		Rd := Ra[0], (Ra >> 1); C := Ra[31]
SRC Rd,Ra	100100	Rd	Ra	0000000000100001		Rd := C, (Ra >> 1); C := Ra[31]
SRL Rd,Ra	100100	Rd	Ra	0000000001000001		Rd := 0, (Ra >> 1); C := Ra[31]
SEXT8 Rd,Ra	100100	Rd	Ra	0000000001100000		Rd[0:23] := Ra[24]; Rd[24:31] := Ra[24:31]
SEXT16 Rd,Ra	100100	Rd	Ra	0000000001100001		Rd[0:15] := Ra[16]; Rd[16:31] := Ra[16:31]
WIC Ra,Rb	100100	Ra	Ra	Rb	01101000	ICache_Tag := Ra, ICache_Data := Rb
WDC Ra,Rb	100100	Ra	Ra	Rb	01100100	DCache_Tag := Ra, DCache_Data := Rb
MTS Sd,Ra	100101	00000	Ra	1100000000000 & Sd		Sd := Ra , where Sd=001 is MSR
MFS Rd,Sa	100101	Rd	00000	1000000000000 & Sa		Rd := Sa , where Sa=000 is PC, 001 is MSR, 011 is EAR, and 101 is ESR
MSRCLR Rd,Imm	100101	Rd	00001	00 & Imm14		Rd := MSR; MSR := MSR ^ Imm14
MSRSET Rd,Imm	100101	Rd	00000	00 & Imm14		Rd := MSR; MSR := MSR ^ Imm14
BR Rb	100110	00000	00000	Rb	00000000000	PC := PC + Rb
BRD Rb	100110	00000	10000	Rb	00000000000	PC := PC + Rb

Table 1-5: **MicroBlaze Instruction Set Summary** ***(Continued)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
BRKI Rd,Imm	101110	Rd	01100	Imm		PC := s(Imm); Rd := PC; MSR[BIP] := 1
BEQI Ra,Imm	101111	00000	Ra	Imm		if Ra = 0: PC := PC + s(Imm)
BNEI Ra,Imm	101111	00001	Ra	Imm		if Ra /= 0: PC := PC + s(Imm)
BLTI Ra,Imm	101111	00010	Ra	Imm		if Ra < 0: PC := PC + s(Imm)
BLEI Ra,Imm	101111	00011	Ra	Imm		if Ra <= 0: PC := PC + s(Imm)
BGTI Ra,Imm	101111	00100	Ra	Imm		if Ra > 0: PC := PC + s(Imm)
BGEI Ra,Imm	101111	00101	Ra	Imm		if Ra >= 0: PC := PC + s(Imm)
BEQID Ra,Imm	101111	10000	Ra	Imm		if Ra = 0: PC := PC + s(Imm)
BNEID Ra,Imm	101111	10001	Ra	Imm		if Ra /= 0: PC := PC + s(Imm)
BLTID Ra,Imm	101111	10010	Ra	Imm		if Ra < 0: PC := PC + s(Imm)
BLEID Ra,Imm	101111	10011	Ra	Imm		if Ra <= 0: PC := PC + s(Imm)
BGTID Ra,Imm	101111	10100	Ra	Imm		if Ra > 0: PC := PC + s(Imm)
BGEID Ra,Imm	101111	10101	Ra	Imm		if Ra >= 0: PC := PC + s(Imm)
LBU Rd,Ra,Rb	110000	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd[0:23] := 0, Rd[24:31] := *Addr
LHU Rd,Ra,Rb	110001	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd[0:15] := 0, Rd[16:31] := *Addr
LW Rd,Ra,Rb	110010	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd := *Addr
SB Rd,Ra,Rb	110100	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd[24:31]
SH Rd,Ra,Rb	110101	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd[16:31]
SW Rd,Ra,Rb	110110	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd
LBUI Rd,Ra,Imm	111000	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd[0:23] := 0, Rd[24:31] := *Addr
LHUI Rd,Ra,Imm	111001	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd[0:15] := 0, Rd[16:31] := *Addr
LWI Rd,Ra,Imm	111010	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd := *Addr

Table 1-5: **MicroBlaze Instruction Set Summary *(Continued)***

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
SBI Rd,Ra,Imm	111100	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd[24:31]
SHI Rd,Ra,Imm	111101	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd[16:31]
SWI Rd,Ra,Imm	111110	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd

Annexe B

Table 5-1: **XilKernel Function Summary**

int pthread_create (pthread_t thread, pthread_attr_t* attr, void* (*start_func)(void*), void* param)
void pthread_exit (void *value_ptr)
int pthread_join (pthread_t thread, void **value_ptr)
pthread_t pthread_self (void)
int pthread_detach (pthread_t target)
int pthread_equal (pthread_t t1, pthread_t t2)
int pthread_getschedparam (pthread_t thread, int *policy, struct sched_param *param).
int pthread_setschedparam (pthread_t thread, int policy, const struct sched_param *param).
int pthread_attr_init (pthread_attr_t* attr)
int pthread_attr_destroy (pthread_attr_t* attr)
int pthread_attr_setdetachstate (pthread_attr_t* attr, int dstate)
int pthread_attr_getdetachstate (pthread_attr_t* attr, int *dstate)
int pthread_attr_setschedparam (pthread_attr_t* attr, struct sched_param *schedpar)
int pthread_attr_getschedparam (pthread_attr_t* attr, struct sched_param* schedpar).
int pthread_attr_setstack (const pthread_attr_t *attr, void *stackaddr, size_t stacksize).
int pthread_attr_getstack (const pthread_attr_t *attr, void **stackaddr, size_t *stacksize).
pid_t get_currentPID (void)
int kill (pid_t pid)
int process_status (pid_t pid, p_stat *ps)
int yield (void)
int sem_init (sem_t *sem, int pshared, unsigned value)
int sem_destroy (sem_t* sem)
int sem_getvalue (sem_t* sem, int* value)
int sem_wait (sem_t* sem)
int sem_trywait (sem_t* sem)

Table 5-1: **Xilkernel Function Summary *(Continued)***

int sem_timedwait (sem_t* sem, unsigned ms)
int sem_post (sem_t* sem)
sem_t* sem_open (const char* name, int oflag, ...)
int sem_close (sem_t* sem)
int sem_unlink (const char* name)
int msgget(key_t key, int msgflg)
int msgctl (int msqid, int cmd, struct msqid_ds* buf)
int msgsnd (int msqid, const void *msgp, size_t nbytes, int msgflg)
ssize_t msgrcv (int msqid, void *msgp, size_t nbytes, long msgtyp, int msgflg)
int shmget (key_t key, size_t size, int shmflg)
int shmctl (int shmid, int cmd, struct shmid_ds *buf)
void* shmat (int shmid, const void *shmaddr, int flag)
int shm_dt (void *shmaddr)
int pthread_mutex_init pthread_mutex_t* mutex, const pthread_mutexattr_t* attr)
int pthread_mutex_destroy (pthread_mutex_t* mutex)
int pthread_mutex_lock (pthread_mutex_t* mutex)
int pthread_mutex_trylock (pthread_mutex_t* mutex)
int pthread_mutex_unlock (pthread_mutex_t* mutex)
int pthread_mutexattr_init (pthread_mutexattr_t* attr)
int pthread_mutexattr_destroy (pthread_mutexattr_t* attr)
int pthread_mutexattr_settype (pthread_mutexattr_t* attr, int type)
int pthread_mutexattr_gettype (pthread_mutexattr_t* attr, int * type)
int bufcreate (membuf_t *mbuf, void *memptr, int nblks, size_t blksiz)
int bufdestroy (membuf_t mbuf).
void* bufmalloc (membuf_t mbuf, size_t siz)
void buffree (membuf_t mbuf, void* mem).
unsigned int xget_clock_ticks ().
time_t time (time_t *timer)
unsigned sleep (unsigned int ms)
unsigned int register_int_handler (int_id_t id, void (*handler)(void*), void *callback)
void unregister_int_handler (int_id_t id)
void enable_interrupt (int_id_t id)

Table 5-1: **Xilkernel Function Summary *(Continued)***

void `disable_interrupt` (int_id_t `id`)
void `acknowledge_interrupt` (int_id_t `id`)
int `elf_process_create` (void* `start_addr`, int prio)
int `elf_process_exit` (void)