Université Mohamed Khider de Biskra
Faculté des Sciences et de la Technologie
Département de génie électrique

MÉMOIRE DE MASTER

Sciences et Technologies
Electronique
Electronique des systèmes embarqués

Réf. : ……..

Présenté et soutenu par :
NASRI ANISSA

Le : lundi 25 juin 2018

Conception et implémentation d'un microcontrôleur PIC16F84 sur FPGA

Jury :

Mme.	Hendaoui Mounira	MCB	Université de Biskra	Président
Mr.	Dhiabi Fathi	MAA	Université de Biskra	Encadreur
Mr.	Rahmani Nacer Eddine	MAA	Université de Biskra	Examinateur

Année universitaire : 2017 - 2018

الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي و البحث العلمي
Ministère de l'enseignement Supérieur et de la recherche scientifique

Université Mohamed Khider Biskra
Faculté des Sciences et de la Technologie
Département de Génie Electrique
Filière : Electronique

Option : Electronique de système embarqué

Mémoire de Fin d'Etudes
En vue de l'obtention du diplôme:

MASTER

Thème

Conception et implémentation d'un Microcontrôleur PIC16F84 sur FPGA

Présenté par :

NASRI ANISSA

Avis favorable de l'encadreur :

DHIABI FATHI

Avis favorable du Président du Jury

HENDAOUI MOUNIRA

Cachet et signature

الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي و البحث العلمي
Ministère de l'enseignement Supérieur et de la Recherche scientifique

Université Mohamed Khider Biskra
Faculté des Sciences et de la Technologie
Département de Génie Electrique
Filière : Electronique
Option : Electronique des systèmes embarqués

Thème :

Conception et implémentation d'un Microcontrôleur PIC16F84 sur FPGA

Proposé par : *DHIABI FATHI*
Dirigé par : *DHIABI FATHI*

RESUME

Le but du projet de fin d'étude est la mise en œuvre du circuit électronique PIC16F84 dans le circuit programmable FPGA du constructeur ALTERA CYCLONE II. Cette FPGA peut être programmée par deux langages de programmation différents, VHDL et VERILOG. La méthodologie de conception est la suivante: concevoir un simple compilateur pour la verification et la conversion des instructions écrites en langage assembleur vers le code hexadecimal. Ensuite implémenter un programme de cette structure afin de pouvoir mettre en œuvre quelques fonctions de PIC16F84. Nous avons utilisé le langage VHDL dans l'nvirennement QUARTUS II 7.2 pour implémenter le simulateur de programme PIC16F84 sur FPGA.

ملخص

الهدف من مشروع نهاية الدراسة هو تنفيذ الدارة الإلكترونية الميكرومراقبPIC 16F84 في الدارة المنطقية القابلة للبرمجةFPGA الخاصة بالشركة المصنّعة التيرا سيكلون2 حيث يمكن برمجة هذه الدارة بواسطة لغتي برمجة مختلفتين هما VHDL اوVERILOG, ومنهجية التصميم تكون على النحو التالي :

تصميم مترجم بسيط لفحص و تحويل تعليمات التجميع المكتوبة الى الشيفرة السداسي عشرة ثم تنفيذ برنامج من هذا الهيكل من أجل تنفيذ بعض وظائف الميكرومراقب حيث استخدمنا لغة VHDL في برنامج الكوارتز(Quartus II 7.2) لتنفيذ برنامج محاكاة الميكرومراقب على الدارة المنطقية القابلة للبرمجة.


بسم الله الرحمن الرحيم

Dédicaces

Je dédie ce modeste travail

A ma mère Naima, que dieu me le garde

Et mon père Fateh,

A ma grand-mère Chahla

Pour Leurs amour, leurs bonté, leurs sacrifice,

Leurs encouragements perpétuels

A mon frère Rafik et ma sœur Ahlème

A tous mes tentes, et surtout tente Houria et toute ma famille,

A mon encadreur Mr Dhiabi Fathi

A tous mes collègues Ismail et Imen et Oussama et Naima

Et mes amis Samira et Aicha et Samah et Imen et Hafida et partout

A tous ceux qui ont sacrifié leur temps pour la science

Et à tous ceux qui utilisent la science pour le bien

Et la prospérité de l'humanité

ANISSA NASRI

Remerciement

Je remercie Dieu Allah le tout puissant de m'avoir donné courage et patience, qui m'a permis d'accomplir ce modeste travail.

Je tiens en premier à exprimer ma grande gratitude envers mon encadreur Mr. **Dhiabi Fathi**, qui m'a apporté son aide et ses conseils précieux et de m'avoir inspiré ce sujet et suivi de très près sans ménager son temps ni si efforts, qu'il trouve dans ce travail mon témoignage respectueux.

Je remercie Mme. **Hendaoui Mounira**, maitre assistance B au département d'électronique, pour avoir bien voulu accepter de présider le jury.

Je remercie Mr. **Rahmani Nacer Eddine**, maître de conférences A d'avoir accepté d'examiner ce mémoire et d'être membre du jury.

Et je remercie encours Mr. **Bkhouche Khaled,** pour aide à la correction de ce travail.

Je tiens à exprimer mes sincères remerciements pour mes parents, ma famille et toute personne ayant participé de loin ou de prés pour l'aboutissement de ce modeste travail.

Liste des figures

Liste des Abréviations

PIC : Périphéral Interface Controller ou Contrôleur d'interface périphérique.

RISC : Reduced Instructions Set Construction.

CISC : Complex Instructions Set Construction.

ALU : unité arithmétique et logique.

SFRs : Special Function Registers.

FPGA : Field Programmable Gâte Arrays ou réseaux logiques programmables.

PLD : programmable logic device.

LCA : Logic Cell Array.

CLB : Configurable Logic Block.

LC : Logic Cell.

LE : Logic Element.

IOB : Input Output Block.

PIP : Programme InterConnect Points.

ASIC : Application Specific Integrated Circuit.

VHDL : VHSIC Hardware Description Langage.

VHSIC : Very High Scale Integrated Circuit.

IEEE : Institut of Electrical and Electronics Engineers.

Sommaire

CHAPITRE III : Implantation PIC 16F84 SUR FPGA

Introduction générale

Les FPGA (Field Programmable Gâte Arrays ou "réseaux logiques programmables") sont des composants entièrement reconfigurables ce qui permet de les reprogrammer a volonté afin d'accélérer notablement certaines phases de calculs. L'avantage de ce genre de circuit est sa grande souplesse qui permet de les réutiliser à volonté dans des algorithmes différents en un temps très court. Le progrès de ces technologies permet de faire des composants toujours plus rapides et a plus haute intégration, ce qui permet de programmer des applications importantes. Cette technologie permet d'implanter un grand nombre d'applications et offre une solution d'implantation matérielle à faible cout pour des compagnies de taille modeste pour qui, le cout de développement d'un circuit intègre spécifique implique un trop lourd investissement.

Les microcontrôleurs PIC (ou PIC micro dans la terminologie du fabricant) forment une famille de microcontrôleurs de la société Micro chip. Ces microcontrôleurs sont dérivés du PIC16F84 développé à l'origine par la division microélectronique de General Instruments. Le nom PIC n'est pas officiellement un acronyme, bien que la traduction en « Périphéral Interface Controller » (Contrôleur d'interface périphérique) soit généralement admise. Cependant, à l'époque du développement du PIC16F84 par General Instruments, PIC était un acronyme de « Programmable Intelligent Computer » ou «Programmable Integrated Circuit».

L'objectif de ce travail et d'implémenter quelques fonctionnalités du PIC16F84 dans le FPGA Altera Cyclon 2. Il ne s'agit pas vous vous en doutez de faire rentrer un pic de force dans un malheureux FPGA. Il s'agit de copier la façon dont est faite un PIC et de le programmer dans un FPGA. On pourra ensuite programmer et utiliser le FPGA de la même façon qu'un PIC. En effet un PIC n'est qu'un assemblage assez complexe de portes logiques mis dans un boitier. Un FPGA est quant-a-lui un composant rempli de portes logiques en attente d'être reliées.

L'outil utilisé pour la programmation de FPGA est le langage VHDL dans l'environnement du logiciel Quartus version 7.2. Le VHDL pouvant être vu comme un langage de description de schémas logiques il est aisé à comprendre que si on dispose de la description VHDL du pic, il est possible de le programmer dans un FPGA.

Ce mémoire intitulé « Conception et Implémentation d'un microcontrôleur pic16f84 sur un FPGA » se divise en une introduction générale, trois chapitres et une conclusion générale. Il est organisé de la manière qui suit :

Introduction générale

Le premier chapitre : le microcontrôleur PIC16F84.

Le deuxième chapitre : FPGA (Field Programmable Gâte Arrays)/ le langage VHDL.

Le troisième chapitre : l'implémentation PIC16F84 sur FPGA.

Conclusion générale

Chapitre I

LE MICROCONTROLEUR PIC16F84

I .1. Introduction:

Un microcontrôleur est un circuit intégré qui rassemble aux éléments essentiels d'un Ordinateur : processeur, mémoires (mémoire morte pour le programme, mémoire vive pour les données), unités de périphériques et interfaces d'entrées/sorties. Les microcontrôleurs se caractérisent par un plus haut degré d'intégration, une plus faible consommation électrique, une vitesse de fonctionnement plus faible et un coût réduit par rapport aux microprocesseurs polyvalents utilisés dans les ordinateurs personnels.

Par rapport à des systèmes électroniques à base de microprocesseurs et autres composants séparés, les microcontrôleurs permettent de diminuer la taille, la consommation électrique et le coût des produits. Ils ont ainsi permis de démocratiser l'utilisation de l'informatique dans un grand nombre de produits et de procédés.

Les microcontrôleurs sont fréquemment utilisés dans les systèmes embarqués, comme les contrôleurs des moteurs automobiles, les télécommandes, les appareils de bureau, l'électroménager, les jouets, la téléphonie mobile, etc. [1]

I.2. Qu'est-ce qu'un PIC :

Un PIC est un microcontrôleur de chez Micro-chip. Ses caractéristiques principales sont: Séparation des mémoires de programme et de données (architecture Harvard) : On obtient ainsi une meilleure bande passante et des instructions et des données pas forcément codées sur le même nombre de bits. Communication avec l'extérieur seulement par des ports : il ne possède pas de bus d'adresses, de bus de données et de bus de contrôle comme la plupart des microprocesseurs.

Utilisation d'un jeu d'instructions réduit, d'où le nom de son architecture : RISC (Reduced Instructions Set Construction). Les instructions sont ainsi codées sur un nombre réduit de bits, ce qui accélère l'exécution (1 cycle machine par instruction sauf pour les sauts qui requirent 2 cycles). En revanche, leur nombre limité oblige à se restreindre à des instructions basiques, contrairement aux systèmes d'architecture

CISC (Complex Instructions Set Construction) qui proposent plus d'instructions donc codées sur plus de bits mais réalisant des traitements plus complexes.

Il existe trois familles de PIC :

- ✓ Base-Line : Les instructions sont codées sur 12 bits
- ✓ Mid-Line : Les instructions sont codées sur 14 bits
- ✓ High-End : Les instructions sont codées sur 16 bits

Un PIC est identifié par un numéro de la forme suivant : xx(L) XXyy –zz

- xx : Famille du composant (12, 14, 16, 17, 18)

- L : Tolérance plus importante de la plage de tension

- XX : Type de mémoire de programme

C - EPROM ou EEPROM

CR - PROM

F - FLASH

- yy : Identification

- zz : Vitesse maximum du quartz

Nous utiliserons un PIC 16F84 –10, soit :

- 16 : Mid-Line

- F : FLASH

- 84 : Type

- 10 : Quartz à 10 MHz au maximum. [2]

I.3. PIC 16F84 :

Device	Program Memory (words)	Data RAM (bytes)	Data EEPROM (bytes)	Max. Freq (MHz)
PIC16F83	512 Flash	36	64	10
PIC16F84	1 K Flash	68	64	10
PIC16CR83	512 ROM	36	64	10
PIC16CR84	1 K ROM	68	64	10

Figure I.1: Liste des composants présentés dans la documentation n°DS30430C. [3]

Il s'agit d'un microcontrôleur 8 bits à 18 pattes. La documentation technique N °DS30430C porte sur plusieurs composants (Figure I.1). [2]

Caractéristiques du PIC16F84:

Fonctionne à 10 Mhz maximum. (20 Mhz pour le 16F84A)

- 35 instructions (composant RISC),
- 1Ko de mémoire (1024 mots de 14 bits) Flash pour le programme,
- 68 octets de RAM,
- 64 octets de d'EEprom,
- 1 compteur/ timer de 8 bits,
- 1 Watch dog,
- 4 sources d'interruption,
- 13 entrées/sorties configurables individuellement,
- Mode SLEEP. [4]

I.3:a. Brochage et fonction des pattes :

La Figure I.2 montre le brochage du circuit. Les fonctions des pattes sont les suivantes:

- VSS, VDD : Alimentation

- OSC1, 2 : Horloge

- RA0-4 : Port A

- RB0-7 : Port B

- T0CKL : Entrée de comptage

- INT : Entrée d'interruption

- MCLR : Reset : 0V

Choix du mode programmation : 12V - 14V exécution : 4.5V - 5.5V. [2]

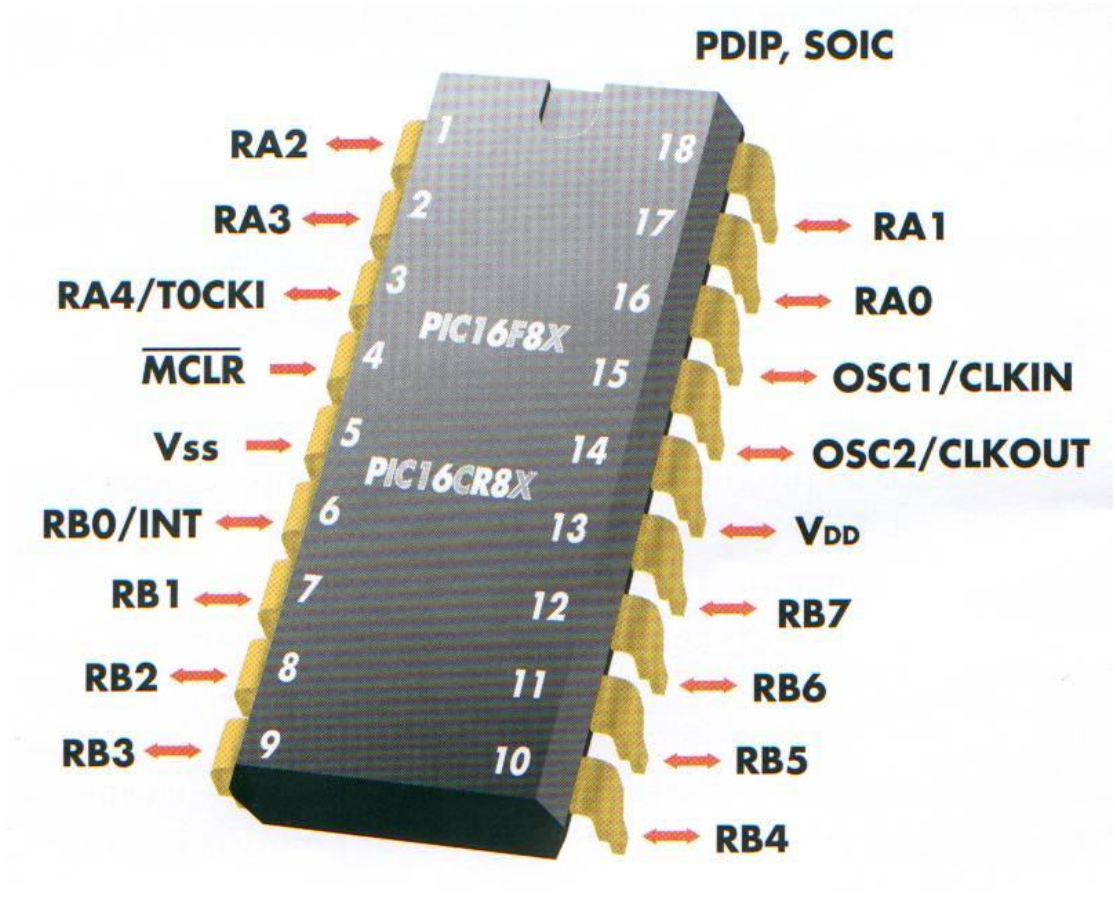


Figure I.2: Brochage du circuit. [5]

I.3:b. Architecture générale :

La Figure I.3 présente l'architecture générale du circuit. Il est constitué des éléments suivants:

- un système d'initialisation à la misUe sous tension (power-up timer, …)

- un système de génération d'horloge à partir du quartz externe (timing génération)

- une unité arithmétique et logique (ALU)

- une mémoire flash de programme de 1k "mots" de 14 bits

- un compteur de programme (program counter) et une pile (stack)

- un bus spécifique pour le programme (program bus)

- un registre contenant le code de l'instruction à exécuter

- un bus spécifique pour les données (data bus)

- une mémoire RAM contenant

- les SFR

- 68 octets de données

- une mémoire EEPROM de 64 octets de données

- 2 ports d'entrées/sorties

- un compteur (timer)

- un chien de garde (watchdog). [2]

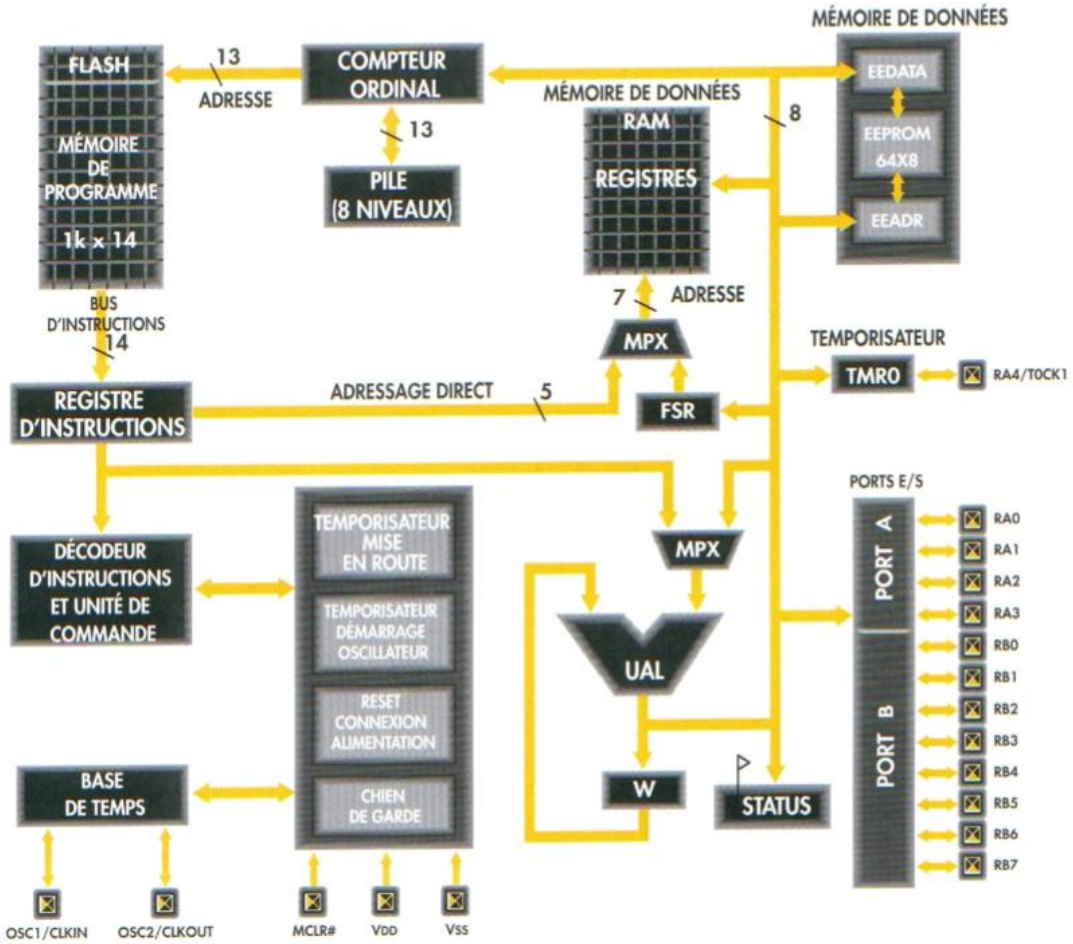


Figure I.3: Architecture générale du PIC 16F8X. [5]

I.4. Organisation de la mémoire :

Le PIC contient de la mémoire de programme et de la mémoire de données. La structure Harvard des PICs fournit un accès séparé à chacune. Ainsi, un accès aux deux est possible pendant le même cycle machine. [2]

I.4:a. Mémoire de programme :

C'est elle qui contient le programme à exécuter. Ce dernier est téléchargé par liaison série. La Figure I.4 montre l'organisation de cette mémoire. Elle contient 1k "mots" de 14 bits dans le cas du PIC 16F84, même si le compteur de programme (PC) de 13 bits peut en adresser 8k. Il faut se méfier des adresses images ! L'adresse 0000h contient le vecteur du reset, l'adresse 0004h l'unique vecteur d'interruption du PIC.
La pile contient 8 valeurs. Comme le compteur de programme, elle n'a pas d'adresse dans la plage de mémoire. Ce sont des zones réservées par le système. [2]

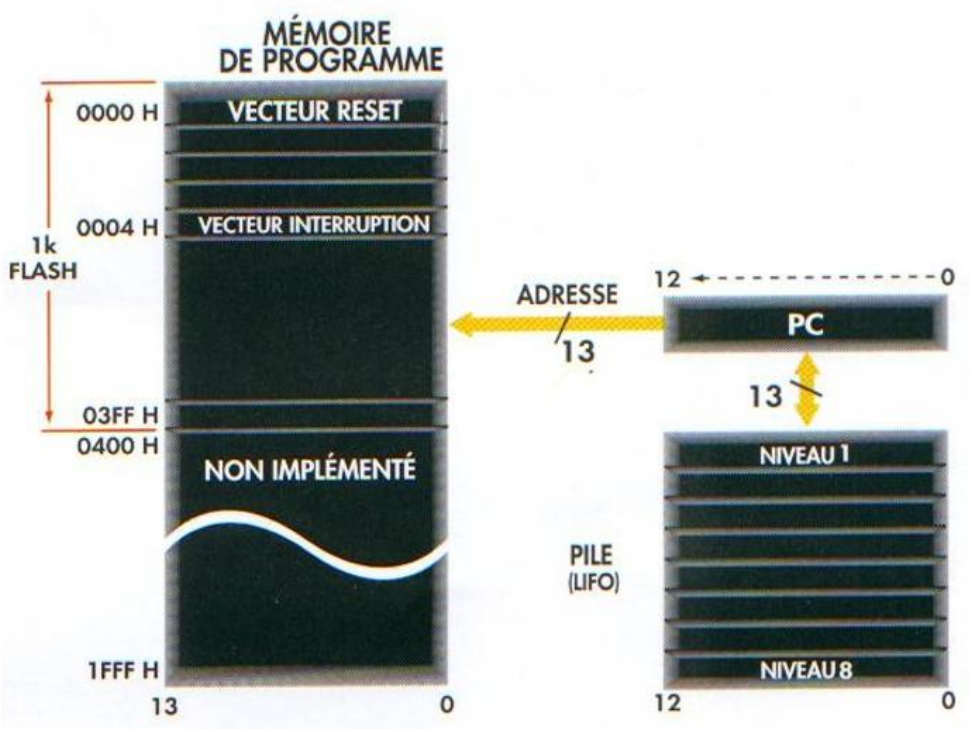


Figure I.4: Organisation de la mémoire de programme et de la pile. [5]

I.4:b. Mémoire de données :

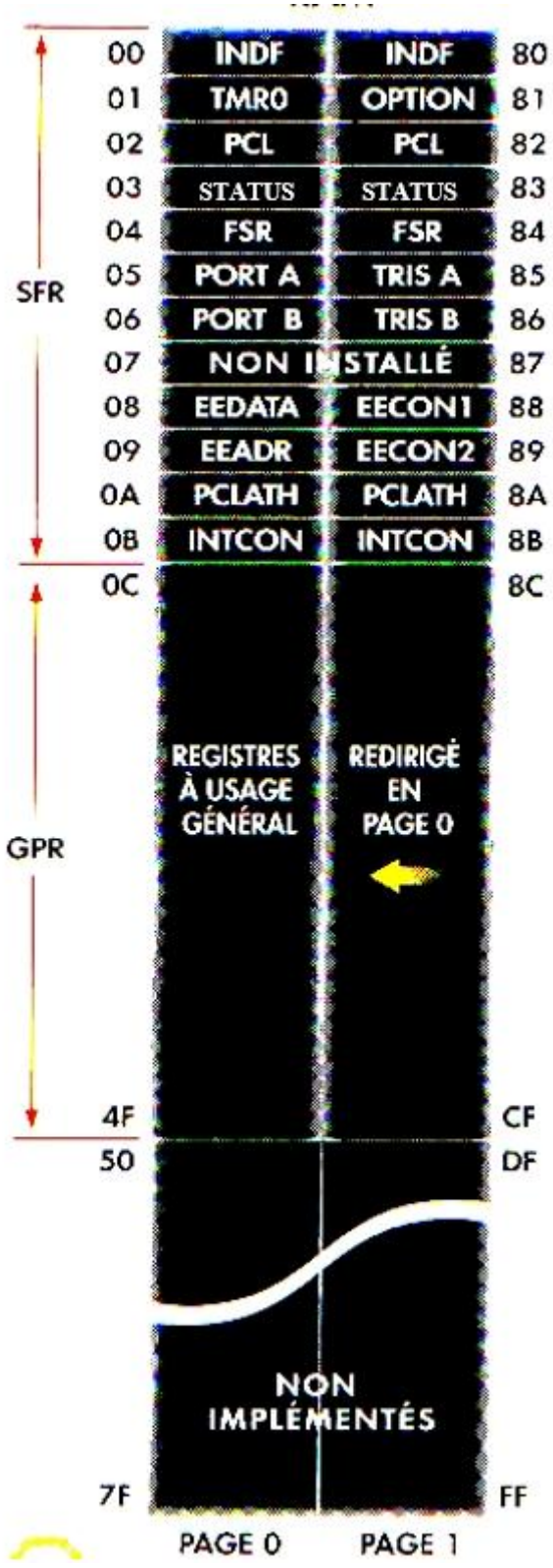


Figure I.5: Organisation de la mémoire de données. [5]

Elle se décompose en deux parties de RAM (Figure I.5) et une zone EEPROM. La première contient les SFRs (Special Function Registers) qui permettent de contrôler les opérations sur le circuit. La seconde contient des registres généraux, libres pour l'utilisateur. La dernière contient 64 octets.

Comme nous le verrons dans le paragraphe IV, les instructions orientées octets ou bits contiennent une adresse sur 7 bits pour désigner l'octet avec lequel l'instruction doit travailler. D'après la Figure I.5, l'accès au registre TRISA d'adresse 85h, par exemple, est impossible avec une adresse sur 7 bits. C'est pourquoi le constructeur a défini deux banques. Le bit RP0 du registre d'état (STATUS.5) permet de choisir entre les deux. Ainsi, une adresse sur 8 bits est composée de RP0 en poids fort et des

7 bits provenant de l'instruction à exécuter. [2]

I.4:b.1- Registres généraux :

Ils sont accessibles soit directement soit indirectement à travers les registres FSR et INDF. [2]

I.4:b.2- Registres spéciaux - SFRs :

Ils permettent la gestion du circuit. Certains ont une fonction générale, d'autres une fonction spécifique attachée à un périphérique donné. La Figure I.6 donne la fonction de chacun des bits de ces registres. Ils sont situés de l'adresse 00h à l'adresse 0Bh dans la banque 0 et de l'adresse 80h à l'adresse 8Bh dans la banque 1.Les registres 07h et 87h n'existent pas.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other resets (Note 3)
Bank 0											
00h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- ----	---- ----
01h	TMR0	8-bit real-time clock/counter								xxxx xxxx	uuuu uuuu
02h	PCL	Low order 8 bits of the Program Counter (PC)								0000 0000	0000 0000
03h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
04h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
05h	PORTA	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	---u uuuu
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	uuuu uuuu
07h		Unimplemented location, read as '0'								---- ----	---- ----
08h	EEDATA	EEPROM data register								xxxx xxxx	uuuu uuuu
09h	EEADR	EEPROM address register								xxxx xxxx	uuuu uuuu
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC (1)					---0 0000	---0 0000
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
Bank 1											
80h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- ----	---- ----
81h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
82h	PCL	Low order 8 bits of Program Counter (PC)								0000 0000	0000 0000
83h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
84h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
85h	TRISA	—	—	—	PORTA data direction register					---1 1111	---1 1111
86h	TRISB	PORTB data direction register								1111 1111	1111 1111
87h		Unimplemented location, read as '0'								---- ----	---- ----
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	---0 q000
89h	EECON2	EEPROM control register 2 (not a physical register)								---- ----	---- ----
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC (1)					---0 0000	---0 0000
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u

Legend: x = unknown, u = unchanged, - = unimplemented read as '0', q = value depends on condition.
Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a slave register for PC<12:8>. The contents of PCLATH can be transferred to the upper byte of the program counter, but the contents of PC<12:8> is never transferred to PCLATH.
2: The $\overline{TO}$ and $\overline{PD}$ status bits in the STATUS register are not affected by a $\overline{MCLR}$ reset.
3: Other (non power-up) resets include: external reset through $\overline{MCLR}$ and the Watchdog Timer Reset.

Figure I.6: Description des SFR. [3]

INDF (00h - 80h) : Utilise le contenu de FSR pour l'accès indirect à la mémoire.

TMR0 (01h) : Registre lié au compteur.

PCL (02h - 82h) : Contient les poids faibles du compteur de programmes (PC). Le registre PCLATH (0Ah-8Ah) contient les poids forts.

STATUS (03h - 83h) : Il contient l'état de l'unité arithmétique et logique ainsi que les bits de sélection des banques (Figure I.7).

FSR (04h - 84h) : Permet l'adressage indirect

PORTA (05h) : Donne accès en lecture ou écriture au port A, 5 bits. Les sorties sont à drain ouvert. Le bit 4 peut être utilisé en entrée de comptage.

PORTB (06h) : Donne accès en lecture ou écriture au port B. Les sorties sont à drain ouvert. Le bit 0 peut être utilisé en entrée d'interruption. EEDATA (08h) : Permet l'accès aux données dans la mémoire EEPROM.

EEADR (09h) : Permet l'accès aux adresses de la mémoire EEPROM.

PCLATCH (0Ah - 8Ah) : Donne accès en écriture aux bits de poids forts du compteur de programme.

INTCON (0Bh - 8Bh) : Masque d'interruptions.

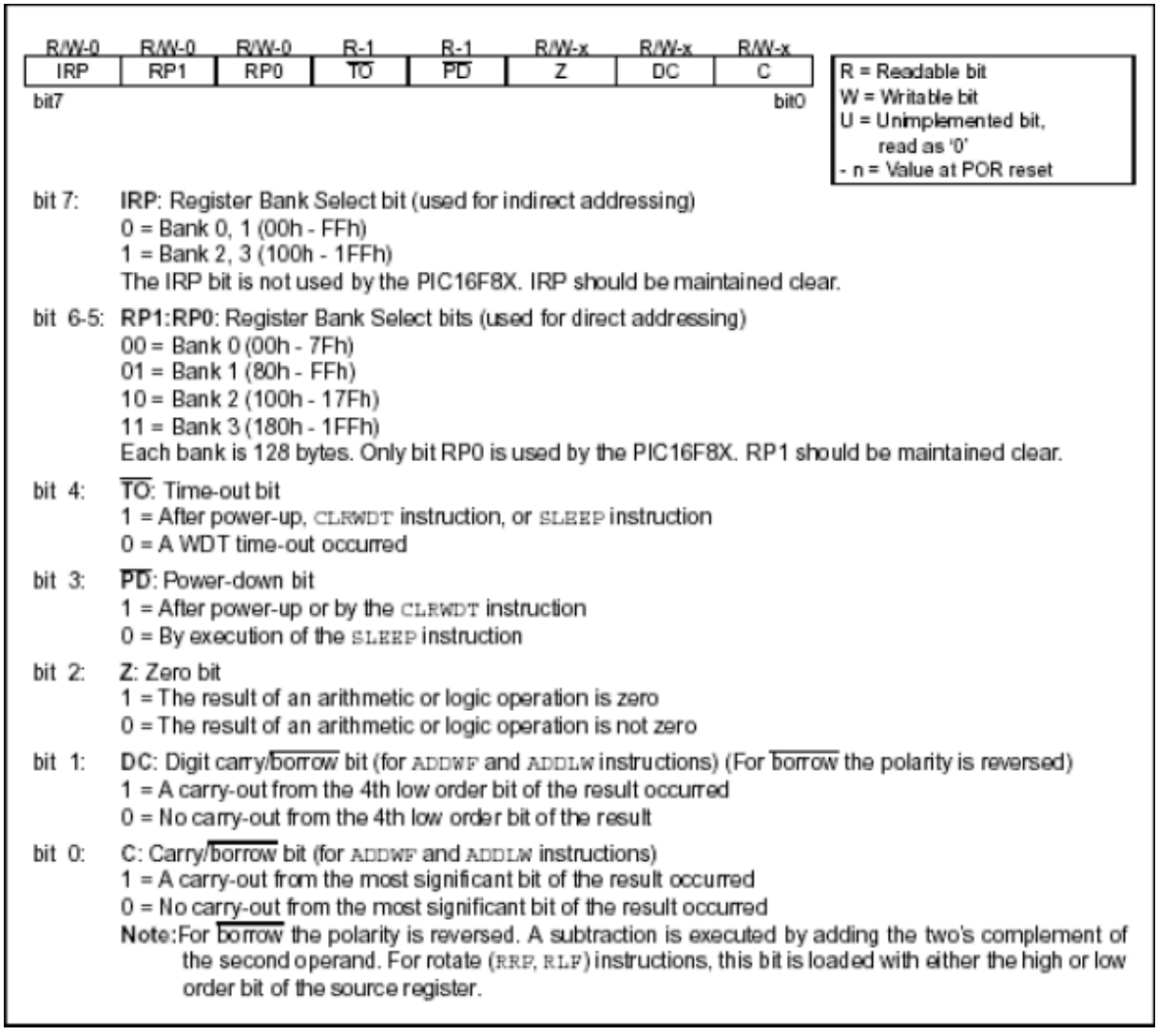
OPTION_REG (81h) : Contient des bits de configuration pour divers périphériques.

TRISA (85h) : Indique la direction (entrée ou sortie) du port A.

TRISB (86h) : Indique la direction (entrée ou sortie) du port B.

EECON1 (88h) : Permet le contrôle d'accès à la mémoire EEPROM.

EECON2 (89h) : Permet le contrôle d'accès à la mémoire EEPROM. [2]



R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: **IRP**: Register Bank Select bit (used for indirect addressing)
0 = Bank 0, 1 (00h - FFh)
1 = Bank 2, 3 (100h - 1FFh)
The IRP bit is not used by the PIC16F8X. IRP should be maintained clear.

bit 6-5: **RP1:RP0**: Register Bank Select bits (used for direct addressing)
00 = Bank 0 (00h - 7Fh)
01 = Bank 1 (80h - FFh)
10 = Bank 2 (100h - 17Fh)
11 = Bank 3 (180h - 1FFh)
Each bank is 128 bytes. Only bit RP0 is used by the PIC16F8X. RP1 should be maintained clear.

bit 4: $\overline{TO}$: Time-out bit
1 = After power-up, CLRWDT instruction, or SLEEP instruction
0 = A WDT time-out occurred

bit 3: $\overline{PD}$: Power-down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit 2: **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1: DC: Digit carry/$\overline{borrow}$ bit (for ADDWF and ADDLW instructions) (For $\overline{borrow}$ the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0: C: Carry/$\overline{borrow}$ bit (for ADDWF and ADDLW instructions)
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result occurred
Note: For $\overline{borrow}$ the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low order bit of the source register.

Figure I.7: Registre d'étai du PIC - STATUS. [3]

I.4:b.3-Mémoire EEPROM :

Le PIC possède une zone EEPROM de 64 octets accessibles en lecture et en écriture par le programme. On peut y sauvegarder des valeurs, qui seront conservées même si l'alimentation est éteinte, et les récupérer lors de la mise sous tension. Leur accès est spécifique et requiert l'utilisation de registres dédiés. La lecture et l'écriture ne peut s'exécuter que selon des séquences particulières décrite au paragraphe. [2]

I.5. Modes d'adressages :

On ne peut pas concevoir un programme qui ne manipule pas de données. Il existe trois grands types d'accès à une donnée ou modes d'adressage :

- Adressage immédiat : La donnée est contenue dans l'instruction.
- Adressage direct : La donnée est contenue dans un registre.
- Adressage indirect: L'adresse de la donnée est contenue dans un pointeur. [2]

I.5:a. Adressage immédiat :

La donnée est contenue dans l'instruction.

Exemple : movlw 0xC4 ; Transfert la valeur 0xC4 dans W. [2]

I.5:b. Adressage direct :

La donnée est contenue dans un registre. Ce dernier peut être par un nom (par exemple W) ou une adresse mémoire.

Exemple : movf 0x2B, 0 ; Transfert dans W la valeur contenue à l'adresse 0x2B**.**

! L'adresse 0x2B peut correspondre à 2 registres en fonction de la banque choisie (Figure I.8). Le bit RP0 permet ce choix, le bit RP1 étant réservé pour les futurs systèmes à 4 banques. [2]

I.5:c. Adressage indirect :

L'adresse de la donnée est contenue dans un pointeur. Dans les PIC, un seul pointeur est disponible pour l'adressage indirect : FSR. Contenu à l'adresse 04h dans les deux banques, il est donc accessible indépendamment du numéro de banque. En utilisant l'adressage direct, on peut écrire dans FSR l'adresse du registre à atteindre. FSR contenant 8 bits, on peut atteindre les deux banques du PIC 16F84. Pour les PIC contenant quatre banques, il faut positionner le bit IRP du registre d'état qui sert alors de $9^{ème}$ bit d'adresse (Figure I.8). [2]

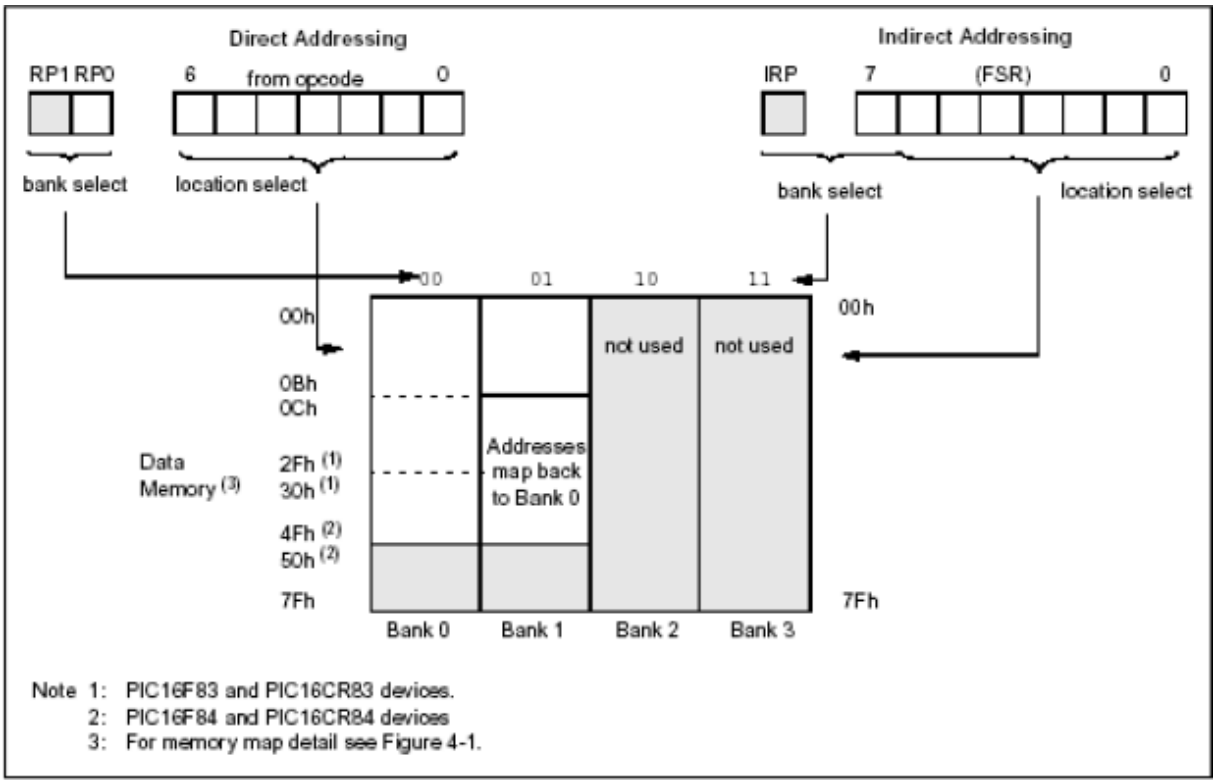


Figure I.8: Adressages direct et indirect à la mémoire de données. [3]

L'accès au registre d'adresse contenue dans FSR se fait en utilisant le registre INDF. Il se trouve à l'adresse 0 dans les deux banques. Il ne s'agit pas d'un registre physique.

On peut le voir comme un autre nom de FSR, utilisé pour accéder à la donnée elle même, FSR servant à choisir l'adresse.

Exemple : movlw 0x1A ; Charge 1Ah dans W

Movwf FSR ; Charge W, contenant 1Ah, dans FSR

Movf INDF, 0 ; Charge la valeur contenue à l'adresse 1Ah dans W

I.6. Ports d'entrées/Sorties :

Le PIC 16F84 est doté de deux ports d'entrées/Sorties appelés PortA et PortB. [2]

I.6:a. Port A :

Il comporte 5 pattes d'entrée/sortie bidirectionnelles, notées RAx avec x= {0, 1, 2, 3, 4} sur le brochage du circuit (Figure I.2). Le registre PORTA, d'adresse 05h dans la banque 0, permet d'y accéder en lecture ou en écriture. Le registre TRISA, d'adresse 85h dans la banque 1, permet de choisir le sens de chaque patte (entrée ou sortie) : un bit à 1 positionne le port en entrée, un bit à 0 positionne le port en sortie.

La Figure I.9 donne le câblage interne d'une patte du port A :

- "Data Latch" : Mémorisation de la valeur écrite quand le port est en sortie.
- "TRIS Latch" : Mémorisation du sens (entrée ou sortie) de la patte.

- "TTL input buffer" : Buffer de lecture de la valeur du port. La lecture est toujours réalisée sur la patte, pas à la sortie de la bascule d'écriture.
- Transistor N : En écriture : Saturé ou bloqué suivant la valeur écrite.

En lecture : Bloqué.

- Transistor P : Permet d'alimenter la sortie. [2]

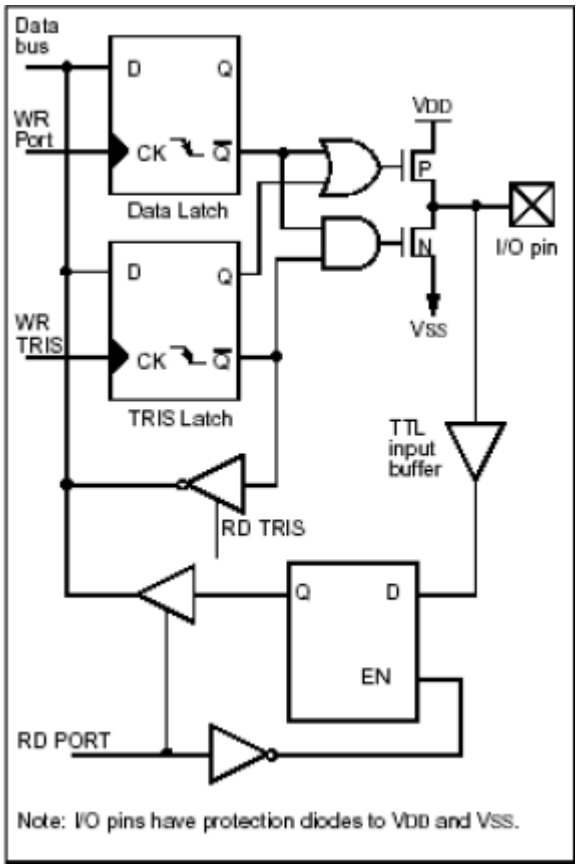


Figure I.9: Câblage interne d'une patte du port A. [3]

La patte RA4 peut aussi servir d'entrée de comptage pour le timer0.

I.6:b. Port B :

Il comporte 8 pattes d'entrée/sortie bidirectionnelles, notées RBx avec X = {0, 1, 2, 3, 4, 5, 6 ,7} sur le brochage du circuit (Figure I.2).

Le registre PORTB, d'adresse 06h dans la banque 0, permet d'y accéder en lecture ou en écriture. Le registre TRISB, d'adresse 86h dans la banque 1, permet de choisir le sens de chaque patte (entrée ou sortie) : un bit à 1 positionne le port en entrée, un bit à 0 positionne le port en sortie.

Le câblage interne d'une porte du port B ressemble beaucoup à celui du port A (Figure I.10). On peut noter la fonction particulière pilotée par le bit RBPU (OPTION_REG.7) qui permet d'alimenter (RBPU=0) ou non (RBPU=1) les sorties.

Les quatre bits de poids fort (RB7-RB4) peuvent être utilisés pour déclencher une interruption sur changement d'état.

RB0 peut aussi servir d'entrée d'interruption externe. [2]

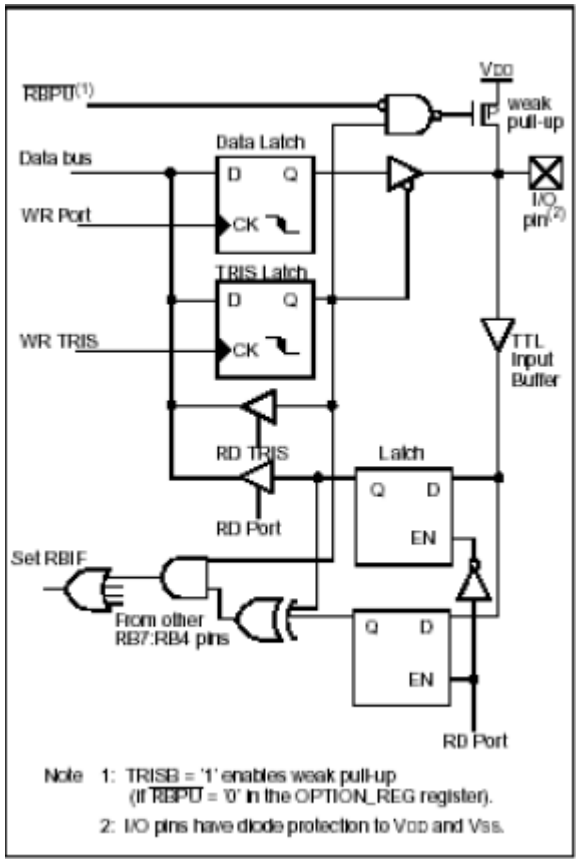


Figure I.10: Câblage interne d'une patte du port B. [3]

I.7. Compteur :

Le PIC 16F84 est doté d'un compteur 8 bits. La Figure I.10 en donne l'organigramme. [2]

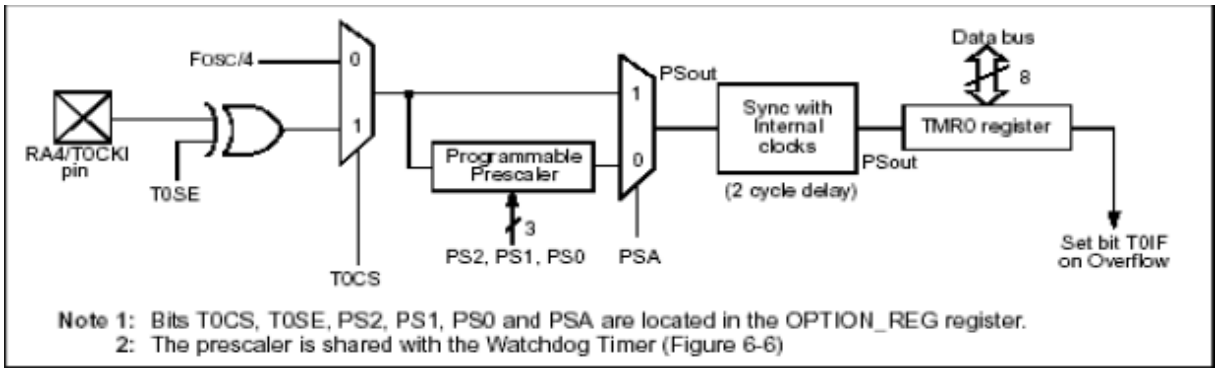


Figure I.11: Organigramme du Timer0. [3]

I.7:a. Registre TMR0 :

C'est le registre de 8 bits qui donne la valeur du comptage réalisé. Il est accessible en lecture et en écriture à l'adresse 01h dans la banque 0.

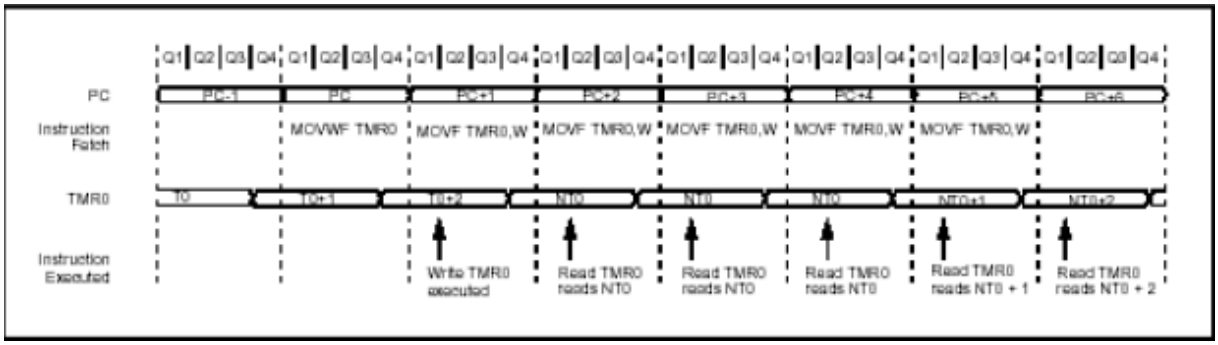


Figure I.12: Prise en compte de l'écriture dans le registre TMR0. [3]

Lors d'une écriture dans TMR0, le comptage est inhibé pendant deux cycles machine (Figure I.12). Si l'on veut déterminer un temps avec précision, il faut tenir compte de ce retard au démarrage. [2]

I.7:b. Choix de l'horloge :

Le timer 0 peut fonctionner suivant deux modes en fonction du bit T0CS (OPTION_REG.5). En mode timer (T0CS=0), le registre TMR0 est incrémenté à chaque cycle machine (si le pré-diviseur n'est pas sélectionné).

En mode compteur (T0CS=1), le registre TMR0 est incrémenté sur chaque front montant ou chaque front descendant du signal reçu sur la broche RA4/T0CKl en fonction du bit T0SE (OPTION_REG.4). Si T0SE=0, les fronts montants sont comptés, T0SE=1, les fronts descendants sont comptés. [2]

I.7:c. Pré-diviseur :

En plus des deux horloges, un pré-diviseur, partagé avec le chien de garde, est disponible. La période de l'horloge d'entrée est divisée par une valeur comprise entre 2 et 256 suivant les bits PS2, PS1 et PS0 (respectivement OPTION_REG.2, .1 et .0) (Figure I.13). Le bit PSA (OPTION_REG.3) permet de choisir entre la pré-division de timer0 (PSA=0) ou du chien de garde (PSA=1). [2]

PSA	PS2	PS1	PS0	/tmr0	/WD
0	0	0	0	2	1
0	0	0	1	4	1
0	0	1	0	8	1
0	0	1	1	16	1
0	1	0	0	32	1
0	1	0	1	64	1
0	1	1	0	128	1
0	1	1	1	256	1
1	0	0	0	1	1
1	0	0	1	1	2
1	0	1	0	1	4
1	0	1	1	1	8
1	1	0	0	1	16
1	1	0	1	1	32
1	1	1	0	1	64
1	1	1	1	1	128

Figure I.13: Valeurs du pré-diviseur en fonction de PSA, PS2, PS1 et PS0. [3]

I.7:d. Fin de comptage et interruption :

Le bit T0IF (INTCON.2) est mis à 1 chaque fois que le registre TMR0 passe de FFh à 00h. On peut donc tester ce bit pour connaître la fin de comptage. Pour compter 50 événements, il faut donc charger TMR0 avec la valeur 256-50=206 et attendre le passage de T0IF à 1. Cette méthode est simple mais bloque le processeur dans une boucle d'attente.

On peut aussi repérer la fin du comptage grâce à l'interruption que peut générer T0IF en passant à 1. Le processeur est ainsi libre de travailler en attendant cet événement. [2]

I.7:e. Registres utiles à la gestion de timer0 :

Plusieurs registres ont été évoqués dans ce paragraphe. Ils sont synthétisés dans la Figure I.14. [2]

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other resets
01h	TMR0	Timer0 module's register								xxxx xxxx	uuuu uuuu
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 0000
81h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
85h	TRISA	—	—	—	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	---1 1111	---1 1111

Legend: x = unknown, u = unchanged, - = unimplemented read as '0'. Shaded cells are not associated with Timer0.

Figure I.14: Registres utiles à la gestion de timer0. [3]

I.8. Les jeux d'instructions :

Les opérandes peuvent être de plusieurs types:

- f : adresse mémoire de registres (registre file adresse) de 00 à 7F
- W : registre de travail
- d : sélection de destination : d=0 vers W, d=1 vers f
- pp : numéro de PORT entre 1 et 3 sur deux bits
- bbb : adresse de bit dans un registre 8 bits (sur 3 bits)
- k : champ littéral (8, ou 11 bits)
- PC compteur programme

INSTRUCTIONS OPERANT SUR REGISTRE (direct)			indicateurs	Cycles
ADDWF	**F,d**	W+F → {W,F ? d}	C,DC,Z	1
ANDWF	**F,d**	W and F → {W,F ? d}	Z	1
CLRF	**F**	Clear F	Z	1
CLRW		Clear W	Z	1
CLRWDT		Clear Watchdoc timer	TO', PD'	1
COMF	**F,d**	Complémente F → {W,F ? d}	Z	1
DECF	**F,d**	décrémente F → {W,F ? d}	Z	1
DECFSZ	**F,d**	décrémente F → {W,F ? d} skip if 0		1(2)
INCF	**F,d**	incrémente F → {W,F ? d}	Z	1
INCFSZ	**F,d**	incrémente F → {W,F ? d} skip if 0		1(2)
IORWF	**F,d**	W or F → {W,F ? d}	Z	1
MOVF	**F,d**	F → {W,F ? d}	Z	1
MOVWF	**F**	W → F		1
RLF	**F,d**	rotation à gauche de F a travers C → {W,F ? d}	C	1
RRF	**F,d**	rotation à droite de F a travers C → {W,F ? d}		1
SUBWF	**F,d**	F – W → {W,F ? d}	C,DC,Z	1
SWAPF	**F,d**	permute les 2 quartets de F → {W,F ? d}		1
XORWF	**F,d**	W xor F → {W,F ? d}	Z	1

INSTRUCTIONS OPERANT SUR BIT				
BCF	**F,b**	RAZ du bit b du registre F		1
BSF	**F,b**	RAU du bit b du registre F		1
BTFSC	**F,b**	teste le bit b de F, si 0 saute une instruction		1(2)
BTFSS	**F,b**	teste le bit b de F, si 1 saute une instruction		1(2)

INSTRUCTIONS OPERANT SUR DONNEE (Immediat)				
ADDLW	**K**	W + K → W	C,DC,Z	1
ANDLW	**K**	W and K → W	Z	1
IORLW	**K**	W or K → W	Z	1
MOVLW	**K**	K → W		1
SUBLW	**K**	K – W → W	C,DC,Z	1
XORLW	**K**	W xor K → W	Z	1

INSTRUCTIONS GENERALES				
CALL	**L**	Branchement à un sous programme de label L		2
GOTO	**L**	branchement à la ligne de label L		2
NOP		No operation		1
RETURN		retourne d'un sous programme		2
RETFIE		Retour d'interruption		2
RETLW	**K**	retourne d'un sous programme avec K dans W		2
SLEEP		se met en mode standby	TO', PD'	1

{W, F ? d} signifie que le résultat va soit dans W si d=0 ou w, soit dans F si d= 1 ou f [4] C et DC,Z,PD,TO si les bits de registre d'étai du PIC-STATUS (figure I.7).

I.9. Conclusion :

Dans ce chapitre, nous avons brièvement présenté le microcontrôleur pic16f84, on a noté le brochage et l'architecture de pic et la caractéristique de ce circuit.

Chapitre II
FPGA /le langage VHDL

II.1. Introduction :

FPGA (Field Programmable Gâte Arrays ou "réseaux logiques programmables") sont des composants entièrement reconfigurables ce qui permet de les reprogrammer a volonté afin d'accélérer notablement certaines phases de calculs.

L'avantage de ce genre de circuit est sa grande souplesse qui permet de les réutiliser à volonté dans des algorithmes différents en un temps très court.

Le progrès de ces technologies permet de faire des composants toujours plus rapides et a plus haute intégration, ce qui permet de programmer des applications importantes.

Cette technologie permet d'implanter un grand nombre d'applications et offre une solution d'implantation matérielle à faible cout pour des compagnies de taille modeste pour qui, le cout de développement d'un circuit intègre spécifique implique un trop lourd investissement. [6], Il y a 4 principales catégories disponibles commercialement:

- Tableau symétrique.
- En colonne.
- Mers de portes.
- Les PLD hiérarchique. [7]

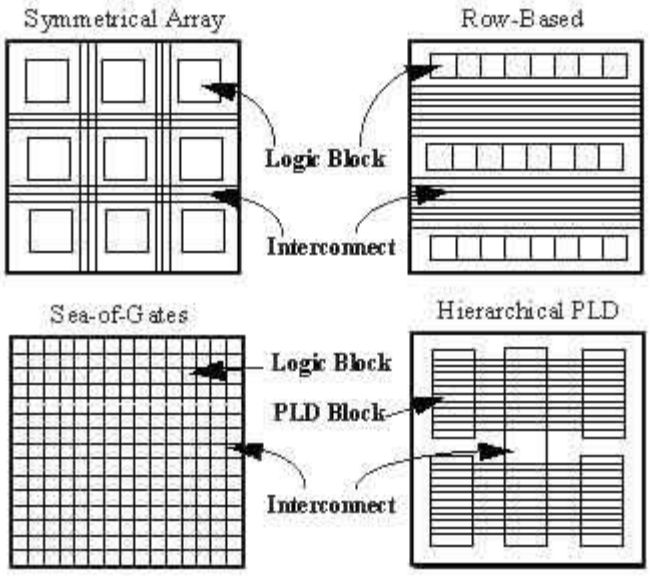


Figure II.1: Les différentes classes de FPGA. [7]

II.2. Configuration et reconfiguration des FPGA :

Un système reconfigurable est un système qui est constitué de composants ou entités à architecture modifiable afin de répondre à un objectif bien déterminé. Ce système reconfigurable dispose d'un mécanisme permettant de choisir une nouvelle configuration et de la mettre en place dans le cadre du processus de reconfiguration.

Les circuits FPGA sont un type de ces circuits reconfigurables. Ils sont programmables ou configurables sur les cartes sur quelles ils sont implantés par l'utilisateur. Cette reconfigurabilité est une propriété nécessaire face aux systèmes à charges et contraintes variables. Le FPGA est une abréviation anglaise qui signifie (réseau des portes programmables) ce qui est décrit dans la figure suivante : [8]

Figure II.2: Reprogrammabilité sur site d'un **FPGA [8]**

II.3. Technologies de programmation des FPGA :

Pour franchir les inconvénients susmentionnés des mémoires, et dans le but de faire un ensemble de technologies complémentaire adaptable suivant l'environnement des cahiers de charges, il existe trois types d'FPGA reprogrammables suivant la technologie de mémorisation pour répondre aux différentes applications.

Les trois principales technologies d'FPGA sont : [9]

II.3.1. Technologie de programmation par **RAM (XILINX et ALTERA) :**

Cette technologie permet d'avoir une reconfiguration rapide des FPGA. Les points de connexions sont des ensembles de transistors commandés. L'inconvénient majeur de cette technologie c'est qu'elle nécessite beaucoup de place et il est nécessaire de sauvegarder le design du FPGA dans une autre mémoire Flash.

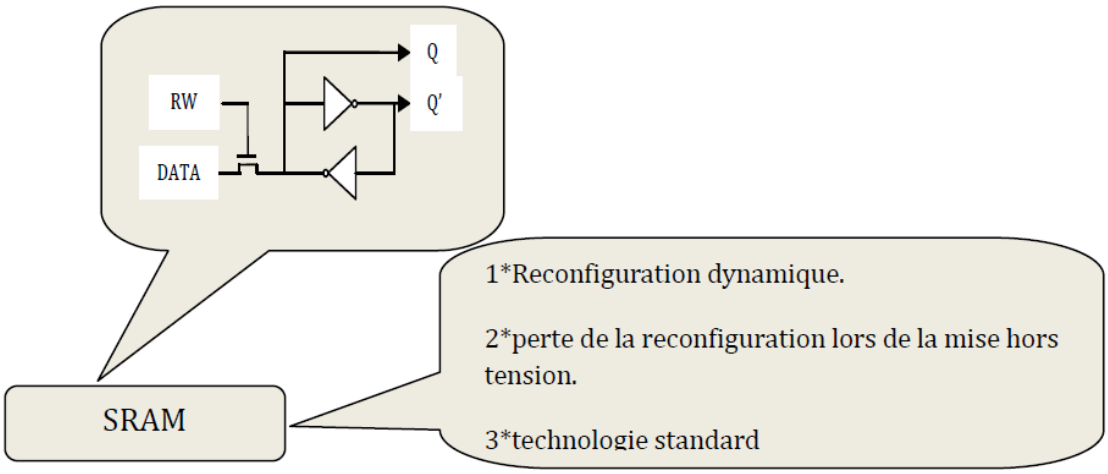


Figure II.3: Caractéristiques des technologies **SRAM** [9]

II.3.2. Technologie de programmation par **EEPROM** ou **FLASH (LATTICE et ACTEL) :**

Cette technologie garde sa configuration mais un nombre limité de configuration avec une configuration plus lente par rapport à SRAM.

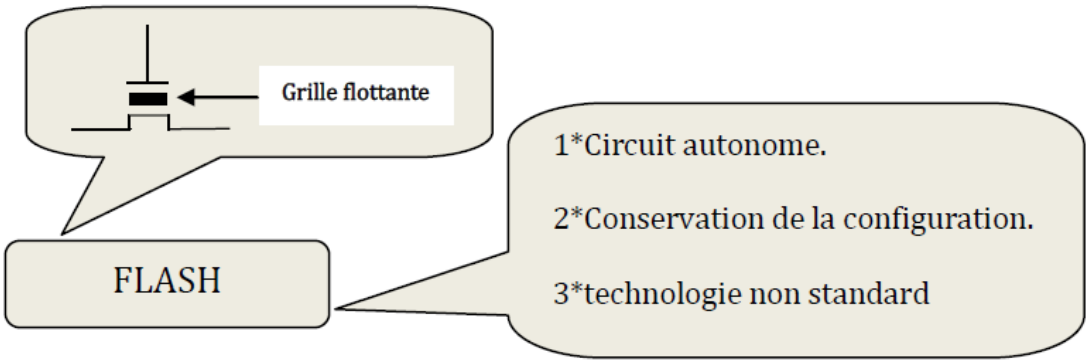


Figure II.4 : Caractéristiques des technologies **FLASH** [9]

II.3.3. Technologie de programmation par **ANTI-FUSIBLE (ACTEL) :**

Les points de connexions sont du type ROM, c'est-à-dire que la modification du point est irréversible. Pour comprendre le mécanisme de connexion sans rentrer dans les détails des semi-conducteurs, on considère que le point de connexion est le point de rencontre de deux segments conducteurs ou lignes conductrices. Le nom anti-fusible vient du fait que l'état initial du fusible ou la couche isolante est présente et il n'y a pas de contact pour l'établir, il faut détruire le fusible ce qui est contradictoire au fonctionnement habituel d'un fusible. Des composants moins génériques mais plus petits et plus rapides ont été développés.

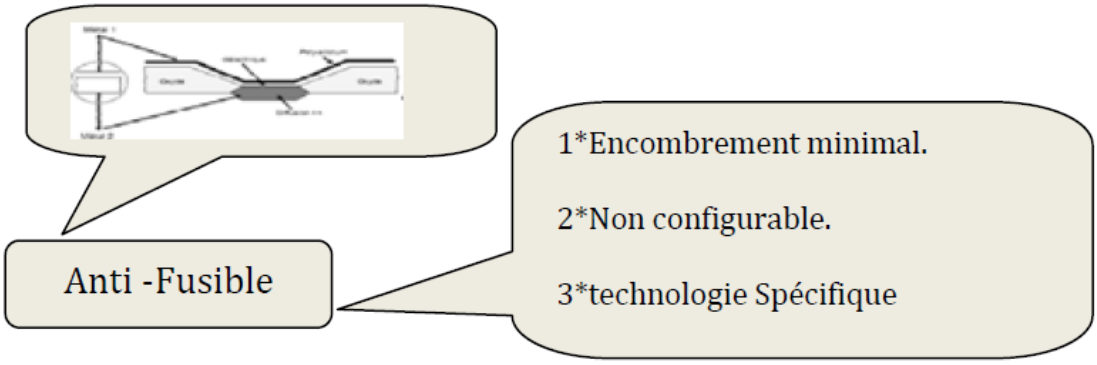


Figure II.5: Caractéristiques des technologies **ANTI-FUSIBLES** [9]

II.4. Architecture interne des FPGA :

On appelle les FPGA quelques fois LCA, abréviation anglaise de « **L**ogic **C**ell **A**rray » signifiant réseau de cellules logiques. Pour réussir à implanter un système dans un FPGA de manière efficace, il est indispensable de bien connaître sa structure interne et ses limites du point de vue performances. Les composants logiques programmables sont des circuits composés de nombreuses cellules logiques élémentaires librement assemblables. Celles-ci sont connectées de manière définitive ou réversible par programmation afin de réaliser les ou les fonctions numériques désirées. Un FPGA (**F**ield-**P**rogrammable **G**ate **A**rray) est un circuit intégré avec une structure adaptable par l'utilisateur et composée d'un réseau de cellules élémentaires ou d'éléments logiques programmables CLB et IOB répartis régulièrement et reliés entre eux grâce à des connections qui forment une matrice de routage programmable pour obtenir un comportement spécialisé du circuit dans sa globalité. Puisque tous les éléments sont programmables.

L'ensemble des systèmes reconfigurables FPGA est subdivisé en trois catégories suivant les fonctions préexistantes et des possibilités de les interconnectées. Ces catégories sont : des systèmes reconfigurables nommés "**grain fin**", des systèmes reconfigurables nommés "**grain moyen**" et des systèmes reconfigurables nommés " **grain large**".

L'architecture interne des FPGA est différente d'un fondeur à un autre et même entre les différentes gammes du même constructeur mais rien n'empêche que leurs ressemblances peuvent être rassemblées dans le Schéma représentatif de la figure suivante:[10]

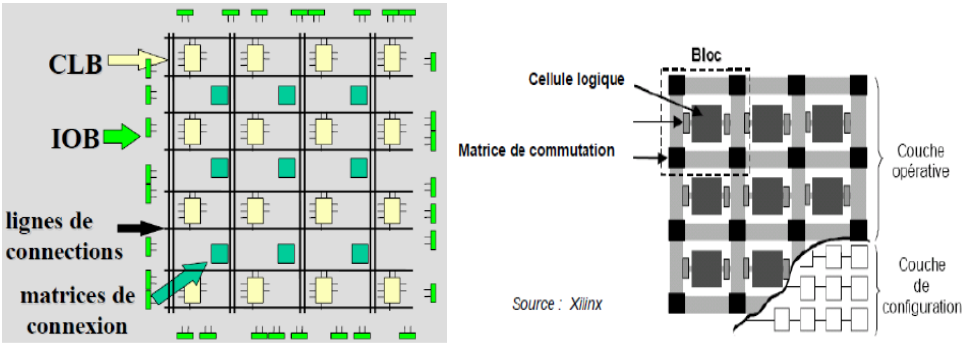


Figure II.6 : Architecture interne d'un **FPGA [10]**

❖ Les macro-cellules internes sont appelées :

➢ **S**oit **CLB** qui est la dénomination adoptée par Xilinx et abréviation anglaise de

« **C**onfigurable **L**ogic **B**lock », signifiant bloc logique configurable.

➢ **S**oit **LC** qui est le nom choisi par Cyprès et abréviation anglaise de

« **L**ogic **C**ell », signifiant cellule logique.

➢ **S**oit **LE** qui c'est l'appellation d'Altéra abréviation anglaise de « **L**ogic **E**lement» signifiant élément logique.

❖ Les macro-cellules sur la périphérie sont appelées : **IOB** abréviation anglaise de « **I**nput **O**utput **B**lock », signifiant bloc logique d'entrées sorties.

❖ L'ensemble des points de connexion est appelé **PIP**, abréviation anglaise de « **P**rogramme **I**nterConnect **P**oints ». La granularité des FPGA par les macro-cellules CLB nous permet d'implémenter des fonctions logiques «combinatoires ou séquentielles » complexes car chaque CLB est constitué d'une partie combinatoire et d'une partie séquentielle. Chaque fonction est décomposée en petites fonctions booliennes qui peuvent être contenues par de petites cellules élémentaires SLICES. Ces dernières comportent des LUT pour la partie combinatoire et une ou des bascules (généralement de type D) pour la partie séquentielle.

Les architectures existantes peuvent être regroupées en trois grandes catégories suivant la manière dont les blocs logiques sont organisés comme le montre la figure suivante:

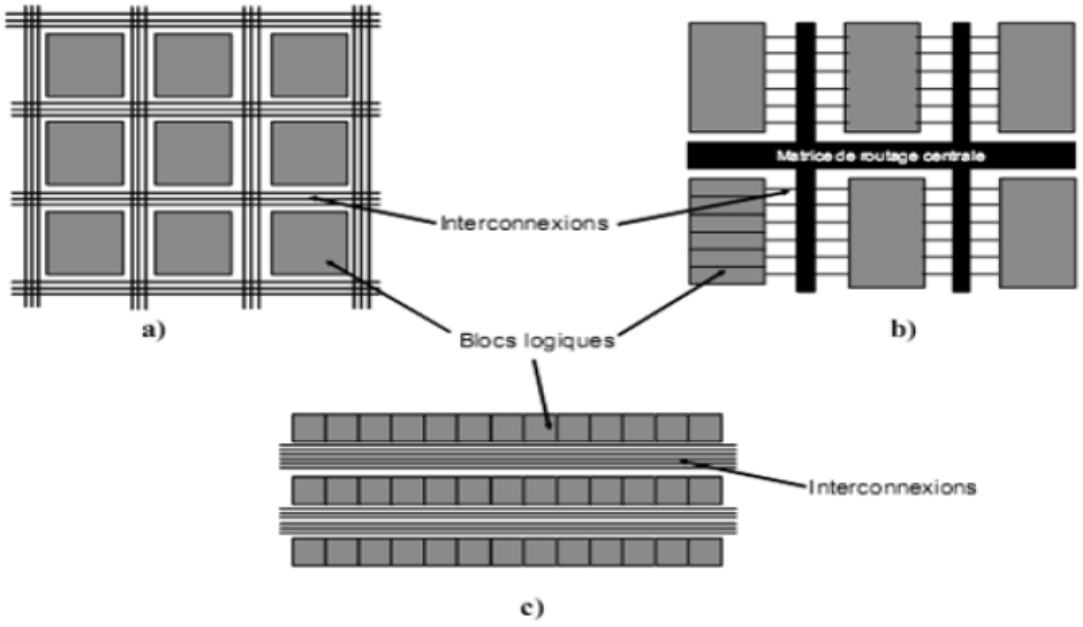


Figure II.7 : *Différentes architectures des* ***FPGA [10]***

II.5. Programmation des FPGA:

On commence par décrire le design soit en utilisant un langage de description matériel (tel VHDL, Verilog, ABEL,...) soit en rentrant directement le schématique. Le synthétiseur va générer la netlist, ensuite il faudra placer tous ces composants (si c'est possible, en effet, certains FPGA ne permettent pas d'émuler des Latchs) dans un FPGA et effectuer le routage entre les différentes cellules logiques. Au terme de ces étapes, le synthétiseur aura générer le bitstream qui sera prêt à être envoyer vers le FPGA. C'est seulement à ce moment là que la programmation proprement dite pourra avoir lieu.

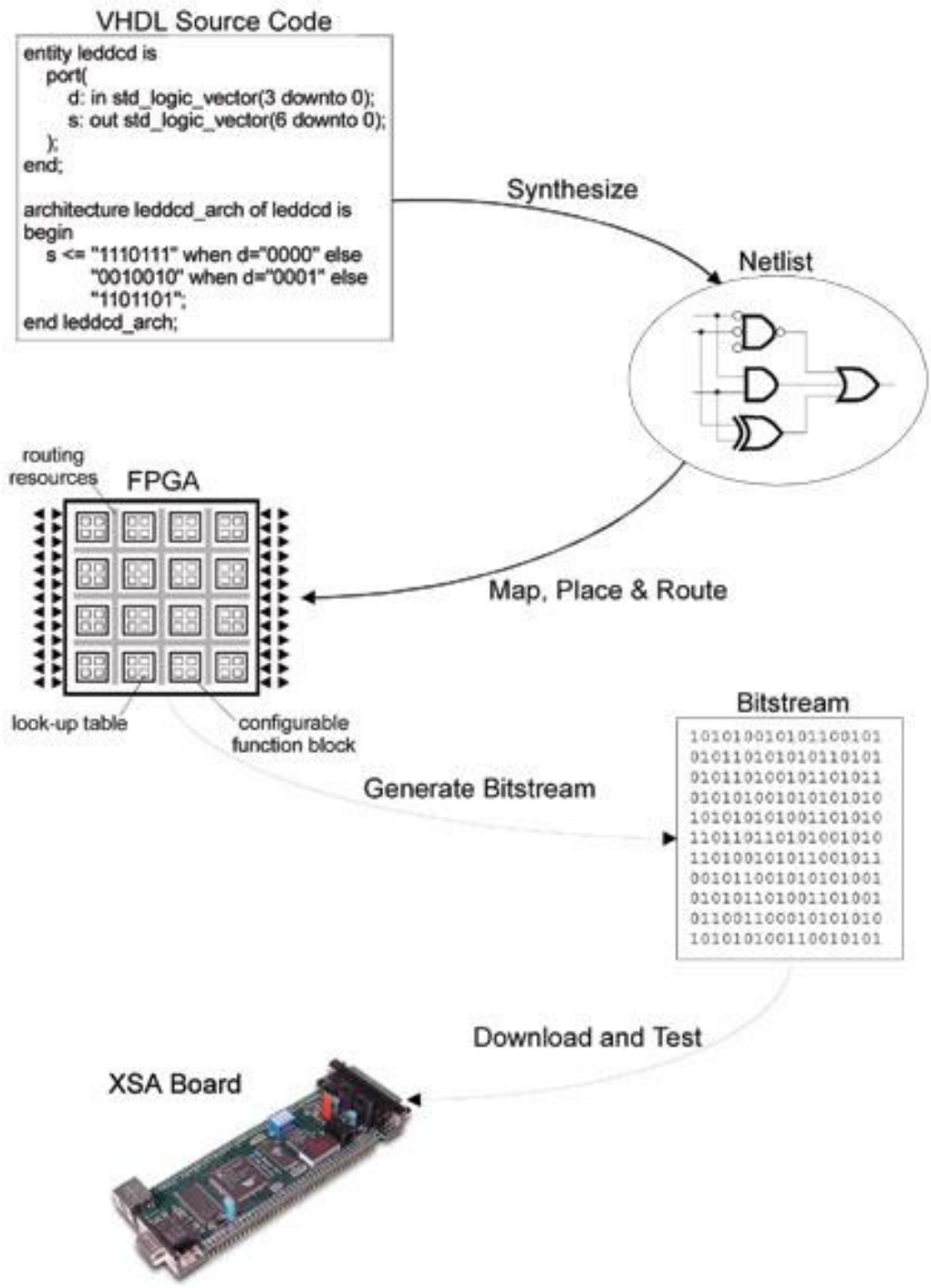


Figure II.8: Programmation d'un FPGA. [7]

II.6. Applications :

Les FPGA sont utilisés dans diverses applications nécessitant de l'électronique numérique (télécommunications, aéronautique, transports…). Ils sont également utilisés pour le prototypage d'ASIC. Les FPGA sont généralement plus lents, plus chers à l'unité et

consomment davantage d'énergie que leur équivalent en ASIC (Application Specific Integrated Circuit).

Applications Cependant, ils ont plusieurs avantages :

Délai de mise sur le marché plus court, car ce sont des composants standards, temps de conception plus court, car on réutilise des fonctions de base dont la reconfiguration autorise une validation préalable moins stricte, coût inférieur pour de petites séries (moins de 10 000 unités). Avec l'évolution technologique, cette quantité tend à augmenter : en effet, le prix d'une puce est proportionnel à sa surface, qui diminue avec la finesse de gravure, tandis que les coûts initiaux pour fabriquer un ASIC (conception, tests, masques de gravure) sont en forte augmentation. Il est parfois possible de transformer directement un FPGA en une version ASIC plus Rapide, moins chère et consommant moins (car les matrices de routage sont remplacées par une couche de métallisation fixe).

Plusieurs FPGA modernes possèdent la possibilité d'être reconfigurés (on parle de configuration lorsqu'il s'agit de programmation du matériel) partiellement à la volée. Ceci permet d'obtenir des systèmes reconfigurables - par exemple une unité centrale dont les instructions changent dynamiquement en fonction des besoins.

Les FPGA modernes sont assez vastes et contiennent suffisamment de mémoire pour être configurés pour héberger un cœur de processeur ou un DSP, afin d'exécuter un logiciel. On parle dans ce cas de processeur softcore, par opposition aux microprocesseurs hard-core enfouis dans le silicium.

Aujourd'hui, Les fabricants de FPGA intègrent même un ou plusieurs cœurs de processeur « hard-wore » sur un même composant afin de conserver les ressources logiques configurables du composant. Ceci n'exclut pas l'utilisation de processeur soft-wore possédant de nombreux avantages. On tend donc vers des « Systems On Chip », comme pour le microcontrôleur il y a quelques décennies, avec en plus de la logique configurable selon l'utilisateur. La mémoire des tout derniers FPGA est encore insuffisante pour exécuter des logiciels embarqués un peu complexes et on doit avoir recours à des mémoires externes (ROM, RAM). Cependant, la loi de Moore n'est pas encore à bout de souffle et celles-ci devraient être intégrées dans quelques années et suffiront à une grande partie des applications embarquées. [11]

II.7. Fabricants :

Parmi les fabricants de tels circuits programmables, on trouve Abound Logic, Achronix, Microsemi (ex. Actel), Intel PSG (ex. Altera), Atmel, Cypress, Lattice Semiconductor, Nallatech, QuickLogic, SiliconBlue, Tabula Inc., Tier Logic et Xilinx. D'autres sociétés sont probablement encore à l'état de développement initial et n'ont pas publiquement annoncé de produits. [11]

II.8. Avantages des FPGA:

- Délai de conception.
- Reconfiguration à chaud.
- Prototypage rapide.
- Délai de fabrication.
- Facilité de test. [10]

II.9. Inconvénients des FPGA:

- Prix unitaire trop élevé pour les très grandes séries.
- Performance électriques inférieures aux puces spécialisées (notamment en fréquence).
- Faible taux d'utilisation du circuit. [10]

II.10. Le langage VHDL :

II.10.1. Introduction :

VHDL sont les initiales de VHSIC Hardware Description Langage, VHSIC étant celles de Very High Scale Integrated Circuit. Autrement dit, VHDL signifie : langage de description matériel s'appliquant aux circuits intégrés à très forte intégration. [12]

II.10.2. Historique :

Dans les années 80, le département de la défense aux Etats-Unis fait un appel d'offre pour avoir un langage de description matérielle numérique unique. Le langage VHDL (VHSIC Hardware Description Langage) est inventé pour répondre à ces critères. Ce langage se base sur le VHSIC (Very High Speed Integrated Circuit) qui est un projet de recherche nationale mené par le groupement IBM/Texas Instruments / Intermetrics. Ce langage est ouvert au domaine public en 1985 et deviendra une norme en 1987 sous la dénomination d'IEEE 10761987. Des changements minimes se feront pour la seconde normalisation en

1993 et il portera le nom VHDL'93. La dernière évolution de la norme est la norme IEEE 1076-2001. Les avantages qui se dégagent de ce langage sont la réutilisation de fichiers grâce à la notion de paquet et la généricité individuelle des modèles. Cependant, la norme IEEE 1076-2001 ne permet pas seule la description mixte (numérique et analogique). Le Verilog et le VHDL ont des capacités techniques équivalentes.

Le choix du langage est souvent dicté par la « culture » de l'équipe de développement ou de l'équipe de recherche. De plus, ce choix est parfois imposé par les outils disponibles, les logiciels de simulation et de synthèse, dont les langages associés sont fixés par des aspects économiques. Nous pouvons noter toutefois que les outils actuels permettent de mélanger ces deux langages. [13]

II.10.3. Structure d'une description VHDL simple :

Une description VHDL est composée de 2 parties indissociables à savoir :

- L'entité (ENTITY), elle définit les entrées et sorties.

- L'architecture (ARCHITECTURE), elle contient les instructions VHDL permettant de réaliser le fonctionnement attendu. [14]

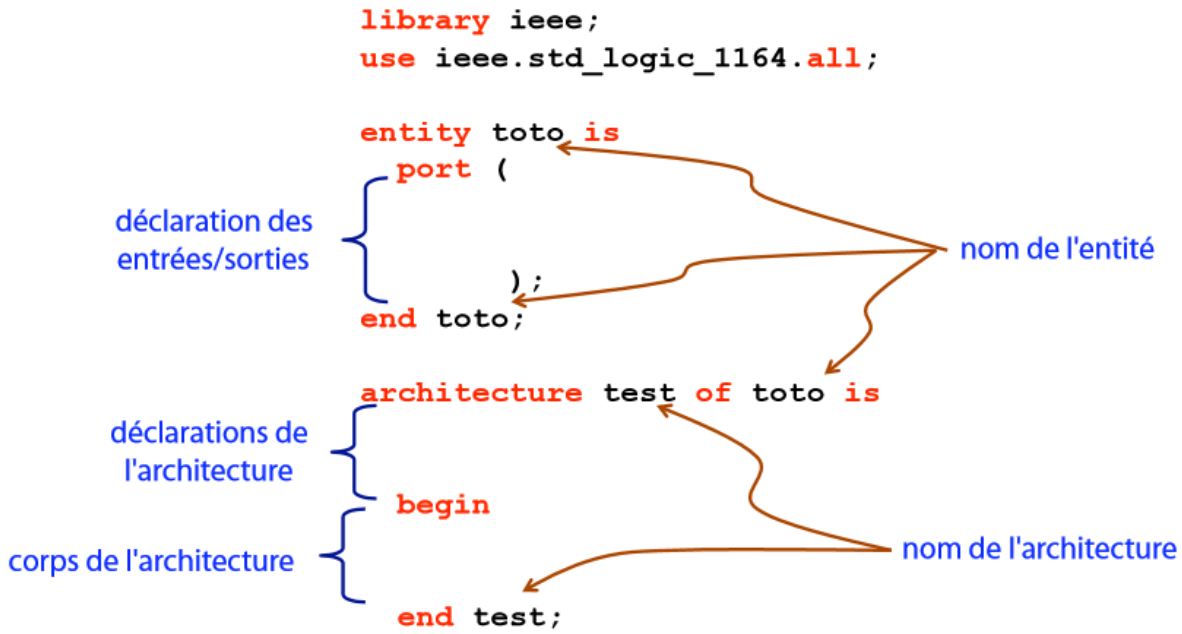


Figure II.9: structure d'un programme VHDL [15]

II.10.3.1. Déclaration des bibliothèques :

Toute description VHDL utilisée pour la synthèse a besoin de bibliothèques. L'IEEE (Institut of Electrical and Electronics Engineers) les a normalisées et plus particulièrement la bibliothèque IEEE1164. Elles contiennent les définitions des types des signaux électroniques, des fonctions et sous programmes permettant de réaliser des opérations arithmétiques et logiques,... [14]

```
Library ieee;
Use ieee.std_logic_1164.all;
 Use ieee.numeric_std.all;
 Use ieee.std_logic_unsigned.all;
```

```
-- cette dernière bibliothèque est souvent utilisée pour l'écriture de compteurs
```

La directive Use permet de sélectionner les bibliothèques à utiliser.

II.10.3.2. Déclaration de l'entité et des entrées / sorties (I/O) :

Elle permet de définir le NOM de la description VHDL ainsi que les entrées et sorties utilisées, l'instruction qui les définit c'est port : [14]

Syntaxe:

```
entity NOM_DE_L_ENTITE  is
port  (Description des signaux d'entrées /sorties …);
End  NOM_DE_L_ENTITE ;
```

II.10.3.3. Déclaration de l'architecture correspondante à l'entité :

Description du fonctionnement :

L'architecture décrit le fonctionnement souhaité pour un circuit ou une partie du circuit.
En effet le fonctionnement d'un circuit est généralement décrit par plusieurs modules VHDL. Il faut comprendre par module le couple ENTITE/ARCHITECTURE. Dans le cas de simples PLDs on trouve souvent un seul module.

L'architecture établit à travers les instructions les relations entre les entrées et les sorties. On peut avoir un fonctionnement purement combinatoire, séquentiel voire les deux séquentiel et combinatoire. [14]

Syntaxe:

Architecture NOM_DE_L_ ARCHITECTURE **OF** NOM_DE_L_ENTITE **IS**

Zone de déclaration

Begin

Description de la structure logique

End NOM_DE_L_ ARCHITECTURE ;

II.10.4. Les Opérateurs du langage :

- Opérateurs logiques : and, or, nand, nor, xor et xnor
- Opérateur relationnel : =, /=, <, <=,> et >=
- Opérateurs de décalage : sll, srl, sla, sra, rol et ror
- Opérateur d'addition : +, - et &
  - ✓ Concaténation & : vecteur_4bit <=A & B & ‘‘10’’
- Opérateurs de signe + et –
- Opérateur de multiplication : *, /, mod et rem
- Opérateur divers : **, abs et not [16]

II.10.5. Types d'instructions de l'architecture :

II.10.5.1. Instructions concurrentes :

Les instructions concurrentes sont décrites directement dans le corps de l'architecture. L'objectif de ces relations étant d'affecter des valeurs à des composants ou de réaliser des connexions, alors elles n'ont pas d'ordre d'exécution. Elles sont exécutées en parallèle. [18]

II.10.5.1.1. Affectation simple :

Dans une description VHDL, c'est certainement l'opérateur le plus utilisé. En effet il

Permet de modifier l'état d'un signal en fonction d'autres signaux et/ou d'autres opérateurs.

```
NOM_ D'UNE_GRANDEUR <=V ALEUR _OU _NOM_D'UNE _GRANDEUR ;
```

II.10.5.1.2. L'affectation conditionnelle :

```
L'interconnexion est cette fois soumise à une ou plusieurs conditions.

NOM_D'UNE_GRANDEUR <= Q1 when CONDITIONI else

                      Q2 when CONDITION2 else

                      …

                      Qn ;
```

On note l'absence de ponctuation à la fin des lignes intermédiaires.

II.10.5.1.3. Affectation sélective :

Cette instruction permet d'affecter différentes valeurs à un signal, selon les valeurs prises par un signal dit de sélection.

```
With EXPRESSION select

        NOM_D'UNE_GRANDEUR <= Q1 when valeur1,

                              Q2 when valeur2,

                              … ,

                              Qn when others ;
```

On note la présence d'une virgule (et non un point virgule) à la fin des lignes intermédiaires.

II.10.5.1.4. L'instruction concurrente générate :

Cette instruction a deux formes:

La forme itérative, exemple:

```
LABEL_BOUCLE: for 1 in 0 to N generate

--suite d'instructions concurrentes
```

```
end generate LABEL_BOUCLE; --le label est facultatif ici.
```

La forme conditionnelle, exemple:

```
LABEL_BOUCLE: if CONDITION_BOOLEENNE  generate

suite d'instruction concurrente

end generate LABEL_BOUCLE; -- le label et facultatif ici
```

II.10.5.1.5. Le processus :

Avec les processus, une liste de sensibilité définit les signaux autorisant les actions. Deux syntaxes sont couramment utilisées. Dans la première les signaux à surveiller sont énumérés dans une liste de sensibilité juste après le mot **process** :

```
Process(LISTE _ D E _ SENSIBILITE)

NOM_DES _OBJETS_INTERNES: « type» ; -- zone facultative

begin

INSTRUCTIONS_SEQUENTIELLES;

end process ;
```

Le déclenchement sur un front montant d'un élément de la liste de sensibilité (CLK par exemple) se fait par les instructions suivantes:

```
If (CLK 'event'and CLK = '1') then

INSTRUCTIONS;

end if;
```

Avec la seconde syntaxe on utilise l'instruction wait pour énoncer la liste de sensibilité:

```
Process
NOM_DES_OBJETS_INTERNES : « type» ; -- zone facultative
begin
wait on (LISTE_DE_SENSIBILITE) ;
INSTRUCTIONS_ SEQUENTIELLES;
end process
```

Le déclenchement sur un front montant d'un élément de la liste de sensibilité (CLK par exemple) se fait en replaçant l'instruction wait par la forme suivante:

```
Wait until (CLK=’even’ and CLK=’1’)
```

L'instruction process peut être précédée d'une étiquette afin de faciliter la lecture du programme:

```
LABEL: Process(LISTE _ DE_SENSIBILITE
        NOM_DES_OBJETS_INTERNES : « type»; -- zone facultative
    Begin
        INSTRUCTIONS _SEQUENTIELLES;
end process LABEL;
```

II.10.5.2. Instructions séquentielles :

Les instructions séquentielles sont internes aux processus, aux procédures et aux fonctions (sous-programmes). Elles constituent les outils de base des descriptions comportementales. Ces instructions, pour la plupart, sont inspirées des langages classiques de programmation. Parmi elles, on retrouve **if-then-else** qui permet de réaliser les boucles conditionnelles. **Case-then** pour tester des valeurs et choisir l’opération à effectuer. L’instruction **wait**, qui permet de suspendre l’exécution d’un processus jusqu’à ce que la durée spécifiée soit évaluée. Les boucles telque : **for-loop, while-loop**… [18]

II.10.5.2.1. L'affectation séquentielle :

Même syntaxe « <= » et même signification que pour une instruction concurrente; pour les variables, on utilise « : = ».

II.10.5.2.2. Le test «if .. then .. elsif .. else .. end if» :

La syntaxe de cette instruction est la suivante:

```
If CONDITIONI then
        INSTRUCTION 1 ;
elsif CONDITION 2 then
        INSTRUCTION 2 ;
elsif CONDITION3 then
        INSTRUCTION 3 ;
        ...
else INSTRUCTIONX;
end if ;
```

Les conditions sont examinées les unes après les autres dans l'ordre d'écriture; dès que l'une d'elle est vérifiée, on sort de la boucle et les conditions suivantes ne sont pas examinées.

II.10.5.2.3. Le test « case .. when .. end case» :

```
Case EXPRESSION is
        when ETAT l => INSTRUCTION 1;
        when ETAT2 => INSTRUCTION 2;
        …
```

```
            when others => INSTRUCTION n ;

end case ;
```

Ces instructions sont particulièrement adaptées à la description des machines d'états.

II.10.6. SOUS-PROGRAMMES :

IL existe 2 types de sous-programmes:

-Les fonctions

-Les procédures

Le langage VHDL permet l'utilisation et la création de fonctions ou de procédures que l'on peut appeler à partir d'une architecture ou d'un paquetage.

II.10.6.1. Les fonctions :

La fonction exécute en séquentiel et reçoit des paramètres d'entrées et renvoie un paramètre de sortie pour le programme appelant.

Syntaxe:

FUNCTION nom de la fonction **(**liste des paramètres de la fonction avec leur type**)** **RETURN**

Type du paramètre de retour **IS**

Zone de déclaration des variables**;**

BEGIN

Instructions séquentielles**;**

RETURN nom de la variable de retour ou valeur de retour**;**

END;[17]

Exemple :

```
function f (a, b, c : std_logic) return std_logic is
 variable x : std_logic;
  begin
      x := ((not a) and (not b) and c);
   return x;
 end f;[15]
```

II.10.6.2. les procédures :

les procédures, elles diffèrent des fonctions par le fait qu'elles acceptent des paramètres dont la direction peut être IN, INOUT et OUT. Une procédure ne possède donc pas un ensemble de paramètres d'entrées et un paramètre de sortie mais un ensemble de paramètres d'entrées-sorties et aucun paramètre spécifique de sortie.

Syntaxe :

Procedure nom de la procédure **(**liste des paramètres de la procédure avec leur direction et type**) RETURN IS**

Zone de déclaration des variables**;**

BEGIN

Instructions séquentielles**;**

END [nom de la procédure*]***;[17]**

Exemple:

```
type byte is array (7 downto 0) of std_logic;
•••
procedure ByteToInteger (ib: in byte; oi: out integer) is
variable result : integer := 0;
begin
for i in 0 to 7 loop
if ib(i) = '1' then
result := result + 2**i;
end if;
end loop;
oi := result;
end ByteToInteger;[15]
```

II.11.Conclusion :

Dans ce chapitre nous avons introduit et architecture les circuits FPGA, à deux langages de programmation différents : VHDL et VERILOG puis nous choisie le langage VHDL, on a noté la définition et la structure d'un programme VHDL.

Chapitre III

Implantation PIC 16F84 SUR FPGA

III.1. Introduction :

Dans ce travail on a implémenté un microcontrôleur pic 16f84 sur FPGA dans l'environnement du logiciel Quartus. Pour ce faire, on réalise la conversion du langage assembleur pic 16f84 au langage VHDL. Cette opération se compose de 4 parties :

-conversion des instructions assembleur PIC16F84 en VHDL.

-écriture du langage assembleur en hexadécimale dans la ROM en langage VHDL.

-compilation du programme VHDL par Quartus.

-implémentation du programme VHDL sur l'FPGA altéra cyclone II.

III.2. Technologie Altéra :

III.2.1. Quartus II :

III.2.2. Présentation :

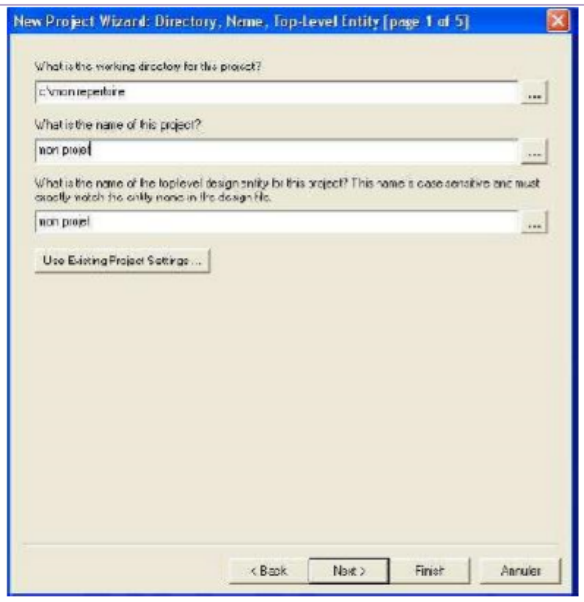
Quartus est un logiciel proposé par la société Altéra, permettant la gestion complète d'un flot de conception FPGA. Ce logiciel permet de faire une saisie graphique ou une saisie texte (description VHDL) d'en réaliser une simulation, une synthèse et une implémentation sur cible reprogrammable.

En ce qui suit un tutorial des principales fonctionnalités nécessaires à la création d'un projet, son analyse et en fin son implémentation sur l'FPGA. Ce tutorial est construit avec des tables explicatives contenant des captures d'écran ainsi que des instructions à suivre.

Figure III.1: Fenêtre du logiciel Quartus II version 7.2 d'Altera

III.2.3. Création d'un nouveau projet :

Pour créé un nouveau projet cliquer sur **File** puis sur **New Project Wizard** en fin sur **OK**



Choisir l'emplacement du répertoire ou seront stockés tous les fichiers du projet. Choisir le nom de votre projet. Choisir le nom de l'entité maître du projet (niveau le plus haut dans le design). Conseil : il faut créer un répertoire par projet. Ne pas le créer dans le dossier : **c :\altera\72\quartus** mais dans un répertoire de travail.

Figure III.2. Création d'un nouveau projet sous Quartus II

Cliquer sur **Next** puis quand la fenêtre **Add Files** apparaît recliquer sur **Next**.

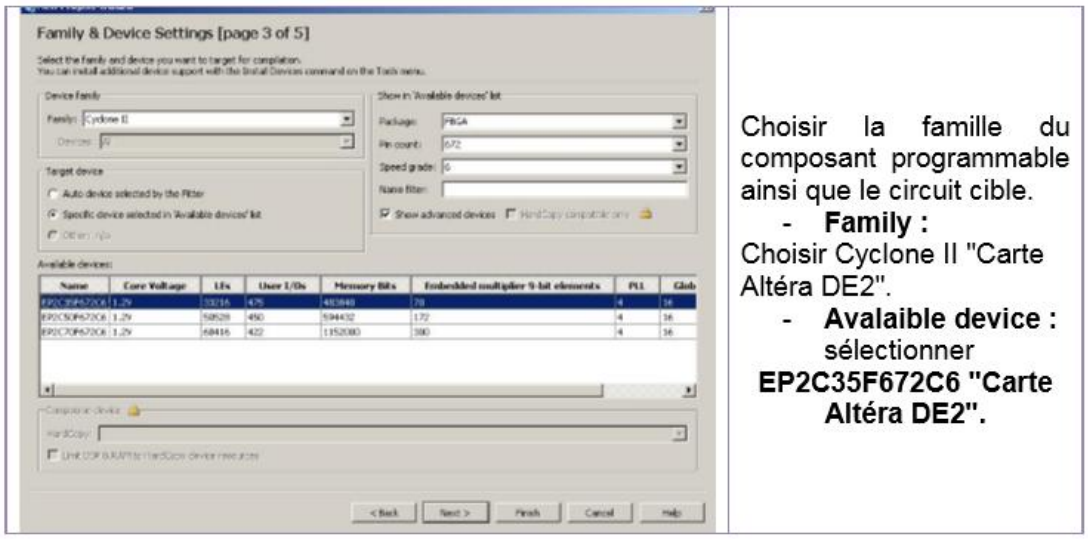


Figure III.3: Configuration du choix du circuit cible a implémenter

Après avoir valider les choix précédents en cliquant sur **Next**, la fenêtre **EDA Tool Settings** apparaît. Cliquer encore une fois sur **Next** ce qui fait apparaitre une fenêtre récapitulative :

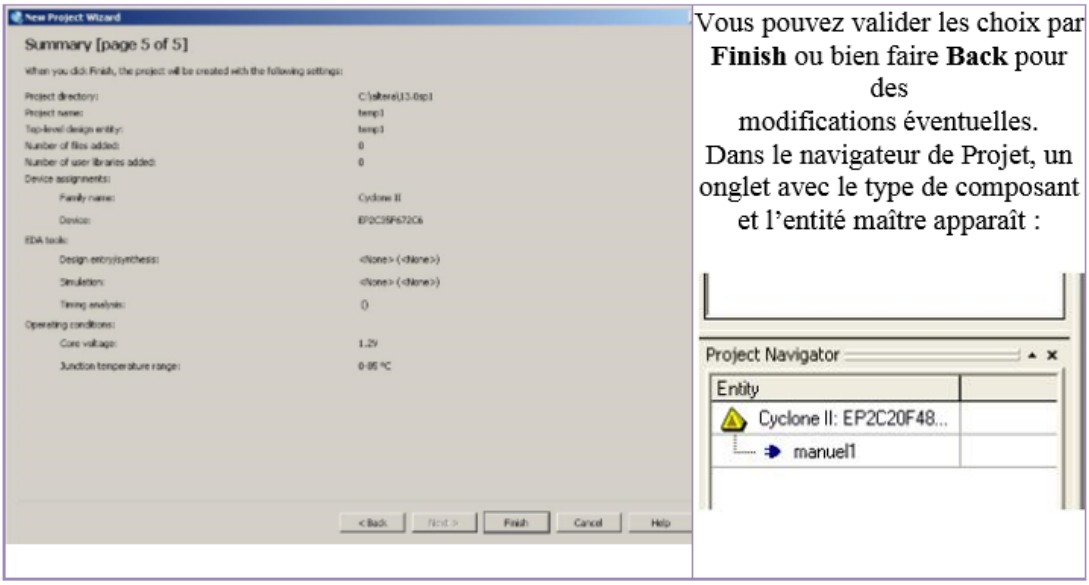
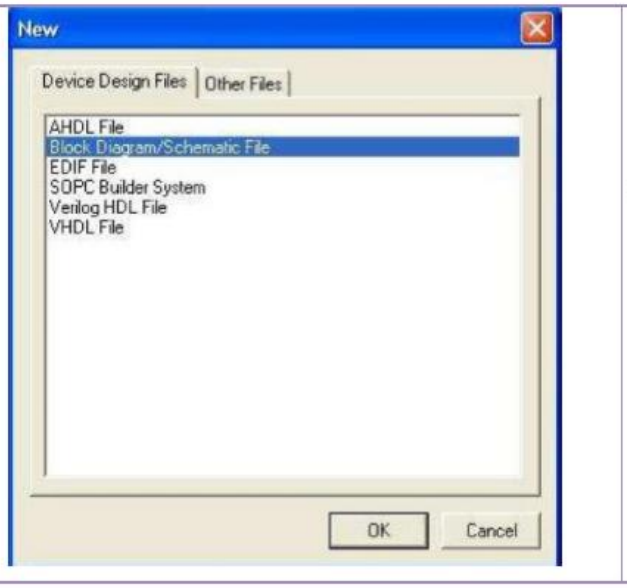


Figure III.4: Validation de la configuration et affichage du sommaire du projet

III.2.4. Saisie d'un projet &Création d'un schéma :

Pour créé un schéma relatif à un projet cliquer sur **File** puis sur **New**



Pour créer un schéma relatif à un projet cliquer sur **File** puis sur **New** ensuite sélectionner **Block diagram / Schematic file** et en fin cliquer sur **OK**

Figure III.5: Saisi du projet et création du schéma

Pour sauvegarder le schéma choisir **File** ensuite en clique sur **Save as** pour choisir l'emplacement de l'enregistrement ensuite le nom du fichier.

C**onseil :** Une feuille blanche se crée intitulée Block1.bdf. On prendra soin de sauver cette feuille sous le nom de l'entité maître, car c'est maintenant cette feuille de saisie graphique qui a la hiérarchie la plus haute dans le projet.

Figure III.6: Utilisation de la boite à outils

III.2.5. Création d'un fichier VHDL :

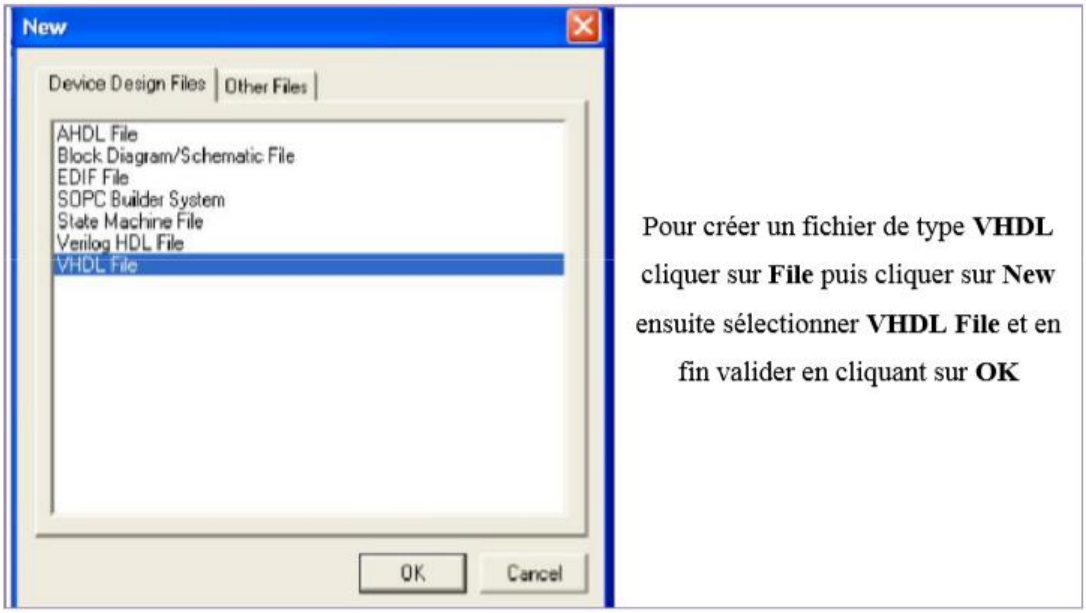


Figure III.7: Création d'un fichier VHDL sous Quartus II

Figure III.8: Editeur de code VHDL sous Quartus II

Une fois le code VHDL saisie, il convient de le sauver (**File** puis **Save As**) puis d'en vérifier la syntaxe.

Conseil : Il est important de sauver le fichier sous le même nom que l'entité. Cela évite des intersections d'entité entre fichiers.

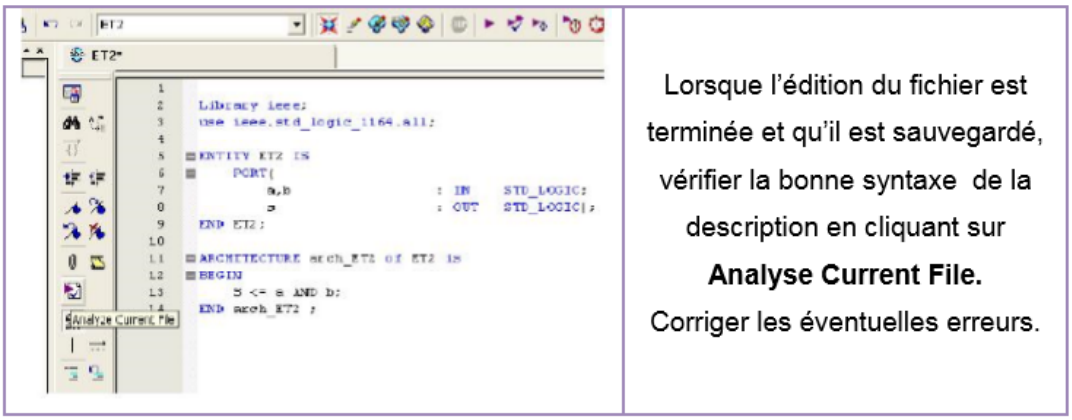


Figure III.9: Sauvegarde du script VHDL et analyse des erreurs

III.2.6. Création d'un symbole :

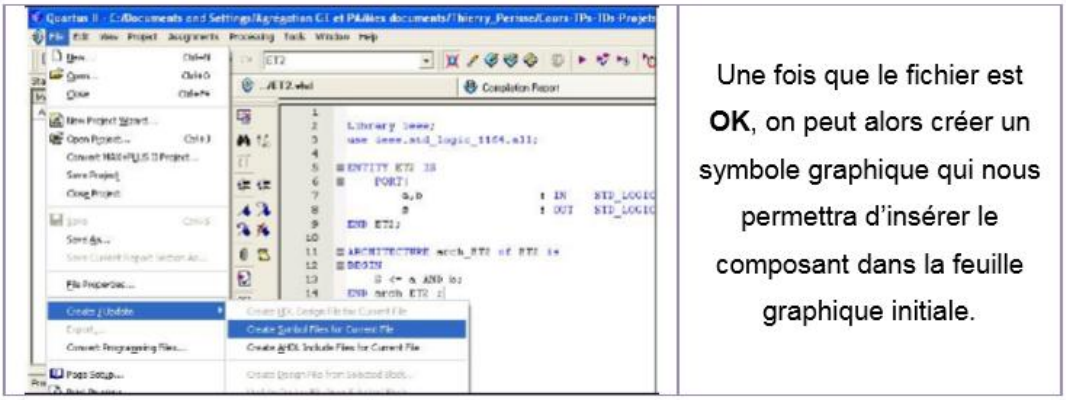


Figure III.10: Création d'un symbole équivalent aux instructions du script VHDL

III.2.7. Compilation :

Durant la compilation, Quartus va réaliser 4 étapes :

a- La transformation des descriptions graphiques et textuelles en un schéma électronique à base de portes et de registres : c'est la **synthèse logique.**

b- L'étape de Fitting (ajustement) consiste à voir comment les différentes portes et registres (produit par la synthèse logique) peuvent être placés en fonction des ressources matérielles du circuit cible (EP2C35F672C6) : c'est la **synthèse physique.**

c- L'assemblage consiste à produire les fichiers permettant la programmation du circuit. Ce sont des fichiers au format Programmer Object Files (.pof), SRAM Object Files (.sof),

Hexadécimal (Intel-Format) Ouput Files (.hexout), Tabular Text Files (.ttf), et Raw Binary Files (.rbf).

d- L'analyse temporelle permet d'évaluer les temps de propagation entre les portes et le long des chemins choisis lors du fitting.

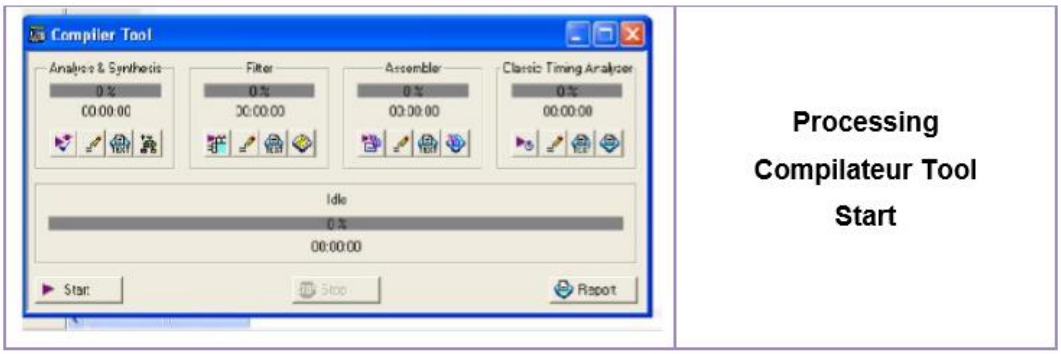


Figure III.11:l'outil "Compilateur" en vue de sa mise en marche

Normalement, il ne doit pas y avoir d'erreur. Si ce n'est pas le cas, on vérifie dans la zone **Processing** (en bas où s'affichent les messages) la source du problème.

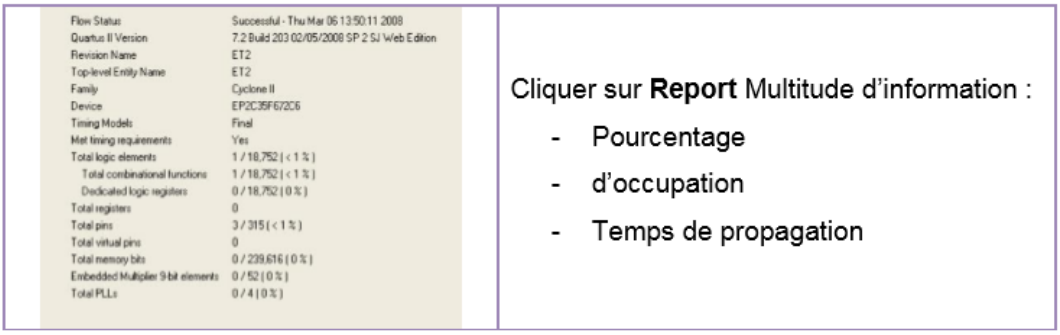


Figure III.12: Capture d'écran du rapport d'information

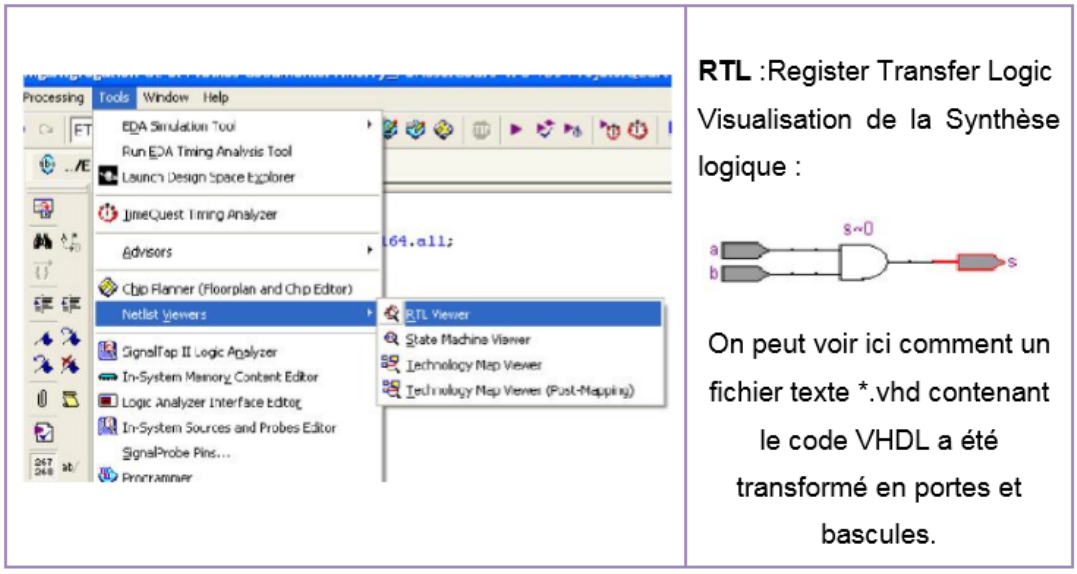


Figure III.13: Visualisation de la synthèse logic

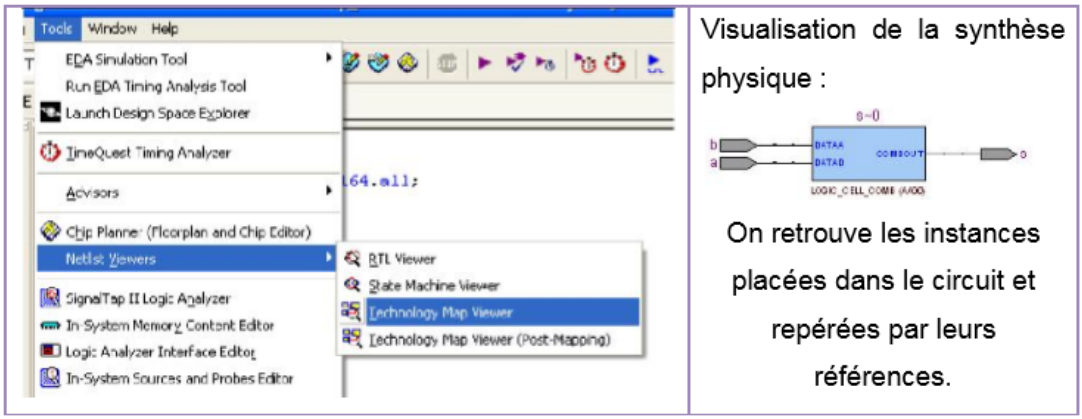
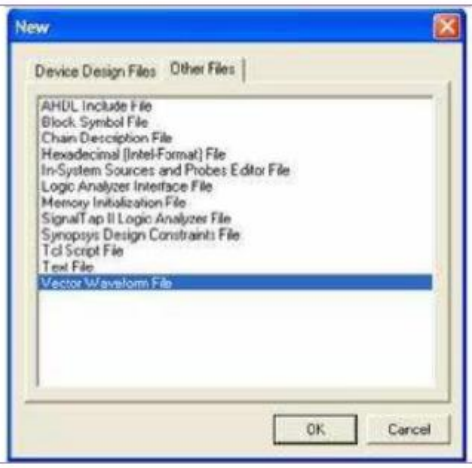


Figure III.14: Visualisation de la synthèse physique

III.2.8. Simulation d'un circuit :

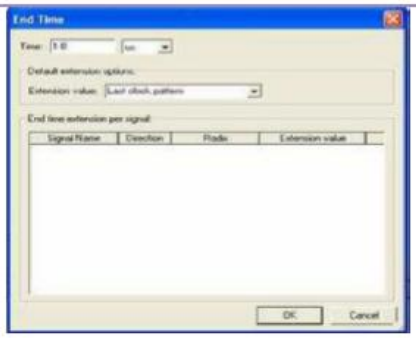
- ✓ La partie du circuit à simuler doit être munie de pins d'entrée/sortie.
- ✓ Elle doit aussi se trouver au niveau le plus élevé de la hiérarchie. Si ce n'est pas le cas, pour l'y mettre : dans le « **Project Navigator** », cliquer avec le bouton droit de la souris sur le nom du fichier, puis sur **Set as Top-Level Entity.**
- ✓ Il faut également vérifier qu'il n'y ait pas d'erreur dans le circuit en cliquant sur **Processing** puis sur Start, et enfin sur **Start Analysis & Elaboration.**
- ✓ Le circuit étant prêt, il faut maintenant créer le fichier contenant les informations sur les signaux à appliquer sur les entrées du composant et la liste des signaux que l'on veut analyser.

Cliquer sur **File** puis sur **New**



Sélectionner l'onglet « **Other Files** » et cliquer sur **Vector Waveform File**.

Figure III.15: Création du fichier des vecteurs de test



Par défaut, la durée de simulation est de **1 µs**. Pour la modifier, cliquer sur **Edit**, puis **EndTime**. Une fois la durée modifiée, cliquer sur **OK**.

Figure III.16: Choix des durées des vecteurs de test

Sauvegarder le fichier sous son nom définitif avec son extension (**.vwf**) en cliquant sur **File** puis **Save As.**

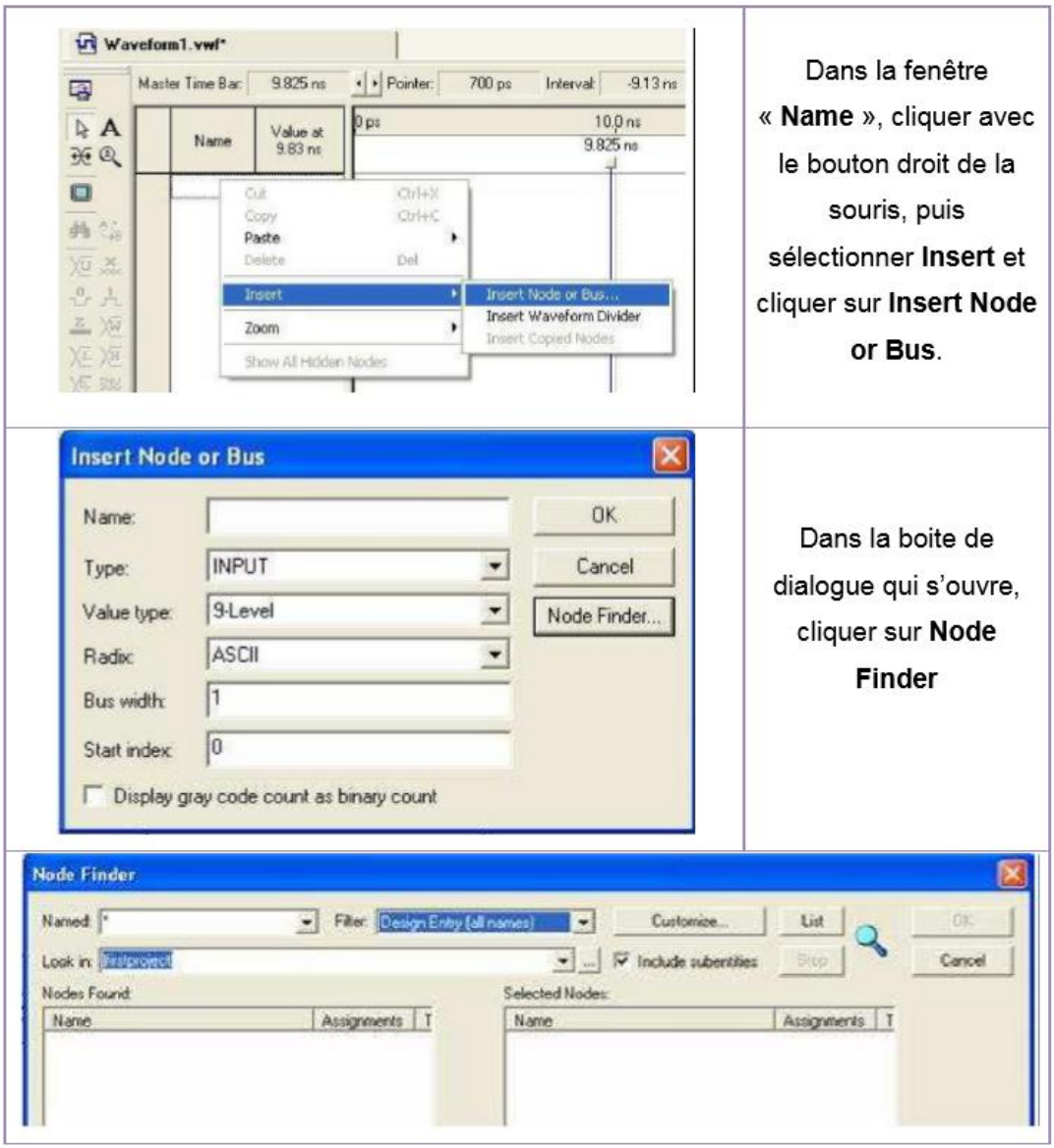


Figure III.17: Insertion des stimuli afin de les générer

Dans la catégorie « **Filter** », choisir **all names**. Cliquer ensuite sur le bouton **List**. Ajouter les signaux souhaités dans la fenêtre **Selected Nodes**, en cliquant sur  Cliquer sur **OK** pour fermer les différentes fenêtres et revenir à l'éditeur de signaux.

Afin de simuler le design, il convient de lui injecter des stimuli. Lorsque ces stimuli sont générés à partir d'un fichier on dit que l'on utilise un **fichier de Bench.**

Cliquer avec le bouton droit de la souris sur le nom d'un signal, sélectionner **Value**,

Puis choisir la valeur du signal dans le menu.

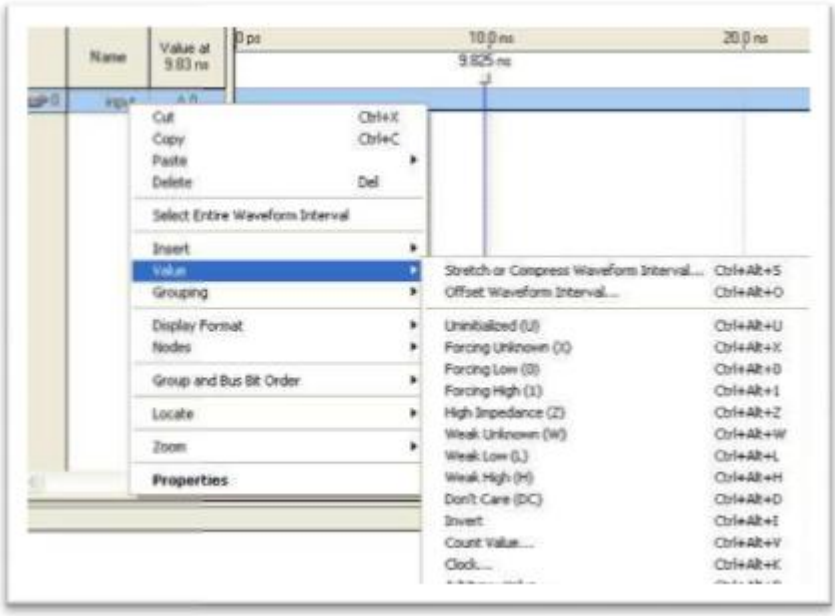


Figure III.18:menu du choix de la valeur du signal

Il est possible d'effectuer la même opération sur une partie seulement d'un signal en sélectionnant une zone dans la partie «chronogramme». Il faut pour cela maintenir le bouton gauche de la souris appuyé en déplaçant le curseur. Lorsque tous les signaux d'entrées sont définis, sauvegarder le fichier.

III.2.9. Simulation Fonctionnelle :

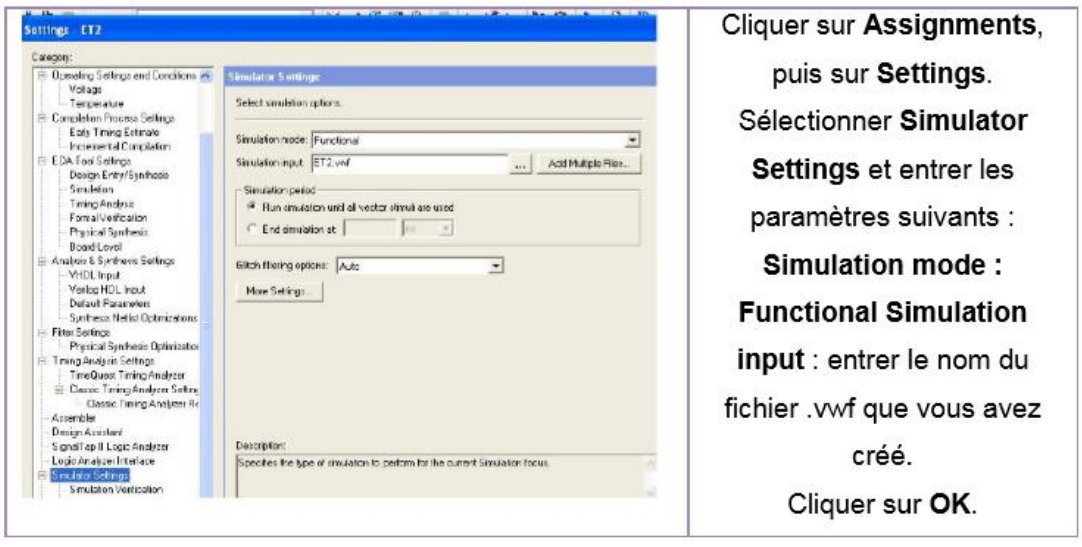


Figure III.19: Configuration du mode de simulation

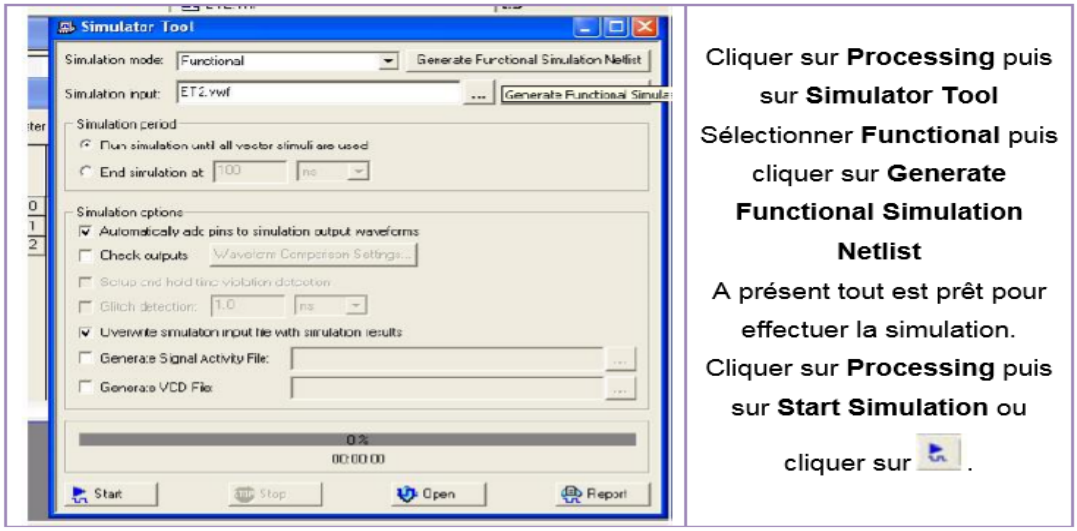


Figure III.20: Finalisation et démarrage de la simulation

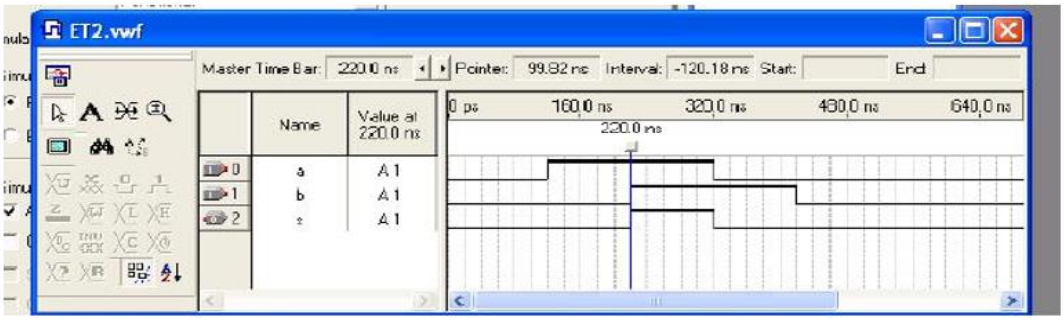


Figure III.21: Chronogramme de la simulation fonctionnelle

III.2.10. Simulation Temporelle :

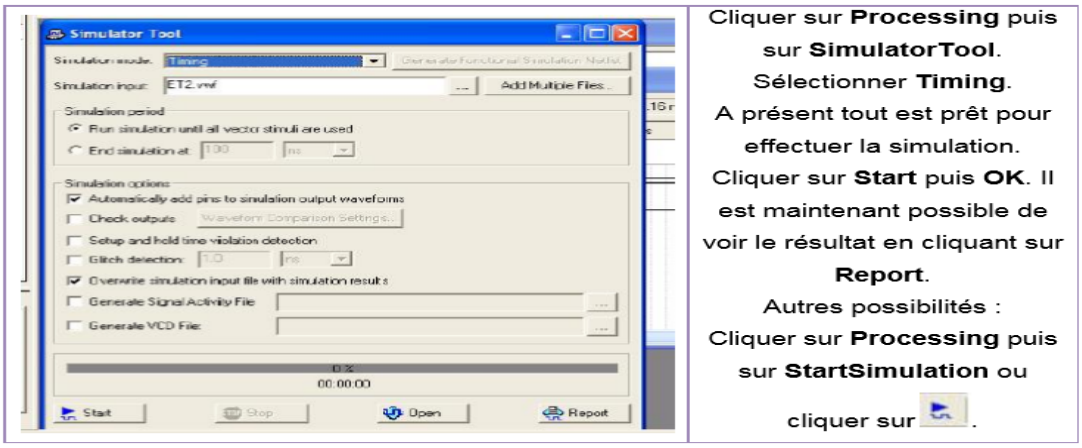


Figure III.22: Configuration et démarrage de la simulation temporelle

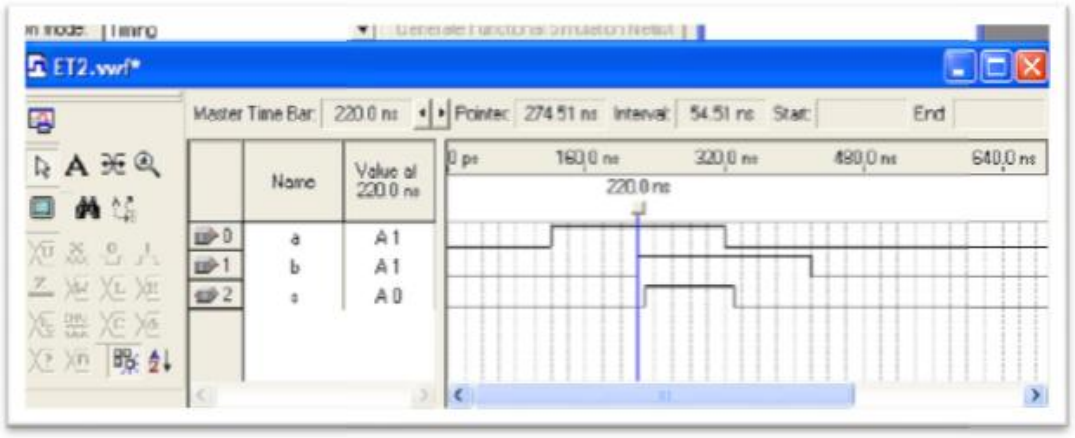


Figure III.23: Chronogramme de la simulation temporelle

Avant la programmation du circuit il faut assigner les pins d'entrées et de sorties du design aux broches du circuit physique.

III.2.11. Affectation des entrées et des sorties :

Cliquer sur **Assignements** puis sur **pins**

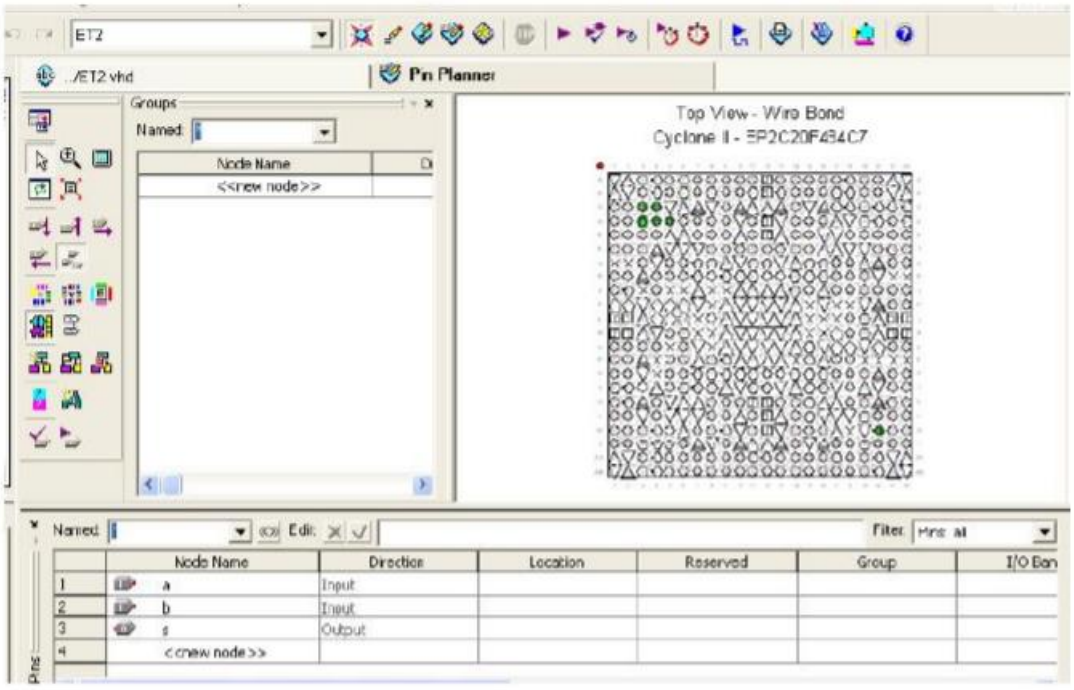


Figure III.24: Assignements des pins avec Quartus

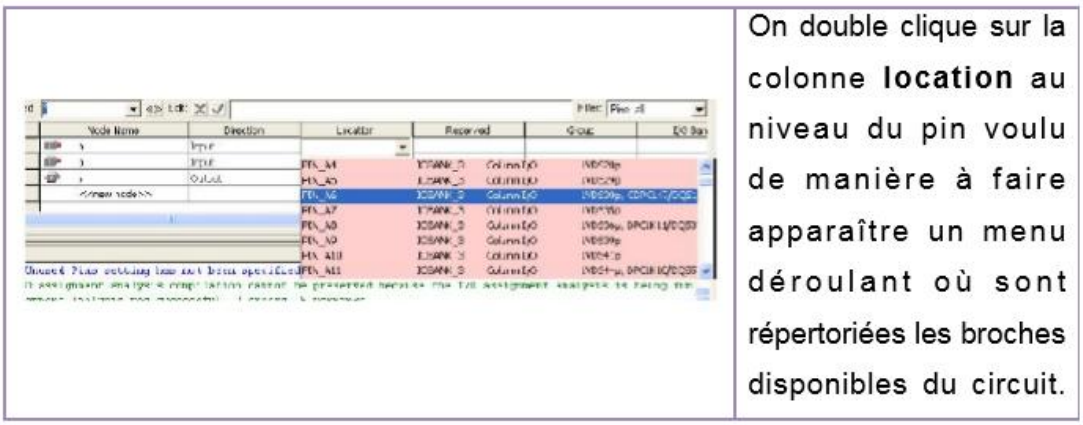


Figure III.25: Localisation des broches disponible du circuit

La liste des broches utilisables pour le FPGA et sortant sur les connecteurs est donnée dans le manuel de la carte DE2 d'Altéra. Ne pas oublier de compiler avant la programmation.

III.2.12. Programmation du circuit :

La programmation du circuit se fait via le protocole JTAG. Pour cela, vérifier que la connexion entre le PC et la carte DE1 via le module USB-Blaster est opérationnelle. Si tout est **ok** lancer le programmateur : Cliquer sur **Tools** puis sur **Programmer.**

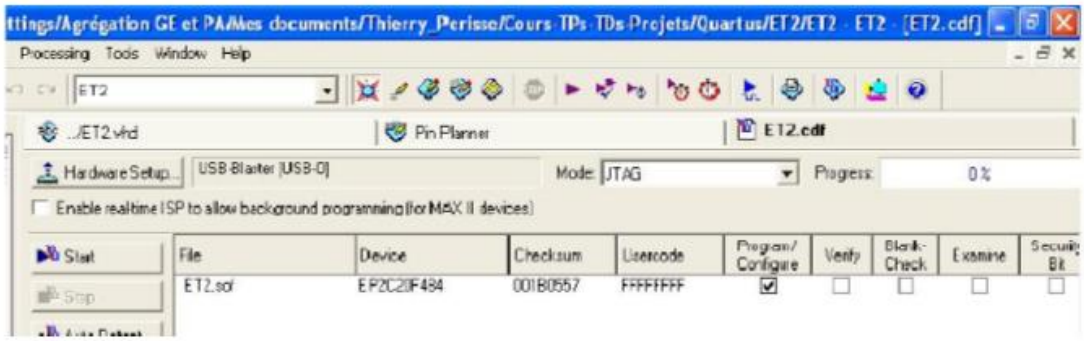


Figure III.26:étape pré-programmation avec Quartus

Vérifier que le fichier avec l'extension .**sof** est bien là (sélectionner le) et que la case **Program/Configure** est cochée, puis cliquer sur **Start**.

III.3. Transformé quelques instructions assembleur de pic 16f84 en langage VHDL :

Cette section est **difficile**. Même si elle ne fait intervenir que des notions du niveau indiqué, il est conseillé d'avoir du recul sur les notions présentées pour bien comprendre ce qui suit. Cependant, ce contenu n'est pas fondamental et peut être sauté en première lecture. Il est temps de présenter maintenant l'ensemble des 35 instructions du PIC 16F84 codées sur 14 bits.

III.3.1. L'instruction MOVLW K :

Cette instruction chargement littéral de W (W = k)

Leur op code (binary) est : 11 00xx kkkk kkkk

Alors leur programme en VHDL est :

```
case Inst is
    when "110000" =>    W<=k;
                        adr<=std_logic_vector(unsigned(adr)+1);

    when others   =>     romout := "--------------";


end case;
```

Après l'exécution est :

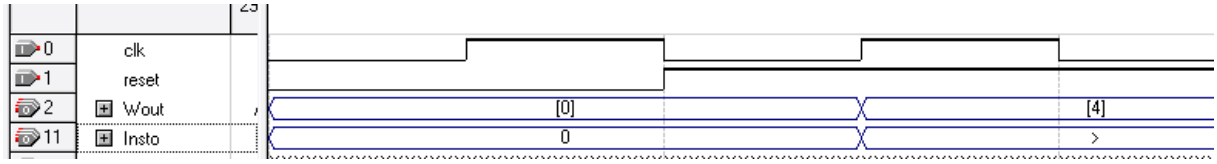


Figure III.27 : chronogramme de simulation en fonctionnel de MOVLW K

III.3.2. L'instruction ADDLW K :

Cette instruction addition de W et k,

Leur op code (binary) est : 11 111x kkkk kkkk

Alors leur programme en VHDL est :

```
case Inst is
    when "110000" =>    W<=k;
                        adr<=std_logic_vector(unsigned(adr)+1);
    when "111110" =>    W<=std_logic_vector(unsigned(W)+unsigned(k));
                        adr<=std_logic_vector(unsigned(adr)+1);

    when others   =>     romout := "--------------";


end case;
```

Après l'exécution est :

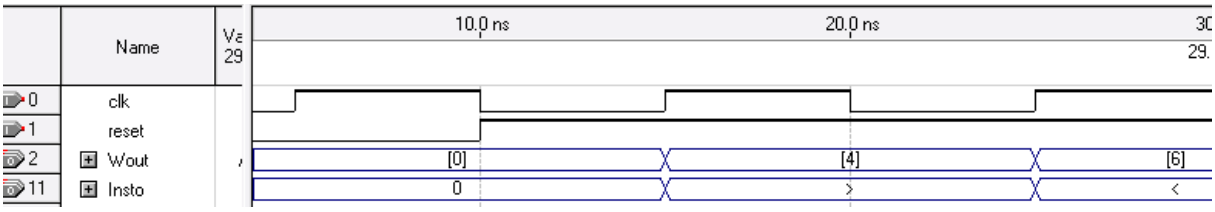


Figure III.28 : chronogramme de simulation en fonctionnel de ADDLW K

III.3.3. L'instruction SUBLW K :

Cette instruction soustraction de W et k

Leur op code (binary) est : 11 110x kkkk kkkk

Alors leur programme en VHDL est :

```
case Inst is
    when "110000" =>   W<=k;
                       adr<=std_logic_vector(unsigned(adr)+1);
    when "11 110" =>   W<=std_logic_vector(unsigned(W)sub unsigned(K));
                       adr<=std_logic_vector(unsigned(adr)+1);
    when others   =>    romout := "--------------";


end case;
```

Après l'exécution est :

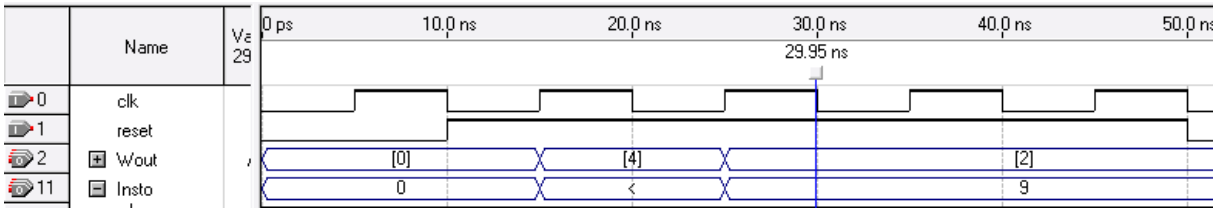


Figure III.29 : chronogramme de simulation en fonctionnel de SUBLW K

III.3.4. L'instruction ANDLW K :

Cette instruction et littéral avec W (W = k AND W)

Leur op code (binary) est : 11 1001 kkkk kkkk

Alors leur programme en VHDL est :

```
case Inst is
    when "110000" =>   W<=k;
                       adr<=std_logic_vector(unsigned(adr)+1);
    when "11 100" =>   W<=std_logic_vector(unsigned(W)and unsigned(K));
                       adr<=std_logic_vector(unsigned(adr)+1);
    when others   =>    romout := "--------------";


end case;
```

Après l'exécution est :

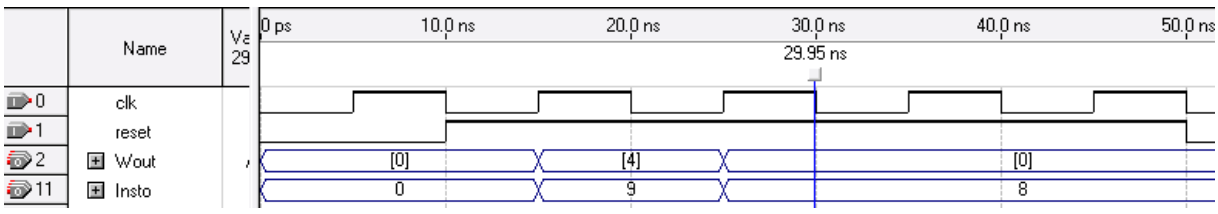


Figure III.30: chronogramme de simulation en fonctionnel de ANDLW K

III.3.5. L'instruction IORLW K :

Cette instruction OU Inclusif littéral avec W (W = k OR W)

Leur op code (binary) est : 11 1000 kkkk kkkk

Alors leur programme en VHDL est :

```
case Inst is
    when "110000" =>   W<=k;
                       adr<=std_logic_vector(unsigned(adr)+1);
    when "11 100" =>   W<=std_logic_vector(unsigned(W)or unsigned(K));
                       adr<=std_logic_vector(unsigned(adr)+1);
    when others   =>    romout := "--------------";


end case;
```

Après l'exécution est :

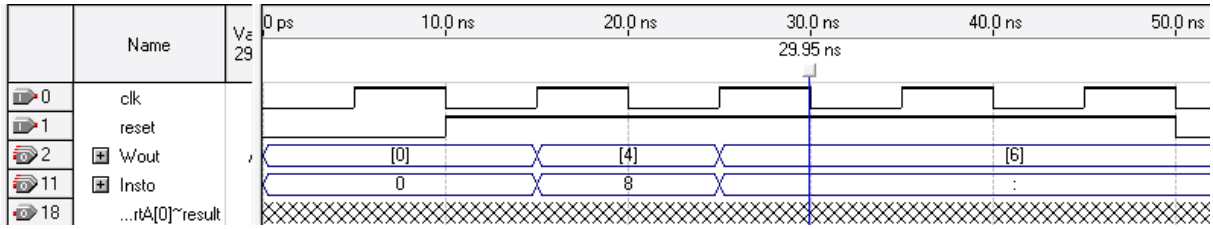


Figure III.31 : chronogramme de simulation en fonctionnel de IORLW K

III.3.6. L'instruction XORLW k :

Cette instruction OU exclusif or littéral avec W (W = k XOR W)

Leur op code (binary) est : 11 1010 kkkk kkkk

Alors leur programme en VHDL est :

```
case Inst is
    when "110000" =>   W<=k;
                       adr<=std_logic_vector(unsigned(adr)+1);
    when "11 101" =>   W<=std_logic_vector(unsigned(W)xor unsigned(K));
                       adr<=std_logic_vector(unsigned(adr)+1);
    when others   =>    romout := "--------------";


end case;
```

Après l'exécution est :

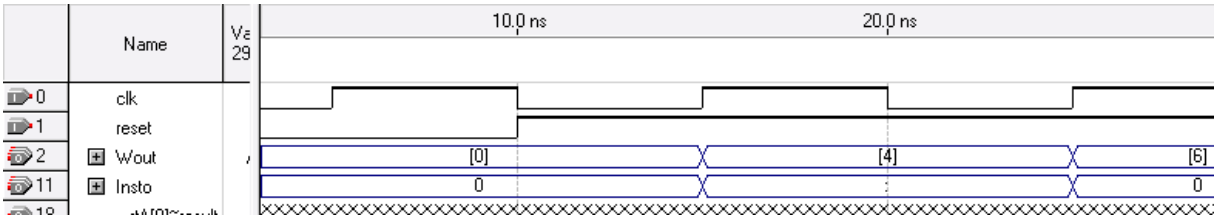
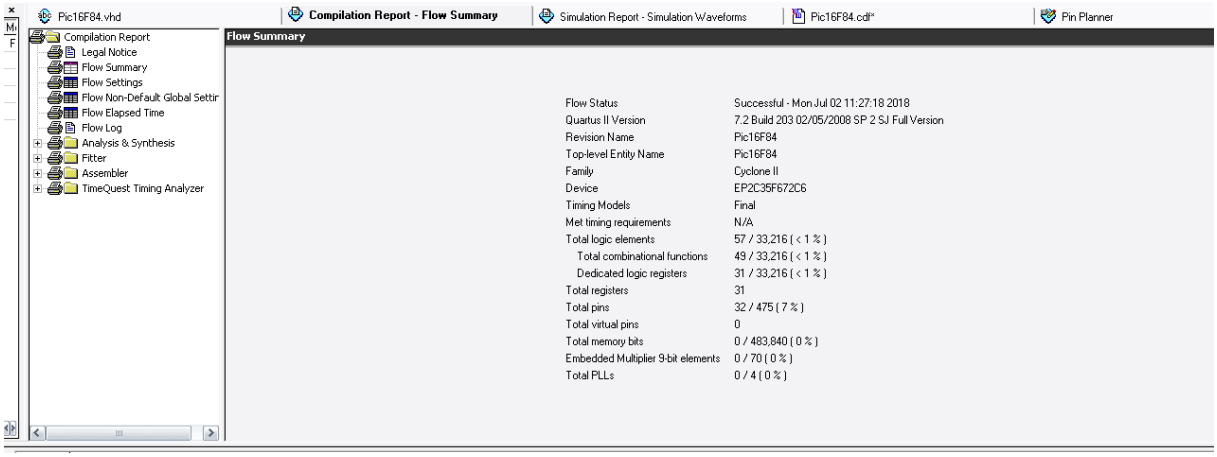


Figure III.32 : chronogramme de simulation en fonctionnel de XORLW K

Et finalement comme application j'ai exécutée tout les instructions dans les programmes que fait l'opération :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all ;




entity Pic16F84 is port(clk :in std_logic;reset: in std_logic; PortA,PortB: inout std_logic_vector(7 downto 0);
Wout: out std_logic_vector(7 downto 0);Insto: out std_logic_vector(5 downto 0));
end Pic16F84;

architecture ARCH of PIC16F84 is
signal adr    : std_logic_vector(12 downto 0);
signal W      : std_logic_vector(7 downto 0);


  procedure EPROM ( Clk : in std_logic; adr : in std_logic_vector(12 downto 0); romout  : out std_logic_vector(13 downto 0)) is
  begin
    if Clk'event and Clk = '1' then
        case to_integer(unsigned(adr)) is
            when 000000 => romout := "11000000000100";  -- 0x0000
            when 000001 => romout := "11111000000010";  -- 0x0002
            when 000002 => romout := "11110000000010";  -- 0x0004
            when 000003 => romout := "11100100000010";  -- 0x0006
            when 000004 => romout := "11100000000010";  -- 0x0008
            when 000005 => romout := "11101000000010";  -- 0x000A
            when 000006 => romout := "XXXXXXXXXXXXXX";  -- 0x000C
            when 000007 => romout := "XXXXXXXXXXXXXX";  -- 0x000E
            when 000008 => romout := "XXXXXXXXXXXXXX";  -- 0x0010
            when 000009 => romout := "XXXXXXXXXXXXXX";  -- 0x0012
            when 000010 => romout := "XXXXXXXXXXXXXX";  -- 0x0014
            when 000011 => romout := "XXXXXXXXXXXXXX";  -- 0x0016
            when 000012 => romout := "XXXXXXXXXXXXXX";  -- 0x0018
            when 000013 => romout := "XXXXXXXXXXXXXX";  -- 0x001A
            when others => romout := "--------------";
        end case;
    end if;

 end  EPROM;
begin

process (Clk)
variable romout: std_logic_vector(13 downto 0);
variable Inst: std_logic_vector(5 downto 0);
variable k: std_logic_vector(7 downto 0);
 begin
 if Clk'event and Clk = '1' then
   if (reset='0') then
     adr<="0000000000000";
   end if;
   EPROM (Clk,adr,romout);
   Inst:=romout(13 downto 8);
   k:=romout(7 downto 0);


    case Inst is
        when "110000" =>   W<=k;--MOVLW k : Chargement littéral de W (W = k)
                           adr<=std_logic_vector(unsigned(adr)+1);
        when "111110" =>   W<=std_logic_vector(unsigned(W)+ unsigned(k));--ADDLW k : Addition de W et k
                           adr<=std_logic_vector(unsigned(adr)+1);
        when "111100" =>   W<=std_logic_vector(unsigned(W)- unsigned(k));--SUBLW k : Soustraction de W et k
                           adr<=std_logic_vector(unsigned(adr)+1);
        when "111001" =>   W<=std_logic_vector(unsigned(W)AND unsigned(k));--ANDLW k : ET littéral avec W (W = k AND W)
                           adr<=std_logic_vector(unsigned(adr)+1);
        when "111000" =>   W<=std_logic_vector(unsigned(W)or unsigned(k));--IORLW k : OU Inclusif littéral avec W (W = k OR W)
                           adr<=std_logic_vector(unsigned(adr)+1);
        when "111010" =>   W<=std_logic_vector(unsigned(W)xor unsigned(k));--XORLW k : OU exclusif or littéral avec W (W = k XOR W)
                           adr<=std_logic_vector(unsigned(adr)+1);

       when others   =>    romout := "--------------";


    end case;

  Insto<=Inst;
  Wout <= W;

 end if;

end process;


end;
```

Après la compilation est :



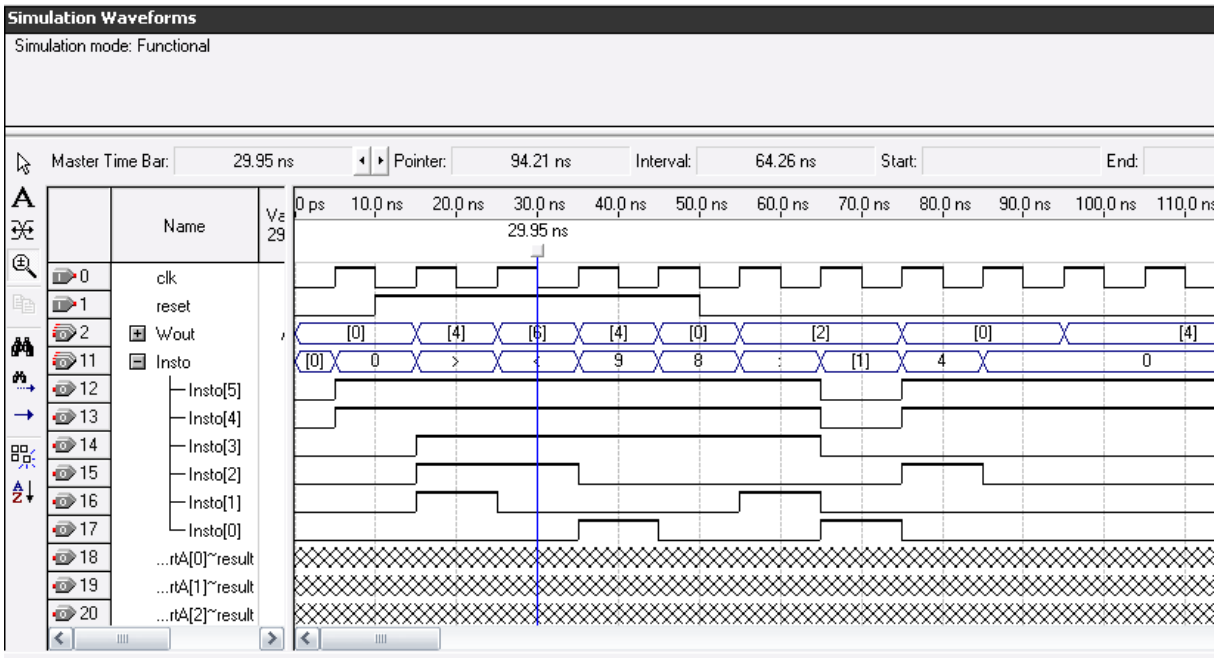


Figure III.33 : chronogramme de simulation en fonctionnel de programme principale

Le chronogramme suivant est :

Chargement de W avec 4, W=>4 ; et après adition de W et 2, W=>6 ; soustraction de W et 2, W=>4 ; et W and 2, W=>0 ; W or 2, W=>2 ; et W xor 2, W=>0.

III.4. Conclusion :

Dans ce chapitre on a implémenté un microcontrôleur PIC 16F84 sur FPGA dans l'environnement du logiciel Quartus II. Pour ce faire, on réalise la conversion du langage assembleur PIC 16F84 au langage VHDL. Cette opération se compose de 4 parties :

-conversion des instructions assembleur PIC16F84 en VHDL.

-écriture du langage assembleur en hexadécimale dans la ROM en langage VHDL.

-compilation du programme VHDL par Quartus II.

-implémentation du programme VHDL sur l'FPGA altéra cyclone II.

Enfin, nous avons pu implémenter certaines des instructions "PIC" sur le "FPGA, ce qui prouve qu'on peut vraiment implémenter un PIC16F84 sur le FPGA à utilisation VHDL.

Conclusion générale

Au bout de notre cursus en master "Electronique des Systèmes Embarqués ", nous avons été chargés de réaliser un projet de fin d'études.

Notre travail s'est basé sur créer un programme sur le circuit programmable "FPGA" pour simuler les fonctions du circuit intégré PIC 16f84.

Pour aboutir à notre objectif, nous avons suivi les étapes suivantes :

-conversion des instructions assembleur PIC16F84 en VHDL.

-écriture du langage assembleur en hexadécimale dans la ROM en langage VHDL.

-compilation du programme VHDL par Quartus II.

-implémentation des instructions de PIC avec VHDL sur l'FPGA altéra cyclone II.

Enfin, nous avons pu implémenter certaines des instructions "PIC" sur le "FPGA.

Comme perspectives, on propose de simuler d'autre fonctions du pic16f84 comme les instructions de lecture/écriture des ports, communication avec le port série ainsi que simuler d'autre microcontrôleurs.

Bibliographie

[1] Adjiba Brahim et Chalghoum Abdelmonaim « **Commande des équipements électriques par microcontrôleurs "Simulations et Réalisations"** » thèse MASTER ACADEMIQUE, Université Echahid Hamma Lakhdar d'El-Oued 2015

[2] PIC16F84 " **Philippe Hoppenot Professeur dans université d'evry "**

[3] Microchip : "**PIC16F8X – 18-pin Flash/EEPROM 8-bit microcontrollers DS30430C,**
-http://www.microchip.com/1010/suppdoc/

[4] Les microcontrôleurs **PIC** de **Microchip** Le 16F84.pdf

[5] https://elearn.univ-ouargla.dz/2013-2014/courses/MEP805/document/...

[6] www.**cder**.**dz**/vlib/**bulletin**/**pdf/bulletin_024_05**.**pdf**

[7] http://proxacutor.free.fr/index.htm

[8] *Synopsys,* « ***SaberHDL : language-Independant Mixed –Signal Multi-Technology Simulator*** », Synposys Inc, Etats-Unis d'Amérique 2003.

[9] *D.SMITH* « ***HDL Chip Design: A pratical Guide for Designing, Synthesis & Simulating Asics &FPGAs using VHDL or Verilog*** » Doone Pubns, 2006

[10] *D. DECKERS* « ***Composants programmables*** » Cours de 3è Bachelier en Electronique appliquée, Institut Supérieur Industriel de Mons Haute Ecole de la Communauté française en Hainaut Unité Electronique 2008-2009

[11] https://fr.wikipedia.org/wiki/Circuit_logique_programmable.

[12] Alain Vachoux « ***Le langage VHDL*** » Laboratoire de Systèmes Microélectroniques alain.vachoux@epfl.ch / Ecole polytechnique fédérale de lausanne, 2003

[13] Céline GUILLEMINOT « ***ÉTUDE ET INTÉGRATION NUMÉRIQUE D'UN SYSTÈME MULTICAPTEURS AMRC DE TÉLÉCOMMUNICATION BASÉ SUR UN PROTOTYPE VIRTUEL UTILISANT LE LANGAGE DE HAUT NIVEAU VHDL-AMS*** » MEMOIRE de THESE DOCTORAT de L'UNIVERSITE de TOULOUSE II, 2005

[14] Philippe LECARDONNEL & Philippe LETENNEUR « ***Introduction à la Synthèse logique V.H.D.L.*** » / Lycée Julliot de la Morandière – GRANVILLE – 2001

[15] Eduardo Sanchez EPFL « ***Le langage VHDL*** » / PDF

[16] Etienne Messerli « ***Introduction au langage VHDL*** » Unité d'enseignement : Bases des systèmes logiques (BSL), Mise à jour le 6 octobre 2008

[17] support de cours à Mr dhiabi fathi en 3éme année électronique, université Mohamed khider / 2018

[18] Mr. Nachef Toufik « ***Implémentation d'une instrumentation sur un FPGA*** » / mémoire de magister, université mouloud mammeri de tizi- ouzou, 2011