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

MÉMOIRE DE MASTER

Sciences et Technologies
Télécommunications
Réseaux et télécommunications

Préparé Par : 26-06-2022

Hammani Noudjoud

Le :

Applications de la FPGA sur les circuits combinatoires et séquentiels

---

Le Jury de soutenance :

Melle.	**TOUMI Abida**	**Pr Univ Biskra**	**Président**
Mr.	**SBAA Salim**	**Pr Univ Biskra**	**Encadreur**
Melle.	**ZEHANI Soraya**	**MCA Univ Biskra**	**Examinateur**

Année universitaire:2021-2022

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

MÉMOIRE DE MASTER

Sciences et Technologies
Télécommunications
Réseaux et télécommunications

Applications de la FPGA Sur Les Circuits Logiques et Séquentiels

---

Le : 26 Juin 2022

Présenté par :

Hammani Noudjoud

Avis favorable de l'encadreur :

Pr. SBAA Salim

Signature Avis favorable du Président du Jury

Mme. TOUMI Abida

Cachet et signature


﴿رَبِّ أَوْزِعْنِي أَنْ أَشْكُرَ نِعْمَتَكَ الَّتِي أَنْعَمْتَ عَلَيَّ وَعَلَى وَالِدَيَّ وَأَنْ أَعْمَلَ
صَالِحًا تَرْضَاهُ وَأَدْخِلْنِي بِرَحْمَتِكَ فِي عِبَادِكَ الصَّالِحِينَ﴾
النمل(19)
#ايها_المريد
www.almoreed.com


(رَبِّ اغْفِرْ لِي وَلِوَالِدَيَّ وَلِمَنْ دَخَلَ بَيْتِيَ مُؤْمِنًا وَلِلْمُؤْمِنِينَ وَالْمُؤْمِنَاتِ
وَلَا تَزِدِ الظَّالِمِينَ إِلَّا تَبَارًا)
نوح: 28
هويتنا

الإهداء

اللهم لك الحمد قبل أن ترضى ولك الحمد إذا رضيت ولك الحمد بعد الرضا,
وصلت رحلتي الجامعية إلى نهايتها بعد تعب ومشقة وها أنا ذا أختم بحث تخرجي بكل همة ونشاط.

وأمتن لكل من كان له الفضل في مسيرتي من يزيدني انتسابي له وذكره فخرا واعتزاز وإلى من سهر الليالي من أجل تربيتي وتعليمي , وجعلني أكبر في أزكى وأطهر فضيلة أبي العزيز رحمه الله واسكنه فسيح جناته .

إلى قرة عيني , إلى من جعلت الجنة تحت قدميها ... إلى التي حرمت نفسها وأعطتني , ومن نبع حنانها سقتني .... إلى من وهبتني الحياة أمي العزيزة حفظها الله وأطال في عمرها .

إلى إخوتي الأحباء والزوج الكريم حفظهم الله . إلى اعز أصدقائي إلى كل من شاءت الأقدار أن تجمعني بهم حدائق الدراسة من أشقاء.

كما ارفع كلمة الشكر للبروفيسور المشرف سبع سليم وعلى نصائحه وتوجيهاته والمعلومات القيمة التي ساهمت في إثراء موضوع دراستنا في جوانبها المختلفة .

وفي الأخير لا يسعنا إلا أن ندعو الله عزوجل أن يرزقنا السداد والثبات والغنى .

شكر وتقدير

بادئ ذي بدء، سأكون ممتنًا وشاكرًا لله الرحمن الرحيم على منحي الصبر والقوة والإرادة لإكمال هذا العمل

.

أود أن أعبر عن تقديري الخالص لمشرفي المحترم والمتعاطف الأستاذ سبع سليم على صبره وتوجيهه ودعمه

أود أن أشكر الأستاذ ذيابي فتحي جزيل الشكر على مساعدتي

أود أن أشكر أعضاء لجنة التحكيم على وقتهم في قراءة هذه الرسالة

أود أن أشكر جميع الأساتذة على لطفهم في الرد على المقابلة

أخيرًا و ليس أخرا، أنا ممتن لكل من ساعدني بطريقة أو بأخرى على تحقيق هذا العمل

Résumé

L'électronique moderne se dirige de plus en plus vers le numérique qui présente de nombreux avantages sur l'analogique : grande insensibilité aux parasites, reconfigurabilité, facilité de stockage de l'information. Les circuits programmables ont été développés dans ces dernières décennies pour remplacer la conception hardware des fonctions logiques, néanmoins, ces derniers sont conçus à partir des circuits logiques combinatoires et séquentiels. Parmi ces circuits sont PLA, PAL, CPLD et terminent par FPGA qui trouvent des applications courantes dans plusieurs domaines civiles et militaires. Ce mémoire va exploiter les connaissances des circuits combinatoires et séquentiels afin de les rendre programmable implémentés sur des circuits FPGA. Ce travail va servir comme une initiation à la programmation de ces circuits par la langage VHDL, à partir du logiciel Quartus 2 sur une carte ALTERA DE1.

Mots clés : PAL, CPLD, FPGA, VHDL, circuits combinatoires et séquentiels.

Abstract

Modern electronics is moving more and more towards digital, which has many advantages over analog: great insensitivity to interference, reconfigurability, ease of storing information. Programmable circuits have been developed in recent decades to replace the hardware design of logic functions; however, these are designed from combinatorial and sequential logic circuits. Among these circuits are PLA, PAL, CPLD and terminated by FPGA which find common applications in several civil and military fields. This thesis will exploit the knowledge of combinatorial and sequential circuits in order to make them programmable implemented on FPGA circuits. This work will serve as an introduction to the programming of these circuits by the VHDL language, from the Quartus 2 software on an ALTERA DE1 card.

Keywords: PLA, PAL, CPLD, FPGA, VHDL, combinatorial and sequential circuits.

الملخص

تتجه الإلكترونيات الحديثة أكثر فأكثر نحو الرقمية ، والتي تتمتع بالعديد من المزايا على التناظرية: عدم حساسية كبيرة للتداخل ، وإعادة التكوين ، وسهولة تخزين المعلومات. تم تطوير الدوائر القابلة للبرمجة في العقود الأخيرة لتحل محل تصميم الأجهزة للوظائف المنطقية ؛ ومع ذلك ، تم تصميم هذه من دوائر المنطق الاندماجي والمتسلسل. من بين هذه الدوائر PLA و PAL و CPLDوالتي تم إنهاؤها بواسطة FPGA والتي تجد تطبيقات مشتركة في العديد من المجالات المدنية والعسكرية. ستستغل هذه الأطروحة معرفة الدوائر التوافقية والمتسلسلة من أجل جعلها قابلة للبرمجة في دوائر .FPGA سيكون هذا العمل بمثابة مقدمة لبرمجة هذه الدوائر بواسطة لغة VHDL ، من برنامج Quartus 2 على بطاقة.ALTERA DE1

الكلمات الرئيسية: PLA، PAL، CPLD، FPGA، VHDL ، الدوائر التوافقية والمتسلسلة.

Liste des Figures

Liste des Figures

Liste des Figures

Liste des Tableaux

Liste des Acronymes

CPLDs	**Complex Programmable Logic Device**
CLBs	**Configurable Logic Blocs**
DCB	**Décimal Codé Binaire**
FPGAs	**Field Programmable Gate Arrays**
GAL	**Generic Array Logic**
HDL	**Hardware Description Language**
IOBs	**Input Output Blocs**
LE	**Logic Element**
LUT	**Look-Up Table**
MROM	**Mask Read Only Memory**
OLMC	**Ouput Logic Macro-Cell**
PLD	**Programmable Logic Device**
PLA	**Programmable Logic Array**
PAL	**Programmable Array Logic**
PLD	**Programmable Logic Device**
RS	**Register Synchrone**
RAM	**Random Access Memory, ou mémoire à accès aléatoire**
ROM	**Read-Only Memory ou mémoire à lecture seule**
SOP	**SOPhisticated behavioral modeling**
SPLD	**Simple Programmable Logic Device**
VHSIC	**Very High Speed Integrated Circuit**

Table des Matières

Introduction Générale

Le développement des circuits électroniques est transféré de l'analogique vers le numérique ne s'est pas achevé par les circuits logiques combinatoires et séquentiels, par contre le cout de ces circuits et leurs problèmes d'intégration a grand échelle posent de plus en plus ces problèmes.

Les solutions envisagées pour résoudre ces problèmes sont faites par les circuits programmables qui nous permettent de gagner surtout le cout en profitant du développement de l'intégration grand échelle des circuits logiques.

La structure des machines de von Neumann fait partie des systèmes programmables, par contre son application en temps réel est limitée par la rapidité des systèmes réels et la dynamique des algorithmes de traitement de ces derniers.

Les solutions de traitement des systèmes en temps réels sont améliorées par l'apparition des circuits programmable PLD (Programmable Logic Device), d'où ils dérivent les circuits FPGA (Field-Programmable Gate Array). Ce qui explique que le temps d'exécution d'un programme de simulation des systèmes représenté par des circuits programmable, nécessite des séquences, d'où la limitation des systèmes a microprocesseurs. Le développement des circuits programmables tels que FPGA nous permet de basculer vers l'exécution en temps réels des systèmes

Pour cette raison que notre mémoire est basé sur l'étude des circuits FPGA et leurs programmations

L'organisation générale de notre mémoire est articulée sur quatre chapitres :

- **Le chapitre 1** : traite les circuits combinatoires et les circuits séquentiels qui font la base de représentation et de synthèse de tous les systèmes numériques, on a montré la différence entre un circuit combinatoire et un circuit séquentiel en présentant quelques exemples de ces circuits
- **Le chapitre 2** : évoque les circuits programmables PLD qui font la base des différents types de conception telle que PAL, PLA, GAL et FPGA. Nous avons montré par la suite la structure du circuit FPGA. La carte DE1 de ALTERA qui contienne le FPGA utilisé, nécessite un moyen de communication avec l'ordinateur qui va par la suite la programmer, d'où l'utilisation d'un logiciel d'adaptation entre carte et micro ordinateur qui se fait à travers le logiciel qui est disponible dans notre cas, Quartus2.
- **Le chapitre 3** : Comme les FPGAs sont programmables, l'utilisation de Langage de programmation de ces circuits est nécessaire, dans ce chapitre, nous avons étudié le langage

VHDL comme langage de programmation de la FPGA. Sans oublier de signaler qu'il existe d'autre langage comme le VERILOG. Nous nous sommes seulement intéressés par les détails du langage VHDL.

- **Le chapitre 4** : Après avoir étudié dans les chapitres qui précédent, les circuits programmables et leurs langage de programmation, nous a établie des applications concernant les circuits combinatoires et séquentiels appliqués sur FPGA de la carte DE1 de ALTERA.
- **Finalement**, une conclusion générale est présentée pour souligner les résultats obtenus

Chapitre I : Les Circuits Combinatoire et Circuits Séquentiels

I.1 Introduction

Il existe deux types de circuits logiques : combinatoires, et séquentiels.

Les circuits combinatoires sont créés à partir de portes logiques dont la sortie dépend seulement des entrées actuelles. Dans les circuits séquentiels, il y a du feedback entre la sortie et les entrées : on a des éléments de mémoire, et la sortie dépend des entrées actuelles et de l'état de la mémoire. [11]
Les circuits séquentiels utilisés dans la vie courante ont besoin de mémoire. Ce chapitre présente les méthodes de base de stockage d'information. Les circuits combinatoires présentés jusqu' à maintenant n'avaient aucun élément de mémoire : la sortie dépendait seulement des entrées. Pour les circuits séquentiels, la sortie dépend non seulement des entrées actuelles, mais aussi de la sortie précédente. Ces circuits agissent comme éléments de stockage et comme mémoire. L'analyse de ces circuits est donc différente de celle des circuits combinatoires. [12]

I.2 Circuits combinatoires

Il existe plusieurs circuits et on présente les plus utilisés

I.2.1 Porte Inverseuse (NOT)

La porte de base est l'inverseuse. Par la parole, nous pourrions dire : «A est égal à non B»; ainsi, cette porte est aussi souvent appelée une porte non. Le symbole représentatif est le même que celui du tampon, sauf qu'une bulle d'inversion (un cercle) est placée sur la sortie. [1]

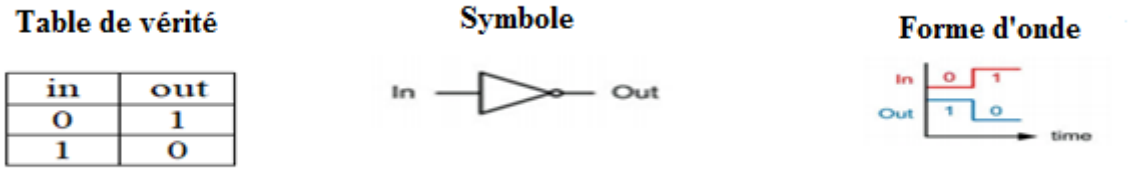


Figure I.1:Porte inverseuse tableau de vérité, Symbole de la fonction logique et forme d'onde logique [1]

I.2.2 Porte ET (AND)

La fonction ET est obtenue en multipliant les entrées $F=A \cdot B$. La table de vérité et le symbole associé à cette fonction sont [8] :

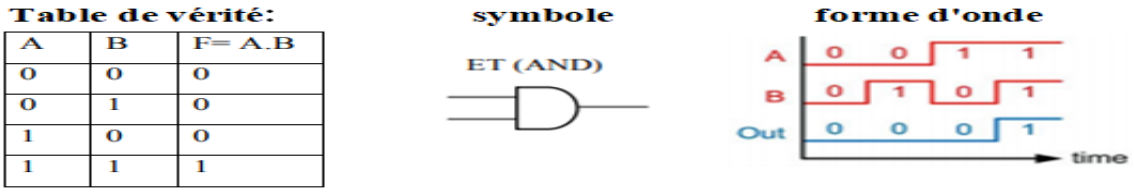


Figure I.2: Porte ET table de vérité, symbole de la fonction logique et forme d'onde logique [1]

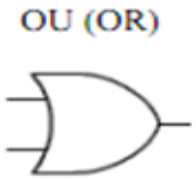
I.2.3 Porte OU (OR)

La fonction OU est obtenue par la somme des entrées du système $F = A + B$. La table de vérité et le symbole associés à cette fonction sont [8] :

Table de vérité

A	B	F= A+B
0	0	0
0	1	1
1	0	1
1	1	1

symbole



forme d'onde

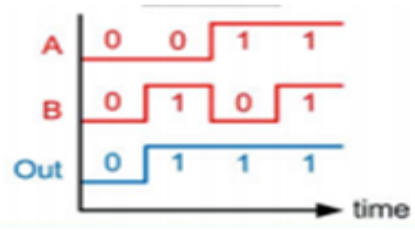


Figure I.3: table de vérité, Symbole de porte OR d'entrée, fonction logique et forme d'onde logique [1]

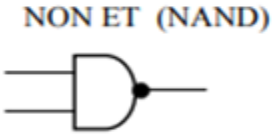
I.2.4 Porte NON_ET (NAND)

La fonction NON_ET est obtenue en complémentant la fonction ET : $\boldsymbol{F = \overline{A.B}}$ la table de vérité et le symbole associé à cette fonction sont [8] :

Table de vérité

A	B	F= A.B	$F = \overline{A.B}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

symbole



forme d'onde

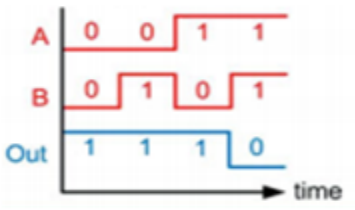


Figure I.4:Table de vérité, Symbole de porte NAND d'entrée, , fonction logique et forme d'onde logique [1]

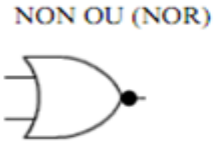
I.2.5 Porte ON_OU(NOR)

La fonction NON_OU est obtenue en complémentant la fonction OU : $\boldsymbol{F = \overline{A + B}}$
La table de vérité et le symbole associé à cette fonction sont [8] :

Table de vérité

A	B	F= A+B	$F = \overline{A + B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

symbole



forme d'onde

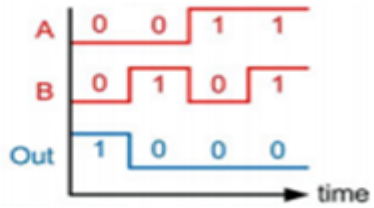


Figure I.5: Table de vérité de porte NOR, symbole et fonction logique et forme d'onde logique [1]

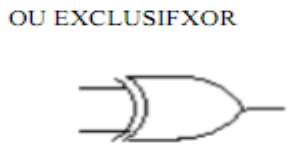
I.2.6 Porte OU EXCLUSIF (XOR)

La fonction OU EXCLUSIF ne vaut 1 que si les deux entrées sont différentes.
Elle s'écrit $F = A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$.
La table de vérité et le symbole associé à cette fonction sont [8]:

Table de vérité

A	B	$F = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

symbole



forme d'onde

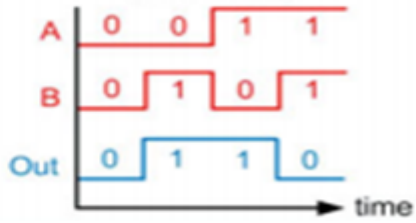


Figure I.6: Table de vérité de porte XOR, Symbole et fonction logique et forme d'onde logique [1]

I.2.7 Décodeur

Un décodeur est un circuit qui prend dans un code binaire des sorties qui sont affirmées pour les valeurs spécifiques de ce code. Le code peut être de n'importe quel type ou taille. [1]

Les décodeurs sont généralement nommés selon leurs fonctions, m-à-n (par exemple, un décodeur 2 à 4). Les décodeurs typiques ont n entrées et $2^n = m$ sorties. Chaque combinaison d'entrées n'active qu'une seule sortie à la fois : la sortie activée correspond à la valeur binaire de l'adresse. On a souvent un signal de contrôle (enable). La table de vérité d'un décodeur 2 à 4 est montrée à la Tableau I.1. [11]

Entrées		**Sorties**			
A1	A0	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Tableau I.1: Table de vérité d'un décodeur 2 à 4

I.2.7.1 Décodeur avec entrée de validation

La plupart des décodeurs auront une entrée de validation. Cette entrée permet d'activer le fonctionnement du décodeur. Si EN= 0, toutes les sorties sont à 0. Si EN=1, le décodeur fonctionne normalement. On peut aussi avoir un signal de contrôle inversé $\overline{EN}$. [11]

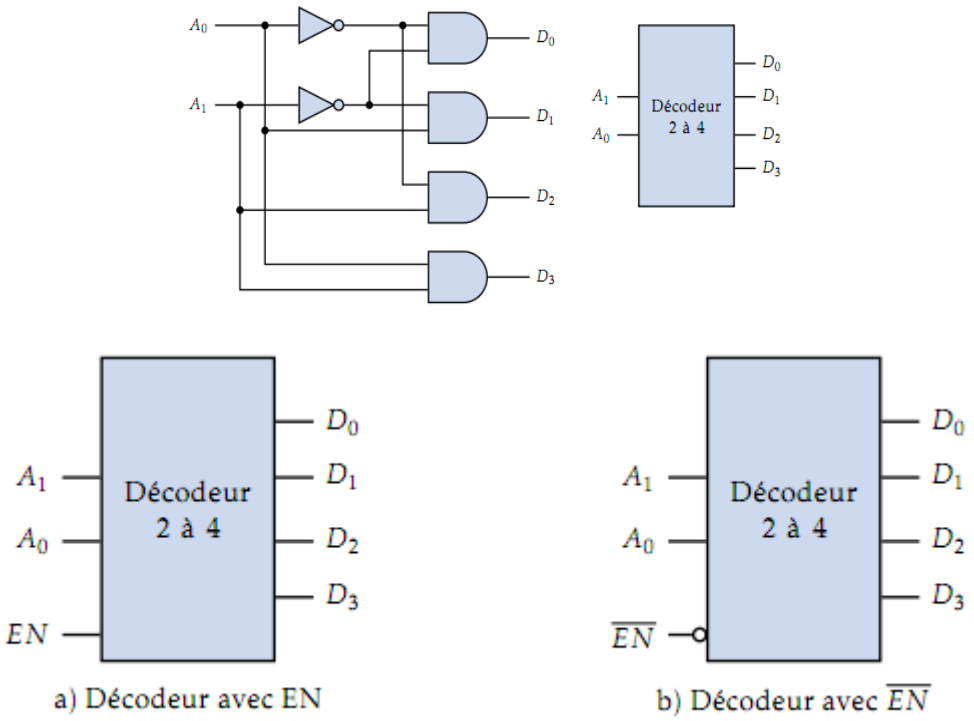


Figure I.7: Circuit et schéma d'un décodeur 2 à 4[11]

I.2.7.2 Synthèse avec les décodeurs :

Un décodeur peut-être utilisé pour faire la synthèse d'une fonction logique. Un décodeur génère à la sortie les 2^n min-termes des n variables d'entrée. Puisque toute fonction booléenne peut-être exprimée comme une somme de min-termes, on peut utiliser un décodeur avec une porte OU à la sortie pour créer la fonction voulue. Tout circuit combinatoire avec n entrées et m sorties peut-être réaliser avec un décodeur n à 2n et m portes OU. [11]

I.2.7.3 Synthèse de grands décodeurs

Des décodeurs avec des entrées de validation peuvent être combinés pour créer des plus gros décodeurs. Par exemple, on peut utiliser 2 décodeurs 3×8 pour faire un décodeur 4 ×16. La quatrième variable est utilisée pour activer un ou l'autre des décodeurs 3×8. Un exemple est montré à la figure I.8. L'entrée W active seulement un des deux décodeurs à la fois. [11]

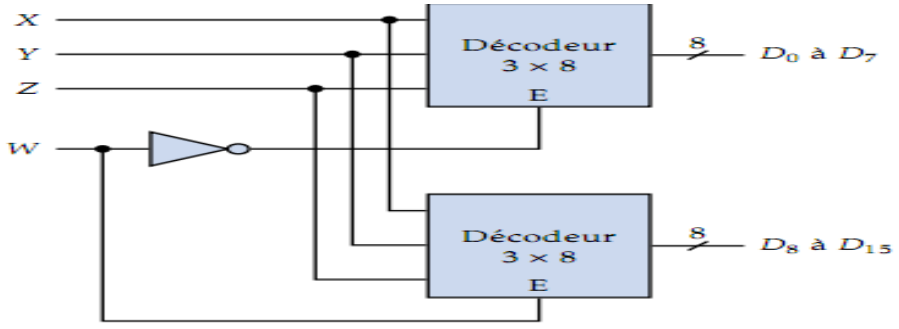


Figure I.8: Décodeur 4 ×16 créé avec deux décodeurs 3 × 8. **[11]**

I.2.8 Encodeurs

Un encodeur fonctionne à l'opposé comme un décodeur. Une assertion sur un port d'entrée spécifique correspond à un code unique sur le port de sortie. [8]

Symbole :

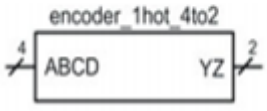


Table de vérité:

A	B	C	D	Y	Z
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Figure I.9: Symbole et Table de Vérité de l'Encodeur

I.2.9 Multiplexeur

Le multiplexeur est un circuit qui permet de sélectionner une entrée parmi plusieurs et acheminer cette entrée à une sortie unique. Le choix de l'entrée se fait par une série de lignes de sélection. Habituellement, on a 2^n entrées et n bits de sélection, et une seule sortie sur un multiplexeur. Les bits de sélections sont aussi appelés des adresses. Le multiplexeur est très utilisé dans les systèmes numériques et les ordinateurs. Il permet d'utiliser une seule ligne conductrice reliant plusieurs sources à une destination : chaque source partage la ligne conductrice. [11]

I 2.9.1 Multiplexeur 1 à 4 bits

Table de vérité

A	B	C	D	SUL0	SUL1	F
1	0	0	0	0	0	1
0	1	0	0	0	1	1
0	0	1	0	1	0	1
0	0	0	1	1	1	1

symbole

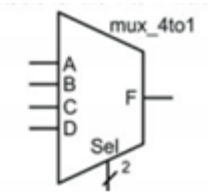


***Figure I.10: Table de vérité et symbole d'un multiplexeur 1 à 4 bits*[1]**

I.2.10 Démultiplexeurs

Un démultiplexeur fonctionne de manière complémentaire à un multiplexeur. Un démultiplexeur a une entrée qui est acheminée vers l'une de ses multiples sorties. La sortie active est dictée par une entrée sélectionnée. Un demux à n lignes sélectionnées qui choisissent d'acheminer l'entrée à l'une de ses 2^n sorties. Quand une sortie n'est pas sélectionnée, il produit une logique 0. [1]

Table de vérité

W	X	Y	Z	SUL0	SUL1
A	0	0	0	0	0
0	A	0	0	0	1
0	0	A	0	1	0
0	0	0	A	1	1

symbole

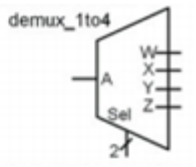


Figure I.11: Table de vérité et symbole d'un Démultiplexeurs 1 à 4 [1]

I.3 Circuits Séquentiels

I.3.1 Bascules

On peut avoir des problèmes de synchronisation si on utilise des verrous dans des circuits. La sortie d'un verrou change aussi longtemps que l'entrée d'activation est haute. Ceci veut dire que si l'entrée du verrou varie (ou n'est pas stable) pendant que l'activation est haute, la sortie variera elle aussi, ce qui peut générer des erreurs. Pour remédier à ce problème, on utilise plutôt des bascules, au lieu des verrous, pour avoir une mémoire. [11]

I.3.1.1 Bascule RS (Bascule Asynchrone sans Horloge) :

Est constituée par deux entrées (S mise à 1 (Set)) et (R mise à 0 (Reset)) et de deux sorties Q et $\overline{\mathbf{Q}}$. [16]

- ✓ S mise à 1 implique que Q=1 (Q est forcé à un par S)
- ✓ R mise à 0 implique que Q=0 (Q est forcé à zéro par R)

Table de vérité

S	R	Q_t	Q_{t+1}	$\overline{Q}_{t+1}$	
0	0	0	0	1	Garder l'état
0	0	1	1	0	précédent
1	0	0	1	0	Mise à 1
1	0	1	1	0	
0	1	0	0		Mise à 0
0	1	1	0		
1	1	0	×	×	Indéterminé
1	1	1	×	×	

Schéma logique

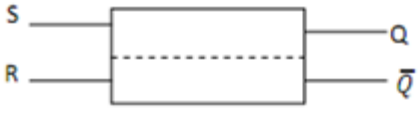


Figure I.12: Bascule RS et symbole

I.3.1.2 Bascule JK

Le symbole logique et la table de vérité d'une bascule JK sont donnés à la figure I.11. La bascule JK permet d'exécuter trois opérations : placer la sortie à 0, placer la sortie à 1, ou faire le complément de la sortie actuelle. [11]

Table de vérité

J	K	Q(t+1)
0	0	Q(t) Aucun changement
0	1	0 Reset
1	0	1 set
1	1	$\bar{Q}(t)$ complément

Schéma logique

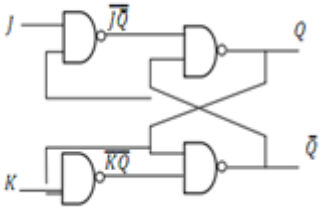


Figure I.13: Bascule JK et symbole

I.3.1.3 Bascule JK synchrone

Elle est contrôlée par le signal T

Table de vérité :

T	K	J	Q_{t+1}	$\bar{Q}_{t+1}$
0	×	×	Q_t	$\bar{Q}_t$
1	0	0	Q_t	$\bar{Q}_t$
1	0	1	1	0
1	1	0	0	1
1	1	1	$\bar{Q}_t$	Q_t

Schéma logique :

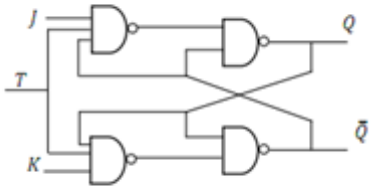


Figure 14: Bascule JK synchrone [8]

I.3.1.4 Bascule D

La bascule D est utilisée pour éliminer l'état indéterminé (interdit), il est résumé par les deux entrées R et S par une seule entrée D telle que S=D et R=$\bar{D}$.

Table de vérité

T	D	Q_{t+1}	$\bar{Q}_{t+1}$
0	0	Q_t	$\bar{Q}_t$
0	1	Q_t	$\bar{Q}_t$
1	0	0	1
1	1	1	0

Chronogramme :

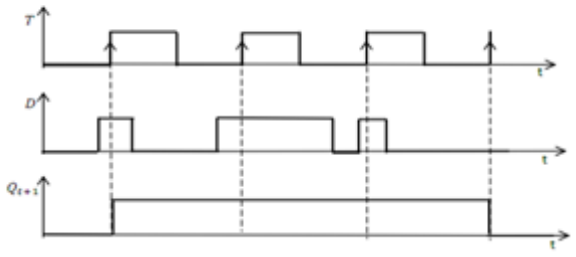


Figure I.15: table de vérité et chronogramme pour une bascule D [8]

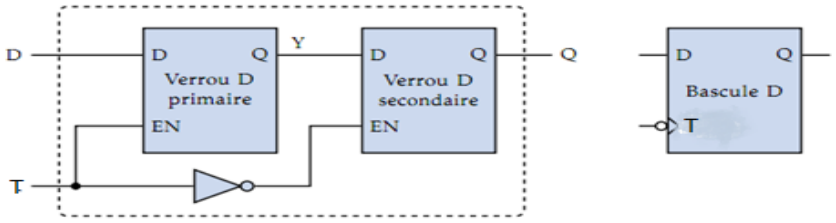


Figure I.16: Bascule D et symbole

I.3.1.5 Bascule T

Le symbole logique et la table de vérité d'une bascule T sont donnés à la figure .1.17 . La bascule T (toggle) permet de faire le complément ou non de l'état. **[11]**

Table de vérité

T	Q_t	Q_{t+1}	$\overline{Q}_{t+1}$	
0	0	0	1	Mémorisation
0	1	1	0	de l'état
1	0	1	0	Fonctionnement
1	1	0	1	en bascule T

Chronogramme

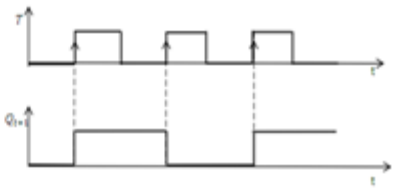


Figure I.17: Table de vérité et Chronogramme d'un Bascule T

Equation logique : [8]

$Q_{t+1}=\overline{Q_t}$

I.3.2 Compteurs

I.3.2.1 Définition

Les compteurs sont des ensembles des bascules montées en série ou en parallèle pour compter soit incrémenter ou décrémenté, on distingue deux types : les compteurs Asynchrone et Synchrone. [8]

I.3.2.2 Compteur binaire en cascade

Un compteur binaire en cascade est une série de bascules ou la sortie d'une bascule est branchée à l'entrée CLK de la bascule suivante. La bascule qui contient le bit le moins significatif reçoit les impulsions d'entrée. Un exemple de compteur à 4 bits avec des bascules D . À chaque fois qu'A0 change de 1 à 0, la valeur d'A1 est complémentée. À chaque fois qu'A1 change de 1 à 0, la valeur d'A2 est complémentée, et ainsi de suite. [13]

I.3.2.3 Compteur DCB en cascade

Un compteur Décimal Codé Binaire (DCB) va passer à travers la séquence de 0 à 9 puis recommencer à 0.Il faut 4 bascules pour créer ce circuit, puisque le DCB nécessite 4 bits. La séquence est montrée à la figure I.18. [13]

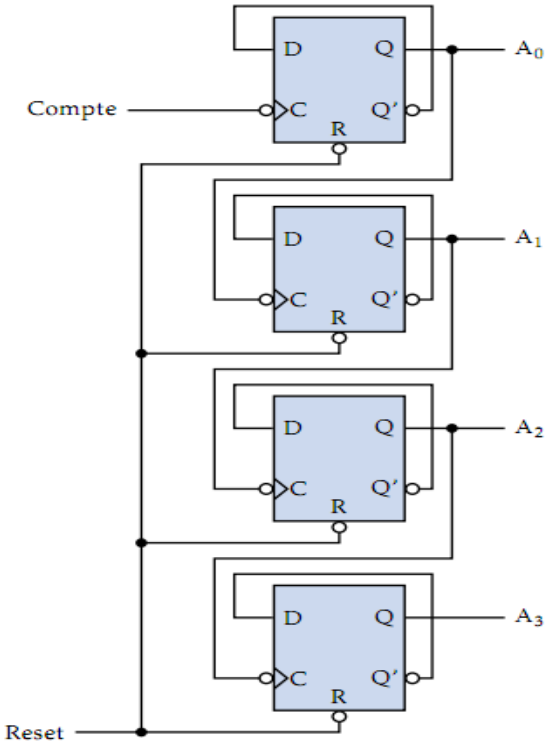


Figure I.18: Compteur DCB en cascade à 4 bits [13]

I.3.2.4 Compteurs asynchrones

Ils sont formés par des bascules montées en série tel que chaque bascule donne l'impulsion à la bascule suivante :

Un compteur modulo n peut compter de zéro jusqu'à (n-1), tel que : Si n=8 donc est compté de 0 à 7, et si n=10 donc est compté de 0 à 9.

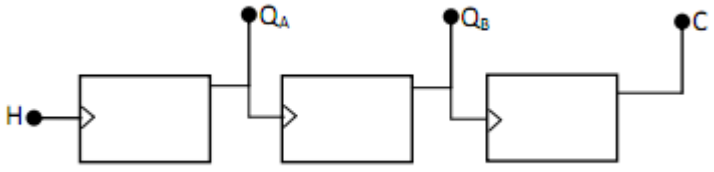


Figure I.19: Schéma d'un compteur Asynchrone [8]

Chronogramme :

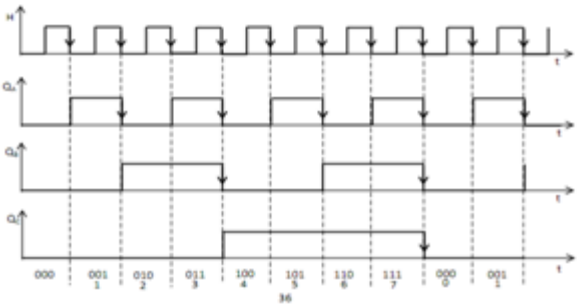


Table de vérité :

Q_A	Q_B	Q_C	H
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Figure I.20: Table de vérité et Chronogramme d'un Compteurs asynchrones

I.3.2.5. Compteurs synchrones

- Différents des compteurs en cascade: l'horloge est appliquée 0 toutes les bascules
- Une horloge commune active toutes les bascules simultanément, plutôt qu'une à la fois. *[8]*

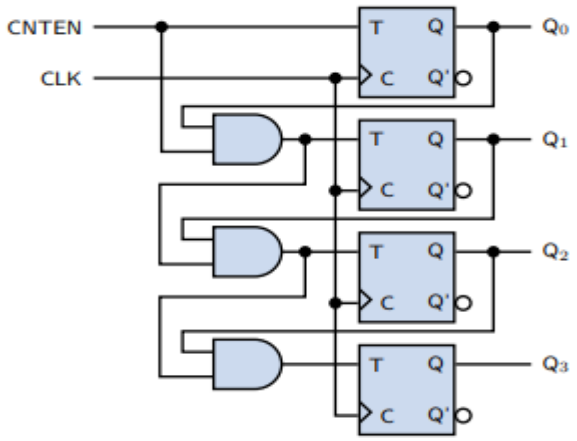


Figure I.21: Compteur binaire synchrone `a 4 bits

- Le circuit de la figure 8 possède aussi une entrée d'activation globale (CNTEN)
- Chaque bascule T va seulement inverser si CNTEN = 1
- Attention `a la vitesse de l'horloge: si l'horloge est trop rapide, il est possible qu'un changement au LSB n'ait pas le temps de se propager au MSB*[8]*

I.3.3 Les Registres

I.3.3.1 Définition

Un registre qui peut décaler de l'information binaire d'une cellule à une autre dans une direction spécifique est appelé un registre à décalage. Un registre à décalage est constitué d'une série de bascules en cascade, ou la sortie d'une bascule est branchée à l'entrée de la bascule suivante. Un exemple est montré à la Fig.1.22. [13]

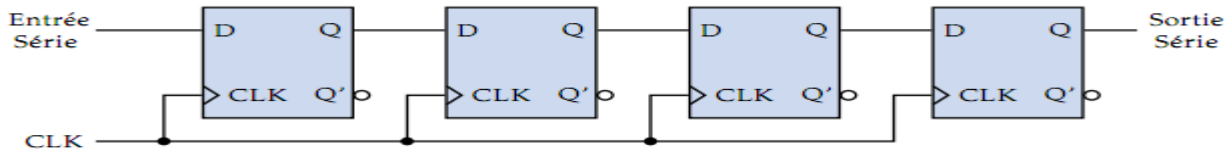


Figure I.22: Registre à décalage de 4 bits

I.3.3.2 Transfert sériel

Un système numérique est dit sériel, si l'information est transformée et manipulée 1 bit à la fois. L'information est transférée un bit à la fois du registre source au registre de destination. Un exemple est montré à la Figure I.23 , ou on a aussi un système pour contrôler quand l'information est transférée : on a une entrée d'activation, l'entrée CTRL permet de contrôler quand l'horloge globale est passée aux registres, ce qui permet de contrôler le transfert des données. [13]

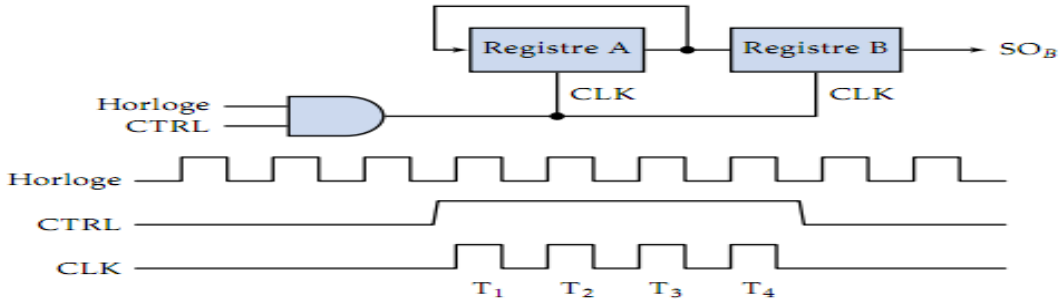


Figure I.23: Transfert sériel

I.3.3.3 Addition sérielle

La plupart des opérations dans un ordinateur sont exécutées en parallèle, parce que les calculs sont plus rapides. Cependant, ceci nécessite des circuits plus complexes et plus gros. Dans des circuits ou la taille est importante, on utilise parfois des opérations sérielles plutôt que parallèle.

Le circuit de la figure ci dessous montre un additionneur sériel. On utilise des registres pour faire l'addition bit par bit. Le report de sortie est transféré à une bascule D, qui devient ensuite le report d'entrée pour les 2 bits suivants. [13]

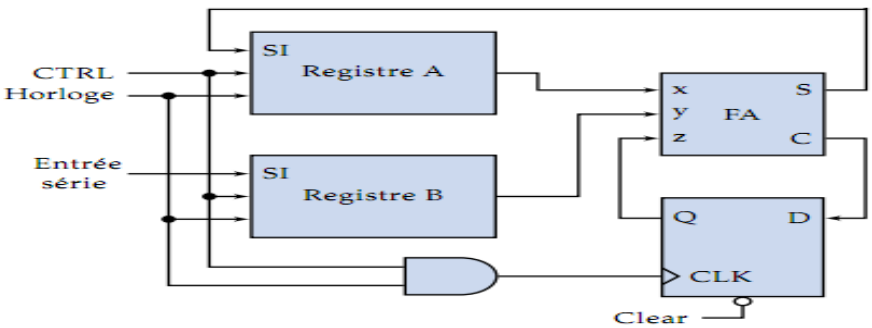


Figure I.24: Additionneur sériel

I.3.3.4 Type de registres

Il existe plusieurs types de registres : [8]

- Registre à entrées parallèles et sorties parallèles (Registre à chargement parallèle).
- Registre à entrée série et sortie série
- Registre à entrée série et sortie parallèle.
- Registre à entrée parallèle et sortie série.
- Registre à décalage circulaire

I.4 Conclusion

Dans ce chapitre, on a présenté les circuits de bases combinatoires et séquentiels par lesquels tous les circuits logiques complexes et programmables sont basés. On a vu comment on diffère entre les circuits logiques combinatoires et les circuits séquentiels. L'utilisation de ces circuits est nécessaire dont la conception des circuits complexes tels que PLA, CPLD et FPGA.

Chapitre II : Circuits programmables FPGA

II.1 Introduction

Une mémoire est un élément clé des systèmes numériques. Lorsqu'il y a traitement d'information, les données en mémoire sont transférées à des registres, puis une série d'opérations sont effectuées avec ces informations. À la fin des opérations, les résultats sont à nouveau transférés en mémoire. [14]

Il existe deux types de mémoire utilisées dans les systèmes numériques : RAM (Random Access Memory, ou mémoire à accès aléatoire) et ROM (Read-Only Memory ou mémoire à lecture seule). Les deux opérations de base d'accès à la mémoire sont l'écriture et la lecture. Un RAM peut effectuer les deux opérations, tandis qu'un ROM peut seulement effectuer l'opération de lecture. On ne peut pas modifier l'information écrite dans un ROM. [14]

Un ROM est un dispositif logique programmable (PLD). La programmation de la mémoire est une opération hardware qui permet de spécifier les bits qui sont en mémoire. [14]

Il existe d'autres types de mémoire programmables : [14]

- PLA : programmable logic array.
- PAL : programmable array logic.
- FPGA: field-programmable gate array.

Ce chapitre donne un aperçu des dispositifs logiques programmables (DLP). Le terme PLD est utilisé comme description générique pour tout circuit qui peut être programmé pour implémenter la logique numérique. La technologie et les architectures des DLP ont évolué au fil du temps. Le but de ce chapitre est de fournir une compréhension de base des principes des dispositifs logiques programmables.

A travers cette étude nous allons :

- Décrire l'architecture de base et l'évolution des dispositifs logiques programmables.
- Décrire l'architecture de base des circuits programmables (FPGA)

II.2 SPDLs (PLDs simples)

Les services PAL, PLA et GAL sont collectivement appelés DLP simples (DLPS). Voici une description de chacune de ces architectures. [14]

PLDs	Simple PLD (SPLD)	PAL PLA Registered PAL/PLA GAL
	Complex PLD (CPLD)	
	FPGA	

Tableau II.2: Un résumé de l'évolution des DLP est présenté dans le tableau ci-dessous

II.2.1 Réseau logique programmable (PLA)

L'un des premiers DLP commerciaux développés à l'aide de la technologie des circuits intégrés modernes était Réseau logique programmable (PLA). En 1970, Texas Instruments a introduit le PLA avec une architecture qui a soutenu la mise en œuvre de la somme arbitraire des produits expressions logiques. Le PLA a été fabriqué avec un réseau dense de portes ET, appelé un plan ET, et un réseau dense de portes OU, appelé plan OR. Les entrées au PLA avaient chacune un inverseur afin de fournir la variable d'origine et son complément. Des expressions logiques SOP (SOPhisticated behavioral modeling ou modélisation comportementale sophistiquée) arbitraires pourraient être implémentées en créant des connexions entre les entrées, le plan AND et le plan OR. Les PLA d'origine ont été fabriqués avec tous les fonctions, à l'exception des connexions internes pour mettre en œuvre les fonctions SOP. Lorsqu'un client a fourni l'expression SOP souhaitée, les connexions ont été ajoutées comme l'étape de fabrication. Cette technique de configuration était semblable à une approche MROM (Mask Read Only Memory). La figure II.1 montre l'architecture de base d'un PLA. [1]

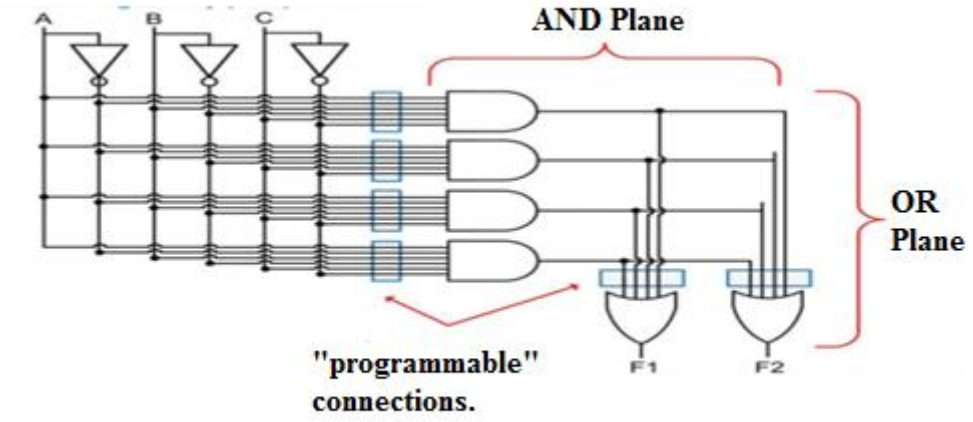


Figure II.1: Architecture de réseau logique programmable (PLA).[1]

II.2.2 Logique de réseau programmable (PAL)

Comme le circuit PLA, l'augmentation du nombre d'entrées pour le circuit PAL ne provoque pas la ET pour doubler la taille de chaque entrée supplémentaire comme pour le circuit PROM.

Lorsque les PAL ont été introduites pour la première fois, elles sont devenues l'outil de travail programmable de l'industrie pour les petits modèles en raison de leur coût plus bas, plus grande vitesse, et la facilité d'utilisation. Quand plusieurs PAL ou GAL sont inclus sur la même puce ou plusieurs PLA sont inclus sur la même puce, l'industrie se réfère à ces dispositifs comme CPLDs (dispositifs logiques programmables complexes).

Aujourd'hui, les appareils programmables sont des CPL (pour les modèles de petite et moyenne taille) et APP (pour les dessins de taille moyenne à très grande).

Dans le circuit PAL de la figure II.2, il n'y a pas de partage d'expression produit-terme parce que chaque La sortie de Porte AND ne peut pas être utilisée par plus d'une entrée de Porte OR. C'est le seul inconvénient du PAL par rapport au PLA qui a le partage d'expression produit-terme. [2]

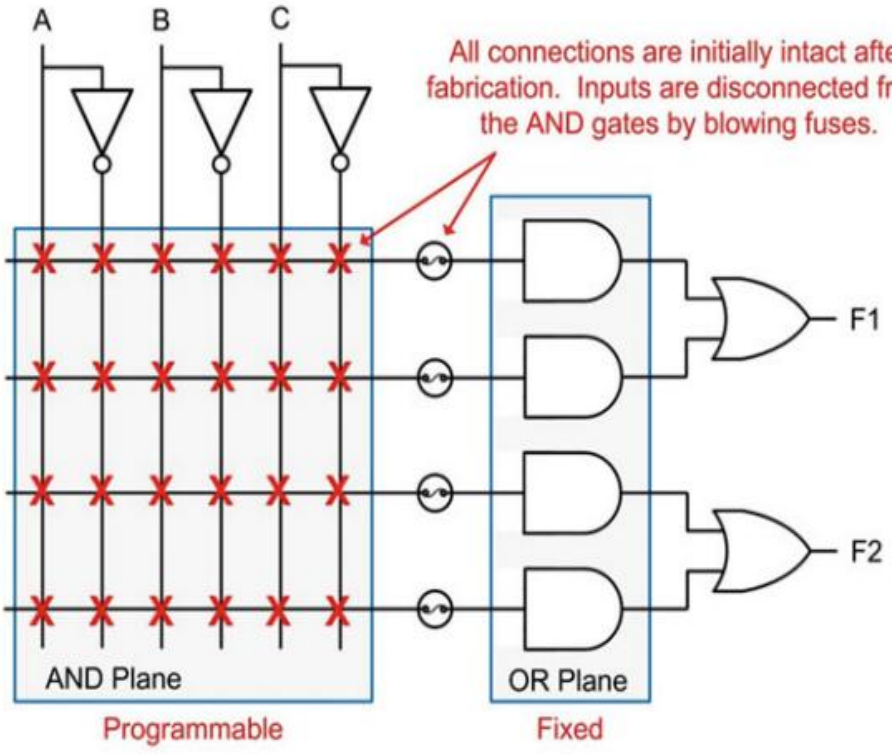


Figure II.25: Architecture PAL (Programmable Array Logic) [1]

II.2.3 Logique de réseau générique (GAL: Generic Array Logic)

À mesure que la popularité du PAL augmentait, des fonctionnalités supplémentaires ont été mises en œuvre pour soutenir des conceptions plus sophistiquées. L'une des améliorations les plus importantes a été l'ajout d'une logique de sortie macro-cellule (OLMC : Ouput Logic Macro-Cell). Une OLMC a fourni un D-Flip-Flop et un multiplexeur sélectionnable de sorte que la sortie du circuit de la SOP du PAL pourrait être utilisée soit comme la sortie du système ou l'entrée à un D-Flip-Flop. Cela a permis la mise en œuvre de la logique séquentielle et des machines à états séquentiels. L'OLMC pourrait également être utilisée pour acheminer la broche E/S dans le PAL pour augmenter le nombre d'entrées possibles dans les expressions SOP. Enfin, l'OLMC a fourni un multiplexeur pour permettre la rétroaction de la sortie PAL ou de la sortie D-Flip Flop. Cette architecture a été nommée logique de tableau générique (GAL :Generic Array Logic) pour distinguer ses caractéristiques d'un standard PLA. La figure II.3 montre l'architecture d'un GAL composé d'un PLA et d'une OLMC. [1]

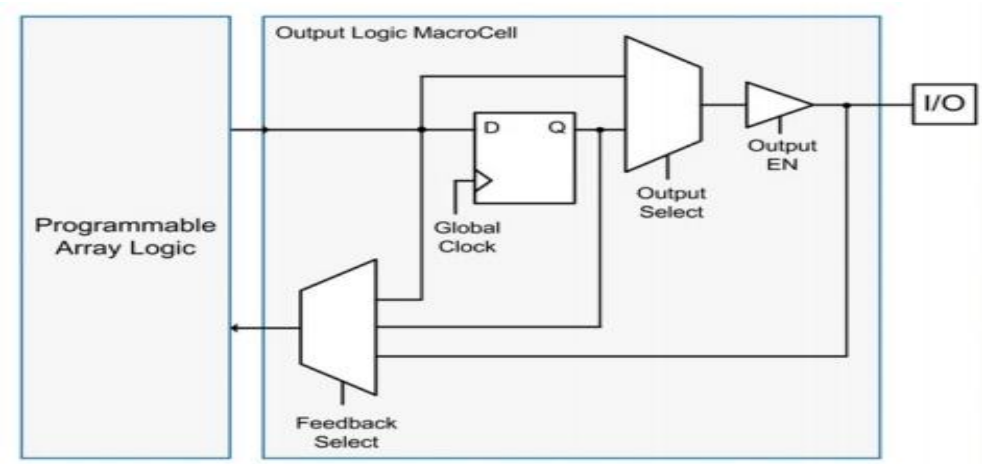


Figure II.3: Architecture logique générique (GAL)

II.3 Dispositifs logiques programmables complexes (CPLDs)

À mesure que la demande d'appareils programmables de plus grande taille augmentait, l'architecture du PAL n'était pas en mesure de prendre efficacement l'expansion, dû à un certain nombre de raisons , entre autres, la quantité de circuits nécessaires sur la puce a augmenté géométriquement en raison de la nécessité d'une connexion à chaque ET en plus de la zone associée aux OLMC supplémentaires. Cela a mené à une nouvelle architecture que l'interconnexion sur puce a été partitionnée sur plusieurs PALs sur une seule puce. Ce partitionnement signifie que toutes les entrées de l'appareil ne peuvent pas être utilisées par chaque PAL, de sorte que la complexité de la conception augmente; cependant, les ressources programmables supplémentaires l'emportaient sur

cet inconvénient, et cette architecture était largement adoptée. Cette nouvelle architecture a été appelé un dispositif logique programmable complexe (CPLD).

Le terme dispositif logique programmable simple (SPLD) a été créé pour décrire tous les PLD précédents architectures (PLA, PAL, GAL). La figure II.4 montre l'architecture de la CPLD.[1]

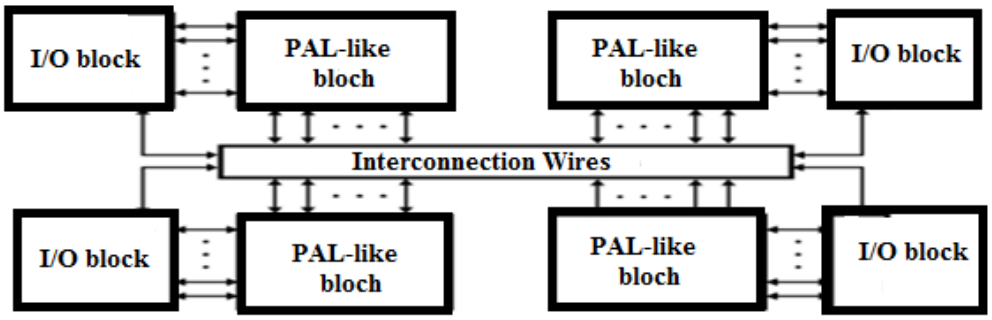


Figure II.4: Architecture complexe PLD (CPLD)[4]

II.4 FPGAs (Field Programmable Gate Arrays)

Un FPGA se compose d'un tableau de blocs logiques programmables (ou d'éléments logiques) et d'un réseau d'interconnexions programmables qui peuvent être utilisée pour connecter n'importe quel élément logique à n'importe quel autre élément logique.[3]

Chaque bloc logique contenait des circuits pour mettre en œuvre des circuits logiques combinatoires arbitraires en plus d'un D-Flip-Flop et un multiplexeur pour l'orientation des signaux. Cette architecture a effectivement mis en œuvre une OLMC dans chaque bloc, fournissant ainsi une flexibilité ultime et fournissant d'importantes ressources pour la logique séquentielle. Aujourd'hui, FPGAs sont les dispositifs logiques programmables les plus couramment utilisés avec Altera Inc. et Xilinx Inc. étant les deux plus grands fabricants. La figure II.5 montre l'architecture générique d'un FPGA.[1]

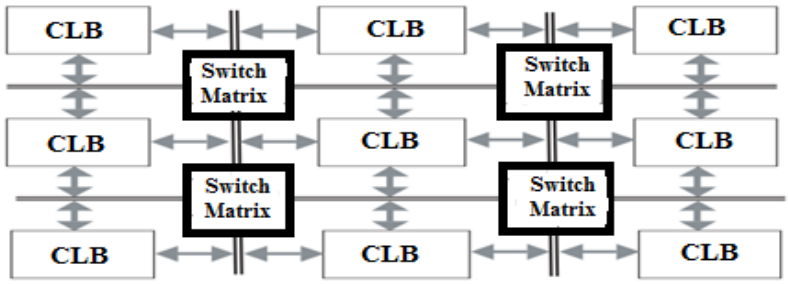


Figure II.5: Architecture FPGA (Field-programmable gate array) [5]

II.4.1 Différent domaines d'application des FPGAs

Les applications des circuits FPGAs sont utilisées dans les domaines :

- ✓ Informatique : Périphériques spécialisés.
- ✓ Machinerie industrielle : Contrôleur pour machines.
- ✓ Télécommunications : Traitement d'images, Filtrage.
- ✓ Instrumentation : Équipement médical, Prototypage.
- ✓ Transport : Contrôle d'avions et métros.
- ✓ Aérospatiale: Satellites, Radar, la détection ou la surveillance……etc

II.4.2 Les cinq principaux atouts de la technologie FPGA

On peut citer les atouts suivants :

- ❖ Performances.
- ❖ Coût.
- ❖ Fiabilité.
- ❖ Temps de mise sur le marché.
- ❖ Maintenance à long terme.

II.4.3 Les blocs logiques CLBs (Configurable Logic Blocs)

La structure de reconstruction primaire dans le FPGA est le bloc logique configurable (CLB : Configurable Logic Block) ou (LE : Logic Element). Xilinx Inc. utilise le terme CLB, tandis qu'Altera utilise LE. La logique combinatoire est implémentée en utilisant un circuit appelé table de vérité (LUT : Look-Up Table), qui peut être implémenté à n'importe quelle table de vérité arbitraire. Les détails d'une LUT sont donnés dans la section suivante. Le CLB/LE contient également un D-Flip-Flop pour la logique séquentielle. Un multiplexeur de direction de signal est utilisé pour sélectionner si la sortie du CLB/LE provient de la LUT ou de la D-Flip-Flop. Le LUT peut être utilisé pour diriger une expression logique combinatoire dans l'entrée D de la D-Flip-Flop, créant ainsi une topologie hautement efficace pour les machines d'état finies. Un réseau de routage global est utilisé pour fournir des signaux communs au CLB/LE tels que l'horloge, la réinitialisation et l'activation. Ce réseau de routage global peut fournir ces signaux communs à l'ensemble du FPGA ou des groupes locaux de CLB/LEs. La figure II.6 montre la topologie d'un CLB/LE simple.[1]

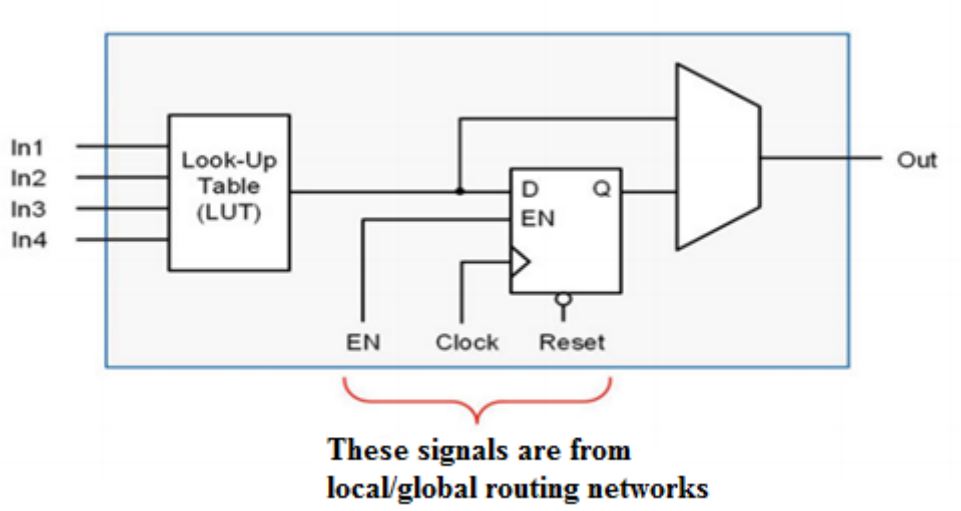


Figure II.6:Bloc logique configurable simple FPGA (ou élément logique) [1]

II.4.4 Les cellules d'entrées/sorties IOBs (Input Output Blocs)

Les FPGAs contiennent également des blocs d'entrée/sortie (**IOBs :** Input/Output Blocks) qui fournissent des fonctionnalités programmables pour l'interfaçage avec les circuits externes. Les IOBs contiennent à la fois des circuits de conducteur et de récepteur de sorte qu'ils peuvent être programmés pour être soit des entrées ou des sorties. Les bascules D sont incluses dans les circuits d'entrée et de sortie pour prendre en charge la logique synchrone. La figure II.7 montre l'architecture d'un IOB dans le circuit FPGA.[1]

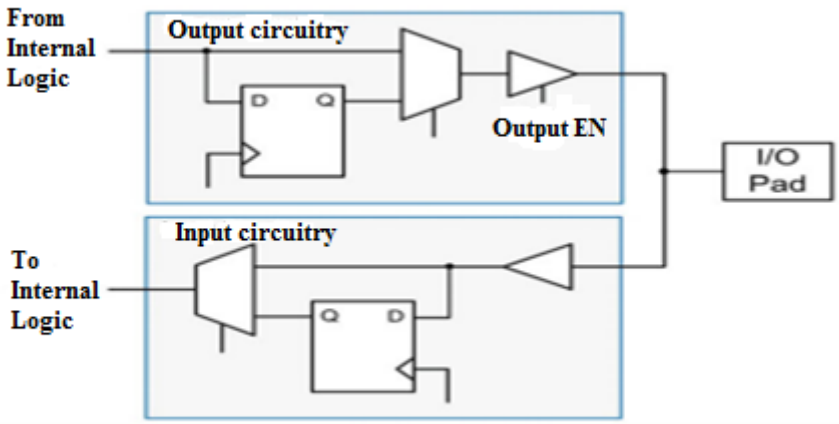


Figure II.7:Bloc entrée/sortie FPGA (IOB)[1]

II.4.5 Technologies de programmation des FPGA

Il existe trois types d' FPGAs reprogrammables suivant la technologie de

Mémorisation pour répondre aux différentes applications :

- ANTIFUSIBLE (la plus ancienne, configurable une seule fois).
- FLASH (non-volatile).
- SRAM (volatile, la plus utilisée, représente plus de 80 % du marché).

II.4.6 Les Avantages

- Technologie « facile » à maîtriser.
- Reprogrammable.
- Temps de développement réduit.
- Idéal pour le prototypage (rapide).
- Coût peu élevé.
- Parallélisme de traitement.
- Flexibilité et la possibilité de réduire
- Fortement les délais de développement et commercialisation.
- La reconfiguration, parfois en temps réel.

II.5 Package (Trousse) DE1

Le pack DE1 contient tous les composants nécessaires pour utiliser la carte DE1 en conjonction avec un ordinateur exécutant le logiciel Microsoft Windows**. [10]**

II.5.1 Contenu du colis

Figure II.8: Contenu du paquet DE1

- Le forfait DE1 comprend :

- Carte DE1
- Câble USB pour la programmation et le contrôle FPGA
- CD-ROM contenant la documentation DE1 et les documents à l'appui, y compris le manuel de l'utilisateur, l'utilitaire du panneau de commande, les conceptions de référence et les démonstrations, les fiches techniques des appareils, les tutoriels et un ensemble d'exercices de laboratoire.
- CD-ROM contenant le logiciel Altera's Quartus® II Web Edition et le processeur embarqué Nios® II
- Sac de six couvercles en caoutchouc (silicone) pour les supports de carte DE1. Le sac contient également des goupilles d'extension, qui peuvent être utilisées pour faciliter l'exploration avec l'équipement de test des en-têtes d'extension E/S de la carte
- Couvercle en plastique transparent pour la carte
- Alimentation murale de 7,5 V c.c. [10]

II.5.2 Carte DE1

Pour assembler les supports inclus pour la carte DE1 :

- Assembler un capot en caoutchouc (silicone), comme indiqué sur la Figure II.9 , pour chacun des six supports en cuivre de la carte DE1
- Le capot en plastique transparent offre une protection supplémentaire et est monté sur le dessus de la carte à l'aide de supports et de vis supplémentaires [10]

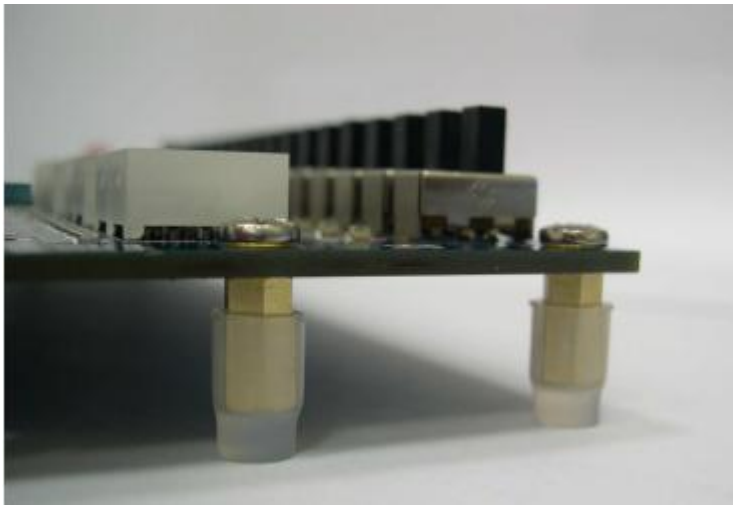


Figure II.9: Pieds de la carte DE1

II.5.3 Carte Altera DE1

Cette partie présente les caractéristiques et les caractéristiques de conception de la carte DE1 [10]

II.5.3.1 Disposition et composants

La figure II.10 montre une photographie de la carte DE1. Il décrit la disposition de la carte et indique l'emplacement des connecteurs et des composants clés [10]

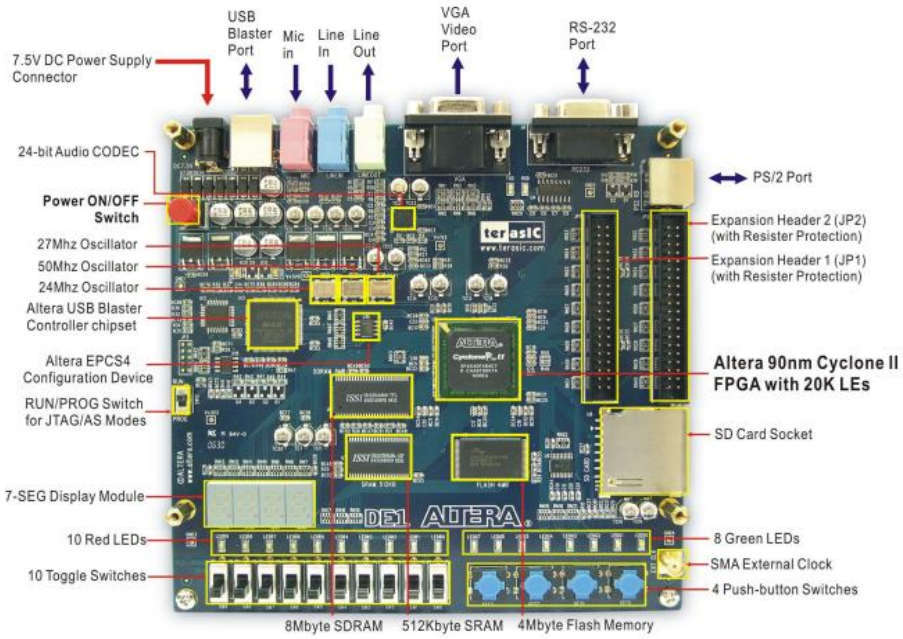


Figure II.10: Carte DE1

La carte DE1 dispose de nombreuses fonctionnalités qui permettent à l'utilisateur de mettre en œuvre un large éventail de circuits conçus, des circuits simples aux différents projets multimédias.

Le matériel suivant est fourni sur la carte DE1 :

- ✓ Dispositif FPGA Altera Cyclone® II 2C20
- ✓ Dispositif de configuration série Altera – EPCS4
- ✓ Blaster USB (embarqué) pour la programmation et le contrôle de l'API utilisateur ; les modes de programmation JTAG et Active Serial (AS) sont pris en charge
- ✓ SRAM de 512 Ko
- ✓ SDRAM 8 Mo
- ✓ Mémoire flash de 4 Mo
- ✓ Prise pour carte SD
- ✓ 4 boutons-poussoirs
- ✓ 10 interrupteurs à bascule
- ✓ 10 LED rouges pour l'utilisateur
- ✓ 8 LED vertes pour l'utilisateur

- ✓ oscillateur de 50 MHz, oscillateur de 27 MHz et oscillateur de 24 MHz pour les sources d'horloge
- ✓ CODEC audio de qualité CD 24 bits avec prises d'entrée, de sortie et de microphone
- ✓ CNA VGA (réseau de résistance 4 bits) avec connecteur VGA-out
- ✓ Émetteur-récepteur RS-232 et connecteur à 9 broches
- ✓ Connecteur souris/clavier PS/2
- ✓ Deux têtes d'extension à 40 broches avec protection contre les résistances
- ✓ Alimenté par un adaptateur CC de 7,5 V ou un câble USB [10]

II.5.3.2 Schéma fonctionnel de la carte DE1

La figure II.11 présente le schéma fonctionnel de la carte DE1. Afin d'offrir une flexibilité maximale à l'utilisateur, toutes les connexions sont effectuées via le dispositif Cyclone II FPGA. Ainsi, l'utilisateur peut configurer le FPGA pour implémenter n'importe quelle conception de système. [10]

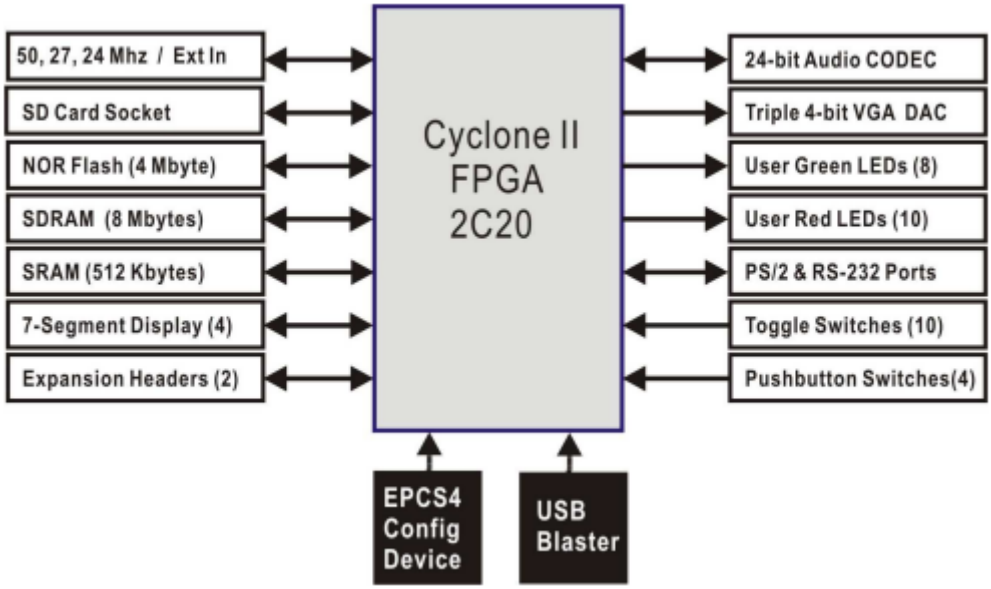


Figure II.11: Schéma fonctionnel de la carte DE1 [10]

II.5.3.3 Mise sous tension de la carte DE1

La carte DE1 est livrée avec un flux de bits de configuration pré-chargé pour démontrer certaines caractéristiques de la carte. Ce flux de bits permet également aux utilisateurs de voir rapidement si la carte fonctionne correctement. Pour mettre la carte sous tension, procédez comme suit : *[10]*

- Connecter le câble USB fourni de l'ordinateur hôte au connecteur USB Blaster sur la carte DE1. Pour la communication entre l'hôte et la carte DE1, il est nécessaire d'installer le logiciel du pilote Altera USB Blaster. Si ce pilote n'est pas déjà installé sur l'ordinateur hôte, il peut être installé comme expliqué dans le tutoriel Mise en route avec la carte DE1 d'Altera. Ce tutoriel est disponible sur le CD-ROM du système DE1 et sur les pages Web Altera DE1. *[10]*
- Connecter l'adaptateur 7,5 V à la carte DE1.
- Connecter un moniteur VGA au port VGA sur la carte DE1.
- Connectez votre casque au port audio Line-out de la carte DE1.
- Tourner le commutateur RUN/PROG sur le bord gauche de la carte DE1 en position RUN ; la position PROG est utilisée uniquement pour la programmation du mode AS.
- Mettre le système sous tension en appuyant sur l'interrupteur ON/OFF de la carte DE1.

À ce stade, vous devriez observer ce qui suit : *[10]*

- ✓ Toutes les LED de l'utilisateur clignotent.
- ✓ Tous les affichages à 7 segments parcourent les chiffres 0 à F.
- ✓ Le moniteur VGA affiche l'image illustrée dans les illustrations fig II.12 et fig II.13, conformément à SW0.
- ✓ Régler le commutateur à bascule SW9 sur la position DOWN ; un bruit de 1 kHz devrait retentir.
- ✓ Réglez le commutateur à bascule SW9 sur la position UP et connectez la sortie d'un lecteur audio au connecteur Line-in de la carte DE1 ; sur votre casque, vous devriez entendre la musique jouée par le lecteur audio (MP3, PC, iPod ou similaire).
- ✓ Vous pouvez également connecter un microphone au connecteur Microphone-in de la carte DE1 ; votre voix sera mélangée à la musique diffusée par le lecteur audio.

Figure II.12: Modèle de sortie VGA par défaut lorsque SW0 est réglé sur la position DOWN

Figure II.13: Modèle de sortie VGA par défaut lorsque SW0 est en position UP

II.6 Quartus II

Cette partie a pour but de vous initier à l'utilisation du logiciel Quartus II de la société Altéra ; les informations que vous trouverez dans ce sujet vous permettront de démarrer dans la création d'un projet. Elles ne constituent en rien une documentation complète et nous vous conseillons de consulter l'aide en ligne : https://docplayer.fr/10362460-Manuel-d-utilisation-de-quartus-ii.html, ou de parcourir le site Altéra pour une plus ample connaissance des outils logiciels.

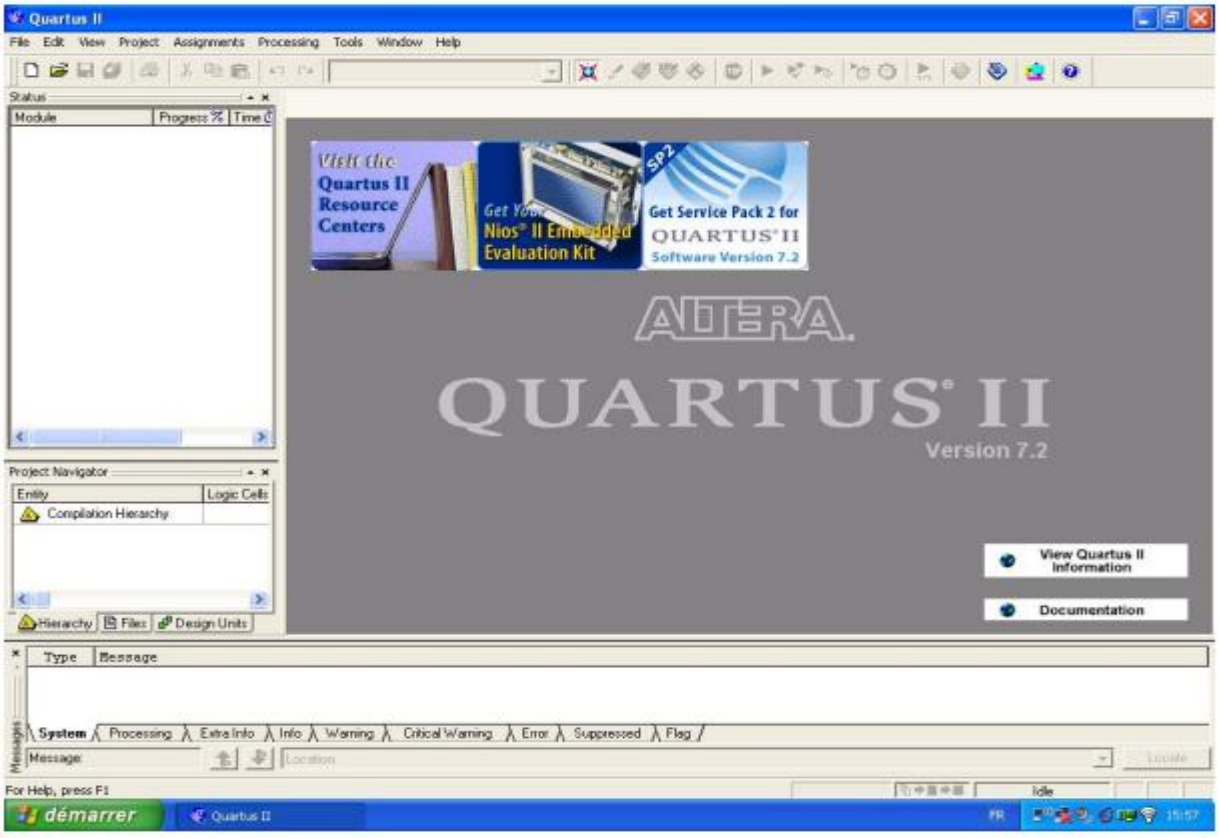


Figure II.14: La Page d'accueille de Quartus II

Quartus est un logiciel proposé par la société Altéra, permettant la gestion complète d'un flot de conception CPLD ou **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.

II.7 Conclusion

Nous avons présenté dans ce chapitre les mémoires et l'évolution des circuits combinatoires et séquentiels afin d'arriver à la définition des circuits FPGAs. La programmation de ces circuits nécessite un logiciel d'adaptation. Pour notre carte DE1, le logiciel approprié est le Quartus , ce dernier est utilisé pour montrer le démarrage de l'utilisation et la programmation de la carte FPGA ,DE1 par ALTERA. Le langage de programmation utilisé VHDL sera développé dans le chapitre suivant.

Chapitre III：Langage de Programmation des Circuits FPGA

III.1 Introduction

L'abréviation VHDL signifie Very Hardware Description Language (VHSIC : Very High Speed Integrated Circuit). Ce langage a été écrit dans les années 70 pour réaliser la simulation de circuits électroniques. On l'a ensuite étendu en lui rajoutant des extensions pour permettre la conception (synthèse) de circuits logiques programmables (P.L.D. Programmable Logic Device).[7]

Les sociétés de développement et les ingénieurs ont voulu s'affranchir des contraintes technologiques des circuits. Ils ont donc créé des langages dits de haut niveau à savoir VHDL et VERILOG. Ces deux langages font abstraction des contraintes technologies des circuits PLDs.

Il faut avoir à l'esprit que ces langages dits de haut niveau permettent de matérialiser les structures électroniques d'un circuit. En effet les instructions écrites dans ces langages se traduisent par une configuration logique de portes et de bascules qui est intégrée à l'intérieur des circuits PLDs. C'est pour cela qu'on préfère parler de description VHDL ou VERILOG que de langage.

Dans notre travail, on s'intéressera seulement à VHDL et aux fonctionnalités de base de celui-ci lors des phases de conception ou synthèse (c'est à dire à la conception de PLDs).

III.2 Le HDL (Hardware Description Language)

Est une instance d'une classe de langage informatique ayant pour but la description formelle d'un système électronique.

Il peut généralement :

- Décrire le fonctionnement du circuit,
- Décrire sa structure,
- Et servir à vérifier sa fonctionnalité par simulation ou preuve formelle.

Ils avaient pour but de modéliser, donc de simuler, mais aussi de concevoir, des outils informatiques permettant de traduire automatiquement une description textuelle en dessins de transistors. [9]

À la différence d'un langage de programmations logicielles, la syntaxe et la sémantique d'un HDL incluent des notations explicites pour exprimer le temps et la concurrence qui sont les attributs principaux du matériel.

Les classes de langages dont la seule caractéristique est de décrire un circuit par une hiérarchie de blocs interconnectés est appelée une Netlist. [10]

Un synthétiseur logique permet de transformer un circuit décrit dans un langage de description de matériel en une Netlist. [9]

III.3 Les avantages du langage VHDL

- La portabilité.
- La conception de haut niveau.
- La possibilité de décrire des systèmes très complexes en quelques lignes de code.
- De plus, le VHDL :
  - Peut-être simulé,
  - Peut être traduit en schéma de portes logiques. [6]

III.4 Les unités de compilation VHDL

L'analyse d'un modèle VHDL peut s'effectuer sur des parties du code ou "unités de compilation". Il existe 5 types d'unités de compilation :

- Entité (vue externe)
- Architecture (vue interne)
- Configuration (couple: entité-architecture)
- Paquetage (déclarations globales, …)
- Corps du paquetage (sous-programmes, …)

En VHDL, une structure logique est décrite à l'aide d'une entité et d'une architecture de la façon suivante :

Figure: III.1 Structure Logique

III.4.1 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 de signaux électroniques,

Des fonctions et sous programmes permettant de réaliser des opérations arithmétiques et logiques,... [7]

```
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
```

III.4.2 Déclaration de l'entité

```
entity Nom de l'entité is
        port  (les signaux d'entrée : in (type des signaux);
               les signaux de sortie: out (type des signaux)
               );
end Nom de l'entité;
```

III.4.2.1 Le TYPE

Le TYPE utilisé pour les signaux d'entrées / sorties est : [7]

- le std_logic pour un signal.
- le std_logic_vector pour un bus composé de plusieurs signaux.

Les valeurs que peuvent prendre un signal de type std_logic sont : [1]

- 'U' : non initialisé
- 'X' : niveau inconnu, forçage fort
- '0' : niveau 0, forçage fort
- '1' : niveau 1, forçage fort
- 'Z' : haute impédance
- 'W' : niveau inconnu, forçage faible
- 'L' : niveau 0, forçage faible
- 'H' : niveau 1, forçage faible
- '-' : quelconque (don't care)

- Type BIT: ce type peut prendre deux valeurs : ‘0’ ou ‘1’. Ces deux valeurs ne sont pourtant pas suffisantes pour satisfaire les besoins de la synthèse logique : par exemple, l'état haute impédance n'est pas envisagé !
- Type BIT_VECTOR: ce type permet de manipuler des grandeurs résultantes de l'association d'éléments de type BIT (i.e. des vecteurs de bits ou mot).
- Types STD_LOGIC et STD_LOGIC_VECTOR: ces types sont inclus dan la bibliothèque IEEE 1164,
- STD_LOGIC_1164: [1]

```
Library IEEE;
Use IEEE.std_logic_1164.all;
```

- Types CHARACTER et STRING : Les variables de type CHARACTER prennent 95 valeurs différentes correspondant à une partie du jeu de caractères ASCII. Le type STRING résulte de l'association de plusieurs éléments de type CHARACTER.
- Type SEVERITY_LEVEL : Ce type est employé avec une fonction spécifique aux programmes de tests et permet de caractériser le niveau de gravité programmable lors d'incidents de simulation. Les variables de ce type peuvent prendre les valeurs suivantes : NOTE, WARNING, ERROR, FAILURE.

III.4.2 .2 Le SENS du signal

Le SENS du signal

- in : pour un signal en entrée.
- out : pour un signal en sortie.
- inout : pour un signal en entrée sortie

III.4.3 Déclaration de l’architecture correspondante à l’entité : [1]

architecture **Nom de l'architecture** of **Nom de l'entité** is

partie déclarative optionnelle : types, constantes, signaux locaux, composants.

begin

Corps de l’architecture.

(Suite d’instructions parallèles : affectations de signaux; processus explicites; blocs; instanciation (i.e. importation dans un schéma) de composants.

end **Nom de l'architecture** ;

III.5 Opérateurs VHDL :

III.5.1 Opérateurs logiques : [1]

Opérateur	Opération
not	négation logique
and	And logique
nand	Nand logique
or	Or logique
nor	Nor logique
xor	Xor logique
xnor	Xnor logique

Tableau III.3: Opérateur Logiques

Décalage à droite :

-- Si A est de type std_logic_vector(7 downto 0)

S1 <= ‘0’ & A(7 downto 1); -- décalage d’un bit à droite

S1 <= "000" & A(7 downto 3); -- décalage de trois bits à droite. [7]

Décalage à gauche :

-- Si A est de type std_logic_vector(7 downto 0)

S1 <= A(6 downto 0) & ‘0’; -- décalage d’un bit à gauche

S1 <= A(4 downto 0) & "000"; -- décalage de trois bits à gauche[7]

III.5.2 Opérateurs arithmétiques : [7]

Opérateur	VHDL
ADDITION	+
SOUSTRACTION	-
MULTIPLICATION	*
DIVISION	/

Tableau III.4: Opérateur Arithmétiques

III.5.3 Opérateurs relationnels :

VHDL contient les opérateurs relationnels suivants. Ces opérateurs comparent deux entrées de la même tapez et retourne le type booléen (c.-à-d., vrai ou faux). [1]

Opérateur	Renvoie true si la comparaison est
=	Egal
/=	Différent
<	moins de
<=	inférieure ou égale
>	supérieure à
>=	supérieure ou égale

Tableau III.5: Opérateurs Relationnels

III.6 Les instructions du mode « concurrent »

III.6.1 Affectation conditionnelle

Cette instruction modifie l'état d'un signal suivant le résultat d'une condition logique entre un ou des signaux, valeurs, constantes.[11]

```
signal_name   <= expression_1 when condition_1 else
                 expression_2 when condition_2 else
                 expression_n; [1]
```

III.6. 2. Affectation sélective

Cette instruction permet d'affecter différentes valeurs à un signal, selon les valeurs prises par un signal dit de sélection. [7]

```
with input_name select
signal_name <= expression_1 when condition_1,
               expression_2 when condition_2,
               expression  n when others; [1]
```

III.7 Les instructions du mode séquentiel

III.7.1 Définition d'un PROCESS[7]

```
[Nom_du_process :] process(Liste_de_sensibilité_nom_des_signaux)
Begin
-- instructions du process
end process [Nom_du_process] ; [1]
```

3.6.2. Les deux principales structures utilisées dans un process :

- **L'assignation conditionnelle :**

```
if condition then
        instructions
[else if condition then instructions]
[else instructions]
end if ; [11]
```

- **L'assignation sélective**

```
Case signal_de_slection is
When valeur_de_sélection => instructions
[When others => instructions]
 End case; [11]
```

III.8 Concepts de programmation conditionnelle

III.8. 1 If/Then déclarations [1]

```
if boolean_condition then sequential_statement
end if;
if boolean_condition then sequential_statement_1
else sequential_statement_2
end if;
if boolean_condition_1 then sequential_statement_1
elsif boolean_condition_2 then sequential_statement_2
elsif boolean_condition_n then sequential_statement_n
end if;
if boolean_condition_1 then sequential_statement_1
elsif boolean_condition_2 then sequential_statement_2
elsif boolean_condition_n then sequential_statement_n
else sequential_statement_n+1
end if;
```

III.8. 2 instructions case [1]

```
case (input_name) is
when choice_1 <= sequential_statement(s);
when choice_2 <= sequential_statement(s);
:
:
when choice_n <= sequential_statement(s);
end case;
```

Lorsque toutes les conditions d'entrée possibles (ou choix) ne sont pas spécifiés, une clause **when others** est utilisée pour fournir des assignations de signal pour toutes les autres conditions d'entrée. Ce qui suit est la syntaxe d'une instruction case qui utilise une clause **when others. [1]**

```
case (input_name) is
when choice_1 <= sequential_statement(s);
when choice_2<=sequential_statement(s);
:
:
when others <= sequential_statement(s);
end case;
```

Plusieurs choix qui correspondent aux mêmes assignations de signal peuvent être délimités dans l'énoncé de cas. Ce qui suit est la syntaxe pour une instruction de cas avec des choix délimités par pipe. **[1]**

```
case (input_name) is
when choice_1 | choice_2 <=sequential_statement(s);
when others <= sequential_statement(s);
end case;
```

Le signal d'entrée d'une instruction **case** doit être un nom de signal unique. Si plusieurs scalaires doivent être utilisés comme expression d'entrée pour une instruction **case**, ils doivent être

concaténés soit en dehors du processus résultant en un nouveau vecteur de signal, soit dans le processus résultant en un nouveau vecteur variable. **[1]**

III.8. 3 Boucles infinies (Infinite Loops) [1]

```
loop
exit when boolean_condition; -- optional exit statement
next when boolean_condition; -- optional next statement
sequential_statement(s);
end loop;
```

III.8. 4 Boucles While [1]

```
while boolean_condition loop
sequential_statement(s);
end loop;
```

III.8. 5 Boucles for [1]

```
for loop_variable in min to max loop
sequential_statement(s);
end loop;
```

Voici la syntaxe d'un VHDL pour une boucle dans laquelle la variable de boucle décrémente de max à min de la plage. **[1]**

```
for loop_variable in max downto min loop
sequential_statement(s);
end loop;
```

III.9 Paquetages : (Packages)

Un paquetage permet de rassembler des déclarations et des sous-programmes, utilisés fréquemment dans une application, dans un module qui peut être compilé à part, et rendu visible par l'application au moyen de la clause use.

- ✓ Les paquetages prédéfinies: Un compilateur VHDL est toujours assorti d'une librairie, décrite par des paquetages, qui offrent à l'utilisateur des outils variés.
- ✓ Les paquetages créés par l'utilisateur : L'utilisateur peut créer ses propres paquetages. Cette possibilité permet d'assurer la cohérence des déclarations dans une application complexe, évite

d'avoir à répéter un grand nombre de fois ces mêmes déclarations et donne la possibilité de créer une librairie de fonctions et procédures adaptée aux besoins des utilisateurs. **[1]**

III.10 Conclusion

Nous avons présenté dans ce chapitre dans les applications des circuits combinatoires et séquentiels, le langage de programmation des circuits FPGA qui est VHDL. Par le biais du logiciel Quartus, on peut brancher la carte DE1 pour notre cas avec le PC et utiliser le langage VHDL afin de concevoir les circuits programmables demandés par l'utilisateur. .

Chapitre IV : Applications des circuits sur FPGA

IV.1 Introduction

Nous allons développer dans cette partie la programmation VHDL des circuits combinatoires et séquentiels. L'utilisation du Quartus est nécessaire pour le branchement entre le PC et la carte DE1 ainsi que l'application du programme approprié sur le circuit FPGA.

IV.2 Implémentation de quelques circuits logiques dans les circuits FPGA

IV.2.1 Porte Inverseur

Le programme de l'inverseur est comme suite :

```
Library ieee ;
Use ieee.std_logic_1164.all;
Entity inverseur is
Port (A: in bit;F: out bit);
End inverseur ;
Architecture inverseur_ARCH of  inverseur is
Begin
F <=not A ;
End inverseur_ARCH;
```

- Ouvrir Quartus II comme n'importe quel logiciel, il présente alors L'interface suivant, comprenant 4 zones ou fenêtres principales :

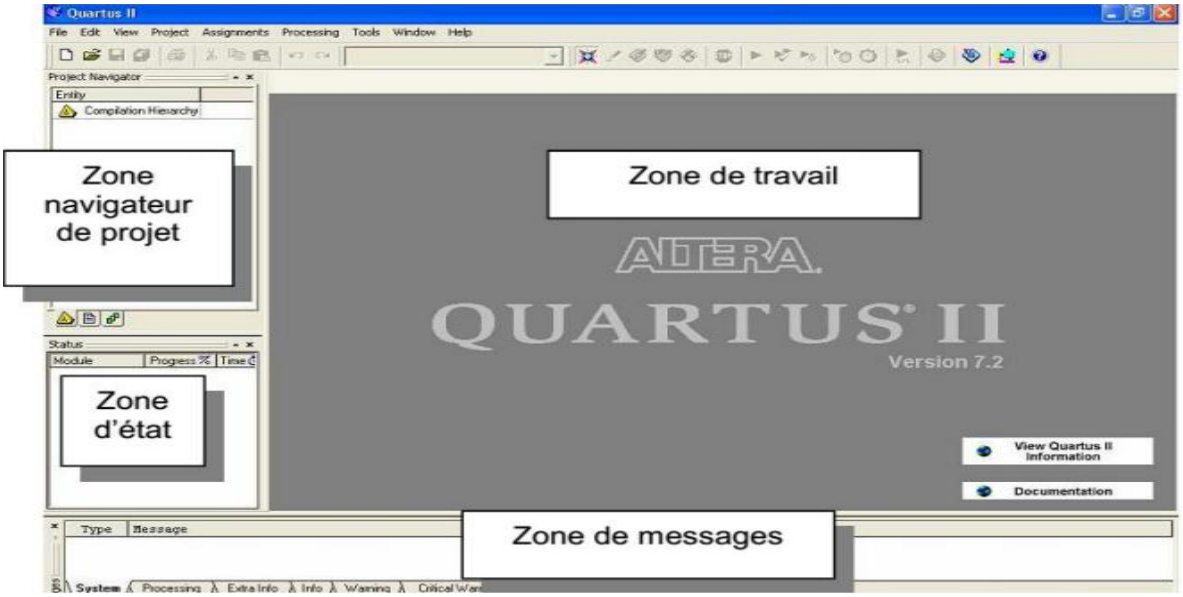


Figure IV.1: La Page d'accueille de Quartus II

1. une zone « navigateur de projet » permettant de gérer les différents fichiers d'un projet ;
2. une zone de travail permettant la synthèse du projet ;
3. une zone d'état (Status) permettant de voir l'avancement de la tâche en cours ;
4. une zone de messages.

Création d'un nouveau projet wizard :

On clique sur **File**, aller sur **New Project Wizard**

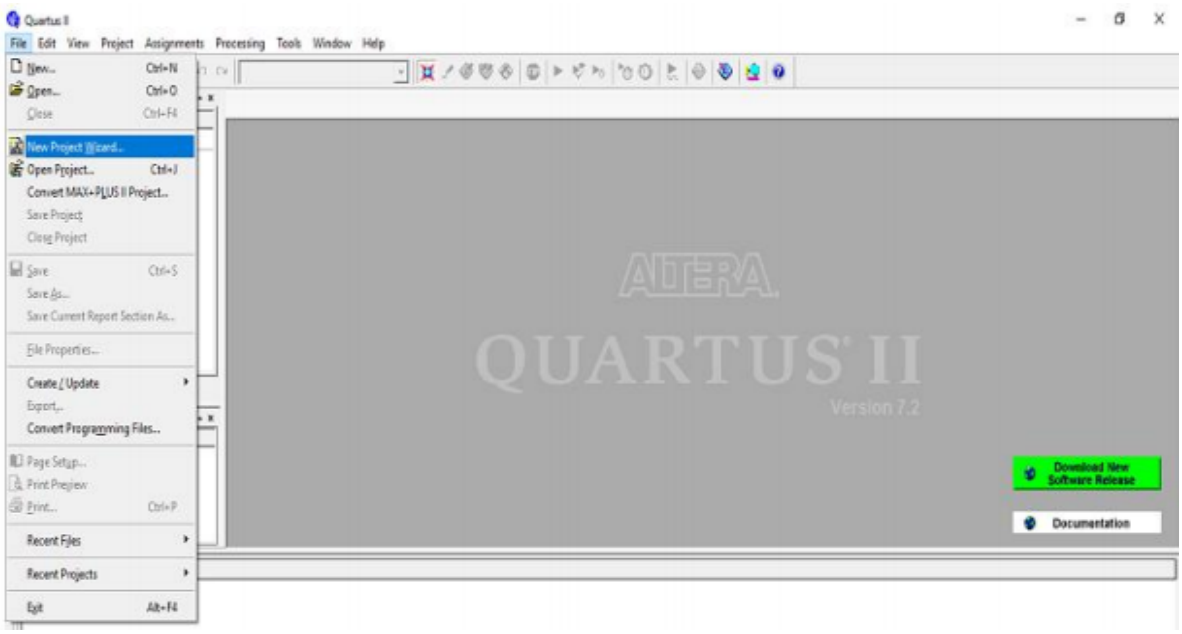


Figure IV.2. **Création d'un nouveau projet wizard**

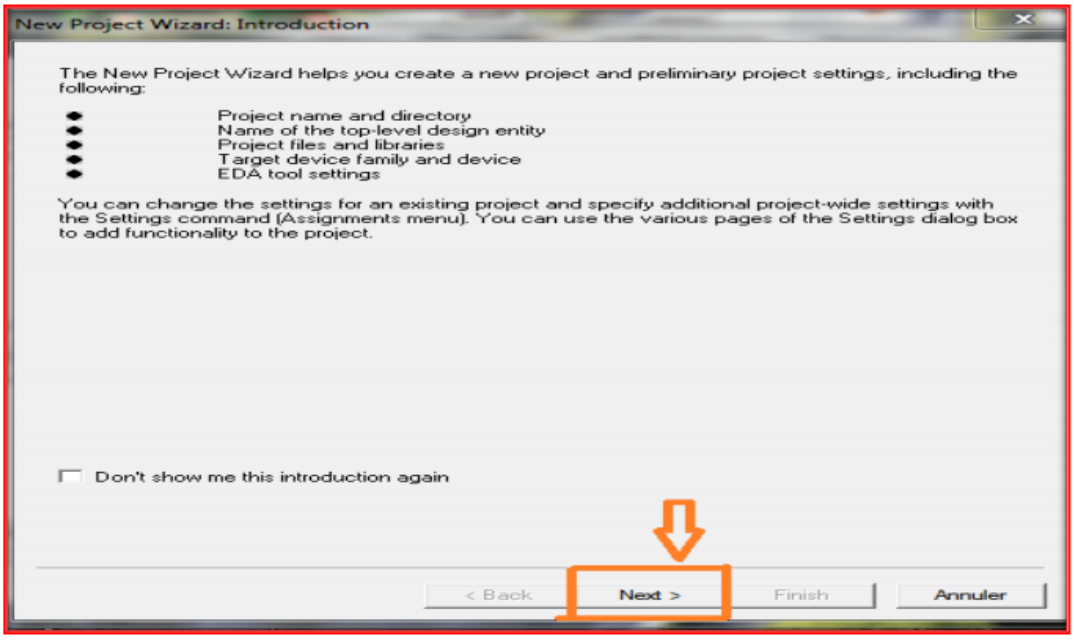


Figure IV.3. **New project Wizard :Introduction**

- On clique sur **Next** :
- Après cette étape on va Choisir l'emplacement du répertoire ou seront stockés tous les fichiers du projet. (1)
- Choisir le nom de votre projet.(2)
- Choisir le nom de l'entité du projet (niveau le plus haut dans le design).(3)

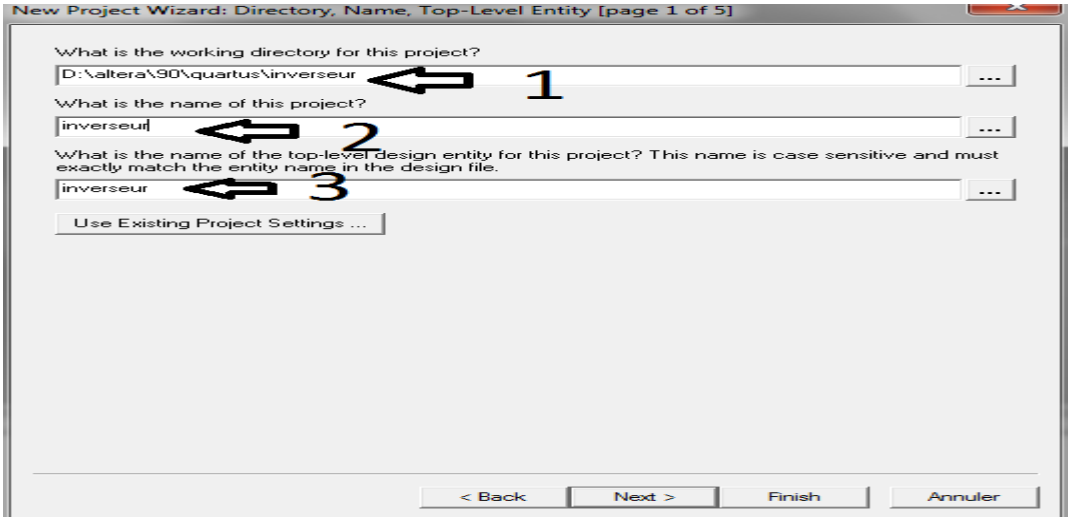


***Figure IV.26 New* project Wizard :Directory**

- On Cliquer sur **Next** puis quand la fenêtre **Add Files** apparaît :
- Choisir la famille ainsi que le circuit cible :

Family : Choisir : **Cyclone II Carte Altéra DE1**.

Avalaible device: sélectionner par exemple **EP2C20F484C7**.

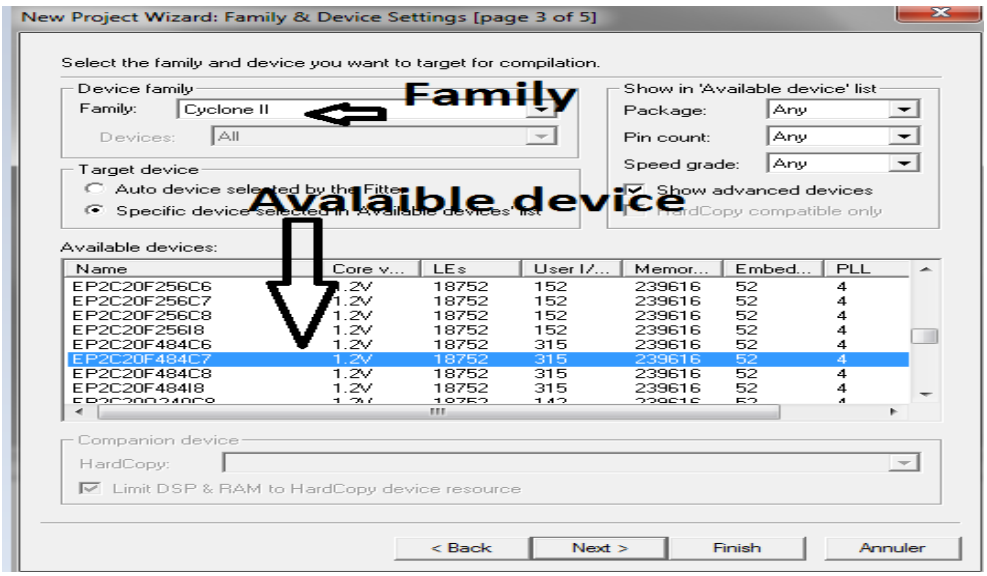


***Figure IV.5.* New project Wizard :Family &Device**

- Apres cela choisir (la famille..) on clique sur **Next** et il va afficher une fenêtre qui contient le résumer des paramètres du projet.
- Pour valider les choix on clique sur **Finish** ou bien faire **Back** pour des modifications.

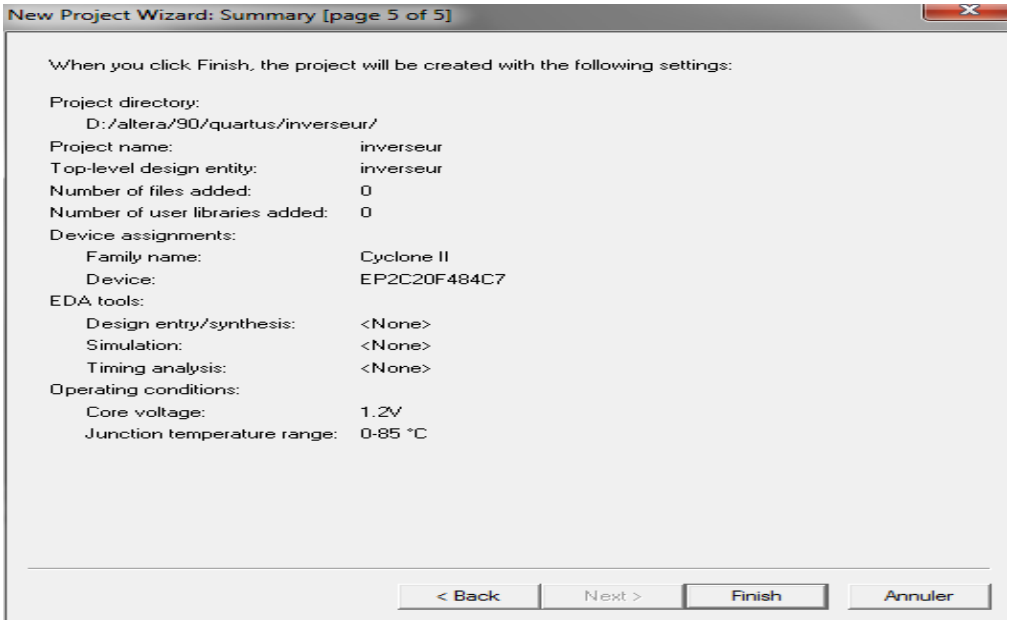


Figure IV.6.New Project Wizard :Summary

- Après cela, on crée un nouveau fichier VHDL après avoir cliqué sur **New** :

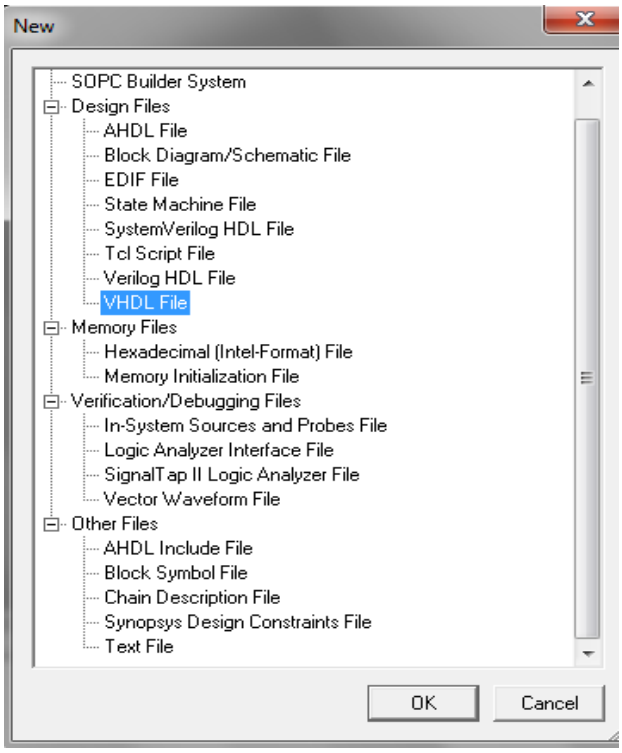


Figure IV.7. **Choisi le type de Fichier VHDL File**

On écrit le programme du porte **inverseur** en langage VHDL et on compile pour détecter les erreurs

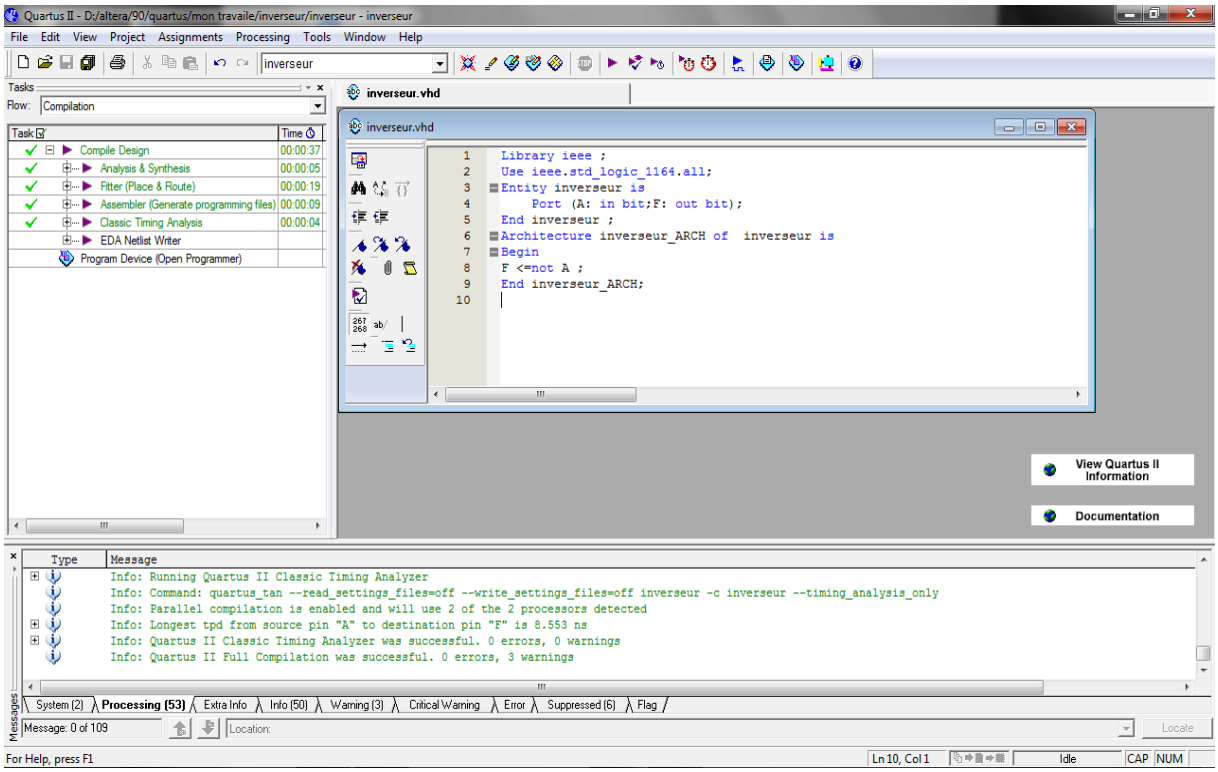


Figure IV.8. **le programme VHDL**

- pour vérifier la bonne syntaxe de la description en cliquant **Processing** Analyse Current File.
- Si il y'a des erreurs, on vérifie dans la zone **Processing** (en bas où s'affichent les messages) la source du problème et on Corrige les éventuelles erreurs.

Durant la compilation, Quartus va réaliser 4 étapes :

- ✓ La transformation des descriptions graphiques et textuelles en un schéma électronique à base de portes et de registres : c'est la synthèse logique.
- ✓ 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 (EP2C20F484C7) : c'est la synthèse physique.
- ✓ L'assemblage consiste à produire les fichiers permettant la programmation du circuit ; Dans notre cas, nous utiliserons toujours le format SOF pour les FPGA.
- ✓ L'analyse temporelle permet d'évaluer les temps de propagation entre les portes et le long des chemins choisis lors du fitting.

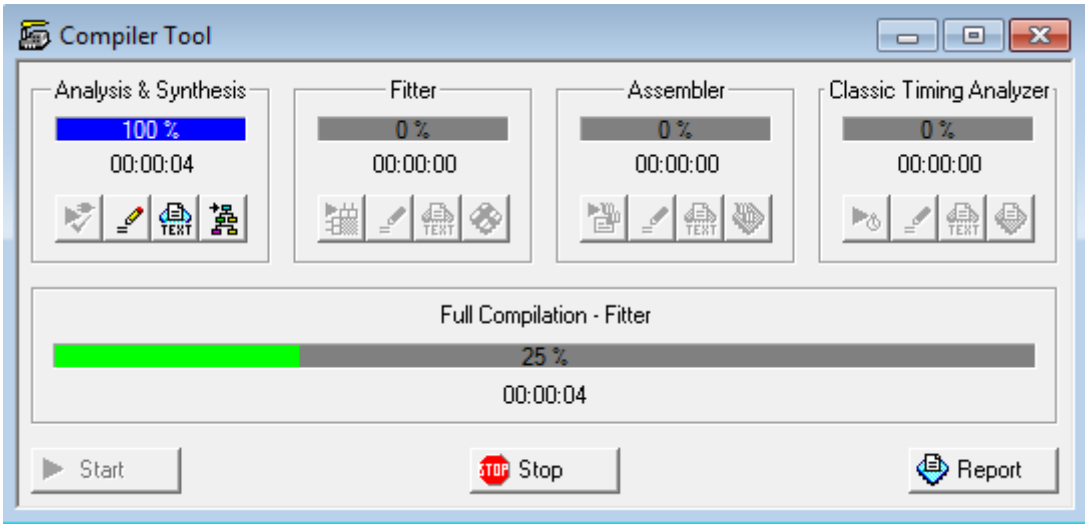


Figure IV.9. **compilation**

- On passe de mode ‘’**timing**’’ au mode ‘’ **Functionnel**’’ en cliquant sur **assignment**>>**timing** >>**simulation setting**

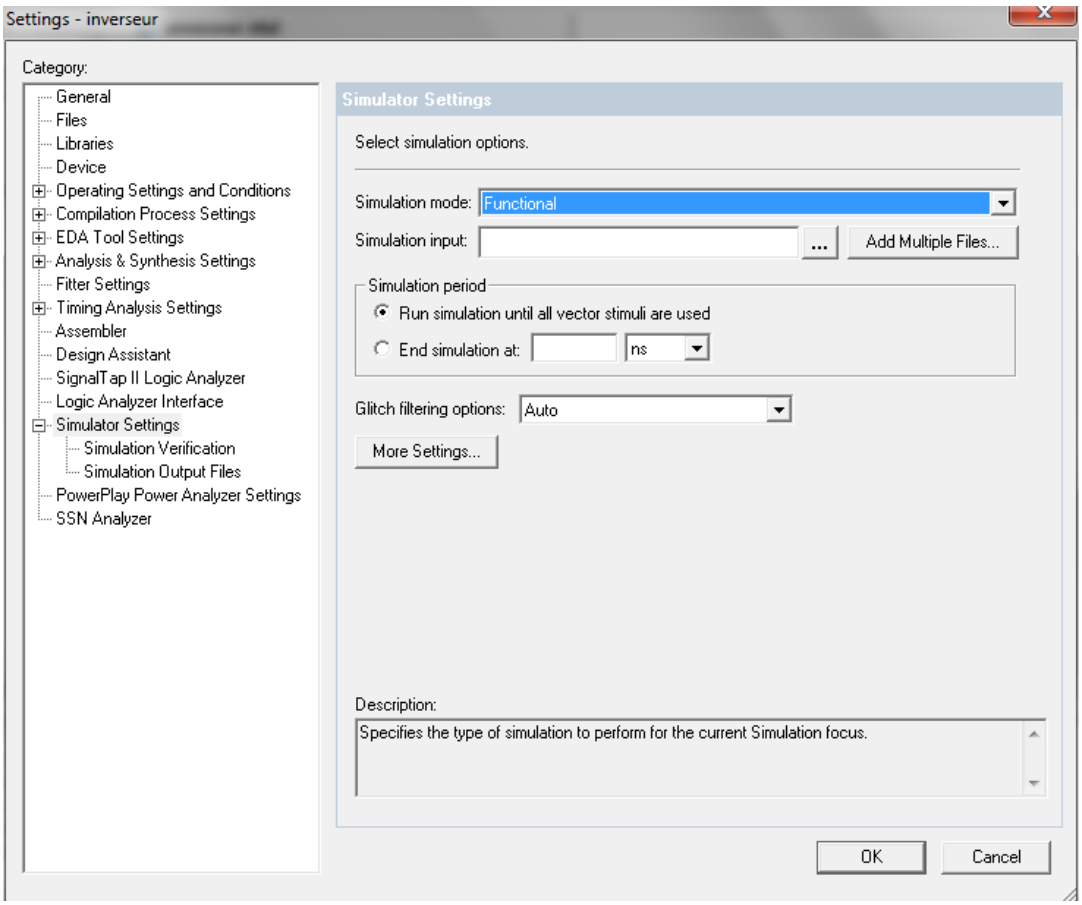


Figure IV.10.Choisi le Mode de Simulation

- ✓ Apres avoir cliqué sur **New**, on se dirige vers <<**vector waveform file**>> pour créer la table de vérité de notre porte et insérer les valeurs d’entrées.

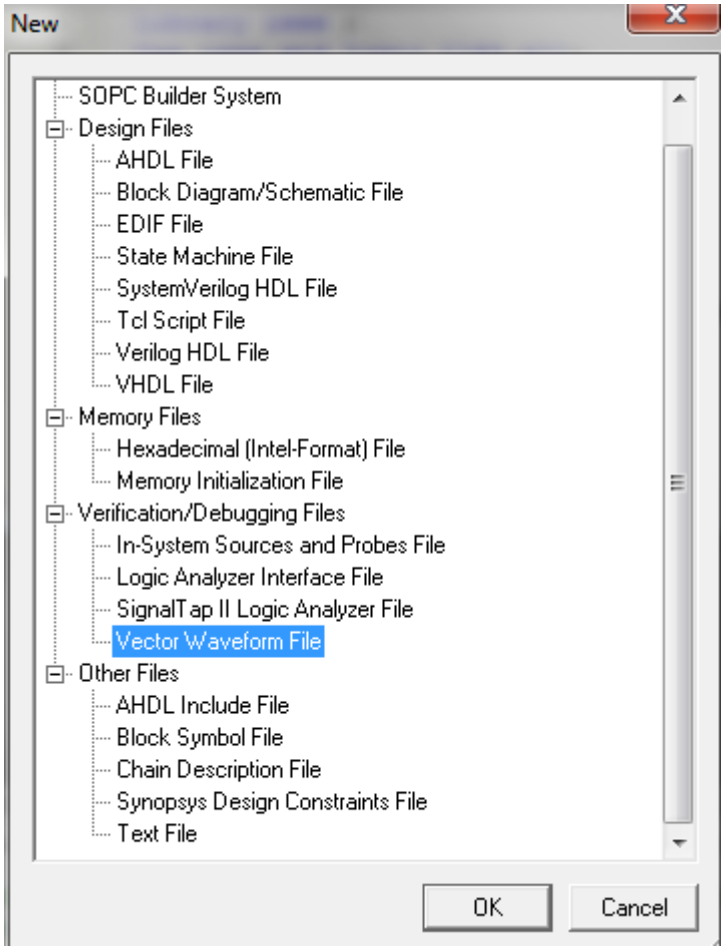


Figure IV.11 Choisi le Type de Vérification

- Pour insérer les valeurs d'entrées, cliquer « **Edit> Insert> Insert Node or Bus**» tout en étant ouvert le fichier de vecteur de **test .vwf** puis clique sur « **Insert Node or Bus** ».
- Ensuite sur <<**Node Finder** >> ce qui permet de lancer le navigateur de signaux

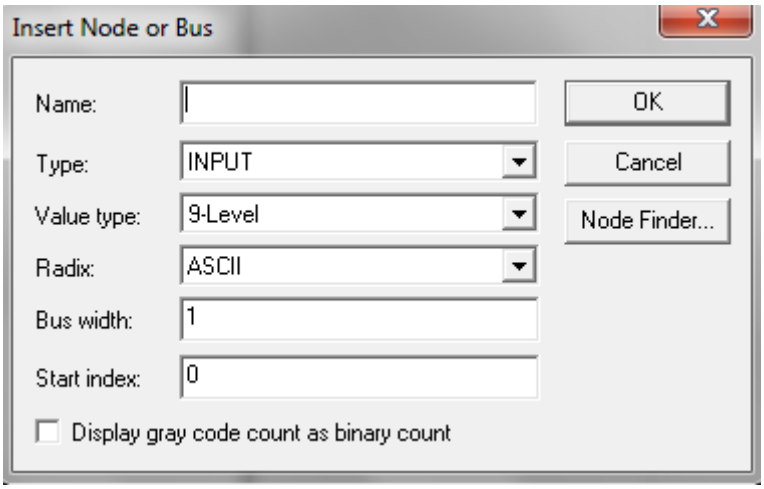


Figure IV.12. Insérer Node or Bus

- ✓ Ensuite, on sélectionne '**List**' pour laisser apparaitre toutes les entrées et sorties, il nous suffit ensuite de les sélectionner.

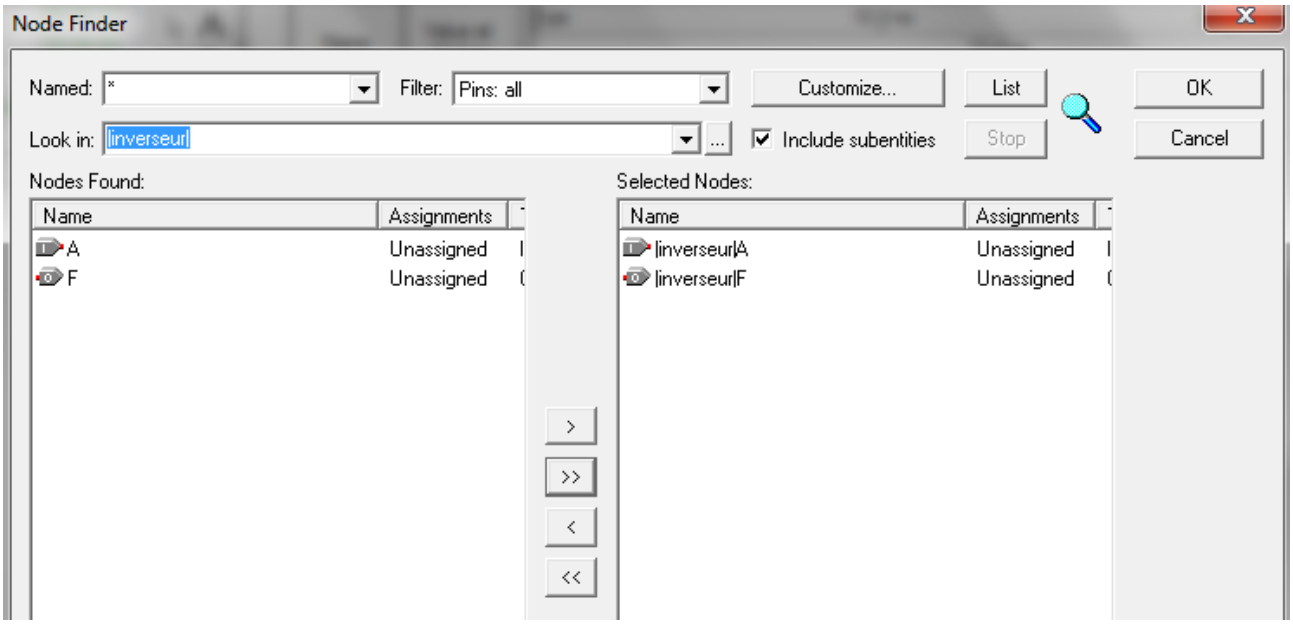


Figure IV.13 Node Finder

- ✓ On adapte A (les entrées) selon la table de vérité de inverseur (0101)

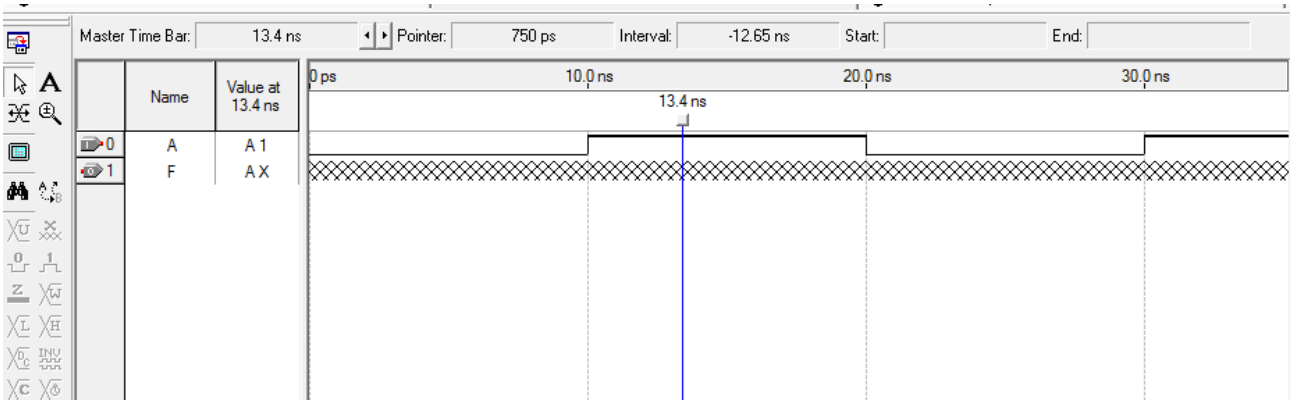


Figure IV.14 Waveforms 1.vwf

- ✓ Après compilation :

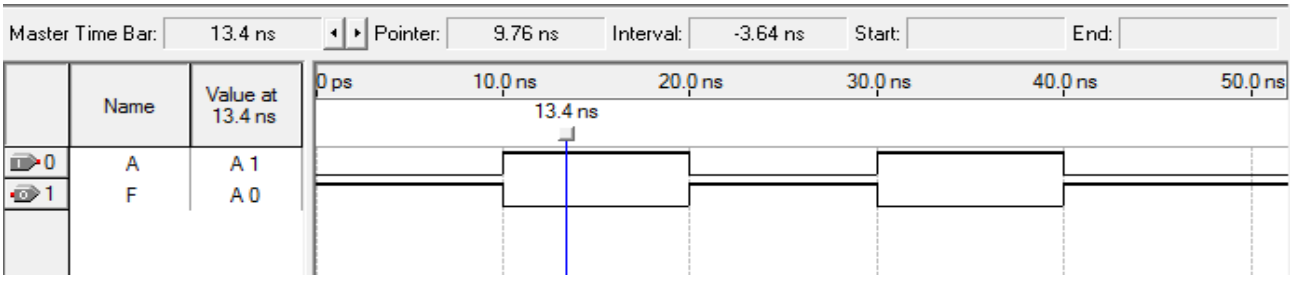


Figure II.15 simulation Waveforms

On peut éditer le temps de retard en cliquant sur **Processing**, et ensuite << **Generate Functional Simulation Netlist** >> et on peut varier la durée du signal en cliquant sur **Edit**, end time.

Cliquer sur **Assignements**, puis aller sur **Pin Planner**, la figure ci-dessous apparait :

- ✓ On s'occupe ensuite de **Pin Planner** en le sélectionnant dans la barre d'outils. On remplit le tableau du bas avec la location de chaque entrée et sortie selon le kit et le tableau de valeur c'est cette étape qui va nous permettre de rattacher le kit à notre programme.

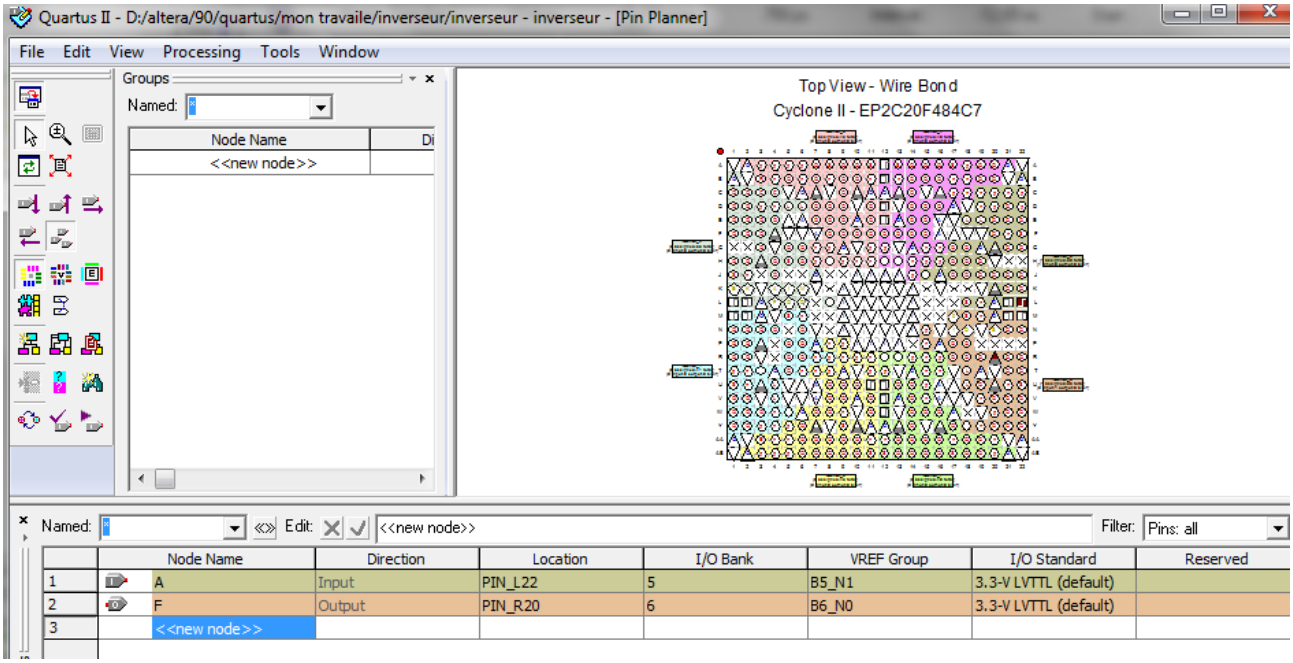


Figure II.16 **.Pin Planner**

Signal Name	FPGA pin no
SW [0]	PIN_L22
SW [1]	PIN_L21
SW [2]	PIN_M22
SW [3]	PIN_V12
SW [4]	PIN_W12
SW [5]	PIN_U12
SW [6]	PIN_U11
SW [7]	PIN_M2
SW [8]	PIN_M1
SW [9]	PIN_L2

Tableau IV.6: Pin Assignments for Slide Switches

Signal Name	FPGA pin no
KEY[0]	PIN_R22
KEY[1]	PIN_R21
KEY[2]	PIN_T22

KEY[3]	PIN_T21

Tableau IV.7: Pin Assignments for Push-buttons

Signal Name	FPGA pin no
CLOCK_27 [0]	PIN_D12
CLOCK _27[1]	PIN_E12
CLOCK_24[2]	PIN_B12
CLOCK_24[3]	PIN_A12

Tableau IV.8: Pin Assignments for CLOCK

Signal Name	FPGA Pin No
LEDR[0]	PIN_R20
LEDR[1]	PIN_R19
LEDR[2]	PIN_U19
LEDR[3]	PIN_Y19
LEDR[4]	PIN_T18
LEDR[5]	PIN_V19
LEDR[6]	PIN_Y18
LEDR[7]	PIN_U18
LEDR[8]	PIN_R18
LEDR[9]	PIN_R17
LEDG[0]	PIN_U22
LEDG[1]	PIN_U21
LEDG[2]	PIN_V22
LEDG[3]	PIN_V21
LEDG[4]	PIN_W22
LEDG[5]	PIN_W21
LEDG[6]	PIN_Y22
LEDG[7]	PIN_Y21

Tableau IV.9: Pin Assignments for LEDs.

- ✓ La dernière étape consiste à programmer le composant, c'est à dire implanter la description dans la cible matérielle (FPGA dans notre cas), il faut pour cela que le projet ait auparavant été compilé.

Cette opération se fait dans notre cas via un des ports USB du Pc connecté au port BLASTER de la carte DE1.

- ✓ Pour programmer le composant, il faut cliquer sur l'icône correspondante.
- ✓ Un écran apparait, avec un fichier d'extension, **cdf** listé il s'agit du fichier de programmation final :

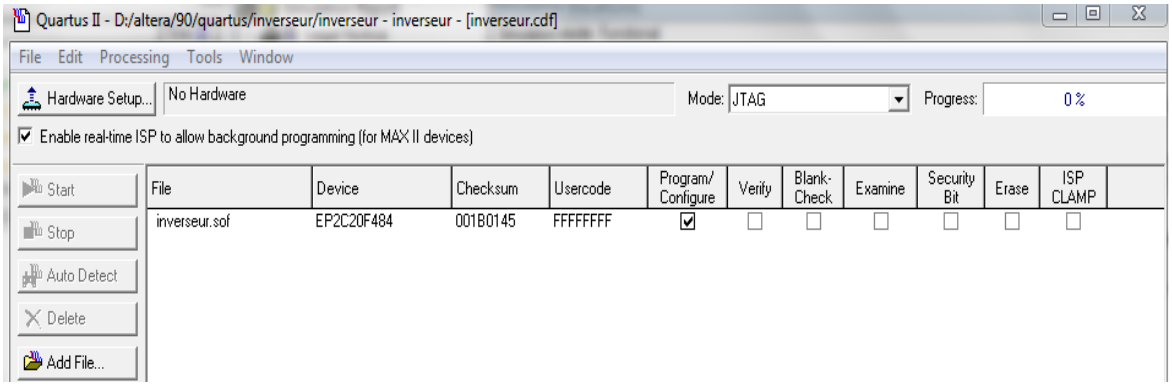


Figure IV.17. **inverseur.cdf**

- ✓ Si aucun matériel de programmation n'est défini (inscription '**NO HARDWARE'**), cliquez alors sur '**HARDWARE SETUP**' et sélectionnez '**USB BLASTER'**.
- ✓ Cocher les cases '' Program /Configure'' et ''Verify'' et lancer la programmation en cliquant sur **START** le fonctionnement est maintenant prêt à être testé sur la platine de test.

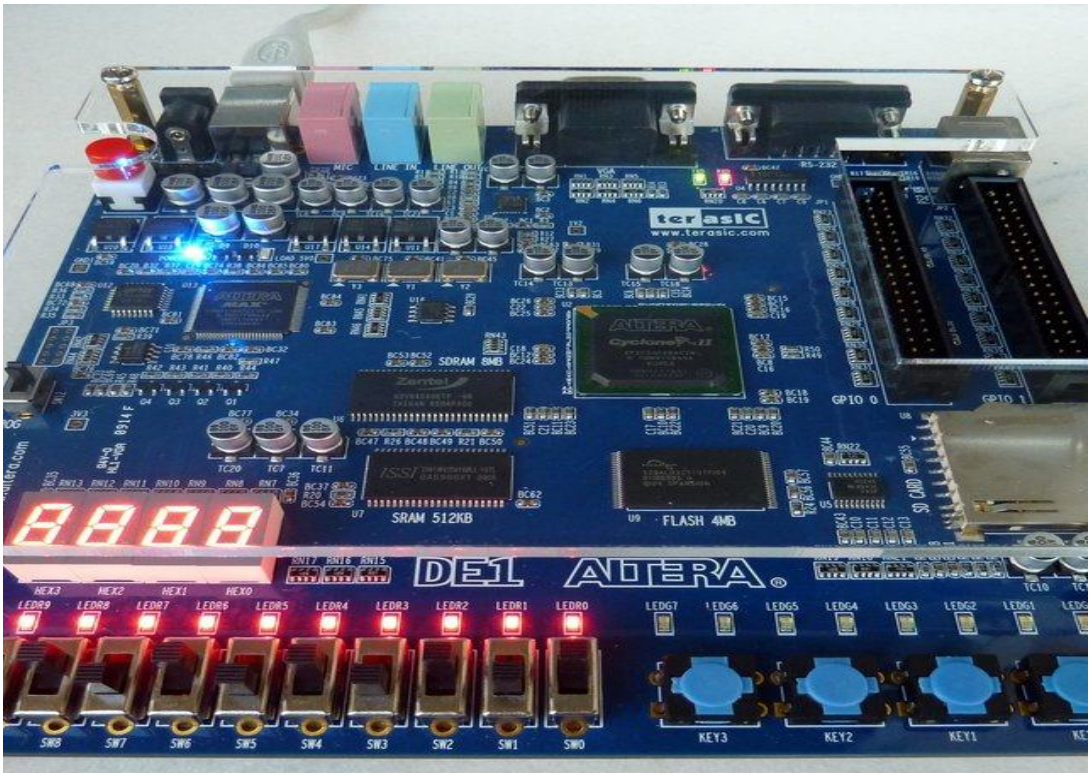


Figure IV.18CARTE FPGA DE1 ALTERA

Figure IV.19 Vue RTL de inverseur

IV.2.2 Porte AND

Alors le programme de la Porte AND est comme suite :

```
Library ieee ;
Use ieee.std_logic_1164.all;
Entity AND_EX is
   Port (
        A: in bit;
        B: in bit;
       F: out bit
       );
End AND_EX;
Architecture EX_ARCH of AND_EX is
Begin
F<=A and B;
End AND_EX_ARCH;
```

Et enfin dans la rubrique **EX_ARCH** ou Simulation, on simule notre porte en donnant des valeurs pour l'entrée A et B pour voir la sortie F :

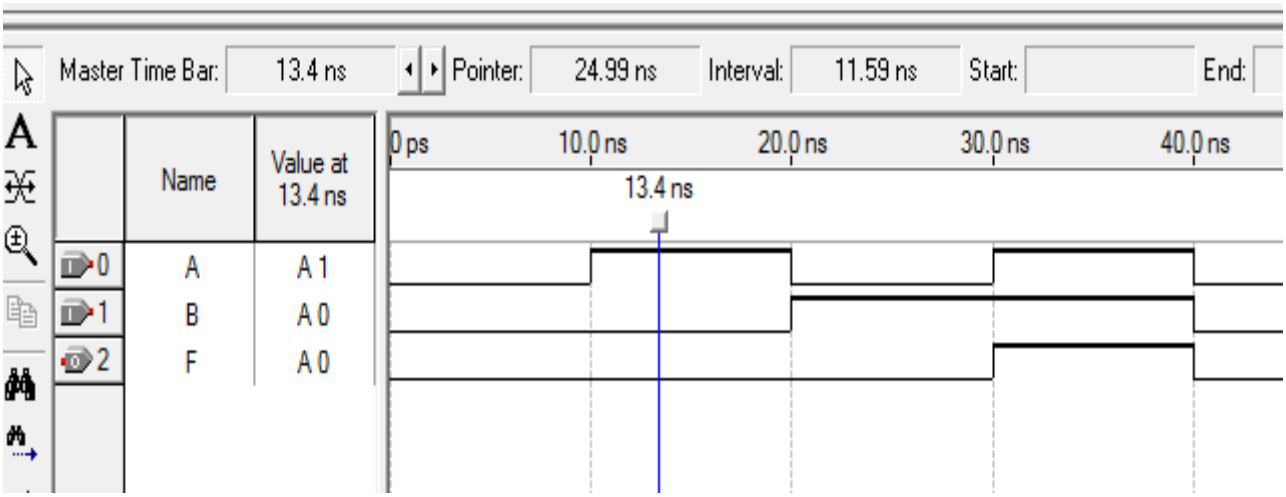


***Figure IV.20*: la simulation de la porte AND**

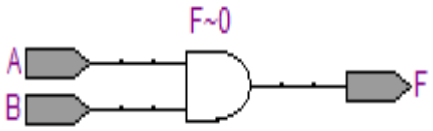


Figure IV.21. Vue RTL de la porte AND

IV.2.3 Porte OR

Alors le programme de la porte OR est comme suite :

Le même programme qui précède sauf que on change F <=A or B;

Et enfin dans la rubrique **porte EX_ARCH** ou Simulation, on simule notre porte en donnant des valeurs pour l'entrée A et B pour voir la sortie F:

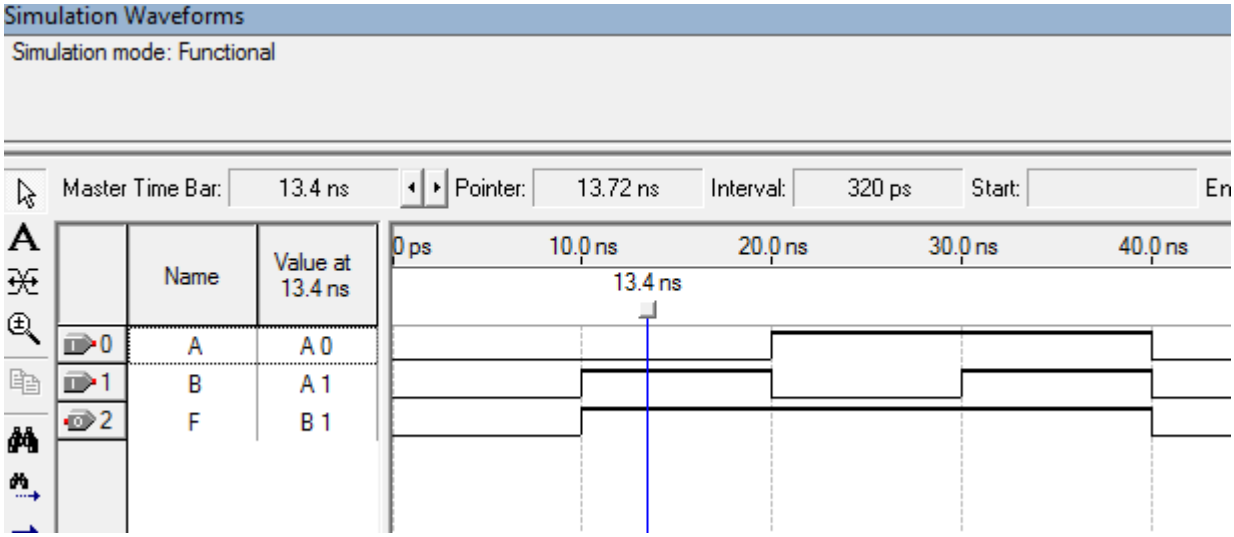


Figure IV.22: la simulation de la porte OR

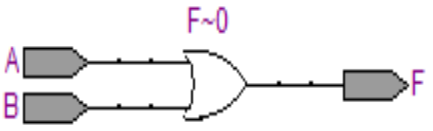


Figure IV.23. Vue RTL de la porte OR

IV.2.4 Porte NAND

Alors le programme de la porte NAND est comme suite :

Le même programme qui précède sauf que on change F <=A nand B;

Et enfin dans la rubrique **EX_ARCH** ou Simulation, on simule notre porte en donnant des valeurs pour l'entrée A et B pour voir la sortie F:

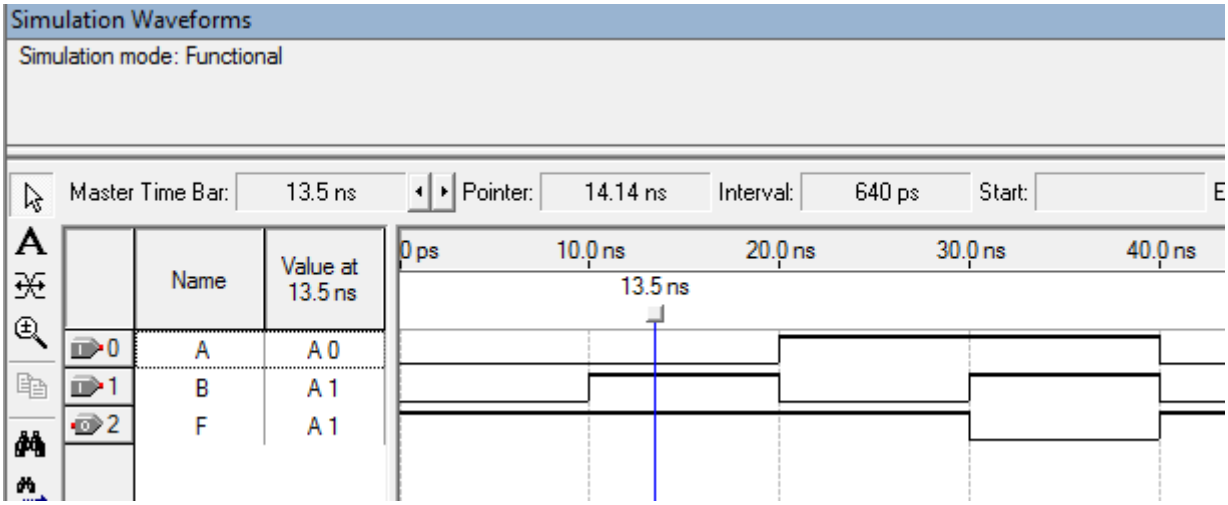


Figure IV.24 **la simulation de la porte_Nand**

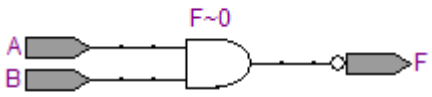


Figure IV.25. **Vue RTL de la porte_Nand**

IV.2.5 Porte NOR

Alors le programme de la porte NOR est comme suite :

Le même programme qui précède sauf que on change F <=A nor B;

Et enfin dans la rubrique **EX_ARCH** ou Simulation, on simule notre porte en donnant des valeurs pour l'entrée A et B pour voir la sortie F:

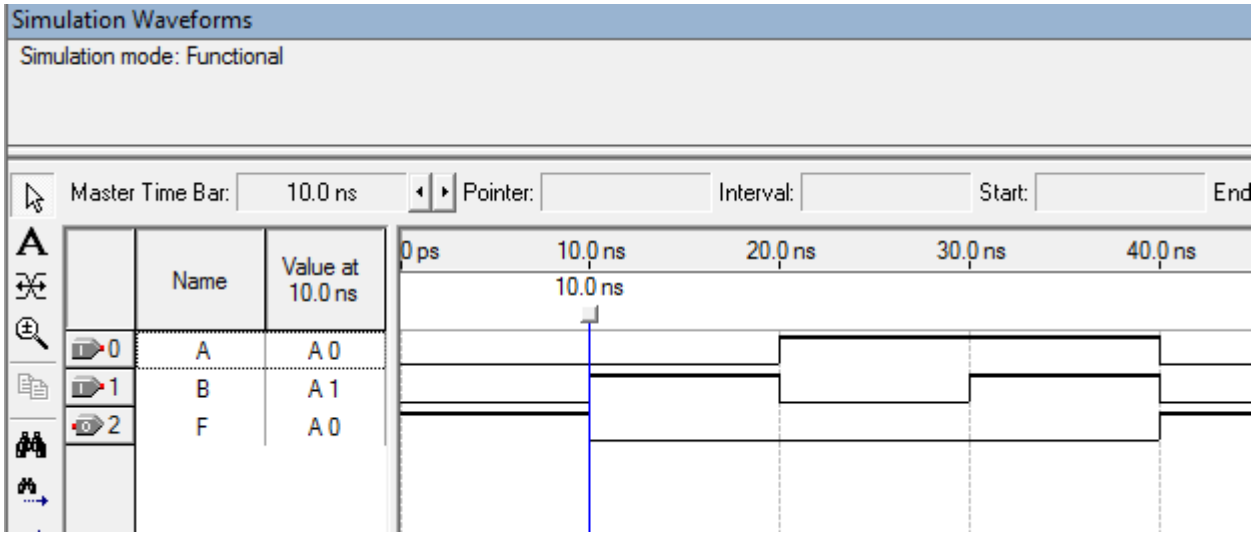


***Figure IV.26*: la simulation de la porte_nor**

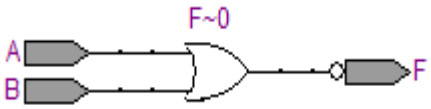


***Figure IV.27*. Vue RTL de la porte_nor**

IV.2.6 Porte XOR

Alors le programme de la porte XOR est comme suite :

Le même programme qui précède sauf que on change F <=A xor B;

Et enfin dans la rubrique **EX_ARCH** ou Simulation, on simule notre porte en donnant des valeurs pour l'entrée A et B pour voir la sortie F:

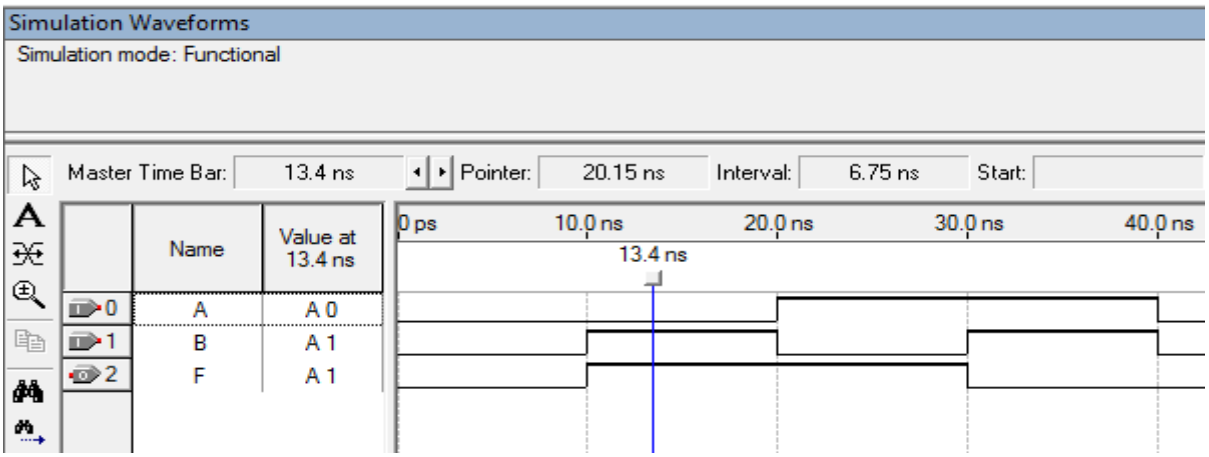


***Figure IV.28*: la simulation de la porte_xor**

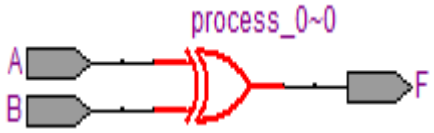


Figure IV.29. **Vue RTL de la porte_XOR**

IV.2.7 Décodeur

Alors le programme du décodeur est comme suite :

```
Library ieee ;
Use ieee.std_logic_1164.all;
Entity decoder is
Port ( A : in bit_vector (1 downto 0);
D: out bit_vector (3 downto 0) );
End decoder;
Architecture decoder_ARCH of decoder is
Begin
With (A) select
D <= "1000" when "00",
"0100" when "01",
"0010" when "10",
"0001" when others ;
End decoder_ARCH;
```

Et enfin dans la rubrique **decoder_ARCH** Simulation, on simule notre décodeur en donnant des valeurs pour l'entrée A0 et A1 pour voir la sortieD0,D1,D2 et D3:

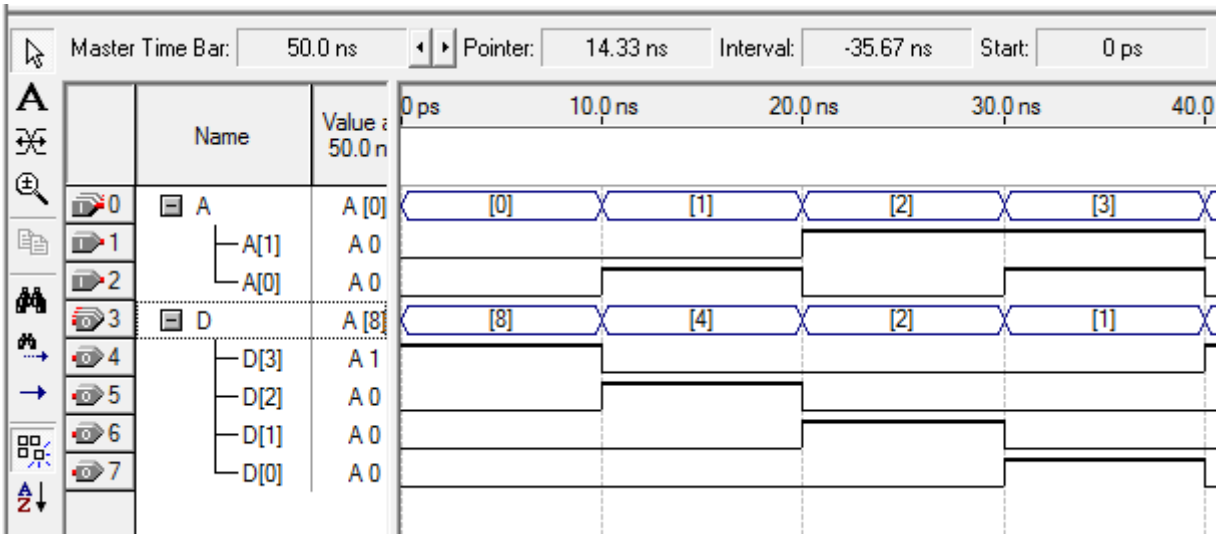


Figure IV.30*: simulation du décodeur *2 à 4

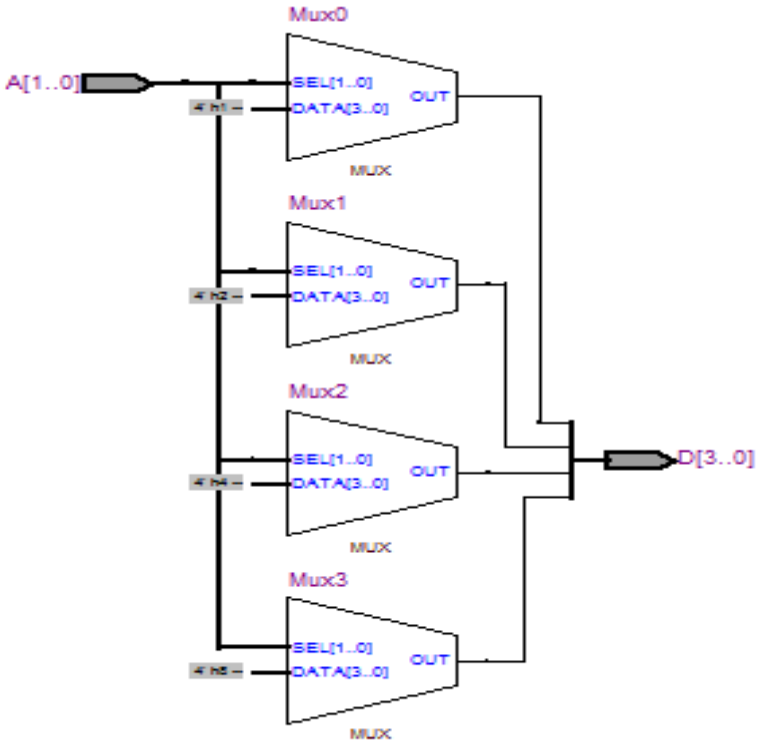


Figure IV.31*. Vue RTL du décodeur *2 à 4

IV.2.8 Encodeurs

Alors le programme de encoder est comme suite :

```
Library ieee ;

Use ieee.std_logic_1164.all;

Entity encoder is Port (ABCD: in bit_vector (3 downto 0) ;

        YZ: out bit_vector(1downto 0));

End  encoder ;

Architecture  encoder_ARCH of  encoder is

Begin

  With (ABCD) select

YZ<="00"when "0001",

   "01"when "0010",

   "10"when "0100",

   "11"when "1000",

   "00" when others;

End encoder_ARCH;
```

Et enfin dans la rubrique **encoder_ARCH** Simulation, on simule notre encoder en donnant des valeurs pour l'entrée A,B,C et D pour voir la sortie Y et Z :

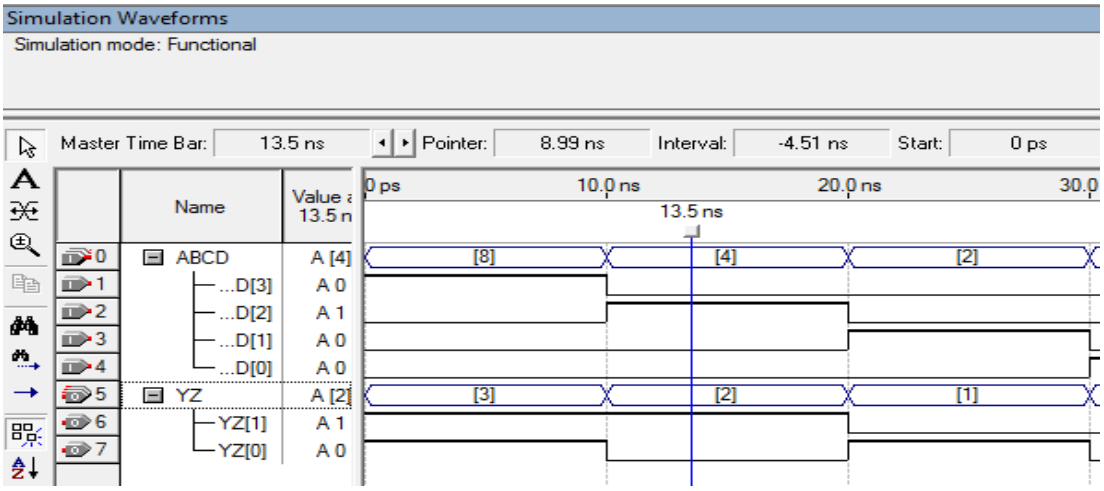


***Figure IV. 32*: simulation Encodeur**

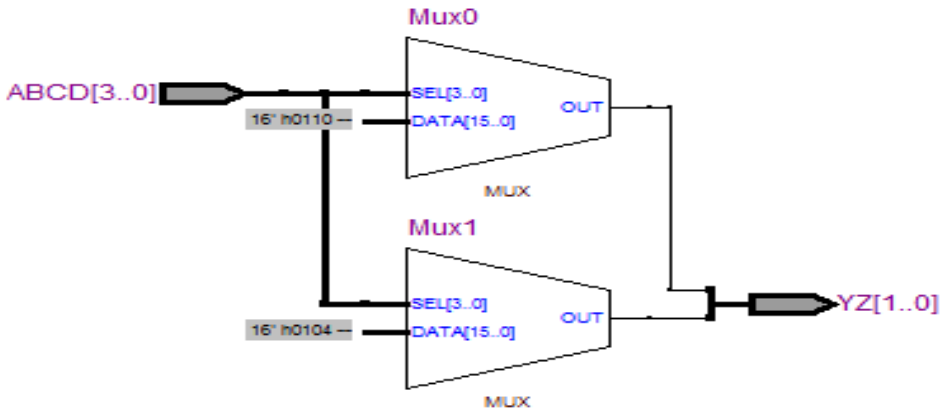


Figure IV.33 **Vue RTL Encodeur**

IV.2.9 Afficheurs 7 segments

Un décodeur d'affichage à 7 segments est un circuit utilisé pour conduire des affichages de caractères qui sont couramment trouvés dans des applications telles que les horloges numériques et les appareils ménagers. Un affichage de caractères est composé de sept LED individuelles, généralement étiquetées a–g. L'entrée du décodeur est l'équivalent binaire du caractère décimal ou hexadécimal qui doit être affiché. La sortie du décodeur est la disposition des LED qui formeront le caractère. Les décodeurs à deux entrées peuvent faire passer les caractères « 0 » à « 3 ». Les décodeurs à trois entrées peuvent faire passer les caractères « 0 » à « 7 ». Les décodeurs à quatre entrées peuvent faire passer les caractères « 0 » à « F», le cas des caractères hexadécimaux étant « A, b, c ou C, d, E et F ». Examinons un exemple de conception manuelle d'un décodeur à trois entrées à sept segments. La première étape du processus est de créer la table de vérité pour les sorties qui va conduire les LEDs dans l'écran. Nous allons appeler ces sorties S0,S1, ...,S6. L'exemple 6.4 montre comment construire la table de vérité pour le décodeur d'affichage à 7 segments. Dans ce tableau, une logique 1 correspond à la LED étant allumée.[1]

Table de vérité:

E0	E1	E2	E3	S0	S1	S2	S3	S4	S5	S6	AF
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9
1	0	1	0	1	1	1	0	1	1	1	A
1	0	1	1	0	1	1	1	1	0	1	B
1	1	0	0	1	0	0	1	1	1	0	C
1	1	0	1	0	0	1	1	1	1	1	D
1	1	1	0	1	0	0	1	1	1	1	E
1	1	1	1	1	0	0	0	1	1	1	F

AF: affichage 7 segments

Symbole :

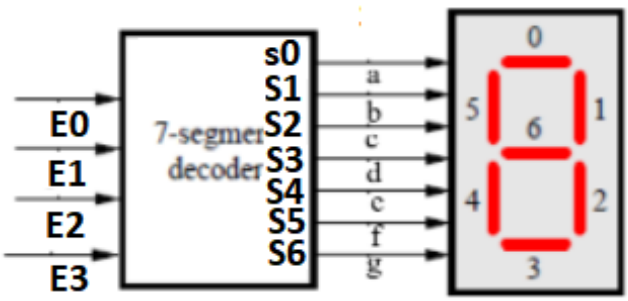


Figure IV.34: Symbole 7 segments

Alors le programme de 7_segment est comme suite :

```
Library ieee ;
Use ieee.std_logic_1164.all;
Entity ex_7_segment is
   Port ( E: in std_logic_vector(3 downto 0) ;S: out std_logic_vector(0 to 6) );
End  ex_7_segment;
Architecture   ex_7_segment_ARCH of ex_7_segment  is
Begin
  With (E) select
S <=     "0000001" when  "0000",
        "1001111" when  "0001",
        "0010010" when  "0010",
        "0000110" when  "0011",
        "1001100" when  "0100",
        "0100100" when  "0101",
        "0100000" when  "0110",
        "0001111" when  "0111",
        "0000000" when  "1000",
        "0000100" when  "1001",
        "0001000" when  "1010",
        "1100000" when  "1011",
        "0110001" when  "1100",
        "1000010" when  "1101",
        "0110000" when  "1110",
        "0111000" when  "1111",
        "0000000" when others;
End ex_7_segment_ARCH;
```

SIGNAL	FPGA pin	Description
HEX0 [0]	PIN_J2	Seven Segment digit 0[0]
HEX0 [1]	PIN_J1	Seven Segment digit 0[1]
HEX0 [2]	PIN_H2	Seven Segment digit 0[2]
HEX0 [3]	PIN_H1	Seven Segment digit 0[3]
HEX0 [4]	PIN_F2	Seven Segment digit 0[4]
HEX0 [5]	PIN_F1	Seven Segment digit 0[5]
HEX0 [6]	PIN_E2	Seven Segment digit 0[6]

Table. Brochage de l'afficheur 7 segments HEX0 (carte DE1)

Et enfin dans la rubrique **ex_7_segment_ARCH** Simulation, on simule notre 7_segment en donnant des valeurs pour l'entrée E0, E1, E2 et E3 pour voir la sortie S:

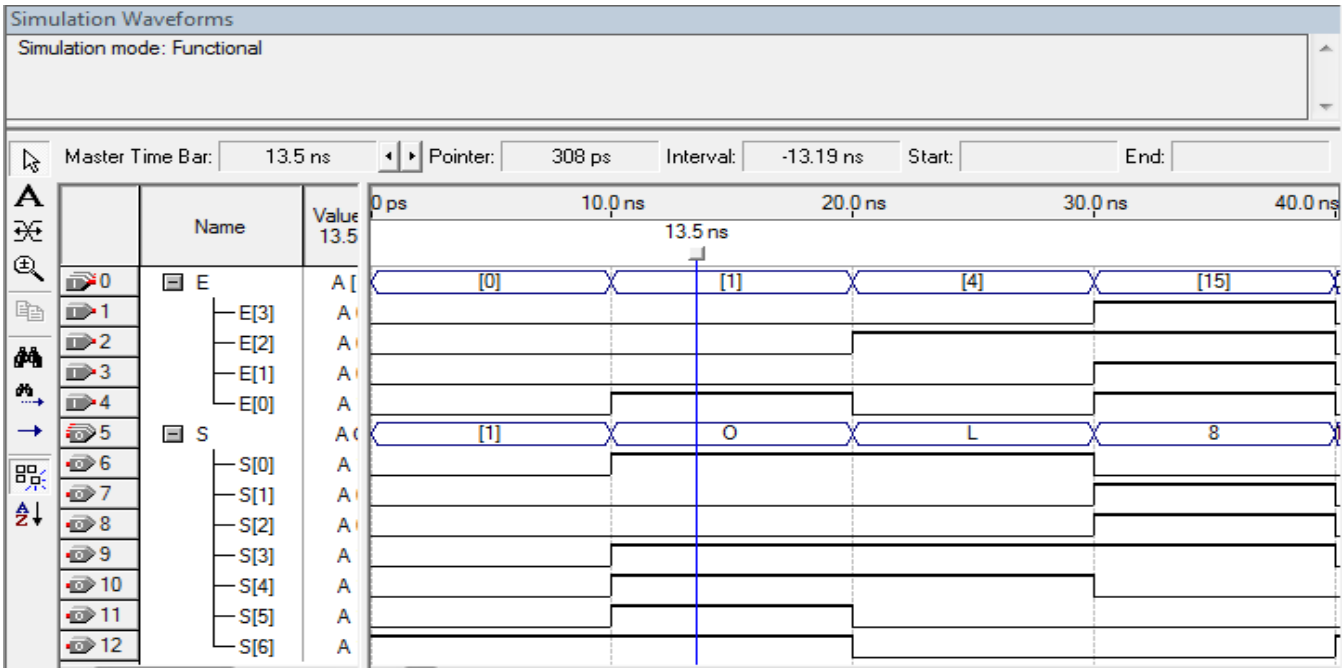


***Figure IV.35*: la simulation de ex_7_segment**

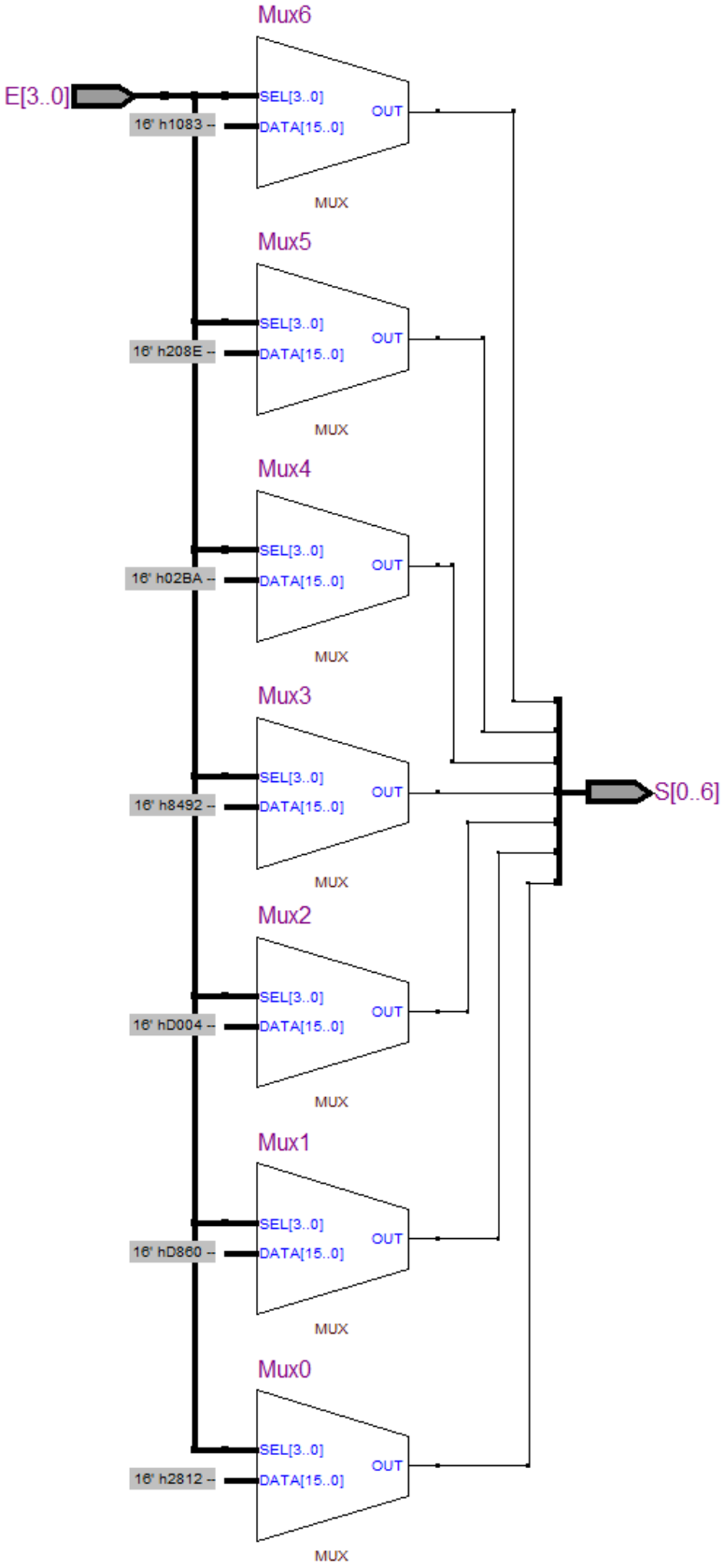


Figure IV.36. **Vue RTL de x_7_segment**

IV.2.10 Multiplexeur 4 vers 1

Alors le programme de la porte_ multi est comme suite :

```
Library ieee ;

Use ieee.std_logic_1164.all;

Entity port_multi is

   Port (

        sul0 : in bit;

        sul1 : in bit;

        A: in bit;

        B: in bit;

        C: in bit;

        D: in bit;

       F: out bit

       );

End port_multi;

Architecture port_multi_ARCH of port_multi is

Begin

F <=( A and not sul0 and not sul1) or ( B and  sul0 and  not sul1 ) or ( C and  not sul0 and sul1 ) or ( D and sul0 and sul1 );

End port_multi_ARCH;
```

Et enfin dans la rubrique **port_multi_ARCH** Simulation, on simule notre Multiplexeur en donnant des valeurs pour l'entrée A et B pour voir la sortie F:

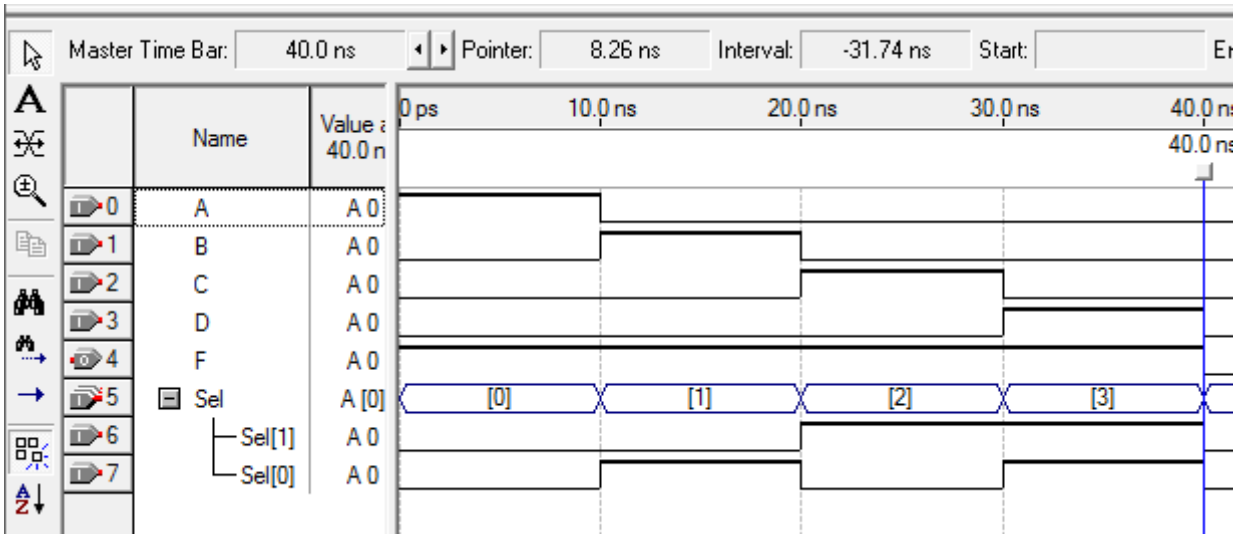


***Figure IV.37*: simulation du circuit multiplexeur**

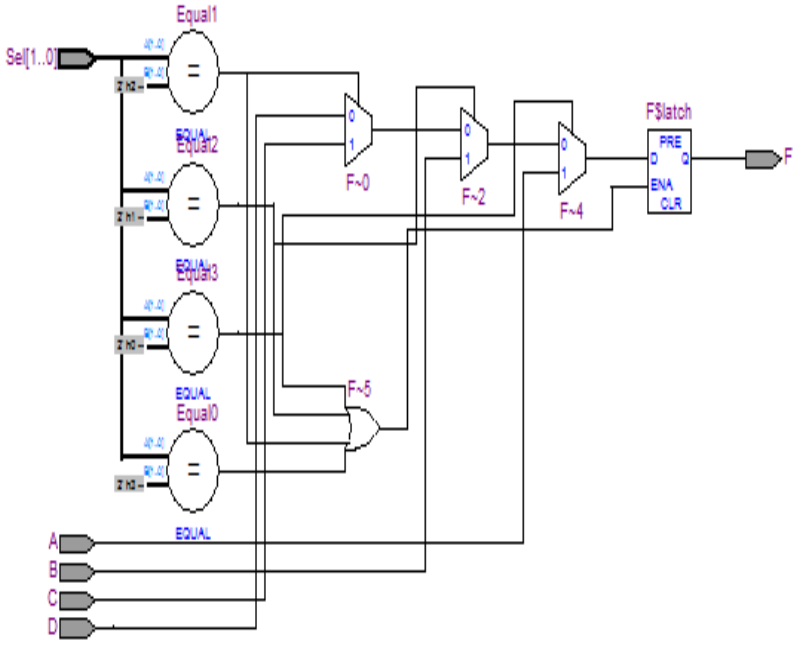


***Figure IV.38.*Vue RTL du circuit multiplexeur**

IV.2.11 Démultiplexeurs 1 vers 4

Alors le programme de EX_DEMULT est comme suite :

```
Library ieee ;
Use ieee.std_logic_1164.all;
Entity EX_DEMULT is
   Port (
        sul0 : in bit;
        sul1 : in bit;
         A : in bit;
        W: out bit;
        X: out bit;
        Y: out bit;
        Z: out bit
       );
End EX_DEMULT;
Architecture EX_DEMULT_ARCH of EX_DEMULT is
Begin
W <=( A and not sul0 and not sul1);
X<=( A and  sul0 and  not sul1 ) ;
Y <= ( A and  not sul0 and sul1 );
Z <=( A and sul0 and sul1 );
End EX_DEMULT_ARCH;
```

Et enfin dans la rubrique **EX_DEMULT_ARCH** Simulation, on simule notre Démultiplexeurs en donnant des valeurs pour l'entrée A, sul0 , sul1 et sel pour voir la sortie W,X,Y,Z :

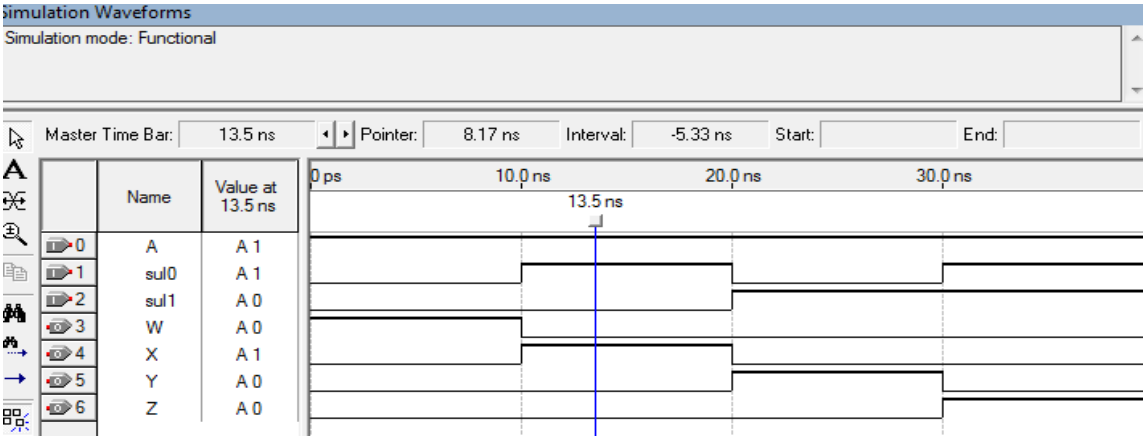


Figure IV.39: la simulation de circuit démultiplexeur

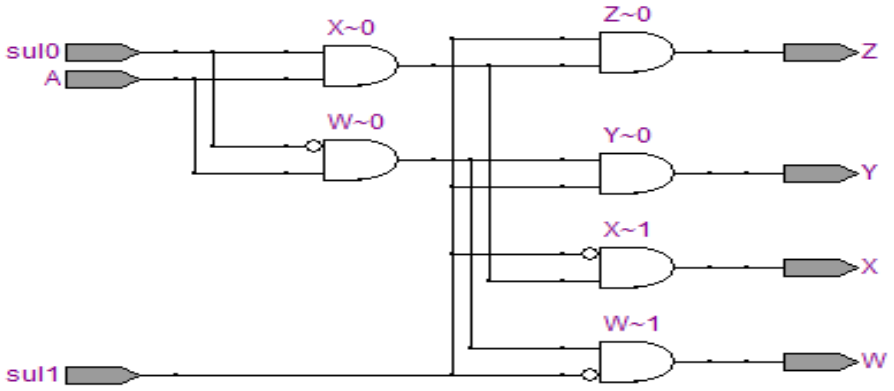


Figure IV.40. Vue RTL du circuit démultiplexeur

IV.3 Les instructions du mode séquentiel

IV.3.1 Bascule

IV.3.1.1 Bascule D

Le latch est considéré comme une mémoire. Tant que clk est à '1', la sortie q prend les valeurs de d. Lorsque clk passe à '0', la dernière valeur de d est mémorisée. Programmer un Latch.

Alors le programme de la d_latch est comme suite :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity d_latch is
    port ( clk  : in  std_logic;
           d : in  std_logic;
           q : out  std_logic;
           qb : out  std_logic);
 end d_latch;
architecture d_latch_arch of d_latch is
signal t1, t2 : STD_LOGIC; --tampons
begin
process (clk)
begin
if clk= '1' then
t1<= d; t2<= not d;
else
t1<= t1; t2<= t2;
end if;
end process;
q<= t1; qb<= t2;
end  d_latch_arch;
```

Et enfin dans la rubrique **d_latch_arch** Simulation, on simule notre bascule D latch en donnant des valeurs pour l'entrée reset, d et clk pour voir les sorties q et qb :

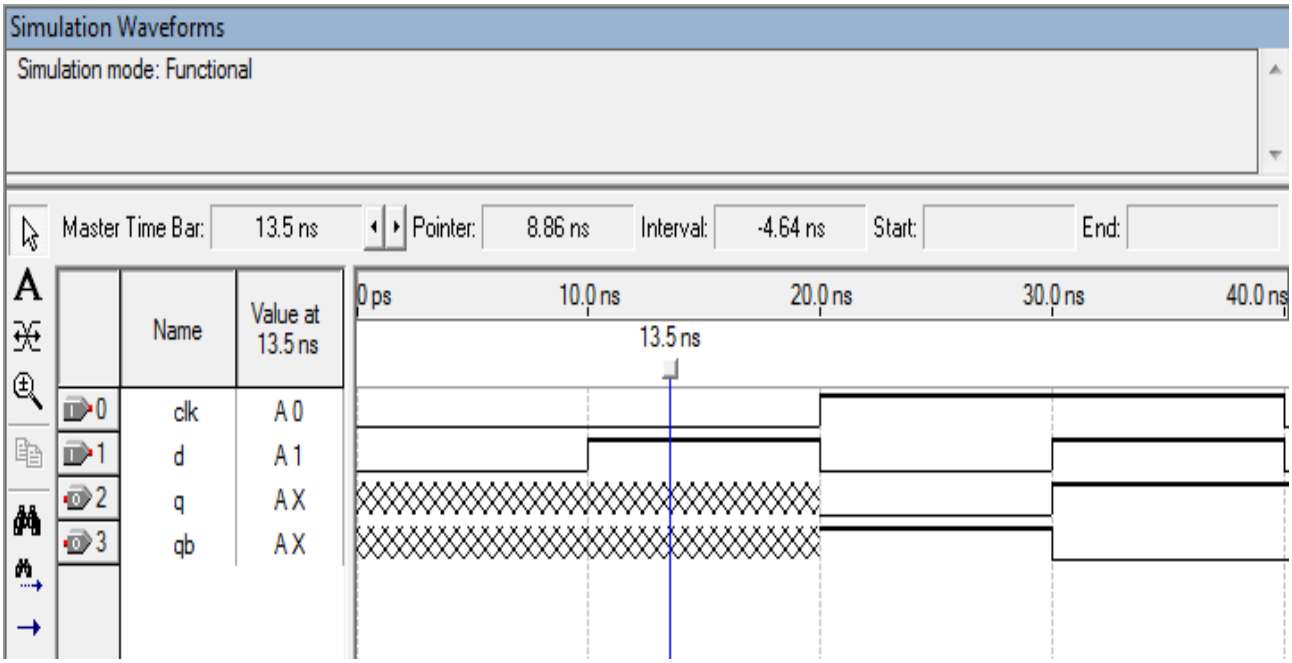


Figure IV.41: la simulation de la bascule d_latch

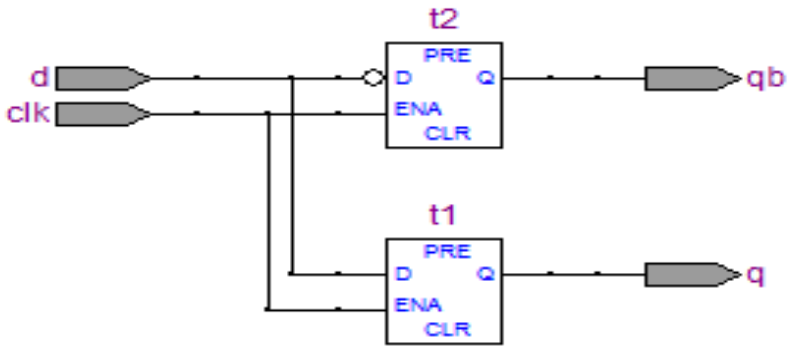


***Figure IV.42*. Vue RTL de la bascule d_latch**

Alors le programme de la bascule_d est comme suite :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity  bascule_d is
   Port ( reset : in  STD_LOGIC;
          D: in  STD_LOGIC;
          Clk : in  STD_LOGIC;
          Q : out  STD_LOGIC;
          Qb : out  STD_LOGIC);
end bascule_d;
architecture bascule_d_ARCH of bascule_d  is
begin
process (clk)
begin
if clk='1' and clk'event then
  if reset= '1' then Q<='0'; Qb<='1';
 else Q<= D; Qb<= not D;
end if;
end if;
end process;
end bascule_d_ARCH;
```

Et enfin dans la rubrique **bascule_d_ARCH** Simulation, on simule notre bascule_D on donnant des valeurs pour l'entrée reset, d et clk pour voir les sorties q et qb:

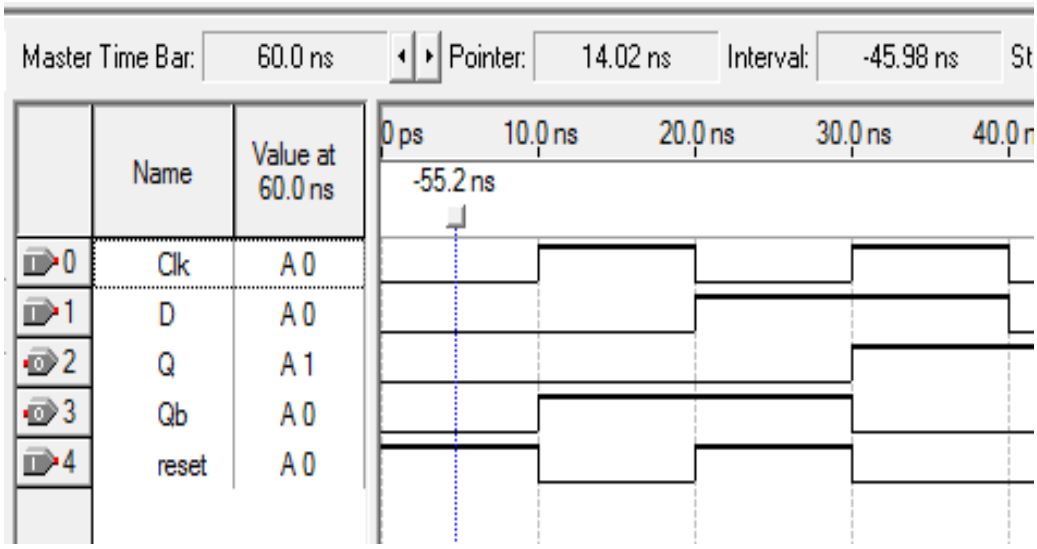


Figure IV.43: la simulation de la bascule_d

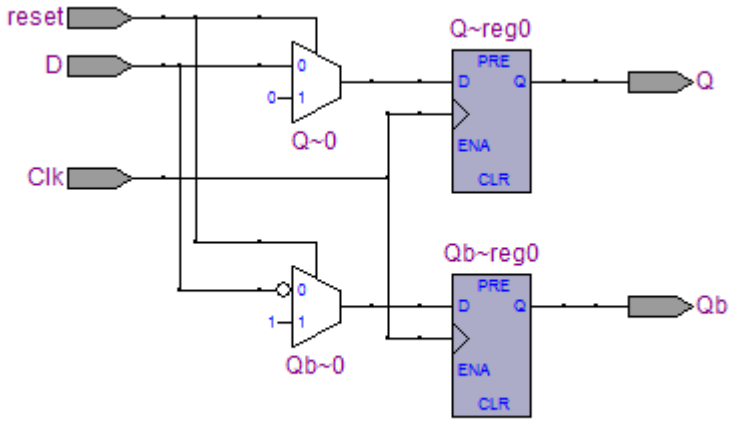


Figure IV.44. Vue RTL de la bascule_d

IV.3.1.2 Bascule T

Alors le programme de la BASCULET est comme suite :

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity BASCULET is
port (
D,CLK : in std_logic;
S : out std_logic);
end BASCULET;
architecture DESCRIPTION of BASCULET is
signal S_INTERNE : std_logic; -- Signal interne
begin
PRO_BASCULET : process (CLK)
Begin
if (CLK'event and CLK ='1') then
if (D ='1') then
S_INTERNE <= not (S_INTERNE);
end if;
end if;
end process PRO_BASCULET;
S <= S_INTERNE;
end DESCRIPTION;
```

Et enfin dans la rubrique **DESCRIPTION** Simulation, on simule notre bascule T on donnant des valeurs pour l'entrée D et Clk pour voir la sortie S :

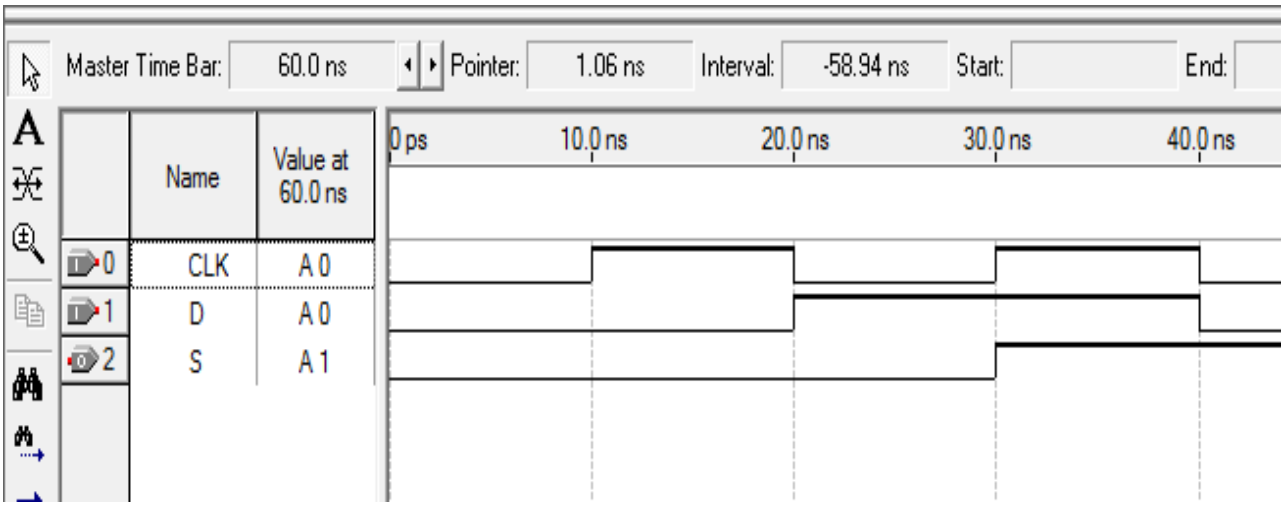


Figure IV.45: la simulation de BASCULET

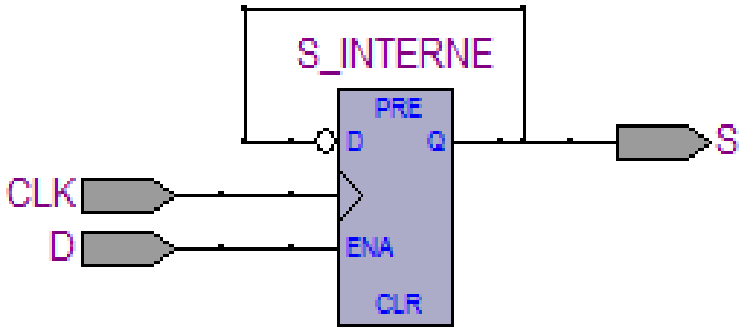


Figure IV.46. Vue RTL de la BASCULET

IV.3.2 Compteur

Alors le programme d'un compteur_simple est comme suite :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity compteur_simple is
Port ( clk : in  std_logic;
q : out  std_logic_vector (2 downto 0));
End compteur_simple;
architecture compteur_simple_arch of compteur_simple is
signal S: std_logic_vector (2 downto 0);
begin
process (clk)
begin
if  rising_edge (clk) then
S <= S + 1;
else
S<= S;
end if;
end process;
q<= S;
end compteur_simple_arch;
```

Et enfin dans la rubrique **compteur_simple_arch** Simulation, on simule notre compteur on donnant des valeurs pour l'entrée clk pour voir les sorties q0, q1et q2 :

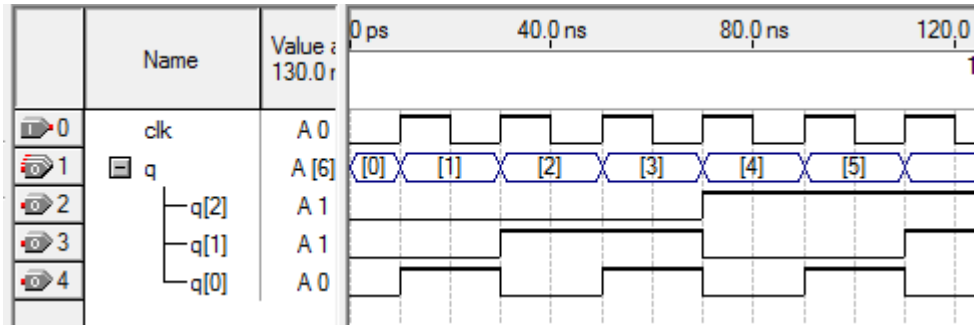


Figure IV.47:la simulation du compteur_simple

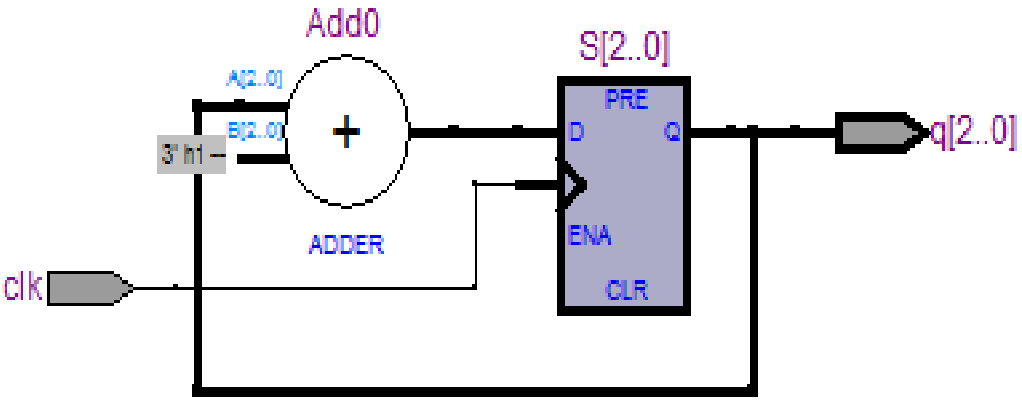


Figure IV.48. Vue RTL du compteur_simple

le programme d'un pour le compteur modulo 6 est comme suite :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity compteur_modulo3 is
   port ( clk, reset : in  std_logic;
         modul : out  std_logic_vector (2 downto 0));
end compteur_modulo3;
architecture compteur_modulo3_arch of compteur_modulo3 is
signal temp: std_logic_vector (2 downto 0);
begin
process (reset, clk)
begin
     if reset= '1' then  temp<="000";
     else  if clk'event and clk= '1' then
            if temp ="011"  then  temp <="000";
            else temp <= temp + 1;
            end if;
     end if;
     end if;
end process;
modul<= temp;
end compteur_modulo3_arch;
```

Et enfin dans la rubrique compteur_modulo3_arch Simulation, on simule notre compteur on donnant des valeurs pour l'entrée reset, Clk pour voir la sortie modul 0, modul 1et modul 2 :

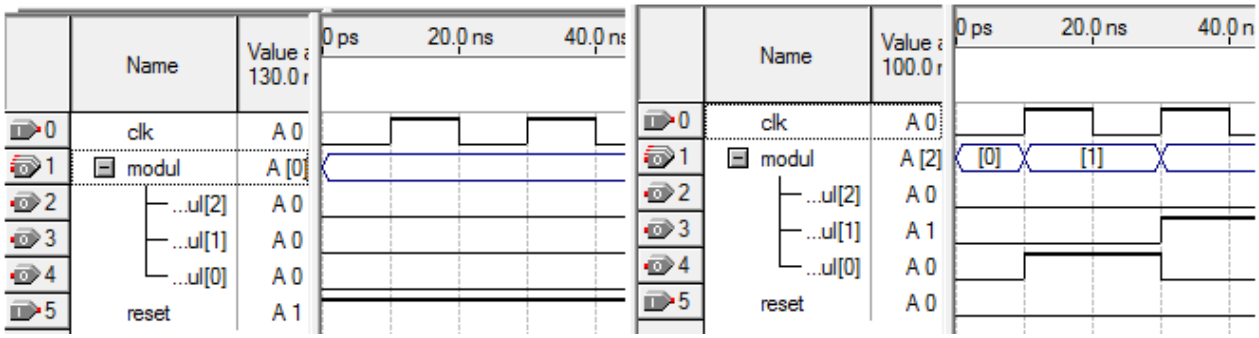


Figure IV.49: la simulation du compteur_modulo3

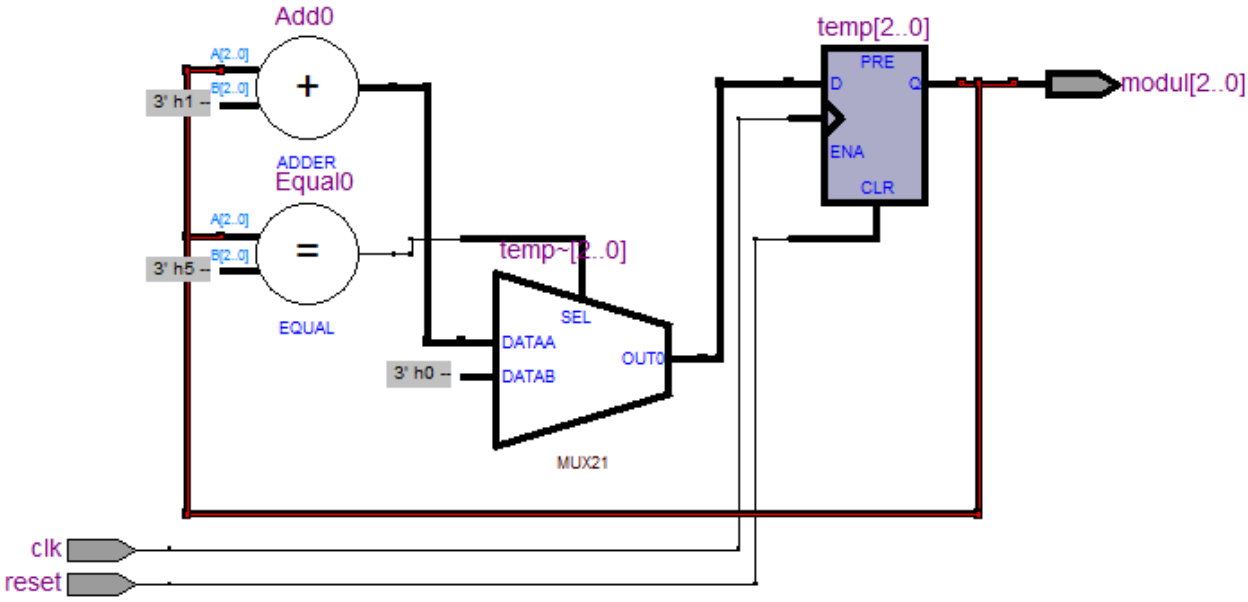


Figure IV.50. Vue RTL du compteur_modulo3

Le programme pour le compteur_decimal avec reset asynchrone active à l'état haut est comme suite

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity compteur_decimal is
   port ( clk, reset : in  std_logic;
         digit : out  std_logic_vector (3 downto 0));
end compteur_decimal;
architecture arch_decimal of compteur_decimal is
signal temp: std_logic_vector (3 downto 0);
begin
process (reset, clk)
begin
     if reset= '0' then  temp<="0000";
     else  if clk'event and clk= '1' then
            if temp ="1001"  then  temp <="0000";
            else temp <= temp + 1 ;
            end if;
          end if;
     end if;
end process;
digit<= temp;
end arch_decimal;
```

Et enfin dans la rubrique **arch_decimal** Simulation, on simule notre compteur_decimal on donnant des valeurs pour l'entrée reset, d et Clk pour voir la sortie q et qb :

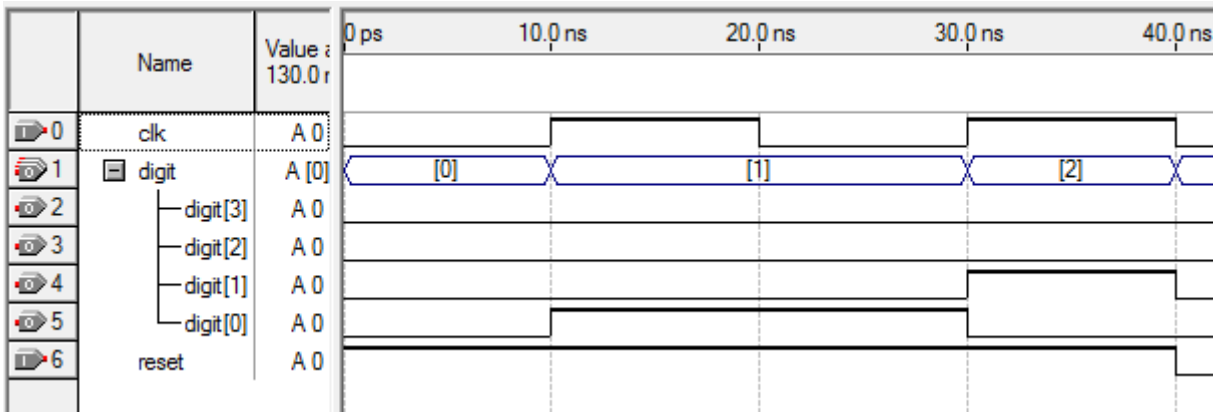


Figure IV.51: la simulation du compteur_decimal .

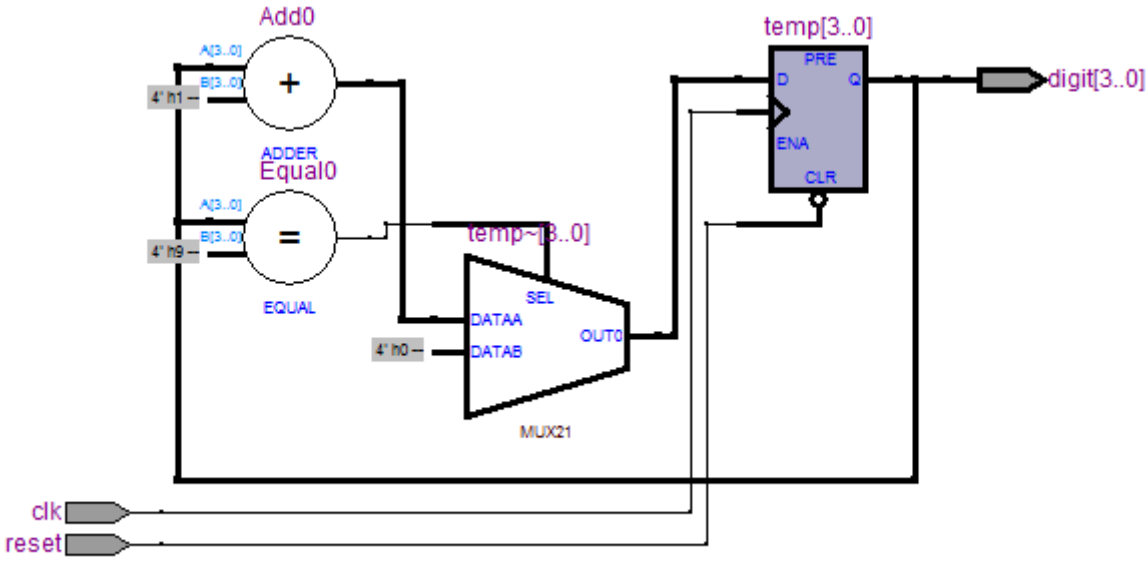


Figure IV.52Vue RTL du compteur_decimal .

IV.3.3 Les Registres

Alors le programme d'un registre_Nbits est comme suite :

*

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;
Entity registre_Nbits is
   Generic (n: natural:=4);
   Port (reset: in STD_LOGIC;
         D: in STD_LOGIC_VECTOR (N-1 downto 0);
         Clk: in STD_LOGIC;
         Q: out STD_LOGIC_VECTOR (N-1 downto 0);
         Qb: out STD_LOGIC_VECTOR (N-1 downto 0));
End registre_Nbits;
Architecture Behavioral of registre_Nbits is
Signal s, sb: std_logic_vector (n-1 downto 0);
Begin
Process (clk, reset,d)
Begin
If reset= '1' then s<=(others=>'0'); sb<=(others=>'1');
Else if clk='1' and clk'event then s<=d; sb<=not d;
Else s<= s; sb<= sb;
End if;
End if;
End process;
q<=s; qb<=sb;
End Behavioral;
```

Et enfin dans la rubrique **behavioral** Simulation, on simule notre registre_Nbits on donnant des valeurs pour l'entrée reset, d et Clk pour voir la sortie q et qb :

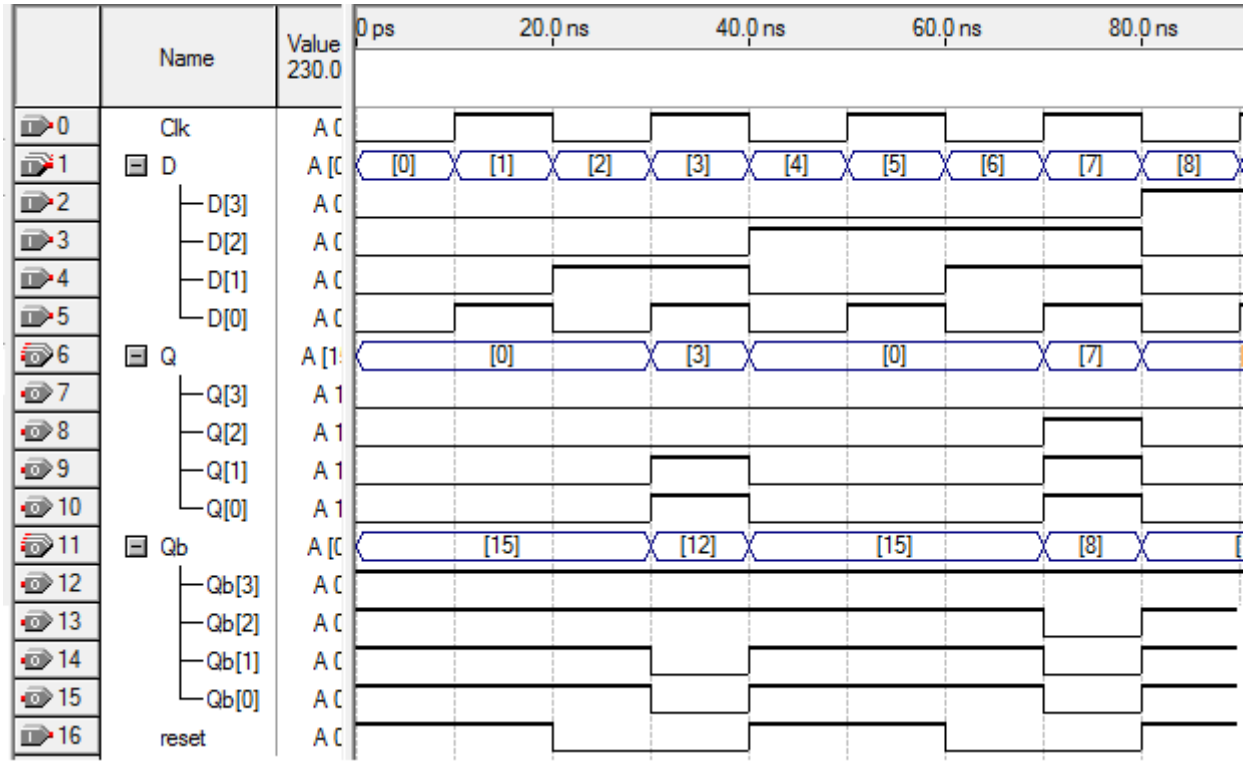


Figure IV.53: la simulation du registre à Nbits (N=16)

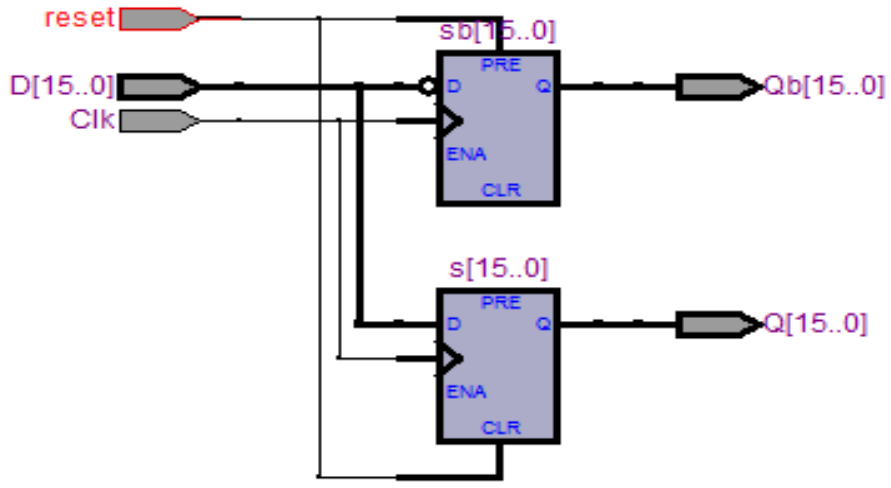


Figure IV.54.Vue RTL du registre à Nbits (N=16)

Alors le programme registre composé de 4 bascules D latch est comme suite :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity regis_latch is
   port ( clk, reset  : in  std_logic;
        d : in  std_logic_vector (3 downto 0);
        q : out  std_logic_vector (3 downto 0);
        qb : out  std_logic_vector (3 downto 0));
end regis_latch;
architecture regis_latch_arch of regis_latch is
signal E1, E2: std_logic_vector (3 downto 0); -- tampons
begin
process (d, reset, clk)
begin
    if reset= '1' then E1<="0000"; E2<="1111";
    else if clk= '1' then
         E1<= d;  E2<= not d;
         else E1<= E1;  E2<= E2;
    end if;
        end if;
    end process;
    q<= E1;  qb<= E2;
end regis_latch_arch;
```

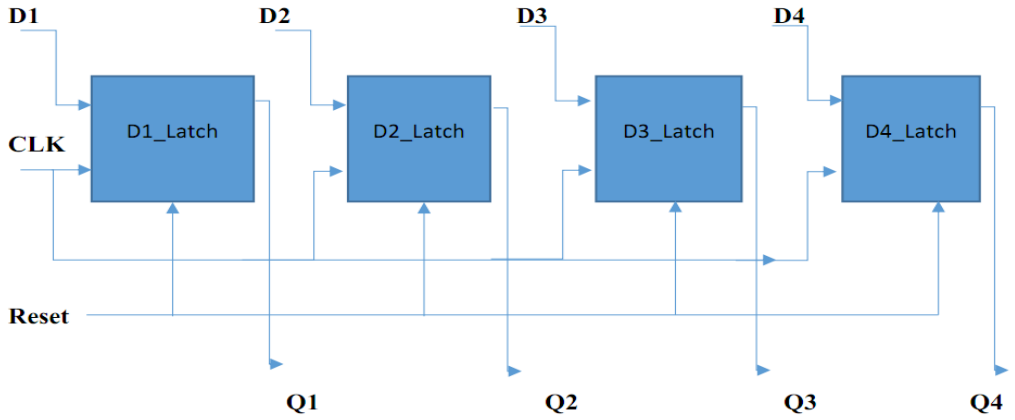


Figure IV.55: Registre composé de 4 bascules D latch.

D1	D2	D3	D4	reset	CLK	Q0	Q1	Q2	Q3	Qb0	Qb1	Qb2	Qb3
0	0	1	1	0	0	0	0	1	1	1	1	0	0
1	1	0	0	0	1	1	1	0	0	0	0	1	1
0	0	1	1	1	0	0	0	1	1	1	1	0	0

Et enfin dans la rubrique **regis_latch_arch** Simulation, on simule notre registre composé de 4 bascules D latch on donnant des valeurs pour l'entrée reset, clk pour voir les sorties q et qb :

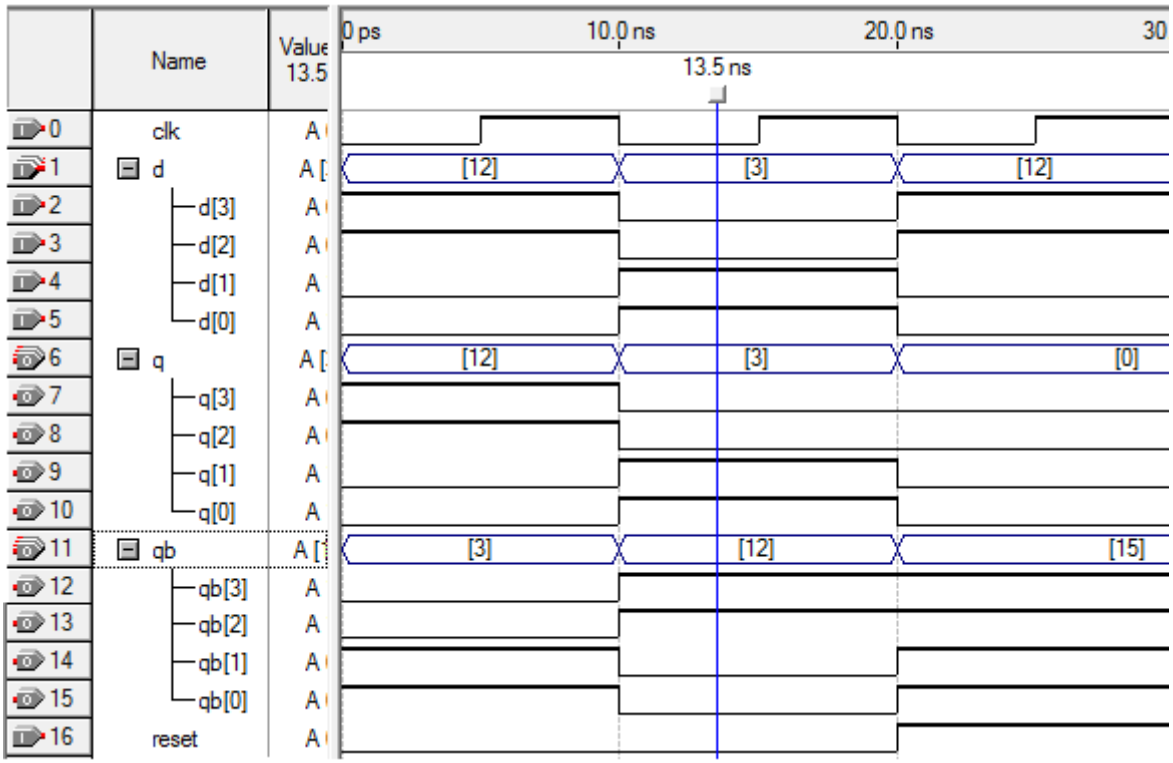


Figure IV.56: la simulation de regis_latch

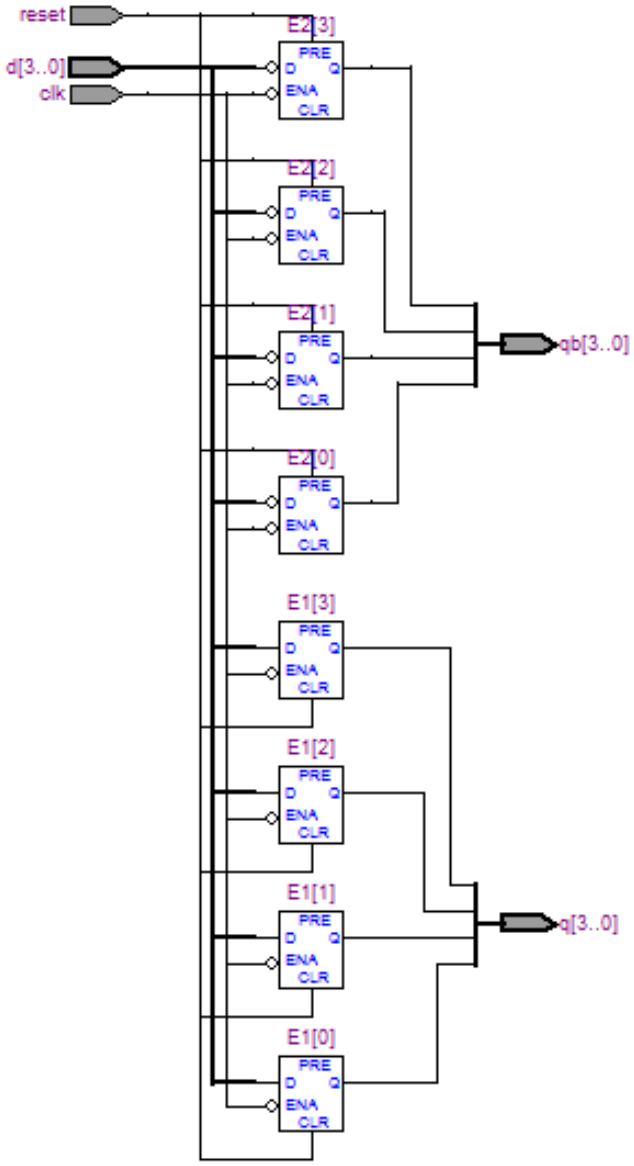


Figure IV.57 Vue RTL de registre_latch

Alors le programme registre_5bits est comme suite :

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;
Use IEEE.STD_LOGIC_ARITH.ALL;
Use IEEE.STD_LOGIC_UNSIGNED.ALL;
Entity registre_5bits is
   Port (reset: in  STD_LOGIC;
         D: in  STD_LOGIC_VECTOR (4 downto 0);
         Clk : in  STD_LOGIC;
         Q : out  STD_LOGIC_VECTOR (4 downto 0);
         Qb : out  STD_LOGIC_VECTOR (4 downto 0));
End registre_5bits;
Architecture Behavioral of registre_5bits  is
Signal s,sb:std_logic_vector(4 downto 0);
Begin
Process (clk,reset,d)
Begin
If reset= '1' then s<="00000"; sb<="11111";
Else if clk='1' and clk'event then s<=d; sb<=not d;
Else s<= s; sb<= sb;
End if;
End if;
End process;
q<=s; qb<=sb;
End Behavioral;
```

Et enfin dans la rubrique **Behavioral** Simulation, on simule notre registre_5bits on donnant des valeurs pour l'entrée reset, d et clk pour voir les sorties q et qb :

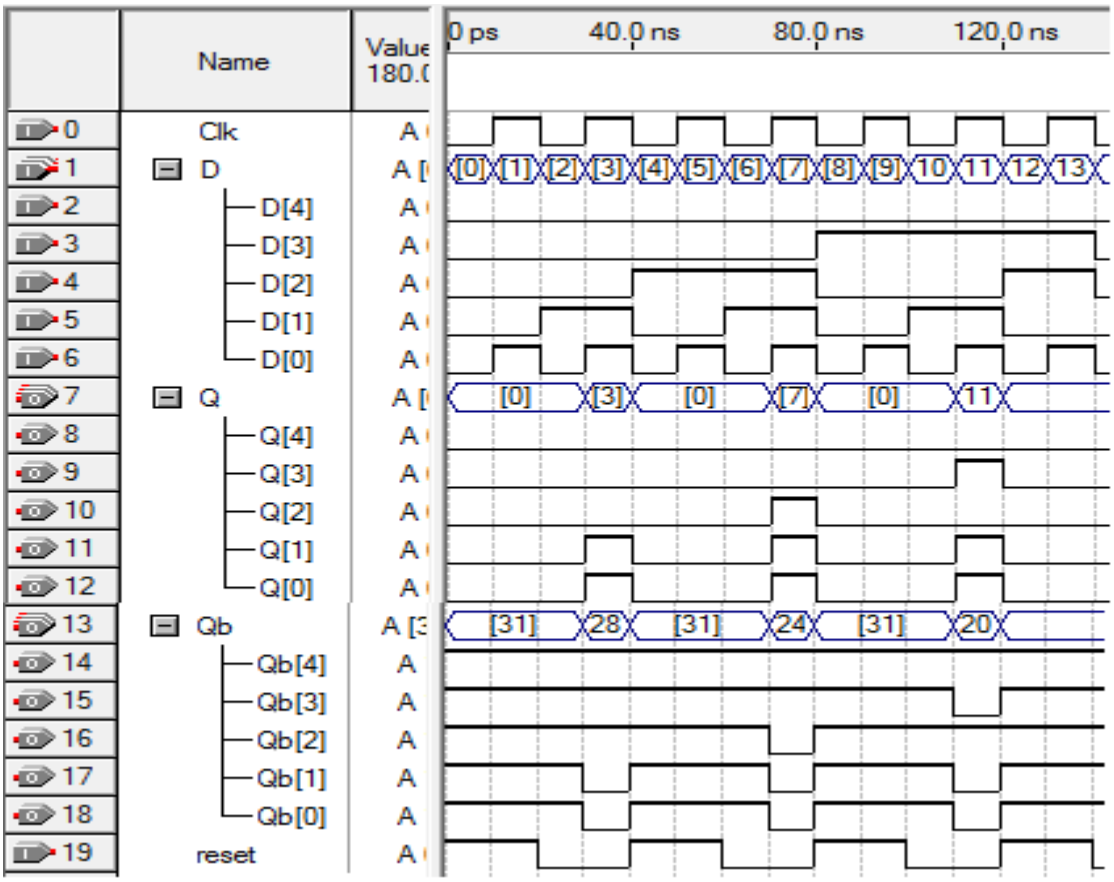


Figure IV.58: la simulation de registre_5bits

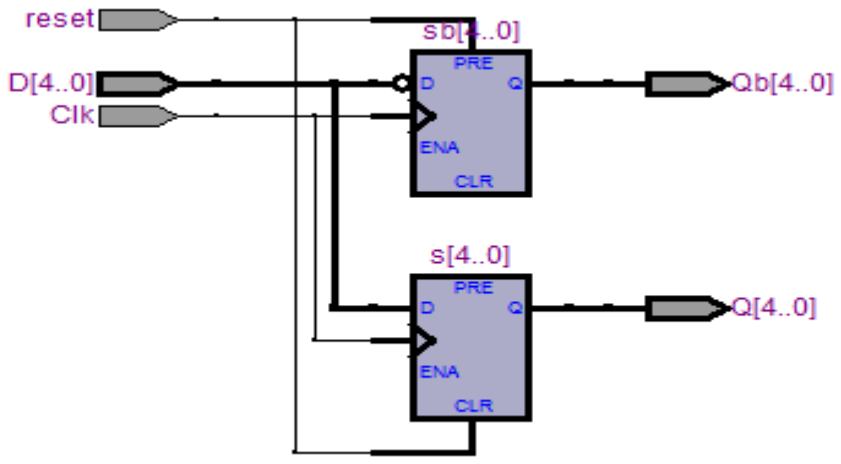


Figure IV.59Vue RTL de registre_5bits

IV.4 Conclusion

A travers le logiciel Quartus 2,on a pu utiliser ce dernier afin de programmer les circuits combinatoires et séquentiels par la programmation VHDL. Ce logiciel nous aide non seulement la programmation de ces circuits mais aussi de simuler les signaux résultant de cette conception, ainsi que la génération des modèles des circuits par RTL de ces systèmes.

Les problèmes rencontrés dans notre simulation s'articulent sur les erreurs syntaxiques qu'on doit tenir compte par compilation en respectant la logique de conception de ces circuits

Conclusion Générale

Conclusion Générale

L'objectif de ce travail consistait à la programmation des circuits programmables tels que FPGA qui nous permet de basculer vers l'exécution en temps réels des systèmes.

nous avons étudié FPGA du type cyclone II de Altéra, on a utilisé le langage de description de matériel VHDL qu'on a développé au Chapitre III en présentant ses avantages et ses fonctionnalités, on a exposé que c'est un langage complet destiné à représenter le comportement ainsi que l'architecture des systèmes numériques.

On a étudié aussi les circuits logiques programmables FPGAs, on a découvert les avantages que présentent de point de vue flexibilité, configuration et reprogrammation à volonté qui permet la modification et l'amélioration des conceptions réalisées. Ensuite, on a opéré en VHDL de quelques fonctions des éléments : AND, OR, Registre, compteur, bascule en utilisant le logiciel Quartus II pour leurs implémentations sur FPGA. Les résultats obtenus nous ont permis de prouver le bon fonctionnement des circuits logiques programmable FPGAs, et de saisir l'importance de leurs utilisations, suite aux traitements parallèles des instructions qui conçoivent le système conçu par l'utilisateur.

Comme perspective et travaux futurs, on propose de réaliser d'autres implémentations des fonctionnalités d'autres systèmes numériques sur FPGA.

Références

[1]:LAMERES, BROCK J « Introduction to logic circuits and logic design with VHDL». SPRINGER NATURE (2019).

[2]: Richard S Sandige_ Michael L Sandige - Fundamentals of digital and computer design with VHDL-McGraw Hill (2012).

[3] Adrien Blanchardon: « Synthèse d'architectures de circuits FPGA tolérants aux défauts ». Thèse, Pierre and Marie Curie Université, Paris, France, 2015

[4] :Brown S. Vranesic Z. - Fundamentals of Digital Logic with VHDL-McGraw-Hill (2004).

[5]: Volnei A. Pedroni - Circuit Design with VHDL-The MIT Press (2004).

[6] : Charles H. Roth Jr, Lizy K. John - Digital Systems Design Using VHDL-Cengage Learning (2007).

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

[8] : Djamal Gozim, Kamel Guesmi. LOGIQUE COMBINATOIRE ET SEQUENTIELLE. Licence. Algérie. 2019. hal-02927680

[9] : COURS EN LIGNE : « LES HDL » **http://hdl.telecom-paristech.fr/.**

[10] : Altera DE1 Board. User manual , www.terasic.com

[11] : Gabriel Cormier, cours en ligne, « Logique combinatoire ». GELE2442, Chapitre 5, Université de Moncton, CANADA, http://www8.umoncton.ca/umcm-cormier_gabriel/Circuitslogiques.html

[12] : Gabriel Cormier, cours en ligne « Circuits séquentiels». GELE2442, Chapitre 6, Université de Moncton, CANADA, http://www8.umoncton.ca/umcm-cormier_gabriel/Circuitslogiques.html

[13] : Gabriel Cormier, cours en ligne « Registres et compteurs». GELE2442, Chapitre 7, Université de Moncton, CANADA, http://www8.umoncton.ca/umcm-cormier_gabriel/Circuitslogiques.html

[14] : Gabriel Cormier, cours en ligne « Mémoire et logique programmable». GELE2442, Chapitre 8, Université de Moncton, CANADA, http://www8.umoncton.ca/umcmcormier_gabriel/Circuitslogiques.html