MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE FERHAT ABBAS – SETIF
UFAS (ALGERIE)

MEMOIRE

Présentée à la Faculté des Sciences de l'ingénieur
Département d'électronique
Pour l'Obtention du Diplôme de

MAGISTER

Option : Instrumentation

Par
MR : MEZZAH Ibrahim

Thème

ETUDE ET CONCEPTION D'UN MICROCONTROLEUR IP SUR FPGA

Soutenu le : 02 décembre 2008 devant la commission d'examen :

Mr F. DJAHLI	Prof à l'université de Sétif	**Président**
Mr A. KHELLAF	Prof à l'université de Sétif	**Rapporteur**
Mr N. AMARDJIA	MC à l'université de Sétif	**Examinateur**
Mr N. BOUROUBA	MC à l'université de Sétif	**Examinateur**
Mr H. CHEMALI	MC à l'université de Sétif	**Invité**

REMERCIMENTS

Mes remerciements les plus sincères vont tout d'abord au Professeur A. Khellaf qui a encadré ce travail et au Docteur H. Chemali qui a consacré beaucoup du temps et d'efforts pour m'apprendre le métier de conception des systèmes électroniques.

Je tiens aussi à leur exprimer mes sentiments de gratitude pour le soutien qu'ils m'ont apporté et la bienveillance qu'ils n'ont cessé de manifester à mon égard.

Je tiens à remercier F. Djahli Professeur au département d'électronique de Sétif pour avoir accepté de présider le jury.

Mes remerciements sont aussi adressés à N. Amardjia et N. Bourouba, maitres de conférences au département d'électronique de Sétif, qui ont accepté d'examiner ce travail.

Je remercie beaucoup ma sœur Farida pour l'aide qu'elle m'a apportée dans ce travail et durant toute ma formation, je la remercie aussi pour son encouragement.

Je remercie particulièrement messieurs Koussa Salah et Rezki pour m'avoir aidé à acquérir la carte de développement.

Je remercie tous ceux qui m'ont aidé, je cite spécialement mes amis Tarek et Bilel et mon frère Walid.

RESUME

Le but de ce travail est la conception d'un microcontrôleur sous forme d'un IP logiciel « MCIP » en exploitant VHDL sur une plateforme de développement de *Xilinx ISE9.1* et un simulateur *ModelSim* de *Mentor Graphics*.

La méthode suivie repose essentiellement sur la subdivision du microcontrôleur en modules génériques facilement reconfigurables. L'architecture RISC employée permet d'optimiser les ressources et d'atteindre des fréquences de travail élevées.

Notre contrôleur doté d'un jeu de 72 instructions, réparties en 5 catégories, offre une souplesse de programmation et un traitement rapide des opérations courantes des microcontrôleurs du marché.

L'implantation du MCIP sur FPGA de la famille *Spartan3E* de *Xilinx* a permis d'atteindre des fréquences de travail dépassant 50 MHz.

La validation fonctionnelle du MCIP est passée par une phase très exhaustive de vérification et par une technique d'injection de fautes au niveau comportemental.

Afin d'assurer l'exactitude et les séquencements adéquats suivis lors de l'exécution des différentes instructions du répertoire du MCIP, nous avons développé plusieurs routines et programmes dont les données produites sont facilement vérifiables. Comme la gestion du temps est primordiale et fait intervenir les ressources internes, nous avons développé une application de gestion des horaires de prière qui a elle-même contribué d'une manière significative à la maitrise des outils de développement et à la valorisation du MCIP conçu.

Le développement du MCIP sous forme IP représente la première pierre de l'édifice de construction d'un système électronique moderne.

Mots clés : IP, VHDL, FPGA, Microcontroller.

Glossaire

ALU	*Arithmetic & Logic Unit*
APTD	*Automatic Prayer Time Display*
ASIC	*Application Specific Integrated Circuit*
CAO	*Conception Assistée par Ordinateur*
CI	*Circuit Intégré*
CLB	*Configurable Logic Block*
CPLD	*Complex Programmable Logic Device*
CPU	*Central Processing Unit*
DCM	*Digital Clock Manager*
DSP	*Digital Signal Processor*
FBGA	*Fine Pitch Ball Grid Array*
FPGA	*Field Programmable Gate Array*
GPR	*General Purpose Registre*
GPS	*Global Positioning System*
HDL	*Hardware Design Language*
IOB	*Input Output Block*
IP	*Intellectual Property*
JTAG	*Joint Test Action Group*
LCA	*Logic Cell Array*
LFSR	*Linear Feedback Shift Register*
LUT	*Look Up Table*
MCU	*Microcontrôleur Unit*
NoC	*Network on Chip*
PC	*Program counter*
PLL	*Phase Locked Loop*
RISC	*Reduced instruction set computer*
RTL	*Register Transfer Level*
SoC	*System on Chip*
SoPC	*System on Programmable Chip*
SFR	*Special Function Registre*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*
VITAL	*VHDL Initiative Towards ASIC Libraries*

Sommaire

Introduction

L’alliance, ces dernières années, des microcontrôleurs sous forme d’IP cores (*Intellectual Property*) et des FPGA (*Field Programmable Gate Array*) a donné naissance à des plateformes de développement d’une très grande flexibilité en termes de coût et d’exploitation [1].

Au sein de tout système électronique moderne, de la télécommande d’un téléviseur au GPS (*Global Positioning System*), il y a au moins un microcontrôleur (MCU). Dans la majorité des applications courantes, plusieurs MCU sont utilisés en fin de chaine d’exploitation et sont en conséquence dédiés à réaliser des tâches spécifiques en comparaison aux traitements de tâches générales effectuées par les microprocesseurs.

Les microcontrôleurs sont généralement conçus pour réduire le prix, équiper les modules industriels automatiques et en mode embarqué sur FPGA, peuvent être reprogrammés pour changer divers fonctions. Cette dernière action permet une flexibilité d’utilisation sur divers produits même en présence d’interfaces variées.

Plus petits, plus légers, plus de fonctionnalités, plus performants, plus rapides, moins de consommation, telles sont les caractéristiques souhaitables des systèmes électroniques d'aujourd'hui. C'est dû principalement à la capacité d'intégration de plus en plus élevée que l'incrustation d'un système complet sur une même puce est rendue possible. Ces systèmes sur une puce, appelés SoC (*System on Chip*), se développent très rapidement et trouvent place dans la majorité des applications grand public, scientifique, militaire, spatiale et particulièrement dans les systèmes embarqués [1].

Cependant, dans le nouveau contexte de développement où les critères prédominants sont la rapidité de développement et la maîtrise des coûts, d'autres approches de conception se sont imposées aux méthodes de conception traditionnelles pour mieux gérer la complexité inhérente aux SoC. En effet, la réutilisation, la reconfiguration, la programmation à haut niveau et l'abstraction sont les nouveaux leviers de la conception moderne comprenant :

- De puissants circuits programmables (CPU, DSP, MCU) et configurables (FPGA, CPLD) sont de plus en plus proposés [1].
- Des outils de synthèse associés à des langages de conception et de simulation puissants, tels que les langages VHDL et Verilog sont développés [1].
- De nombreuses fonctions complexes (cryptage, codage, compression …etc) sont désormais disponibles sous forme de modules réutilisables pré validés, appelés IP [2].

Notre travail s'articule essentiellement sur la conception d'un projet d'utilité scientifique et industrielle qui utilise les nouvelles plateformes de développement. Il s'agit de concevoir un microcontrôleur sous forme d'un IP soft dénommé MCIP. Ce microcontrôleur modélisable en VHDL doit avoir une architecture structurée, riche et configurable. Les fonctionnalités du MCIP seront validées sous une plateforme VHDL de *Xilinx*. Le MCIP devrait être implanté sur un FPGA et piloter une application réelle. Les caractéristiques du MCIP doivent au minimum se rapprocher des microcontrôleurs du marché.

La présentation de ce mémoire est organisée en 4 chapitres. Nous présentons dans le premier chapitre les spécificités des SoC et IP cores, les techniques de réalisation des systèmes sur puces en considérant les aspects matériels (composants reprogrammables, ASIC) et les aspects logiciels (HDL, synthèse logique, placement & routage) [1,20].

L'architecture du microcontrôleur développé et ses principales caractéristiques sont exposées dans le deuxième chapitre.

Nous détaillons dans le troisième chapitre la conception du MCIP, les résultats de simulation et de synthèse, une technique d'injection de fautes a l'échelle comportementale et nous présentons aussi une implantation du circuit sur un FPGA.

Le chapitre 4 est consacré à la présentation d'une application développée autour du MCIP.

Une conclusion expose les résultats obtenus, situe l'intérêt du travail accompli et donne des perspectives quant à la suite du MCIP.

Chapitre 1

Techniques de conception

4.1. INTRODUCTION

Les nouvelles conceptions des circuits électroniques modernes, répondant aux exigences du marché, ont imposé et introduit de nouvelles techniques d'intégration. En effet, suite aux progrès technologiques continuels, l'intégration a connu des évolutions multiples du SoC aux NoC (*Network on Chip*) …etc. Les dernières plateformes de développement ont réussi à résoudre la majorité des difficultés inhérentes à la complexité croissante accompagnant les circuits mis en œuvre mais le prix de développement reste très élevé. Dans la gamme moyenne des CI, les progrès sont significatifs et les outils de conception sont à la portée des petites entreprises et organismes de recherche. Pour des raisons de coût et d'évolutivité de la technologie, les systèmes de développement ont imposé les FPGA et circuits compatibles comme cellules de base de validation et d'expérimentation [1].

Cette nouvelle tendance, a fait émerger une nouvelle manière de renforcer la réutilisabité des circuits déjà développés et elle est le succès derrière les progrès et la rapide mise au marché de produits dans ce domaine. Désormais, les concepteurs fournissant des blocs appelés IP, sont organisés en grandes firmes et les produits (hard et soft) sont répertoriés et proposés aux constructeurs afin qu'ils choisissent les modules adaptés à leurs besoins pour construire des circuits électroniques complexes. Ce nouveau métier, consiste donc à se spécialiser dans le développement de modules électroniques plus ou moins complexes sans se préoccuper de l'implantation physique, ce qui améliore l'efficacité et l'organisation de tout le processus de la production électronique.

4.2. PRINCIPE DE CONCEPTION

La méthodologie actuelle de conception consiste à partitionner, après spécification, le système en modules hardware et software. Le software est typiquement implémenté en langage C/C++ et le hardware en langage de description hardware (VHDL, Verilog, SystemC …etc) [2, 3]. La co-simulation du système entier est réalisée par la suite [4] (*figure 1.1*).

La conception d'un système électronique est définie comme étant :

- La construction d'une architecture matérielle composée de blocs logiques standards (processeurs, mémoires …etc), de blocs logiques spécifiques et de bus de communications.

- La production et développement des ressources logicielles adaptées.

L'approche système sur silicium (*System on Chip* SoC) est la cohabitation de ces ressources sur une même puce.

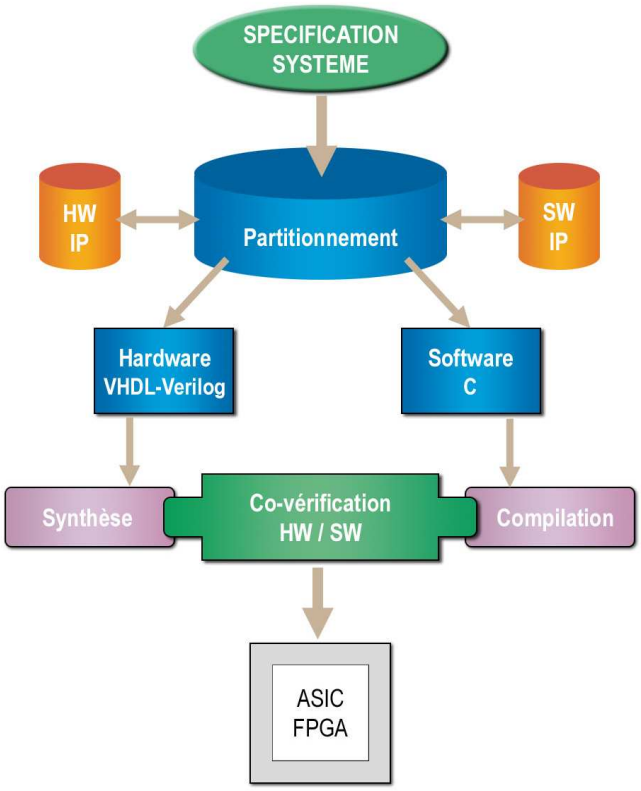


Figure 1.1: Flot de conception de systèmes sur puces

1.2.1 SOC et IP

Un SoC est un circuit intégré constitué de plusieurs millions de transistors, et même si les outils de CAO sont performants, les concepteurs ne peuvent plus se permettre de concevoir un système complet sans utiliser des briques de base. On se dirige alors vers une méthodologie de conception basée sur la réutilisation de matériel. Un SoC est constitué généralement d'un bus système et de différents éléments tels que processeur, DSP, RAM, ROM ...etc. Ces éléments se trouvent sous la forme de cœurs réutilisables, aussi appelés "IP cores". Trois grandes classes caractérisent les IP [5, 3, 6] :

1. IP HARD : C'est un bloc physiquement implanté, très optimisé en performances (puissance, taille ...etc) pour une technologie spécifique (cet effet limite sa portabilité). Il a l'avantage d'être prédictif au niveau des performances finales et la propriété intellectuelle est fortement préservée.
2. IP SOFT : Le coeur est livré sous sa forme HDL (*Hardware Design Language*) synthétisable. Le principal avantage de cette classe d'IP est que grâce à cette

description de haut niveau, la flexibilité et la portabilité du cœur sont assurées. Cependant, la protection de la propriété intellectuelle n'est pas assurée puisque le fournisseur d'IP livre son composant virtuel avec le code RTL (*Register Transfer Level*).

3. IP FIRM : Un IP de ce type est conçu de façon à optimiser sa surface et ses performances à travers un placement des modules qui le composent. Le coeur de type FIRM inclut une combinaison du RTL synthétisable, les références de la bibliothèque technologique cible, le détail de connectivité *netlist* complet ou partiel de portes. La protection de la propriété intellectuelle peut ne pas être assurée.

1.2.2 Technologies cibles

Il existe plusieurs technologies pour l'implantation des circuits intégrés développés en HDL. On note que bien qu'il soit vrai que le but d'un HDL et en particulier de la synthèse est de devenir indépendant de la technologie et des contraintes qui lui y sont liées, il est également envisageable d'orienter un peu la manière dont est conçu et décrit le projet afin qu'il s'adapte aux capacités de la cible [7].

Les deux principales technologies reposent sur les ASIC et FPGA [7, 8] :

- ASIC (*Application Specific Integrated Circuit*) : c'est un circuit intégré spécifique à l'application qui est fabriqué spécialement et sur mesure pour une technologie donnée;
- FPGA (*Field Programmable Gate Array*) : circuit intégré programmable totalement fabriqué mais configurable grâce à une programmation des éléments constitutifs.

L'approche **SoC** (*System on Chip*) (technologie ASIC) répond aux besoins de performance et d'intégration mais :

- ♦ Elle est peu adaptée à l'évolutivité des systèmes
- ♦ Elle reste réservée aux grands volumes de production
- ♦ La fabrication et le test sont des étapes longues et coûteuses.

L'approche **SoPC** (*System on Programmable Chip*) (technologie FPGA) résout ces problèmes (développement et prototypage rapides, possibilité de reconfiguration en quelques millisecondes et à volonté) mais :

- ♦ La densité d'intégration est moindre
- ♦ La consommation est plus grande
- ♦ Les performances sont moindres

4.3. LES CIRCUITS PROGRAMMABLES FPGA

Les FPGA sont des circuits intégrés programmables totalement fonctionnels : on programme les fonctions logiques, les flips flops, les reset, l'interconnexion des modules, les entrées/sorties [7, 9, 10]. Les FPGA sont des circuits très complexes qui peuvent avoisiner plusieurs millions de transistors avec les technologies actuelles [11]. Selon la manière de programmation, on distingue principalement trois types de FPGA :

- FPGA programmés par RAM, tels que fabriqués par *Xilinx* et *Altera* [12, 13]
- FPGA programmés par mémoire non volatile reprogrammable, par exemple EEPROM. (fabriqués par *Lattice*) [14]
- FPGA programmés par fusible (ou anti-fusible), produits typiquement par *Actel* [15].

1.3.1 L'architecture des circuits FPGA

Bien qu'il existe actuellement plusieurs fabricants de circuits FPGA, *Xilinx* et *Altera* demeurent toujours les plus connus. Plusieurs technologies et principes organisationnels seront illustrés en usant des circuits *Xilinx* que nous avons employés [12].

Un FPGA est constitué entre autres, de CLB (*Configurable Logic Blocks*), d' IOB (*Input Output Blocks*) et d'un réseau dense de canaux de routage qui parcourt tout le circuit (*figure 1.2*). De nombreuses fonctions liées à la programmation, à la gestion de l'horloge, de reset, du démarrage sont également incluses dans le circuit. Les FPGA complexes offrent aussi des blocs fonctionnels tels que RAM, multiplieur, DSP, CPU …etc.

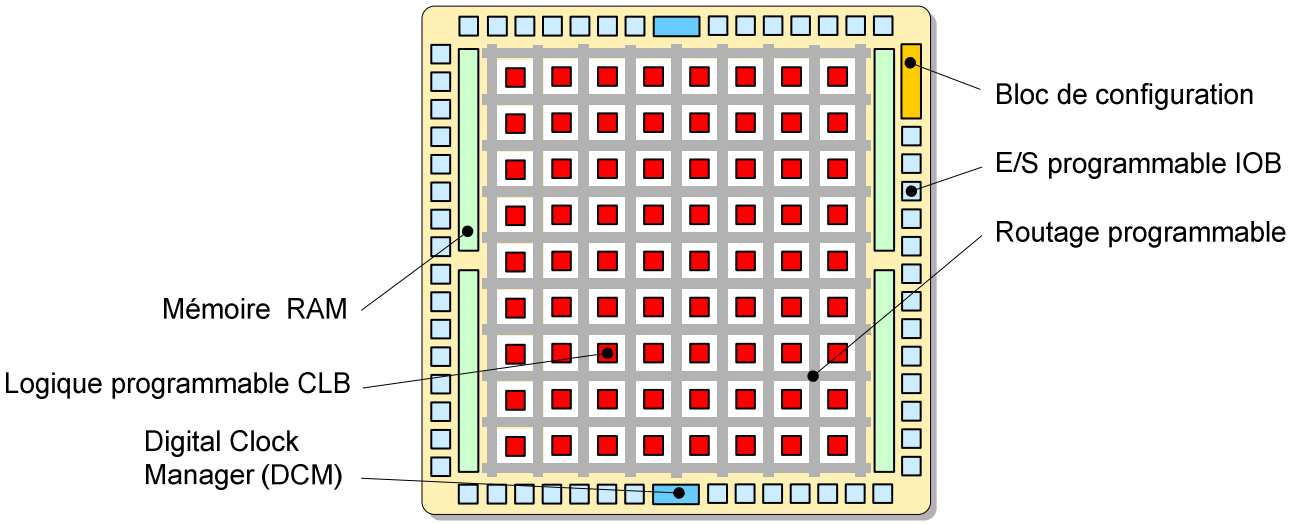


Figure 1.2: Architecture générale d'un FPGA

En général, l'architecture retenue par *Xilinx* se présente sous forme de deux couches (*figure 1.3*) [8, 12] :

- une couche configurable ;
- une couche réseau mémoire SRAM ou LCA (*Logic Cell Array*).

La couche configurable est constituée d'une matrice de blocs logiques configurables CLB permettant de réaliser des fonctions combinatoires et 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.

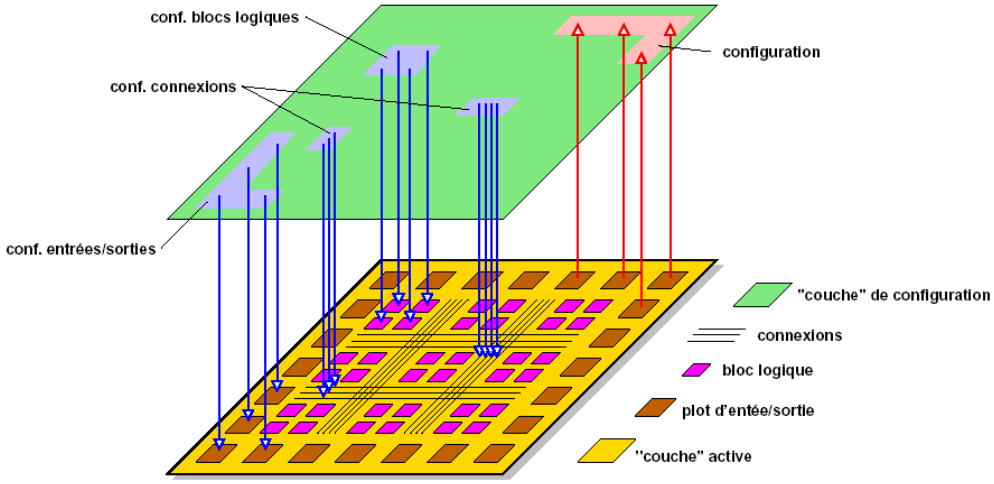


Figure 1.3: Structure en "couches" d'un FPGA

La programmation du circuit consistera par le biais de l'application d'un potentiel adéquat sur la grille de certains transistors à interconnecter les éléments des CLB et des IOB afin de réaliser les fonctions souhaitées et d'assurer la propagation des signaux. Ces potentiels sont tout simplement mémorisés dans le réseau mémoire SRAM (*Figure 1.4*).

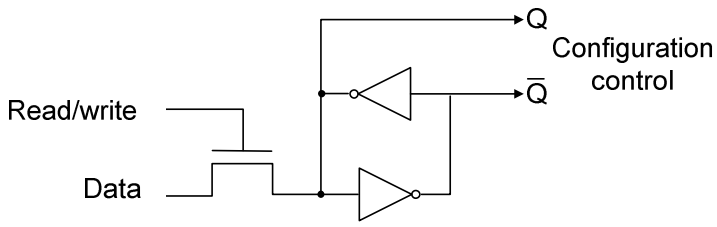


Figure 1.4: Cellule SRAM

La configuration du circuit est mémorisée sur la couche réseau SRAM et stockée aussi 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 perçoit aisément qu'un même circuit peut être exploité successivement avec des ROM différentes puisque sa programmation interne n'est jamais définitive. On voit tout le pari que l'on peut tirer de cette souplesse en particulier lors d'une phase de mise au point [16].

1.3.2 Les éléments d'un FPGA

1.3.2.1 Les CLB (*Configurable Logic Bloc*)

Les blocs logiques configurables sont les éléments déterminants des performances du FPGA. La *figure 1.5* nous montre le schéma d'un CLB de la famille XC4000 de *Xilinx* [17]. Chaque bloc est composé d'un bloc de logique combinatoire basé sur des LUT (*Look Up Table*) et d'un bloc de mémorisation.

1.3.2.2 Les blocs d' E/S IOB (*Input Output Bloc*)

Les blocs d'entrée/sortie servent d'interface entre les broches et le cœur du FPGA. Chaque bloc IOB contrôle une broche du composant et peut être défini en entrée, en sortie, bidirectionnel et en mode haute impédance.

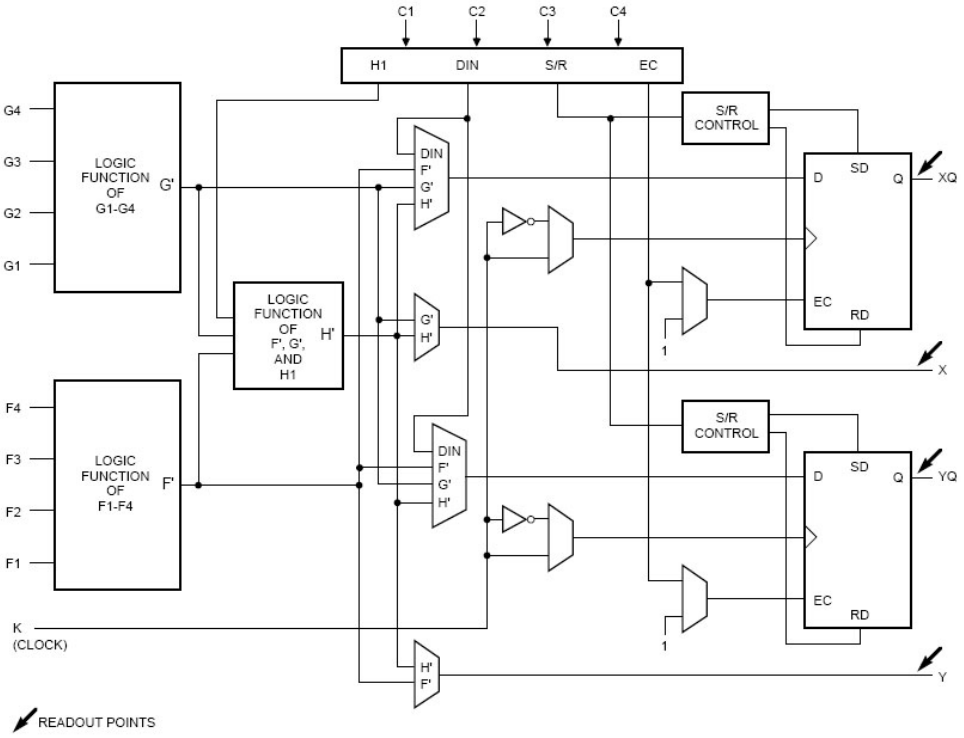


Figure 1.5: Bloc de logique programmable (CLB du XC4000)

1.3.2.3 Les différents modes d'interconnexions

Les connexions internes dans les circuits FPGA sont réalisées avec de segments métallisés et des matrices programmables réparties sur la totalité du circuit, horizontalement et verticalement entre les divers CLB (*figure 1.6*). Les connexions entre les diverses lignes sont assurées par des transistors MOS dont l'état est contrôlé par des cellules SRAM. Le rôle de ces interconnexions est de relier avec un maximum d'efficacité les blocs logiques et les entrées/sorties afin que le taux d'utilisation dans un circuit donné soit le plus élevé possible. Pour y parvenir à cet objectif, *Xilinx* propose trois modes d'interconnexions selon la longueur et la destination des liaisons. Nous disposons :

- d'interconnexions à usage général, positionnées entre les rangées et les colonnes des CLB et des IOB. Elles sont utilisées pour relier un CLB à n'importe quel autre.
- d'interconnexions directes permettant l'établissement de liaisons entre les CLB et les IOB avec un maximum d'efficacité en termes de vitesse et d'occupation du circuit. De plus, il est possible de connecter directement certaines entrées d'un CLB aux sorties d'un autre adjacent.
- de longues lignes qui sont de longs segments métallisés parcourant toute la longueur et la largeur du composant, elles permettent éventuellement de transmettre avec un minimum de retard les signaux entre les différents éléments dans le but d'assurer un synchronisme aussi parfait que possible. De plus, ces longues lignes permettent d'éviter la multiplicité des points d'interconnexion.

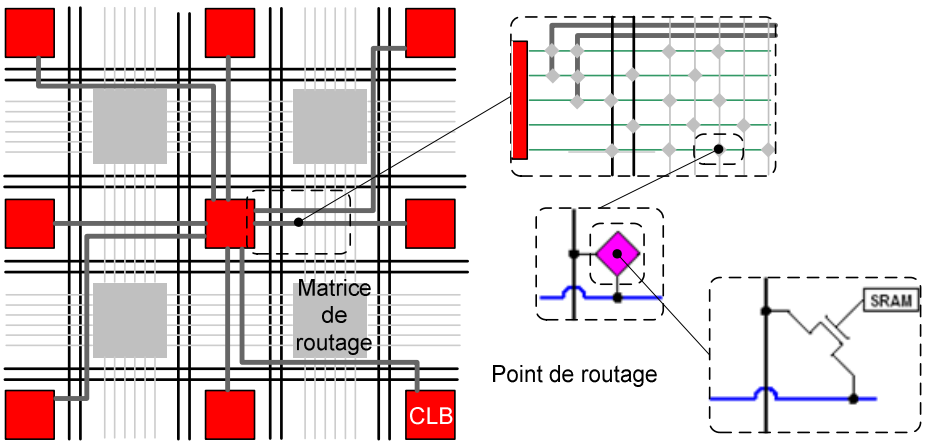


Figure 1.6: Structure générale du routage

1.3.3 Les caractéristiques des FPGA

Grâce aux évolutions de la technologie microélectronique, les FPGA deviennent de plus en plus performants avec des capacités sans cesse augmentées (plus que 330 000 *logic-cells* pour le circuit *Virtex 5* de *Xilinx* [18]). Longtemps réalisées autour de blocs de logique configurable à base de LUT, les FPGA peuvent aujourd'hui comporter de larges mémoires RAM configurables, des opérateurs arithmétiques complexes (comme les *Block Select RAM*, les blocs multiplieurs du *Virtex*) et des cœurs de microprocesseurs (tels que le cœur *microblaze* intégré sur *Virtex 4* [12]). La *table 1.1* présente la quantité de ressources disponibles pour les différents composants de la série *Virtex 5* proposée par le fabricant *Xilinx*.

	VIRTEX-5	LX	LXT	SXT	FXT	
	Part Number	XC5VLX330	XC5VLX330T	XC5VSX240T	XC5VFX100T	XC5VFX200T
Logic Resources	Slices	51 840	51 840	37 440	16 000	30 720
	Logic Cells	331 776	331 776	239 616	102 400	196 608
	CLB Flip-Flops	207 360	207 360	149 760	64 000	122 880
Memory Resources	Maximum Distributed RAM (Kbits)	3 420	3 420	4 200	1 240	2 280
	Block RAM/FIFO w/ECC (36 Kbits	288	324	516	228	456
	Total Block RAM (Kbits)	10 368	11 664	18 576	8 208	16 416
Clock Resources	Digital Clock Manager (DCM)	12	12	12	12	12
	Phase Locked Loop (PLL)	6	6	6	6	6
I/O Resources	Maximum Single-Ended Pins	1 200	960	960	680	960
	Maximum Differential I/O Pairs	600	480	480	340	480
Embedded Hard IP Resources	DSP48E Slices	192	192	1 056	256	384
	PowerPC 440 Processor Blocks	-	-	-	2	2
	PCI Express Endpoint Blocks	-	1	1	3	4
	10/100/1000 Ethernet MAC Blocks	-	4	4	4	8
Configuration	Configuration Memory (Mbits)	79,8	82,7	79,7	39,4	70,9

Table 1.1: Ressources de la série *Virtex 5* [18]

1.3.4 Le développement des systèmes sur FPGA

En général, le développement des systèmes sur FPGA se réalise, après la description du système, en cinq étapes principales (*figure 1.7*) :

- la synthèse logique,
- la simulation fonctionnelle,
- la projection et le placement / routage,
- la simulation temporelle,
- la génération de fichier de configuration.

La saisie d'un schéma à partir de cellules de base permet un développement "bas

niveau" qui rend difficile la réalisation de circuits complexes où chaque changement où amélioration remet en cause toute la description. Cette contrainte a conduit à étudier des techniques de génération de circuits à partir de spécifications de "haut niveau" qui sont basées sur l'utilisation d'un langage de description de matériel (Verilog, VHDL et SystemC …etc).

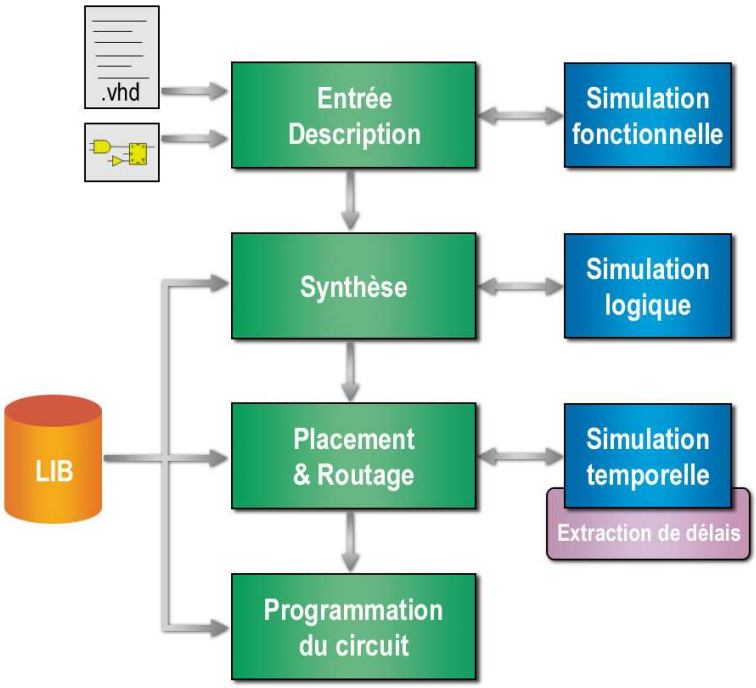


Figure 1.7: Flot de conception pour FPGA

4.4. LE LANGAGE VHDL [7, 19]

Le langage ***VHDL*** (*VHSIC Hardware Description Language, avec VHSIC : Very High Speed Integrated Circuits*) permet la description de tous les aspects d'un système matériel (*Hardware system*) : son comportement, sa structure et ses caractéristiques temporelles. Le VHDL est donc plus qu'un langage de description haut niveau :

- La description en VHDL est aussi simulable: Il est possible de lui appliquer des stimuli (également décrits en VHDL) et d'observer l'évolution des signaux du modèle dans le temps.
- Le langage VHDL est aussi utilisé pour la synthèse, pour produire automatiquement à partir d'une description au niveau RTL ou algorithmique, un circuit optimisé à base de portes logiques. Cette application très importante du langage sort toutefois du cadre de sa définition initiale et comporte des limitations.

- Le langage VHDL est un standard IEEE depuis 1987 sous la dénomination IEEE Std. 1076-1987 (VHDL-87). Il est sujet à révision tous les cinq ans. La dernière révision est celle de 2007 (IEEE Std. 1076-2007 ou VHDL-2007).

1.4.1 VHDL dans le flot de conception

Le flot de conception basé sur VHDL part d'une description du système à réaliser au niveau RTL comme indiqué à la *figure 1.8*. Le VHDL intervient dans plusieurs étapes: la modélisation, la simulation et la synthèse. La réalisation du placement et routage du circuit sous forme de *layout* est réalisée par un outil de placement et de routage qui nécessite une description d'entrée dans un format différent de VHDL: EDIF ou XNF est alors utilisé.

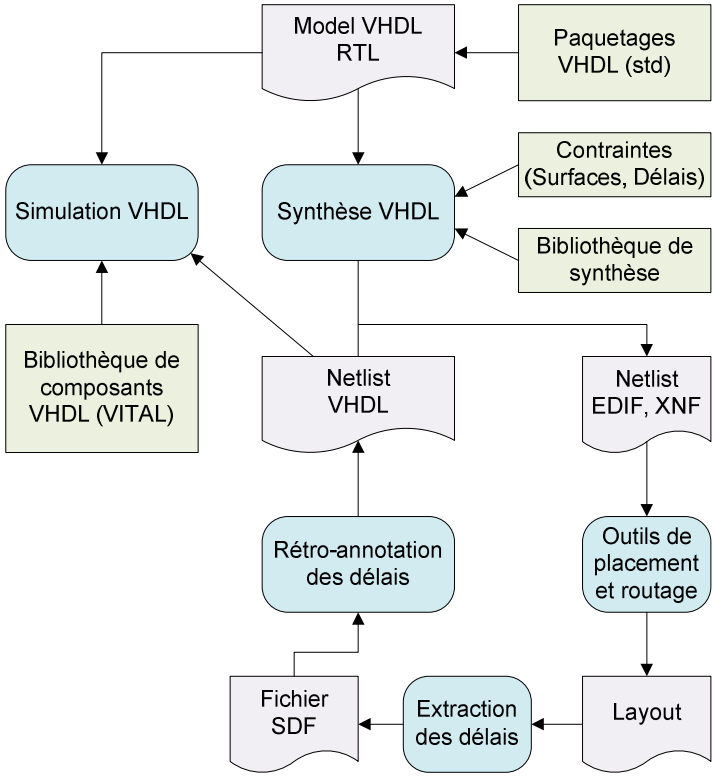


Figure 1.8: Flot de conception basé sur VHDL

1.4.2 VHDL et modélisation

La structure d'une description VHDL est composée de deux parties: Entité et architecture (*figure 1.9*).

1/ Entité : L'entité définit la vue externe ou l'interface d'un composant matériel. L'énoncé **entity** définit le nom de l'entité qui fait référence au type de composant, aux noms des ports d'entrées/sorties, direction des ports (entrée, sortie entrées/sorties) ainsi que leurs types logiques (un port de 1 bit, un bus de 8 lignes …etc).

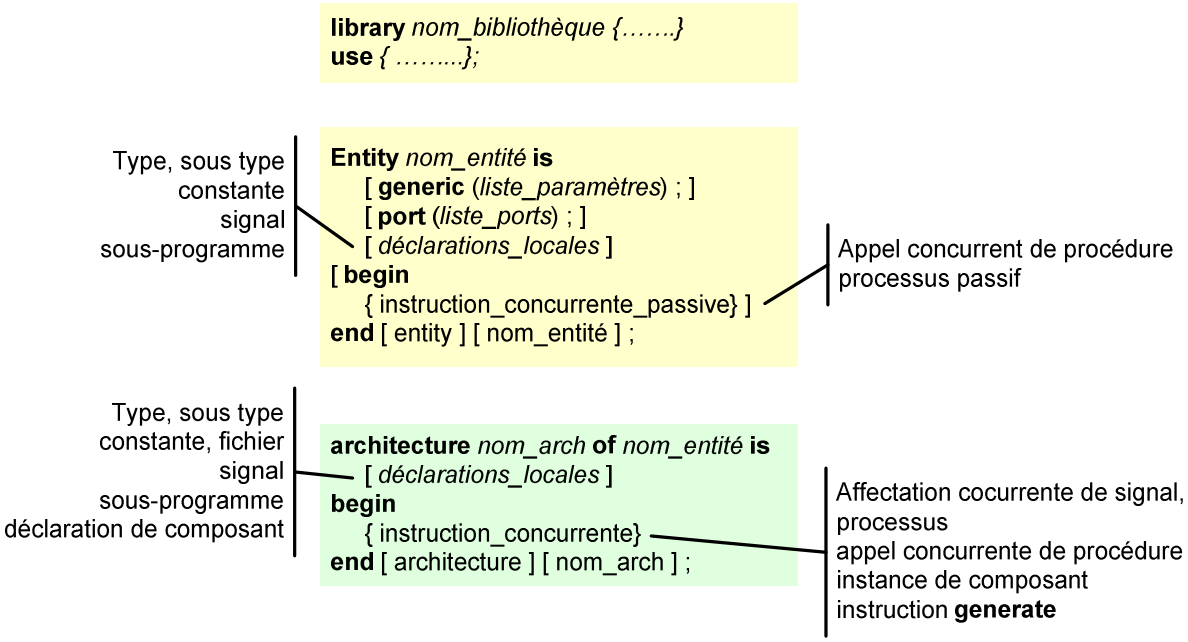


Figure 1.9: Structure d'une description VHDL

2/ Architecture : Elle contient la réalisation de la fonction. L'énoncé **architecture** contient un identificateur (nom) de l'architecture ainsi que l'indication du nom de l'énoncé *entity* qui définit ses entrées/sorties. Il contient aussi la déclaration des signaux internes et des sous éléments de ce circuit. Le langage VHDL permet trois type de réalisation de l'architecture : fonctionnelle, algorithmique et structurelle [19, 7].

- ***La description fonctionnelle*** (*flot de données*) décrit les transformations d'un flot de données de l'entrée à la sortie (un ensemble d'opérations logiques appliquées à des signaux). Ce style est adapté à la description de comportements logiques combinatoires.
- ***La description algorithmique*** consiste à décrire le comportement du composant sous la forme d'une séquence d'instructions appelée **process**. VHDL dispose de toutes les instructions séquentielles essentielles: instructions conditionnelles, sélectives et de boucles.
- ***La description structurelle*** consiste à décrire un modèle sous la forme d'une interconnexion de composants communiquant par l'intermédiaire de signaux. Cette description utilise le mécanisme dit d'***instanciation*** qui permet d'inclure directement une entité de conception dans le modèle. Par analogie avec un circuit imprimé, on "soude" chaque instance d'entité de conception (de composant discret) dans l'architecture (sur la carte).

On note que plusieurs architectures peuvent être associées à une même entité et une architecture peut contenir à la fois des interconnexions de composants, des instructions concurrentes (flot de données) et des processus.

1.4.3 VHDL & synthèse

1.4.3.1 Les bases de la synthèse

La synthèse est une opération complexe qui a un grand impact sur les différentes parties du projet [7]. Elle influence le cycle de développement, la manière de valider le circuit, mais également la manière de le décrire. La synthèse consiste à transformer une description comportementale en un circuit optimisé à base de portes logiques.

La synthèse se base aussi sur une bibliothèque de synthèse contenant les descriptions de toutes les cellules (portes) disponibles dans la technologie utilisée. Les informations essentielles sont, pour chaque cellule: sa fonction (*ex:* NAND, flip-flop), sa surface et les délais associés aux chemins d'entrées-sorties.

La synthèse se réalise selon des objectifs (exemple : *optimisation*). L'optimisation est gouvernée par un ensemble de contraintes fournies par l'utilisateur sur la surface et/ou les délais que doit satisfaire le circuit. La définition de ces contraintes est établie dans un langage propre à l'outil de synthèse. La phase d'optimisation logique cherche à reformuler les équations logiques (pour un circuit combinatoire) ou à minimiser le nombre d'états (pour un circuit séquentiel) représentant la fonction d'un bloc RTL.

Le résultat de la synthèse logique est un circuit de portes logiques dont on peut en tirer une nouvelle description VHDL (*figure 1.8)*. Les modèles VHDL des portes logiques sont usuellement définis selon la norme VITAL (*VHDL Initiative Towards ASIC Libraries*) pour permettre la rétro-annotation des délais dûs aux interconnexions [20, 21].

1.4.3.2 Le sous-ensemble VHDL pour la synthèse

Le VHDL est un langage très puissant et modulaire. La possibilité de créer de nouveaux types de signaux, de surcharger des procédures et des fonctions fait plus ressembler le VHDL à un langage orienté objet qu'à un langage de description de matériel. La gestion des fichiers ou la description de fonctions est certes très intéressante, mais leur intérêt pour la synthèse est limité. Cette prolifération de moyens, peut aboutir à un code trop complexe qui aboutit à un design lourd et confus. En se limitant à un sous ensemble de fonctionnalités offertes par VHDL, on peut implémenter de manière efficace et rapide n'importe quelle fonction. Le VHDL de synthèse est un sous-ensemble du VHDL généraliste.

En effet, ce n'est parce qu'une fonction peut être décrite en VHDL qu'elle soit synthétisable. La difficulté est de savoir ce qui est synthétisable ou pas. Il y a certains cas clairs comme l'instruction **after**. Cette instruction fait référence au temps, elle est donc non synthétisable. Mais la frontière du domaine des descriptions VHDL synthétisables n'est pas toujours aussi nette. Dans beaucoup de cas, c'est la manière dont les instructions sont utilisées qui rend la description non synthétisable. La limite peut même varier d'un synthétiseur à un autre. En prenant des précautions et en pensant toujours circuit lorsque nous écrivons des descriptions, il est possible d'assurer que celles-ci seraient synthétisables.

De plus, afin d'optimiser le résultat produit par un synthétiseur, il est nécessaire de s'imposer certaines limitations, voici quelques conseils d'un expert [20]

1- L'emploi de code concurrent: les fonctionnalités décrites de manière concurrentes sont souvent découpées en petites fonctions élémentaires, plus proches du matériel. Par contre, le code séquentiel a tendance à créer des process de plus en plus gros avec des **if** et des **case** imbriqués, rendant le code moins robuste et illisible.

2- Il faut se limiter à la logique "standard" définie dans le paquetage *IEEE 1164*. il est déconseillé d'employer des librairies fournies par le synthétiseur, car on peut facilement implémenter ces fonctions avec le package *IEEE std_logic_unsigned* et garder ainsi une portabilité du code VHDL.

3- Les synthétiseurs sont excellents pour optimiser la logique de très bas niveau, par contre, ils ont beaucoup plus de peine à analyser le design dans son ensemble: le travail de l'ingénieur est donc de découper les fonctions pour que le synthétiseur sera utilisé au maximum de ses capacités.

4- Les éléments mémoires étant décrits dans des **process**, il est déconseillé d'y incorporer la logique combinatoire, car non seulement, ça n'apporte aucun avantage, mais crée des structures en rupture avec les structures physiques.

1.4.4 VHDL & validation

La validation des conceptions (design) est l'élément primordial du travail de développement, puisqu'elle permet de diminuer les risques et les coûts en permettant une détection anticipée des anomalies. Par expérience, 80% des erreurs découvertes lors des tests ou de l'utilisation auraient pu être détectées facilement lors de la simulation, 10% des erreurs étant dues à des problèmes de timing et 10% à des situations imprévisibles (ou difficilement prévisibles) [20]. La phase de simulation est certainement la plus longue du cycle de conception d'un circuit intégré.

Il existe plusieurs types de simulations relatives aux différentes phases de développement, on cite par exemple :

Simulation logique: Le modèle VHDL peut être validé par simulation logique en utilisant des *testbenchs* écrits en VHDL. Un *testbench* comprend une entité vide, la déclaration d'une instance du composant à tester et un ensemble de stimuli ou vecteurs de test. La simulation après synthèse logique ne prend cependant en compte que les délais des portes.

Simulation temporelle: Après placement et routage, il est possible d'en extraire les valeurs des éléments parasites (résistances, capacités) associés aux interconnexions et de calculer les délais correspondants. Ces délais sont stockés dans un fichier au format SDF (*Standard Delay Format*), puis réinjectés dans la description VHDL obtenue après synthèse logique (*figure 1.8*). Cette fois ci la simulation (après placement & routage) prend en compte les délais des portes logiques et des interconnexions.

4.5. LE KIT DE DEVELOPPEMENT XILINX SPARTAN-3E [22]

Les kits de développement permettent de valider des conceptions à moindres coûts et en temps réduit. Munis de ressources matérielle et logicielles, ces outils constituent désormais des moyens efficients dans le processus de développement d'un circuit complexe. Plus performantes, plus complexes, très riches en ressources optionnelles et peu couteuses sont les caractéristiques des cartes de développement actuelles [23].

Dans le cadre de notre travail, nous utiliserons le Spartan-3E Starter Kit (*figure 1.10*) afin de valider notre IP. Les caractéristiques de la carte de ce kit seront présentées dans la section suivante.

Figure 1.10: Spartan-3E Starter Kit

Les principales caractéristiques de la carte Spartan-3E Starter Kit sont (*figure 1.11*) :

- FPGA Xilinx de type Spartan-3E XC3S500E
- 4 Mbit PROM Flash Xilinx de configuration de plate-forme

- CPLD Xilinx de type CoolRunner XC2C64A, 64-macrocell
- 64 MByte (512 Mbit) de DDR SDRAM, interface de données x16, 100+ MHz
- 16 MByte (128 Mbit) de PROM Flash parallèle (Intel StrataFlash)
- 16 Mbits de PROM Flash série SPI (STMicro)
- Afficheur LCD 2 lignes, 16 caractères
- Port PS/2 de clavier ou souris
- Connecteur VGA
- 10/100 Ethernet PHY
- 2 ports RS-232 9-pin (style DTE et DCE)
- Interface USB de chargement/débogage de FPGA/CPLD
- Oscillateur d'horloge de 50 MHz
- 4 sorties de convertisseurs Numérique-Analogique(DAC) (SPI)
- 2 entrées de convertisseurs Analogique-Numérique(ADC) (SPI) de gain programmable pré-amplifié
- Encodeur à axe rotatif avec bouton poussoir
- 8 LEDs
- 4 interrupteurs
- 4 boutons poussoirs
- Entrée d'horloge type SMA
- 8-pin DIP socket pour oscillateur d'horloge auxiliaire

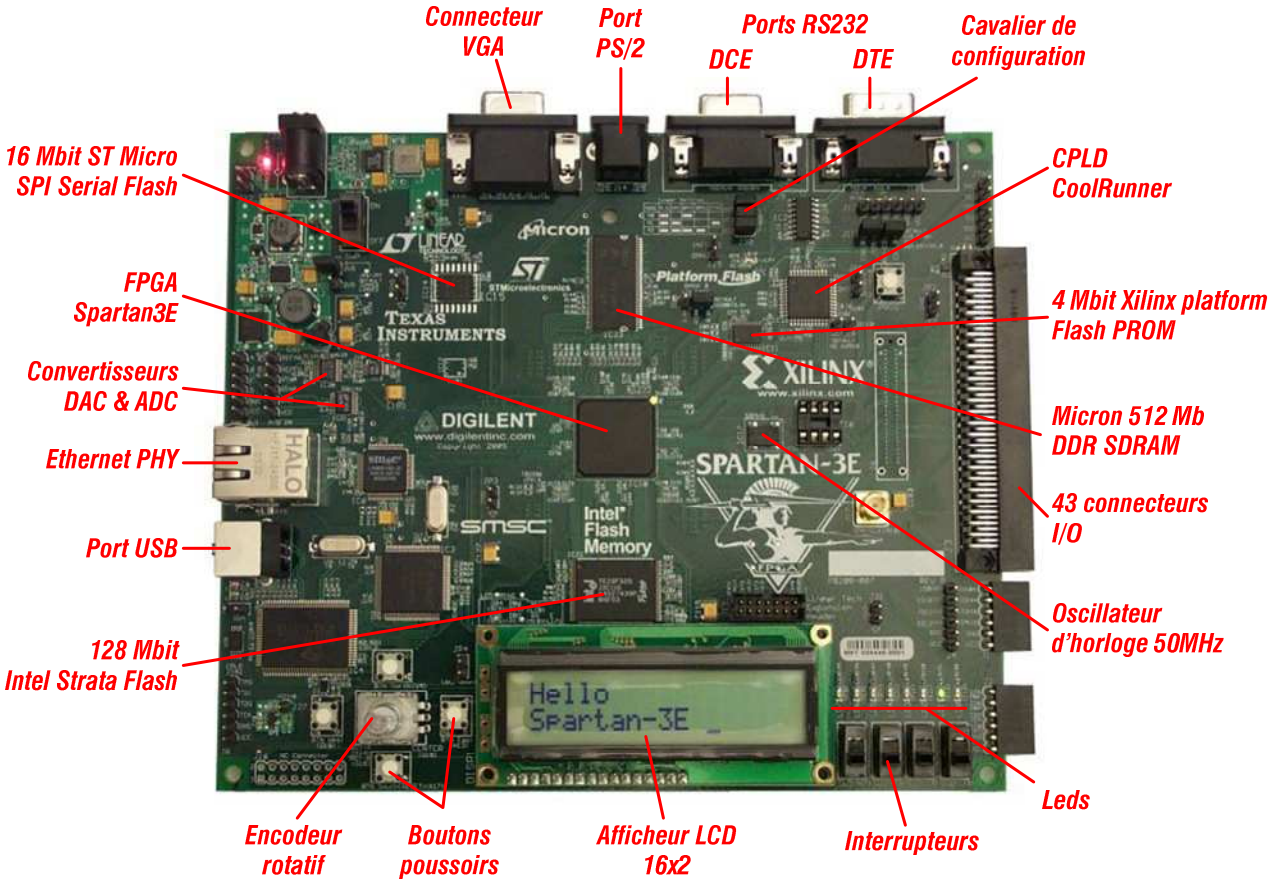


Figure 1.11: Carte de développement du Spartan-3E Starter Kit

1.5.1 Le FPGA Spartan-3E XC3S500E [24]

La Spartan-3E XC3S500E (voir *table 1.2*) se présente sous le format d'un package 320 pins FBGA (*Fine Pitch Ball Grid Array*) (*annexe 1*) et contient plus de 10 000 cellules logiques. Les détails de cette FPGA sont donnés dans les figures 1.12, 13, 14 et 15 :

Device	System Gates	Equivalent Logic Cells	CLB Array (One CLB = Four Slices) Rows	Columns	Total CLBs	Total Slices	Distributed RAM bits	Block RAM bits	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differetial I/O Pairs
XC3S100E	100K	2 160	22	16	240	960	15K	72K	4	2	108	40
XC3S250E	250K	5 508	34	26	612	2 448	38K	216K	12	4	172	68
XC3S500E	500K	10 476	46	34	1 164	4 656	73K	360K	20	4	232	92
XC3S1200E	1200K	19 512	60	46	2 168	8 672	136K	504K	28	8	304	124
XC3S1600E	1600K	33 192	76	58	3 688	14 752	231K	648K	36	8	376	156

Table 1.2: Ressources de la série Spartan-3E

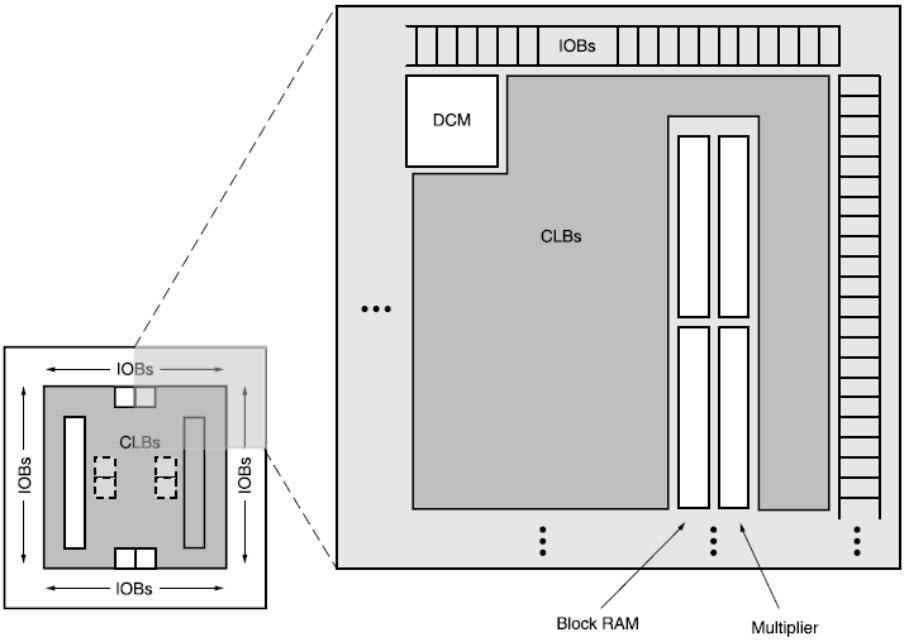


Figure 1.12: Architecture des Spartan-3E

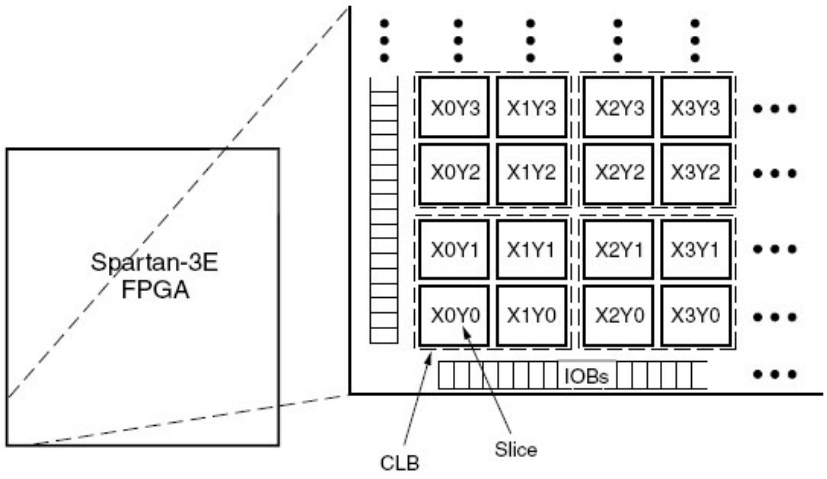


Figure 1.13: Emplacement des CLB dans Le FPGA

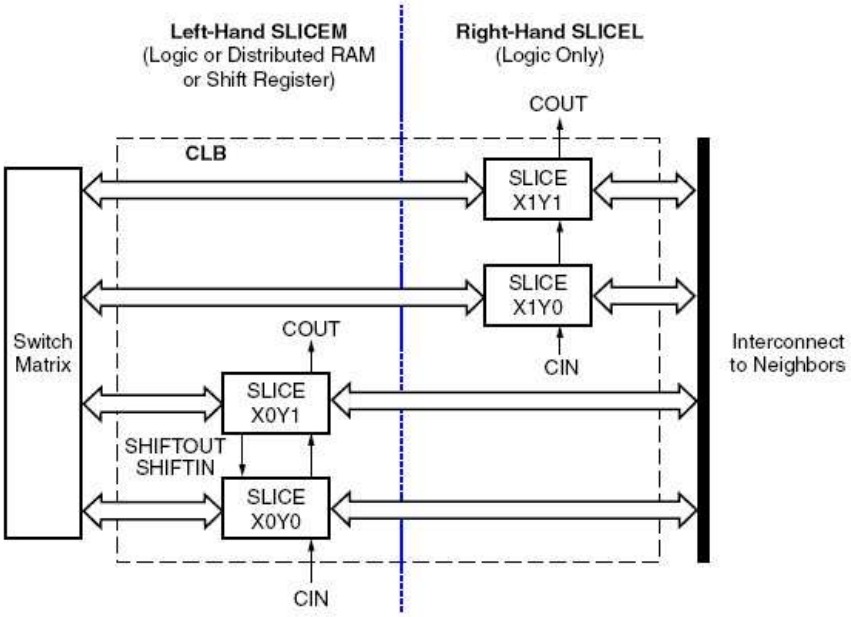


Figure 1.14: Arrangement des Slices dans le CLB

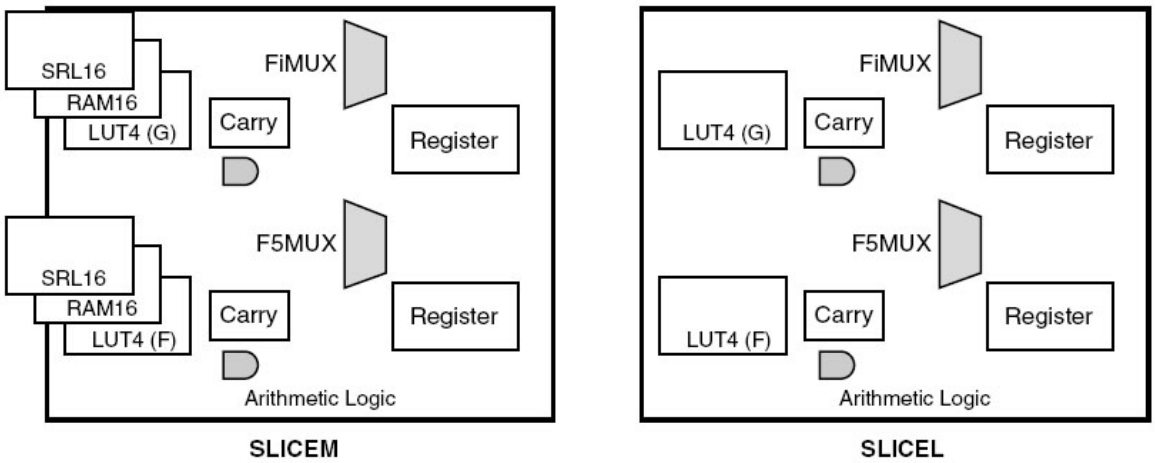


Figure 1.15: Ressources d'un Slice

1.5.2 Les options de configuration du FPGA

Une application typique des FPGA utilise une mémoire non volatile pour sauvegarder la configuration qui s'effectue après la mise sous tension du circuit. Le Stater Kit utilisé offre, entre autres, trois différentes mémoires sources de configuration et supporte les méthodes de configuration suivantes :

- Chargement de la configuration directement dans le FPGA via JTAG, en utilisant le port USB.
- Chargement de la configuration à partir de la mémoire :
    - PROM Flash Xilinx, en utilisant Master serial mode,
    - PROM Flash série SPI, en utilisant SPI mode,
    - PROM Flash parallèle (StrataFlash), en utilisant BPI mode.

Le mode de configuration du FPGA (source de configuration) est sélectionné par un cavalier (jumper) [22].

1.5.3 Programmation du FPGA à travers le port USB

Après une compilation réussie d'un design sur un FPGA cible sous l'environnement de développement Xilinx, on peut procéder à la programmation du FPGA à travers le port USB. La programmation de FPGA, dans ce cas, consiste à charger sa configuration via le port USB en utilisant l'outil iMPACT (*figure 1.16*).

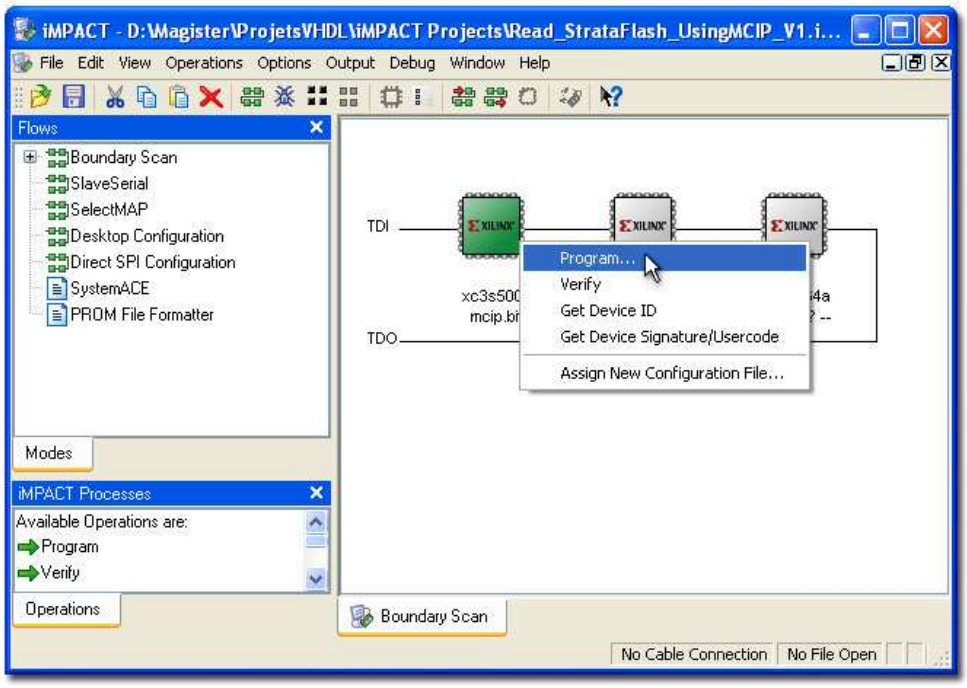


Figure 1.16: Programmation du FPGA avec l'outil iMPACT

4.6. CONCLUSION

Dans ce chapitre, nous avons présenté les techniques de réalisation des systèmes sur puces en termes d'aspects matériels (composants reprogrammables, ASIC) et en termes d'aspects logiciels (HDL, synthèse logique, placement & routage). Nous pensons que les nouvelles technologies de FPGA permettent une évolutivité croissante et donc peuvent assurer tout nouvel besoin et le couvrir à terme sans nécessité de recours à repenser les architectures dans leur totalité.

L'exploitation d'un langage de haut niveau tel VHDL permet de multiplier les essais, d'optimiser de diverses manières l'architecture développée et de vérifier à divers niveaux de simulation la fonctionnalité des produits conçus.

Le Spartan-3E Starter Kit est utilisé, dans ce travail, pour développer et concevoir un système de gestion des horaires de prière grâce à un microcontrôleur de type IP et sous l'environnement de Xilinx ISE et du simulateur ModelSim de *Mentor Graphics*.

Chapitre 2
Architecture du MCIP

2.1. DESCRIPTION DU PROJET

Le but de ce travail est la conception d'un microcontrôleur **MCIP** (**M**icro-**C**ontroller IP) sous la forme d'un IP soft. Pour y parvenir très rapidement, nous avons conçu notre microcontrôleur selon une architecture compatible à celle d'un PIC18 de *Microchip* [25] (*annexe 2*) et qui répond aux spécifications de notre cahier des charges. Les justificatifs liés directement à la sélection d'une architecture existante et sa transformation en IP sont donnés ci-dessous :

- Un microcontrôleur sous format d'un core IP trouvera une grande utilisation dans le développement de SoC [7].
- Se référencier à une architecture existante optimisée revient généralement moins cher d'une part et préserve les atouts du champ d'application d'autre part.
- Se référencier à une architecture existante est une aide précieuse certaine mais le produit à développer risque de ne pas produire d'effet spectaculaire. Alors nous avons considéré la transformation à l'échelle reconfiguration, test et particulièrement adopter une solution générique à différents niveaux de conception pour réduire les efforts de vérification et limiter les spécifications. Néanmoins, le choix des différents types de périphériques et des caractéristiques spéciales demeurera.
- Le choix de l'architecture type PIC18 est fortement lié à la domination du marché mondial des microcontrôleurs de *Microchip* qui, non seulement possèdent un environnement de développement riche et performant (disponible pour la validation de notre IP) mais aussi qualifiés de circuits à rendements élevés dus à un certain nombre de dispositifs architecturaux généralement trouvés dans des microprocesseurs RISC (*Harvard architecture, Long Word Instructions, Single Cycle Instructions, Instruction Pipelining, Reduced Instruction Set*) [26].
- La conception de notre IP au niveau comportemental bénéficiera des avantages de portabilité, de reconfiguration et donc trouvera une place privilégiée dans les systèmes embarqués (*embedded systems*) grâce à la plateforme de VHDL.
- Indisponibilité à la date de présent projet d'un IP compatible au PIC18, mais disponibilité d'une large gamme d'IP compatibles au PIC16 et à ses concurrents (*table 2.1*).

- Le PIC18 n'est pas une simple structuration du PIC16 mais dispose de ressources largement supérieures et d'une structure complexe.

Part Number	*Product Description*	*Provider*
MIL- PIC 16xx	8-bit, high performance, software IP Core, fully compatible with popular RISC microprocessor PIC16F84 or PIC16F74	Microtech International
PIC16C6x	PIC 16C6X compatible	Full Custom Design
ACPIC	PIC core (PIC16C55)	ASICentrum
DRPIC166X	High Performance Configurable 8-bit RISC Microcontroller	Digital Core Design
DRPIC1655X	Single Cycle High Performance 8-bit RISC Microcontroller	Digital Core Design
DFPIC165X	High Performance 8-bit RISC Microcontroller	Digital Core Design
DFPIC1655X	High Performance 8-bit RISC Microcontroller	Digital Core Design
CorePool FHG_8051	Parameterizable High Performance 8-Bit Microcontroller Core	Fraunhofer IIS - CorePool
DW8051	8051 8-Bit MicroController	Synopsys
Flip8050-Cyclone	microcontroller - Highest processing power	Dolphin Integration

Table 2.1: Exemples d' IP microcontrôleurs commercialisés par *Design & Reuse* (PIC16 et µC Intel) [27]

2.2. DESCRIPTION DE L'ARCHITECTURE DE MCIP

Elle est basée sur une architecture Harvard où la mémoire de programme est séparée de celle des données (bus séparés). Ceci améliore, pour un certain nombre d'exploitation, les performances comparativement à celles produites par l'architecture traditionnelle de Von Neumann où le programme et les données sont cherchés à partir de la même mémoire à l'aide du même bus. Notre bus de données est sur 8 bits et la largeur de l'instruction est sur 16 bits. L'architecture du microcontrôleur développé est donnée sur la *figure 2.1*.

Les principales caractéristiques de cette architecture sont expliquées ci-dessous. Pour éviter de reproduire les explications standards liées aux différents blocs de cette architecture (voir manuel de PIC18C de *Microchip*), nous avons explicité seulement les principaux éléments où s'est manifesté notre contribution.

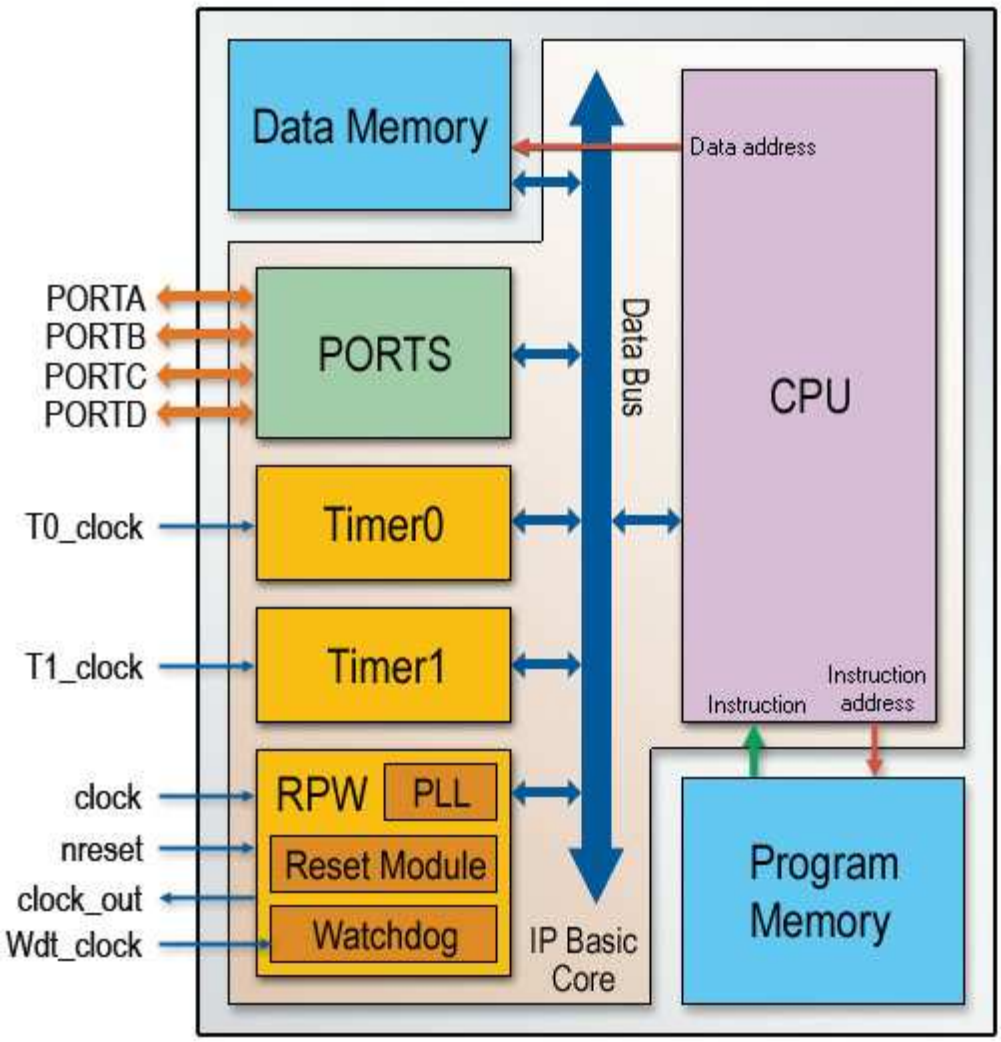


Figure 2.1: Architecture du microcontrôleur MCIP

2.2.1 Instruction et timing

Notre horloge d'entrée ***clock*** est localement divisée par 4 moyennant une **PLL** pour produire quatre horloges 'non recouvrantes', à savoir ***Q1***, ***Q2***, ***Q3*** et ***Q4***. Le compteur programme est incrémenté à chaque ***Q1***, et l'instruction est cherchée de la mémoire et verrouillée dans le registre d'instruction entre ***Q1*** et ***Q4***. En effet l'instruction est décodée et exécutée pendant le prochain cycle ***Q1*** - ***Q4***. Un "cycle d'instruction" ***TCY*** se compose de quatre cycles de ***Q*** (***Q1***, ***Q2***, ***Q3*** et ***Q4***) (*figure 2.2*). La largeur d'instruction étant de 16 bits, l'instruction entière est donc cherchée dans un cycle simple de machine, excepté pour les instructions double mots. Dans le cas de branchement où le contenu du compteur programme est modifié, il y a un retard d'un cycle dans l'exécution du programme.

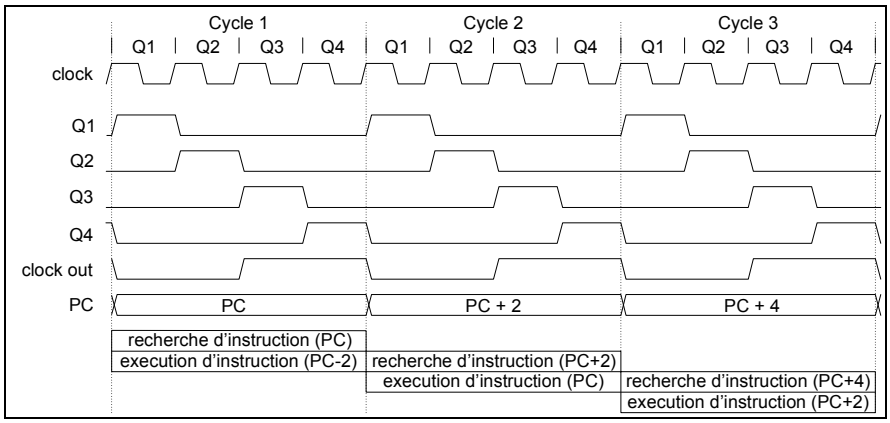


Figure 2.2: Cycle d'instruction & flot de pipeline

Les détails d'un cycle d'exécution d'instruction en relation avec les cycles ***Qi*** sont propres à chaque instruction. Ces détails peuvent être généralisés comme suit:

Q1: Cycle de décodage de l'instruction,

Q2: Cycle de lecture d'une donnée,

Q3: Traitement de la donnée,

Q4: Cycle d'écriture de données.

2.2.2 Pipeline d'instruction

L'architecture est basée sur un pipeline à deux étages qui recouvre les cycles recherche et exécution des instructions (*figure 2.2*). La recherche de l'instruction prend un ***TCY***, alors que l'exécution prend un autre ***TCY***. Cependant, en raison du chevauchement, une instruction s'exécute efficacement en un cycle (Single Cycle Instruction).

2.2.3 CPU

L'unité centrale de traitement est responsable de l'interprétation des instructions chargées dans la mémoire de programme pour contrôler l'opération du circuit. Plusieurs instructions traitent des données d'où la nécessité d'une unité arithmétique et logique. En plus de l'accomplissement d'opérations arithmétiques et logiques, l'**ALU** contrôle les bits d'état qui composent le registre **STATUS**. L'unité centrale de traitement commande les accès aux mémoires de données et de programme.

2.2.3.1 Compteur de programme

Le compteur de programme (**PC**) indique l'adresse de l'instruction à chercher pour l'exécution. Le **PC** est sur 21 bits, et adresse chaque octet (plutôt que mot) dans la mémoire de programme. L'octet inférieur (PC<7:0>) s'appelle **PCL**. Ce registre peut être lu et écrit.

L'octet (PC<15:8>) s'appelle **PCH**. La lecture et l'écriture directes de ce registre ne sont pas possibles: ces opérations peuvent être exécutées à travers le registre **PCLATH**. L'octet supérieur (PC<23:16>) s'appelle le registre **PCU**. Comme le **PCH**, la lecture et l'écriture de **PCU** sont exécutées indirectement à travers le registre **PCLATU**. La structure de **PC** est **PCU<4:0>:PCH<7:0>:PCL<7:0>** et elle est équivalente à **PC<20:0>** (*figure 2.3*).

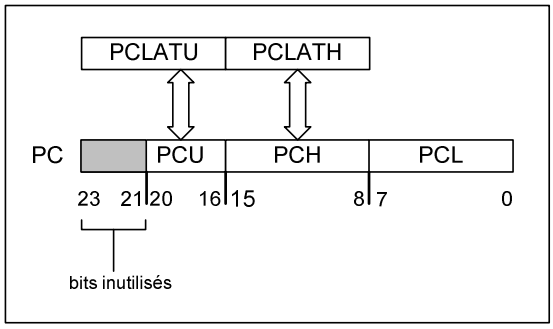


Figure 2.3: Structure du compteur programme

2.2.3.2 Unité arithmétique et logique (ALU)

Le microcontrôleur contient une **ALU** de 8 bits et un registre de travail **WREG** de 8 bits. L'**ALU** est capable d'exécuter des opérations entre les données dans le registre de travail **WREG** et n'importe quel autre registre de données (*file register*) (*figure 2.4*).

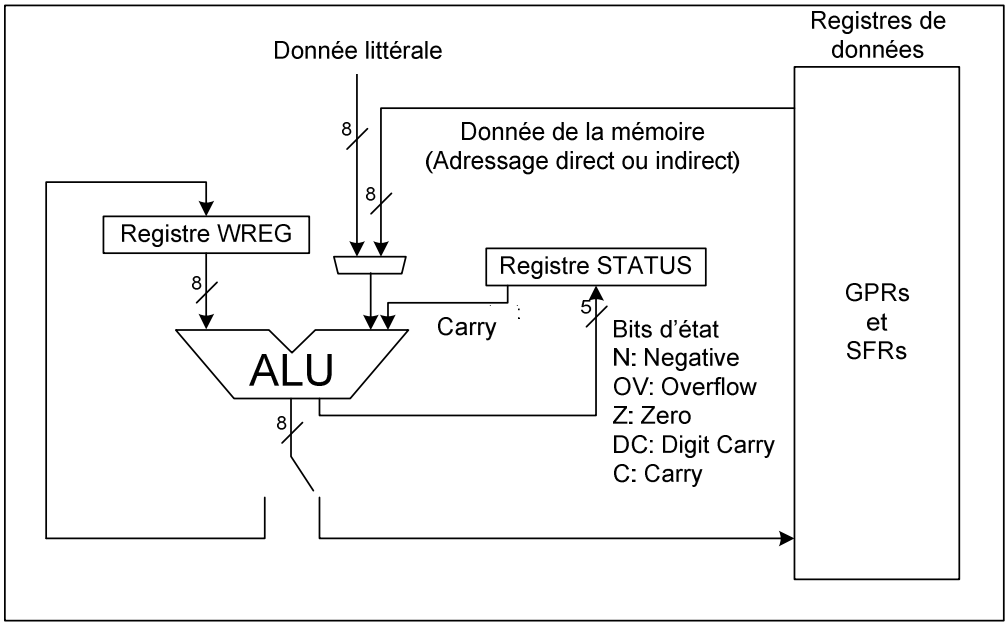


Figure 2.4: Fonctionnement de l'ALU

2.2.3.3 Multiplieur câblé 8x8

L'architecture intègre un multiplieur câblé qui permet d'augmenter les performances en temps et en ressources par rapport à une multiplication logicielle. Cet apport de performance

permet de développer des applications réservées aux DSP. Le multiplieur 8x8 fonctionne en un cycle simple et place le résultat de 16 bits dans la paire de registres de 8 bits <**PRODH**:**PRODL**>.

2.2.4 Organisation de la mémoire

Il y a deux blocs de mémoire dans l'architecture; mémoire de programme et mémoire de données. Chaque bloc a son propre bus, de sorte que l'accès à chaque bloc puisse se produire pendant le même cycle d'instruction.

2.2.4.1 Mémoire de programme

Le compteur programme, étant de 21 bits, est capable d'adresser jusqu'à 2M octets (1M mots) d'espace mémoire de programme. L'espace de programme est mis en application comme bloc contigu simple. Le vecteur de reset est à l'adresse ***000000h***, le vecteur d'interruption prioritaire élevé est à l'adresse ***000008h***, et le vecteur faible priorité d'interruption est à l'adresse ***000018h*** (*figure 2*.5).

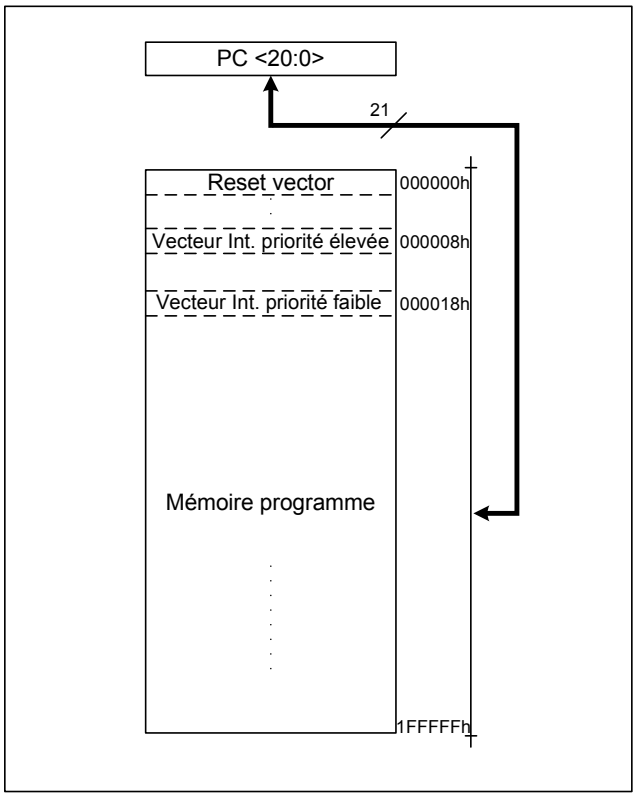


Figure 2.5: Organisation de la mémoire de programme

2.2.4.2 Mémoire de données

La mémoire de données peut être décomposée en **RAM** tout usage (**GPR**) (*General Purpose Registres*) et en registres de fonction spéciale (**SFR**) (*Special Function Registres*).

Les **SFR** sont employés pour commander l'état du microcontrôleur et les fonctions des modules périphériques. Les **GPR** sont employés pour le stockage de données de l'utilisateur. Chaque registre à une adresse de 12 bits, ce qui permet d'adresser 4096 octets. Cette mémoire est divisée en 16 Banques de 256 octets comme le montre la *figure 2.6*. Les adresses des différents **SFR** sont données par la *table 2.2*.

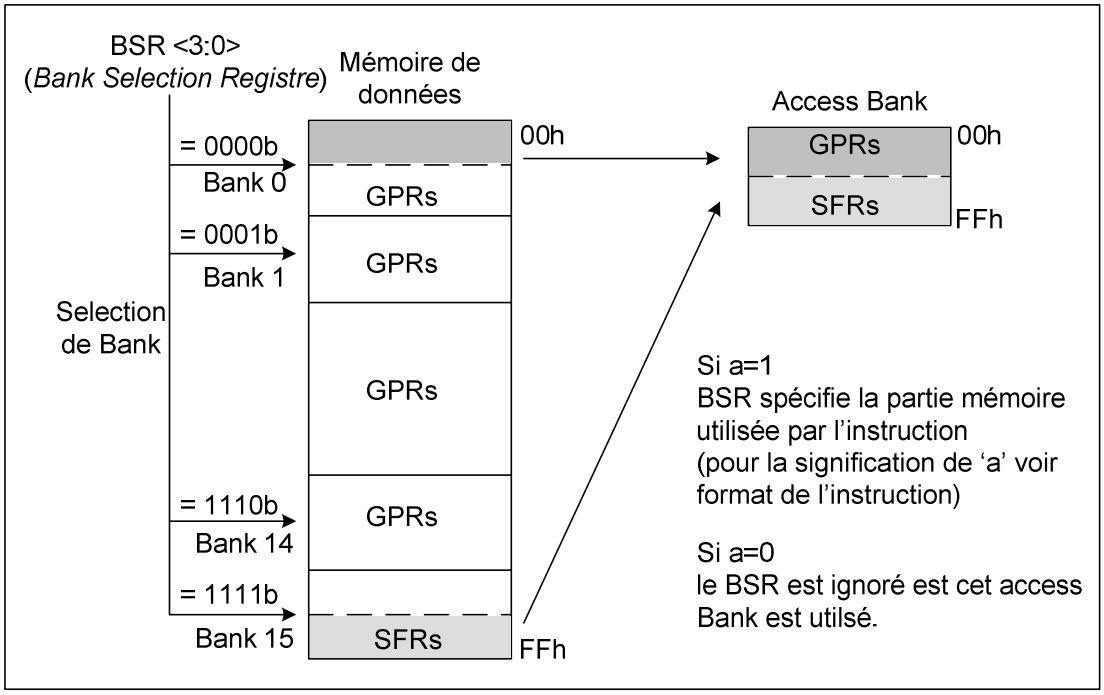


Figure 2.6: Organisation de la mémoire de données

Adress	Nom Registre	Adress	Nom Registre	Adress	Nom Registre	Adress	Nom Registre	Adress	Nom Registre
FFFh	TOSU	FEFh	INDF0	FDFh	INDF2	FCFh	TMR1H	F8Fh	-
FFEh	TOSH	FEEh	POSTINC0	FDEh	POSTINC2	FCEh	TMR1L	F8Eh	-
FFDh	TOSL	FEDh	POSTDEC0	FDDh	POSTDEC2	FCDh	T1CON	F8Dh	-
FFCh	STKPTR	FECh	PREINC0	FDCh	PREINC2	FCCh	-	F8Ch	LATD
FFBh	PCLATU	FEBh	PLUSW0	FDBh	PLUSW2			F8Bh	LATC
FFAh	PCLATH	FEAh	FSR0H	FDAh	FSR2H	.	.	F8Ah	LATB
FF9h	PCL	FE9h	FSR0L	FD9h	FSR2L	.	.	F89h	LATA
FF8h	TBLPTRU	FE8h	WREG	FD8h	STATUS	.	.	F88h	-
FF7h	TBLPTRH	FE7h	INDF1	FD7h	TMR0H	.	.	F87h	-
FF6h	TBLPTRL	FE6h	POSTINC1	FD6h	TMR0L	F96h	-	F86h	-
FF5h	TABLAT	FE5h	POSTDEC1	FD5h	T0CON	F95h	TRISD	F85h	-
FF4h	PRODH	FE4h	PREINC1	FD4h	-	F94h	TRISC	F84h	-
FF3h	PRODL	FE3h	PLUSW1	FD3h	-	F93h	TRISB	F83h	PORTD
FF2h	INTCON	FE2h	FSR1H	FD2h	-	F92h	TRISA	F82h	PORTC
FF1h	INTCON2	FE1h	FSR1L	FD1h	WDTCON	F91h	-	F81h	PORTB
FF0h	INTCON3	FE0h	BSR	FD0h	-	F90h	-	F80h	PORTA

Table 2.2: Les registres SFR

2.2.4.3 La Pile

La pile permet l'imbrication maximale de 31 appels de programme, elle contient les adresses de retour de branchement et elle est de taille de 32 x 21-bits. L'espace de pile n'est pas considéré comme une partie de mémoire programme ou de données.

Le PC est chargé (*PUSHed*) dans la pile quand une instruction **APPEL** est exécutée ou une interruption est servie. La pile est restituée (*POPed*) en cas d'un **RETURN**, de **RETLW** ou d'une exécution d'instruction de **RETFIE** *(figure 2.7)*. Quand le débordement de la pile se produit, le pointeur de pile **STKPTR** reste au top (à l'adresse ***11111b***) et un **RESET** général du dispositif est généré.

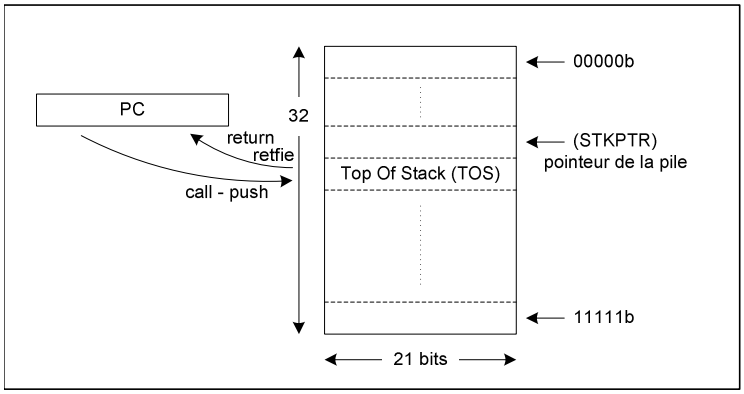


Figure 2.7: Organisation de la pile

2.2.5 La table de lecture

La table est conçue pour transférer les données entre la mémoire de programme et la mémoire de données. L'adresse de la donnée à chercher est fournie par **TBLPTR** (**TBLPTRU<4:0>:TBLPTRH:TBLPTRL**) et la donnée est placée dans le registre **TABLAT** (*figure 2.8*).

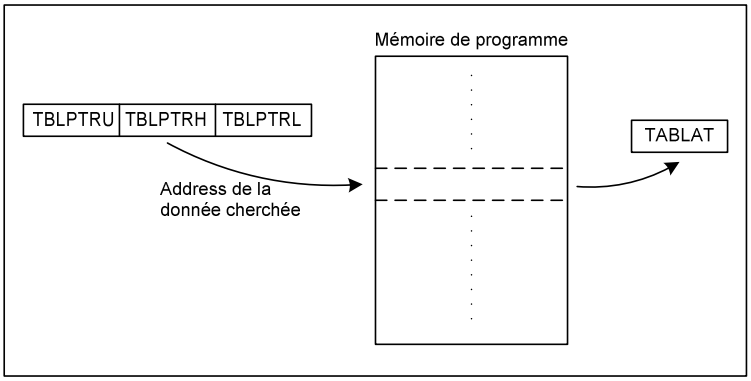


Figure 2.8: Utilisation de la table de lecture

2.2.6 Les ports d'E/S

Le microcontrôleur dispose des ports d'E/S de 8 bits. Ces ports permettent la gestion et le contrôle d'autres circuits à l'extérieur de MCIP. Notre circuit est doté de 4 ports **PORTA**, **PORTB**, **PORTC** et **PORTD**. Selon le mode de fonctionnement: chaque bit de port peut être configuré comme entrée ou sortie par un registre **TRIS**: pour le **PORTA** par exemple, si **TRISA** (0) = 0 alors le bit ***A0*** du port est configuré comme sortie, dans le cas contraire il est configuré comme entrée.

2.2.7 Les Timers

Le timer est en fait un compteur de 8/16 bits à cycle variable cadencé soit au rythme de l'horloge interne (**clock/4**) soit au rythme de l'horloge externe (**clock**). En effectuant une lecture de ce compteur, nous pouvons déterminer le "temps qui s'écoule" et faire des actions à des moments précis. Le timer est utilisé pour réaliser des temporisations précises. Il est composé principalement d'un pré-diviseur (*Prescaler*), d'un compteur ainsi que d'un système d'aiguillage piloté par des bits de configuration (*figure 2*.9). Notre architecture englobe deux timers identiques Timer0 et Timer1.

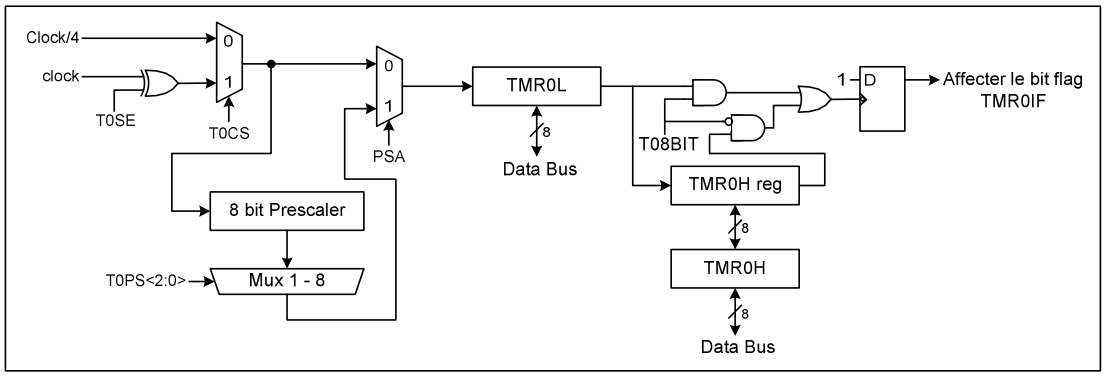


Figure 2.9: Structure de Timer

2.2.8 Les interruptions

Les sources d'interruption possibles prises en considération sont :

• Interruption externe sur **INT0** ou **INT1** (broches **RB0** ou **RB1**),

• Détection de changement d'état sur les broches **RB7:RB4,**

• Dépassement des Timers.

L'architecture met en application deux sortes d'interruptions programmées: de priorité élevée ou de priorité faible. Si la priorité d'interruption n'est pas employée, toutes les interruptions sont traitées en tant que priorité élevée. Quand une interruption est reconnue, le

PC est forcé d'adresser ***000008h*** ou ***000018h*** selon le niveau d'interruption. Quand le **PC** est forcé au vecteur d'interruption, les registres de **PCLATU** et de **PCLATH** ne sont pas modifiés. Avant que le registre de **PCLATH** soit modifié par la routine de service d'interruption, le contenu du **PCLATH** doit être sauvegardé ainsi il peut être réappelé. Le circuit logique du mécanisme d'interruption est illustré sur la *figure 2.10*. La *table 2.3* donne la description des registres de contrôle des interruptions.

INTCON		
BIT7	**GIE/GIEH**	Global Interrupt Enable bit
BIT6	**GIEL**	Peripheral Interrupt Enable bit
BIT5	**IPEN**	Interrupt Priority Enable bit
BIT4	**INT1IF**	INT1 External Interrupt Flag bit
BIT3	**TMR1IF**	TMR1 Overflow Interrupt Flag bit
BIT2	**TMR0IF**	TMR0 Overflow Interrupt Flag bit
BIT1	**INT0IF**	INT0 External Interrupt Flag bit
BIT0	**RBIF**	RB Port Change Interrupt Flag bit
INTCON2		
BIT7	**INT0IE**	INT0 External Interrupt Enable bit
BIT6	**INT0IP**	INT0 External Interrupt Priority bit
BIT5	**INT1IE**	INT1 External Interrupt Enable bit
BIT4	**INT1IP**	INT1 External Interrupt Priority bit
BIT3	**TMR1IE**	TMR1 Overflow Interrupt Enable bit
BIT2	**TMR1IP**	TMR1 Interrupt Priority bit
BIT1	**RBIE**	RB Port Change Interrupt Enable bit
BIT0	**RBIP**	RB Port Change Interrupt Priority bit
INTCON3		
BIT3	**INTEDG0**	External Interrupt0 Edge Select bit
BIT2	**INTEDG1**	External Interrupt1 Edge Select bit
BIT1	**TMR0IE**	TMR0 Overflow Interrupt Enable bit
BIT0	**TMR0IP**	TMR0 Interrupt Priority bit

Table 2.3: Les registres de contrôle des interruptions

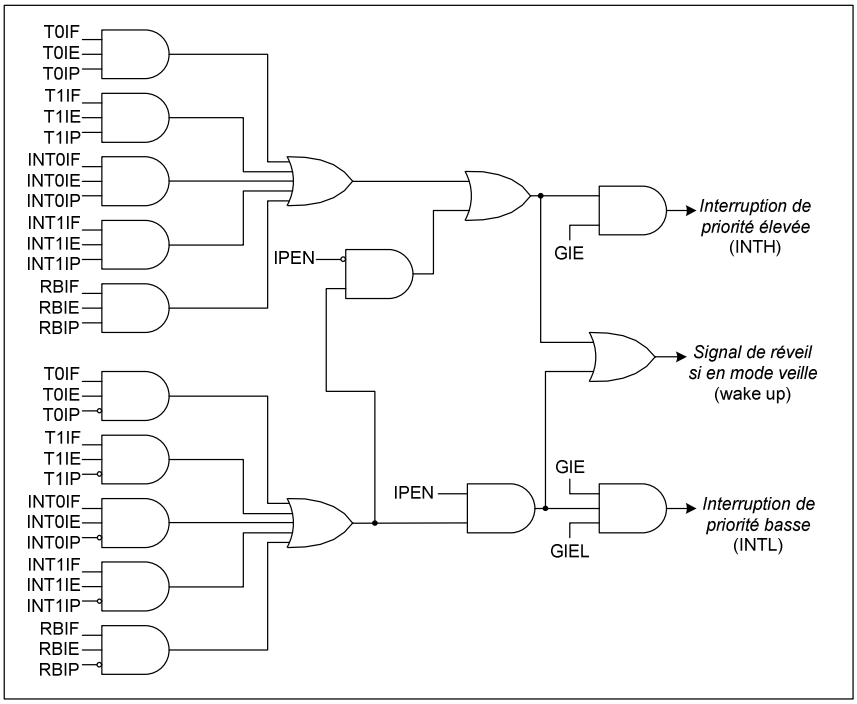


Figure 2.10: Architecture du module d'interruption

2.2.9 Watchdog et mode SLEEP

Le watchdog et le *sleep mode* sont deux mécanismes supplémentaires qui renforcent la fiabilité du MCIP. Le watchdog, étant un timer "free runing", est un mécanisme de surveillance du bon déroulement du programme. Il peut être utilisé pour restituer le mode de fonctionnement normal, ou causer un reset du circuit en cas d'anomalies de fonctionnement. Le watchdog est activé/désactivé par un bit de configuration WDTEN.

La fonction Sleep arrête l'activité du système et réduit sa consommation en énergie en arrêtant l'activité de la PLL. Dans ce mode, la consommation est très basse, permettant l'alimentation par batterie. Le fonctionnement normal peut être repris (sortie du mode Sleep) s'il y a une interruption, un reset ou expiration du temps du watchdog. La *figure 2.11* montre l'architecture du watchdog. La *table 2.4* donne la description du registre de contrôle du watchdog.

WDTCON		
BIT3	**WDTPS2**	Watchdog Timer Postscale Select bits
BIT2	**WDTPS1**	
BIT1	**WDTPS0**	
BIT0	**WDTEN**	Watchdog Timer Enable bit

Table 2.4: Le registre de contrôle du watchdog

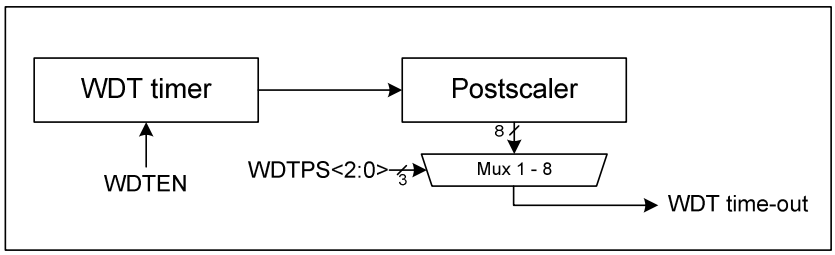


Figure 2.11 L'architecture du watchdog

2.2.10 Le jeu d'instructions

Le jeu d'instruction du MCIP est de 72 instructions (*Table 2.5*) qui comporte toutes les instructions du PIC18 à l'exception des 4 instructions liées à l'écriture sur PROM du type Flash ou compatible. La majorité des instructions est codée sur un simple mot (16 bits). Chaque instruction est divisée en un code d'opération **OPCODE** spécifiant le type d'instruction et un ou plusieurs opérandes. Les instructions sont groupées en cinq catégories :

- Opérations *Byte-oriented*,
- Opérations *bit-oriented*,
- Opérations littérales,
- Opérations de contrôle,
- Opérations de transfert de données de mémoire programme → mémoire de données.

Mnemonic, Opérands		Description	Cycles	16-bit instruction word				Status affected
				MSb			LSb	
BYTE-ORIENTED FILE REGISTER OPERATIONS								
ADDWF	f, d, a	Add WREG and f	1	0010	01da	ffff	ffff	C, DC, Z, OV, N
ADDWFC	f, d, a	Add WREG and carry bit to f	1	0010	00da	ffff	ffff	C, DC, Z, OV, N
ANDWF	f, d, a	AND WREG with f	1	0001	01da	ffff	ffff	Z, N
CLRF	f, a	Clear f	1	0110	101a	ffff	ffff	Z
COMF	f, d, a	Complement f	1	0001	11da	ffff	ffff	Z, N
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 ou 3)	0110	001a	ffff	ffff	None
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 ou 3)	0110	010a	ffff	ffff	None
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 ou 3)	0110	000a	ffff	ffff	None
DECF	f, d, a	Decrement f	1	0000	01da	ffff	ffff	C, DC, Z, OV, N
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010	11da	ffff	ffff	None
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100	11da	ffff	ffff	None
INCF	f, d, a	Increment f	1	0010	10da	ffff	ffff	C, DC, Z, OV, N
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0011	11da	ffff	ffff	None
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	0100	10da	ffff	ffff	None
IORWF	f, d, a	Inclusive OR WREG with f	1	0001	00da	ffff	ffff	Z, N
MOVF	f, d, a	Move f	1	0101	00da	ffff	ffff	Z, N
MOFF	f_s, f_d	Move f_s (source) to 1st word	2	1100	ffff	ffff	ffff	None
		f_d (destination) 2nd word		1111	ffff	ffff	ffff	
MOVWF	f, a	Move WREG to f	1	0110	111a	ffff	ffff	None
MULWF	f, a	Multiply WREG with f	1	0000	001a	ffff	ffff	None
NEGF	f, a	Negate f	1	0110	110a	ffff	ffff	C, DC, Z, OV, N
RLCF	f, d, a	Rotate Left f through Carry	1	0011	01da	ffff	ffff	C, Z, N
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100	01da	ffff	ffff	Z, N
RRCF	f, d, a	Rotate Right f through Carry	1	0011	00da	ffff	ffff	C, Z, N
RRNCF	f, d, a	Rotate Right f (No Carry)	1	0100	00da	ffff	ffff	Z,N
SETF	f, a	Set f	1	0110	100a	ffff	ffff	None
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	0101	01da	ffff	ffff	C, DC, Z, OV, N
SUBWF	f, d, a	Subtract WREG from f	1	0101	11da	ffff	ffff	C, DC, Z, OV, N
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C, DC, Z, OV, N
SWAPF	f, d, a	Swap nibbles in f	1	0011	10da	ffff	ffff	None
TSTFSZ	f, a	Test f, Skip if 0	1 (2 or 3)	0110	011a	ffff	ffff	None
XORWF	f, d, a	Exclusive OR WREG with f	1	0001	10da	ffff	ffff	Z, N
BIT-ORIENTED FILE REGISTER OPERATIONS								
BCF	f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None
BSF	f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None
BTFSC	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011	bbba	ffff	ffff	None
BTFSS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010	bbba	ffff	ffff	None

Mnemonic Opérands		Description	Cycles	16-bit instruction word MSB			LSB	Status affected
BTG	f, b, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None
CONTROL OPERATIONS								
BC	n	Branch if Carry	1 (2)	1110	0010	nnnn	nnnn	None
BN	n	Branch if Negative	1 (2)	1110	0110	nnnn	nnnn	None
BNC	n	Branch if Not Carry	1 (2)	1110	0011	nnnn	nnnn	None
BNN	n	Branch if Not Negative	1 (2)	1110	0111	nnnn	nnnn	None
BNOV	n	Branch if Not Overflow	1 (2)	1110	0101	nnnn	nnnn	None
BNZ	n	Branch if Not Zero	1 (2)	1110	0001	nnnn	nnnn	None
BOV	n	Branch if Overflow	1 (2)	1110	0100	nnnn	nnnn	None
BRA	n	Branch Unconditionally	2	1101	0nnn	nnnn	nnnn	None
BZ	n	Branch if Zero	1 (2)	1110	0000	nnnn	nnnn	None
CALL	n, s	Call subroutine 1st word	2	1110	110s	Kkkk	Kkkk	None
		2nd word		1111	Kkkk	Kkkk	Kkkk	
CLRWDT	--	Clear Wartchdag Timer	1	0000	0000	0000	0100	None
DAW	--	Decimal Adjust WREG	1	0000	0000	0000	0111	C
GOTO	n	Go to address 1st word	2	1110	1111	Kkkk	kkkk	None
		2nd word		1111	kkkk	Kkkk	kkkk	
NOP	--	No Operation	1	0000	0000	0000	0000	None
NOP	--	No Operation	1	1111	xxxx	xxxx	xxxx	None
POP	--	Pop top of return stack (TOS)	1	0000	0000	0000	0110	None
PUSH	--	Push top of return stack (TOS)	1	0000	0000	0000	0101	None
RCALL	n	Relative Call	2	1101	1nnn	nnnn	nnnn	None
RESET	--	Software device RESET	1	0000	0000	1111	1111	All
RETFIE	s	Return from interrupt enable	2	0000	0000	0001	000s	GIE/GEIH, GIEL
RETURN	s	Return From Subroutine	2	0000	0000	0001	001s	None
SLEEP	--	Go into standby mode	1	0000	0000	0000	0011	None
LITERAL OPERATIONS								
ADDLW	k	Add literal and WREG	1	0000	1111	kkkk	kkkk	C, DC, Z, OV, N
ANDLW	k	AND literal with WREG	1	0000	1011	kkkk	kkkk	Z, N
IORLW	k	Inclusive OR literal with WREG	1	0000	1001	kkkk	kkkk	Z, N
LFSR	f, k	Move literal (12 bit) 1st word	2	1110	1110	00ff	kkkk	None
		to FSRx 2nd word		1111	0000	kkkk	kkkk	
MOVLB	k	Move literal to BSR <3:0>	1	0000	0001	0000	kkkk	None
MOVLW	k	Move literal to WREG	1	0000	1110	kkkk	kkkk	None
MULLW	k	Multiply literal with WREG	1	0000	1101	kkkk	kkkk	None
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None
SUBLW	k	Subtract WREG from literal	1	0000	1000	kkkk	kkkk	C, DC, Z, OV, N
XORLW	k	Exclusive OR literal with WREG	1	0000	1010	kkkk	kkkk	Z, N
DATA MEMORY ← PROGRAM MEMORY OPERATIONS								
TBLRD*		Table read	2	0000	0000	0000	1000	None
TBLRD*+		Table read with post-increment	2	0000	0000	0000	1001	None
TBLRD*-		Table read with post-decrement	2	0000	0000	0000	1010	None
TBLRD+*		Table read with pre-increment	2	0000	0000	0000	1011	None

Table 2.5: Le jeu d'instructions du MCIP

Format de l'instruction

La *figure 2.12* montre les principaux formats d'instructions. Comme on peut le constater, la partie ***OPCODE*** n'utilise que 4 à 8 bits permettant ainsi d'optimiser le jeu d'instruction.

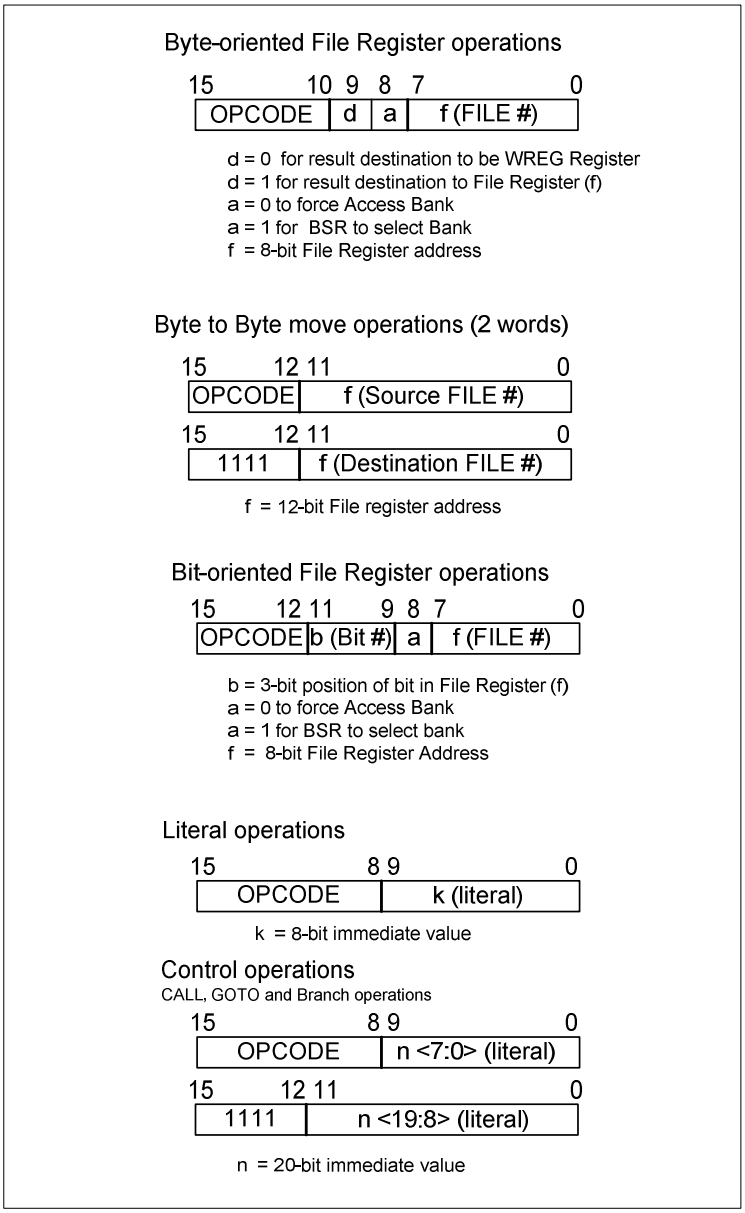


Figure 2.12: Les formats d'instruction

2.3. CONCLUSION

Les différents éléments composant l'architecture de notre microcontrôleur MCIP tirée de celle du PIC18 de *Microchip* sont explicités dans ce chapitre. La conception du MCIP à l'aide d'une plateforme VHDL de *Xilinx* sur un FPGA sera présentée dans le prochain chapitre.

Chapitre 3

Conception du MCIP

3.1. METHODOLOGIE ET APPROCHE SUIVIES

Le projet de développement du **MCIP** englobe des parties logicielles et matérielles comme l'illustre la *figure 3.1*. La première partie est la spécification de l'architecture du microcontrôleur **MCIP** (exposée dans le *chapitre 2*), la suite du flot de conception de l'**IP** est composée de plusieurs étapes et fait appel à de nombreux outils de développement garantissant simplicité, modularité et efficacité d'accomplissement. Notre démarche de conception est basée essentiellement sur deux éléments clés : Une maîtrise d'un outil de conception et de développement et une maîtrise d'une architecture de microcontrôleur.

Le flot de conception (*figure 3.1*) comporte les étapes suivantes :

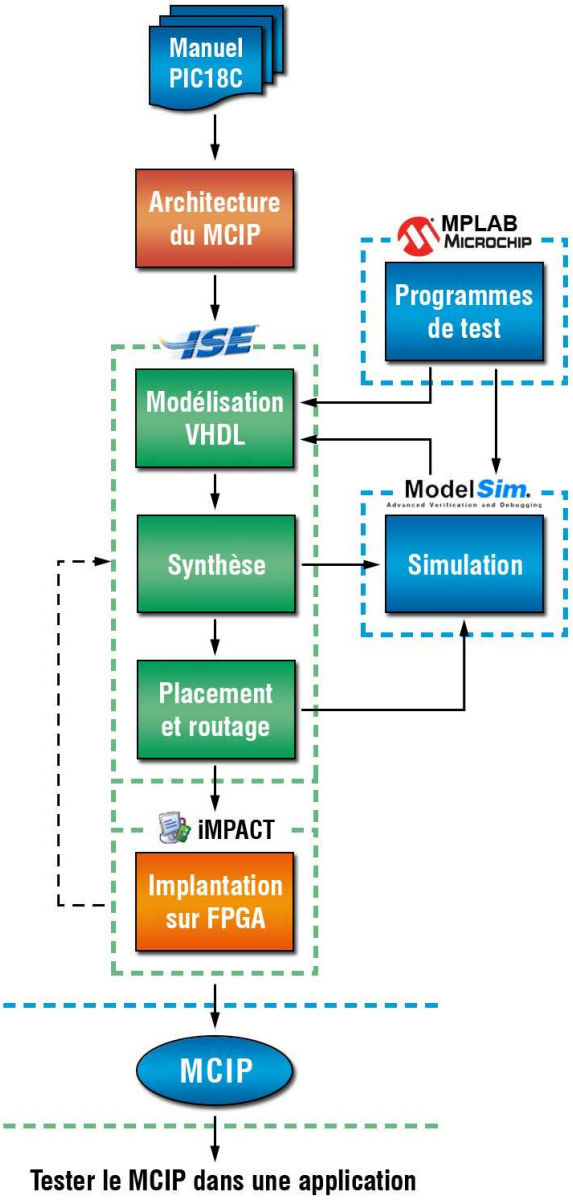


Figure 3.1: Flot de conception

1. Elaboration du modèle VHDL du **MCIP** et synthèse en utilisant l'environnement de développement **Xilinx ISE 9.1**.
2. Validation et vérification fonctionnelle en utilisant le simulateur (VHDL) **ModelSim** en conjonction avec l'environnement de développement des microcontrôleurs **MPLAB** de *Microchip*.
3. Après la vérification fonctionnelle, on exécute une simulation temporelle après phase de placement & routage.
4. Une fois le code du **MCIP** est élaboré et vérifié, on procède à l'implantation du circuit conçu sur FPGA (dans notre cas sur une Spartan-3E de *Xilinx*) pour compléter sa validation. Dans cette implantation notre **MCIP** exécutera un programme simple d'affichage d'un texte sur LCD.

3.2. LES OUTILS DE DEVELOPPEMENT

3.2.1 L'environnement de développement Xilinx ISE 9.1

XILINX ISE se présente comme l'un des logiciels CAO les plus utilisés pour la mise en œuvre de circuits logiques programmables sur FPGA et CPLD [28].

L'environnement de développement **Xilinx ISE 9.1** permet la conception à haut niveau en utilisant les langages VHDL ou Verilog. Il offre une multitude d'outils et moyens de programmation très souples qui permettent au développeur de trouver des solutions aux contraintes et spécifications du circuit cible en le soumettant aux différents environnements d'expérimentation possibles. **Xilinx ISE** intègre ainsi les outils nécessaires pour accomplir les tâches de développement d'un circuit sur FPGA (description, synthèse, édition de banc d'essai et simulation, programmation) (*figure 3.2*). **Xilinx ISE** offre aussi la possibilité de décrire le banc d'essai (*testbench*) sous forme de chronogrammes (*waveform*), et se charge aussi de le traduire en code VHDL.

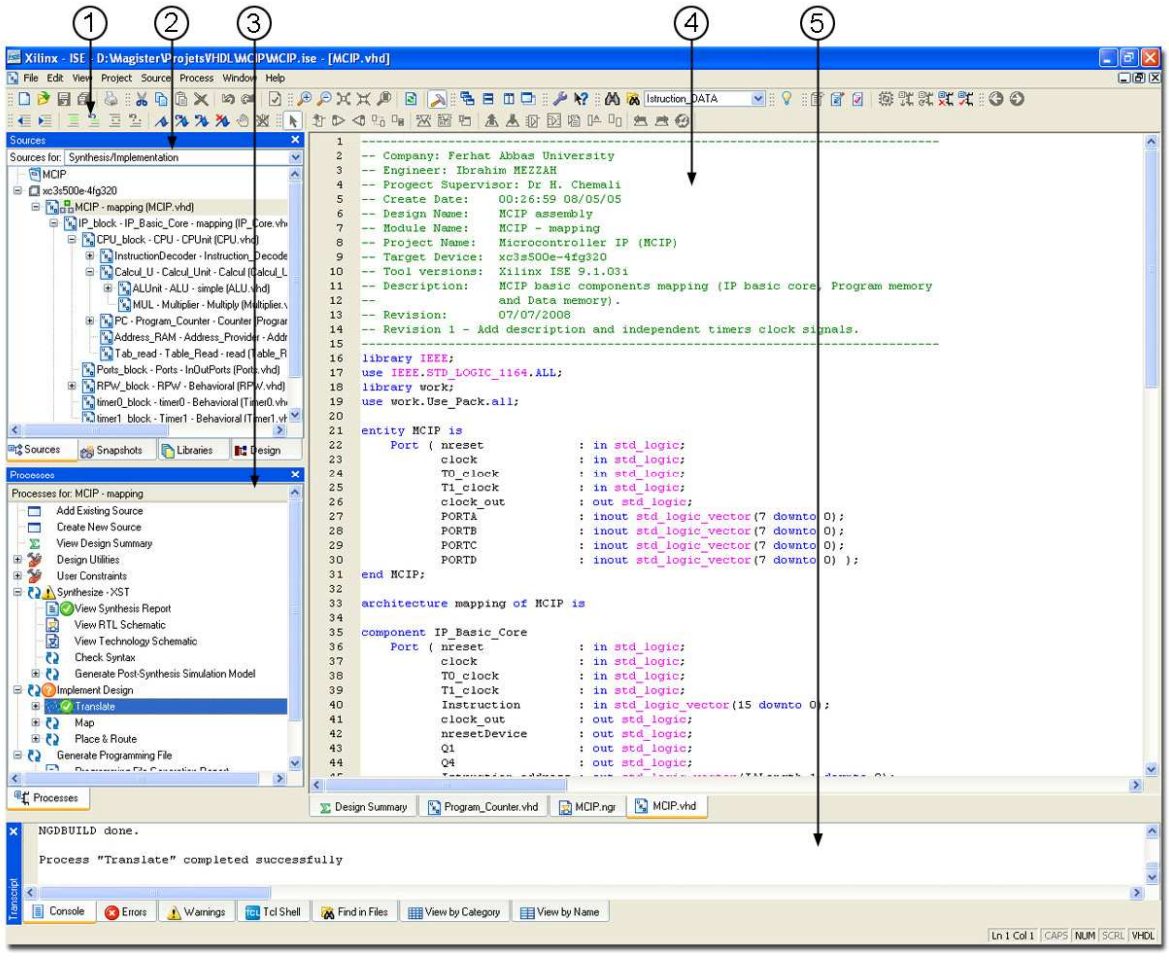


Figure 3.2: L'interface du Xilinx ISE 9.1

L'organisation du navigateur de projet **Xilinx ISE 9.1** est la suivante (*figure 3.2*) :

1- La barre d'outils

2- La fenêtre des sources (modules) qui composent le projet

3- La fenêtre des processus de développement

4- L'espace de travail qui affiche le contenu des différentes sources en exploitation

5- La fenêtre de transcription qui affiche l'évolution des processus.

3.2.2 Le simulateur ModelSim XE III/Starter 6.3c

Le simulateur **ModelSim** de *Mentor Graphics* permet de vérifier la fonctionnalité et le timing du design ou une partie du design (*figure 3.3*) [28] tout en faisant appel à partir de **Xilinx ISE**. Le **ModelSim** interprète le code VHDL (ou Verilog), exécute les testbenchs et affiche des résultats binaires sujets à des analyses complémentaires. (La version utilisée dans ce travail est limitée en performance : *restricted version*).

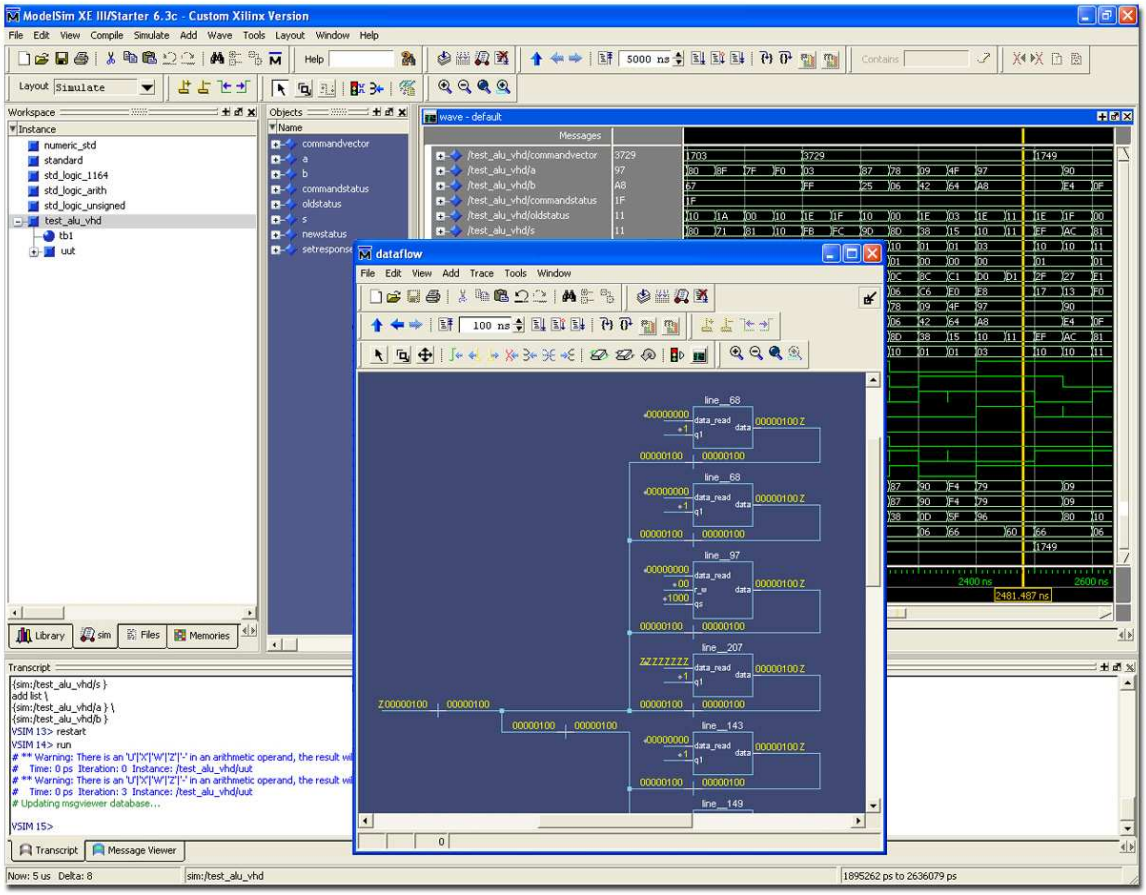


Figure 3.3: L'interface du simulateur ModelSim de *Mentor Graphics*

3.2.3 L'environnement de développement MPLAB IDE v8.10

Microchip offre un grand nombre d'outils de soutien pour faciliter le développement d'une application autour de ses microcontrôleurs [25]. Ces outils sont intégrés dans un même environnement dénommé **MPLAB** (*figure 3.4*) et permettent d'effectuer les opérations suivantes :

1. Editer les fichiers sources en utilisant plusieurs langages. Le **MPLAB** supporte les deux langages: - L'assembleur (**MPASM)**,
   - Le langage C (**MPLAB C18)**.
2. Simuler et déboguer le programme en utilisant **MPLAB-SIM**. Pendant une exécution pas à pas, l'utilisateur peut examiner et modifier n'importe qu'elle donnée ou définir des stimuli. Il permet aussi de suivre l'évolution de l'exécution du programme en examinant les différentes zones de données (Trace). Les informations examinées peuvent être exportées sous forme de fichiers textes.
3. Emuler le circuit en utilisant **MPLAB-ICE**.
4. Programmation de microcontrôleurs.

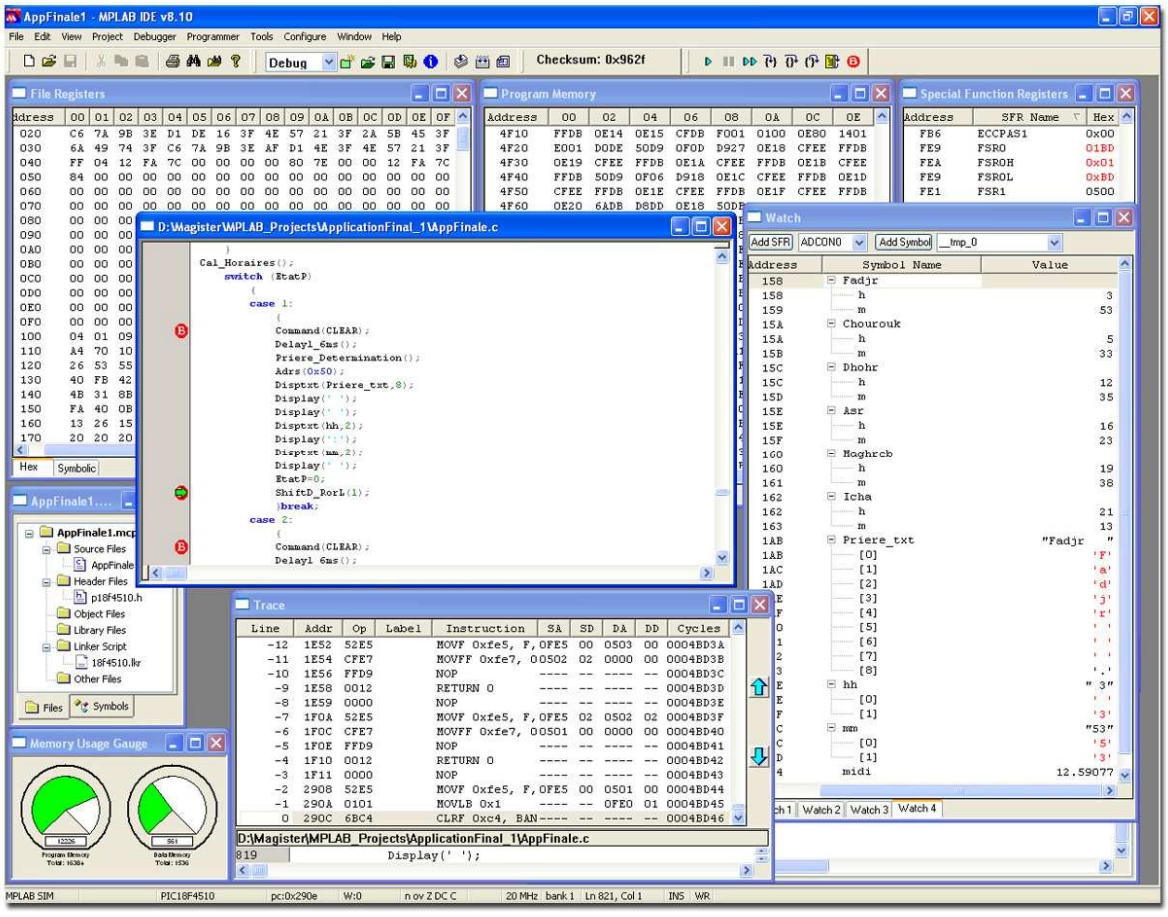


Figure 3.4: L'environnement de développement MPLAB IDE

3.3. ELABORATION DU MODELE VHDL

La *figure 3.5* présente l'hiérarchie du modèle VHDL de MCIP étudié. Après une étude exhaustive des architectures de processeurs et microcontrôleurs et une multitude d'essais, nous avons pu définir une structure soignée qui produit un code structuré, modulaire et lisible.

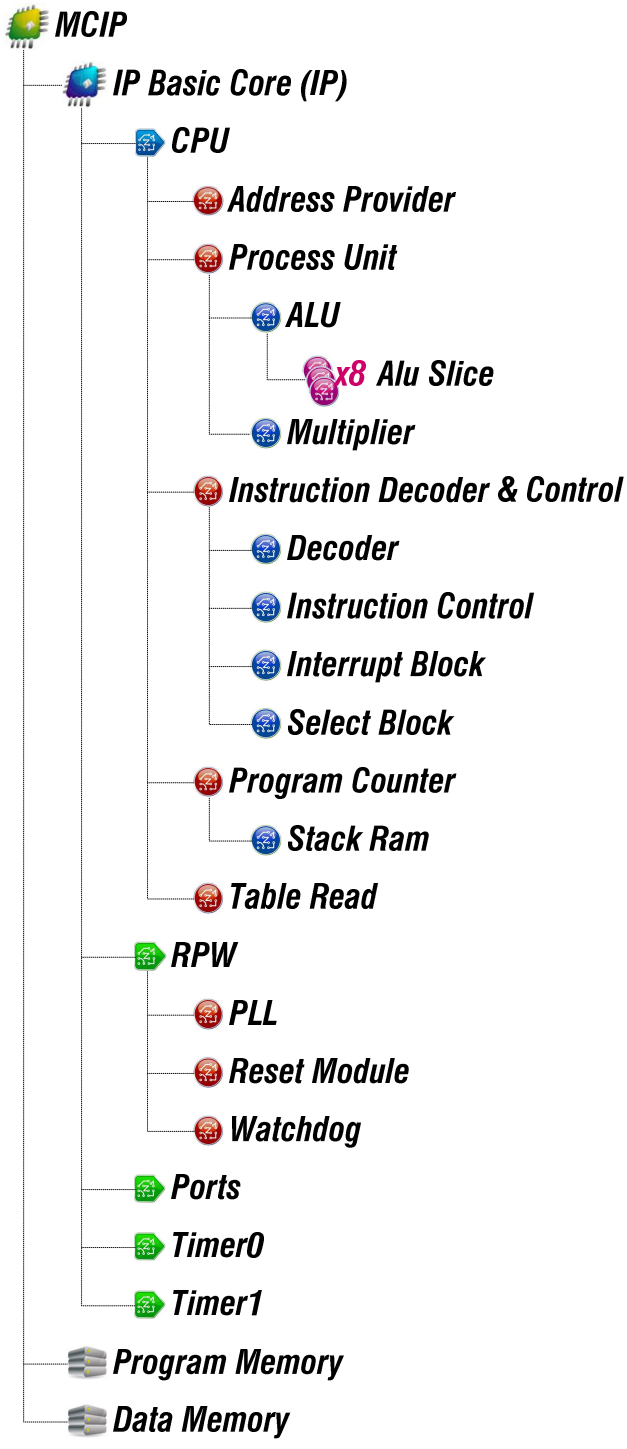


Figure 3.5: Hiérarchie du modèle VHDL de MCIP

La décomposition du modèle MCIP en sous modules présentés à la *figure 3.5*, découle directement de l'analyse de son architecture présentée au *chapitre 2*. Certaines fonctions indépendantes sont facilement repérables (timers, ports, Watchdog, ALU …etc) tandis que d'autres nous imposent un partitionnement réfléchi tels que les sous modules de décodage et de contrôle. Cette modularité est exigée dans le cahier des charges et assure une grande flexibilité de développement. L'hiérarchie du MCIP montrée à la *figure 3.5* est effectivement obtenue à la fin de notre conception, mais beaucoup de restructurations ont été nécessaires au cours de l'évolution et de la maitrise des routines de programmation. L'ajout de quelques modules et le regroupement de certains étaient nécessaires pour optimiser la structure finale du MCIP.

La structure du MCIP est subdivisée en trois parties principales :

1- le noyau (***IP Basic Core***) qui englobe tous les éléments fonctionnels du microcontrôleur: Le **CPU**, l'ensemble **RPW**, les **ports** et les **timers**.

L'**IP Basic Core** est défini comme une entité configurable pour gérer des mémoires de programme de différentes tailles.

2- ***Program Memory***.

3- ***Data Memory***.

Ces deux mémoires (Data, Program) sont définies selon des structures régulières et simples. L'**IP Basic Core,** étant la composante fondamentale, est étudié sans l'intégration des mémoires pour offrir un maximum de flexibilité lors de son exploitation dans les SoC. Il constitue en conséquence l'IP conçu que nous avons nommé MCIP. Il est laissé le soin à l'utilisateur de configurer uniquement ce noyau pour prendre en charge l'application chargée dans une mémoire séparée (mémoire de programme standard) (*figure 3.6*).

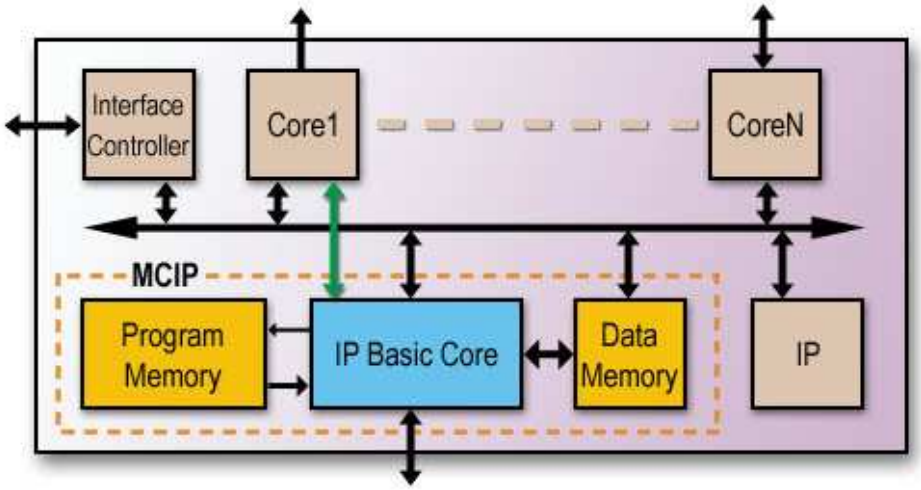


Figure 3.6: MCIP dans un SoC

En ce qui concerne la partie programmation, une bonne spécification produit généralement un bon impact sur le code de MCIP. Une fois tous les éléments de l'architecture reconnus et explicités, la transposition de leurs fonctions en code VHDL et la validation des éléments ainsi obtenus sont régies par des règles générales de conception en VHDL. Le code VHDL du modèle de MCIP est constitué des éléments suivants :

1- Un paquetage (Use_Pack),

2- Les modules du MCIP (CPU, Timer0, PLL ...etc),

3- Les testbenchs des différents modules (tb_ALU, tb_.....etc).

3.3.1 Le paquetage (Use_Pack)

Ce paquetage est utilisé pour déclarer des sous types et des constantes spécifiques qui seront utilisées dans tout le modèle (*listing 1*).

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

package Use_Pack is

 constant DALength : integer := 12;
 constant Ram_size : integer := 2**(DALength); --Data Address Length

 constant IALength : integer := 15;
 constant Rom_size : integer := 2**(IALength); --Inst. Address Length

 constant enable  : std_logic := '1';
 constant disable : std_logic := '0';

 constant Enabled  : std_logic := '1';
 constant Disabled : std_logic := '0';

 subtype Command_vector_cu_type is std_logic_vector(13 downto 0);
 subtype Command_status_type is std_logic_vector(4 downto 0);
 subtype Command_vector_pc_type is std_logic_vector(6 downto 0);

end Use_Pack;
```

Listing 1: Le paquetage Use_pack

3.3.2 Les testbenchs

La modularité du modèle a énormément simplifié l'opération de vérification. Les modules sont testés au fur et à mesure du développement. Les testbenchs des différents modules sont élaborés de deux manières complémentaires:

1- Testbenchs waveform utilisés souvent à cause de la facilité et la rapidité de leur production (*figure 3.*7). Puisque le simulateur exécute les testbenchs écrits aussi en

VHDL, ISE se charge de convertir le testbench weveform en testbench VHDL avant le lancement de la simulation.

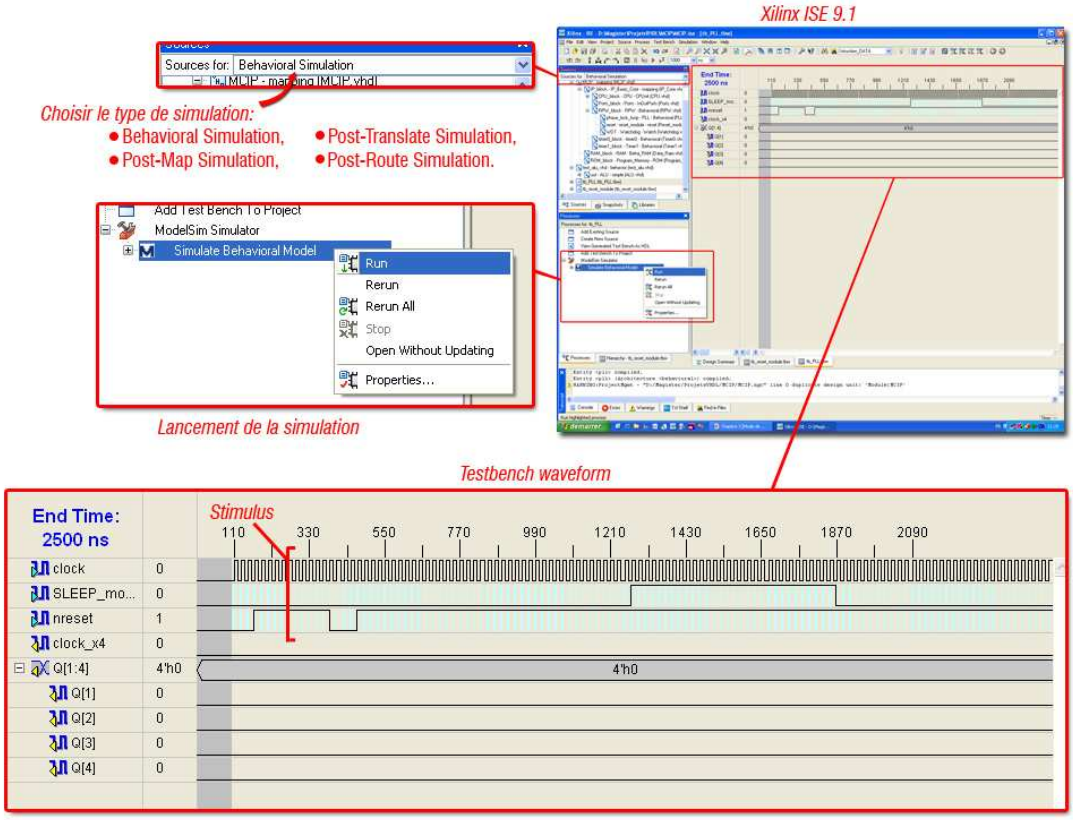


Figure 3.7: Exemple de Testbench waveform

2- Testbenchs écrits en VHDL sont chargés de comparer les résultats de la simulation avec ceux attendus et permettent d'afficher des messages qui signaleraient des erreurs et une aide à leur localisation. Un exemple de testbench est présenté en *annexe 3* (testbench de l'ALU2).

3.4. CONCEPTION ET VALIDATION DES MODULES DE L'IP BASIC CORE

Le choix d'exécuter une instruction (un cycle) en 4 périodes d'horloge moyennant les signaux ***Q1***, ***Q2***, ***Q3*** et ***Q4*** de la **PLL** facilite énormément la conception et réduit le risque d'exécution incorrecte. Une instruction qui est en rapport avec l'**ALU** s'exécute de la manière suivante :

Q1 : Décodage de l'instruction et lecture de la donnée.

Q2 : Chargement de la donnée dans l'**ALU** (dans **FREG**) au front montant de ***Q2*** et exécuter l'opération pendant ***Q2***.

Q3 : Placer le résultat dans le bus de données.

Q4 : Chargement du résultat dans le registre cible au front montant de ***Q4***.

Tous les registres **SFRs** et **GPRs** du MCIP sont synchronisés par le front montant de ***Q4***. Les données sont présentées sur le bus de données (si c'est nécessaire) pendant ***Q1*** (lecture) et ***Q3*** (écriture) et le bus est maintenu en haute impédance ('**Z**') ailleurs.

Nous exposons dans ce qui suit, les différents modules conçus et développés et leur expérimentation en analysant certains résultats de simulation.

3.4.1 Module RPW

Conformément à la structure développée, le module **RPW** est constitué des sous modules **Reset**, **PLL**, et **Watchdog**.

3.4.1.1 La PLL

Tout le timing de l'architecture repose sur les ***Qi*** générés par le module **PLL**, il est donc impératif de veiller scrupuleusement à l'aspect des signaux de ce module. La **PLL**, dont l'entité est donnée à la *figure 3.8*, est conçue de telle manière que les temps de maintien pour la validation des données soient prévus: Par exemple, une donnée est mise sur le bus de donnée pendant ***Q3***, son affectation à une destination donnée se fait en ***Q4***, cela veut dire q'un temps de maintien de données doit être prévu pour ne pas perdre cette donnée. Cela est réalisé en autorisant le front descendant de ***Q3*** après le front montant de ***Q4*** (*annexe 4*), et c'est idem pour ***Q1*** et ***Q2***.

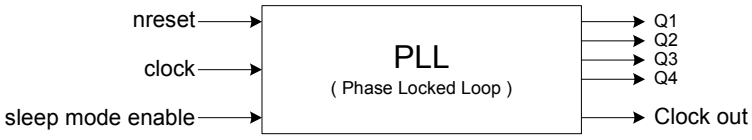


Figure 3.8: Entité PLL

On note aussi que le mode **SLEEP** est en relation directe avec le module **PLL**. En effet, la mise en veille du MCIP revient à arrêter la génération des signaux ***Qi*** qui fait fonctionner l'architecture : suite au décodage de l'instruction **SLEEP**, le signal ***sleep_mode_enable*** est activé au début de ***Q1***, la **PLL** continue à générer les signaux ***Qi*** puis s'arrête après ***Q4*** permettant ainsi la recherche d'une nouvelle instruction. La **PLL** revient en marche après désactivation du signal ***sleep_mode_enable*** grâce à un signal ***wake-up***.

La vérification de ce module est réalisée avec un testbench de type waveform. Une partie des résultats de la simulation fonctionnelle de la **PLL** fournie par **ModelSim** est donnée à la *figure 3.9* montrant l'activité de la **PLL** en mode normal et son comportement en mode **SLEEP**. La simulation après placement/routage est particulièrement importante pour

ce module : la *figure 3.10* montre comment le front montant de ***Q2*** se produit après un délai de *1.6 ns* du front descendant de ***Q1***, et le front montant de ***Q4*** se produit après *1.5 ns* du front descendant de ***Q3***. Ces délais dépendent directement de l'implémentation physique.

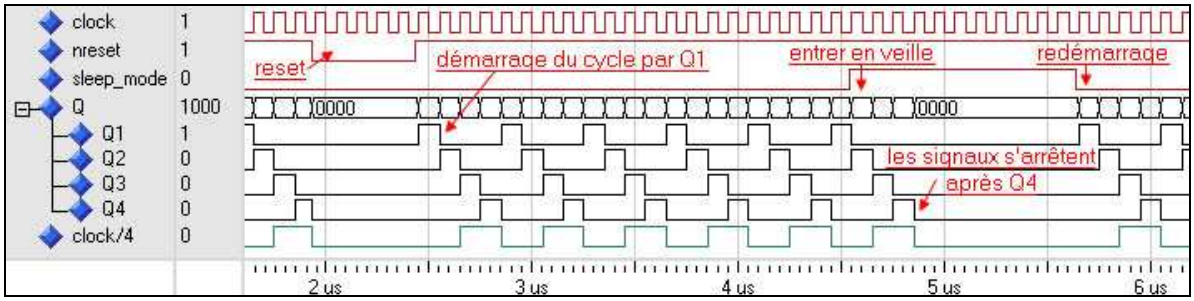


Figure 3.9: Résultats de la simulation fonctionnelle de la PLL

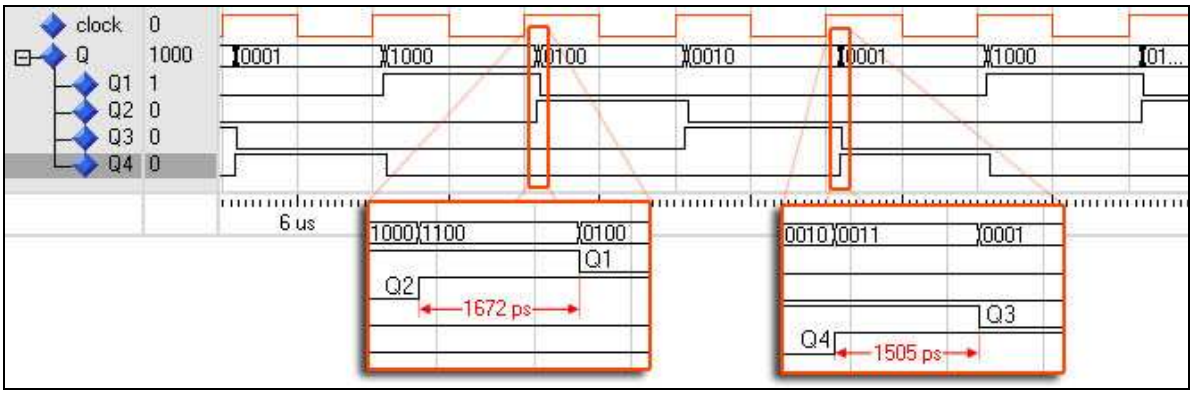


Figure 3.10: Simulation après placement et routage de la PLL

3.4.1.2 Le Module Reset

Ce module génère le reset global du système. Son entité illustrée à la *figure 3.11*, présente les différentes sources d'initialisation. Le signal ***nreset device*** est activé ('0') suite à une activation de l'un des signaux d'entrées. Dans le cas d'un reset "interne" (***soft reset, stack overflow, WDT time out***) la durée de l'activation de ***nreset*** est égale à 4 cycles d'horloge pour garantir l'initialisation du système.

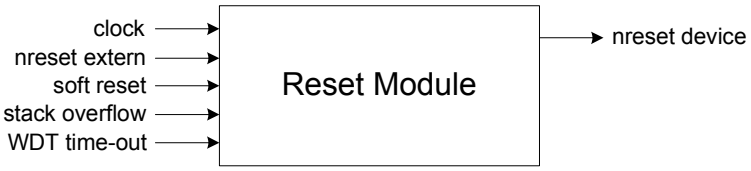


Figure 3.11: Entité Reset Module

Ce module est validé à l'aide d'un testbench de type waveform où il est montré une activation effective du reset conformément aux exigences et conditions planifiées.

3.4.1.3 Le Watchdog

Le programme VHDL du Watchdog est donné en *annexe 4*. Sa vérification fonctionnelle est réalisée avec deux testbenchs :

- le premier sert à vérifier les différentes fonctionnalités du module (*figure 3.13.a*).
- le deuxième "long" testbench est établi pour atteindre l'expiration du temps et observer la signalisation de la sortie ***wdt_time_out*** (*figure 3.13.b*), pour différentes valeurs de ***WDTPS***.

Figure 3.12: Entité Watchdog

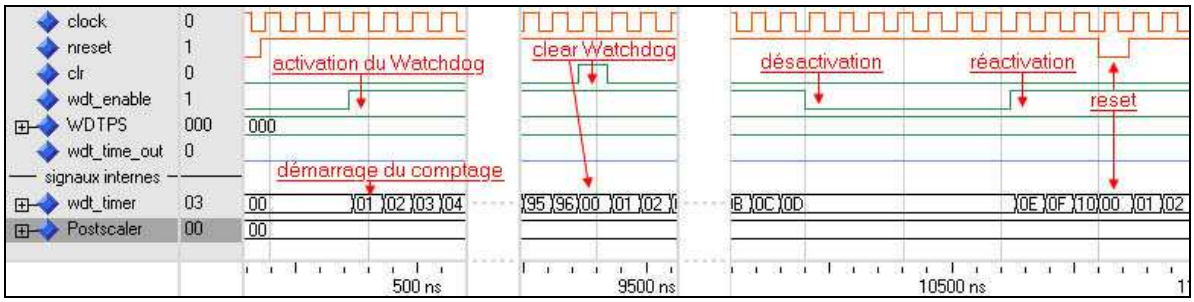


Figure 3.13.a: Simulation du Watchdog: testbench1

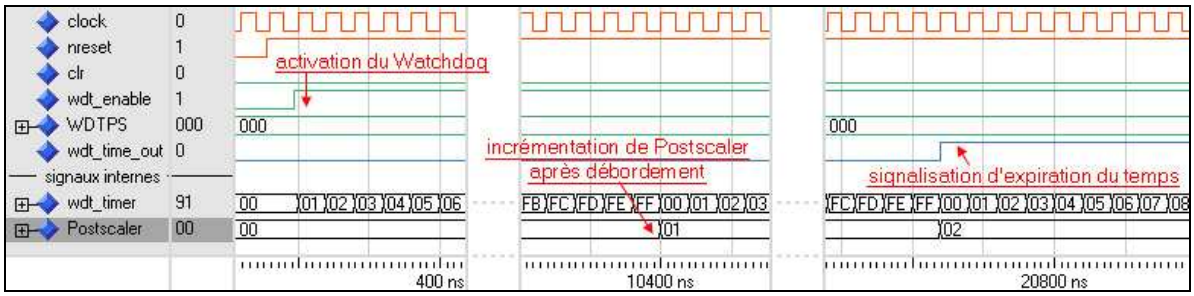


Figure 3.13.b: Simulation du Watchdog: testbench2

3.4.1.4 Assemblage de RPW

Les 3 sous-modules décrits ci-dessus sont assemblés sous la même entité **RPW**. Le registre de contrôle du Watchdog **WDTCON** est intégré dans ce module. Les résultats de simulation qui valident le fonctionnement de l'ensemble sont présentés sur la *figure 3.1*5.

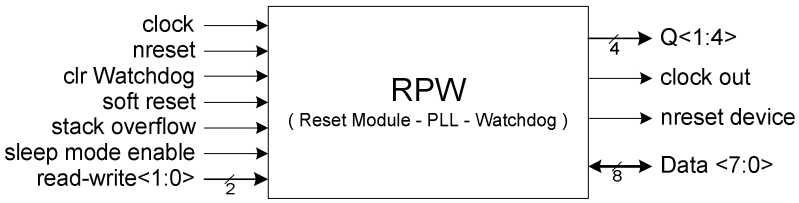


Figure 3.14: Entité RPW

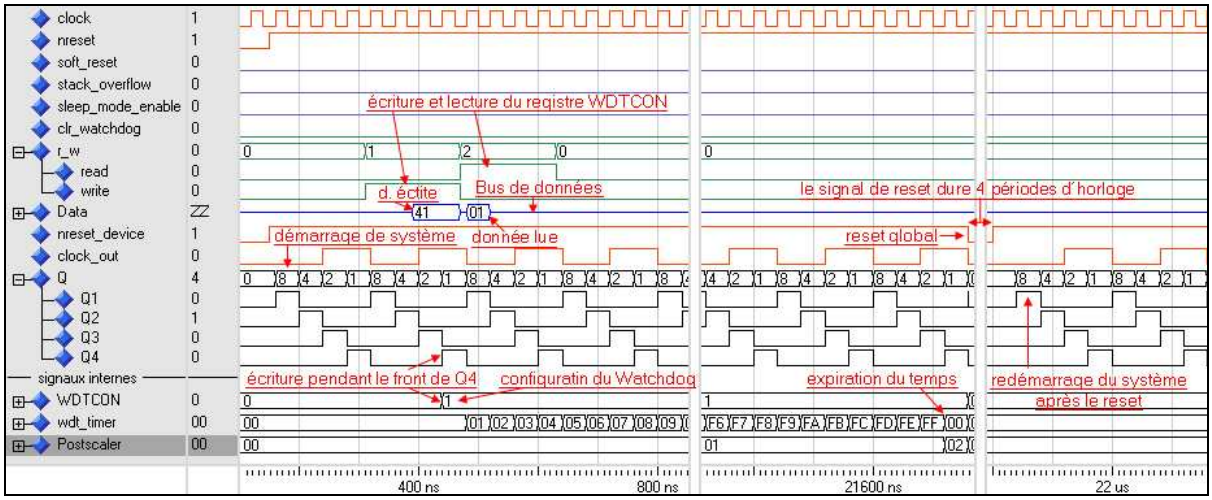


Figure 3.15: Résultats de simulation RPW

3.4.2 Le module des Ports

Le module des ports est un groupement de 4 ports d'E/S. Au niveau de la programmation, il s'agit de duplication d'un même **process** donné par le *listing 2*. Le **PORTB** est caractérisé par ses entrées d'interruptions comme indiquées par les résultats de simulation.

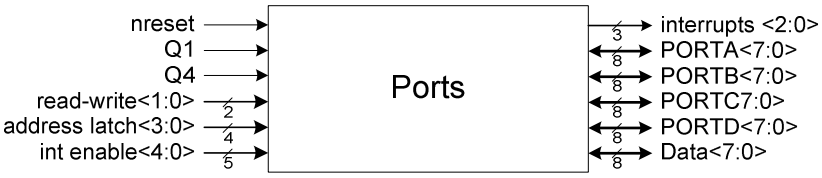


Figure 3.16: Entité Ports

```
PORTA : process(TRISA, LATA)
          begin
            for i in 0 to 7 loop
                  if TRISA(i) = '0' then
                  PORTA(i) <= LATA(i);
                  else
                        PORTA(i) <= 'Z';
                  end if;
            end loop;
   end process;
```

Listing 2: Process de modélisation de port d'E/S

La vérification de ce module revient à écrire et à lire les différents registres des ports et à vérifier les interruptions (*figure 3.17*).

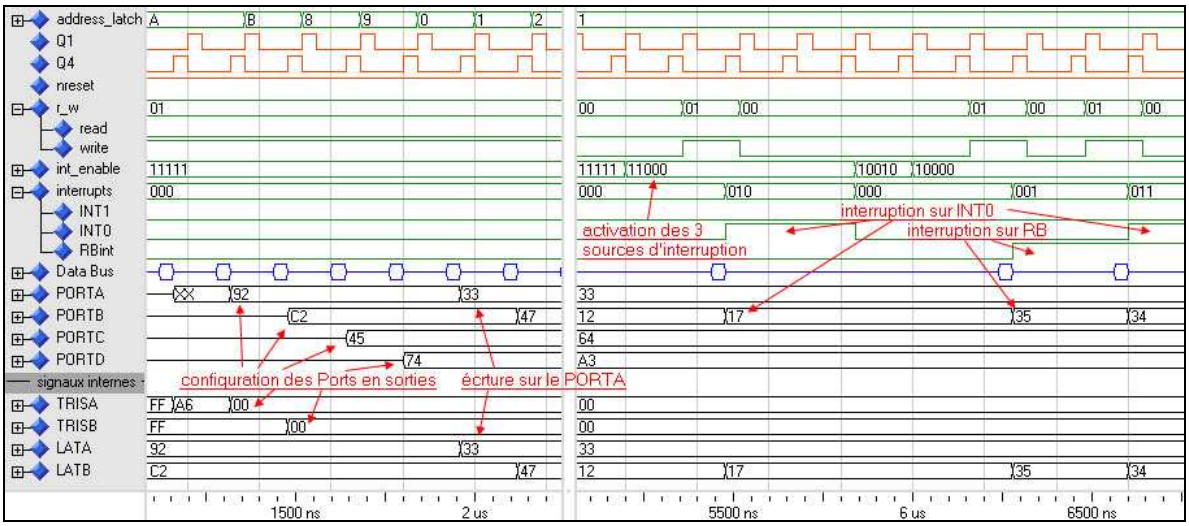


Figure 3.17: Résultats de simulation de Ports

3.4.3 Modules Timer0 et Timer1

Ces deux modules identiques sont conçus selon la spécification fournie au *chapitre 2*. Chaque Timer intègre ses propres **SFRs**; le Timer0 contient les registres **TMR0**, **TMR0H** et **T0CON**. La *figure 3.18* présente quelques résultats de simulation de ce Timer. Le programme VHDL est donné en *annexe 4*.

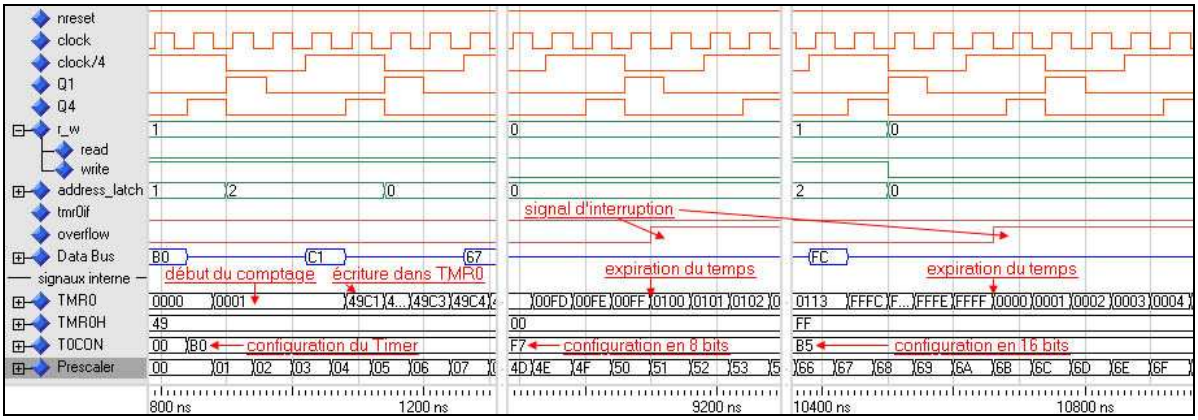


Figure 3.18: Résultats de simulation de Timer0

3.4.4 Module CPU

3.4.4.1 Unité de calcul

La structure générale de ce module est montrée sur la *figure 3.19*. C'est un bloc organisationnel qui comprend l'unité arithmétique et logique, le multiplicateur et les différents registres de travail.

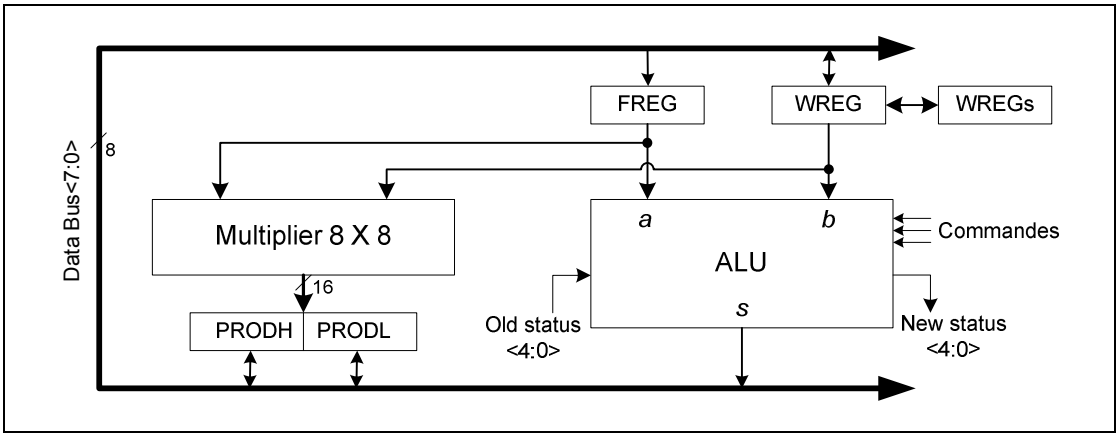


Figure 3.19: Structure générale de Process Unit

A/ Module ALU

Vu l'intérêt de ce module, et son impact sur le fonctionnement général, plusieurs études sont élaborées dont deux architectures sont consignées ci-dessous.

Architecture 1

La première architecture de l'**ALU** a été construite autour d'un partitionnement du circuit selon le type d'opération. Ainsi nous avons décomposé le circuit en deux blocs fonctionnels (*figure 3.20*) :

1- **Bloc 1**: exécute les opérations arithmétiques (*figure 3.21.a*).

2- **Bloc 2**: exécute les opérations logiques, de rotations et opérations 'bit oriented' (*figure 3.21.b*).

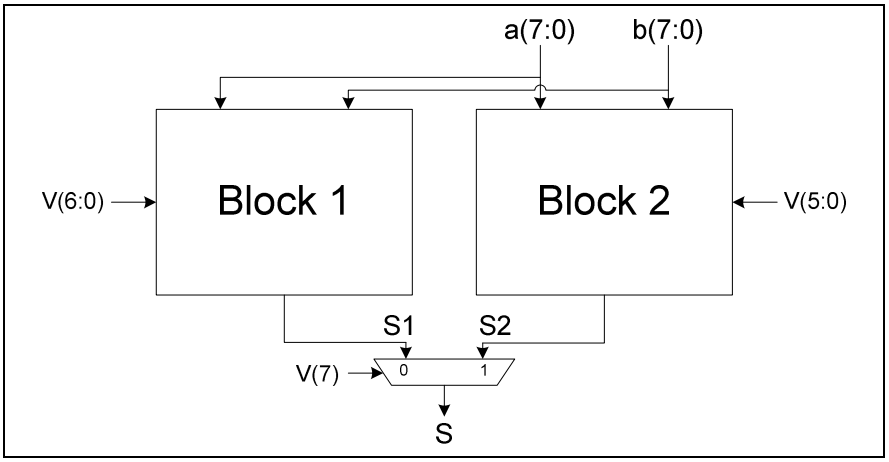


Figure 3.20: La structure de l'ALU1

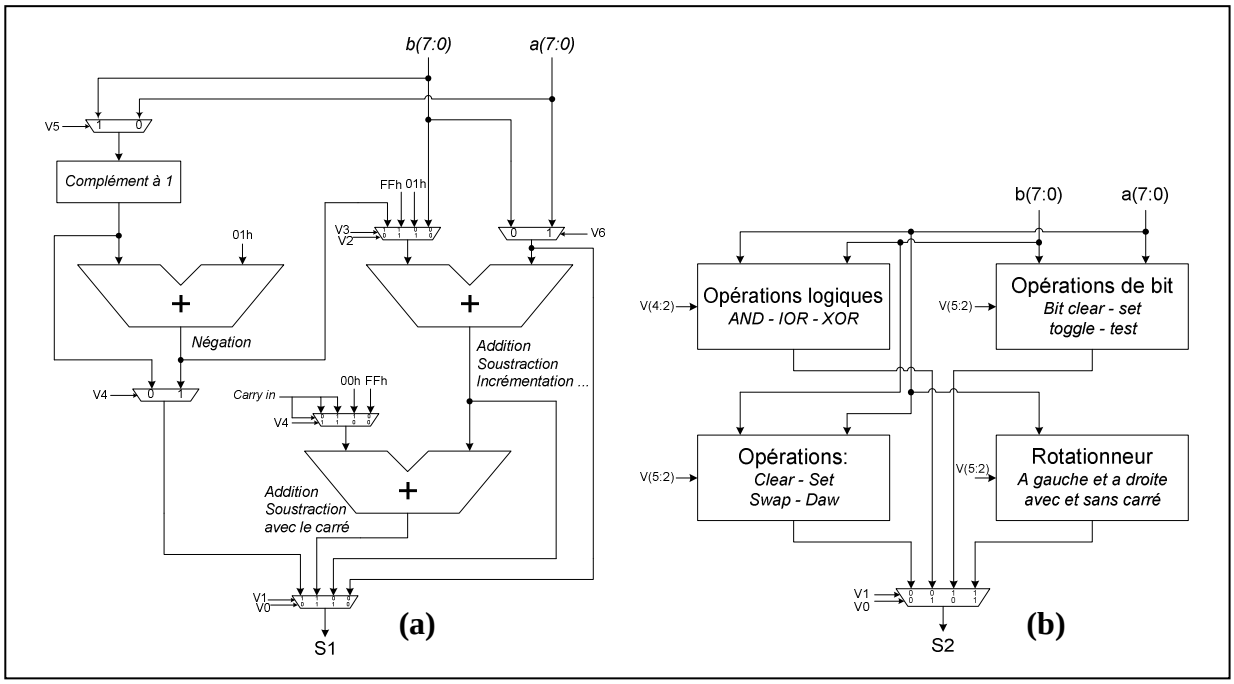


Figure 3.21: Structure détaillée de l'ALU1

Suite à cette structure, la définition des vecteurs de commandes ***V<7:0>*** selon l'opération à exécuter est donnée par la *table 3.2*. Cette table indique aussi les bits d'état affectés pour chaque opération.

Opération	Command_Vector V<7:0>							Command Status N	OV	Z	DC	C
Opérations arithmétiques												
ADD a,b	0	1	0	0	0	0	1	1	1	1	1	1
ADDC a,b	0	1	0	1	0	1	1	1	1	1	1	1
COM a	0	0	0	0	0	1	0	1	0	1	0	0
CP a,b	0	1	1	0	1	0	1	0	0	0	0	0
DEC a	0	1	0	0	1	0	1	1	1	1	1	1
INC a	0	1	0	0	0	0	1	1	1	1	1	1
MOV a	0	1	0	0	0	0	0	1	0	1	0	0
MOV b	0	0	0	0	0	0	0	0	0	0	0	0
NEG a	0	0	0	1	0	1	0	1	1	1	1	1
SUBB a,b	0	0	0	0	1	1	1	1	1	1	1	1
SUB b,a	0	1	1	0	1	0	1	1	1	1	1	1
SUBB b,a	0	1	1	0	1	1	1	1	1	1	1	1
TST a	0	1	0	0	0	0	0	0	0	0	0	0
Opérations logiques												
AND a,b	1	0	0	0	0	1	0	1	0	1	0	0
IOR a,b	1	0	0	0	1	0	0	1	0	1	0	0
XOR a,b	1	0	0	1	0	0	0	1	0	1	0	0

Opération	Command_Vector V<7:0>							Command Status N	OV	Z	DC	C
Opérations de rotation												
RLC a	1	0	0	0	0	1	1	1	0	1	0	1
RLNC a	1	0	0	0	1	0	1	1	0	1	0	0
RRC a	1	0	0	1	0	0	1	1	0	1	0	1
RRNC a	1	0	1	0	0	0	1	1	0	1	0	0
Opérations sur bit												
BT a	1	0	0	0	0	1	0	0	0	0	0	0
BC a	1	0	0	0	1	0	0	0	0	0	0	0
BS a	1	0	0	1	0	0	0	0	0	0	0	0
BTG a	1	0	1	0	0	0	0	0	0	0	0	0
Autres												
CLR a	1	0	0	0	0	1	1	0	0	1	0	0
SET a	1	0	0	0	1	0	1	0	0	0	0	0
DAW b	1	0	0	1	0	0	1	0	0	0	0	1
SWAP a	1	0	1	0	0	0	1	0	0	0	0	0

Table 3.2: Vecteurs de commande d'opérations

L'explication narrative du code de ce module est complexe. Cette structure nécessite la déclaration d'un grand nombre de signaux et l'élaboration de plusieurs **process** s'accompagnant d'un effort supplémentaire relatif à la vérification.

La vérification consiste à expérimenter l'**ALU** en exécutant toutes les opérations pour différents types d'opérandes soigneusement choisis pour réussir une bonne couverture de test**.** Ces opérandes sont décrits avec un testbench waveform. Certains résultats de simulation sont présentés à la *figure 3.22*.

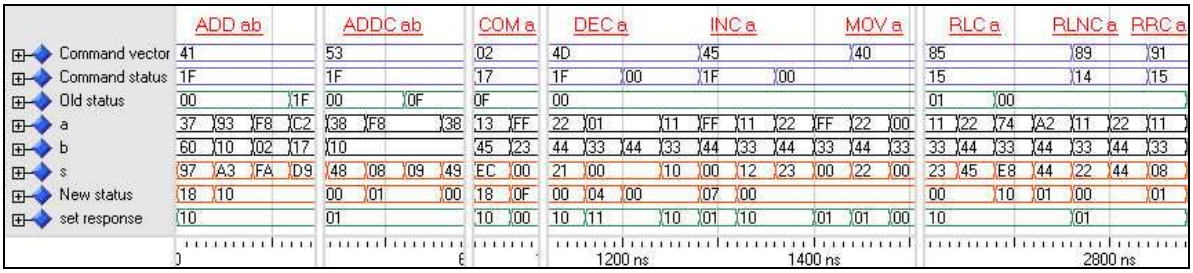


Figure 3.22: Résultats de simulation de l'ALU1

Le rapport de synthèse de ce module (*annexe 5*) indique que le nombre de slices utilisées (sur la XC3S500E Spartan-3E) pour l'implantation du module est de *112* et que le délai du chemin combinatoire maximal est de *16,35 ns*. Ces deux paramètres sont considérés lors de la comparaison et sélection de l'architecture de l'**ALU** à implanter dans le projet final.

Architecture 2

La deuxième architecture de l'**ALU** est axée autour d'un chemin de propagation du 'carry' [7]. Cette structure est totalement régulière et constituée de tranches absolument identiques.

La structure d'une tranche est très simple (*figure 3.23*) : elle est constituée tout d'abord de deux multiplexeurs à quatre entrées chacun. Leurs entrées ***g0*** à ***g3*** et ***p0*** à ***p3*** sont des bits de configuration de la fonction (type d'opération à effectuer). La valeur de ces signaux est identique pour toutes les tranches de l'**ALU**. Les signaux ***g*** se rapportent à la partie de génération du résultat, et les signaux ***p*** se rapportent à la partie de propagation du carry.

Le *tableau 3.3* donne la liste des opérations logiques (exhaustive) et arithmétiques qui sont offertes par cette structure. Pour former une **ALU** de N bits, il suffit d'interconnecter N tranches où les carry_out sont connectés aux carry_in des tranches du niveau supérieur (*figure 3.24*).

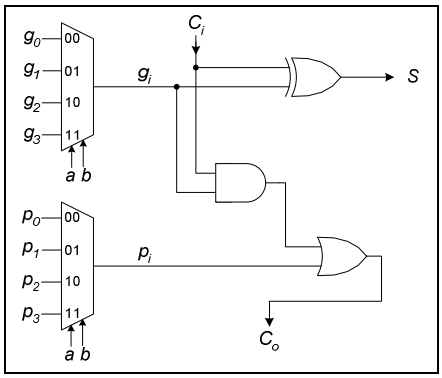


Figure 3.23: Tranche 1 bit de l'ALU

P3	P2	P1	P0	g3	g2	g1	g0	Opération
0	0	0	0	0	0	0	0	S=0
0	0	0	0	0	0	0	1	S=a nor b
0	0	0	0	0	0	1	1	Complément à 1 de a: S=not a (C_i =o) Complément à 2 de a: S=-a (C_i =1)
0	0	0	0	0	0	1	0	S=(not a) and b
0	0	0	0	0	1	0	1	Complément à 1 de b: S=not b (C_i =o) Complément à 2 de b: S=-b (C_i =1)
0	0	0	0	0	1	0	0	S=a and (not b)
0	0	0	0	0	1	1	0	S=a xor b
0	0	0	0	0	1	1	1	S=a nand b
0	0	0	0	1	0	0	0	S= a and b
0	0	0	0	1	0	0	1	S= nxor b
0	0	0	0	1	0	1	0	S=b ou incrémentation de b: S=b+1 (C_i =1)
0	0	0	0	1	0	1	1	S=(not a) or b
0	0	0	0	1	1	0	0	S= a ou incrémentation de a : S=a+1 (C_i =1)
0	0	0	0	1	1	0	1	S=a or (not b)
0	0	0	0	1	1	1	0	S=a or b
0	0	0	0	1	1	1	1	S=1
1	0	0	0	0	1	1	0	Addition: S=a+b+ C_i
0	1	0	0	1	0	0	1	Soustraction: S=a-b-(1-C_i)
0	0	1	0	1	0	0	1	Soustraction: S=b-a-(1-C_i)
1	1	0	0	0	0	1	1	Décrémentation: S=a-(1-C_i)
1	0	1	0	0	1	0	1	Décrémentation: S=b(1-C_i)
1	1	0	0	0	0	0	0	Décalage à gauche: S=C_i, C_o=a
1	0	1	0	0	0	0	0	Décalage à gauche: S=C_i, C_o=b

Table 3.3: Les opérations arithmétiques et logiques possibles [7]

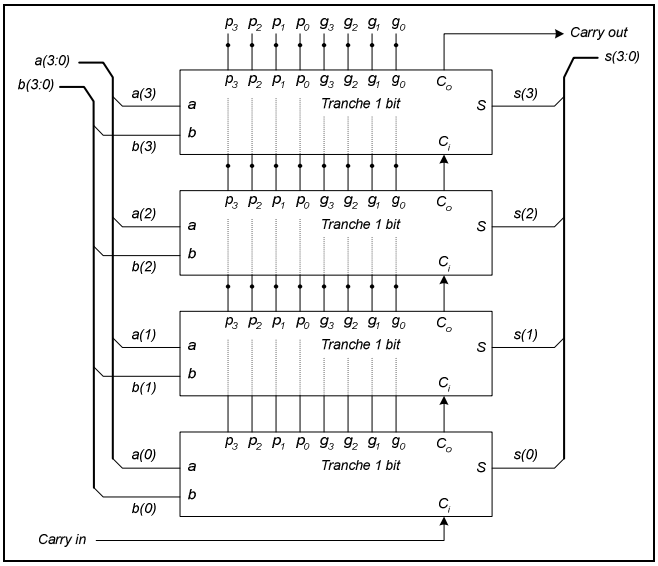
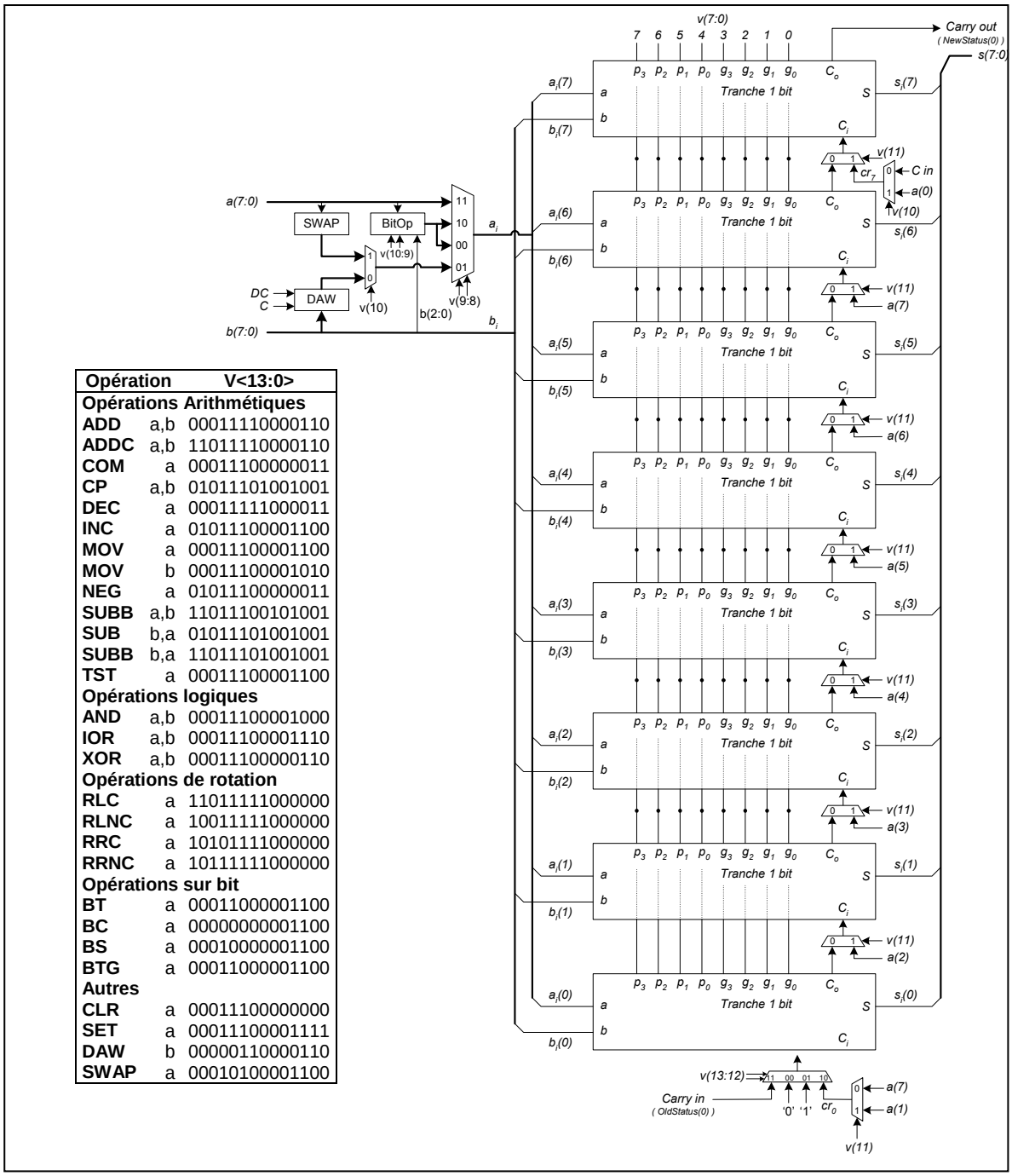


Figure 3.24: Exemple d'ALU 4 bits

Pour concevoir la deuxième **ALU**, huit tranches arithmétiques sont interconnectées pour former une **ALU** de 8 bits. Alors nous avons adapté la structure des interconnexions entre les tranches pour réaliser la fonction de décalage à droite (multiplexeurs) (*figure 3.25*). La structure est complétée pour couvrir d'autres opérations non réalisées par les tranches (DAW, SWAP, opérations par bit). Le vecteur de commande est étendu à 14 bits **V<13:0>**. Les vecteurs de commandes déduis selon l'opération à exécuter sont présentés sur la *figure 3.25.*



Opération		V<13:0>
Opérations Arithmétiques		
ADD	a,b	00011110000110
ADDC	a,b	11011110000110
COM	a	00011100000011
CP	a,b	01011101001001
DEC	a	00011111000011
INC	a	01011100001100
MOV	a	00011100001100
MOV	b	00011100001010
NEG	a	01011100000011
SUBB	a,b	11011100101001
SUB	b,a	01011101001001
SUBB	b,a	11011101001001
TST	a	00011100001100
Opérations logiques		
AND	a,b	00011100001000
IOR	a,b	00011100001110
XOR	a,b	00011100000110
Opérations de rotation		
RLC	a	11011111000000
RLNC	a	10011111000000
RRC	a	10101111000000
RRNC	a	10111111000000
Opérations sur bit		
BT	a	00011000001100
BC	a	00000000001100
BS	a	00010000001100
BTG	a	00011000001100
Autres		
CLR	a	00011100000000
SET	a	00011100001111
DAW	b	00000110000110
SWAP	a	00010100001100

Figure 3.25: Interconnexion des tranches de l'ALU2

La modélisation VHDL de cette structure est très simple, puisqu'elle est régulière et se compose de tranches simples (*listing* 3). Le code VHDL est totalement représentatif du schéma défini. Le code VHDL de cette **ALU** est donné en *annexe 4*. On notera que le calcul du ***Carry*** et du ***DC***, qui se faisait avec un **process** dans la première architecture, est bien simplifié dans cette deuxième technique.

```
entity alu_slice is
    Port ( g   : in   std_logic_vector(3 downto 0);
           p   : in   std_logic_vector(3 downto 0);
               a   : in   std_logic;
           b   : in   std_logic;
           ci  : in   std_logic;
           s   : out  std_logic;
           co  : out  std_logic);
end alu_slice;

architecture simple of alu_slice is

      signal gi01 : std_logic;
      signal gi23 : std_logic;
      signal gi   : std_logic;
      signal pi   : std_logic;
      signal pi01 : std_logic;
      signal pi23 : std_logic;

begin

      gi01 <= g(0)   when (b='0') else g(1);
      gi23 <= g(2)   when (b='0') else g(3);

      pi01 <= p(0)   when (b='0') else p(1);
      pi23 <= p(2)   when (b='0') else p(3);

      gi   <= gi01   when (a='0') else gi23;
      pi   <= pi01   when (a='0') else pi23;

      s    <= gi xor ci;
      co   <= pi or (gi and ci);

end simple;
```

Listing 3: Code d'une tranche ALU

La vérification du module **ALU2** est réalisée avec deux testbench. Le premier est un testbench écrit et chargé d'analyser les résultats de simulation : il compare les résultats obtenus avec les réponses prévues. Ces réponses sont obtenues à partir des données de simulation d'un programme écrit en assembleur sur **MPLAB**. Chaque opération arithmétique et logique est vérifiée avec plusieurs opérandes. Cette simulation nous a permis de corriger certaines erreurs de conception (calcul du Flag OV, calcul DAW -*figure 3.26*-). Quelques

résultats démonstratifs sont donnés à la *figure 3.27.a*. Le deuxième testbench en waveform permet de vérifier les réponses du module suite à des instructions de comparaison (CPFSEQ, CPFSGT ...etc) et de test (TSTFSZ, BTFSC ...etc) (*figure 3.27.b*).

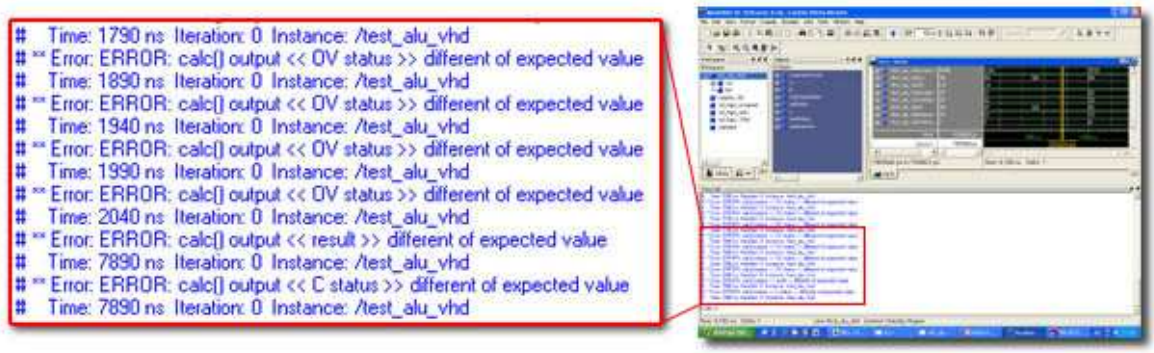


Figure 3.26: Signalisation d'erreurs à la simulation.

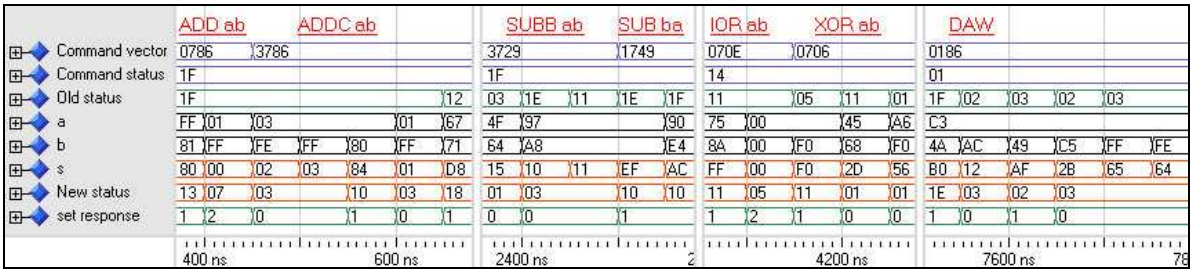


Figure 3.27.a: Résultats de simulation de l'ALU2

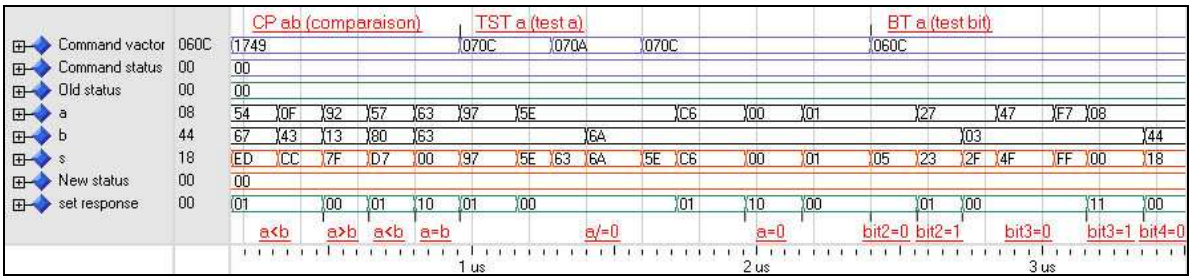


Figure 3.27.b: Résultats de simulation de l'ALU2

La synthèse de ce module se trouve ainsi simplifiée puisque la structure de la tranche est basée sur des **LUT** (*Look Up Table*). Le rapport de synthèse (*annexe 5*) montre que cette **ALU** utilise uniquement 65 Slices et que le délai du chemin combinatoire maximal est de *29,648ns.*

Comparaison des deux ALU

La comparaison des rapports de synthèse des deux architectures révèle que le délai du chemin max passe de *16,358ns* (pour l'**ALU1**) à *29,648ns* (pour l'**ALU2**) soit une augmentation de *81%*. Le grand délai critique de l'**ALU2** provient de la nature série de la structure en plus des délais des multiplexeurs introduits dans le chemin de propagation du carry (voir détail du timing en *annexe 5*). Cependant, l'**ALU2** permet une réduction de 42% du nombre de slices utilisées. Notre choix s'est porté sur la deuxième architecture puisqu'elle occupe moins de surface sur FPGA, et que son fonctionnement est vérifié complètement par les testbenchs développés.

B/ Le Multiplieur

La modélisation VHDL de ce module est directe car la multiplication "**PROD <= a*b**" décrit cette fonction. Le FPGA désigné pour l'implantation contient 16 multiplicateurs de 18 bits prêts à l'emploi. Le synthétiseur reconnaît automatiquement la multiplication dans le code du module et insère un multiplicateur lors de la synthèse du circuit. Ce module est un exemple typique de la simplicité du VHDL et de l'efficacité des FPGA. La *figure 3.28* présente l'entité de ce module.

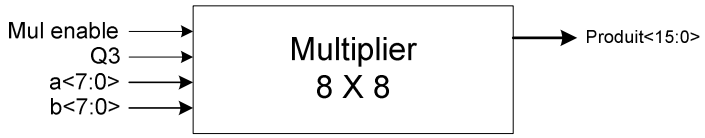


Figure 3.28: Entité Multiplier

C/ Assemblage de l'unité de calcul

L'**ALU** et le multiplieur sont regroupés avec les registres **WREG**, **FREG**, **PROD** dans la même entité dénommée Process Unit (*figure 3.29*).

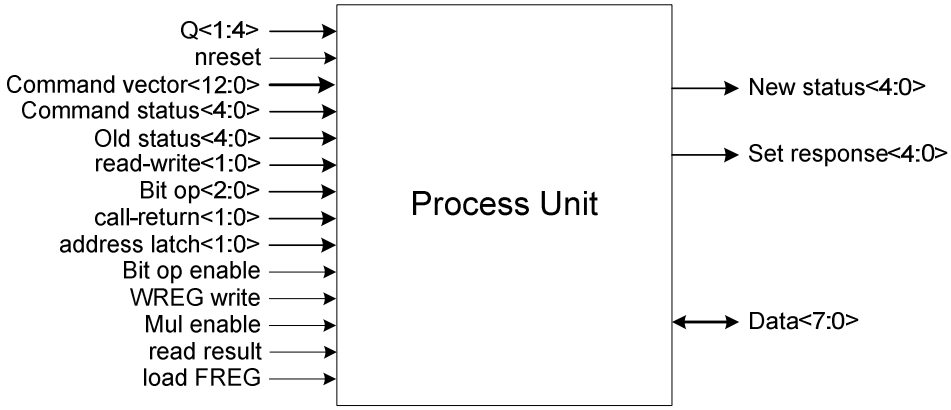


Figure 3.29: Entité Process Unit

3.4.4.2 'Address provider' de la mémoire de données

Ce module fournit l'adresse de donnée selon la description de l'instruction (voir format de l'instruction dans le chapitre précédent). La structure générale de ce module est présentée à la *figure 3.30*. Pour l'adressage indirect, le décodeur **Decod** détecte les adresses indirectes et procède à l'aiguillage de l'adresse **FSR** correspondante.

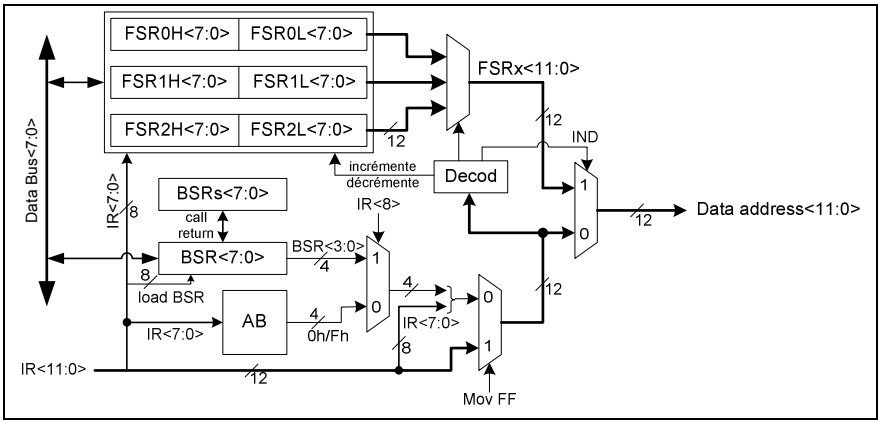


Figure 3.30: Structure du Address Provider

Après modélisation et synthèse, la vérification du circuit est réalisée avec un testbench waveform. Le code VHDL de ce module est donné en *annexe 4*.

Figure 3.31: Entité Address Provider

3.4.4.3 'Program Counter' (PC)

La structure du PC développée englobe, en plus du registre du compteur programme proprement dit, la circuiterie pour gérer les branchements, les appels et toutes les opérations de lecture et d'écriture qui peuvent modifier le contenu du PC à partir du bus. Comme le montre la *figure 3.32*, le vecteur de commande structuré en **Vi** (0 à 6) correspond à la nomination **Command_vector** dans l'entité. Les différents vecteurs de commandes sont explicités dans la *table 3.5*.

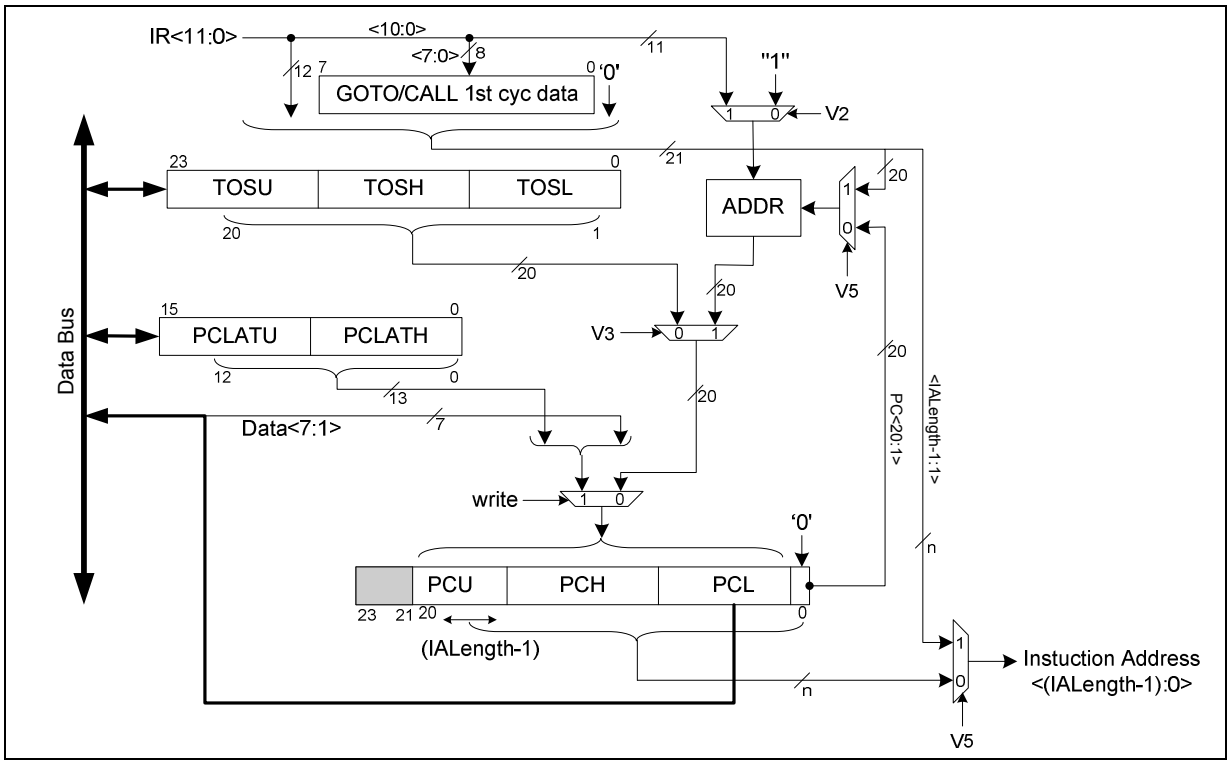


Figure 3.32: Fonctionnement du PC

Opération	**Command_Vector V<6:0>**						
Mode normal (+2)	0	0	0	1	0	1	0
Branchement conditionnel	0	0	0	1	1	0	0
Branchement inconditionnel	0	0	0	1	1	1	0
CALL (1er cycle)	1	0	0	1	0	1	0
CALL (2ème cycle)	1	1	1	1	0	1	0
GOTO (1er cycle)	1	0	0	1	0	1	0
GOTO (2ème cycle)	1	1	0	1	0	1	0
RCALL	0	0	1	1	1	1	0
POP	0	0	1	0	0	1	0
PUSH	0	0	1	1	0	1	0
RETURN / RETFIE	0	0	1	0	0	1	1

Table 3.5: Vecteurs de commande de PC

On note que l'élaboration du code VHDL de ce module a demandé beaucoup d'efforts. En effet, une première structure développée, fonctionnellement correcte, n'était pas synthétisable. Elle est suivie par l'architecture présentée à la *figure 3.32* où des latchs supplémentaires sont définis dans le code VHDL de ce module.

L'adresse d'instruction que fournit le module est de taille générique IALength et elle est déclarée dans le paquetage du MCIP.

La validation de Program Counter est réalisée avec un testbench waveform et consiste à observer le comportement du circuit en réponse aux différentes commandes possibles.

Le listing de ce module est donné en *annexe 4.*

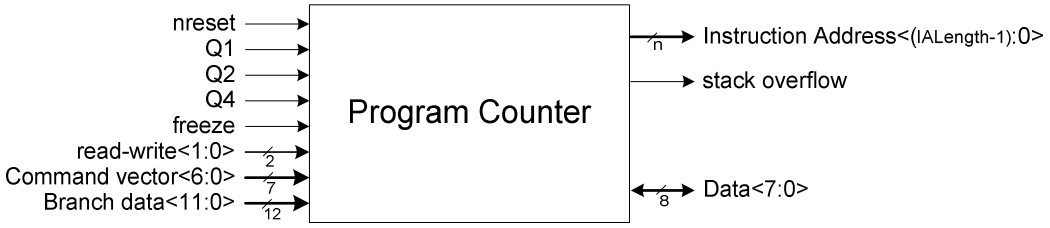


Figure 3.33: Entité Program Counter

Pour des raisons structurelles, la pile est intégrée au module **Program Counter**. **Stack Ram** est une structure régulière simple à modéliser et à synthétiser. Le synthétiseur reconnaît cette structure et attribut 21 cellules RAM distribuées de 32 bits disponibles sur le FPGA choisi (*figure 3.34*).

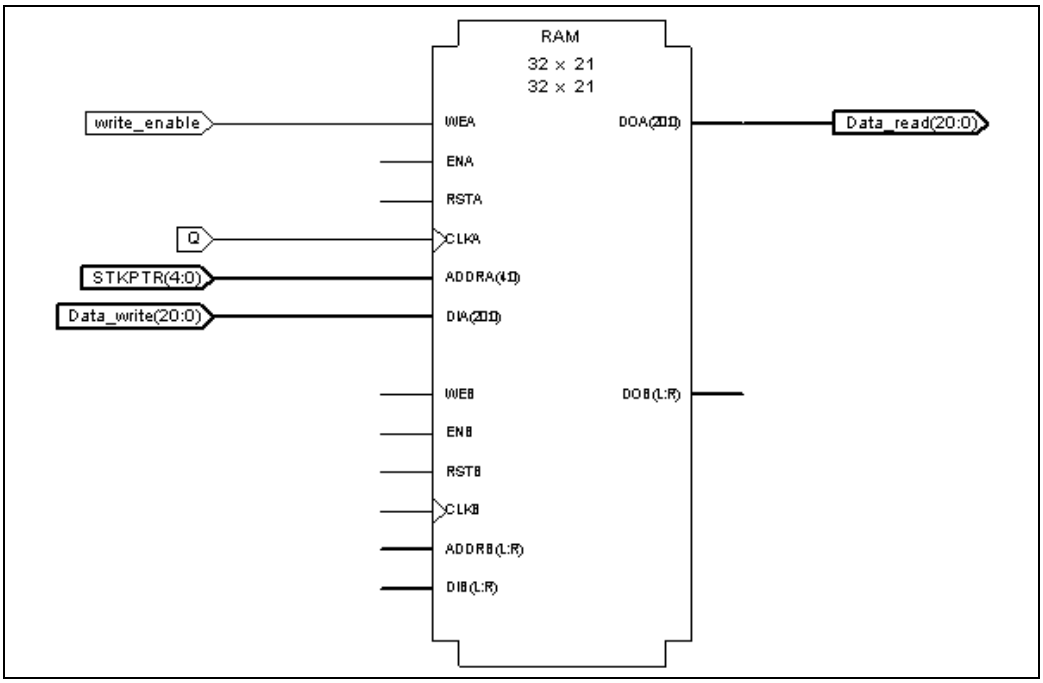


Figure 3.34: Schéma RTL de Stack Ram généré par le synthétiseur

La simulation avec un testbench waveform a validé le bon fonctionnement de ce module.

3.4.4.4 La fonction lecture (Table Read)

Ce module permet de lire des données à partir de la mémoire de programme. Sa structure est donnée par la *figure 3.35*.

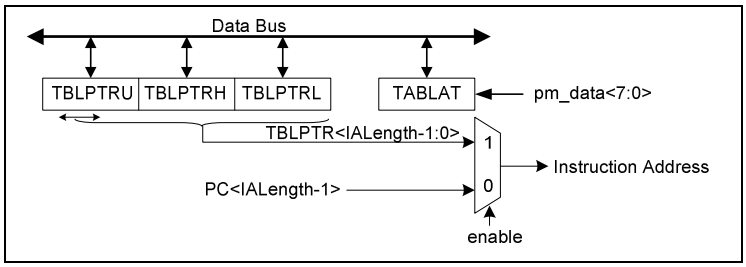


Figure 3.35: structure de la 'Table Read'

Le 'Table pointer' est modélisé, synthétisé et vérifié simplement. Son programme est donné en *annexe 4.*

3.4.4.5 'Instruction Decoder & Control'

Ce module est la dernière partie développée après identification des signaux de commande. Vu sa complexité, il est partitionné en plusieurs blocs fonctionnels.

A/ 'Instruction Control'

Ce module a pour fonction le chargement de l'instruction à exécuter dans le registre d'instruction. En effet, selon les conditions, l'instruction effective peut être différente de l'instruction cherchée, 3 cas se présentent (*figure 3.36*) :

1. Dans le cas d'une interruption de priorité élevée, c'est l'instruction d'appel de l'adresse 00008h qui sera chargée dans le registre d'instruction IR.
2. Dans le cas d'une interruption de priorité basse, c'est l'instruction d'appel de l'adresse 00018h qui sera chargée dans le registre d'instruction IR.
3. Dans le cas de saut conditionnel, si la condition de saut est vérifiée, l'instruction NOP sera chargée dans le registre d'instruction.

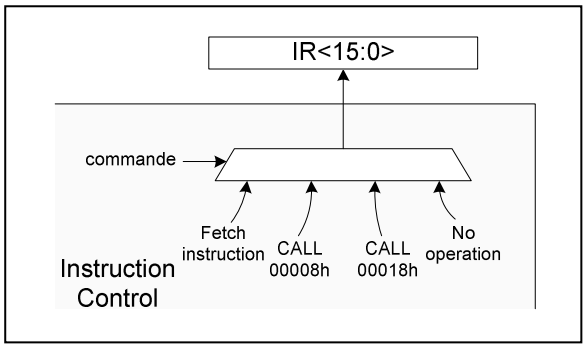


Figure 3.36: Contrôle de l'instruction.

Ce module gère les interruptions en appelant les routines d'interruption, et en contrôlant les bits d'interruption (***GIEH***, ***GIEL***). L'entité de ce module est montrée à la *figure3.37*.

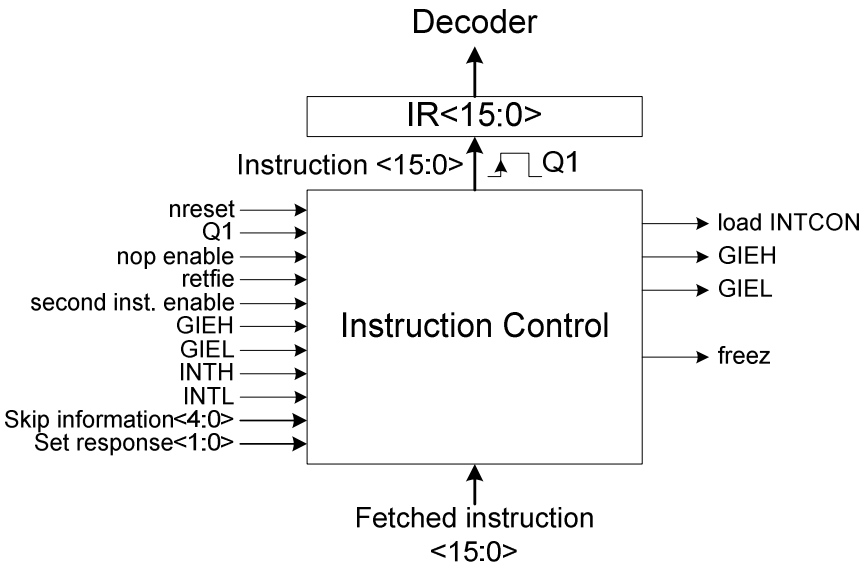


Figure 3.37: Entité Instruction Control

Les stimuli de simulation générés vérifient tous les cas de fonctionnement possibles de ce module (*figure 3.38*).

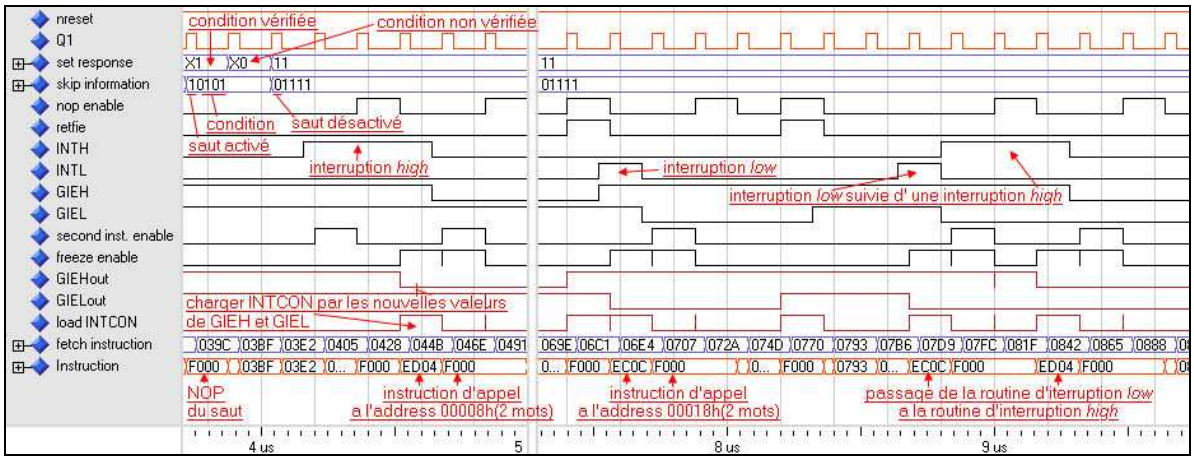


Figure 3.38: Résultats de simulation de Control Instruction

B/ 'Interrupt block'

Ce bloc modélise simplement le circuit d'interruption spécifié dans l'architecture de MCIP (*figure 2.10*). Ce module est validé avec un testbench waveform.

C/ Select Block

Le **Select Block**, purement combinatoire, permet de sélectionner le bloc dont le registre est concerné par l'opération d'écriture ou de lecture décrite par l'instruction. Il joue donc le rôle d'un démultiplexeur, le signal ***read-write command<1:0>*** est multiplexé sur l'un des signaux de sortie ***r_w …*** selon la valeur ***data address <7:0>*** et ***SFR indicator***.

Par exemple : dans le cas d'une écriture dans le registre **TMR0CON**, le **Select Block** active le signal ***read-write timer0***. L'entité de ce module est présentée à la *figure 3.39.*

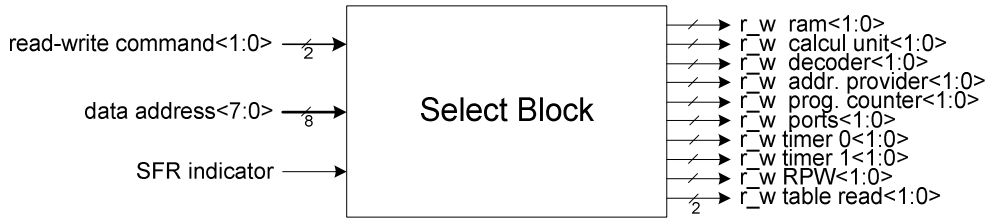


Figure 3.39: Entité Select Block

La vérification de ce module est réalisée avec un testbench waveform, qui consiste à observer l'activation des signaux de sortie en réponse aux différentes adresses appliquées à l'entrée (test non-exhaustif).

D/ La fonction décodage (Decoder)

Après identification de tous les signaux nécessaires pour le fonctionnement du circuit, le décodeur est conçu pour générer ces signaux à partir du décodage de l'instruction (*figure 3.40*). Le signal ***Status<4:1>*** présente au décodeur les valeurs de **N**, **OV**, **Z** et **C** du registre **STATUS** pour lui permettre soit d'effectuer un branchement conditionnel ou continuer directement. Le signal ***Second instruction iden<2:0>*** permet d'identifier la présence du deuxième mot des instructions CALL, GOTO et MOVFF.

Pour simplifier le décodage, les **OPCODE** sont analysés afin de tirer un classement des instructions et ensuite optimiser le code. Ainsi, les instructions sont classées en 16 classes (0-15). Le '**Decoder'** est un circuit combinatoire programmé uniquement avec des ***case*** imbriquées: pour chaque instruction, un nombre de signaux correspondant est activé : ceci se traduit en VHDL par la structure donnée au *listing 4.*

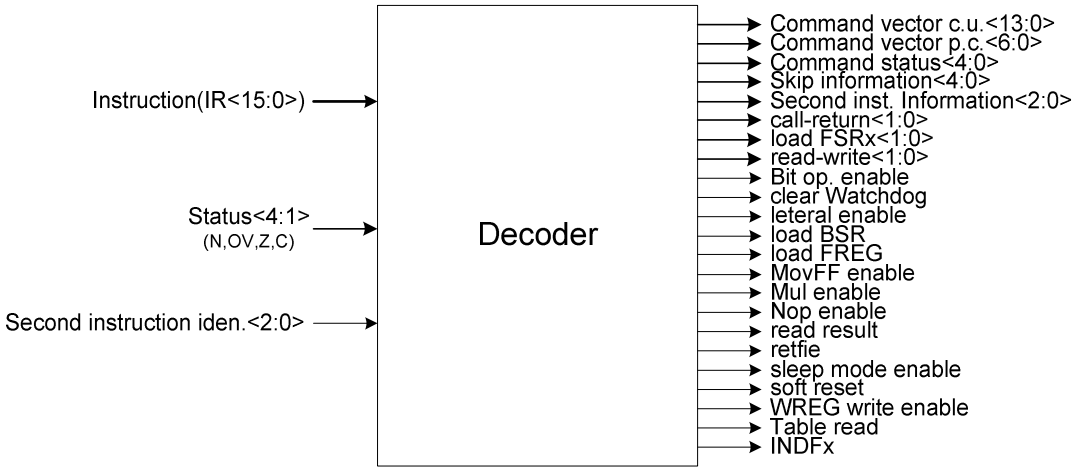


Figure 3.40: Entité Decoder

```
case Instruction(15 downto 12) is
  when "0000" =>                                    -- class 0
  case Instruction(11 downto 10) is
    when "00" =>                                    -- class 0:0
    case Instruction (9) is
      when '1' =>                                   -- MULWF
         Command_vector_cu       <= NOP;
         Command_vector_pc       <= increment;
         Command_status          <= none;
         second_inst_inf         <= sec_disable;
         Skip_inf                <= not_skip;
         call_return             <= none_action;
         nop_enable              <= disabled;
         retfie                  <= disabled;
         read                    <= enabled;
         write                   <= disabled;
         load_FREG               <= enabled;
         read_result             <= disabled;
         WREG_write_enable       <= disabled;
         bit_op_enable           <= disabled;
         literal_enable          <= disabled;
         MUL_enable              <= enabled;
         MOVFF_enable            <= disabled;
         load_FSRx               <= load_disable;
         load_BSR                <= disabled;
         soft_reset_enable       <= disabled;
         sleep_mode_enable       <= disabled;
         clear_watchdog          <= disabled;
         table_read              <= disabled;
         INDFx                   <= enabled;
       when others =>
     case Instruction(8) is
       when '1' =>
        case Instruction (7 downto 4) is
         when "0000" =>                 -- MOVLB
                              .
                              .
                              .
         when others =>                 -- NOP
                              .
                              .
                              .
         end case;
        when others =>
       case Instruction(7 downto 0) is
       when "00000100" =>               -- CLRWDT
                                  .
                                  .
                                  .
       when "00000111" =>               -- DAW
                                  .
                                  .
                                  .
       when "00000110" =>               -- POP
.
.
.
End Decode;
```

Listing 4: La structure du code VHDL du Decoder

La vérification complète de ce module est très longue, néanmoins nous avons observé toutes les sorties correspondantes aux différentes instructions.

E/ Le module de décodage de l'instruction (Instruction Decoder & control)

Ce circuit stratégique dont dépend la réussite globale du projet est développé en dernier, il contient en plus des 4 modules (**Instruction Control**, **Interrupt Block**, **Select Block** et **Instruction Decoder**) le registre d'instruction et quelques registres **SFR** (*figure 3.41*). Le nombre de signaux générés par ce module est de 90. Ces derniers sont utilisés pour commander et contrôler le microcontrôleur.

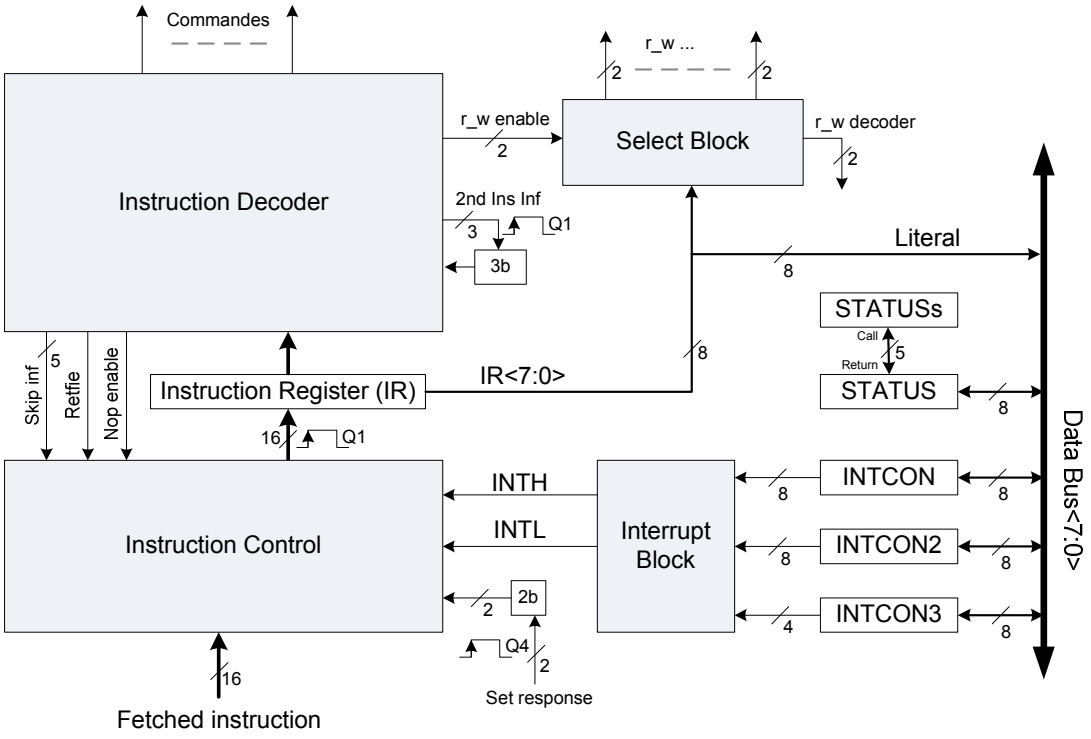


Figure 3.41: Structure du Instruction Decoder & Control

3.5. LES MEMOIRES

La modélisation en VHDL des mémoires est simple. Pour la mémoire de programme, le code de programme d'application est modélisé par une table de constantes (*listing* 5). La taille de cette mémoire ***rom_size*** est définie dans le paquetage du projet. Dans la mémoire de données, les données sont modélisées par une table de signaux (*listing 6*). La taille de cette mémoire ***ram_size*** est aussi définie dans le paquetage.

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

library work;
use work.Use_Pack.all;

entity Program_Memory is
    Port ( Address : in std_logic_vector(IALength-1 downto 0);
           nreset  : in std_logic;
           Q1      : in std_logic;
           Instruction : out std_logic_vector(15 downto 0));
end Program_Memory;

architecture ROM of Program_Memory is

type Rom_Table is array(0 to (Rom_size-1)) of std_logic_vector(7 downto 0);

      constant ROM : Rom_Table :=
                ( 
                X"0A",X"EF",X"2A",X"F0",X"12",X"00",X"FF",X"FF",
                X"FF",X"FF",X"FF",X"FF",X"FF",X"FF",X"FF",X"FF",
                                  .
                                  .
                                  .
                X"D4",X"EC",X"28",X"F0",X"64",X"EC",X"01",X"F0",
                X"FD",X"D7",X"12",X"00",X"FF",X"FF",X"FF",X"FF",
               others => X"FF" );

begin
      process(nreset, Q1)
            begin
             if nreset = '0' then
                   Instruction <= (others => '0');
             else
              if Q1'event and Q1 = '0' then
Instruction <= ROM(CONV_INTEGER( Address(IALength-1 downto 0) + 1 ) )
               & ROM( CONV_INTEGER( Address(IALength-1 downto 0) ) );
               end if;
             end if;
            end process;
end ROM;
```

Code programme :

EF0A
F02A
0012
:
:

Listing 5: Le code VHDL de la mémoire programme

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

library work;
use work.Use_Pack.all;

entity RAM is
    Port ( Address : in std_logic_vector(11 downto 0);
           R_W     : in std_logic_vector(1 downto 0);
           Q1      : in std_logic;
           Q4      : in std_logic;
           DATA    : inout std_logic_vector(7 downto 0));
end RAM;

architecture Beha_RAM of RAM is

type Ram_Table is array(0 to (Ram_size-1)) of std_logic_vector(7 downto 0);

      signal RAM : Ram_Table;
      signal data_read  : std_logic_vector(7 downto 0);
      signal data_write : std_logic_vector(7 downto 0);
```

Table de cellules mémoires

```
begin

       data_write <= DATA;
       DATA_read <= RAM(CONV_INTEGER(Address(DALength-1 downto 0)));

       DATA <= data_read     when Q1 = '1' and R_W(1) = '1' else
              (others => 'Z');

       process
       begin
         wait until rising_edge(Q4);
        if R_W(0) = '1' then
        RAM(CONV_INTEGER(Address(DALength-1 downto 0))) <= Data_write;
        end if;
       end process;

end Beha_RAM;
```

Lecture de données

Ecriture de données

Listing 6: Code VHDL de la mémoire de données

3.6. VERIFICATION DE LA FONCTIONNALITE GLOBALE

La vérification de la fonctionnalité globale du MCIP est très difficile malgré la validation individuelle des sous modules. Cette validation fonctionnelle consiste à exercer l'IP en exécutant différentes séquences d'instructions de vérification. Ce la revient à charger la mémoire programme par les codes des instructions (en hexadécimal), l'exécuter et comparer les résultats obtenus avec les valeurs prévues, ces dernières doivent être aussi calculées. Cette vérification ne peut être menée sans un suivi minutieux de l'évolution pas à pas de dizaines voir de centaines de signaux pendant un nombre de cycles très élevé. Ce débogage génère une grande quantité d'informations à traiter et le fait qu'elles soient des données binaires complique encore la situation. Nous avons effectué toutes les comparaisons manuellement (visuellement). La phase de vérification est la tâche la plus difficile et la plus longue. Nous tenons à noter qu'à travers la vérification, nous avons effectué beaucoup de corrections dans la conception.

3.6.1 Méthodologie

La vérification de la fonctionnalité globale de MCIP englobe les étapes suivantes (*figure 3.42*):

1- **MPLAB**: écrire un programme de test en assembleur en ciblant une fonctionnalité donnée. Le code programme en hexadécimal est exporté dans un fichier *hex.*

2- **Visual C++**: convertir le format du code programme pour l'insérer dans la mémoire programme du modèle VHDL du MCIP. La conversion s'effectue par l'exécution d'un programme développé en C++ chargé à convertir le fichier *hex* en fichier *vhdl*.

3- **Xilinx ISE**: charger le programme converti dans la mémoire de programme du modèle et créer le testbench qui définit l'initialisation du système, l'horloge de fonctionnement, et la durée de simulation.

4- **ModelSim**: lancer la simulation et comparer pas à pas les résultats fournis avec ceux obtenus par la simulation sur **MPLAB** exécutée en parallèle. Les résultats de cette comparaison permettent de corriger les erreurs dans le modèle. (erreur de conception, non-conformité à la spécification …etc).

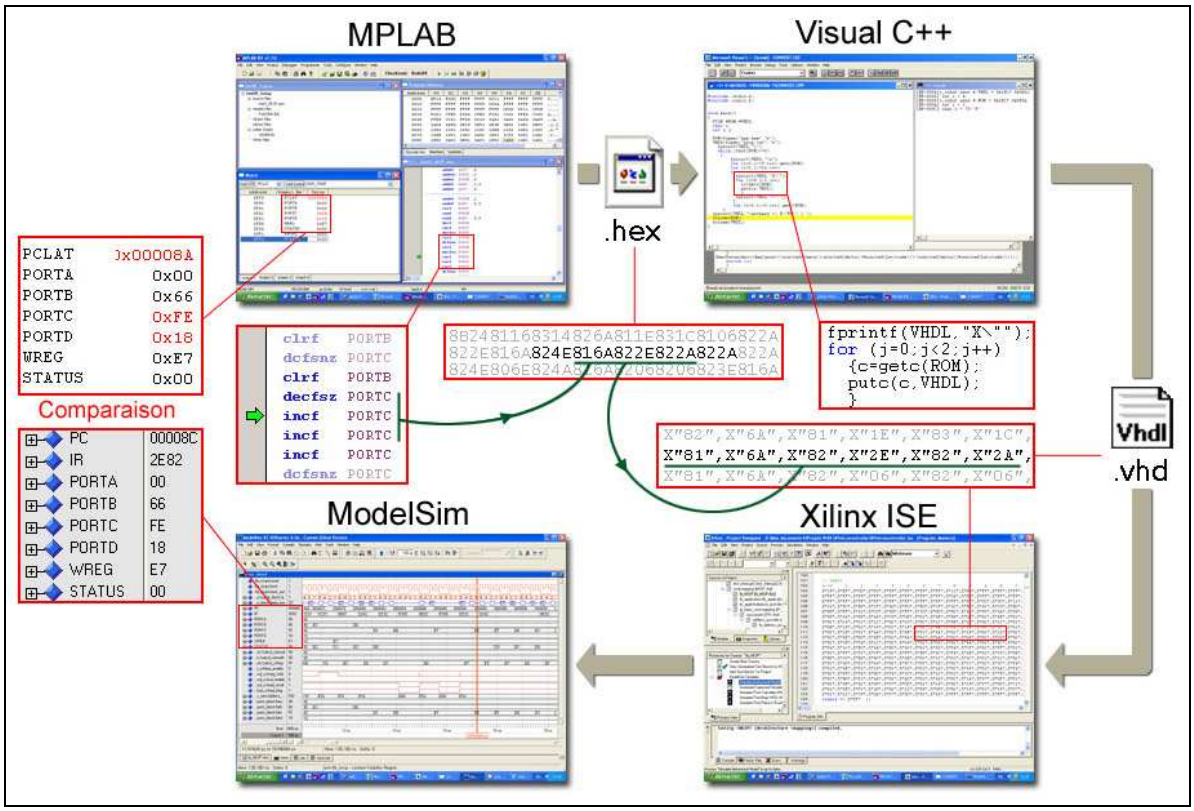


Figure 3.42: L'épopée de stimulus

Ainsi, pour vérifier chacune des *72* instructions dans différents contextes, nous avons exécuté des simulations basées sur les programmes de test suivants :

1- ***Le programme 1***: est destiné à la vérification des opérations de lecture et d'écriture des SFR.

2- ***Programme 2***: pour vérifier les instructions de branchement conditionnel et inconditionnel.

3- ***Programme 3***: pour vérifier toutes les opérations arithmétiques et logiques.

4- ***Programme 4***: pour vérifier les interruptions (priorité ...etc).

5- ***Programme 5***: pour vérifier la fonctionnalité des Timers et Watchdog.

3.6.2 Résultats de la simulation fonctionnelle

Chacun des programmes de test élaborés nous a permis de détecter des failles dans notre conception. Il s'agissait principalement d'erreurs de mapping. Mais on a aussi décelé des erreurs ayant échappé aux premiers tours de vérification. Les corrections adéquates ont étaient apportées à la conception pour enfin aboutir à un modèle ayant une grande assurance de fonctionnement correct. Une partie des résultats obtenus de la simulation fonctionnelle est présentée ci-dessous. Dans la *figure 3.43,* on donne une partie de la réponse du circuit au programme de test n°1. Les résultats de la *figure 3.44* se rapportent au test des interruptions (test n° 4).

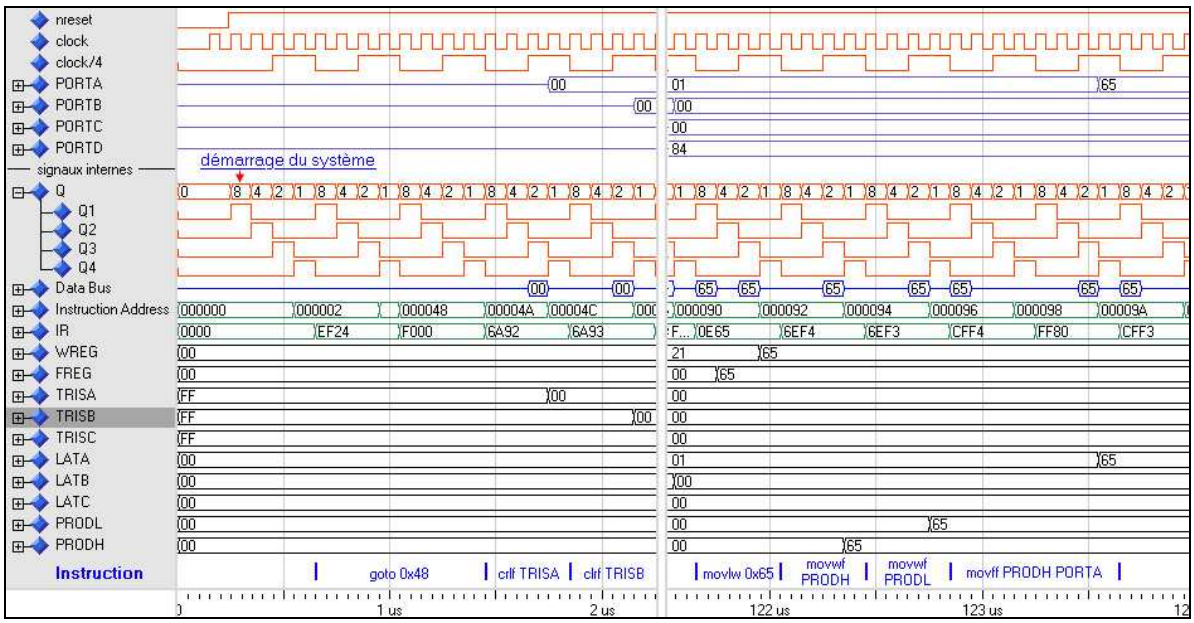


Figure 3.43: Résultats de simulation (programme de test 1)

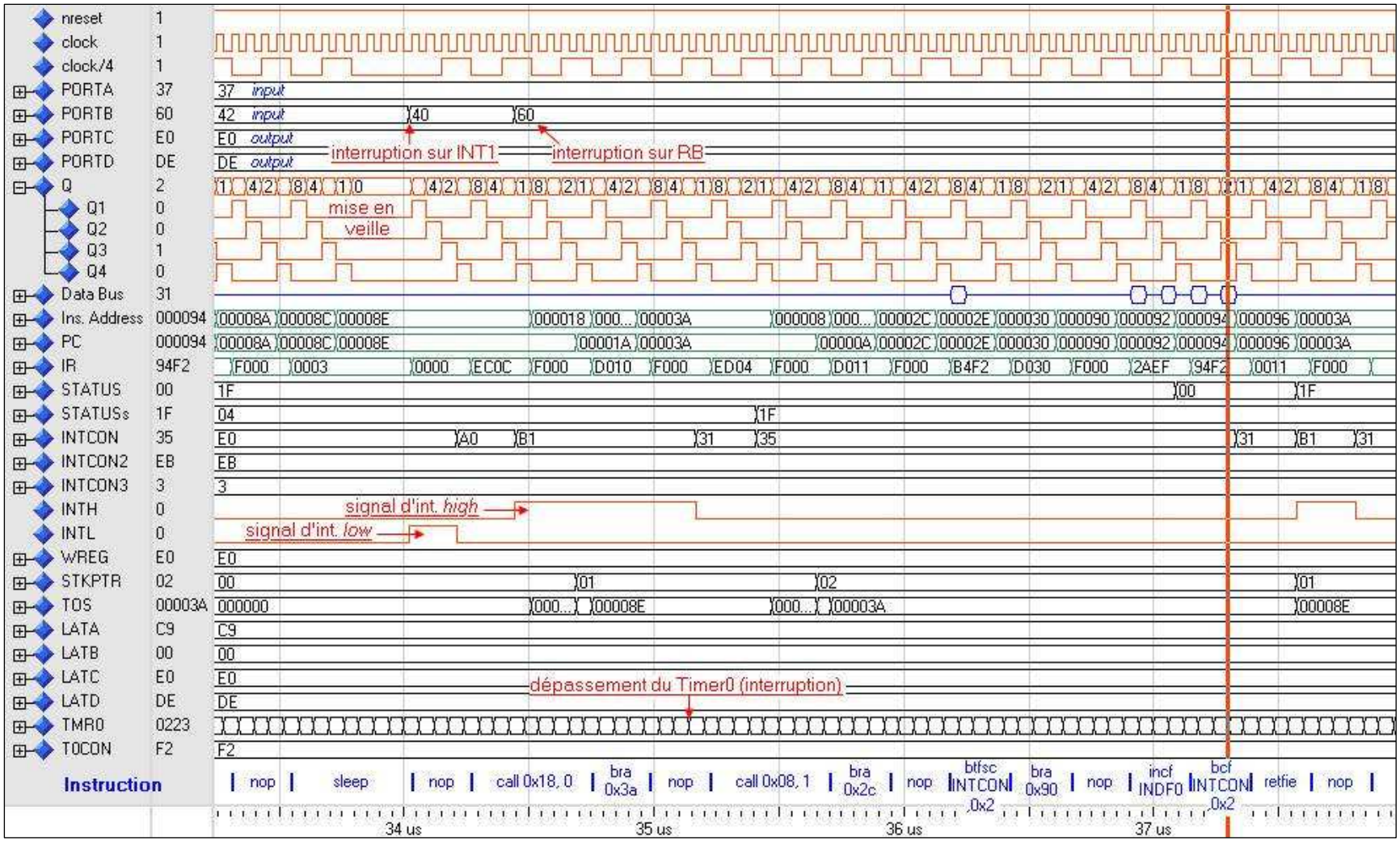


Figure 3.44: Résultats de simulation (programme de test 4)

3.7. VALIDATION SUR CARTE DE DEVELOPPEMENT

La validation du fonctionnement d'un IP par implantation sur FPGA est une phase très importante dans la réalisation d'un projet car elle évite le passage par les méthodes onéreuses de vérifications formelles. Pour vérifier le MCIP développé, on a utilisé le SPARTAN-3E STARTER KIT, c'est une carte de développement de *Xilinx* qui intègre essentiellement un FPGA de type SPARTAN-3E XC3S500E pour des implantations multiples et une multitude de ressources qui permettent l'interaction et la communication avec des systèmes hôtes. Les détails fonctionnels de cette carte sont décrits dans le *chapitre1*.

Pour vérifier l'implantation du MCIP, on a développé un programme de test en langage C sous MPLAB. Ce programme sert à afficher des caractères sur le LCD (16x2 lignes) du Starter Kit et s'exécute de la manière suivante :

- Charger les caractères à afficher dans la mémoire de données à partir de la mémoire de programme par l'utilisation du Table Read.
- Configurer le LCD.
- Transférer les caractères au DD RAM du LCD comme il est montré à la *figure 3.45*.

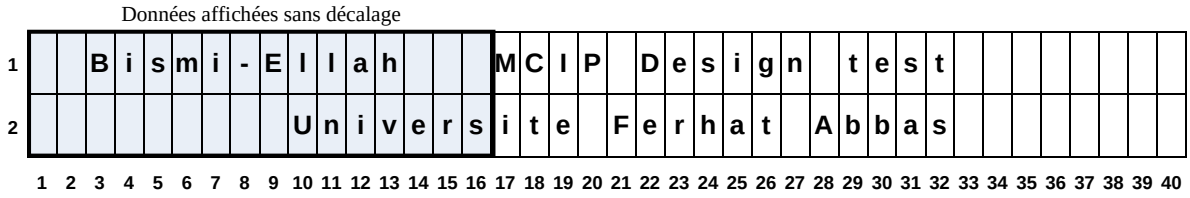


Figure 3.45: Le contenu de la DD-RAM du LCD après chargement des caractères

- Répéter l'opération de décalage à gauche de l'affichage.

La fréquence de l'horloge choisie est de 50MHz car on va utiliser l'oscillateur intégré à la carte.

La taille du programme assembleur généré par le compilateur C18 est 1121 mots. Comme il est montré à la *figure 3.42*, ce programme est exporté en format *hex* et ensuite converti et inséré dans la mémoire de programme du MCIP. Les tailles de mémoires nécessaires pour ce programme sont 4Ko pour la mémoire de programme et 256 octets pour la mémoire de données.

3.7.1 Synthèse

Une partie du résumé du rapport final de synthèse du MCIP pour cette configuration est donné en *listing 7*.

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

Selected Device : 3s500efg320-4

 Number of Slices:                     1766  out of   4656    37%
 Number of Slice Flip Flops:            460  out of   9312     4%
 Number of 4 input LUTs:               3260  out of   9312    35%
    Number used as logic:              3090
    Number used as RAMs:                170
 Number of IOs:                          35
 Number of bonded IOBs:                  35  out of    232    15%
 Number of MULT18X18SIOs:                 1  out of     20     5%
 Number of GCLKs:                         4  out of     24    16%
       .
Timing constraint: Default period analysis for Clock
                   'IP_block/RPW_block/phase_lock_loop/Qs_21'
  Clock period: 17.278ns (frequency: 57.877MHz)
  Total number of paths / destination ports: 582 / 8
-------------------------------------------------------------------------
Delay:               17.278ns (Levels of Logic = 15)
  Source:            IP_block/CPU_block/Calcul_U/FREG_7 (FF)
  Destination:       IP_block/CPU_block/Calcul_U/FREG_7 (FF)
  Source Clock:      IP_block/RPW_block/phase_lock_loop/Qs_21 rising
  Destination Clock: IP_block/RPW_block/phase_lock_loop/Qs_21 rising
```

Listing 7: Une partie du rapport final de synthèse de MCIP

Il révèle que le taux d'utilisation des slices est égal à 37% soit 1766 slices. Le rapport indique que la fréquence maximale est 57,87MHz. Pour estimer cette fréquence, le synthétiseur dispose des durées de tous les chemins logiques de flip-flop à flip-flop synchronisés par la même horloge, la fréquence maximale correspond au temps de propagation maximum.

La *figure 3.46* présente le schéma RTL du MCIP synthétisé.

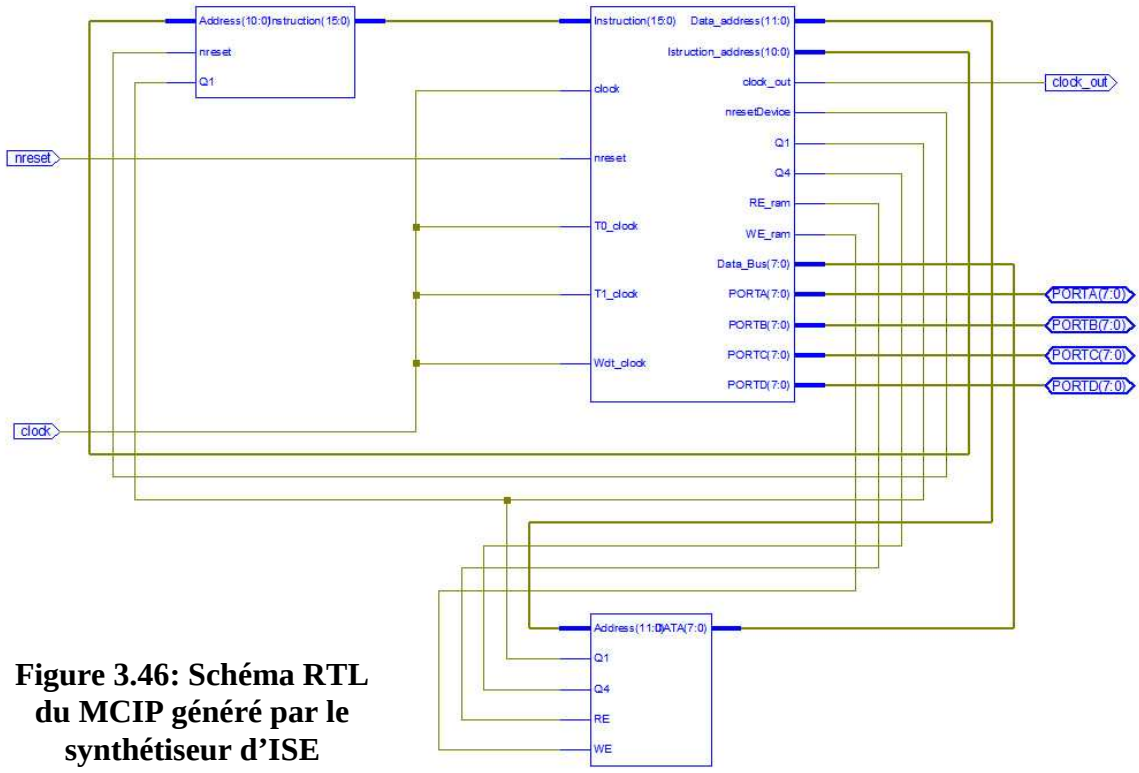


Figure 3.46: Schéma RTL du MCIP généré par le synthétiseur d'ISE

3.7.2 Placement et routage

Après synthèse correcte du MCIP, le placement et routage sur FPGA constituent les phases suivantes. Avant d'effectuer ces tâches, il faut d'abord définir les contraintes technologiques et fonctionnelles. Celles ci concernent essentiellement la fréquence de l'horloge du circuit et l'instanciation des signaux du circuit avec les broches de FPGA. Ces contraintes sont éditées dans un fichier de format *ucf* (*User Constraints File*) ajouté au projet. La fréquence de l'horloge se définit dans l'éditeur de contraintes et elle est prise à 50MHz (High 40%), c'est la fréquence de l'horloge fournie par l'oscillateur qui est intégré sur la carte.

Le placement et le routage du MCIP sont effectués avec succès en respectant les contraintes décrites (*listing 8*). Le rapport de timing a montré que la période minimale est de 17.640ns (*listing 9*) qui correspond à une fréquence maximale de 56.689MHz. La *figure 3.47* montre les éléments utilisés du FPGA où le nombre de slices utilisé est 2188 (46%).

```
NET "clock" TNM_NET = "clock";
TIMESPEC "TS_clock" = PERIOD "clock" 20 ns HIGH 40 %;
#PACE: Start of Constraints generated by PACE
#PACE: Start of PACE I/O Pin Assignments
NET "clock"  LOC = "C9" | IOSTANDARD = LVCMOS33 ;
NET "clock_out"  LOC = "E12" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "nreset"  LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
NET "PORTA<0>"  LOC = "F9" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTA<1>"  LOC = "E9" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTA<2>"  LOC = "D11" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTA<3>"  LOC = "C11" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTA<4>"  LOC = "F11" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTA<5>"  LOC = "E11" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTA<6>"  LOC = "A13" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTA<7>"  LOC = "B13" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTB<0>"  LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "PORTB<1>"  LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "PORTB<2>"  LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "PORTB<3>"  LOC = "F12" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTB<4>"  LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "PORTB<5>"  LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "PORTB<6>"  LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "PORTB<7>"  LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
NET "PORTC<0>"  LOC = "A14" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTC<1>"  LOC = "B14" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTC<2>"  LOC = "C14" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTC<3>"  LOC = "D14" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTC<4>"  LOC = "A16" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTC<5>"  LOC = "B16" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTC<6>"  LOC = "E13" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTC<7>"  LOC = "C4" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTD<0>"  LOC = "A11" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTD<1>"  LOC = "B11" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTD<2>"  LOC = "A8" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTD<3>"  LOC = "G9" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTD<4>"  LOC = "C3" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTD<5>"  LOC = "D7" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTD<6>"  LOC = "C7" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
NET "PORTD<7>"  LOC = "F8" | IOSTANDARD = LVCMOS33 | SLEW = FAST | DRIVE = 8 ;
#PACE: Start of PACE Area Constraints
#PACE: Start of PACE Prohibit Constraints
#PACE: End of Constraints generated by PACE
```

Listing 8: Les contraintes décrites dans le fichier *ucf*

```
Design Summary Report:

 Number of External IOBs                          35 out of 232    15%
   Number of External Input IOBs                  2
      Number of External Input IBUFs              2
        Number of LOCed External Input IBUFs      2 out of 2     100%

   Number of External Output IOBs                 1
      Number of External Output IOBs              1
        Number of LOCed External Output IOBs      1 out of 1     100%

   Number of External Bidir IOBs                 32
      Number of External Bidir IOBs              32
        Number of LOCed External Bidir IOBs       32 out of 32    100%

   Number of BUFGMUXs                        4 out of 24     16%
   Number of MULT18X18SIOs                   1 out of 20      5%
   Number of Slices                       2188 out of 4656   46%
      Number of SLICEMs                    209 out of 2328    8%
        .
        .
--------------------------------------------------------------------------------------------------------
  Constraint                                |  Check  | Worst Case |  Best Case | Timing |   Timing
                                            |         |    Slack   | Achievable | Errors |    Score
--------------------------------------------------------------------------------------------------------
  TS_clock = PERIOD TIMEGRP "clock" 20 ns H | SETUP   |     2.360ns|    17.640ns|       0|           0
  IGH 40%                                   | HOLD    |     0.030ns|            |       0|           0
--------------------------------------------------------------------------------------------------------

Timing summary:
---------------

Timing errors: 0  Score: 0
Constraints cover 1131 paths, 0 nets, and 685 connections
Design statistics:
   Minimum period:  17.640ns   (Maximum frequency:  56.689MHz)
```

Listing 9: Une partie du rapport de placement et routage de MCIP

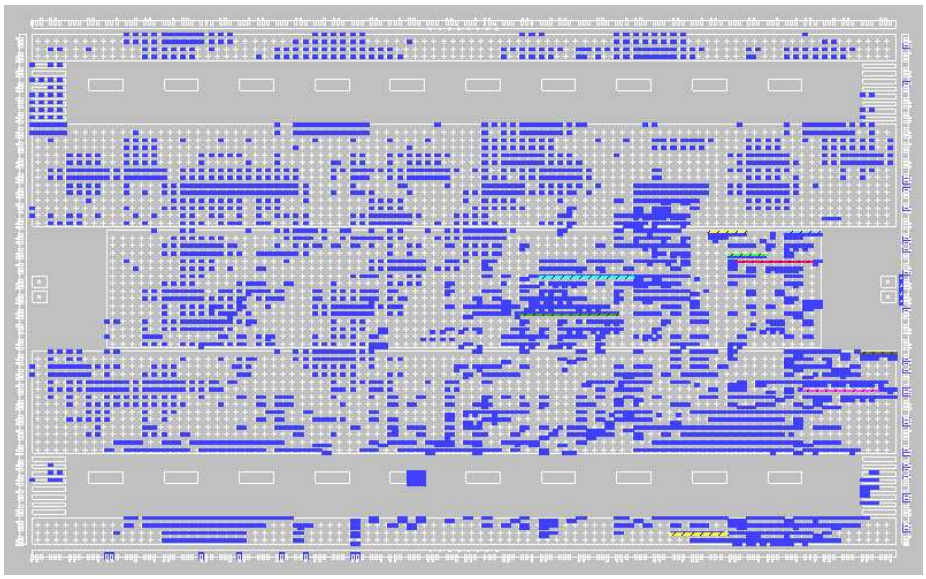


Figure 3.47: Placement de MCIP sur FPGA (XC3S500E)

3.7.3 Simulation après placement et routage

La simulation après placement & routage donne une assurance et produit un impact réel sur les résultats de synthèse et des différentes validations fonctionnelles effectuées auparavant. Elle permet aussi de ressortir et vérifier certains problèmes temporels qui

pouvaient ne pas se manifester avant cette phase [28]. La *figure 3.48* présente le fonctionnement de MCIP au démarrage, les résultats sont identiques à ceux que nous avons obtenus sous MPLAB.

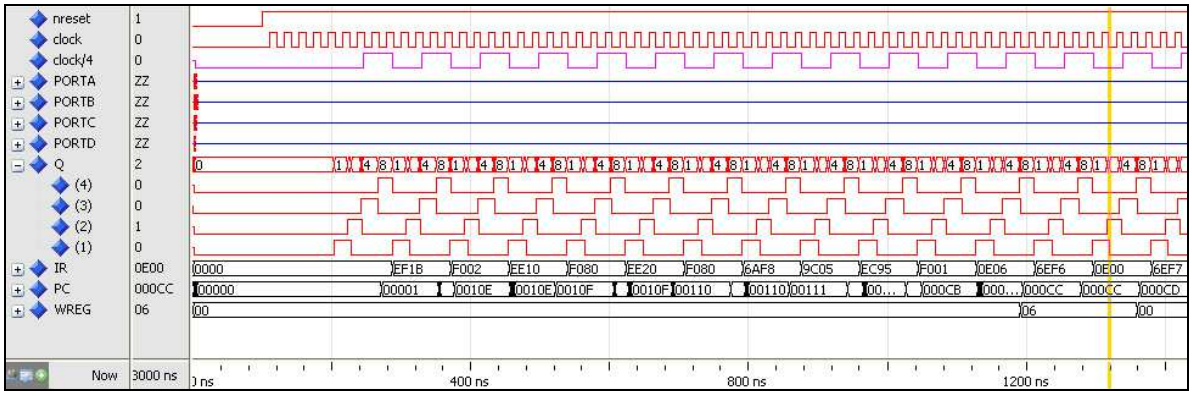


Figure 3.48: Résultats de simulation après placement & routage : au démarrage

3.7.4 Implantation sur FPGA

Cette étape consiste à programmer le FPGA, c'est la phase ou on charge le circuit généré après placement et routage sur le FPGA via le port USB de la carte. La programmation s'effectue en utilisant la routine iMPACT de *Xilinx*.

Après la programmation du FPGA, nous avons réussi à faire fonctionner le MCIP dès le premier essai. En effet, le texte à afficher a défilé correctement sur le LCD du kit (*figure 3.49*) exploitant la fréquence de 50MHz.

Figure 3.49: Gestion de l'affichage sur LCD par le MCIP

3.8. INJECTION DE FAUTES AU NIVEAU VHDL

Pour vérifier si un programme VHDL, développé pour un projet complexe donné, produit effectivement toutes les fonctionnalités attendues (même s'il semblerait que toutes les contraintes et exigences consignées dans le cahier des charges soient réellement prises en compte et considérées au cours des différentes étapes de conception), il est fortement exigé de faire une étude très approfondie du comportement de l'élément développé :

- Pour les états pour lesquels il n'est pas supposé transiter ou s'acheminer d'une part et
- Pour les états jugés fondamentaux dans les cas de non satisfaction fonctionnelle à ne pas produire ou mener à un danger particulier (vie humaine, destruction d'équipements très onéreux …etc) d'autre part.

Alors les outils de vérification de type vérification formelle ou autres sont indispensables. En pratique, il est connu que l'opération vérification est couteuse en temps et exigeante en équipements d'une part, et a besoin d'experts d'autre part. C'est pourquoi malgré les avancés technologiques de ces dernières années, le développement d'un circuit intégré reste une tâche délicate. Des moyens de vérification de type comportemental faciliteraient certainement la mise au point d'un produit car c'est à ce niveau qu'un accès direct aux signaux (degrés de contrôlabilité et d'observabilité élevés) est possible.

Nous proposons une méthode d'injection de fautes à ce niveau d'abstraction qui serait exploitée pour injecter des fautes dans une description VHDL et analyser donc très tôt dans le processus de conception les conséquences de fautes sur le comportement d'un circuit numérique complexe.

Différentes approches ont été proposées pour l'injection de fautes dans des descriptions VHDL [29, 30]. Certaines méthodes suggèrent l'insertion de fautes lors de la simulation tandis que d'autres proposent la modification de la description elle-même pour modéliser les fautes ou l'utilisation des Scan Path. L'incorporation de la technique de l'injection dans la description au lieu de la simulation semble être plus facile et présente un caractère portable entre les différents paquetages de la conception.

Un injecteur de fautes efficace doit être premièrement facile à utiliser, portable et surtout capable d'insérer aussi bien des fautes permanentes que des fautes transitoires à n'importe quel point du circuit. Les fautes transitoires étant les plus difficiles à détecter en pratique exigent un traitement particulier. La cadence, l'endroit et le moment d'injection de fautes sont aussi des spécifications très importantes. Il est nécessaire d'envisager les possibilités d'injection temporelle et spatiale de différents types de fautes (collage à "1" ou à

"0", inversion, fautes multiples …etc).

Nous proposons dans ce contexte un injecteur de fautes à placer dans la description VHDL du circuit à tester et nous détaillons dans ce qui suit son fonctionnement.

3.8.1 La méthodologie d'injection

Le système d'injection de fautes agit en fonction du type de fautes à injecter en modifiant les données ciblées par la faute. Il est composé de deux parties essentielles : le contrôleur d'injection et la logique d'insertion de fautes (*figure 3.50*).

- **Le contrôleur d'injection** contrôle l'insertion de fautes à partir du vecteur de configuration **Ctrl** fourni par l'utilisateur:
  - Ctrl = 0000 : mode d'injection de fautes permanentes.
  - Ctrl = 0001-1111: mode d'injections de fautes transitoires.

  Les bits à changer dans le vecteur de données sont des informations qu'il faut fournir soit par l'utilisateur (dans le cas de faute permanente) ou par le contrôleur (dans le cas de faute transitoire). Dans le cas d'insertion de faute transitoire, il est nécessaire aussi de fournir le moment d'injection généré par un mécanisme pseudo aléatoire. La génération des intervalles pseudo aléatoires est basée sur l'utilisation de LFSR (*Linear Feedback Shift Register*).
- **La logique d'insertion de fautes** introduit les fautes dans le vecteur de données.

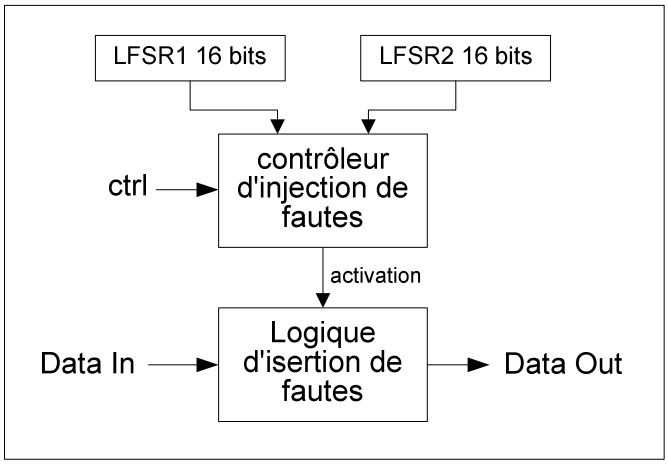


Figure 3.50: Schéma bloc du système d'injection de fautes

3.8.1.1 L'injection de fautes transitoires

Quand Le système reçoit l'un des codes **Ctrl** = 0001 jusqu'à 1111, le contrôleur active le mode d'insertion d'une faute transitoire et procède à la détermination des moments

d'injection d'une manière pseudo-aléatoire. Ceci permet de simuler les fautes à des intervalles réalistes.

A/ Le registre LFSR

Un circuit LFSR est un registre à décalage séquentiel à rétroaction combinatoire (*figure 3.51*). La boucle de rétroaction effectue le XOR de différents bits du registre avant leur réinjection ; le forçant ainsi à générer des séquences binaires pseudo-aléatoires. La séquence des vecteurs générés dépend de l'état initial du registre et du choix des bits participant à la boucle (les valeurs de a_i). Certains choix permettent une séquence de longueur maximum 2^{n-1} pour un registre de *n* bits [31].

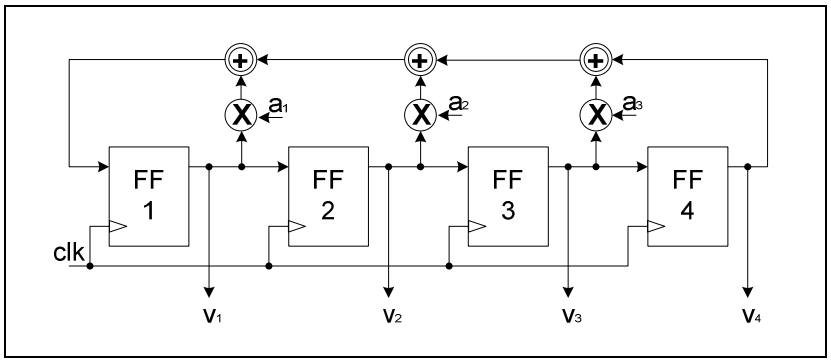
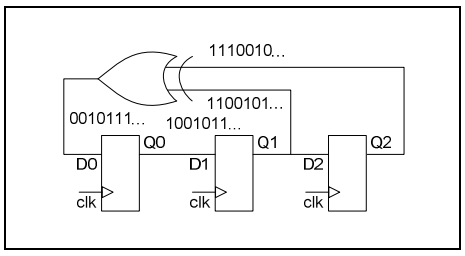


Figure 3.51: Un LFSR à 4 bits

Exemple: La *figure 3.52* montre un exemple de LFSR à 3 bits ainsi que la séquence des vecteurs périodiquement générés avec une période de 7 (2^3-1). L'état initial est $Q_0Q_1Q_2$ = *"111"* avec a_1a_2 = *"01"*



t	$Q^0_{t+1} = Q^1_t \oplus Q^2_t$	$Q^1_{t+1} = Q^0_t$	$Q^2_{t+1} = Q^1_t$	$Q_0Q_1Q_2$
1	1	1	1	**7**
2	0	1	1	**3**
3	0	0	1	**1**
4	1	0	0	**4**
5	0	1	0	**2**
6	1	0	1	**5**
7	1	1	0	**6**
8	1	1	1	**7**

Figure 3.52: Un exemple de compteur pseudo aléatoire à 3 bits

B/ Mécanisme d'injection de fautes transitoires

L'injection de fautes transitoires est réalisée selon l'organigramme de la *figure 3.53*

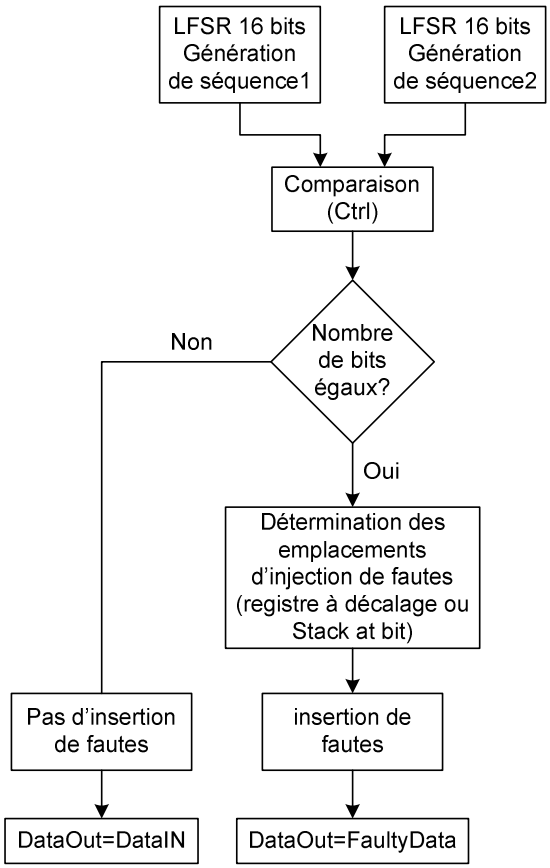


Figure 3.53 : Flot d'injection de fautes transitoires

1/ Deux LFSR fonctionnant en parallèle sont utilisés pour générer des séquences différentes qui seront comparées pour déterminer le nombre de bits égaux. La valeur de **Ctrl** (variant de 1 jusqu'à 15) détermine le nombre de bits des deux séquences qui doivent être égaux pour que l'injection de faute soit autorisée. Autrement dit, le **Ctrl** définit la cadence (pourcentage) d'injection. Les pourcentages théoriques d'insertion de fautes dans le cas aléatoire sont indiqués par la *table 3.6*.

Ctrl	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
%	*50*	*25*	*12.5*	*6.75*	*3.37*	*1.68*	*0.84*	*0.42*	*0.21*	*0.10*	*0.05*	*0.25*	*0.0125*	*0.006*	*0.003*

Table 3.6: Pourcentages d'injection aléatoire de fautes

Exemple : Ctrl = 0001 => La faute sera injectée si un seul bit spécifié dans la 1ère séquence est égale au bit correspondant dans l'autre séquence : parmi les 4 cas possibles (**00**,01,10,**11**) deux cas autorisent la faute => ce qui correspond à une probabilité de 2/4=0.5 (50%).

Dans notre mécanisme d'injection pseudo aléatoire, les pourcentages d'injection sont différents. Les résultats d'étude des pourcentages d'injections de fautes pseudo aléatoires sont donnés par la *table 3.7* et représentés sur la *figure 3.54*. (Ces résultats dépendent des caractéristiques des LFSR).

Ctrl	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
%	*62.65*	*34.6*	*17.89*	*9.16*	*4.69*	*2.37*	*1.13*	*0.57*	*0..28*	*0.15*	*0.064*	*0.038*	*0.015*	*0.009*	*0.006*

Table 3.7: Pourcentages d'injection pseudo aléatoire de fautes

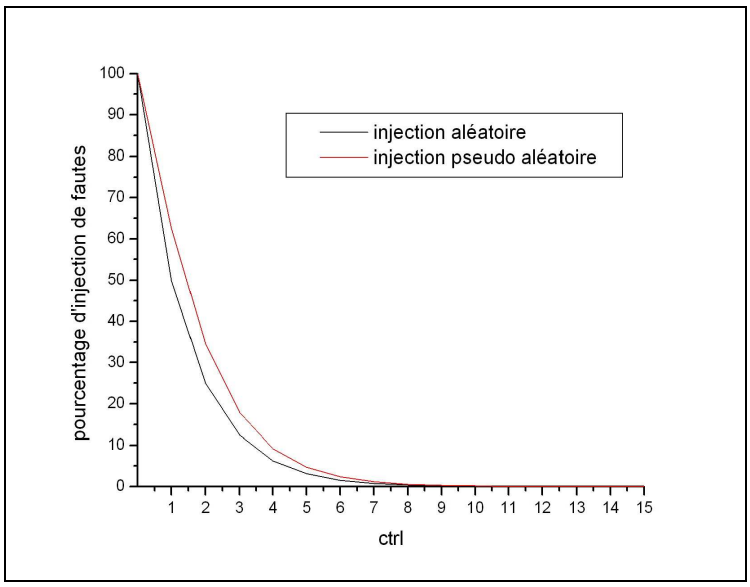


Figure 3.54: Pourcentage d'injection de fautes en fonction de Ctrl

2/ Quand l'injection de faute est autorisée, le contrôleur consulte un registre à décalage de type free running, pour connaître les positions des bits où il faut injecter la faute. Dans ce registre, les bits à 1 indiquent les emplacements d'injection de fautes.

3.8.1.2 L'injection de fautes permanentes

Quand le système reçoit le code **Ctrl** = 0000, la logique d'insertion de fautes injecte un collage à '1' (**StackAtValues** = '1') ou à '0' (**StackAtValues** = '0') dans les bits sélectionnés par **FaultAtBits** (*figure 3.55*).

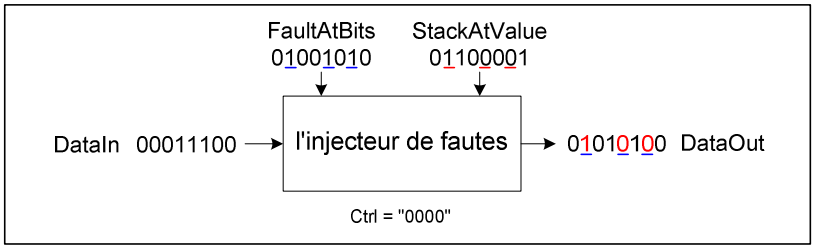


Figure 3.55: Injection de faute permanente

3.8.2 Modèle VHDL du système d'injection de fautes

Le code VHDL de l'injecteur de fautes (**Fault Block**) que nous avons développé est donné en *annexe 4*. Ce module est capable d'injecter des fautes sur *tout signal* dans les parties combinatoires et séquentielles des descriptions VHDL structurelles ou comportementales. L'injection de fautes au signal ciblé est réalisée en "intercalant" le **Fault Block** entre la source du signal et sa destination par simple instanciation. L'instanciation de **Fault Block** est une modification de la description à analyser, ce qui constitue l'inconvénient de cette approche d'injection de faute. Cependant, cet inconvénient est largement compensé par la simplicité de la procédure, la portabilité et les fonctionnalités de **Fault Block**.

Le **Fault Block** étant de nature générique gère les différentes tailles de signaux ciblées (largeur binaire du signal) défini comme ***DataLength*** dans le programme du **Fault Block**.

Exemple d'injection de fautes

Pour démontrer la fonctionnalité de **Fault Block**, nous avons effectué des injections de fautes dans le MCIP. Nous avons choisi, comme démonstration, d'injecter la faute dans la partie commande (***Command vector cu***) afin de suivre toute anomalie naissante que de travailler sur les données généralement suspectes à multiples actions et phases (*figure 3.56*).

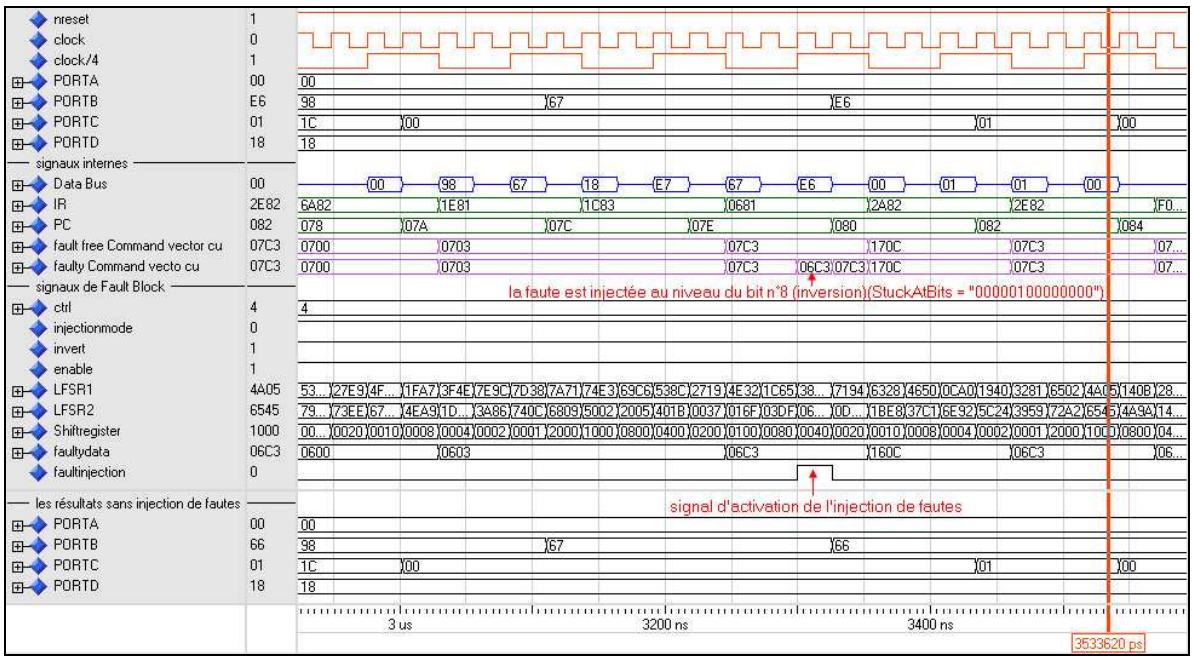


Figure 3.56: Exemple d'injection de fautes

3.9. CONCLUSION

Un programme de plus de *6500* lignes du code VHDL était nécessaire pour la mise au point de notre microcontrôleur MCIP sans compter plus de *25* testbenchs de test et validation qui ont généré à leur tour une quantité immense d'informations dont une grande partie a nécessité un suivi minutieux et une visualisation détaillée. La validation complète du MCIP est l'étape la plus complexe et a nécessité une estimation de *2/3* du temps global de conception. L'expérimentation du MCIP à *50 MHz* avec un FPGA *Spartan-3E (XC3S500E)* a donné pleine satisfaction. Les possibilités de doter notre MCIP de périphériques additionnels ainsi que de bus de test tels que le JTAG (*Boundary Scan*) ou le I^2C sont facilitées grâce à la structure descriptive utilisée et à la vision de reconfigurabilité adoptée dans notre conception.

Le test à haut niveau par injection de fautes est jugé comme une technique prometteuse et simplificatrice du fardeau des opérations de vérification. Une expérimentation du MCIP pour la gestion de l'affichage d'un texte sur LCD du Starter Kit à une fréquence de 50 MHz a été concluante. A ce stade, le MCIP développé peut être considéré comme un produit complet et peut être exploité dans les applications relevant du champ de microcontrôleurs. Néanmoins, une application du MCIP sera présentée, dans le prochain chapitre, pour des opérations de vérification et correction très poussées car il est toujours très difficile de déclarer que le produit mis au point est sans erreurs.

Chapitre 4

Validation et application du MCIP

4.1. PLAN DE VALIDATION

Dans le chapitre précédent, nous avons présenté, à titre démonstratif, l'implantation du MCIP dont la mission était la gestion de l'affichage d'un texte défilant sur un LCD. Il est évident qu'une telle application ne constitue pas un test professionnel du microcontrôleur malgré sa validation complète par simulation.

Une expérience de test fonctionnel consiste à appliquer une séquence d'instructions et le bon comportement du circuit sera déduit à partir des manifestations accomplies ayant des impacts observables. Les instructions doivent couvrir l'ensemble du répertoire du microcontrôleur et activer l'ensemble des blocs fonctionnels. La valeur réelle de couverture de ce programme de test (qualité de détection) ne peut être évaluée quantitativement mais sera établie qualitativement.

Cependant, l'adoption d'une telle démarche exige un choix minutieux de l'application [4]. Après une recherche exhaustive d'applications qui répondent à nos doléances et qui peuvent être implantées localement, nous avons opté pour une application qui gère les horaires de prière. Cette dernière devrait exploiter non seulement des ressources d'entrées-sorties continuellement mais devrait aussi effectuer énormément de calculs reposant sur une utilisation extensive des différents blocs du microcontrôleur et mettant en jeu toutes les instructions de son répertoire. Car il s'agit en fait, de générer des signaux de temps précis, de calculer des fonctions trigonométriques, de résoudre des équations multi-variables et d'afficher des résultats. Cette application est désormais dénommée **APTD** (*Automatic Prayer Time Display*).

Notre travail a commencé par sélectionner une méthode de calcul des horaires de prière qui s'adapte à la programmation et particulièrement en C dans notre cas (*figure 4.1*) car nous avons besoin de deux éléments clés qui nous facilitent l'implantation du projet :

- Le programme de traitement des horaires de prière écrit en C doit produire des données fiables et précises et surtout comparables à celles exploitées par la communauté musulmane (conformes aux tableaux de prières fournis par le ministère des affaires religieuses).
- Ce programme de traitement doit être aussi facilement exploitable en langage machine, c'est-à-dire que son exploitation en C sous l'environnement MPLAB (conversion C-assembleur) nous permet son traitement par le MCIP.

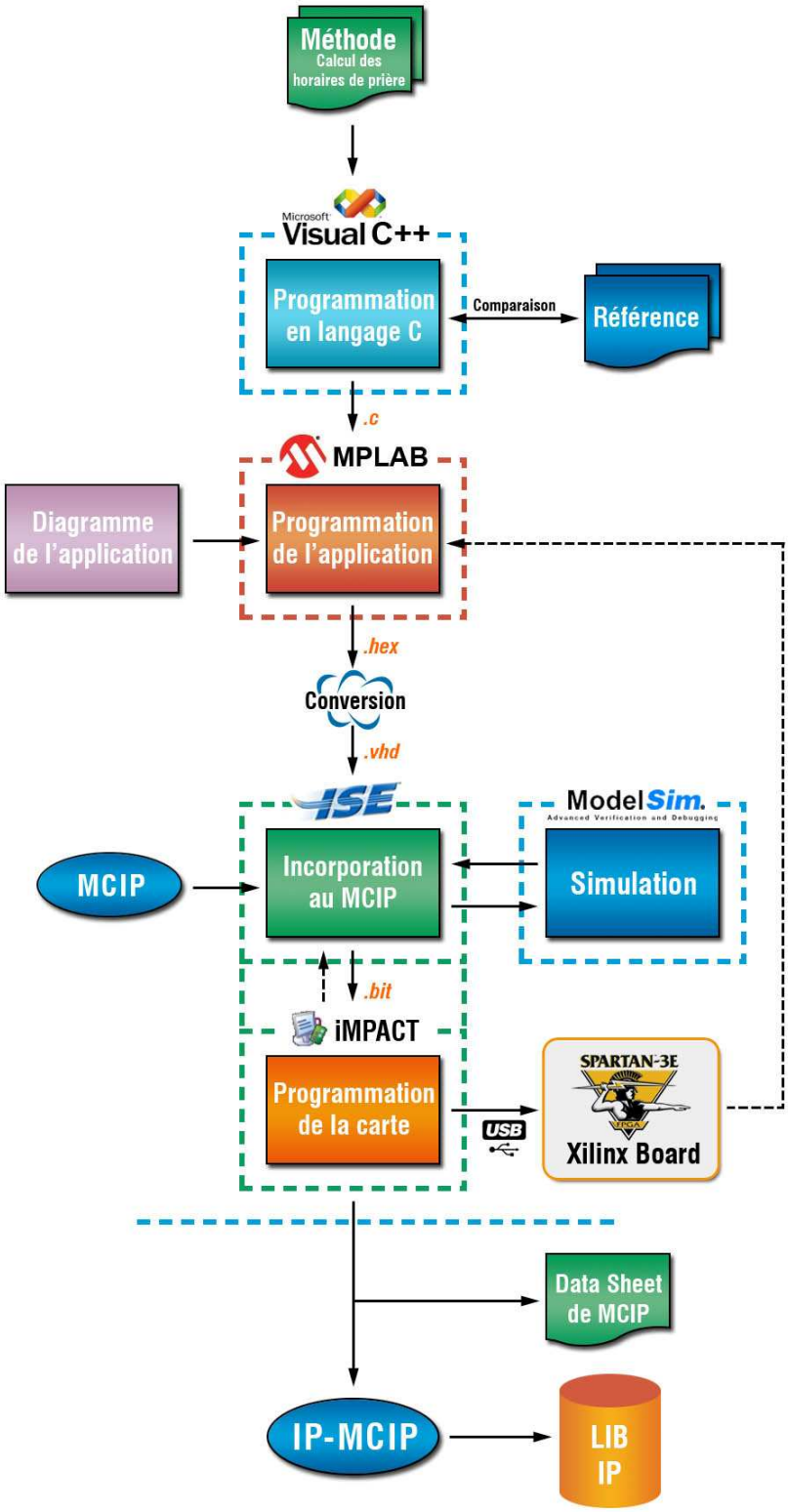


Figure 4.1: Flot de validation

L'étape de traitement sous MPLAB prend en charge la routine de calcul des horaires et les interactions avec le MCIP.

Une fois ce programme gestionnaire des horaires de prière est développé et vérifié, il est exporté et converti en VHDL.

L'étape suivante représente la testabilité réelle de notre MCIP développé et doit être révélatrice du degré de succès du projet en développement. Nous signalons que beaucoup de corrections et d'améliorations ont été nécessaires dans cette phase avant de déclarer notre satisfaction exprimée et quantifiée en termes de valeurs dans les prochains paragraphes.

L'avant dernière étape consiste à l'implantation physique du MCIP sur FPGA pour exécuter l'application APTD.

La finalisation du projet se distingue par la mise en valeur du circuit du microcontrôleur développé sous forme d'un IP Soft acceptable à l'enregistrement sur les sites professionnels d'IP. Le Datasheet de l'IP conçu doit impérativement accompagner le programme générateur (IP et sa protection).

4.2. L'APPLICATION APTD

Dans cette application test, le MCIP qui sera implémenté sur le FPGA est chargé de calculer les horaires de prières pour plusieurs régions définies préalablement et afficher sur le LCD de la carte de développement les données à exploiter. Le réglage des paramètres d'entrée (région, date) s'effectue via des boutons (l'encodeur rotatif et les switchs) disponibles sur cette carte (*figure 4.2*).

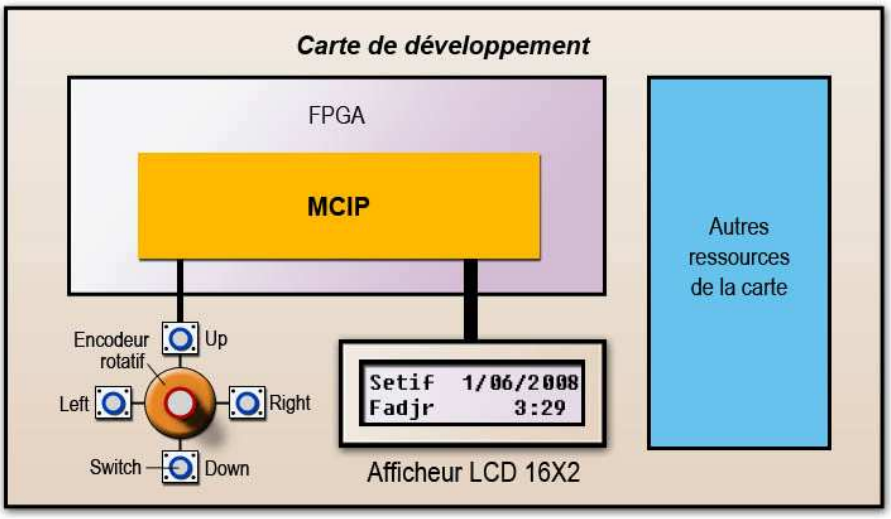


Figure 4.2: Réalisation de l'application APTD sur une carte de développement

Le développement de cette application est passé par une recherche de méthodes de calcul efficaces des horaires de prière.

4.3. DESCRIPTION & SPECIFICATION DU LOGICIEL

La prière est un pilier central de la religion musulmane et doit être accomplie à un temps défini [32]. Les horaires de prière sont en relation avec la position du soleil, dépendent de la situation géographique du lieu et dépendent de la date d'effet. Des méthodes mathématiques ont été proposées pour le calcul de ces horaires en se basant sur le calcul des coordonnées du soleil [33, 34, 35]. Nous présentons dans ce qui suit la procédure appliquée pour le calcul des horaires de prières (*figure 4.3*).

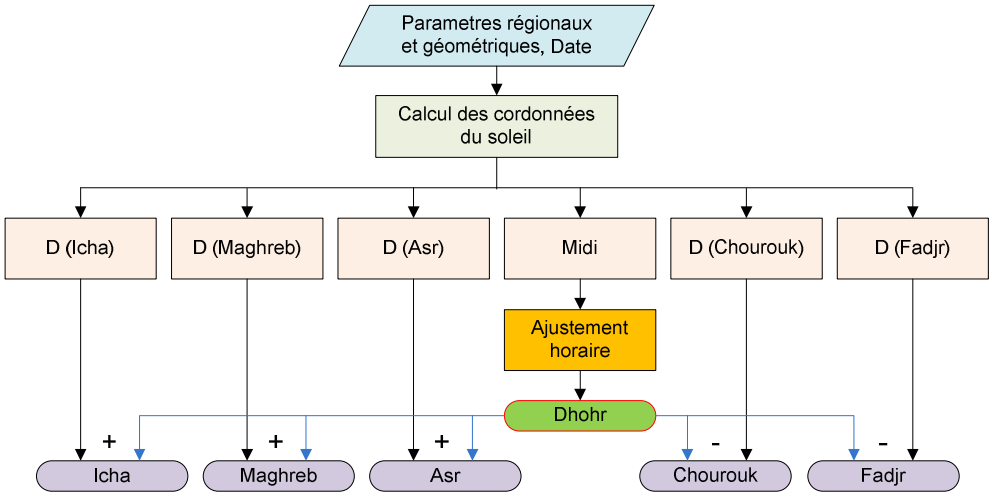


Figure 4.3: Procédure de calcul des heures de prières

4.3.1 Calcul de la position du soleil

La méthode de calcul utilisée permet de déterminer la position du soleil avec une précision de 0.01° [34]. En astronomie, on définit un nombre de coordonnées permettant de localiser les objets dans le ciel, tout comme la latitude et la longitude qui sont utilisées pour donner la position de points à la surface de la terre: on repère la position d'un astre par ses coordonnées sur la sphère céleste, sphère imaginaire située dans les cieux. On suppose que tous les corps célestes sont situés sur cette sphère, sans tenir compte de leur distance réelle à la terre [36]. Les principales coordonnées du soleil sont la déclinaison (δ) et l'ascension droite (α). Par analogie, l'homologue de latitude est la déclinaison et l'équivalent de la longitude est l'ascension droite. Cette dernière est mesurée en heures. La déclinaison du soleil est mesurée en degrés. Les détails de ces coordonnées sont donnés dans la référence [34].

4.3.2 Calcul des horaires de prière

Le calcul est basé sur l'heure du milieu du jour (midi) qui est le temps de la prière de Dhohr. Toutes les autres heures de prière sont calculées par rapport à cette heure. Ce rapport

est déterminé à partir du paramètre *demi arc différentiel* D qui est le temps qui sépare midi du point de calcul [34]. Le *demi-arc différentiel* est donné par la formule suivante:

$$D = \frac{\text{acos}\left(\frac{\sin\theta - (\sin\delta \times \sin x)}{\cos\delta \times \cos x}\right)}{15}$$

δ : représente la déclinaison du soleil.

x : représente latitude du lieu.

θ : représente l'angle entre la position du soleil et l'horizon. C'est ce paramètre qui détermine le temps de la prière: pour chaque prière correspond un θ, et par conséquent un D.

Ainsi :

- **Fadjr = Dhohr – D (Fadjr)**
- **Chourouk = Dhohr – D (Chourouk)**
- **Asr = Dhohr + D (Asr)**
- **Maghreb = Dhohr + D (Maghreb)**
- **Icha = Dhohr + D (Icha)**

4.3.2.1 Calcul de Dhohr

Par définition, l'heure du Dohr est l'heure du milieu du jour ou midi [34]. Pour la calculer pour un jour donné, on utilise la formule suivante:

$$midi = \alpha - ST$$

α : représente l'ascension droite du soleil.

ST : représente le Temps Sidéral [34].

Cette heure représente l'heure de Dhohr absolue, elle doit être ajustée au temps local en ajoutant ou en soustrayant la différence horaire.

Exemple de calcul:

Pour le 1 Janvier 2008, pour la région d'Alger (longitude = 3,3°, latitude = 36,45°)

α = 280,978 = (280,978/15) heure = 18,731 heure

ST = 100,028 = (100,028/15) heure = 6,668 heure

Midi = α - ST = 12,063 heure = 12h 3m 46s.

La différence horaire pour la région d'Alger est de 12° par rapport à (GMT+1) → 48 minutes.

Donc l'heure de Dhohr pour la région de Alger est égale à 12h 51m 46s

On note que le Dohr ne dépend pas de latitude ni de l'altitude.

4.3.2.2 Calcul des heures de Fadjr et d' Icha

Par définition l'heure de Fadjr est établie quand le soleil atteint 18° sous l'horizon Est. L'heure de Icha est quand le soleil descend 18° sous l'horizon Ouest, dans les deux cas, θ = 18°.

$$Fadjr = Dhohr - \frac{\text{acos}\left(\frac{0{,}3090 - (\sin\delta \times \sin x)}{\cos\delta \times \cos x}\right)}{15}$$

$$Icha = Dhohr + \frac{\text{acos}\left(\frac{0{,}3090 - (\sin\delta \times \sin x)}{\cos\delta \times \cos x}\right)}{15}$$

4.3.2.3 Calcul de Asr

Il existe deux définitions du temps de Asr :

- Selon Hanafi (Asr 1): L'heure de la prière est quand l'ombre d'une chose est égale au double de sa taille plus la taille de son ombre au milieu du jour.
- Selon Shafiî (Asr 2): L'heure de la prière est quand l'ombre d'une chose est égale à sa taille plus la taille de son ombre au milieu du jour.

Les valeurs de θ pour les deux définitions sont [34]:

$$\theta(Asr1) = 90 - \text{atan}(1 + \tan(x - \delta))$$

$$\theta(Asr2) = 90 - \text{atan}(2 + \tan(x - \delta))$$

$$Asr1 = Dhohr + \frac{\text{acos}\left(\frac{(90 - \text{atan}(1 + \tan(x - \delta))) - (\sin\delta \times \sin x)}{\cos\delta \times \cos x}\right)}{15}$$

$$Asr2 = Dhohr + \frac{\text{acos}\left(\frac{(90 - \text{atan}(2 + \tan(x - \delta))) - (\sin\delta \times \sin x)}{\cos\delta \times \cos x}\right)}{15}$$

4.3.2.4 Calcul de Chourouk et de Maghreb

Les temps de Chourouk et de Maghreb est quand le soleil descend 0.83° sous l'horizon Est et Ouest successivement (θ=0.8333) [34].

$$Chourouk = Dhohr - \frac{\text{acos}\left(\frac{0{,}0144 - (\sin\delta \times \sin x)}{\cos\delta \times \cos x}\right)}{15}$$

$$Maghreb = Dhohr + \frac{\text{acos}\left(\frac{0{,}0144 - (\sin\delta \times \sin x)}{\cos\delta \times \cos x}\right)}{15}$$

4.3.2.5 Effet de l'altitude

On sait que le soleil se lève sur les hautes régions avant les régions de petites altitudes. Pour prendre en considération l'effet de l'altitude h dans le calcul des heures de Fadjr, Maghreb et Icha, θ sera atténuée d'une valeur égale à : $(0{,}0293\times\sqrt{h})$ [34].

4.3.3 Programmation de la procédure en langage C

Avant d'utiliser cette méthode de calcul, nous avons préféré vérifier sa validité. Pour cela nous avons développé un programme de calcul en langage C en utilisant le VisualC++ (*annexe 6*). Ce programme génère les horaires de prières pour la région d'Alger qui seront comparés avec les tables de prières officielles. Nous avons comparé nos résultats avec ceux affichés pour l'année 2008 et ils sont concordants.

4.4. DEVELOPPEMENT DU PROGRAMME D'APPLICATION SOUS MPLAB

Une fois la méthode de calcul des horaires est validée, nous avons procédé à la production du code programme de l'application complète pour notre microcontrôleur. Selon la description de l'application décrite au *paragraphe §4.2*, le programme est constitué de deux parties essentielles :

- Calcul des horaires selon la méthode validée : Le programme développé sous VisualC++ est exploitable dans le Compilateur C18 du MPLAB.
- Gestion de l'affichage des horaires et acquisition des paramètres d'entrée : en utilisant les interruptions.

4.4.1 Organigramme de l'application

Les *figures 4.4, 4.5, 4.6 et 4.7* présentent les organigrammes du programme développé. L'organigramme présenté à la *figure 4.4* montre la phase d'initialisation. Après affichage d'un message d'accueil sur le LCD, le microcontrôleur est mis en veille. La reprise du mode de fonctionnement normal sera initiée par un événement d'interruption. Cette partie du programme permet d'actionner plusieurs modules et modes du microcontrôleur : le Timer0, le mode 'SLEEP', la gestion des interruptions en mode priorité activée …etc.

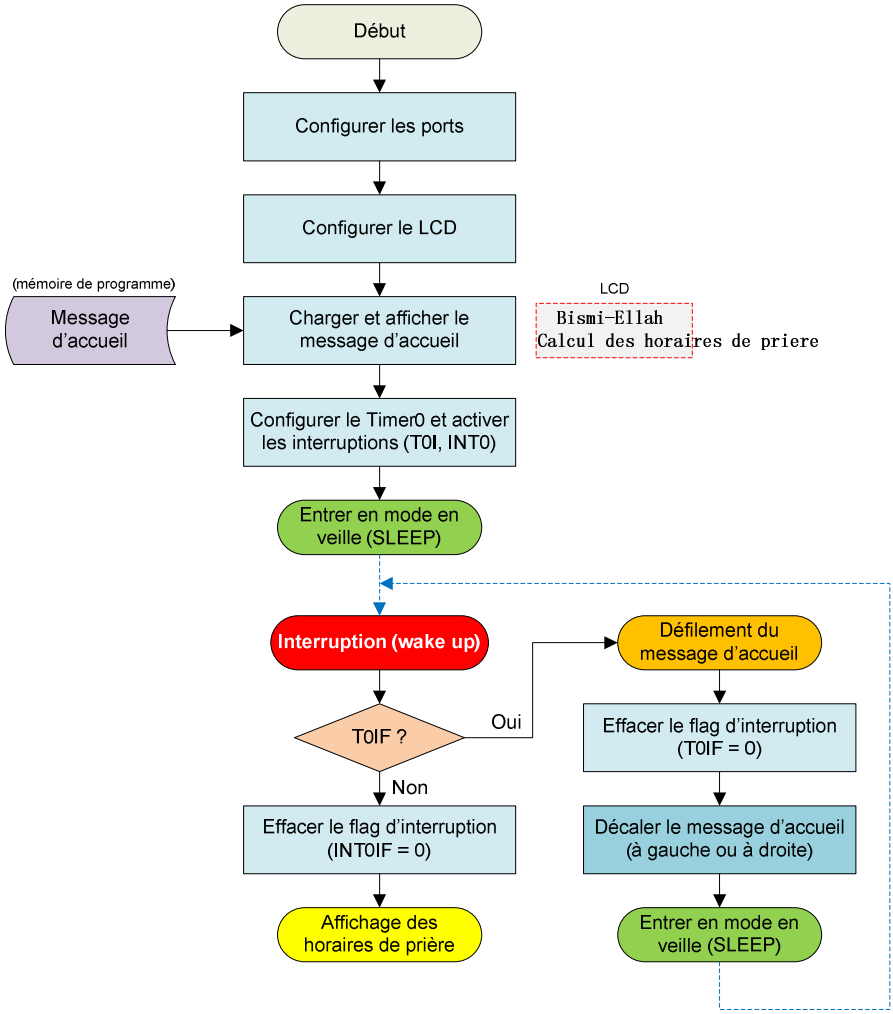


Figure 4.4: Organigramme de la phase d'initialisation du programme

Une interruption sur INT0 (la touche Up) fait passer l'exécution du programme à la deuxième phase de calcul et d'affichage des horaires de prière, le diagramme montré sur la *figure 4.5* montre le déroulement de cette phase. Le programme affiche au début la région (Setif), la date (01/06/2008), la prière (Fadjr) qui sont les données par défaut, et affiche aussi l'horaire calculée (3:29) qui correspond au Fadjr. Les régions, dates et prières peuvent être modifiées par l'enfoncement de l'une des touches :

- Down : modifie la région
- Right : incrémente la prière (prière suivante)
- Left : décrémente la prière (prière précédente).

La rotation de l'encodeur rotatif :

- Rotary A : incrémente la date de un jour
- Rotary B : décrémente la date de un jour.

Le changement de l'un des paramètres décrits entraine un nouveau calcul des horaires et un changement d'affichage. L'affichage d'une nouvelle donnée s'effectue par un effet visuel (défilement), ce qui permet de visualiser mieux le changement et d'exercer d'avantage le microcontrôleur IP.

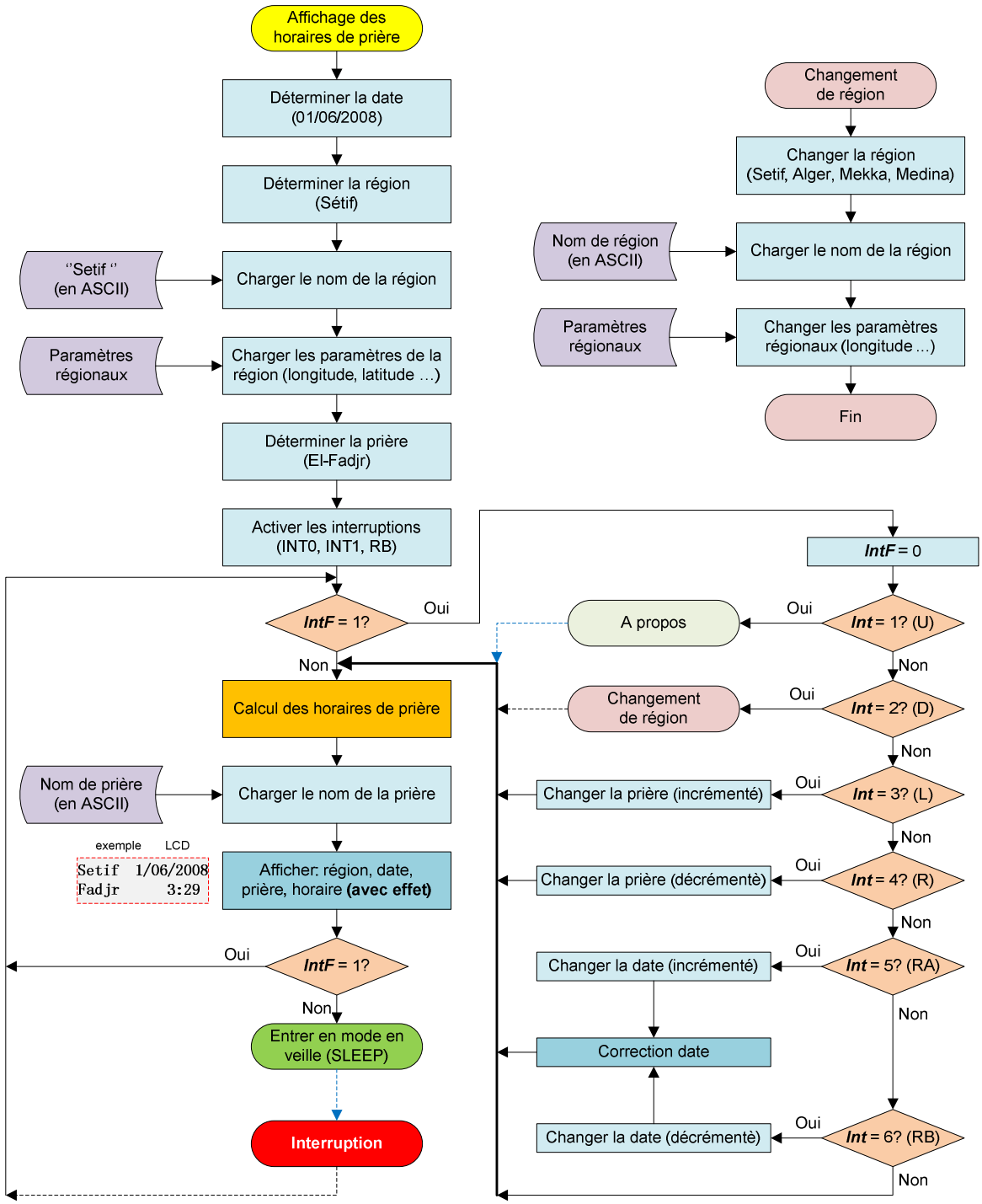


Figure 4.5: Organigramme de la phase d'affichage des horaires

La *figure 4.6* représente le diagramme de la routine d'interruption de la deuxième phase du programme, la source d'interruption est identifiée dans cette routine et elle est marquée par la valeur de la variable ***Int***.

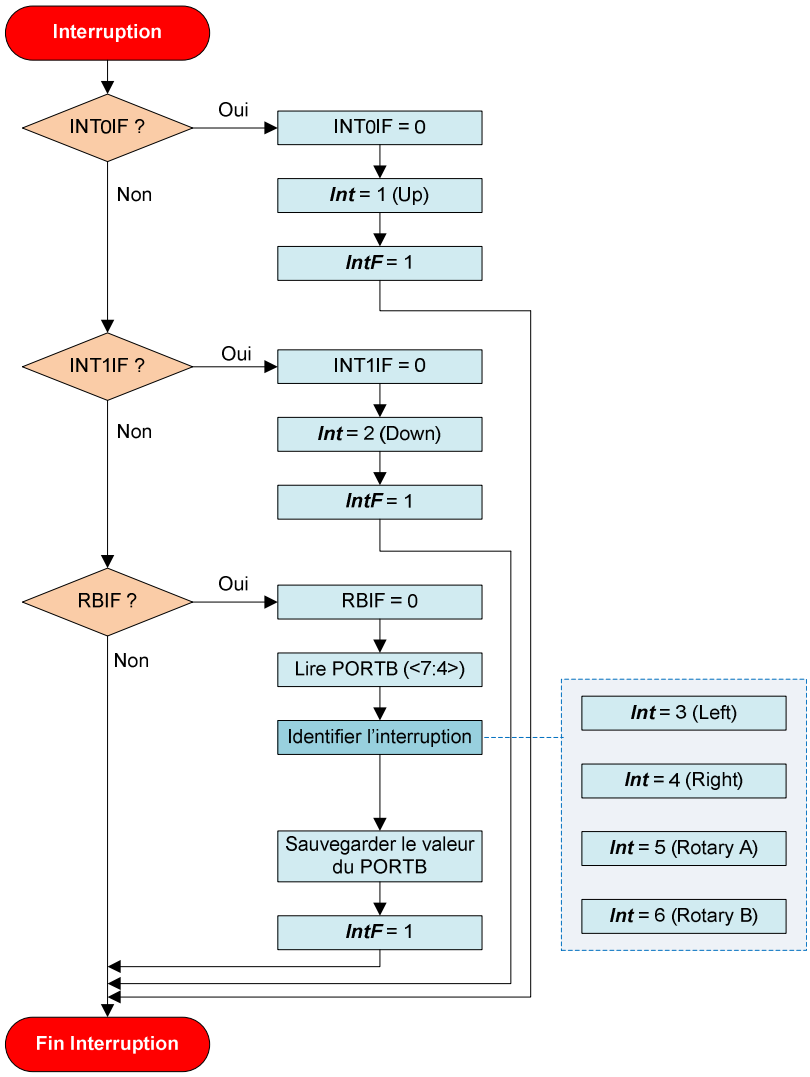


Figure 4.6: Organigramme de la routine de traitement des interruptions

L'interruption sur INT0 (la touche Up) déclenche l'exécution de la routine 'A propos' qui consiste à défiler un texte d'informations. Une activation de la touche Down arrête le défilement du texte et passe au programme principal d'affichage des horaires (*figure 4.7*).

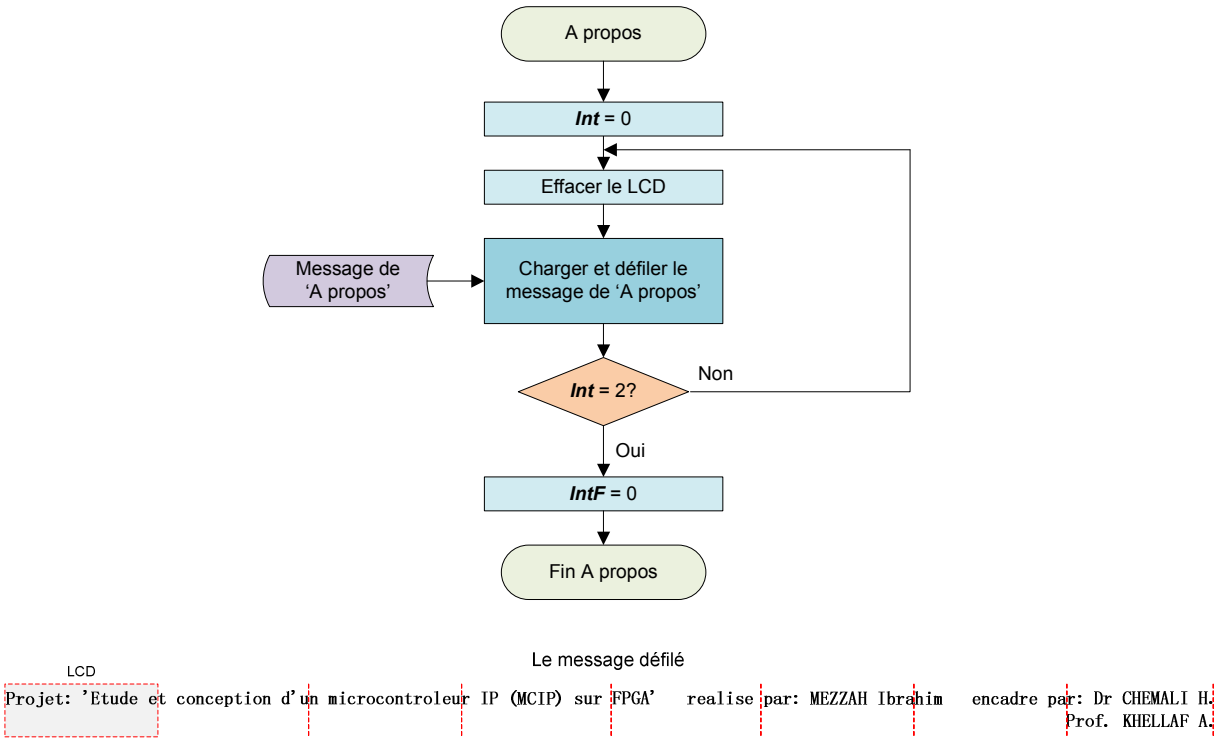


Figure 4.7: Organigramme de la routine 'A propos'

4.4.2 Simulation et test du programme

Après élaboration du programme, on a mené des simulations qui ont exigé certaines corrections (*annexe 7*). Le programme assembleur final généré par le compilateur C18 est de 24388 octets (~24Ko) de taille, dont 18774 octets qui constituent la routine de calcul des horaires (76% du programme).

4.5. INCORPORATION DE L'APPLICATION AU MCIP

En suivant la même procédure montrée à la *figure 3.42*, le programme de calcul des horaires de prière est cette fois ci importé seul dans la mémoire programme de MCIP.

4.5.1 Vérification par simulation et comparaison

La simulation de l'ensemble (MCIP + Programme de calcul des horaires) est exécutée par ModelSim. L'objectif derrière cette simulation est la vérification que le MCIP fournit effectivement les mêmes résultats que ceux fournis par le microcontrôleur PIC18.

Dans la conception du MCIP, on a veillé à ce que chaque instruction s'exécute dans le même nombre de cycles que le PIC18 (*table 2.5*). Cela a pour but, de faciliter la vérification et la comparaison des résultats de simulations. En effet, pour vérifier que le MCIP

fonctionne correctement, on exécute un nombre connu de cycles dans les deux environnements MPLAB et ModelSim et ensuite on vérifie que le MCIP est réellement entrain d'exécuter la même instruction et que le contenu de ses registres est identique à celui des registres du PIC18.

L'exécution du programme se termine après 268727 cycles (nombre total de cycles de simulation). Comme ce nombre est élevé, il est recommandé d'exécuter plusieurs cycles à la fois avant d'effectuer la comparaison, voir plus de 1000 ou même plus de 10000 cycles à la fois.

Si les résultats ne concordent pas après un certain nombre de cycles donnés, l'erreur se trouve entre le dernier cycle exécuté et le dernier cycle qui a donné de bons résultats. On procède après à la recherche du cycle dans lequel l'erreur fut introduite. La localisation du cycle du résultat erroné et par conséquent la faute est difficile vu le nombre élevé de cycles. La correction à apporter, vu la complexité du MCIP, à la conception et au code VHDL sont des tâches délicates. Il est claire aussi que la correction elle même peut introduire d'autres fautes.

4.5.2 Démarche de localisation des erreurs

Comme il était prévu, les premiers programmes de test appliqués au MCIP présentés dans le chapitre précédent n'étaient pas suffisants pour le test de MCIP, bien qu'ils aient aidé à faire beaucoup de corrections à notre design. Le programme de calcul des horaires appliqué au MCIP dans cette phase de test a montré des failles dans la conception et la programmation de MCIP.

L'utilisation du compilateur C18 pour générer le programme de test, qui constitue le calcul des horaires, était une aide importante pour la détection des erreurs de MCIP puisque le programme généré par le compilateur ne prend pas en considération notre microcontrôleur conçu, et il utilise les performances fournies par le PIC18 pour générer le programme assembleur. Donc, pour que le MCIP exécute correctement ce programme, il faut qu'il soit identique au PIC18, aucune simplification ou élimination des éléments essentiels n'était possible, surtout avec notre programme qui est de taille assez importante et qui s'exécute en un nombre élevé de cycles.

Les premières phases de simulation ont montré certaines faiblesses du MCIP et nous ont obligés de revoir de près notre conception. On signale que tous les éléments simplifiés, mal conçus ou négligés dans la conception ont causé des disfonctionnements. On cite l'exemple du module Table Read non intégré initialement à notre architecture, mais suite au

résultat de simulation nous avons constaté son importance et que le programme généré commence à utiliser ce module dès les premiers cycles d'exécution, donc il y a eu nécessité de son intégration à notre architecture de base. L'adressage indirecte constitue un autre exemple où au début n'était pas considéré prioritaire vue sa complexité de conception mais qui est constaté très utile et utilisé extensivement par la suite lors de la simulation. Nous avons reconsidéré son importance dans la conception et le test.

Après l'ajout et la correction de quelques éléments au MCIP, la simulation a finalement donné de bons résultats concernant les premiers cycles de fonctionnement mais nous avons comme même détecté des disfonctionnements en ce qui concerne les cycles lointains. Il faut noter que la procédure de correction a nécessité une stratégie pour éliminer toutes les anomalies sur toute l'étendue des 268727 cycles.

4.5.2 Résultat final de la simulation

Les résultats de la simulation finale ont montré le bon fonctionnement du MCIP. En effet, les contenus de la mémoire de données du PIC18 et celle du MCIP final, produits à la fin de l'exécution du programme, sont identiques (*figure 4.8*).

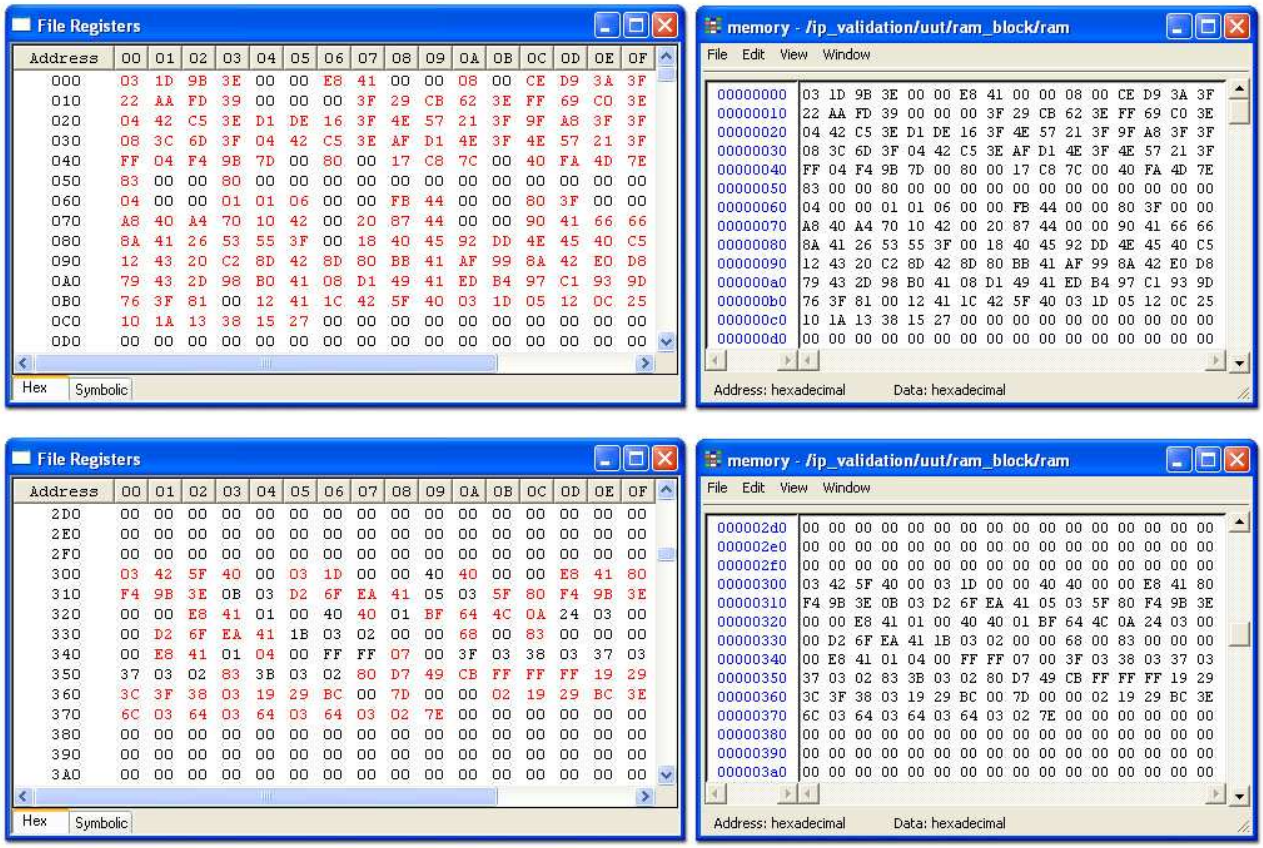


Figure 4.8: Comparaison des résultats finals entre la mémoire de données du PIC18 (à gauche) et celle de MCIP (à droite)

La *figure 4.9* montre les dernières instructions exécutées par MCIP. C'est la même séquence d'instruction qui apparaît dans la fenêtre 'Trace' de l'exécution du programme sur MPLAB (*figure 4.10*). La *figure 4.9* donne aussi les contenus finaux de certains registres du MCIP. Nous remarquons que ces contenus de registres sont les mêmes que ceux du PIC18 donnés à la *figure 4.11*.

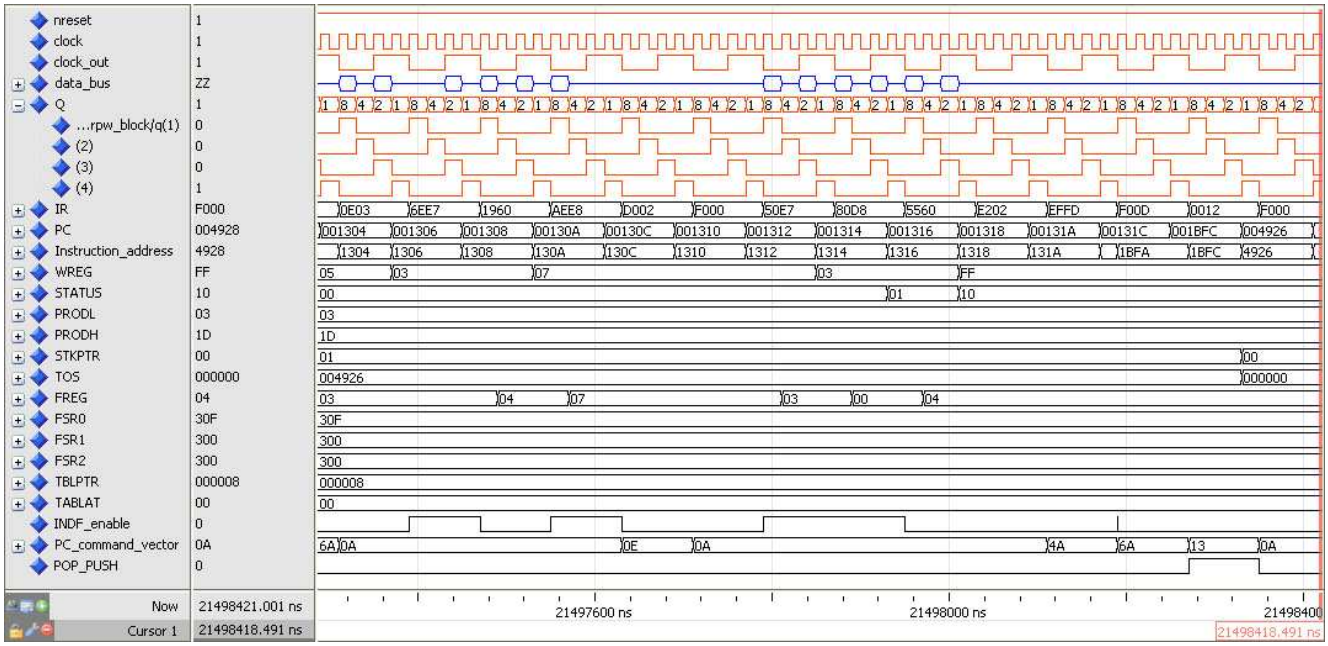


Figure 4.9: Derniers cycles d'exécution du programme par MCIP (calcul prière)

Trace

Line	Addr	Op	Label	Instruction	SA	SD	DA	DD	Cycles
-19	1BEE	CFF4		MOVFF 0xff4, 0	0FF4	1D	00BB	1D	000419A4
-18	1BF0	F0BB		NOP	----	--	----	--	000419A5
-17	1BF2	0100		MOVLB 0	----	--	0FE0	00	000419A6
-16	1BF4	2B60		INCF 0x60, F,	0060	03	0060	04	000419A7
-15	1BF6	EF81		GOTO 0x1302	----	--	----	--	000419A8
-14	1BF8	F009		NOP	----	--	----	--	000419A9
-13	1302	0E03		MOVLW 0x3	W	--	W	03	000419AA
-12	1304	6EE7		MOVWF 0xfe7, A	----	--	0300	03	000419AB
-11	1306	1960		XORWF 0x60, W,	0060	04	----	--	000419AC
-10	1308	AEE8		BTFSS 0xfe8, 0	W	--	W	00	000419AD
-9	130A	D002		BRA 0x1310	----	--	----	--	000419AE
-8	130B	0000		NOP	----	--	----	--	000419AF
-7	1310	50E7		MOVF 0xfe7, W,	0FE7	03	W	03	000419B0
-6	1312	80D8		BSF 0xfd8, 0,	0FD8	00	0FD8	01	000419B1
-5	1314	5560		SUBFWB 0x60, W	0060	04	W	FF	000419B2
-4	1316	E202		BC 0x131c	----	--	----	--	000419B3
-3	1318	EFFD		GOTO 0x1bfa	----	--	----	--	000419B4
-2	131A	F00D		NOP	----	--	----	--	000419B5
-1	1BFA	0012		RETURN 0	----	--	----	--	000419B6
0	1BFB	0000		NOP	----	--	----	--	000419B7

Figure 4.10: Derniers cycles d'exécution du programme par PIC18 (calcul prière)

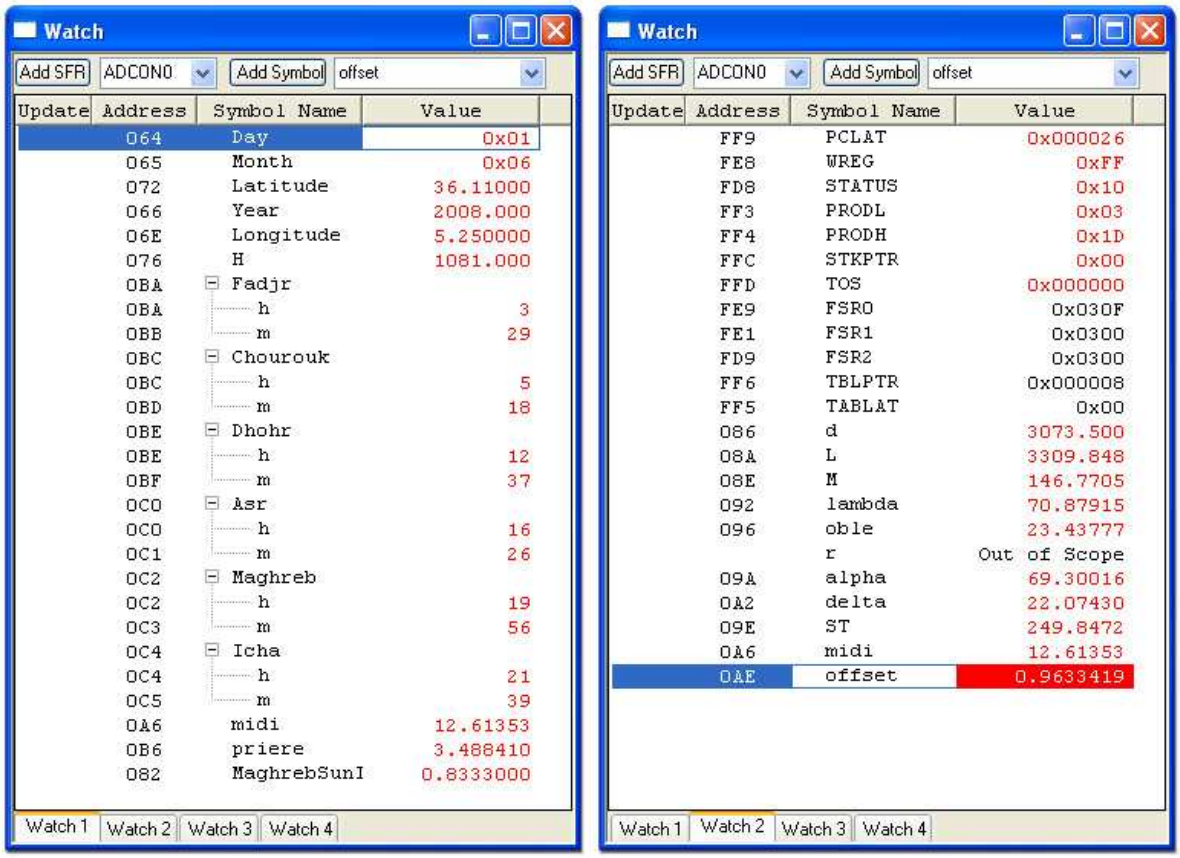


Figure 4.11: L'état final des différents registres du PIC18

4.6. IMPLEMENTATION SUR CARTE DE DEVELOPPEMENT

Après que toutes les corrections soient effectuées sur le MCIP et décision que ce dernier remplit toutes les spécificités du cahier des charges, on a procédé à son implémentation sur FPGA. Comme la taille du programme est importante (24 Ko), il était impossible de l'intégrer dans le FPGA du Spartan Kit. La solution adoptée a eu recours à l'exploitation d'une des mémoires intégrées sur la carte de développement [22]. La mémoire Strata Flash du kit répond à notre besoin puisqu'elle fonctionne en mode parallèle et peut être exploitée en mode 8 ou 16 bits. Néanmoins, un travail supplémentaire est donc nécessaire pour concrétiser le projet global en gérant les échanges entre cette mémoire additive et le FPGA. Les données dans La Strata Flash et le FPGA matérialisent le MCIP et la gestion des horaires de prière.

La *figure 4.12* montre les éléments à implanter sur le FPGA, les ressources exploitées de la carte et les interconnections entre les différents circuits.

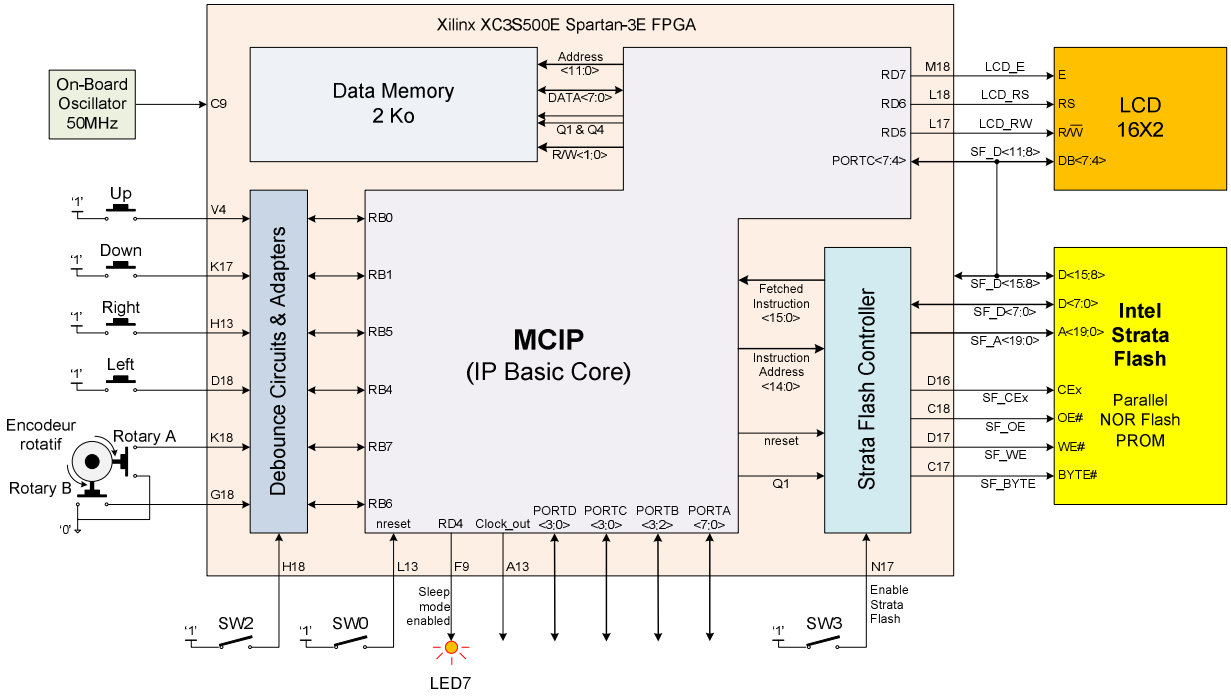


Figure 4.12: Ressources exploitées de la carte pour l'application APTD

Le système à implanter sur le FPGA contient les blocs suivants :

- **Le MCIP (IP Basic Core) :** qui constitue notre microcontrôleur développé et chargé d'exécuter le programme de l'application APTD.
- **La mémoire de données (Data Memory) :** c'est la Ram du MCIP, sa taille est 2 Ko nécessaire pour exécution du programme de l'application qui gère les horaires.
- **Le contrôleur de Strata Flash (Strata Flash Controller) :** s'occupe du chargement des instructions à partir de la mémoire Strata Flash au cours de l'exécution.
- **Les circuits d'anti-rebondissement (Debounce Circuits & Adapters) :** pour filtrer les rebondissements des interrupteurs et de l'encodeur rotatif.

4.6.1 Utilisation de la mémoire Strata Flash

Pour charger cette mémoire, il était nécessaire de concevoir des circuits (en VHDL) pour effectuer cette opération à partir de FPGA puisque l'outil de Xilinx ISE n'offre pas la possibilité de chargement direct à travers le câble USB ou JTAG. A l'aide de ces circuits supplémentaires définis (voir *annexe 8*), on a pu effacer la Strata Flash et la charger par le

programme complet de capacité 24Ko. Par la suite nous avons aussi vérifié le bon chargement du programme dans la mémoire.

Nous avons conçu un autre module **Strata Flash Controller** qui s'occupe de la lecture de l'instruction à exécuter. L'utilisation de Strata Flash en mode 16 bits n'était pas possible à cause des liaisons partagées associées au LCD (SF_D<11 :8> *figure 4.12*). Le Strata Flash Controller conçu charge l'instruction en deux parties dans un même cycle comme le montre la *figure 4.13*. (Le code VHDL de ce module se trouve en *annexe 8*)

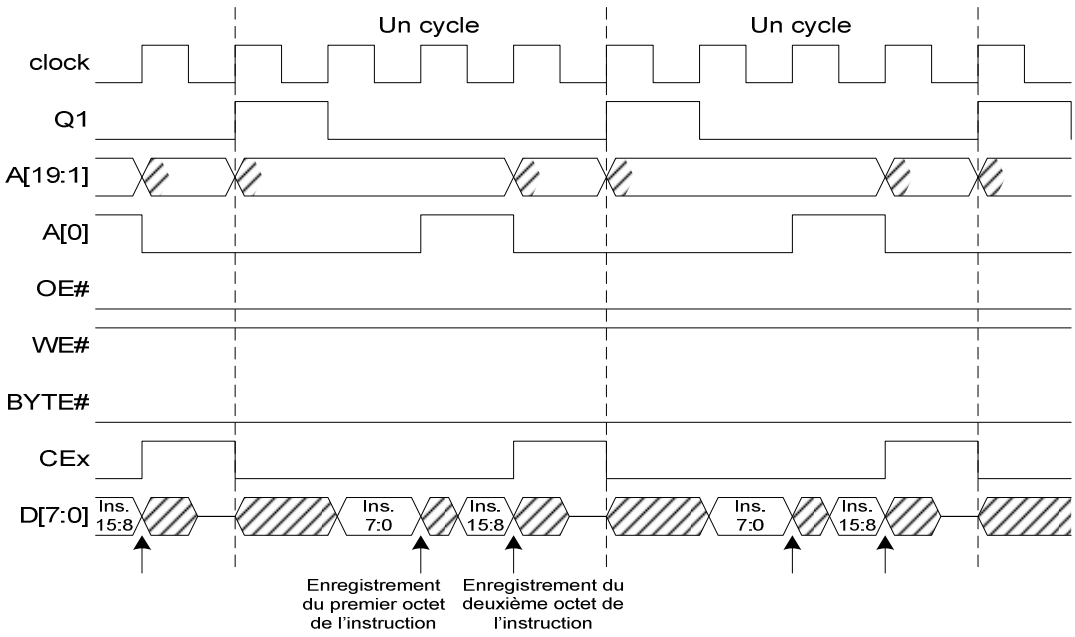


Figure 4.13: Chargement des instructions à partir de la mémoire Strata Flash

Pour que notre chargement de l'instruction s'effectue correctement, il fallait respecter rigoureusement les spécifications de la Strata Flash [37]. Nous avons utilisé une fréquence de 25MHz sous multiple du master clock du kit (50MHz).

4.6.2 Synthèse & Edition de contraintes

La synthèse du design complet est effectuée avec succès (*listing 1*) indiquant une fréquence maximale de 69,81MHz et un taux de 35% d'utilisation des slices du FPGA.

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

Selected Device : 3s500efg320-4

 Number of Slices:                    1642  out of   4656    35%
 Number of Slice Flip Flops:           525  out of   9312     5%
 Number of 4 input LUTs:              3139  out of   9312    33%
    Number used as logic:             2071
    Number used as Shift registers:      2
    Number used as RAMs:              1066
 Number of IOs:                         69
 Number of bonded IOBs:                 69  out of    232    29%
 Number of MULT18X18SIOs:                1  out of     20     5%
 Number of GCLKs:                        5  out of     24    20%
Selected Device : 3s500efg320-4

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

=========================================================================
Timing constraint: Default period analysis for Clock 'IP_block/RPW_block/phase_lock_loop/Qs_21'
  Clock period: 14.323ns (frequency: 69.818MHz)
  Total number of paths / destination ports: 822 / 8
-------------------------------------------------------------------------
Delay:               14.323ns (Levels of Logic = 12)
  Source:            IP_block/CPU_block/Calcul_U/FREG_0 (FF)
  Destination:       IP_block/CPU_block/Calcul_U/FREG_5 (FF)
  Source Clock:      IP_block/RPW_block/phase_lock_loop/Qs_21 rising
  Destination Clock: IP_block/RPW_block/phase_lock_loop/Qs_21 rising

  Data Path: IP_block/CPU_block/Calcul_U/FREG_0 to IP_block/CPU_block/Calcul_U/FREG_5
                              Gate     Net
 Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------- ------------
  FDCE:C->Q             7   0.591   0.743  IP_block/CPU_block/Calcul_U/FREG_0
                                           (IP_block/CPU_block/Calcul_U/FREG_0)
  LUT3:I2->O            2   0.704   0.451  IP_block/CPU_block/Calcul_U/ALUnit/ai<0>19
                                           (IP_block/CPU_block/Calcul_U/ALUnit/ai<0>_map8)
  LUT4:I3->O            1   0.704   0.000  IP_block/CPU_block/Calcul_U/ALUnit/ai<0>44_F (N7834)
  MUXF5:I0->O           1   0.321   0.424  IP_block/CPU_block/Calcul_U/ALUnit/ai<0>44
                                           (IP_block/CPU_block/Calcul_U/ALUnit/ai<0>_map14)
  LUT4:I3->O            2   0.704   0.451  IP_block/CPU_block/Calcul_U/ALUnit/ai<0>86
                                           (IP_block/CPU_block/Calcul_U/ALUnit/ai<0>)
  LUT4_D:I3->LO         1   0.704   0.104  IP_block/CPU_block/Calcul_U/ALUnit/slices[0].slice/co29 (N8026)
  LUT4:I3->O            2   0.704   0.482  IP_block/CPU_block/Calcul_U/ALUnit/ci_2_mux000044
                                           (IP_block/CPU_block/Calcul_U/ALUnit/ci_2_mux0000_map13)
  LUT4_D:I2->O          7   0.704   0.712  IP_block/CPU_block/Calcul_U/ALUnit/ci_3_mux000044
                                           (IP_block/CPU_block/Calcul_U/ALUnit/ci_3_mux0000_map13)
  LUT4:I3->O            1   0.704   0.455  IP_block/CPU_block/Calcul_U/ALUnit/ai<5>70
                                           (IP_block/CPU_block/Calcul_U/ALUnit/ai<5>_map21)
  LUT4_L:I2->LO         1   0.704   0.135  IP_block/CPU_block/Calcul_U/ALUnit/slices[5].slice/gi
                                           (IP_block/CPU_block/Calcul_U/ALUnit/slices[5].slice/gi)
  LUT4:I2->O            3   0.704   0.566  IP_block/CPU_block/Calcul_U/ALUnit/slices[5].slice/Mxor_s_Result1
                                           (IP_block/CPU_block/Calcul_U/result<5>)
  LUT4:I2->O            8   0.704   0.836  Data_Bus<5>LogicTrst358 (Data_Bus<5>LogicTrst_map89)
  LUT4:I1->O           74   0.704   0.000  Data_Bus<5>LogicTrst365 (Data_Bus<5>)
  FDCE:D                    0.308          IP_block/CPU_block/Calcul_U/FREG_5
    ----------------------------------------
 Total                     14.323ns (8.964ns logic, 5.359ns route)
                                    (62.6% logic, 37.4% route)
```

Listing 1: Une partie du rapport final de synthèse

On note qu'avant de procéder à l'implémentation du projet, les différentes contraintes doivent être soigneusement éditées dans le fichier *ucf* du projet (*annexe 9*).

4.6.3 Implantation du système complet APTD

Dans cette partie, on arrive à la phase de validation finale du MCIP qui s'effectue par l'implantation physique du projet. L'environnement de Xilinx ISE a réussi à placer et router le système en respectant les contraintes de fonctionnement et a généré un taux d'utilisation de slices égale à 36% et une fréquence maximale égale à 56MHz (*listing 2*).

```
Design Summary Report:

 Number of External IOBs                   69 out of 232    29%

    Number of BUFGMUXs                      5 out of 24     20%
    Number of MULT18X18SIOs                 1 out of 20      5%
    Number of Slices                     1716 out of 4656   36%
       Number of SLICEMs                  598 out of 2328   25%


    The Delay Summary Report

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

    The AVERAGE CONNECTION DELAY for this design is:        1.905
    The MAXIMUM PIN DELAY IS:                               9.402
    The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   7.779

----------------------------------------------------------------------------------------------------------
  Constraint                                |  Check  | Worst Case |  Best Case | Timing |   Timing
                                            |         |    Slack   | Achievable | Errors |    Score
----------------------------------------------------------------------------------------------------------
  TS_clock = PERIOD TIMEGRP "clock" 20 ns H | SETUP   |    17.681ns|     2.319ns|       0|           0
  IGH 40%                                   | HOLD    |     1.618ns|            |       0|           0
----------------------------------------------------------------------------------------------------------
```

Listing 2: Une partie du rapport final de placement et routage

Par la suite, le chargement du design est effectué avec succès en utilisant la routine iMPACT.

La procédure de validation a commencé par l'affichage du message d'accueil qui se défile sur le LCD (*figure 4.14*) indiquant que le MCIP fonctionne correctement et que le programme s'est chargé convenablement à partir de la mémoire Strata Flash.

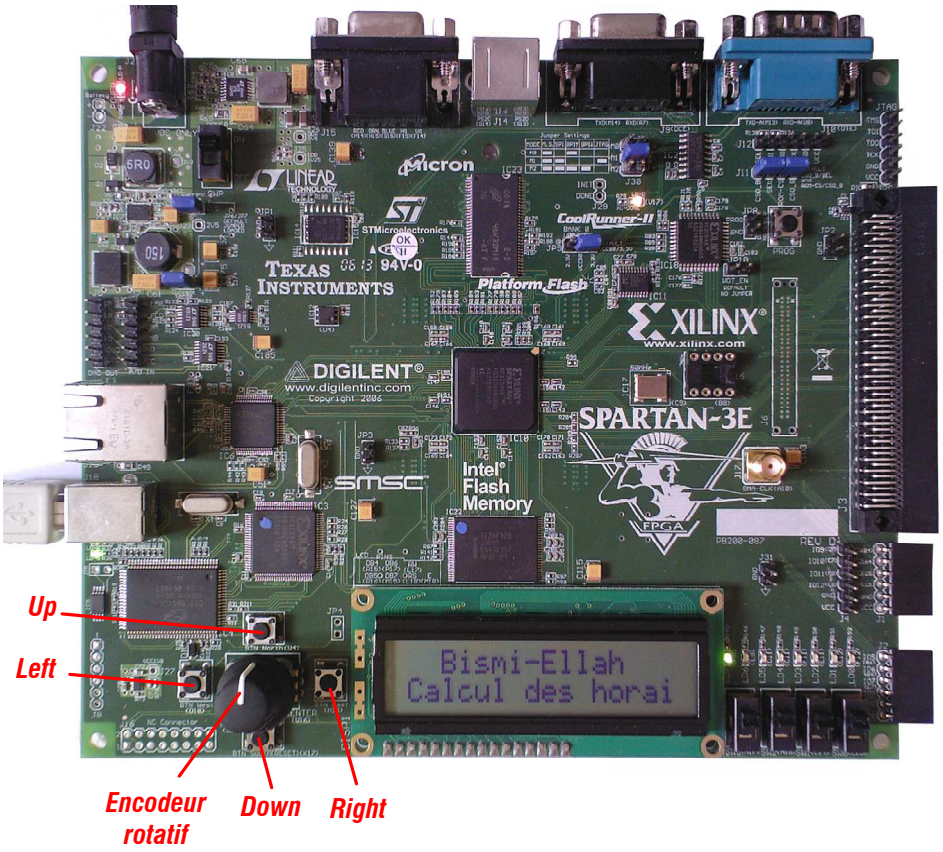


Figure 4.14: Affichage du message d'accueil

Les résultats produits par notre système complet sont présentés ci-dessous :

- Après l'étape d'initialisation, un simple appui sur la touche Up et le système répond par l'affichage de l'horaire calculé d'El-Fadjr sur le LCD pour la région de Sétif et pour la date 01/06/2008 (valeurs par défaut) (*figure 4.15*).

Figure 4.15: Exemple d'affichage d'un horaire de prière

- Le changement d'affichage d'une autre prière s'effectue par l'appui sur la touche Right (incrémenter) ou la touche Left (décrémenter). La *figure 4.16* montre les différents horaires de prière affichés sur le LCD.

Figure 4.16: Affichage des horaires pour la région de Sétif

- Le changement de la valeur de la date s'opère en tournant l'encodeur rotatif dédié dans l'un des sens (incrémentation ou décrémentation). La *figure 4.17* présente quelques captures produites pour différentes dates.

Figure 4.17: Affichage des horaires pour différentes dates

- La donnée relative à la région est changée en appuyant sur la touche Down. La *figure 4.18* présente l'affichage des régions suivantes : Alger, Makka et Medina.

Figure 4.18: Affichage des horaires pour différentes régions

- Un deuxième appui sur la touche Up mène au défilement du message 'A propos' (*figure 4.19*).

Figure 4.19: Affichage du message 'A propos'

Le mode opératoire est donc simple et l'application ADPT choisie est très utile. Pour un système complètement autonome, il serait utile de lier l'affichage continuel du temps réel du point géographique où le système se trouve et l'affichage des horaires de prière.

4.7. CONCLUSION

L'implantation du microcontrôleur est optimisée sur une FPGA de type XC3S500E Spartan-3E qui se situe au milieu de l'échelle des performances des FPGA Spartan.

L'exploitation du système de gestion des horaires de prière sur le Kit Spartan-3E est très simple en phase finale de développement mais il est nécessaire de noter que l'implantation du microcontrôleur sur FPGA et exécutant lui-même le programme de l'application APTD est passée par plusieurs phases d'expérimentations (synchronisation, adaptation électronique, affichage et adaptation adéquate des interruptions …etc).

A titre indicatif, le programme de calcul des horaires a nécessité :

- 11187 opérations 'Call-Return',
- plus de 82737 adressages indirects,
- 2619 opérations de multiplication,
- atteinte du 8^{e} étage du Stack …etc.

La fréquence maximale permise par le Spartan-3E implantant notre microcontrôleur est de 56MHz. Les caractéristiques techniques du MCIP peuvent être consultées sur le site web **msbcore.com** créé à cet effet. Un résumé est donné ci-après.

Overview

MCIP is an advanced RISC microcontroller hardware conform and software compatible with the *Microchip PIC18X* microcontroller. Its architectural features commonly found in RISC microprocessors are:

- Harvard architecture allow separate instruction and data busses with a 16-bit wide instruction word with the separate 8-bit wide data
- Long Word Instructions 16-bit
- Single Word Instructions
- Single Cycle Instructions except for program branches, which require two cycles
- Two stage Instruction Pipelining
- Reduced Instruction Set (72 Instructions)
- Register File Architecture: The register files/data memory can be directly or indirectly addressed. All special function registers, including the program counter, are mapped in the data memory
- Orthogonal (Symmetric) Instructions make it possible to carry out any operation on any register using any addressing mode
- 32 level-deep stack
- Multiple internal and external interrupts sources
- Sleep mode for low power consumption

General description

Memories:

- 2M addressable bytes of program memory
- 4K addressable bytes of data memory

Peripherals:

- **Four 8-bit I/O ports**
  - Software configurable with four TRIS registers
  - Three interrupt on PORTB
- **Timer0 and Timer1**
  - Software selectable operation as a timer or counter in both 8-bit or 16-bit modes
  - Readable and writable registers
  - Dedicated 8-bit, software programmable prescaler
  - Selectable clock source (internal or external)
  - Edge select for external clock
  - Interrupt-on-overflow
- **Watchdog**
  - 8-bit counter
  - Configurable time out period with dedicated 8-bit software programmable prescaler
  - Independent Watchdog clock input

- **Hardware multiplier 8x8**
  - Unsigned multiplication operation in single instruction cycle
  - Dedicated product register pair PRODH:PRODL
  - Make the core adapted to many applications for digital signal processing.
- **Interrupts**
  - Five interrupt sources:
    - External interrupt from the INT0 and INT1 (PORTB)
    - Change on RB7:RB4 pins
    - Timer0 and Timer1 overflow
  - Configurable Interrupt priority feature : low priority level and high priority level
  - Three dedicated registers to control interrupt operation
- **Table read**
  - Move bytes operation from the program memory space to the data RAM

Block diagram

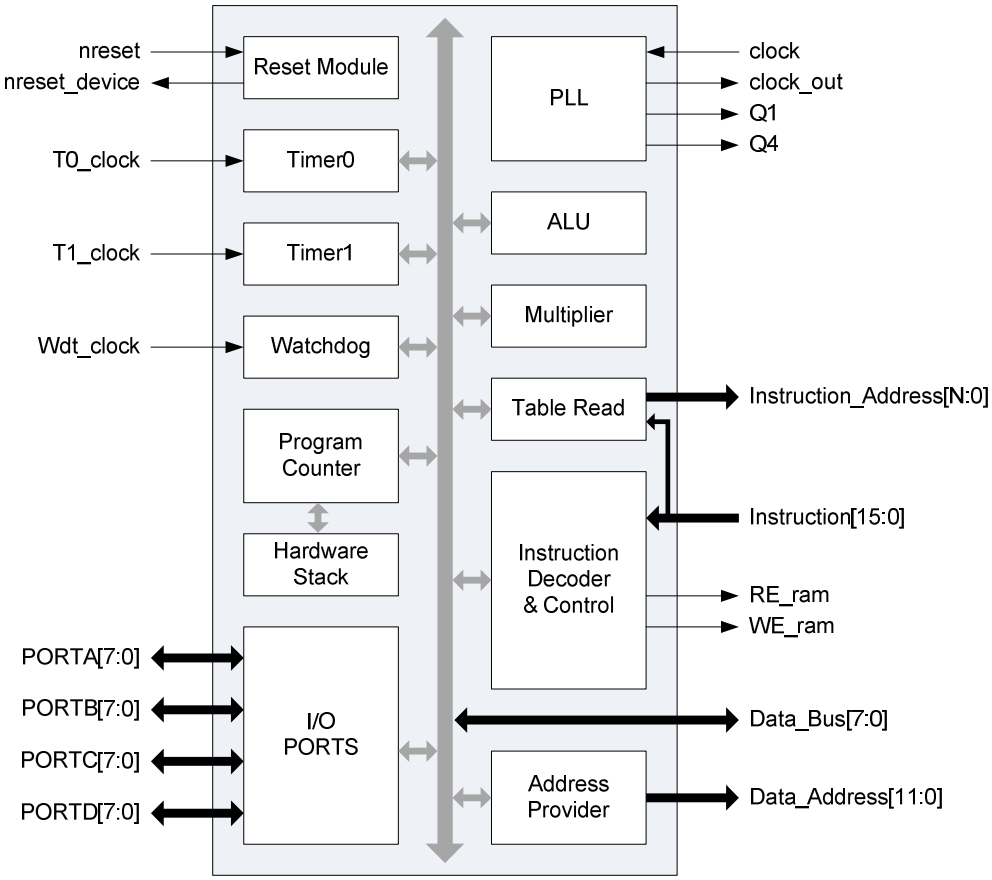


Figure 1: MCIP block diagram

MCIP can be fully customized to fulfill customer needs and easily adjusted to application dedicated requirements.

I/O signals

Table1: Core I/O signals

Name	*Direction*	*Description*
nreset	input	User reset
nreset_device	output	Global reset
clock	input	Global clock
T0_clock	input	Timer0 clock
T1_clock	input	Timer1 clock
Wdt_clock	input	Watchdog clock
clock_out	output	Clock output
Instruction [15:0]	input	Program memory bus
Instruction_address [N:0]*	output	Program memory address bus
Data_bus [7:0]	in-output	Data memory bus
Data_address [11:0]	output	Data memory address bus
RE_ram	output	Data memory read enable
WE_ram	output	Data memory write enable
Q1	output	Pipeline timing signal
Q4	output	Pipeline timing signal
PORTA [7:0]	in-output	Bidirectional I/O Port
PORTB [7:0]	in-output	Bidirectional I/O Port
PORTC [7:0]	in-output	Bidirectional I/O Port
PORTD [7:0]	in-output	Bidirectional I/O Port

* Instruction Address bus wide is generic (N=20 max).

Example implementation statistics

Table2: Example implementation using *Xilinx ISE 9.1* (with 32 stack level)

Device	*Speed grade*	*Slices (without memories)*	*Slices (256B Ram & 2KB Rom)*	*F max*
Virtex 5	-3	422	1495	106 MHz
Virtex 4	-12	869	1913	105 MHz
Virtexe 2	-6	870	1831	90 MHz
Virtex	-6	909	2015	44 MHz
Spartan 3E	-5	878	1766	56 MHz
Spartan 3	-5	863	1924	55 MHz
Spartan 2E	-7	919	2041	59 MHz
Spartan 2	-6	900	2028	52 MHz

Deliverables

- VHDL source code
- Documentation: detailed architecture description
- Verification: modules & global Testbenchs.

Ordering information

MSBcore
Address: 8, Ben Maiza Ahmed Street, Setif 19000, Algeria.
Fax: 213 36 930 177
E-mail: info@msbcore.com
Web site: www.msbcore.com

Conclusion générale

Les nouvelles techniques de conception de systèmes électroniques exigent un niveau d'abstraction élevé et une méthodologie très structurée pour permettre au concepteur de se consacrer uniquement à la conception sans se soucier des détails de mise en œuvre au niveau schématique et intégration. Cependant même à ce niveau d'abstraction, le concepteur se trouve confronté à la complexité toujours croissante sans la réutilisation de certaines des parties maîtrisées et de l'exploitation des IPs.

Développer un IP est désormais devenue une tâche importante et nécessaire. En effet une bibliothèque riche en IPs est une source et un moyen incontournables des concepteurs des circuits modernes de type SoC non seulement pour garantir l'efficacité et la rapidité de conception mais aussi pour assurer une évolution continuelle qui est jugée comme la force d'entrainement du développement de l'électronique.

Dans ce mémoire, nous avons étudié et proposé un IP soft de microcontrôleur dont la conception a bénéficié de plusieurs outils de développement et d'outils d'aide à la conception. La structuration considérée du modèle a prouvé son efficacité particulièrement lors de la vérification qui était une tâche longue, ennuyeuse et parfois même déprimante.

Dans ce travail, nous avons conçu un microcontrôleur complet sous l'environnement de Xilinx ISE pouvant travailler à des fréquences élevées (50 MHz sous des FPGA Spartan3E) et dont les principales caractéristiques techniques sont :

- Un microcontrôleur muni de 72 instructions réparties en 5 groupes,
- Sous forme d'un bloc configurable de 100 pins max,
- 4 Ports d'E/S de 8 bits chacun,
- Un multiplieur matériel 8x8 …etc.

L’implantation de notre microcontrôleur MCIP sur FPGA, du système d’affichage des horaires de prière APTD et leur expérimentation sur un kit de développement représentent les principales tâches étudiées et réalisées avec succès.

Les environnements de programmation et les outils technologiques, très avancés, exploités pour la mise au point de notre microcontrôleur MCIP, sous forme d’un IP compétitif à ceux du marché, montrent le standard élevé du travail accompli d’une part et le degré élevé de connaissances de l’électronique numérique exigé à chacune des tâches réalisées d’autre part.

Les bons résultats, consignés dans ce mémoire, attestent le bon degré de réussite du système conçu et confirment la validité de l’approche suivie.

Notre satisfaction ne serait comme même complète qu’une fois notre IP soit effectivement placé dans une des rubriques d’IP mondialement connues, procédure à laquelle il sera désormais nécessaire de passer par une action purement professionnelle de type technico-commerciale.

Ce pas de conception non négligeable affranchi, nous ouvre de larges perspectives au développement de systèmes embarqués dédiés aux applications grand-publiques.

La validation fonctionnelle du MCIP est passée par une phase très exhaustive de vérifications multiples où la gestion de la grande quantité d’informations générées a exigé non seulement l’appel à des techniques de suivi complexes mais aussi à une prise en charge de visualisations de beaucoup de signaux difficiles à interpréter. Une technique d’injection de fautes sous forme d’un programme générique au niveau comportemental nous a permis d’éviter le recours aux méthodes formelles de vérification.

Les résultats obtenus à travers l’application pratique développée, qui est sélectionnée pour sa sensibilité en termes de ressources exploitées au sein du microcontrôleur, représentent une bonne preuve du bon fonctionnement du produit développé.

Comme perspectives, il est souhaitable que le MCIP soit doté d’un mécanisme Boundary Scan pour améliorer sa testabilité et lui donner plus de chance de faire partie des composants constitutifs d’un SoC moderne et qu’il soit muni d’une interface générique pour la prise en charge des modes de communication les plus utilisés dans les systèmes électroniques avancés.

REFERENCES

[1] **Michael Keating** et **Pierre ricaud**, *"Reuse methodology manual for system-on-Chip designs"*, Kluwer Academic Publishers, 2002.

[2] **Thierry Schneider**, *"Méthodologie de design et techniques avancées"*, Dunod, 2001.

[3] **Damien Hubaux**, **Lotfi Guedria**, **Luc Vandendorpe**, **Michel Verleysen** et **Jean-Didier Legat**, *"Nouvelles méthodes de conception de systèmes électroniques intégrés"*, RENPAR'14 (14^{e} Rencontres francophones du parallélisme des architectures et des systèmes) Hamamet, Tunisie, 2002.

[4] **Semeria**, **L. Ghosh**, *"Methodology for hardware/software co-verification in C/C++"*, Proceedings of IEEE 2000 pp: 405-408.

[5] **Wilton S.J.E.** et **Saleh R.**, *"Programmable logic IP cores in SoC design: opportunities and challenges"*, proceeding of IEEE Custom Integrated Circuits, 2001, pp:63 – 66.

[6] **Andy Montador**, *"Verification Methodology for Standards-based IP & SOC"*, IP Based SoC Design Conference, 2005, France.

[7] **Mounir Benabdenbi**, *"Conception en vue de test de systèmes intégrés sur Silicium (SOC)"*, Thèse de Doctorat à l'université Paris VI, 2002.

[8] Site Web de **Michel Hubin**, *www.*pagesperso-orange.fr/michel.hubin

[9] **Stephen Brown** et **Jonathan Rose**, *"Architecture of FPGAs and CPLDs: A Tutorial"*, *IEEE Design and Test of Computers*, Vol. 13, No. 2, 1996, pp. 42-57.

[10] **Eric Shiflet** et **Lee Hansen,** *"FPGA partial reconfiguration mitigates variability"*, EE Times, Mars 2006.

[11] **Thomas D. Tessier**, *"Rethinking Your Verification Strategies for Multimillion-Gate FPGAs"*, Xilinx application notes, 2002.

[12] Site Web de Xilinx, *www.xilinx.com*

[13] Site Web de Altera, *www.altera.com*

[14] Site web de Lattice, *www.lattice.com*

[15] Site Web d' Actel, *www.actel.com*

[16] **Scott Hauck**, *"The Roles of FPGA's in Reprogrammable Systems"*, Proceedings of the IEEE, VOL. 86, NO. 4, April 1998, pp 615-638.

[17] *"XC4000E and XC4000X Series, data Sheet"*, Version 2001, *www.xilinx.com*

[18] *"Virtex 5 FPGA Family: User Guide"* , Version 2008, *www.xilinx.com*

[19] **Michel Aumieux**, *"Initiation au langage VHDL"*, Dunod Paris, 1999

[20] **Douglas L. Perry**, *"VHDL: programming by example"*, McGraw-Hill, 4e edition, 2002.

[21] **Volnei A. Pedroni**, *"Circuit design with VHDL"*, MIT Press, 5e edition, 2004.

[22] **Ron Landry**, *"FPGA Prototyping as a Verification Methodology"*, D&R Industry Articles, April 2006.

[23] *"Spartan-3E Starter Kit: Board User Guide"*, 2006, Xilinx Inc, http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf

[24] *"Spartan-3E FPGA Family: Complete Data Sheet"*, 2008, Xilinx Inc, http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

[25] Site Web de Microchip, *www.microchip.com*

[26] *"PICmicro 18C MCU Family Reference Manual"*, *www.microchip.com*

[27] Site Web de Design & Reuse, *www.design-reuse.com*

[28] *"Xilinx ISE 9.1 Software Manuals*", *www.xilinx.com*

[29] **Ben Cohen**, *"Forcing Signal Errors with VHDL"*, 2001, *www.vhdlcohen.com*

[30] **S. R. Seward** et **P. K. Lala**, *"Fault Injection for Verifying Testability at the VHDL Level"*, Proceeding of IEEE International test Conference, 2003, pp:131-137.

[31] **M. Abramovici**, **M. A. Breut** et **A. D. Friedman**, *"Digital systems testing and testable design"*, Computer Science Press, England, 1990.

[32] **LE SAINT CORAN**, Sourate An-Nisa, verset 103.

[33] Site web de Jordanian Astronomical Society, *www.jas.org.jo*

[34] **Mohamed Moudjdi Abd-Errasoul**, *"حساب مواقيت الصلاة الشرعية"*, www.nojumi.org/arabic/research/mmar/maoughat

[35] Site web de Mouhaddith, *www.muhaddith.org*

[36] Site web d'astronomie, *www.imcce.fr*

[37] *Datasheet de Intel Strata Flash PROM: "Numonyx™ Embedded Flash Memory", www.numonyx.com/Documents/Datasheets/316577_J3D_Monolithic_DS.p*