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

Centre universitaire Nour Bachir El Bayadh
Institut des Sciences

المركز الجامعي نور البشير البيض
معهد العلوم

Extrait Procès-verbal de la réunion du Conseil Scientifique de l'Institut des Sciences du 11 Octobre 2021

Réf : PV N°.02/CSI/2021

POLYCOPIES EXPERTISES

Le CSI a pris note de l'avis favorable de la part des experts désignés par ce même CSI (voir Procès-verbal du CSI du 09/03/2021), pour l'expertise du polycopié de cours déposé par les Dr BERBER Mohamed, Dr GUETTAF Yacine et Dr BENDELHOUM Mohammed Sofiane.

Intitulé du manuscrit : " *FPGA et VHDL* ".
Nombre de page du manuscrit : *143 pages*.

Le Président du CSI
Pr. Hamid Azzedine

رئيس المجلس العلمي
لمعهد العلوم
أ.د. حميد عزالدين

Le Directeur de l'Institut
Dr ALAMI Omar

الدكتور: علامي عمر
مدير معهد العلوم

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Centre Universitaire Nour Bachir El Bayadh
Institut des Sciences
Département de Technologie

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Polycopié

Cours UEF 1.1.1 intitulé :

INTITULE DU MODULE :

ELECTRONIQUE NUMERIQUE AVANCEE : FPGA ET VHDL

Dr. BERBER Mohamed
Maître de Conférences Classe « A »
Centre Universitaire Nour Bachir – El Bayadh

Dr. GUETTAF Yacine
Maître de Conférences Classe « A »
Centre Universitaire Nour Bachir – El Bayadh

Dr. BENDELHOUM Mohamed Sofiane
Maître de Conférences Classe « B »
Centre Universitaire Nour Bachir – El Bayadh

2021-2022

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Liste des figures :

المجلس العلمي

المركز الجامعي نور البشير بالبيض
المجلس العلمي

Liste des tableaux :

المركز الجامعي نور البشير بالبيض
المجلس العلمي

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Introduction :

Les conceptions électroniques numériques continuent d'évoluer vers des composants plus complexes et à plus grand nombre de broches fonctionnant à des fréquences d'horloge plus élevées, cela rend considérablement plus difficile la conception des cartes de prototypage et débogage dans un laboratoire avec un analyseur logique et un oscilloscope. Cela est dû au fait que les signaux sont de plus en plus difficiles à sonder physiquement et parce que leur sondage est plus susceptible de modifier le fonctionnement du circuit.

Aujourd'hui une grande partie de l'électronique numérique (logique combinatoire et séquentiel), personnalisée est conçue dans des ASIC (Application Specific Integrated Circuit, *Circuit intégré spécifique à l'application*) ou des FPGA (Field Programmable Gate Array, *Réseau de portes programmables*) avec les dispositifs VHDL (Hardware Description Language, *langage de description matérielle*) et FPGA permettent aux concepteurs de développer et de simuler rapidement un circuit numérique sophistiqué, de le réaliser sur un dispositif de prototypage et de vérifier le fonctionnement de l'implémentation physique. Au fur et à mesure que ces technologies mûrissent, elles sont devenues une pratique courante. Nous pouvons désormais utiliser un PC et une carte de prototypage FPGA peu coûteuse pour construire un système numérique complexe et sophistiqué

Ce polycopié est destiné aux étudiants de la première année master électronique (système embarqué) les étudiants auront à étudier les différents types de circuits programmables, ainsi que les différentes méthodes de conception en particulier la programmation en utilisant les langages de description matérielle.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

المركز الجامعي نور البشير بالبيض
المجلس العلمي

1.1 Introduction

Il y a quelques années la réalisation d'un montage en électronique numérique impliquait l'utilisation d'un nombre important de circuits intégrés logiques. Ceci avait pour conséquences un prix de revient élevé, une mise en œuvre complexe et un circuit imprimé de taille importante. Le développement des mémoires utilisées en informatique fut à l'origine des premiers circuits logiques programmables (PLD : Programmable Logic Device). Ces structures (logique programme) ont besoin de s'interfacer entre elles. Elles utilisent généralement pour réaliser ces interfaces des fonctions à base de fonctions logiques élémentaires, compteurs, registres. Le nombre de circuits nécessaires pour remplir ces fonctions peut devenir très vite important. Les fonctions logiques programmables sont des circuits disposants des entrées et des sorties dont l'utilisateur peut programmer le schéma logique d'après les besoins liées à la fonction souhaitée : Logique combinatoire et/ou séquentielle.

Ces composants sont appelés des PLD ce type de produit peut intégrer dans un seul circuit plusieurs fonctions logiques programmables par l'utilisateur. Les PLD sont utilisés pour remplacer l'association de plusieurs boîtiers logiques. Le câblage est simplifié, l'encombrement et le risque de pannes est réduit. Certains PLD ne permettent pas la relecture de la fonction logique programmée, c'est pratique lorsque le programme doit rester confidentiel. Ces circuits disposent d'un certain nombre de broches d'entrées et de sorties. L'utilisateur associe ces broches aux équations logiques (plus ou moins complexes) qu'il programme dans le circuit. Sa mise en œuvre se fait très facilement à l'aide d'un programmateur, d'un micro-ordinateur et d'un logiciel adapté. Rassemblés sous le terme générique PLD, les circuits programmables par l'utilisateur se décomposent en deux familles :

1. les PROM, les PLA, les PAL et les EPLD,
2. les FPGA.

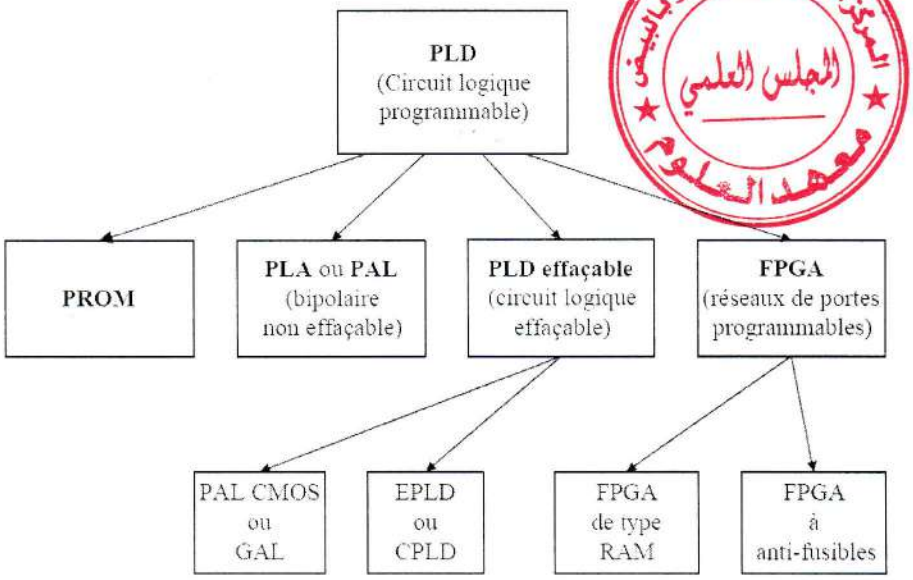


Figure 1. Structure des réseaux logiques combinatoires

Structure des réseaux logiques combinatoires :

1.2 Structure de base d'un PLD :

La plupart des PLDs suivent la structure suivante :

- Un ensemble d'opérateurs « ET » sur lesquels viennent se connecter les variables d'entrée et leurs compléments.
- Un ensemble d'opérateurs « OU » sur lesquels les sorties des opérateurs « ET » sont connectées.
- Une éventuelle structure de sortie (Portes in verseuses, logique 3 états, registres...). Les deux premiers ensembles forment chacun ce qu'on appelle une matrice les interconnexions de ces matrices doivent être programmables. C'est la raison pour laquelle elles sont assurées par des fusibles qui sont « grillés » lors de la programmation. Lorsqu'un PLD est vierge toutes les connexions sont assurées.

Figure 2. Symbolisation des portes logiques pour les PLD

1.3 Convention de notation :

Afin de présenter des schémas clairs et précis, il est utile d'adopter une convention concernant les connexions à fusibles. Les deux figures suivantes représentent la fonction 3 entrées. La figure b) n'est qu'une version simplifiée du schéma de la figure a).

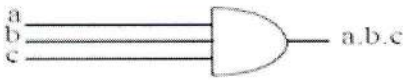


Figure 3. Symbole d'une porte AND à 3 entrées

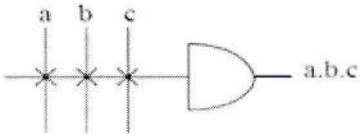


Figure 4. Symbole simplifié d'une porte AND

Un exemple de notation est donné sur la figure ci-contre. La fonction réalisée est S = (a. c) + (b. d). Une croix, à une intersection, indique la présence d'une connexion à fusible non claqué. L'absence de croix signifie que le fusible est claqué. La liaison entre la ligne horizontale et verticale est rompue. La sortie S réalise une fonction OU des 2 termes produits (a.c) et (b.d).

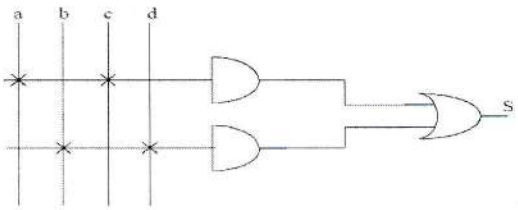
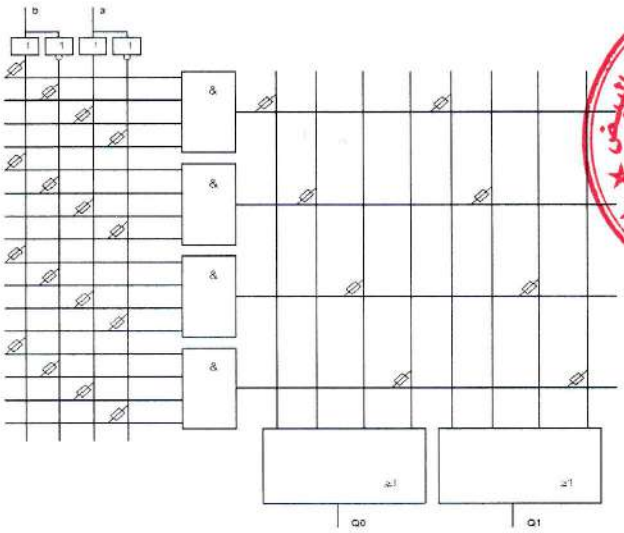


Figure 5. La sortie S réalise une fonction OU avec deux fonction AND.

1.4 Représentation de l'architecture interne d'un PLD :

Un exemple de ce type de structure est présenté par la figure ci-dessous. On peut remarquer que la représentation d'une telle structure est complexe alors que le nombre de portes intégrées est peu important. Les constructeurs ont donc très rapidement adoptés un autre type de représentation rendant les schémas beaucoup plus lisibles. On remarquera que la norme adoptée est américaine. Un exemple de cette représentation est donné par la figure suivante :



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 6. Structure de base d'un PLD

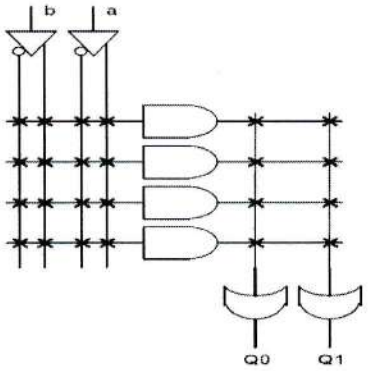


Figure 7. Structure de base avec les normes des constructeurs

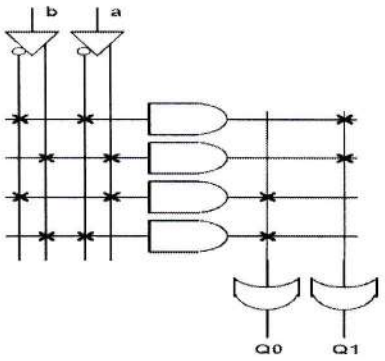
La figure 7 représente la structure interne d'un PLD ayant ses fusibles intacts. Les équations logiques de Q_0 et Q_1 sont :

$$Q_0 = Q_1 = \bar{a}.\bar{b}.a.b + \bar{a}.\bar{b}.a.b + \bar{a}.\bar{b}.a.b + \bar{a}.\bar{b}.a.b = 0.$$

Si on veut obtenir les fonctions suivantes :

$$Q_1 = \bar{a}.\bar{b} + a.b \quad et \quad Q_2 = a.\bar{b} + \bar{a}.b$$

On « grillera » des fusibles de façon à obtenir le schéma suivant de la figure 8 :



x : *Fusible intact*

Figure 8. Structure après programmation

Les premiers circuits programmables apparus sur le marché sont les PROM bipolaires à fusibles. Cette mémoire est l'association d'un réseau de ET fixes, réalisant le décodage d'adresse, et d'un réseau de OU programmables, réalisant le plan mémoire proprement dit. On peut facilement comprendre que, outre le stockage de données qui est sa fonction première, cette mémoire puisse être utilisée en tant que circuit logique. La figure ci-dessous représente la structure logique d'une PROM bipolaire à fusibles.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

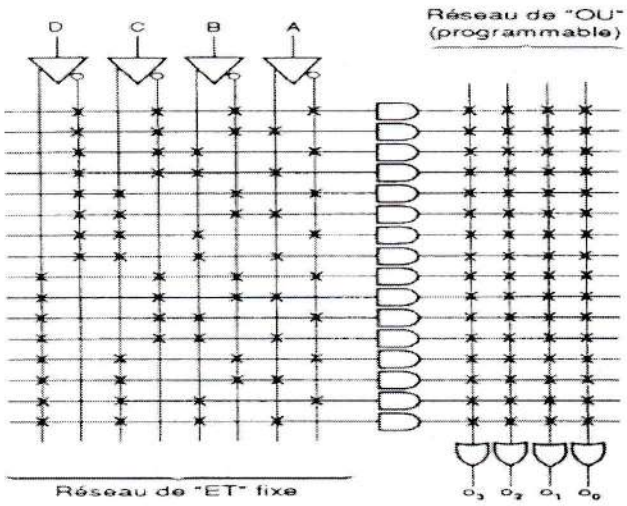


Figure 9. Structure logique d'une PROM bipolaire à fusibles

Chaque sortie O_i peut réaliser une fonction OU de 16 termes produits de certaines combinaisons des 4 variables A, B, C et D. Avec les PROM, les fonctions logiques programmées sont spécifiées par les tables de vérités. Le temps de propagation est indépendant de la fonction implantée.

المركز الجامعي نور البشير بالبيض
المجلس العلمي

1.5 Les différentes familles de PLD :

Il existe plusieurs familles de PLD qui sont différenciées par leur structure interne. Le tableau suivant présente certaines de ces familles.

Type	Nombres de portes intégrées	Matrice ET	Matrice OU	Effaçable
PROM	2000 à 500000	Fixe	Programmable	Non
PAL	10 à 100	Programmable	Fixe	Non
PLA	10 à 100	Programmable	Programmable	Non
GAL	10 à 100	Programmable	Fixe	Electriquement
EPLD	100 à 3000	Programmable	Fixe	Aux UV
FPGA	2000 à 3000	Programmable	Programmable	Electriquement

Table 1. Differentes familles PLD

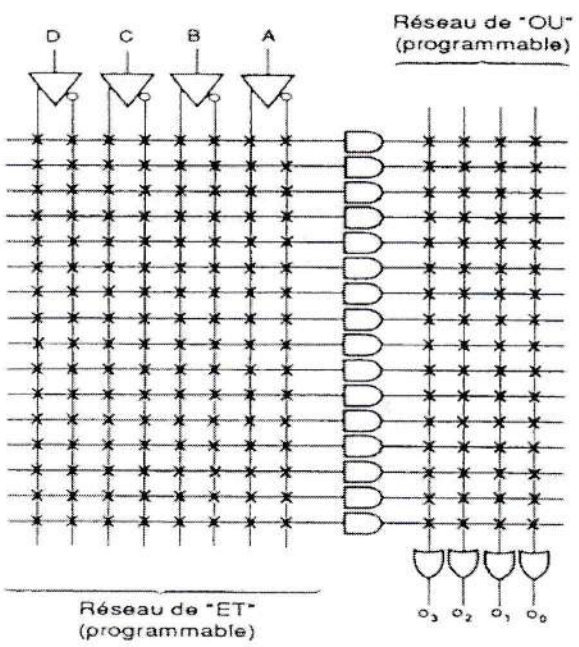
1.6 PROM

Certaines de ces familles possèdent en plus des matrices « ET » et « OU », de la logique séquentielle (Bascules « D », « JK »...) placée après les entrées ou avant les sorties du PLD. Les « PROMs » sont des circuits utilisés en informatique pour mémoriser de façon définitive des données : ce sont des « Mémoires mortes ». Il existe des versions effaçables comme les UVPROM (aux U-V) et les EEPROM (électriquement).

1.7 Les PLA

L'un des premiers PLD commerciaux mis au point à l'aide de la technologie moderne des circuits intégrés était le réseau logique programmable (PLA). En 1970, Texas Instrument a introduit le PLA avec une architecture prenant en charge la mise en œuvre d'expressions logiques arbitraires, somme de produits. Le PLA a été fabriqué avec un réseau dense de portes ET, appelé un plan ET, et un réseau dense de portes OU, appelé un plan OR. Les entrées du PLA avaient chacune un inverseur afin de fournir la variable d'origine et son complément. Des expressions logiques SOP (sum of products, *somme de produit*) arbitraires pourraient être implémentées en créant des connexions entre les entrées, le plan ET et le plan OU. Les PLA d'origine ont été fabriqués avec toutes les caractéristiques nécessaires, à l'exception des connexions finales pour implémenter les fonctions SOP. Lorsqu'un client a fourni l'expression SOP souhaitée, les connexions ont été ajoutées comme étape finale de la fabrication. Cette technique de configuration était similaire à une approche MROM. Un schéma plus compact pour le PLA est dessiné en représentant toutes les entrées dans le ET et OU portes avec un seul fil. Les connexions sont indiquées en insérant des X aux intersections des fils.

La Figure 10 montre ce schéma PLA simplifié mettant en œuvre deux expressions logiques SOP différentes. Le concept du PLA (Programmable Logic Array) a été développé il y a plus de 20 ans. Il reprend la technique des fusibles des PROM bipolaires. La programmation consiste à faire sauter les fusibles pour réaliser la fonction logique de son choix. La structure des PLA est une évolution des PROM bipolaires. Elle est constituée d'un réseau de ET programmables et d'un réseau de OU programmables. Sa structure logique est la suivante :



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 10. Structure logique d'un PLA

Chaque sortie O_i peut réaliser une fonction OU de 16 termes produits des 4 variables A, B, C et D. Avec cette structure, on peut implémenter n'importe quelle fonction logique combinatoire. Ces circuits sont évidemment très souples d'emploi, mais ils sont plus difficiles à utiliser que les PROM. Statistiquement, il s'avère inutile d'avoir autant de possibilité de programmation, d'autant que les fusibles prennent beaucoup de place sur le silicium. Ce type de circuit n'a pas réussi à pénétrer le marché des circuits programmables. La demande s'est plutôt orientée vers les circuits PAL.

1.8 Les PAL :

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

L'un des inconvénients du PLA original était que la programmabilité du plan OU provoquait des retards de propagation importants à travers les circuits logique combinatoire. Afin d'améliorer les performances des PLA, la logique de matrice programmable (PAL) a été introduite en 1978 par la société Monolithic Memories, Inc. Le PAL contenait un plan ET programmable et un plan OU fixe. Le plan OR fixe a amélioré les performances de cette architecture programmable. Bien que le fait de ne pas avoir de plan OU programmable réduise la flexibilité de l'appareil, la plupart des expressions SOP pourraient être manipulées pour fonctionner avec un PAL. Une autre contribution du PAL était que le plan AND pouvait être programmé à l'aide de fusibles. Au départ, toutes les connexions étaient présentes dans le plan AND. Un programmeur externe a été utilisé pour faire sauter les fusibles afin de déconnecter les entrées des portes ET. Alors que l'approche par fusible fournissait une programmation unique, la possibilité de configurer la post-fabrication logique était une avancée significative par rapport au PLA, qui devait être programmé chez le fabricant.

Les PALs sont les circuits logiques programmables les plus anciens à être utilisés pour réaliser des fonctions logiques ». Un composant logique programmable PAL est basé sur le concept qu'il est possible de remmener toute équation logique en une somme de produits. La programmation s'effectue par destruction de fusible (un fusible détruit équivaut à un circuit ouvert), ils ne sont donc programmables qu'une fois, ce qui peut être gênant en phase de développement. Un PAL permet de remplacer jusqu'à 10 boîtiers SSI ou 2 à 3 boîtiers MSI.

1.8.1 Principe d'un PAL :

Ce PAL simplifié comporte 2 entrées I_1 et I_2 et une sortie O. Huit fusibles (F_1 à F_8) permettent de réaliser diverses fonctions logiques. La programmation va consister à faire sauter les fusibles nécessaires afin de réaliser la fonction voulue. La programmation va constituer à détruire les fusibles pour obtenir les fonctions désirées, en sachant que lors de l'achat d'un P.A.L. tous les fusibles sont vierges ou pas détruits.

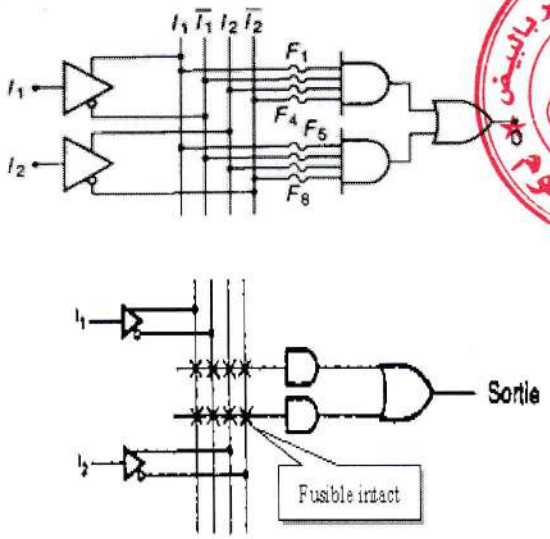


Figure 12. Symbole simplifié d'un PAL

1.8.2 Convention de représentation :

La représentation simplifiée ne montre pas tous les fusibles, les entrées de la porte ET sont regroupées sur une seule ligne. Une croix représente un fusible intact.

Exemple de programmation d'un PAL :

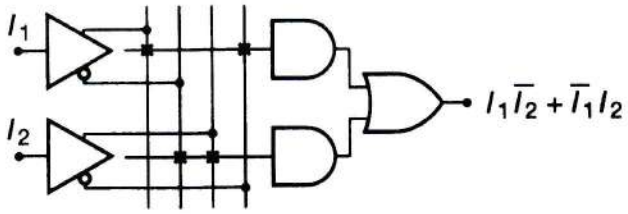


Figure 13. Exemple de programmation d'un PAL

On souhaite réaliser une fonction OU EXCLUSIF : $O = I_1 \oplus I_2 = I_1\overline{I_2} + \overline{I_1}I_2$ La fusion des fusibles est obtenue en appliquant à leurs bornes une tension de 11,5 V pendant 10 à 50 µS (leur tension de fonctionnement est environ de 5V). Cette opération est bien sûr effectuée en utilisant un programmateur adapté. La structure de base de ce PLD est présentée par le schéma suivant.

- Ils possèdent des matrices ET programmables, et des matrices OU fixes.

Figure 11. Structure simplifié d'un PAL

- La fusion des fusibles est obtenue en appliquant à leurs bornes une tension de 11.5V pendant 10 à 50 µS (leur tension de fonctionnement est de 5V).
- Cette opération est sûre effectuée en utilisant un programmateur adapté.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

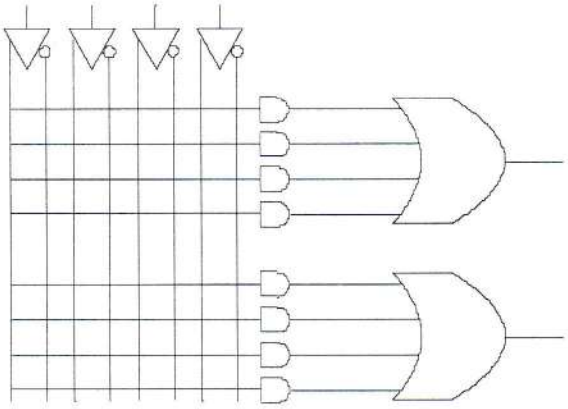


Figure 14. Structure de base d'un PAL

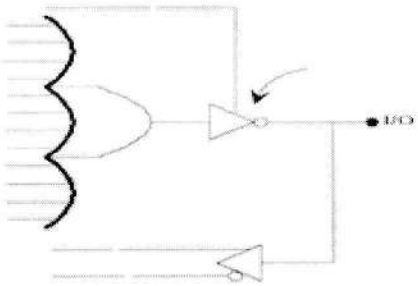


Figure 15. Porte à sortie 3 états

Porte à sortie 3 états, permettant de déconnecter la broche de la matrice "ET" (rendre indépendant) la sortie I/O de l'état logique imposé par la sortie du OU. Dans ce cas la sortie I/O est utilisée en entrée

Certaines broches de ces circuits peuvent être utilisées aussi bien en entrée qu'en sortie grâce à un système de logiques 3 états. La commande de cette dernière est configurée au

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

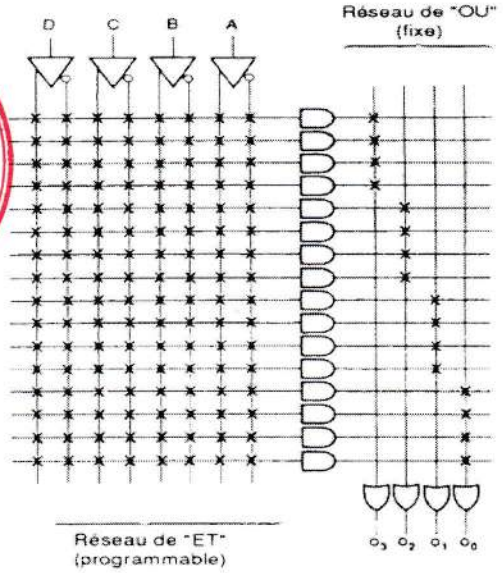


Figure 16. Structure logique d'un PAL

moment de la programmation. La structure de sortie permet aussi de réinjecter les sorties en entrée (Feed-back). Selon le type de PAL la structure de sortie peut être constituée d'une porte « NON», d'une porte « OU » Exclusive, d'une bascule « D » ou d'une combinaison des trois. Le nombre d'entrées et de sorties est lui aussi lié à la référence du PAL. La technologie employée est la même que pour les PLA. La figure qui suit représente la structure logique d'un PAL où chaque sortie intègre 4 termes produits de 4 variables. L'architecture du PAL a été conçue à partir d'observations indiquant qu'une grande partie des fonctions logiques ne requiert que quelques termes produits par sortie. L'avantage de cette architecture est l'augmentation de la vitesse par rapport aux PLA. En effet, comme le nombre de connexions est diminué, la longueur des lignes d'interconnexion est réduite. Le temps de propagation entre une entrée et une sortie est par conséquent réduit.

Le PAL possède toujours des entrées simples sur le réseau de ET programmables, mais aussi des broches spéciales qui peuvent être programmées :

- en entrée simple en faisant passer le buffer de sortie trois états en haute impédance,
- en sortie réinjectée sur le réseau de ET. Cela permet d'augmenter le nombre de termes produits disponibles sur les autres sorties.

1.8.3 Les différentes structures :

Structure générale :

Tout P.A.L. est constitué :

- D'entrées (Input) : I_1 à I_n avec 8<n<20.

- De sorties (Output) Ou d'entrées / sorties (I/O) de type Totem Pôle ou Trois Etats : O1 à On ou IO_1 à IO_n (2<n<15).

On peut trouver aussi :

- Une entrée d'horloge (Clock) : Clk ou Clock.

- Une entrée de validation des sorties trois états : OE (Output Enable) ou Enable.

- Une entrée de remise à zéro des registres : RESET.

D'un point de vue fonctionnel un P.A.L. est constitué d'une zone d'entrée de fusibles ou matrice de programmation et une structure de sortie non programmable déterminant le type de circuit voir schéma ci-dessous.

Structure et symbolisation normalisée :

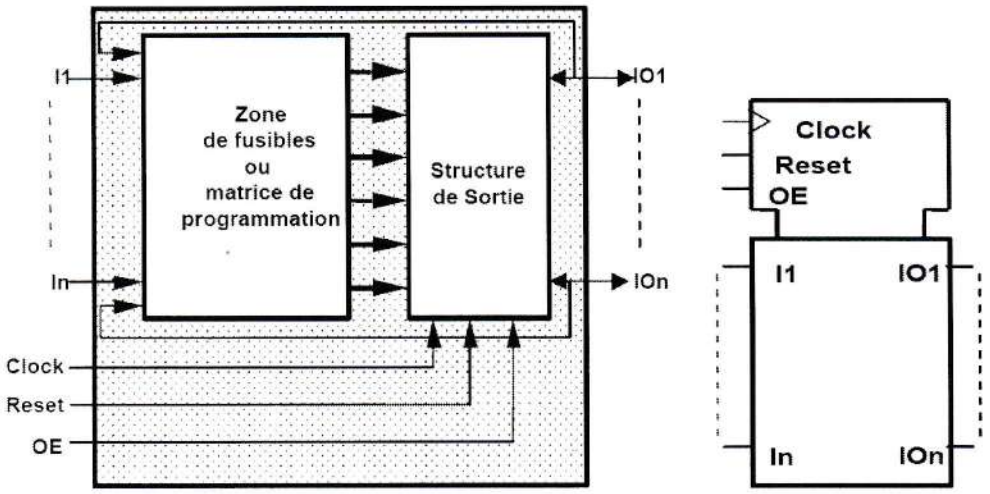


Figure 17. Schema synoptique d'un PAL

Remarque : Sur un schéma comportant un PAL, on doit écrire les équations qui relient les entrées aux sorties ou le nom du document contenant les équations du P.A.L.

La programmation de ces circuits s'effectue par destruction de fusibles. Une fois programmée on ne peut plus les effacer. On distingue deux sous familles :

- Les P.A.L. combinatoires ou P.A.L. simples. Ils sont constitués de fonctions de logique combinatoire.

- Les P.A.L. à registres ou F.P.L.S. Field Programmable Logic Séquencer pour séquenceur logique programmable. Ils sont constitués de logique combinatoire et séquentielle (Registre).

Il existe un grand nombre de P.A.L. utilisant des structures de sorties différentes. On peut distinguer trois types de structures de base :

- Combinatoire.
- Séquentielle.
- Versatile.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

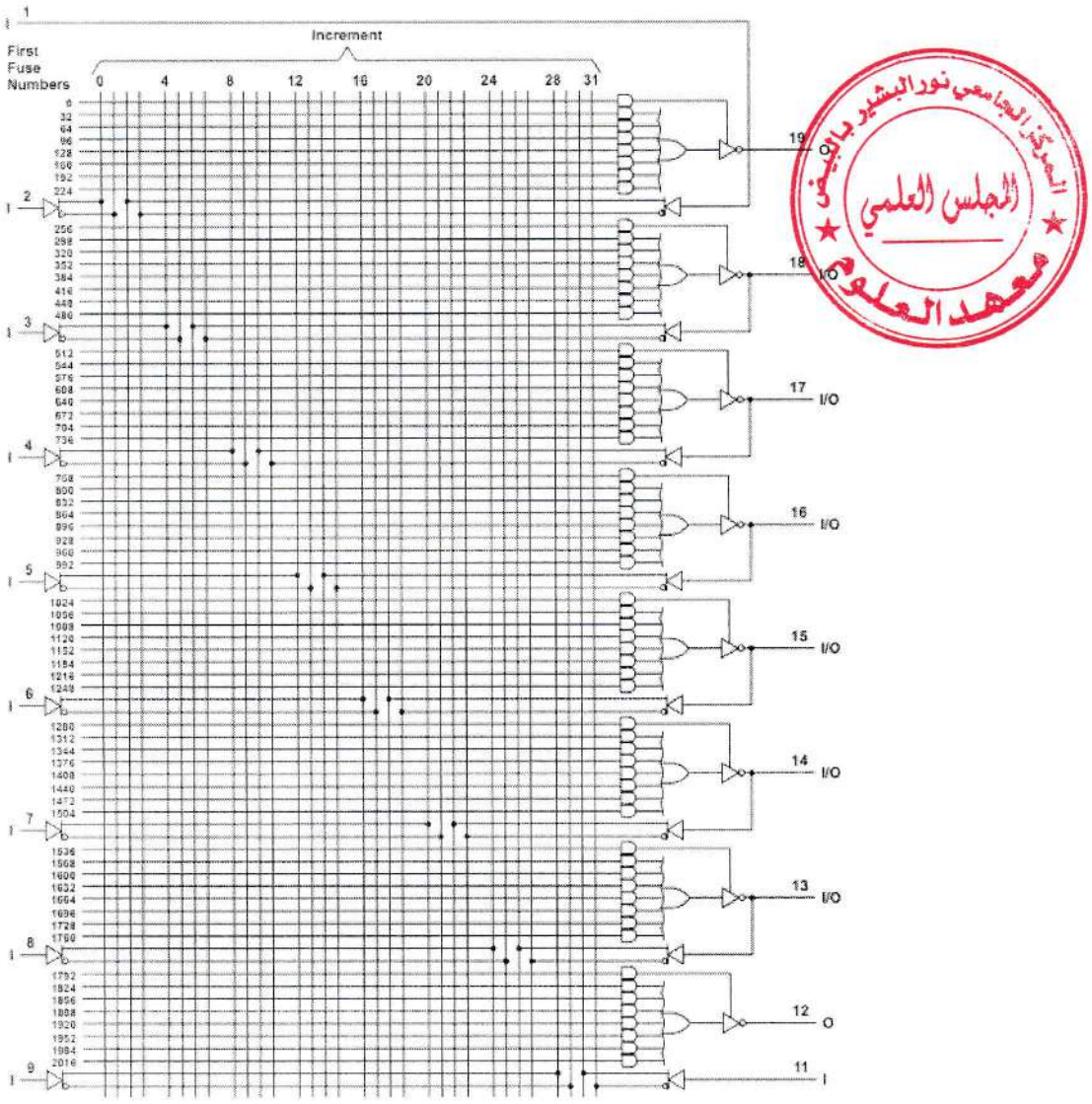
1.8.4 Les différents types d'entrées /sorties :

On distingue 3 principes utilisés pour les sorties. Selon le modèle, un ou plusieurs types de sorties peuvent être utilisés sur un même PAL.

1.8.4.1 Entrées/Sorties combinatoires :

Ces sorties 3 états sont rebouclées vers la matrice de fusibles. Une sortie peut donc servir de variable intermédiaire. En mode haute impédance (la sortie étant inhibée), on peut utiliser une broche de sortie comme étant une entrée. On parle alors d'entrée / sortie (I/O). Il existe trois types :

- H : (High) Porte ET suivit d'une Porte OU. Sortie active à l'état haut.
- L : (Low) Porte ET suivit d'une Porte NON OU. Sortie active à l'état bas.



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 18. PAL16L8

- C : (Combinée) programmable en type H ou L.

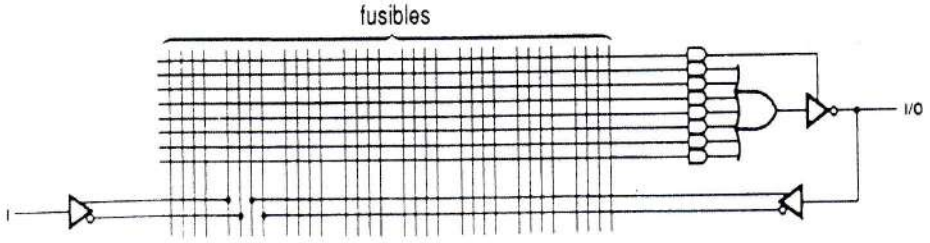


Figure 19. PAL combiné

1.8.4.2 Séquentielle :

Les architectures des PAL ont évolué vers les PAL à registres. Dans ces PAL, la sortie du réseau de fusibles aboutit sur l'entrée d'une bascule D. Ces sorties utilisent une bascule D qui permet la logique séquentielle. Par contre, une sortie à registre ne peut pas être utilisée comme entrée. La sortie Q peut aller vers une sortie, la sortie $\overline{Q}$ étant réinjectée sur le réseau via un inverseur/non inverseur.

Il existe trois types :

1.8.4.2.1 Sorties à registres PAL de type R

Ces circuits sont composés de bascule D. Les sorties des bascules sont de type trois états contrôlées par un signal de validation Enable ou OE, et une horloge est commune à toutes les bascules (clock).

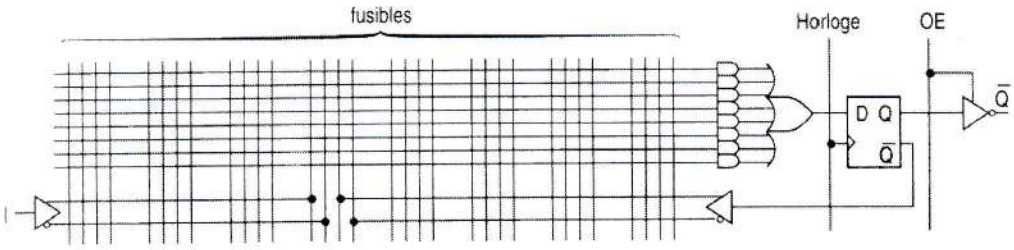


Figure 20. PAL type R

المركز الجامعي البيض
المجلس العلمي
معهد العلوم

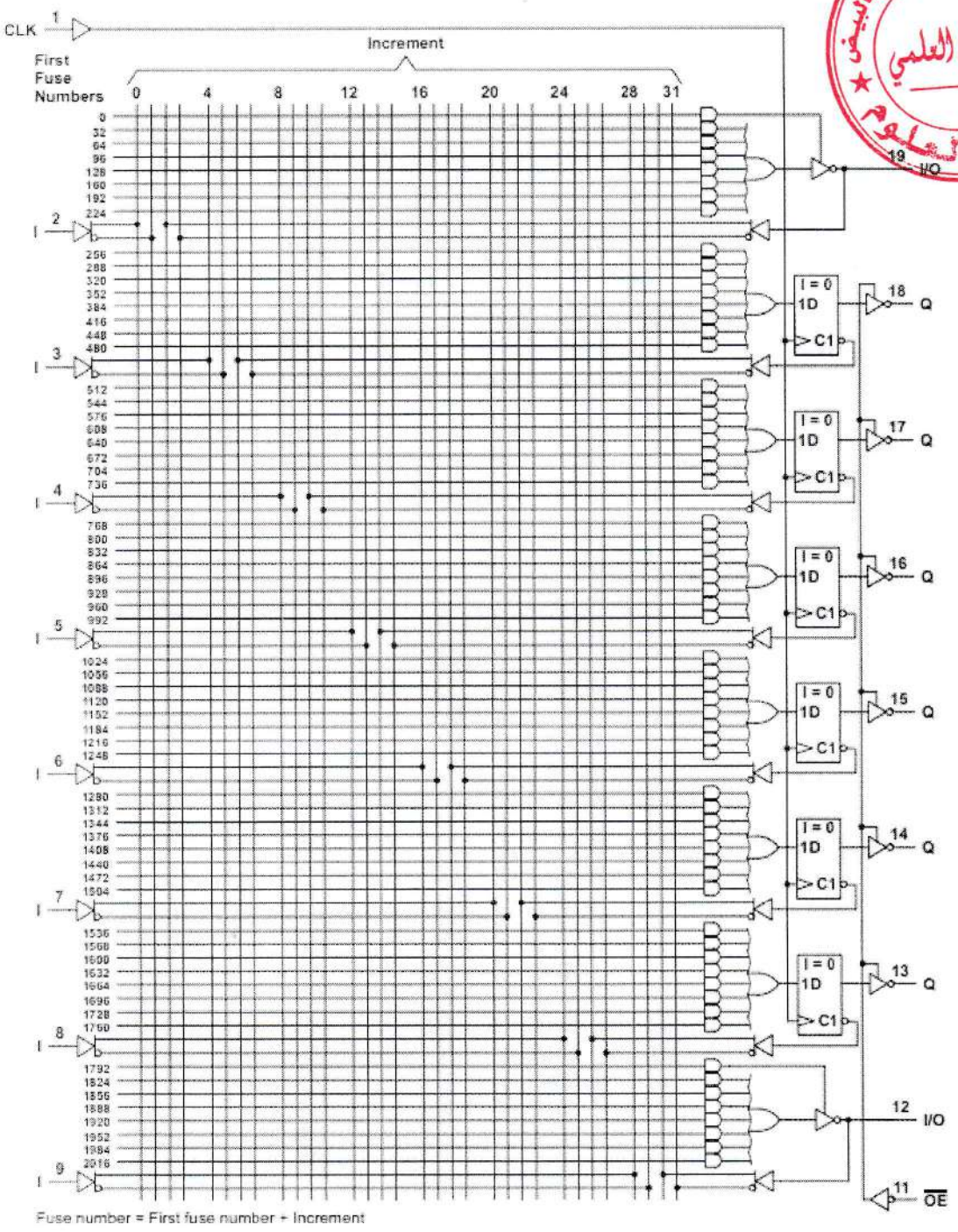


Figure 21.PAL16R6

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

1.8.4.2.2 Sorties à Ou Exclusif et Registre PAL de type X

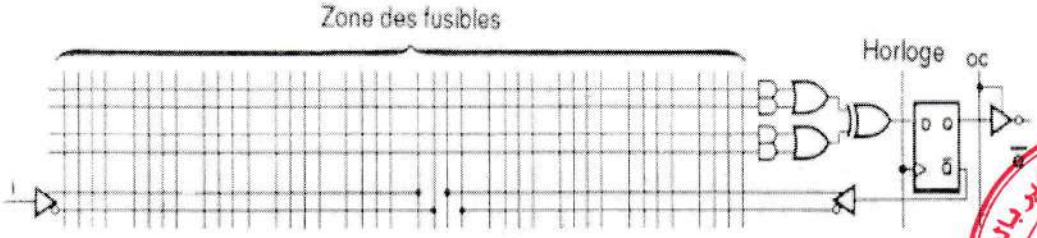


Figure 22. PAL type X

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

1.8.4.2.3 Sorties à Registre asynchrone PAL de type RA

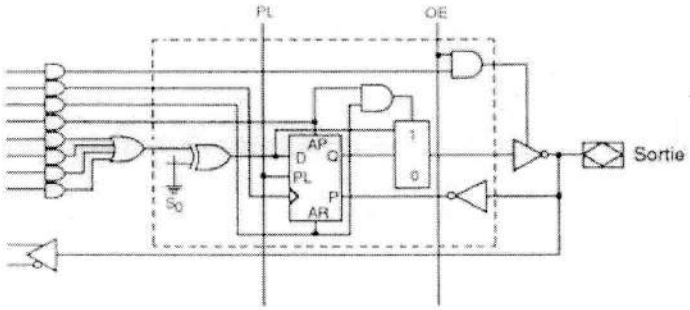


Figure 23. PAL type RA

Les structures de sorties sont beaucoup plus évoluées par rapport aux autres P.A.L., elles se rapprochent des P.A.L. de type versatile.

Elles peuvent prendre quatre configurations suivant les valeurs de AP et AR.

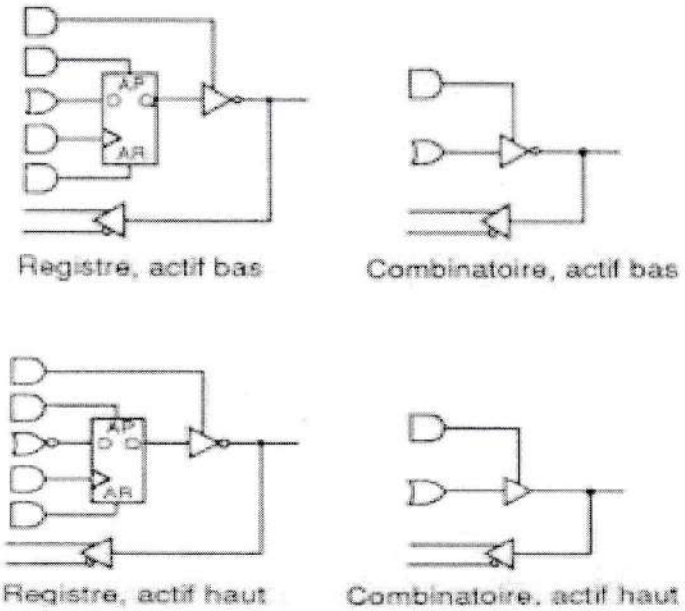


Figure 24. Différentes configurations

Avec cette structure, la sortie ne peut être utilisée comme entrée sur le réseau. L'exemple d'un PAL à registres 16R8 est donné. Il implémente 8 termes produits de 16 variables par sortie.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد الرياضيات

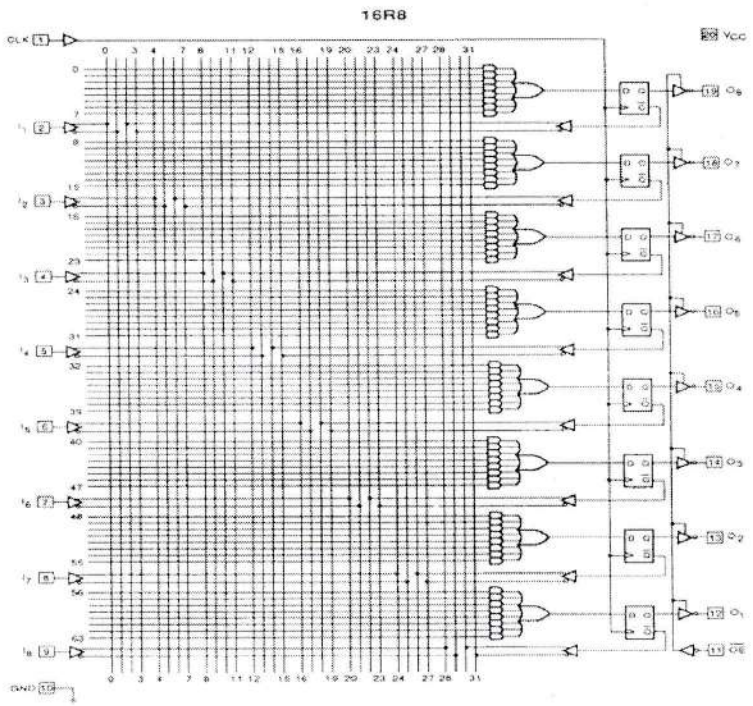


Figure 25. PAL à registre 16R8

D'après la notation employée par les fabricants, la référence 16R8 signifie :

- 16 : Nombre d'entrées au niveau du réseau de ET.
- R : PAL à registres.
- 8 : Nombre de sorties.

Les plus gros PAL standards sont les 20R8 et 20L8.

1.8.4.3 Sorties versatiles PAL de type V

MODE D'EMPLOI

Le PALCE16V8 est un appareil PAL universel. Il en a huit macrocellules configurables indépendamment (MC_0-MC_7). Chaque macrocellule peut être configurée comme sortie enregistrée, sortie combinatoire, E / S combinatoires ou entrée dédiée. La matrice de programmation implémente un tableau logique ET programmable, qui pilote un tableau logique OU fixe. Les tampons pour les entrées de périphérique ont des sorties complémentaires pour fournir une polarité de signal d'entrée programmable par l'utilisateur. Les broches 1 et 11 servent respectivement d'entrées de tableau ou

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

d'activation d'horloge (CLK) et de sortie (ÔË) pour toutes les bascules. Les broches d'entrée inutilisées doivent être directement liées à Vcc ou GND. Les termes de produit avec tous les bits non programmés (déconnectés) supposent l'état HAUT logique et les termes de produit avec à la fois vrai et complément de tout signal d'entrée connecté prennent un état logique BAS. Les fonctions programmables du PALCE16V8 sont automatiquement configurées à partir des spécifications de conception de l'utilisateur, qui peuvent être dans un certain nombre de formats. La spécification de conception est traitée par le logiciel de développement pour vérifier la conception et créer un fichier de programmation. Ce fichier, une fois téléchargé sur un programmeur, configure l'appareil en fonction de la fonction souhaitée par l'utilisateur. L'utilisateur dispose de deux options de conception avec le PALCE16V8. Premièrement, il peut être programmé comme un périphérique PAL standard des séries PAL16R8 et PAL10H8. Le fabricant du programmateur PAL fournira les codes de périphérique pour les architectures de périphérique PAL standard à utiliser avec le PALCE16V8. Le programmeur programmera le PALCE16V8 dans l'architecture correspondante. Cela permet à l'utilisateur d'utiliser les fichiers JEDEC de périphérique PAL standard existants sans y apporter de modifications. Alternativement, l'appareil peut être programmé comme PALCE16V8, ici l'utilisateur doit utiliser le PALCE16V8 code de l'appareil. Cette option permet d'utiliser pleinement la macrocellule.

Options de configuration

Chaque macrocellule peut être configurée comme l'une des suivantes : sortie enregistrée, sortie combinatoire, E / S combinatoire ou entrée dédiée. Dans la configuration de sortie enregistrée, le tampon de sortie est activé par la broche OE. Dans la configuration combinatoire, le tampon est soit contrôlé par un terme de produit, soit toujours activé. Dans la configuration d'entrée dédiée, il est toujours désactivé. Avec à l'exception de MC_0 et MC_7, une macrocellule configurée comme une entrée dédiée dérive le signal d'entrée d'une E / S adjacente. MC_0 dérive son entrée de la broche 11 (OE) et MC7 de la broche 1 (CLK). Les configurations des macrocellules sont contrôlées par le mot de contrôle de configuration. Il contient 2 bits globaux (SGO et SG1) et 16 bits locaux (SLOo à SLO7 et SL1o à SU7). SGO détermine si les registres seront autorisés. SG1 détermine si le PALCE16V8 émulera une famille PALI6R8 ou un périphérique de la famille PALI0H8. Dans chaque macrocellule, SLOx, en conjonction avec SG1, sélectionne la configuration de la macrocellule, et SL1x définit la sortie comme active basse ou active haute pour la macrocellule individuelle. Les bits de configuration fonctionnent

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد

en agissant comme des entrées de commande pour les multiplexeurs de la macrocellule. Il existe quatre multiplexeurs : une entrée de terme de produit, une sélection d'activation, une sélection de sortie et un multiplexeur de sélection de rétroaction. SG1 et SLOx sont les signaux de commande des quatre multiplexeurs. Dans MC_0 et MC_7, SGO remplace SG1 sur le multiplexeur de rétroaction. Cela permet à CLK d'être la broche adjacente pour MC7 et OE la broche adjacente pour MC_0.

Configuration de sortie enregistrée

Les paramètres du bit de contrôle sont SGO = 0, SG1 = 1 et SLOx = 0. Il n'y a qu'une seule configuration enregistrée. Les huit termes du produit sont disponibles en tant qu'entrées de la porte OU. La polarité des données est déterminée par SL1x. La bascule est chargée sur la transition LOW-to-HIGH de CLK. Le chemin de rétroaction provient de Q sur le registre. Le tampon de sortie est activé par OË. Configurations combinatoires Le PALCE16V8 a trois configurations de sortie combinatoires : sortie dédiée dans un périphérique non enregistré, E / S dans un périphérique non enregistré et E / S dans un périphérique enregistré appareil.

Sortie dédiée dans un périphérique non enregistré

Les paramètres du bit de contrôle sont SGO = 1, SG1 = 0 et SLOx = 0. Les huit conditions de produit sont disponibles pour la porte OU. Bien que la macrocellule soit une sortie dédiée, la rétroaction est utilisée, à l'exception des broches 15 et 16. Les broches 15 et 16 n'utilisent pas de rétroaction dans ce mode. Comme CLK et OE ne sont pas utilisés dans un périphérique non enregistré, les broches 1 et 11 sont disponibles en tant que signaux d'entrée. La broche 1 utilisera le chemin de rétroaction de MC7 et la broche 11 utilisera le chemin de rétroaction de MCo.

E / S combinatoires dans un Périphérique

Les paramètres du bit de contrôle sont SGO = 1, SG1 = 1 et SLOC = 1. Seules sept conditions de produit sont disponibles pour la porte OU. Le huitième terme de produit est utilisé pour activer le tampon de sortie. Le signal sur la broche d'E / S est renvoyé au réseau ET via le multiplexeur de rétroaction. Cela permet à la broche d'être utilisée comme entrée. Comme CLK et ÔË ne sont pas utilisés dans un appareil non enregistré, les broches 1 et 11 sont disponibles en tant qu'entrées. La broche 1 utilisera le chemin de rétroaction de MC_0 Et la broche 11 utilisera le chemin de rétroaction de MC_0.

E / S combinatoires dans un périphérique enregistré

Les paramètres de bit de contrôle sont SGO = 0, SG1 = 1 et SLOx = 1. Seules sept conditions de produit sont disponibles pour la porte OU. Le huitième terme de produit est utilisé comme validation de sortie. Le signal de retour est l'E / S correspondante signal.

Configuration d'entrée dédiée

Les paramètres du bit de contrôle sont SGO = 1, SG1 = 0 et SLOx = 1. Le tampon de sortie est désactivé. Sauf pour MC_0 et MC_7, le signal de retour est une E / S adjacente. Pour MC_0 et MC7, les signaux de retour sont les broches 1 et 11. Ces configurations sont résumées dans le tableau 2 et illustrées dans la figure 28.

Polarité de sortie programmable

La polarité de chaque macrocellule peut être active-élevée ou active-basse, soit pour correspondre aux besoins du signal de sortie, soit pour réduire les conditions du produit. La polarité programmable permet d'écrire les expressions booléennes dans leur format le plus compact forme (vraie ou inversée), et la sortie peut toujours être de la polarité souhaitée. Il peut également enregistrer "DeMorganizing" " efforts. La sélection se fait par un bit programmable SL1x qui commande une porte OU exclusif à la sortie de la logique ET / OU. La sortie est active haut si SL1x est 1 et active bas si SL1x est 0.

Le PAL versatile (polyvalent), dont le membre le plus connu est le 22V10, présente une évolution des PAL vers les circuits logiques programmables de plus haut niveau. Mais ils utilisent une structure de cellule de sortie qui s'apparente à un EPLD. D'après la figure suivante, on remarque que la cellule de sortie dispose d'une bascule D pré-positionnable associée à deux multiplexeurs programmables. Les connexions S0 et S1 sont réalisées grâce à des fusibles internes.

Le bloc de sortie des PAL versatiles permet de configurer (par programmation) le mode d'utilisation de la broche de sortie.

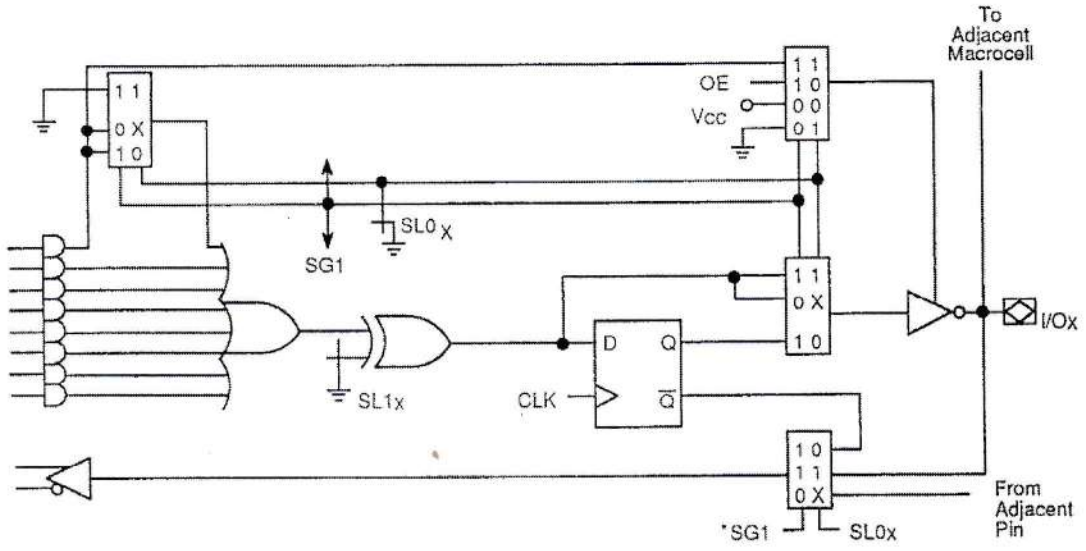


Figure 26. Macro cellule de PALCE16V8

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

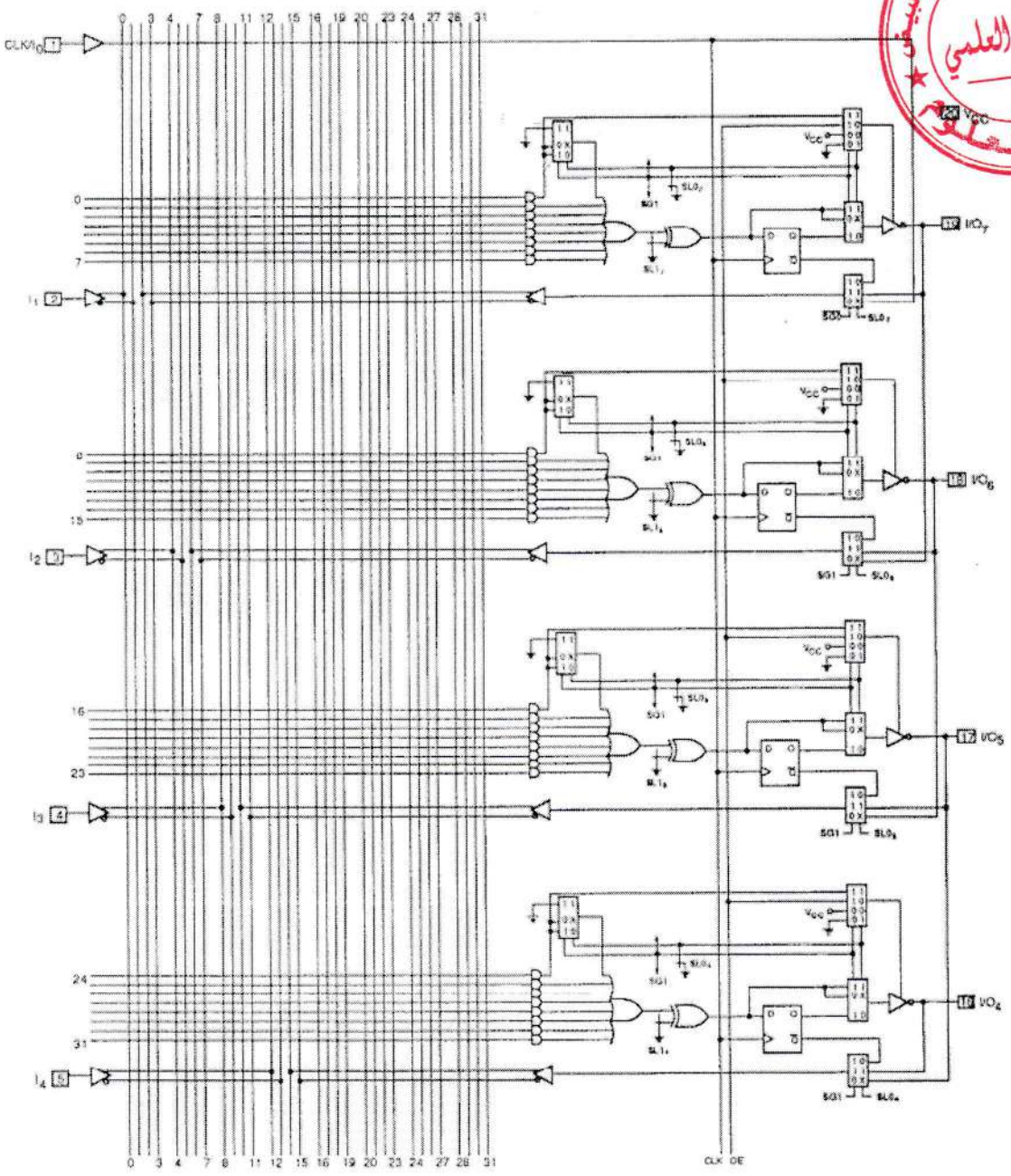


Figure 27. PALCE16V8

Cette sortie peut adopter plusieurs configurations (d'où le terme polyvalent), le 22V10 pouvant donc être utilisé à la place de tous les PAL bipolaires classiques :

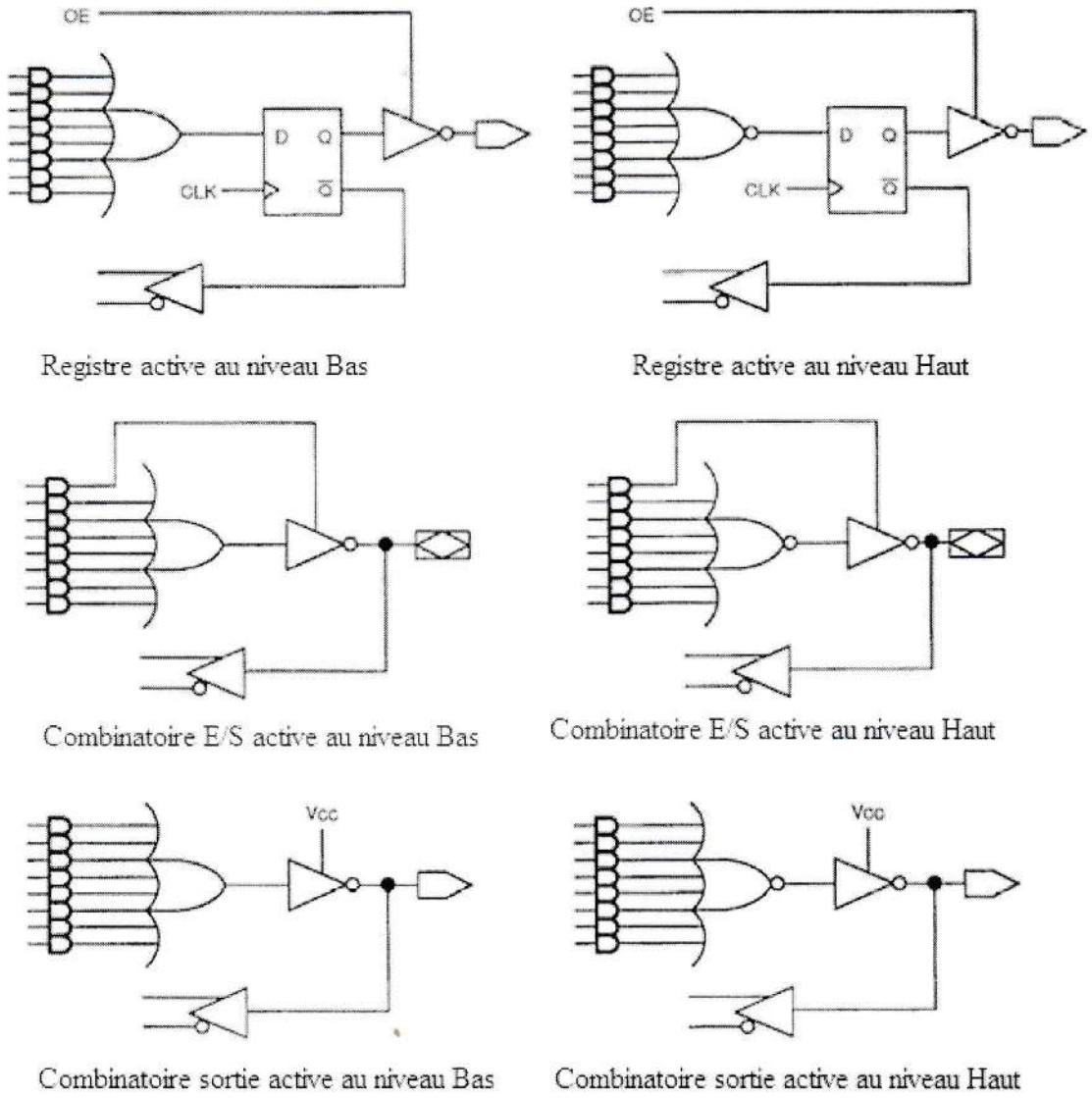
Les structures de sorties dite versatile proposent quatre configurations possibles suivant les valeurs de S0 et S1.

Ce qui donne :

SG0	SG1	SL0x	configuration
0	1	0	sortie à registre
0	1	1	Registre et combinatoire E/S.
1	0	0	sortie combinatoire
1	0	1	Entrée combinatoire
1	1	1	combinatoire E/S.

Table 2. Différentes configurations

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم



Registre active au niveau Bas

Registre active au niveau Haut

Combinatoire E/S active au niveau Bas

Combinatoire E/S active au niveau Haut

Combinatoire sortie active au niveau Bas

Combinatoire sortie active au niveau Haut

Figure 28. Différentes configurations de la macrocellule

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد التكنولوجيا

Les premiers PAL pouvaient être assez facilement programmés à la main. Toutefois, la réalisation de fonctions complexes est devenue rapidement inextricable. Des logiciels de développement sont donc apparus afin de faciliter ce travail.

Tous les PAL disposent d'un fusible ou bit de sécurité. Ce fusible, une fois claqué, interdit la relecture d'un composant déjà programmé. En effet, il arrive que des entreprises indélicates soient tentées de copier les PAL développés par leurs concurrents.

Un des inconvénients des circuits bipolaires à fusibles, est qu'ils ne peuvent pas être testés à la sortie de l'usine. Pour tester leur fonctionnement, il faudrait en effet claquer les fusibles, ce qui interdirait toute programmation ultérieure. A l'origine, les premiers PAL étaient bipolaires puisqu'ils utilisaient la même technologie que les PROM bipolaires à fusibles. Il existe maintenant des PAL en technologie CMOS (appelés GAL (Generic Array Logic) par certains fabricants), programmables et effaçables électriquement, utilisant la même technologie que les mémoires EEPROM. Comme ils sont en technologie CMOS, ils consomment beaucoup moins, en statique, que les PAL bipolaires de complexité équivalente.

1.8.4.4 Les références des PAL

Exemple :

PAL CE 16 V 8 H -5 P C /5

PAL : Type de famille (PAL=Programmable Array Logic)
CE : Technologie (CE= CMOS Effaçable Electriquement)
16 : Nombre d'entrées
V : type de sortie (V : PAL Versatile
H : PAL combinatoire active au niveau Haut
L : PAL combinatoire active au niveau Bas
C : Sortie Complémentaire
R : Sortie à registre
X : Sortie OU exclusif avec registre)
8 : Nombre de sorties
H : Puissance (H=1/2 W 90-125mA)
Q= 1/4 W 55 mA)
-5 : La vitesse (-5 : 5 ns
-7 : 7.5 ns
-10 : 10 ns
-15 : 15 ns
-20 : 20 ns
-25 : 25 ns)
P : Type de boitier (P : 20 broches plastique DIP (PD 020)
J : 20 broches support plombée en plastique
S : 20 broches ensemble en plastique type Gull-Wing (So 020))

C : Conditions d'utilisations (C : commerciale (0°C à +75°C)
I : industriel (-40°C à +85°C))
/5 : Désignation de programmation (blanc : Algorithme initial
/4 : Première révision
/5 : Deuxième révision)

1.8.5 GAL

Au fur et à mesure que la popularité du PAL augmentait, des fonctionnalités supplémentaires ont été mises en œuvre pour prendre en charge des conceptions plus sophistiquées. L'une des améliorations les plus significatives a été l'ajout d'une logique de sortie macrocellule (OLMC). Une OLMC a fourni une bascule D et un multiplexeur sélectionnable afin que la sortie du circuit SOP du PAL puisse être utilisée soit comme sortie système soit comme entrée d'une bascule D. Cela a permis la mise en œuvre de la logique séquentielle et des machines à états finis. La OLMC pourrait également être utilisée pour acheminer la broche d'E/S vers le PAL afin d'augmenter le nombre d'entrées possibles dans les expressions SOP. Enfin, la OLMC a fourni un multiplexeur pour permettre la rétroaction de la sortie PAL ou de la sortie de la bascule D. Cette architecture a été nommée une logique de tableau générique (GAL) pour distinguer ses fonctionnalités d'un PAL standard.

GAL signifie Generic Array Logic ou encore réseau logique le nom de GAL a été déposé par LATTICE SEMICONDUCTOR. Leur fonctionnement est identique aux PAL CMOS.

- Les GAL sont des PAL à technologie CMOS, sont programmables et effaçables électriquement.
- On retrouve les mêmes références qu'en PAL.

Protection contre la duplication :

Les GAL sont dotés d'un bit de sécurité (empêchant la lecture du contenu du circuit). Ils sont constitué de 8 octets appelés signature qui contiennent des infos sur le produits.

Avantage des GALs / aux PALs :

L'inconvénient majeur des PALs est qu'ils ne sont programmables qu'une seule fois. LATTICE a donc pensé, il y a un peu plus de 10 ans, à remplacer les fusibles irréversibles des PALs par des transistors MOS FET pouvant être régénérés. Ceci a donc donné naissance aux GALs que l'on pourrait traduire par « Réseau logique Générique ». Ces circuits peuvent donc être reprogrammés à volonté sans pour autant avoir une durée de

vie restreinte. On peut aussi noter que dans leur structure interne les GALs sont constitués de transistor CMOS alors que les PALs classiques sont constitués de transistors bipolaires. La consommation des GALs est donc beaucoup plus faible. Depuis d'autres constructeurs fabriquent ce type de produit en les appelants « PAL CMOS » (PAL CE). Par soucis de remplacer les PALs, LATTICE a équipé la plupart de ses GALs de macrocellules programmables permettant d'émuler n'importe quel PAL. Ces structures de sortie sont donc du type « Versatile » (V).

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

1.8.6 Hard Array Logic (HAL)

Pour les conceptions matures, les PAL et les GAL pourraient être implémentés en tant que dispositif logique à matrice dure (HAL). Un HAL était une version d'un PAL ou GAL qui avait les connexions de plan ET implémentées pendant la fabrication au lieu de souffler des fusibles. Cette architecture était plus efficace pour les applications à volume élevé car elle éliminait l'étape de programmation post-fabrication et le dispositif n'avait pas besoin de contenir les circuits de programmation.

En 1983, Altera Inc. a été fondée en tant qu'entreprise de dispositifs logiques programmables. En 1984, Altera a sorti sa première version d'un PAL avec une caractéristique unique qu'il pouvait être programmé et effacé plusieurs fois en utilisant un programmateur et une source de lumière UV similaire à une EEPROM.

1.8.7 Les EPLD :

Les EPLD (Erasable Programmable logic Device) sont des circuits programmables électriquement et effaçables, et qui sont aux P.A.L. ce que sont les U.V.P.R.OM. Aux P.R.O.M. Les E.P.L.D. peuvent être effacés par U.V. ou électriquement. Ils sont encore appelés P.A.L. CMOS. Historiquement, les premiers EPLD étaient des GAL effaçables aux U.V. Il existe maintenant des EPLD effaçables électriquement.

Ces circuits, développés en premier par la firme ALTERA, sont arrivés sur le marché en 1985. Les EPLD sont une évolution importante des PAL CMOS. Ils sont basés sur le même principe pour la réalisation des fonctions logiques de base. Les procédés physiques d'intégration permis par les EPLD sont nettement plus importants que ceux autorisés par les PAL CMOS. Ces circuits ont une capacité en nombre de portes et en possibilités de configuration est supérieure à celle des GAL.

En effet, les plus gros EPLD actuellement commercialisés intègrent jusqu'à 24000 portes logiques dont 12000 sont réellement accessibles à l'utilisateur. On peut ainsi loger dans un seul boîtier, l'équivalent d'un schéma logique utilisant jusqu'à 50 à 100 PAL classiques.

- Densité d'intégration supérieure aux PAL.
- Fonctionner à une vitesse au moins égale aux PAL bipolaire.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Description fonctionnelle :

EPLD de la famille MAX :

- ❖ Logic Array broches(LABs)
- ❖ Macro cellules
- ❖ Ex panseur
- ❖ Réseaux d'Interconnections Programmables (PIA)
- ❖ I/O control blocks

Comme les PAL CMOS, les EPLD font appel à la notion de macrocellule qui permet, par programmation, de réaliser de nombreuses fonctions logiques combinatoires ou séquentielles.

Le schéma type de la macrocellule de base d'un EPLD est présenté ci-dessous. On remarque que le réseau logique est composé de 3 sous-ensembles :

• le réseau des signaux d'entrées provenant des broches d'entrées du circuit,

• le réseau des signaux des broches d'entrées/sorties du circuit,

• le réseau des signaux provenant des autres macrocellules.

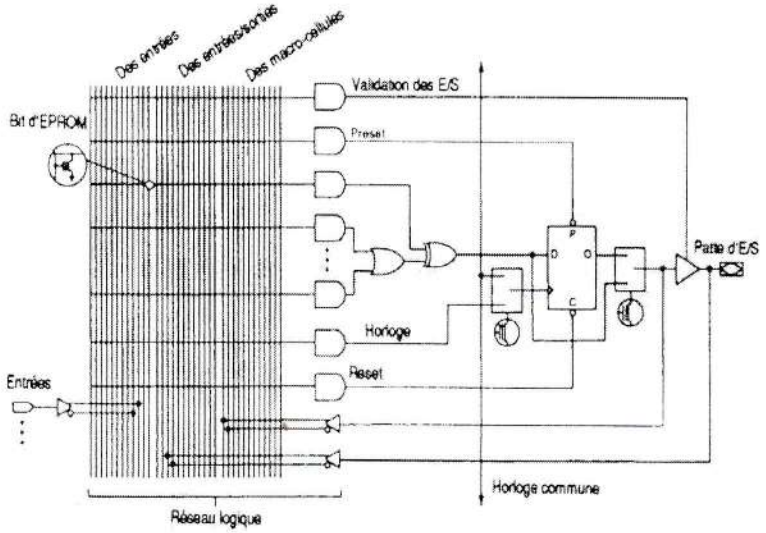


Figure 29. Macrocellule d'un EPLD

Outre la logique combinatoire, la macrocellule possède une bascule configurable (bascule D, T, RS ou JK). Cette bascule peut être désactivée par programmation d'un multiplexeur. Le signal d'horloge peut être commun à toutes les macrocellules, ou bien provenir d'une autre macrocellule via le réseau logique.

La partie nommée OLMC (OUTPUT LOGIC MACROCELL) est versatile, ce qui veut dire qu'il est possible par programmation de choisir entre une configuration de sortie combinatoire ou séquentielle.

La figure ci-dessous montre la structure et la table de fonctionnement d'une OLMC :

- Le multiplexeur 4 vers 1 permet de mettre en circuit ou non la bascule D, en inversant ou non les signaux.
- Le multiplexeur 2 vers 1 permet de réinjecter soit la sortie, soit l'entrée du buffer de sortie vers la matrice.

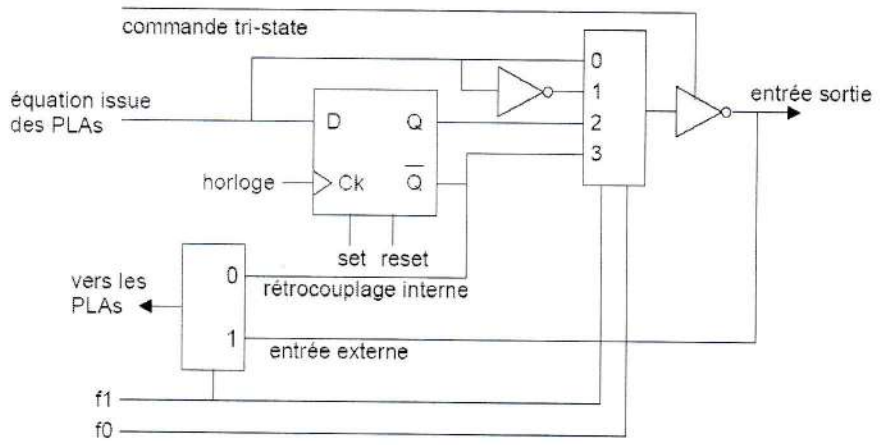


Figure 30. Macro cellule configurable

Quel que soit la famille d'EPLD, la fonctionnalité de la macrocellule ne change guère. En revanche, plus la taille des circuits augmentent, plus les possibilités d'interconnexions et le nombre de macrocellules augmentent. On voit ci-dessous la structure d'un EPLD de la famille MAX 5000 d'ALTERA.

Il existe plusieurs types d'EPLD en technologie CMOS :

- Les circuits programmables électriquement et non effaçables. Ce sont les EPLD de type OTP (One Time Programmable).
- Les circuits programmables électriquement et effaçables aux UV.
- Les circuits programmables électriquement et effaçables électriquement dans un programmateur.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد الرياضيات

• Les circuits programmables électriquement et effaçables électriquement sur la carte (ISP : In Situ Programmable), utilisant une tension unique de 5 V.

Les plus rapides des EPLD ont des temps de propagation (entrée vers sortie sans registre) de l'ordre de 12 ns. En revanche, comme ils sont en technologie CMOS, leur consommation croît avec l'augmentation de la fréquence de fonctionnement. Le taux d'utilisation des ressources d'un EPLD dépasse rarement 80 %. Avec les EPLD, il est possible de prédire la fréquence de travail maximale d'une fonction logique, avant son implémentation. On rencontre parfois le terme CPLD (Complex Programmable Logic Device). Ce terme est généralement utilisé pour désigner des EPLD ayant un fort taux d'intégration.

1.8.8 LES CPLD :

Alors que de la demande de dispositifs programmables augmenté de plus en plus. L'architecture du PAL n'a pas pu évoluer efficacement pour un certain nombre de raisons :

- premièrement, à mesure que la taille des circuits de logique combinatoire augmentait, le PAL a rencontré des problèmes de fan-in dans son plan ET.
- deuxièmement, pour chaque entrée ajoutée au PAL, la quantité des circuits nécessaires sur la puce a augmenté géométriquement en raison de la nécessité d'une connexion à chaque porte ET en plus de la zone associée à la CLOSM supplémentaire.

Cela a conduit à une nouvelle architecture PLD dans laquelle l'interconnexion sur puce a été partitionnée sur plusieurs PAL sur une seule puce. Ce partitionnement signifiait que toutes les entrées de l'appareil ne pouvaient pas être utilisées par chaque PAL, de sorte que la complexité de la conception augmentait, cependant, les ressources programmables supplémentaires ont compensé cet inconvénient, et cette architecture a été largement adoptée. Cette nouvelle architecture a été appelée un dispositif logique programmable complexe (CPLD).

CPLD signifie Complex Programmable Logic Device ces circuits sont composés de plusieurs PALs élémentaires reliés entre eux par une zone d'interconnexion. Leurs architectures sont basées sur celles des PALs. Grâce à cette architecture, ils permettent d'atteindre des vitesses de fonctionnement élevées (plusieurs centaine de Mhz).

Ces circuits ont une capacité en nombre de portes et en possibilités de configuration très supérieure à celle des PALs. Le nombre de portes peut varier entre 100 et 100 000 portes logiques et entre 16 et 1000 bascules.

1.8.8.1 Structure générale d'un CPLD :

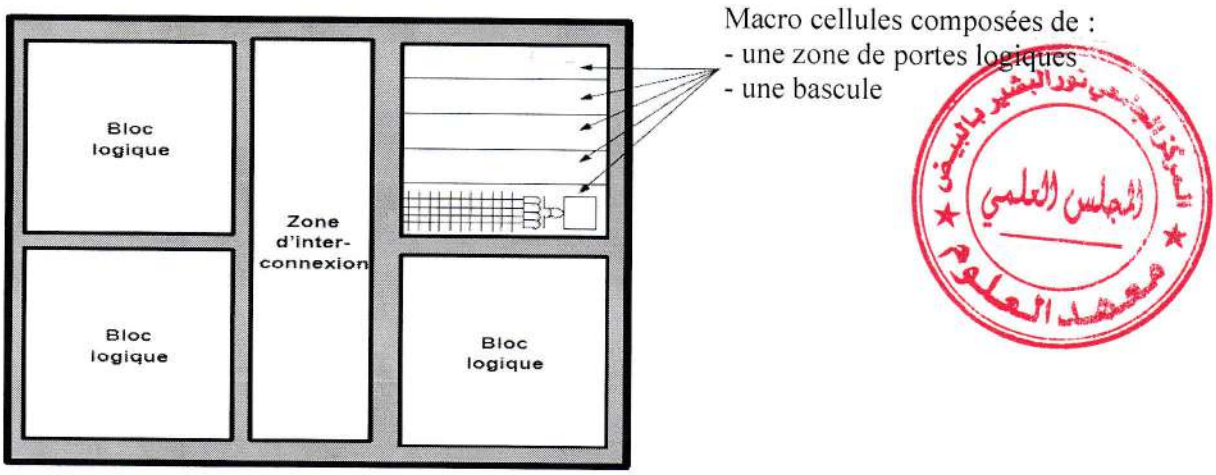
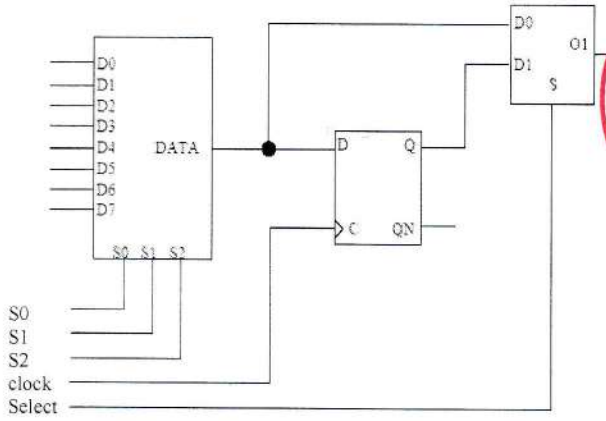


Figure 31. Macrocellule d'un CPLD

1.8.9 Les FPGA :

Pour répondre au besoin de ressources encore plus programmables, une nouvelle architecture a été développée par Xilinx Inc. en 1985. Cette nouvelle architecture a été appelée un réseau de portes programmables sur site (FPGA). Un FPGA se compose d'un tableau de blocs logiques programmables (ou d'éléments logiques) et d'un réseau d'interconnexion programmable qui peut être utilisé pour connecter n'importe quel élément logique à n'importe quel autre élément logique. Chaque circuit logique contenu dans un bloc pour mettre en œuvre des circuits logiques combinatoires arbitraires en plus d'une bascule D et d'un multiplexeur pour la direction du signal. Cette architecture a mis en œuvre efficacement une CLOSM dans chaque bloc, offrant ainsi une flexibilité ultime et fournissant beaucoup plus de ressources pour la logique séquentielle. Aujourd'hui, les FPGA sont les dispositifs logiques programmables les plus couramment utilisés, Altera Inc. et Xilinx Inc. étant les deux plus grands fabricants. La Figure 32 montre l'architecture générique d'un FPGA.

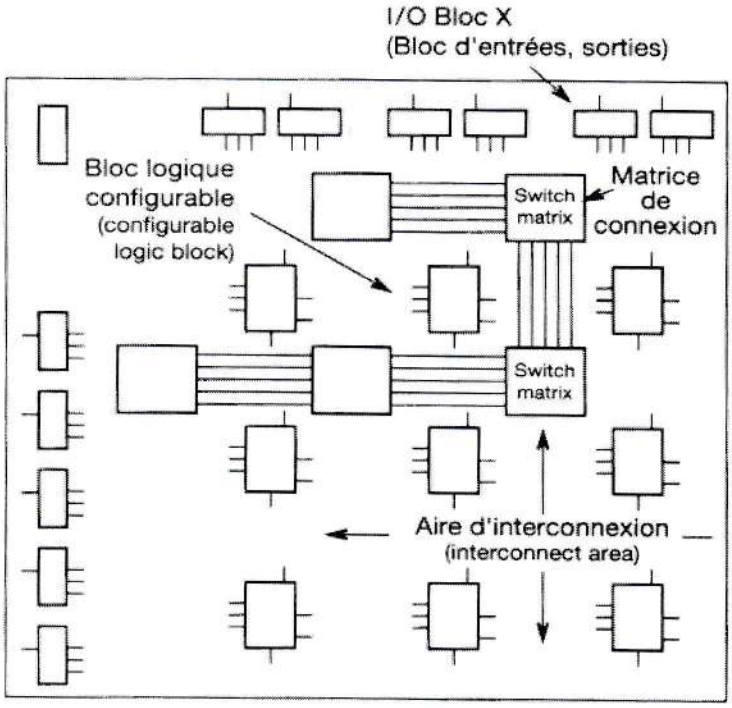


المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 32. Cellule de base d'un FPGA

Les cellules de base d'un FPGA sont disposées en rangées et en colonnes. Des lignes d'interconnexions programmables traversent le circuit, horizontalement et verticalement, entre les diverses cellules. Ces lignes d'interconnexions permettent de relier les cellules entre elles, et avec les plots d'entrées/sorties. Les connexions programmables sur ces lignes sont réalisées par des transistors MOS dont l'état est contrôlé par des cellules mémoires SRAM. Ainsi, toute la configuration d'un FPGA est contenue dans des cellules SRAM. Contrairement aux EPLD, on ne peut pas prédire la fréquence de travail maximale d'une fonction logique, avant son implémentation. En effet, cela dépend fortement du résultat de l'étape de placement routage.

Les FPGAs à la différence des CPLDs sont assimilables à des A.S.I.C. (Application Specific Integrated Circuit) programmables par l'utilisateur. La puissance de ces circuits est telle qu'ils peuvent être composés de plusieurs milliers voire millions de portes logiques et de bascules. Les dernières générations de FPGA intègrent même de la mémoire vive (RAM). Les deux plus grands constructeurs de FPGA sont XILINX et ALTERA. Ils sont composés de blocs logiques élémentaires (plusieurs milliers de portes) qui peuvent être interconnectés. De plus en plus les capacités des CPLDs et des FPGAs se rapprochent. Le principal critère de choix entre les deux familles est la vitesse de fonctionnement. En effet les CPLDs acceptent des fréquences de fonctionnement beaucoup plus élevées que les FPGAs. Chaque bloc configurable est constitué de réseau de portes logiques ou des fonctions logiques complexes (compteur, multiplexeur etc...).



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 33. Structure d'un FPGA de type Xilinx.

Par une simple programmation électrique (d'une mémoire SRAM) on peut :

- configurer un bloc logique ou plusieurs
- interconnecter entre eux les blocs grâce à une matrice de connexion

On peut aussi électriquement déprogrammer ce que l'on avait programmé.

Comparaison entre CPLD et FPGA :

	Avantages	Inconvénients
CPLD	Non volatile	
	Compteur et machines d'états rapides	Les ressources de routage son faibles
	Logique combinatoire ou de contrôle	Fonction réclamant peu de routage
	Les temps d'arrives sont déterministes	
FPGA	Architecture micro programmée, DSP	Les temps d'arrives dépends du routage
	Système séquentielle	Reconfiguration par SRAM
	Densité d'intégration élevée	Nécessite une PROM (non volatile)

Table 3. Tableau comparative entre CPLD et FPGA

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد

2.1 Les Technologies d'interconnexion :

Premier critère de choix d'un circuit programmable, la technologie utilisée pour matérialiser les interconnexions détermine les aspects électriques de la programmation : maintien (ou non) de la fonction programmée en l'absence d'alimentation, possibilité (ou non) de modifier la fonction programmée, nécessité (ou non) d'utiliser un appareil spécial (un programmateur).

L'un des éléments clé des circuits étudie est la connexion programmable. Le choix d'une technologie dépendra essentiellement :

-la densité d'intégration

-la rapidité de fonctionnement une fois le composant programme ; fonction de la résistance à l'état passant et des capacités parasites

-la facilite de mise en œuvre (programmation sur site, reprogrammation etc.)

-la possibilité de maintien de l'information

Connexions programmable une seul fois (OTP : One Time Programming)

2.1.1 Les cellules à fusible :

Première méthode employée, la connexion par fusibles, est en voie de disparition. On ne la rencontre plus que dans quelques circuits de faible densité, de conception ancienne. Leur principe consistait à détruire un fusible conducteur par passage d'un courant fourni par une tension supérieure à alimentation (12 à 25v).

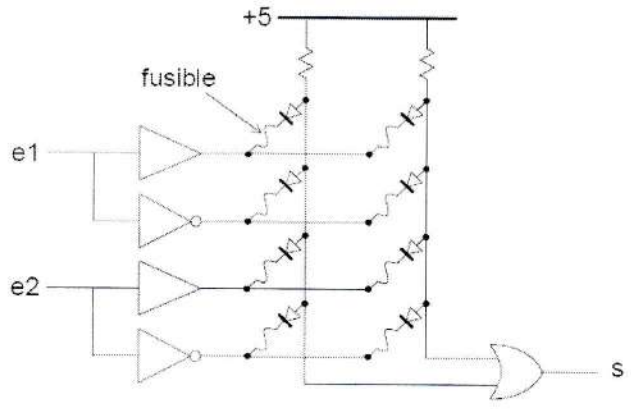


Figure 34. Cellule élémentaire d'un PLD à fusibles

La figure ci-dessus en illustre le principe ; toutes les connexions sont établies à la fabrication.

La connexion est supprimée par claquage du fusible, obtenu par l'application d'une tension (de 12 à 25 V)

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Cette technologie maintenant abandonnées pour des raisons de manque de fiabilité. Le fait de griller les fusibles provoque des perturbations qui peuvent affecter le reste du circuit. De plus, cette programmation est irréversible et ne permet pas donc la reprogrammation.

2.1.2 Les Cellules à anti fusible :

En appliquant une tension importante (6 v pendant 1 ms) a un isolant entre deux zones de semi-conducteur fortement dopées, ce dernier diffuse dans l'isolant et le rend conducteur. Chaque cellule occupe environ 1.8 um^2 (700 um^2 pour un fusible) ; cette technologie très en vogue permet une haute densité d'intégration.

Le principe est, à l'échelle microscopique, celui de la soudure électrique par points. Un point d'interconnexion est réalisé au croisement de deux pistes conductrices (métal ou semi-conducteur selon les procédés de fabrication), séparées par un isolant de faible épaisseur. Une surtension appliquée entre les deux pistes provoque un perçage définitif du diélectrique, ce qui établit la connexion.

2.1.3 Les cellules anti-fusibles à diélectrique

Un antifusible est un élément programmable qui à l'inverse des fusibles n'est passant qu'après programmation. La connexion s'effectue en détruisant un diélectrique

Disposition verticale → gain en surface élaboré par Actel en 1986.

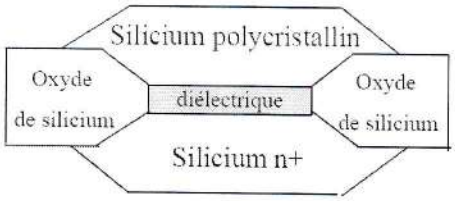


Figure 35. Cellule antifusible à diélectrique

PLICE : Programmable Low Impedance Circuit Element sandwich conducteur/isolant/conducteur surface de la cellule = 1,8 μm^2

2.1.4 Les cellules anti-fusibles en silicium amorphe

Technologie introduite par Cypress même fonction que la précédente avec une résistance plus faible à l'état passant ce qui réduit les délais de propagation à travers les interconnexions.

Cellules reprogrammables :

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد الإعلام الآلي

2.1.5 Les cellules à transistors MOS a grille flottante et EPROM

L'apparition du transistor MOS a grille flottante a permis de rendre le composant bloqué ou passant sans application permanente d'une tension de commande. Le principe consiste à piéger ou non (*à l'aide d'une tension supérieur à la tension habituelle d'alimentation*) des électrons dans la grille.

Programmation : piéger des électrons dans la grille flottante qui s'opposent à la conduction dans le canal ; le transistor est alors équivalent à un interrupteur ouvert. Lorsque le transistor n'est pas programmé, la grille flottante ne contient aucun électron, le canal est conducteur et le transistor est équivalent à un interrupteur fermé. L'extraction éventuelle des électrons piégés permet le retour à l'état initial.

Lorsque le transistor n'est pas programmé, la grille flottante ne contient aucun électron, le canal est conducteur et le transistor est équivalent à un interrupteur fermé.

Le dépôt d'une charge électrique sur la grille isolée d'un transistor fait appel à un phénomène connu sous le nom *d'effet tunnel* : un isolant très mince (une cinquantaine d'angströms, 1 Å = 10^{-10} m) soumis à une différence de potentiel suffisamment grande (une dizaine de volts, supérieure aux 3,3 ou 5 volts des alimentations classiques) est parcouru par un courant de faible valeur, qui permet de déposer une charge électrique sur une électrode normalement isolée. Ce phénomène, réversible, permet de programmer et d'effacer une mémoire. Plusieurs technologies EPROM sont en concurrence.

La figure suivante montre la structure du PLD élémentaire précédent, dans lequel les fusibles sont remplacés par des transistors à grille isolée (technologie FLASH).

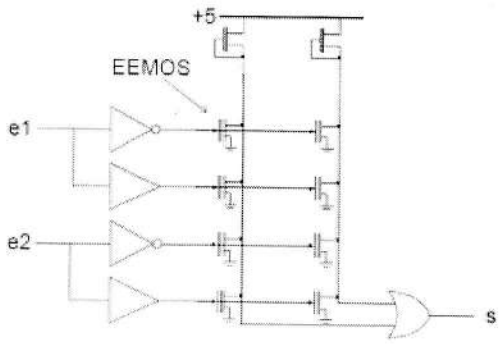
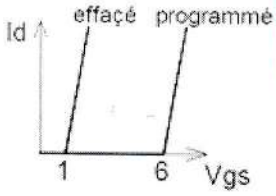


Figure 36. PLD simple a MOS



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 37. Caractéristique $I_D=f(V_{GS})$ pour effacement et programmation

2.1.6 Les Cellules UV EPROM :

Les connexions sont réinitialisable par une exposition à un rayonnement ultra-violet d'une vingtaine de minutes (d'une durée d'environ 20 minutes), permet d'annuler la charge stockée dans la grille flottante. Effacement non sélectif reproductible plus d'un millier de fois.

2.1.7 Les Cellules EEPROM : (Electrically EPROM)

L'effacement et la programmation se font cette fois éclectiquement avec une tension de 12v et peuvent être (75 à 100 μm^2 en CMOS 0.6 µm) et réduit la densité d'intégration possible. D'autre part le nombre de cycles de programmation est limite à un nombre de 100 (en CMOS 0.6 µm) à 10 000 (en CMOS 0.8 µm) à cause de la dégradation des isolants. La programmation ou l'effacement d'une cellule dure quelques ms).

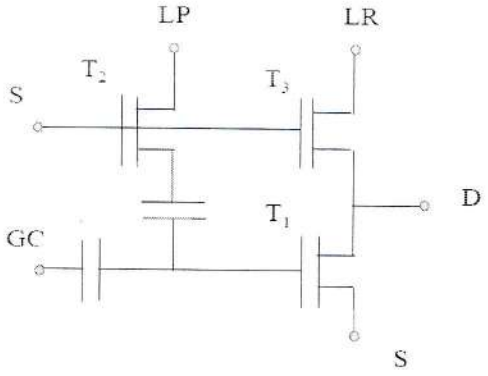


Figure 38. Cellule EEPROM

2.1.8 Les Cellules Flash EEPROM :

L'utilisation de deux transistors par cellule uniquement (5 pour l'EEPROM) et une structure verticale permettent une densité intégration importante (25 μm^2 par cellule en CMOS 0.6 µm) trois à quatre fois plus importante que l'EEPROM, mais quand même 10

المركز الجامعي نور البشير بالبيض
المجلس العلمي

fois moins que la technologie à antifusible. Le nombre de cycle d'écriture (10^4 a 10^6) est également plus grand que pour l'EEPROM car l'épaisseur de l'isolant est plus importante.
Par contre la simplicité de la cellule élémentaire n'autorise pas une reprogrammation sélective (éventuellement par secteur).

La tension de programmation et d'effacement est de 12v, avec un temps de programmation de quelques dizaines de µs pour un temps d'effacement de quelques ms.
Un des inconvénients des cellules flash et EEPROM de nécessiter une alimentation supplémentaire pour la programmation et effacement est pallié les constructeurs en intégrant dans le circuit un système à pompe de charge fournissant cette alimentation. Le composant peut alors être programmé directement sur la carte ou il est utilisé. On parle alors de composant ISP (in situ programmation ou encore suivant les sources, in system programmation.

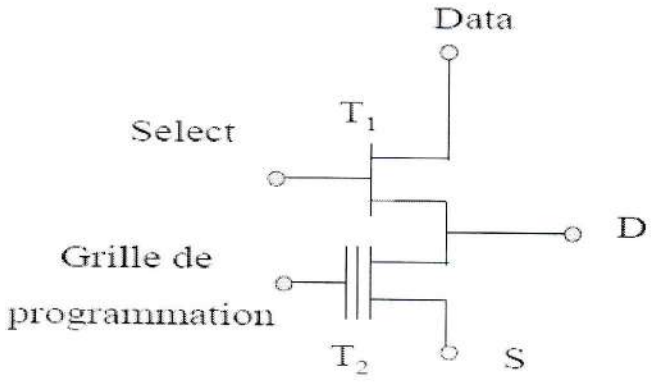


Figure 39. Cellule Flash EEPROM

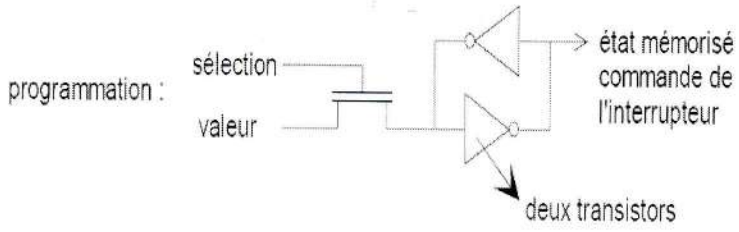
Programmation 1000 fois plus rapide que l'effacement.
Plusieurs cellules sont programmées simultanément.
Nombre de cycles de programmation supérieur à 10000.

2.1.9 Les technologies à RAM statique -SRAM

2.1.10 Les Cellules SRAM a transistors MOS classique :

Ce principe est classiquement choisi pour le FPGA.
Dans les circuits précédents, la programmation de l'état des interrupteurs, conservée en l'absence de tension d'alimentation, fait appel à un mode de fonctionnement électrique particulier. Dans les technologies à mémoire statique (SRAM), l'état de chaque interrupteur est commandé par une cellule mémoire classique à quatre transistors (plus

un transistor de programmation), dont le schéma de principe est celui de la figure suivante.



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 40. Cellule SRAM

Le choix d'une cellule SRAM à 6 transistors permet de bénéficier d'un accès sélectif rapide (quelques ns) en cours d'utilisation. La taille d'une cellule n'est que deux fois plus forte (50 cm^2 par cellule) qu'avec un flash EEPROM.

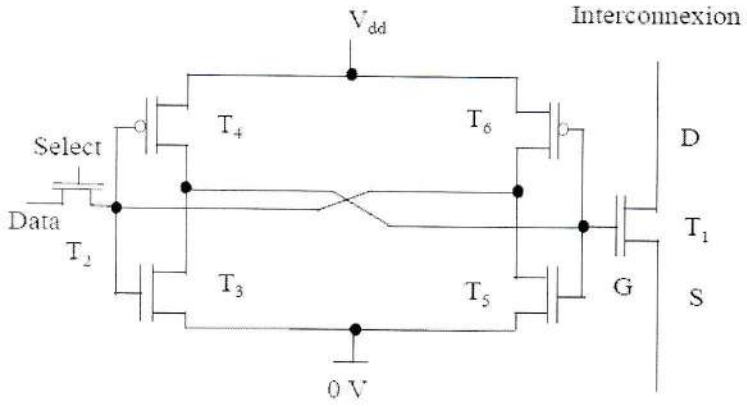


Figure 41. Cellule SRAM à 6 transistors

surface environ 50 μm^2 chargement d'une nouvelle configuration partielle ou totale possible en cours d'utilisation rapidité de quelques dizaines de ns par cellule nécessité de charger la configuration à chaque mise sous tension : « Le fait que d'utiliser une mémoire de type RAM (volatile) impose la recharge de la configuration a chaque mise sous tension : une PROM série mémorise généralement les données». Mémorisation externe.

La modification de la configuration d'un circuit devient alors une opération logique quasi ordinaire, qui ne nécessite pas d'opération électrique spéciale. Ces circuits permettent des reconfigurations, partielles ou totales, en nombre illimité.

Le prix à payer pour cette souplesse est que les cellules SRAM doivent être rechargées à chaque mise sous tension et que chaque interrupteur occupe plusieurs transistors : l'interrupteur lui-même et les transistors de la cellule mémoire.

Critères pour les interconnexions :

- Rapidité de propagation à travers l'interrupteur (produit résistance - capacité parasite)
- Densité possible des interconnexions (surface de la cellule)
- Facilité d'utilisation (ISP, support, PROM de configuration)
- Maintien de la configuration (volatile)
- Reprogrammablilité

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Type d'interconnexion	EPROM	Antifusible	SRAM
Rapidité	-	+	-
Densité	-	+	--
Facilite	+	-	+
Reprogrammablilité	+	-	++

Table 4. Criteres pour les interconnexions

Famille D'ASIC :

Les circuits programmables font partie des ASIC (Application Specific Integrated Circuit) signifiant circuit intégré spécifique à une application). Ils se partagent en plusieurs familles suivant la complexité de la fonction que l'on désire réaliser (de simples portes logiques jusqu'au microprocesseur). Les ASIC programmés chez le fondeur : le circuit est conçu d'un point de vue logiciel par l'utilisateur, puis il est réalisé par le fondeur.

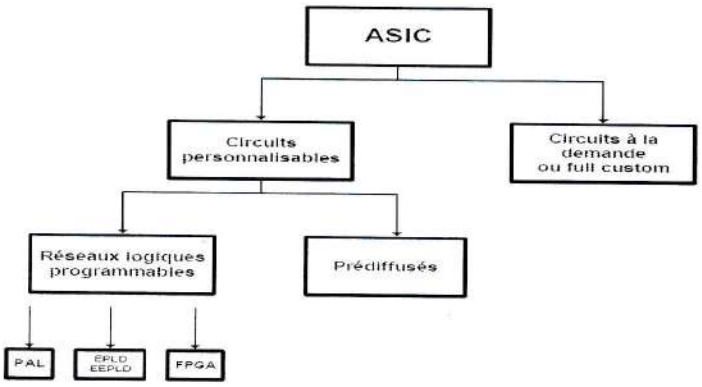


Figure 42. Famille ASIC

Parmi les circuits numériques spécifiques à une application, il faut distinguer deux familles :

• les circuits conçus à partir d'une puce de silicium "vierge" (Full-custom),

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

• les circuits où des cellules standards sont déjà implantées sur la puce de silicium (Semicustom).

"Full custom", on trouve les circuits à la demande et ceux à base de cellules. Le fondeur réalise l'ensemble des masques de fabrication.

"Semi-custom", on trouve les circuits prédiffusés et les circuits programmables. Les cellules standards, déjà implantées sur la puce de silicium, doivent être interconnectées les unes avec les autres. Cette phase de routage est réalisée, soit par masquage chez le fondeur (prédiffusé), soit par programmation. Avant d'aborder le détail de la classification des circuits numériques spécifiques à une application, un aperçu est donné sur les méthodes de réalisation des interconnexions pour les circuits "Semi-custom".

2.1.11 Les circuits Full Custom

Ces circuits sont analogues aux cellules pré caractérisées mais qui sont beaucoup plus compliqués et qui représentent des circuits semi-fini au niveau physique. Posséder une architecture dédiée à chaque application et sont donc complètement définis par les concepteurs. La fabrication nécessite la définition de l'ensemble des masques pour la réalisation. Les temps de fabrication de ces masques et de production des circuits sont de ce fait assez longs. Ces circuits sont ainsi appropriés pour des séries moyennes ou grandes. L'avantage du circuit full custom réside dans la possibilité d'avoir un circuit ayant les fonctionnalités strictement nécessaires à la réalisation des objectifs de l'application. Parmi les circuits full-custom, on distingue :

• Les circuits à la demande,

• Les circuits à base de cellules.

2.1.11.1 Les circuits à la demande :

Ces circuits sont directement conçus et fabriqués par les fondeurs « Le concepteur utilise une bibliothèque de cellules fonctionnelles pré caractérisées électriquement qu'il va assembler. Le fabricant devra tout intégrer sur le silicium et rendre un circuit testé ». Ils sont spécifiques car ils répondent à l'expression d'un besoin pour une application particulière. Le demandeur utilise le fondeur comme un sous-traitant pour la conception et la réalisation et n'intervient que pour exprimer le besoin. Ces circuits spécifiques utilisent au mieux la puce de silicium. Chaque circuit conçu et fabriqué de cette manière doit être produit en très grande quantité pour amortir les coûts de conception.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

2.1.11.2 Les circuits à base de cellules

Les circuits à base de cellules (CBIC : Cell Based Integrated Circuit) permettent des complexités d'intégration allant jusqu'au million de portes. Dans cette catégorie de circuits, on distingue les circuits à base de cellules précaractérisées et les circuits à base de cellules compilées.

***1. Les cellules précaractérisées :* (Cell Array ou Standard Cells)**

Les cellules pré caractérisées sont des entités logiques plus ou moins complexes. Il peut s'agir de cellules de base (portes, bascules, etc.) mais aussi de cellules mémoires (ROM, RAM) ou encore de sous-systèmes numériques complexes (UART, cœur de microprocesseur, PLA, ...).

Toutes ces cellules ont été implantées et caractérisées au niveau physique (d'où la notion de cellules précaractérisées) par le fondeur. La fonctionnalité globale de l'application à réaliser s'obtient en choisissant les cellules appropriées dans une bibliothèque fournie par le fondeur. 2 types de cellules pré caractérisées existent :

- les cellules de hauteur fixe et de largeur variable,
- les cellules de hauteur et de largeur variables.

1 er cas, l'association des cellules permet de définir des canaux pour les interconnexions ; le routage alors est simplifié.

2 éme cas, les canaux ne sont pas bien délimités, ce qui complique le placement-routage.

2. Les circuits à base de cellules compilées

Les circuits à base de cellules compilées sont en fait basés sur l'utilisation de cellules précaractérisées. A la différence des circuits précaractérisés, les cellules ne sont pas utilisables directement mais au travers de modules paramétrables ou modules génériques. Chaque module est créé par la juxtaposition de *n* cellules de même type. La différence entre circuits pré-caractérisés et circuits compilés provient essentiellement de l'outil utilisé pour générer les dessins des masques de fabrication. Ces outils sont appelés des compilateurs de silicium.

Les circuits Semi-Custom

Dans la famille des circuits semi-custom, on distingue deux groupes :

- les circuits pré-diffusés,
- les circuits programmables.

a. Les circuits pré-diffusés : (FPGA ou LCA)

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Ce sont des tranches de silicium comportant des réseaux de portes logiques ou des fonctions logiques plus complexes déjà diffusées sur la puce mais non connectées « *un grand nombre de cellules. Chaque cellule contient soit des portes logiques, soit des transistors et des résistances* ». Le câblage final sera réalisé à la demande du client. La programmation de ce type de circuits revient à assurer la connexion entre ses différents composants. Parmi les circuits prédiffusés, on distingue :

-Les prédiffusés classiques (ou "Gate Array")

-Les réseaux mer de portes ("Sea of Gates").

1. Les circuits pré-diffusés classiques :

Les circuits pré-diffusés classiques possèdent une architecture interne fixe qui consiste, dans la plupart des cas, en des rangées de portes séparées par des canaux d'interconnexion. L'implantation de l'application se fait en définissant les masques d'interconnexion pour la phase finale de fabrication. Ces masques d'interconnexion permettent d'établir des liaisons entre les portes et les plots d'entrées/sorties.

Les circuits pré-diffusés classiques intègrent de 50000 à 1000000 portes logiques et sont intéressants pour des grandes séries. Pour des prototypes ou de petites séries, ils sont progressivement abandonnés au profit des circuits programmables à haute densité d'intégration, comme les FPGA.

La figure suivante donne un exemple de structure pour un prédiffusée classique. Les cellules internes sont de taille fixe et organisées en rangées ou colonnes séparées par les canaux d'interconnexion.

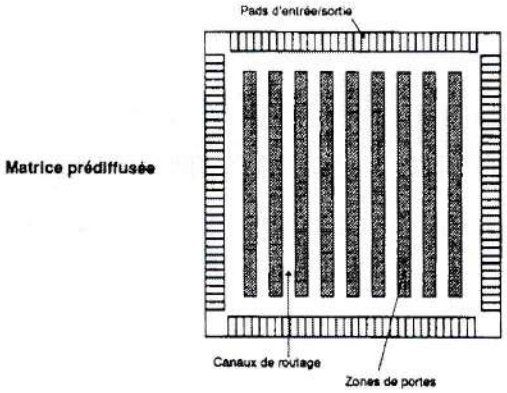


Figure 43: Matrice prédiffusée

2. Les réseaux mer de portes :

Contrairement aux prédiffusée classiques, les circuits mer de portes ne possèdent pas de canaux d'interconnexion, ce qui permet d'intégrer plus d'éléments logiques pour une

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

surface donnée. Les portes peuvent servir, soit comme cellules logiques, soit comme interconnexions. En fait, si ces circuits possèdent la structure logique équivalente à 250000 portes, pratiquement, le nombre moyen de portes utilisables est de l'ordre de 100000, ce qui donne un taux d'utilisation de 40% à 50%.

b. Les réseaux logiques programmables :

Elles permettent à l'utilisateur de programmer ses propres fonctions (combinatoires ou séquentielles). La programmation se fait par fusibles avec des circuits tels que les PAL, PLD, FPLA...etc. ou sans fusibles avec des circuits comme les GAL, EPLD...etc. Ces circuits se présentent comme des réseaux d'opérateurs ET-OU ou des bascules associées à des opérateurs ET-OU. Un circuit programmable peut donc substituer quelques boîtiers SSI ou MSI.

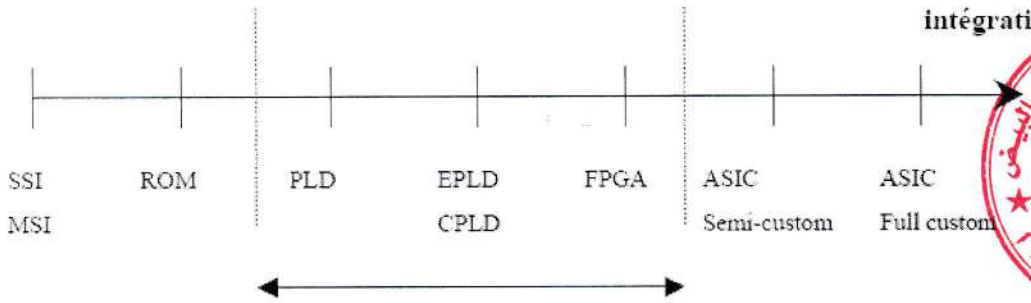
Technologie utilisée pour les interconnexions :

Les cellules standards implantées dans les circuits "Semi-custom" vont de la simple porte jusqu'à une structure complexe utilisant un grand nombre de transistors. Il existe deux manières d'interconnecter ces cellules :

1. Dans les ASIC, les lignes d'interconnexions sont créées par masque (fondeur). Le fondeur réalise les interconnexions des circuits pré-diffusés par métallisation en créant le ou les derniers masques de fabrication.
2. Dans les PLD, les lignes d'interconnexions existent déjà dans le circuit (généralement sous forme de lignes et de colonnes traversant le composant). Il ne reste donc plus qu'à réaliser les bonnes liaisons pour réaliser le chemin voulu afin de relier les cellules logiques. Ces liaisons peuvent se faire :

- par antifusible,
- par cellule mémoire : fusible, EPROM, EEPROM, flash EPROM et SRAM.

Classification des circuits Logiques :



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 44. Circuits Logiques Programmables par L'utilisateur

Performances Comparées :

Complexité (nombre de portes) / volume de production :

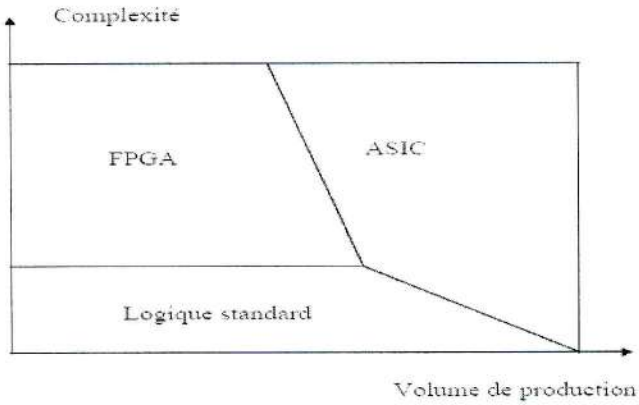


Figure 45. Complexité (nombre de portes) / volume de production

Fréquence utile/nombre de portes :

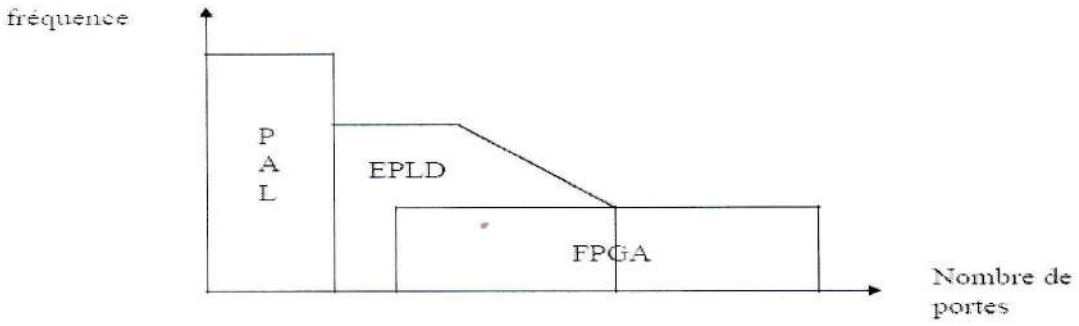


Figure 46. Fréquence utile/nombre de portes

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Comparaison entre les FPGA et les autres circuits spécifiques :

La comparaison et donc le choix entre les différentes technologies car :

- Elle conditionne la conception et l'évolution du produit à concevoir.
- Elle détermine le coût de la réalisation et donc la rentabilité économique du produit.

Comparaison entre les PLD et les ASIC :

Un premier choix doit être fait entre les ASIC et les PLD. Les avantages des PLD par rapport aux ASIC sont les suivants :

• ils sont entièrement programmables par l'utilisateur,

• Ils sont généralement reprogrammables dans l'application, ce qui facilite la mise au point et garantit la possibilité d'évolution,

• les délais de conception sont réduits, il n'y a pas de passage chez le fondeur. En revanche, les inconvénients des PLD par rapport aux ASIC sont les suivants :

• ils sont moins performants en termes de vitesse de fonctionnement (d'un facteur 2 à 3),

• le taux d'intégration est moins élevé (d'un facteur 10 environ),

• les ressources d'interconnexion utilisent en général les 2/3 de la surface de silicium. De plus, le coût de l'ASIC est beaucoup plus faible que le coût du PLD (quoique les choses évoluent très rapidement dans ce domaine, notamment dans la compétition entre FPGA et prédiffusés). Au-delà d'une certaine quantité, l'ASIC est forcément plus rentable que le PLD.

Comparaison entre les FPGA et les EPLD :

Si un PLD est choisi, il faut savoir si on doit utiliser un EPLD ou un FPGA. Les avantages des FPGA par rapport aux EPLD sont les suivants :

• le taux d'utilisation des ressources peut atteindre 80 %, ce qui est meilleur qu'un EPLD,

• ils consomment moins à fonctionnalité identique (< 10 mA par 1000 portes),

• les fonctions réalisables sont plus complexes.

Les inconvénients des FPGA par rapport aux EPLD sont les suivants :

• les EPLD sont plus performants pour certaines fonctions arithmétiques rapides,

• les fréquences de fonctionnement sont variables suivant la méthode de placement routage retenue. Les EPLD ont des fréquences de travail "prédictibles".

En fait, le domaine d'utilisation des FPGA est celui des prédiffusés, par exemple les fonctions logiques ou arithmétiques complexes ou le traitement du signal. Le domaine

d'utilisation des EPLD est plutôt celui des PAL, par exemple les machines d'état complexes.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

المركز الجامعي البيض
المجلس العلمي
معهد العلوم

3.1 Les FPGA (Field Programmable Gate Array).

Les blocs logiques sont plus nombreux et plus simples que pour les CPLDs, mais cette fois les interconnexions entre les blocs logiques ne sont pas centralisées.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

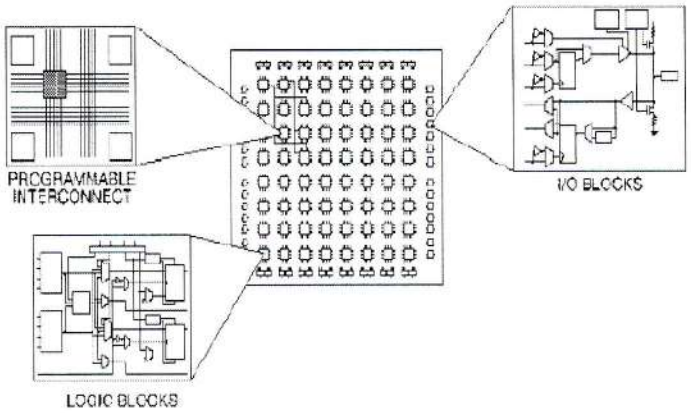


Figure 47. Structure d'une FPGA

Le passage d'un bloc logique à un autre se fera par un nombre de points de connexion (responsables des temps de propagation) fonction de la position relative des deux blocs logiques et de l'état "d'encombrement" de la matrice. Ces délais ne sont donc pas prédictibles (contrairement aux CPLDs) avant le placement routage.

De la phase de placement des blocs logiques et de routage des connexions dépendront donc beaucoup les performances du circuit en termes de vitesse. La figure suivante illustre le phénomène, on peut voir :

une liaison entre deux blocs logiques (BA et BL) éloignés, mais passant par peu de points de connexion, donc introduisant un faible retard.

Une liaison entre deux blocs proches (BD et BH) mais passant par de nombreux points de connexion, donc introduisant un retard important.

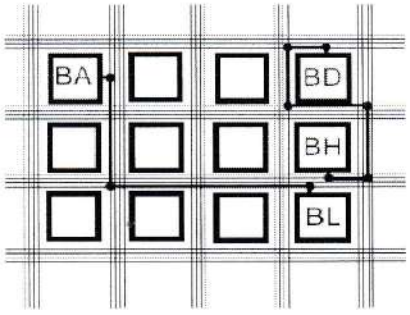


Figure 48. Liaison entre de bloc logique

Les circuits FPGA appelé aussi LCA (logic cells arrays) du fabricant Xilinx utilisent deux types de cellules de base :

- les cellules d'entrées/sorties appelés IOB (Input Output Bloc),
- les cellules logiques appelées CLB (Configurable Logic Bloc). Ces différentes cellules sont reliées entre elles par un réseau d'interconnexions configurable.

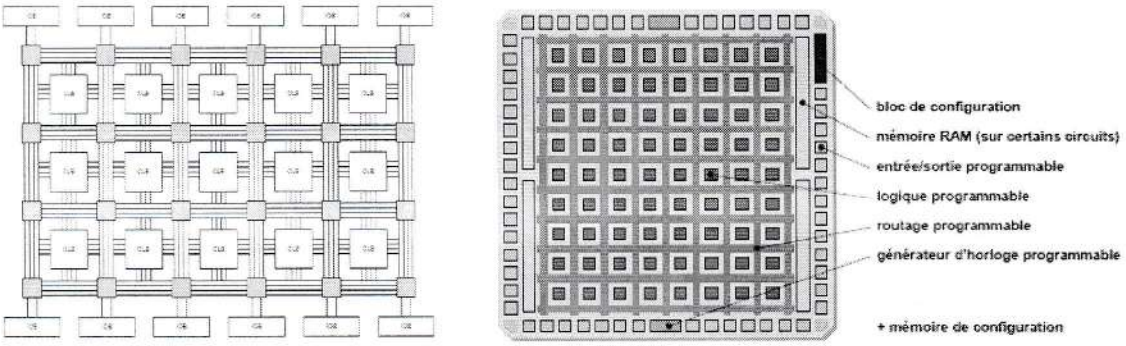


Figure 49. Architecture d'un FPGA

- Matrice de cellules logiques

-Chaque cellule est capable de réaliser une fonction, choisie parmi plusieurs possibles : le choix se fait par programmation

-Les interconnexions entre les cellules sont programmables également

- Deux types, selon la complexité de la cellule :
  - Granularité fine
  - Granularité grossière

-Deux types, selon le mode de programmation :
  - RAM
  - Anti-fusibles

3.2 Blocs logiques programmables

Les blocs logiques configurables (CLB) sont les éléments déterminants les performances du FPGA. Chaque bloc est composé d'un bloc de logique combinatoire composé de deux générateurs de fonctions à quatre entrées et d'un bloc de mémorisation synchronisation composé de deux bascules D. Quatre autres entrées permettent d'effectuer les

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

connexions internes entre les différents éléments du CLB. La figure ci-dessous nous montre le schéma d'un CLB. Il y a deux catégories de blocs de logique programmable : ceux basés sur les multiplexeurs et ceux basés sur les tables de conversion.

Un multiplexeur avec *n* signaux de contrôle peut réaliser toute fonction booléenne à *n* + 1 variables sans l'ajout d'autres portes logiques.

Les CLBs basés sur les tables de conversion utilisent de petites mémoires programmables au lieu de multiplexeurs. Cette approche est similaire à l'approche par multiplexeurs, mais en supposant que les entrées du multiplexeur ne peuvent être que des constantes.

Le CLB est composé de :

- deux tables de conversion (*Look-Up Table – LUT*) programmables à 4 entrées chacune, F et G, qui sont effectivement des mémoires de 16 bits chacune ;

-un multiplexeur 'H' et son entrée associée H1 qui permet de choisir la sortie de l'une des deux tables de conversion ;

-quatre multiplexeurs dont les signaux de contrôle S0 à S3 sont programmables ; et, deux éléments à mémoire configurables en bascules ou loquets.

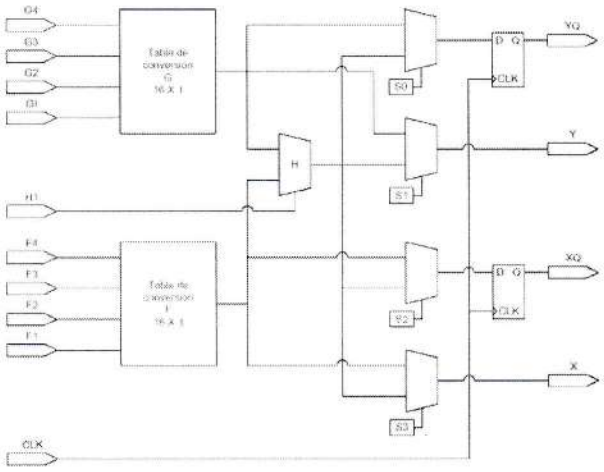


Figure 50. Bloc logique programmable simplifié – Xilinx

Chaque cellule logique, appelée Configurable Logic Block (CLB), est programmée à l'aide d'une *look-up table* (LUT)

Le chargement de la configuration peut prendre plusieurs millisecondes, temps pendant lequel le circuit est inutilisable

On peut générer deux sorties par CLB, combinatoires ou séquentielles. Il est possible de générer 2 fonctions quelconques à 4 variables, une fonction quelconque à 5 variables ou certaines fonctions à 9 variables

· L'unité logique de base est la Logic Cell (LC) : un générateur de fonctions logiques à 4 variables, une logique de *carry* et un élément de mémoire

· Deux LC forment un *slice* et deux *slices* forment un CLB

· En combinant les deux LCs d'un *slice*, on peut implémenter une fonction quelconque à 5 entrées ou certaines fonctions jusqu'à 9 variables.

En combinant les 4 LCs d'un CLB, on peut implémenter une fonction quelconque à 6 entrées ou certaines jusqu'à 19 variables.

· Chaque *slice* contient une chaîne à *carry*, ce qui permet l'implémentation d'un *full adder* par LC.

On peut également utiliser ces chaînes pour réaliser des fonctions logiques plus larges

· L'élément de mémoire du LC peut être configuré comme une bascule ou comme un *latch*, avec CLK et EC, *set* et *reset* (synchrone ou asynchrone).

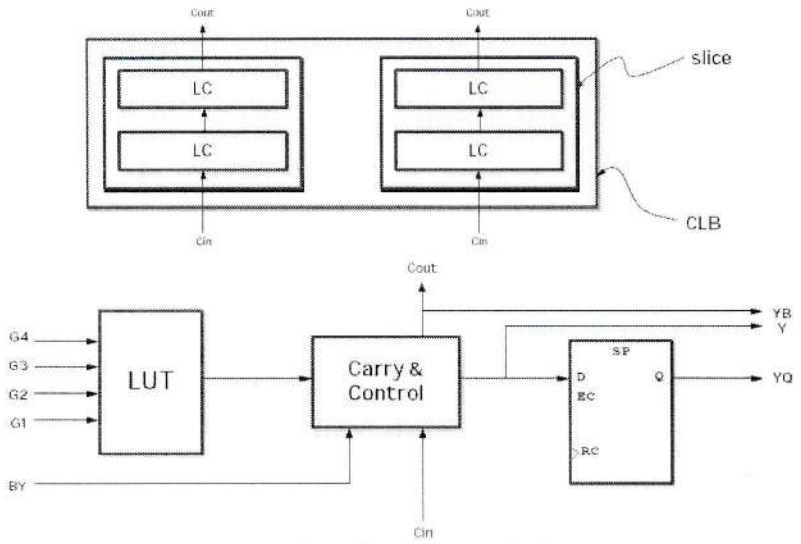


Figure 51. Bloc logique de base

·Chaque LUT (*Look-Up Table*) peut être utilisé comme une RAM 16x1 synchrone. Les deux LUTs d'un slice peuvent se combiner pour obtenir une RAM synchrone de dimension :

une RAM 16x2, une RAM 32x1, deux RAM 16x1(doubles ports) ou une RAM 16x1 et une fonction combinatoire à 4 variables.

· En plus, une LUT peut être utilisé comme registre à décalage

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

• Par circuit, il y a deux colonnes de mémoire RAM, appelée Block Select RAM. Une colonne est formée de plusieurs blocs, un par 4 CLB de hauteur (un FPGA avec 64 CLB de hauteur possède donc 16 blocs de mémoire par colonne, pour un total de 32 pour le circuit).

Input/Output Block (IOB)

La figure 52 présente la structure de ce bloc. Ces blocs entrée/sortie permettent l'interface entre les broches du composant FPGA et la logique interne développée à l'intérieur du composant. Ils sont présents sur toute la périphérie du circuit FPGA. Chaque bloc IOB contrôle une broche du composant et il peut être défini en entrée, en sortie, en signaux bidirectionnels ou être inutilisé (haute impédance).
Chaque IOB possède 3 éléments de mémoire, configurables comme bascules ou latches. Ces trois éléments partagent le signal d'horloge et de *set/reset*, mais chacun possède son propre *enable clock* (EC). Le signal *set/reset* peut être configuré comme *set* ou *reset*, synchrone ou asynchrone.

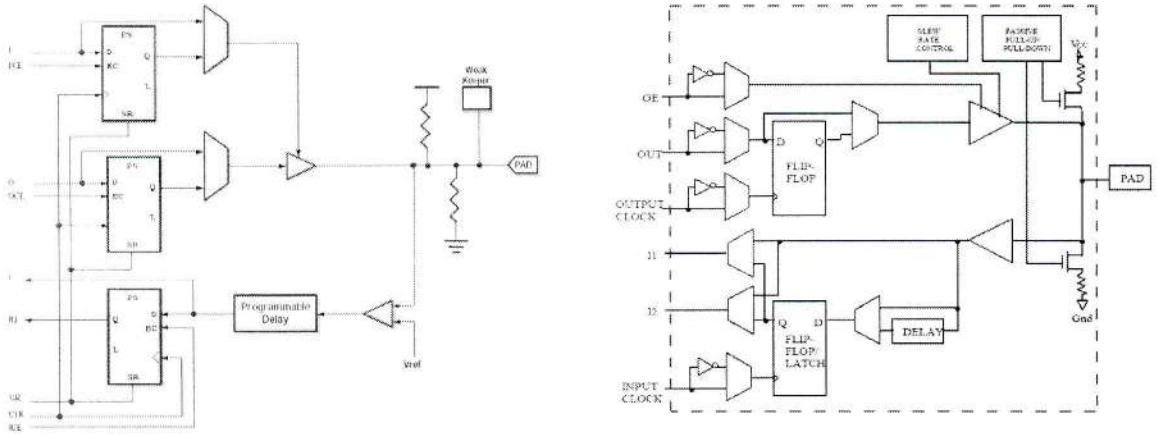


Figure 52. Cellule I/O (IOB)

Les différents types d'interconnexions :

Les connexions internes dans les circuits FPGA sont composées de segments métallisés. Parallèlement à ces lignes, nous trouvons des matrices programmables réparties sur la totalité du circuit, horizontalement et verticalement entre les divers CLB. Elles permettent les connexions entre les diverses lignes, celles-ci sont assurées par des transistors MOS dont l'état est contrôlé par des cellules de mémoire vive ou RAM. Le rôle de ces interconnexions est de relier avec un maximum d'efficacité les blocs logiques et les entrées/sorties. Il y a trois sortes d'interconnexions selon la longueur et la destination des liaisons.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

- d'interconnexions à usage général,
- d'interconnexions directes,
- de longues lignes.

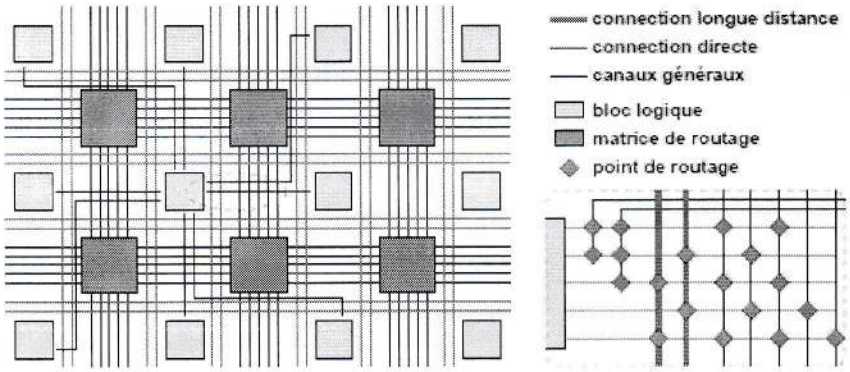


Figure 53. Structure générale du routage

Terminologie :

LE, LAB, ALM, slice, CLB Pour des raisons internes aux différents manufacturiers, plusieurs termes sont utilisés pour parler de l'architecture interne des FPGAs. Pour les FPGAs de la famille Cyclone, Altera utilise le terme :

Logic Element (LE) pour une cellule de base incluant une table de conversion, un additionneur et un registre.

Logic Array Bloc (LAB) regroupe dix LEs. Pour la famille Stratix, Altera a remplacé les LEs par des blocs plus complexes.

Adaptive Logic Modules (ALM). Un ALM comprend deux tables de conversion, deux additionneurs et deux registres. Pour la famille Stratix, un LAB regroupe 10 ALMs. Pour les FPGAs des familles Spartan et Virtex, Xilinx utilise le terme slice pour un module de base incluant deux tables de conversion, deux additionneurs et deux registres.

Configurable Logic Block (CLB) regroupe deux ou quatre slices, selon la famille de FPGA.

Blocs de mémoire intégrée

Les fabricants de FPGA ont commencé à intégrer des modules de plus en plus complexes. Les blocs de mémoire ont été parmi les premiers modules ajoutés à cause du grand besoin en mémoire de la plupart des applications. L'avantage important à intégrer des blocs de mémoire près de logique configurable est la réduction significative des délais de

propagation et la possibilité de créer des canaux de communication parallèle très larges. La figure ci-dessous illustre l'intégration de blocs de mémoire sous la forme d'une colonne entre les CLBs d'un FPGA.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

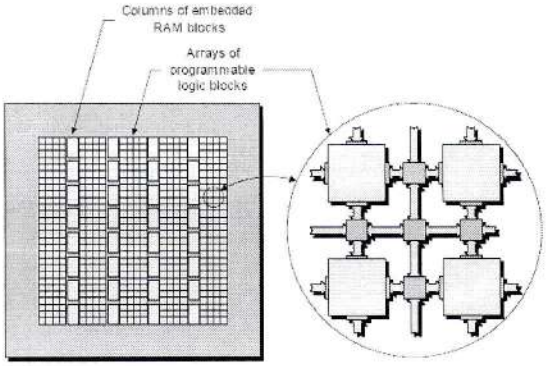


Figure 54. Mémoire RAM intégrée

La quantité de mémoire présente dans les blocs de RAM varie à travers les différentes familles de FPGAs, mais on peut retrouver jusqu'à 10 Méga bits de mémoire dans les plus gros et plus récents modèles. Les blocs peuvent être utilisés indépendamment ou en groupes, offrant une versatilité rarement rencontrée dans les systèmes numériques. De plus, les blocs de mémoire peuvent être utilisés pour implémenter des fonctions logiques, des machines à états, des registres à décalage très larges, etc.

Quelques fabricants de FPGA

Actel ; Altera ; AMD; Atmel; Cypress; Lattice; Lucent (AT&T); Philips ; Quicklogic ; Xilinx ; Zetex (FPGA analogique).

Exemples de constructeurs :

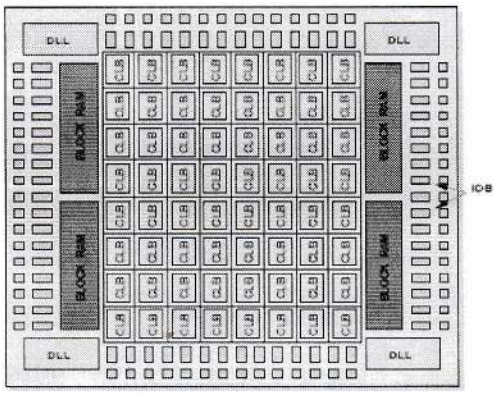


Figure 55. Spartan II E : vue globale

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

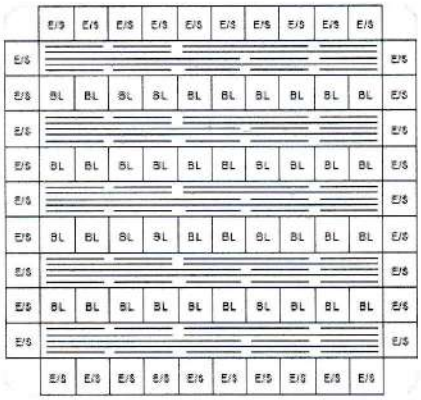


Figure 56. Architecture Actel de base

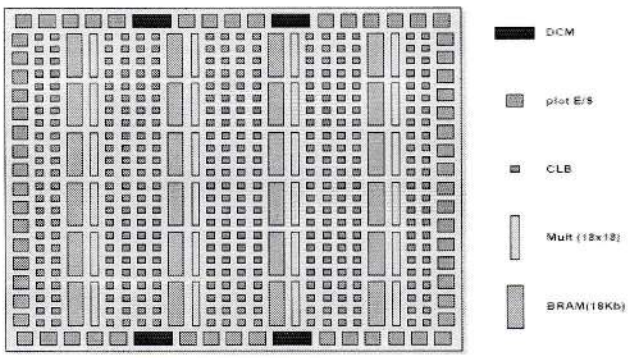


Figure 57. Xilinx Virtex II

Dans le plus gros Virtex II, il y a une matrice de 112x108 CLB, 168 multiplieurs, 168 memoires, 12 DCM. soit l'équivalent de 8 millions de portes logiques.

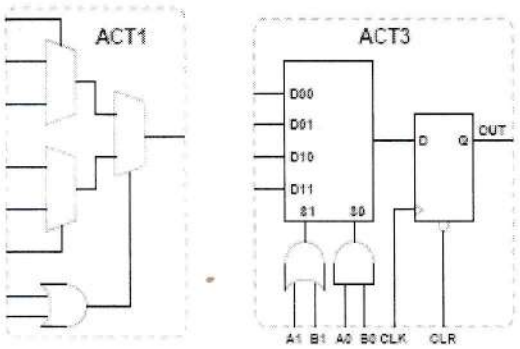
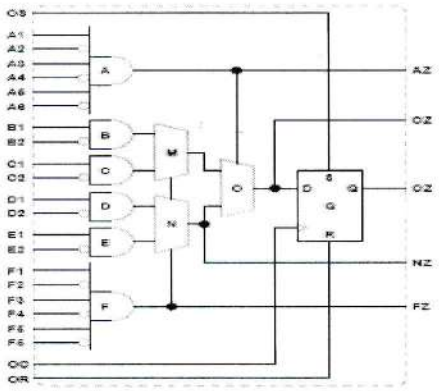


Figure 58. Blocs logiques Actel



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 59. Bloc logique Quicklogic

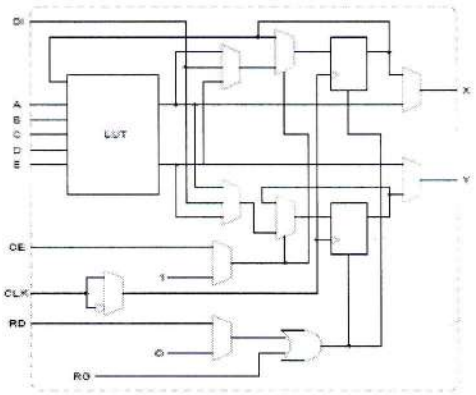


Figure 60. Bloc logique Xilinx Spartan II E

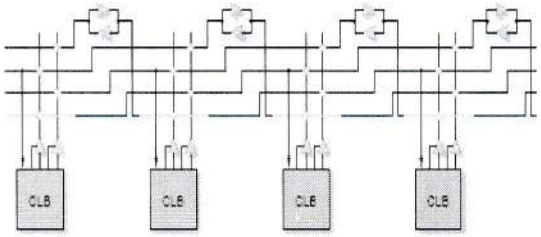


Figure 61. Bloc logique Xilinx 3000

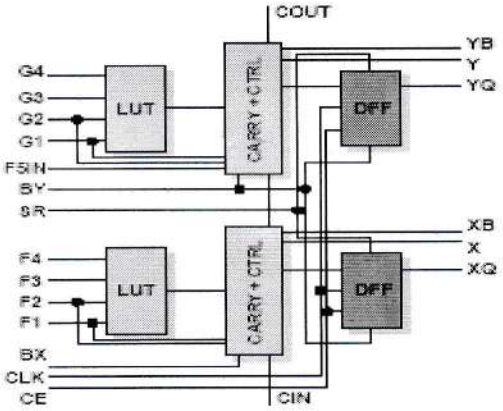


Figure 62. Routage (Xilinx Spartan II E)

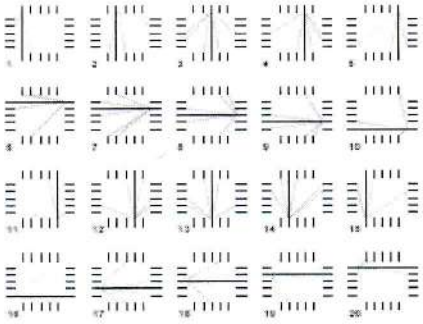
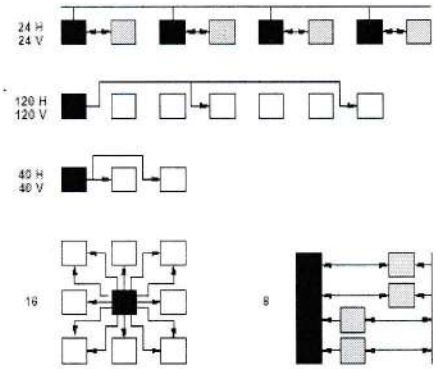


Figure 63. Routage (Xilinx 3000)

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم



المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Figure 64. Routage dans un Virtex II

Applications :

FPGA présents dans de plus en plus d'applications

Traitement et contrôle du signal

Télécommunications (Téléphones portables, GPS)

Jeux vidéos (GameCube, XBOX,..) Médical

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

4.1 Introduction

4.1.1 À propos de VHDL

VHDL est un langage de description de matériel. Il décrit le comportement d'un circuit ou d'un système électronique, à partir duquel le circuit ou le système physique peut ensuite être atteint (mis en œuvre).

VHDL signifie VHSIC Hardware Description Language. VHSIC est lui-même une abréviation de Very High Speed Integrated Circuits, une initiative financée par le département américain de la Défense dans les années 1980 qui a conduit à la création de VHDL.

Sa première version était VHDL 87, plus tard mis à niveau vers le soi-disant VHDL 93. VHDL était le premier langage de description de matériel à être normalisé par l'Institut des ingénieurs électriciens et électroniciens, via la norme IEEE 1076.

Une norme supplémentaire, l'IEEE 1164, a ensuite été ajoutée pour introduire un système logique à valeurs multiples.

VHDL est destiné à la synthèse de circuits ainsi qu'à la simulation de circuits. Cependant, bien que VHDL soit entièrement simulable, toutes les constructions ne sont pas synthétisables. Nous mettrons l'accent sur ceux qui le sont.

Une motivation fondamentale pour utiliser VHDL (ou son concurrent, Verilog) est que VHDL est un langage standard, indépendant de la technologie / du fournisseur, et est donc portable et réutilisable. Les deux principales applications immédiates du VHDL se situent dans le domaine des dispositifs logiques programmables (y compris les CPLD - dispositifs logiques programmables complexes et FPGA - matrices de portes programmables sur site) et dans le domaine des ASIC (circuits intégrés spécifiques à l'application). Une fois le code VHDL écrit, il peut être utilisé soit pour implémenter le circuit dans un dispositif programmable (d'Altera, Xilinx, Atmel, etc.) soit être soumis à une fonderie pour la fabrication d'une puce ASIC. Actuellement, de nombreuses puces commerciales complexes (microcontrôleurs, pour exemple) sont conçues selon une telle approche.

Une dernière remarque concernant VHDL est que, contrairement aux programmes informatiques classiques qui sont séquentiels, ses instructions sont intrinsèquement simultanées (parallèles). Pour cette raison, VHDL est généralement appelé un code plutôt qu'un programme. En VHDL, seules les instructions placées dans un PROCESS, FUNCTION ou PROCEDURE sont exécutées séquentiellement.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

4.1.2 Conception

Comme mentionné ci-dessus, l'un des principaux utilitaires du VHDL est qu'il permet la synthèse d'un circuit ou système dans un appareil programmable (PLD ou FPGA) ou dans un ASIC. Les étapes suivies au cours d'un tel projet sont résumées dans ce chapitre. Nous commençons la conception en écrivant le code VHDL, qui est enregistré dans un fichier avec l'extension (.vhd) et le même nom que le nom de son ENTITY. La première étape du processus de synthèse est la compilation. La compilation est la conversion du langage VHDL de haut niveau, qui décrit le circuit au niveau de transfert de registre (RTL), en une netlist au niveau de la porte. La deuxième étape est l'optimisation, qui est effectuée sur la netlist au niveau de la porte pour la vitesse ou pour la zone. À ce stade, la conception peut être simulée. Enfin, un logiciel de placement et de route (ajusteur) générera la disposition physique d'une puce PLD / FPGA ou générera les masques pour un ASIC.

4.1.3 Les outils EDA

Il existe plusieurs outils EDA (**E**lectronic **D**esign **A**utomation) (Automatisation de la conception électronique) disponibles pour la synthèse, la mise en œuvre et la simulation de circuits à l'aide de VHDL. Quelques outils (place et route, par exemple) sont proposés dans le cadre de la suite de conception d'un fournisseur (par exemple, le Quartus II d'Altera, qui permet la synthèse du code VHDL sur le CPLD / FPGA d'Altera ou la suite ISE de Xilinx pour les puces CPLD / FPGA de Xilinx). Autres outils (synthe

4.2 Structure du code

nous décrivons les sections fondamentales qui composent un morceau de code VHDL: les déclarations LIBRARY, ENTITY et ARCHITECTURE.

4.2.1 Unités VHDL fondamentales

Le code VHDL est composé d'au moins trois sections fondamentales :

LIBRARY : contient une liste de toutes les bibliothèques à utiliser dans la conception. Par exemple : ieee, std, work, etc.

ENTITY : spécifie les broches d'E / S du circuit.

ARCHITECTURE : contient le code VHDL proprement dit, qui décrit comment le circuit doit se comporter (fonction).

Une BIBLIOTHÈQUE est une collection de morceaux de code couramment utilisés. Placer ces morceaux à l'intérieur d'une bibliothèque leur permet d'être réutilisés ou partagés

par d'autres modèles. Le code est généralement écrit sous la forme de FONCTIONS, PROCÉDURES ou COMPOSANTS, qui sont placés à l'intérieur de PACKAGES, puis compilés dans la bibliothèque de destination.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

4.2.2 LIBRARY (bibliothèque)

Pour déclarer une BIBLIOTHÈQUE (c'est-à-dire pour la rendre visible à la conception), deux lignes de code sont nécessaires, l'une contenant le nom de la bibliothèque et l'autre une clause d'utilisation, comme indiqué dans la syntaxe ci-dessous.

```
LIBRARY library_name;
USE
library_name.package_name.package_parts;
```

Au moins trois paquets, provenant de trois bibliothèques déférentes, sont généralement nécessaires dans une conception :

- **ieee.std_logic_1164** (de la bibliothèque ieee),
- **standard** (à partir de la bibliothèque std), et
- **work** (bibliothèque de travail).

Leurs déclarations sont les suivantes :

```
LIBRARY ieee; -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all; -- the end of a statement or
LIBRARY std; -- declaration, while a double
USE std.standard.all; -- dash (--) indicates a comment.
LIBRARY work;
USE work.all;
```

Les bibliothèques *std* et *work* montrées ci-dessus sont rendues visibles par défaut, il n'est donc pas nécessaire de les déclarer ; seule la bibliothèque *ieee* doit être explicitement écrite. Cependant, ce dernier n'est nécessaire que lorsque le type de données STD_LOGIC (ou STD_ULOGIC) est utilisé dans la conception (les types de données seront étudiés en détail dans la section suivante).

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Le but des trois packages / bibliothèques mentionnés ci-dessus est le suivant : le package std_logic_1164 de la bibliothèque ieee spécifie un système logique à plusieurs niveaux ; std est une bibliothèque de ressources (types de données, entrées / sorties de texte, etc.) pour l'environnement de conception VHDL ; et la bibliothèque de travail est l'endroit où nous sauvegardons notre conception (le fichier .vhd, plus tous les fichiers créés par le compilateur, le simulateur, etc.).

En effet, la bibliothèque ieee contient plusieurs packages, dont les suivants :

- *std_logic_1164* : spécifie les systèmes logiques à valeurs multiples STD_LOGIC (8 niveaux) et STD_ULOGIC (9 niveaux).
- *std_logic_arith* : spécifie les types de données SIGNÉ et NON SIGNÉ et les opérations arithmétiques et de comparaison associées. Il contient également plusieurs fonctions de conversion de données, qui permettent de convertir un type en un autre : conv_integer (p), conv_unsigned (p, b), conv_signed (p, b), conv_std_logic_vector (p, b).
- *std_logic_signed* : contient des fonctions qui permettent d'effectuer des opérations avec des données STD_LOGIC_VECTOR comme si les données étaient de type SIGNED.
- *std_logic_unsigned* : contient des fonctions qui permettent d'effectuer des opérations avec des données STD_LOGIC_VECTOR comme si les données étaient de type UNSIGNED.

4.2.3 ENTITY (entité)

Une ENTITY est une liste avec les spécifications de toutes les broches d'entrée et de sortie (PORTS) du circuit. Sa syntaxe est indiquée ci-dessous.

```
ENTITY nom_de_l'entité IS
PORT (
nom_du_port : mode_de_signal type_de_signal ;
nom_du_port : mode_de_signal type_de_signal ;
...);
END nom_de_l'entité;
```

Le mode du signal peut être IN, OUT, INOUT ou BUFFER comme illustré dans la figure 65, IN et OUT sont vraiment des broches unidirectionnelles, tandis que INOUT est bidirectionnel. BUFFER, en revanche, est utilisé lorsque le signal de sortie doit être utilisé (lu) en interne. Le type de signal peut être BIT, STD_LOGIC, INTEGER, etc. Les types de

données seront décrits en détail à la section prochaine. Enfin, le nom de l'entité peut être fondamentalement n'importe quel nom, à l'exception des mots réservés VHDL (les mots réservés VHDL sont répertoriés dans l'annexe E). Exemple : Considérons la porte NAND de la figure 66 son ENTITY peut être spécifié comme :

```
ENTITY porte_nand IS
PORT (a, b : IN BIT;
x : OUT BIT);
END porte_nand;
```

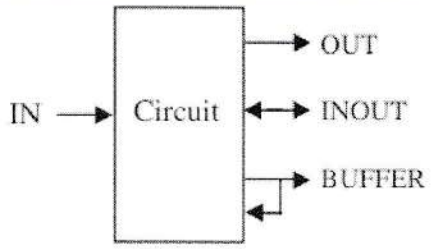


Figure 65. Signal BUFFER

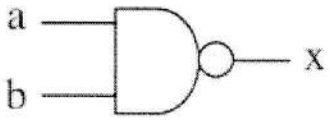


Figure 66. Porte NAND

La signification de ENTITY ci-dessus est la suivante : le circuit a trois broches d'E / S, soit deux entrées (a et b, mode IN) et une sortie (x, mode OUT). Les trois signaux sont de type BIT. Le nom choisi pour l'entité était porte_nand.

4.2.4 ARCHITECTURE

L'ARCHITECTURE est une description du comportement (fonction) du circuit.

Sa syntaxe est la suivante :

```
ARCHITECTURE nom_de_l'architecture OF nom_de_l'entité IS
[declarations]
BEGIN
(code)
END nom_de_l'architecture;
```

Comme indiqué ci-dessus, une architecture comporte deux parties : une partie déclarative (facultative), où les signaux et les constantes (entre autres) sont déclarés, et la partie code (de BEGIN vers le bas). Comme dans le cas d'une entité, le nom d'une architecture peut être fondamentalement n'importe quel nom (à l'exception des mots réservés VHDL), y compris le même nom que celui de l'entité.

Exemple : Considérons à nouveau la porte NAND de la figure 66.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

```
ARCHITECTURE mon_arch OF porte_nand IS
BEGIN
x <= a NAND b;
END mon_arch;
```

La signification de l'ARCHITECTURE ci-dessus est la suivante : le circuit doit effectuer l'opération NAND entre les deux signaux d'entrée (a, b) et affecter (<=) le résultat à la broche de sortie (x). Le nom choisi pour cette architecture était mon_arch.

Dans cet exemple, il n'y a pas de partie déclarative et le code ne contient qu'une seule affectation.

4.2.5 Exemples

Dans cette section, nous présenterons deux premiers exemples de code VHDL. Bien que nous n'ayons pas encore étudié les constructions qui apparaissent dans les exemples, elles aideront à illustrer les aspects fondamentaux concernant la structure globale du code. Chaque exemple est suivi par des commentaires explicatifs et des résultats de simulation.

- **Exemple 1 : Bascule D avec réinitialisation asynchrone**

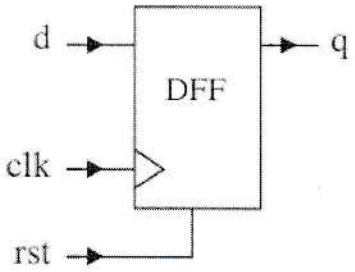


Figure 67. Bascule D

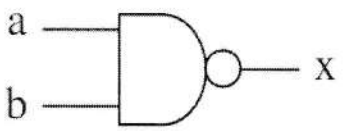


Figure 68. Porte NAND

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد الحقوق

La figure 67 montre le schéma d'une bascule de type D (DFF), déclenchée au front montant du signal d'horloge (clk), et avec une entrée de réinitialisation asynchrone (rst). Lorsque rst='1', la sortie doit être mise au niveau bas, quel que soit clk. Sinon, la sortie doit copier l'entrée (c'est-à-dire q <= d) au moment où clk passe de «0» à «1» (c'est-à-dire lorsqu'un événement ascendant se produit sur clk).

Il existe plusieurs manières de mettre en œuvre le DFF de la figure 67, l'une étant la solution présentée ci-dessous. Une chose à retenir, cependant, est que VHDL est intrinsèquement simultané (contrairement aux programmes informatiques classiques, qui sont séquentiels), donc pour implémenter un circuit cadencé (bascules, par exemple), nous devons `` forcer '' VHDL à être séquentiel. Cela peut être fait à l'aide d'un PROCESSUS, comme indiqué ci-dessous.

```
1 ---------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY bascule_D IS
6 PORT ( d, clk, rst: IN STD_LOGIC;
7 q: OUT STD_LOGIC);
8 END bascule_D;
9 ---------------------------------------
10 ARCHITECTURE behavior OF bascule_D IS
11 BEGIN
12 PROCESS (rst, clk)
13 BEGIN
14 IF (rst='1') THEN
15 q <= '0';
16 ELSIF (clk'EVENT AND clk='1') THEN
17 q <= d;
18 END IF;
19 END PROCESS;
20 END behavior;
21 ---------------------------------------
```

Commentaires :

Lignes 2-3 : Déclaration de la bibliothèque (nom de la bibliothèque et clause d'utilisation de la bibliothèque). Rappelons que les deux autres bibliothèques indispensables (*std* et *work*) sont rendues visibles par défaut.

Lignes 5 à 8 : Entité bascule_D

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Lignes 10–20 : Comportement de l'architecture.

Ligne 6 : ports d'entrée (le mode d'entrée ne peut être que IN). Dans cet exemple, tous les signaux d'entrée sont de type STD_LOGIC.

Ligne 7 : port de sortie (le mode de sortie peut être OUT, INOUT ou BUFFER). Ici la sortie est également de type STD_LOGIC.

Lignes 11–19 : partie de code de l'architecture (à partir du mot BEGIN).

Lignes 12–19 : Un PROCESS (à l'intérieur, le code est exécuté séquentiellement).

Ligne 12 : Le PROCESSUS est exécuté à chaque fois qu'un signal déclaré dans sa liste de sensibilité changements. Dans cet exemple, chaque fois que rst ou clk change, le PROCESSUS est exécuté.

Lignes 14 à 15 : chaque fois que rst passe à «1», la sortie est réinitialisée, quel que soit clk (réinitialisation asynchrone).

Lignes 16–17 : Si rst n'est pas actif, plus clk a changé (un EVENT s'est produit sur clk), plus un tel événement était un front montant (clk = «1»), alors le signal d'entrée (d) est stocké dans le bascule (q <= d).

Lignes 15 et 17 : L'opérateur " <= " est utilisé pour attribuer une valeur à un SIGNAL. Dans contraste, " : = " serait utilisé pour une VARIABLE. Tous les ports d'une entité sont des signaux par défaut.

Lignes 1, 4, 9 et 21 : commentées (rappelez-vous que «--» indique un commentaire). Utilisé seulement pour mieux organiser la conception.

Remarque : VHDL n'est pas sensible à la casse.

- **Exemple 2 : Bascule D et porte NAND**

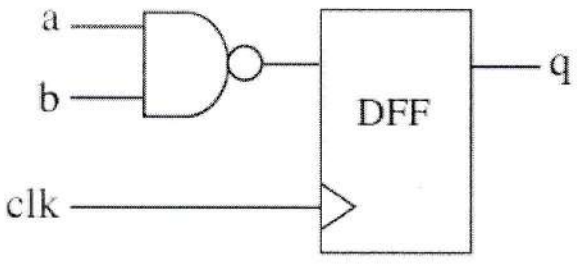


Figure 69. Bascule D avec porte NAND

Le circuit de la figure 68 était purement combinatoire, tandis que celui de la figure 67 était purement séquentiel. Le circuit de la figure 69 est un mélange des deux (sans

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

réinitialisation). Dans la solution qui suit, nous avons volontairement introduit un signal inutile (temp), juste pour illustrer comment un signal doit être déclaré.

```
1 ---------------------------------------
2 ENTITY example IS
3 PORT ( a, b, clk: IN BIT;
4 q: OUT BIT);
5 END example;
6 ---------------------------------------
7 ARCHITECTURE example OF example IS
8 SIGNAL temp : BIT;
9 BEGIN
10 temp <= a NAND b;
11 PROCESS (clk)
12 BEGIN
13 IF (clk'EVENT AND clk='1') THEN q<=temp;
14 END IF;
15 END PROCESS;
16 END example;
17 ---------------------------------------
```

Commentaires :

Les déclarations de bibliothèque ne sont pas nécessaires dans ce cas, car les données sont de type BIT, qui est spécifié dans la bibliothèque std (rappelez-vous que les bibliothèques *std* et *work* sont faites visible par défaut).

Lignes 2 à 5 : exemple d'entité.

Lignes 7-16 : exemple d'architecture.

Ligne 3 : ports d'entrée (tous de type BIT).

Ligne 4 : port de sortie (également de type BIT).

Ligne 8 : Partie déclarative de l'architecture (facultative). La température du signal, de type BIT, a été déclaré. Notez qu'il n'y a pas de déclaration de mode (le mode n'est utilisé que dans les entités).

Lignes 9 à 15 : partie de code de l'architecture (à partir du mot BEGIN).

Lignes 11-15 : UN PROCESS (instructions séquentielles exécutées chaque fois que le signal clk change).

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Lignes 10 et 11-15 : Bien que dans un processus l'exécution soit séquentielle, le processus, dans son ensemble, est concurrençant les autres instructions (externes); ainsi la ligne 10 est exécutée en même temps que le bloc 11-15.
Ligne 10 : opération NAND logique. Le résultat est affecté à la température du signal.
Lignes 13 à 14 : instruction IF. Au front montant de clk, la valeur de temp est affectée à q.
Lignes 10 et 13 : L'opérateur " <¼ " est utilisé pour attribuer une valeur à un SIGNAL.
Dans contraste, " : ¼ " serait utilisé pour une VARIABLE.
Lignes 8 et 10 : peuvent être éliminées, en changeant « q <= a NAND b » à la ligne 13.
Lignes 1, 6 et 17 : commentées. Utilisé uniquement pour mieux organiser la conception.

4.3 Types de données

Afin d'écrire du code VHDL efficacement, il est essentiel de savoir quels types de données sont autorisés, et comment les spécifier et les utiliser. Dans ce qui suit, tous les types de données fondamentaux sont décrits, avec un accent particulier sur ceux qui sont synthétisables. Des discussions sur la compatibilité et la conversion des données sont également incluses.

4.3.1 Types de données prédéfinis

VHDL contient une série de types de données prédéfinis, spécifiés par les normes IEEE 1076 et IEEE 1164. Plus spécifiquement, de telles définitions de type de données peuvent être trouvées dans les packages / bibliothèques suivants :

- package *standard* de de la bibliothèque *std*: définit les types de données BIT, BOOLEAN, INTEGER et REAL.
- Package *std_logic_1164* de la bibliothèque *ieee*: Définit les types de données STD_LOGIC et STD_ULOGIC.
- Package *std_logic_arith* de la bibliothèque *ieee*: Définit les types de données SIGNED et UNSIGNED, ainsi que plusieurs fonctions de conversion de données, comme conv_integer (p), conv_unsigned (p, b), conv_signed (p, b) et conv_std_logic_vector (p, b).
- Packages *std_logic_signed* et *std_logic_unsigned* de la bibliothèque *ieee*: Contiennent des fonctions qui permettent d'effectuer des opérations avec des

données STD_LOGIC_VECTOR comme si les données étaient de type SIGNED ou UNSIGNED, respectivement.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Tous les types de données prédéfinis (spécifiés dans les packages / bibliothèques répertoriés ci-dessus) sont décrits ci-dessous.

- **BIT (et BIT_VECTOR) : logique à 2 niveaux («0», «1»).**

Exemples :

```
SIGNAL x: BIT;
-- x est déclaré comme un signal à un chiffre de type BIT.
SIGNAL y: BIT_VECTOR (3 DOWNTO 0);
-- y est un vecteur de 4 bits, le bit le plus à gauche étant
le MSB (bit le plus significatif).
SIGNAL w: BIT_VECTOR (0 À 7);
-- w est un vecteur 8 bits, le bit le plus à droite étant le
MSB (bit le plus significatif).
```

Sur la base des signaux ci-dessus, les attributions suivantes seraient légales (pour attribuer une valeur à un signal, l'opérateur " <= " doit être utilisé) :

```
x <= '1';
-- x is a single-bit signal (as specified above), whose value
is
-- '1'. Notice that single quotes (' ') are used for a single
bit.
y <= "0111";
-- y is a 4-bit signal (as specified above), whose value is
"0111"
-- (MSB='0'). Notice that double quotes (" ") are used for
-- vectors.
w <= "01110001";
-- w is an 8-bit signal, whose value is "01110001" (MSB='1').
```

- **STD_LOGIC (et STD_LOGIC_VECTOR) :**

Système logique à 8 valeurs introduit dans la norme IEEE 1164.

«X» Forcing Unknown (synthétisable inconnu)

«0» Forcing Low (logique synthétisable «1»)

«1» Forcing High (logique synthétisable «0»)

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

«Z» Haute impédance (tampon à trois états synthétisable)

«W» Faible inconnu

«L» Faible faible

«H» Faible élevé

"-" Je m'en fous

Exemples :

```
SIGNAL x: STD_LOGIC;
- x est déclaré comme un signal à un chiffre (scalaire) de
type STD_LOGIC.
SIGNAL y: STD_LOGIC_VECTOR (3 DOWNTO 0): = "0001";
- y est déclaré comme un vecteur de 4 bits, le bit le plus à
gauche étant
- le MSB. La valeur initiale (facultative) de y est "0001".
Remarquer
- que l'opérateur ": =" est utilisé pour établir la valeur
initiale.
```

La plupart des niveaux *std_logic* sont destinés à la simulation uniquement. Cependant, «0», «1» et «Z» sont synthétisables sans aucune restriction. En ce qui concerne les valeurs «faibles», elles sont résolues en faveur des valeurs de «forçage» dans les nœuds à commande multiple (voir tableau 3.1).

En effet, si deux signaux *std_logic* quelconques sont connectés au même nœud, alors les niveaux logiques conflictuels sont automatiquement résolus conformément au tableau 3.1.

- **STD_ULOGIC (STD_ULOGIC_VECTOR):**

Système logique à 9 niveaux introduit dans la norme IEEE 1164 ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', ' - '). En effet, le système STD_LOGIC décrit ci-dessus est un sous-type de STD_ULOGIC. Ce dernier comprend une valeur logique supplémentaire, «U», qui signifie non résolu. Ainsi, contrairement à STD_LOGIC, les niveaux logiques conflictuels ne sont pas automatiquement résolus ici, donc les fils de sortie ne doivent jamais être connectés ensemble directement. Cependant, si deux fils de sortie ne sont jamais supposés être connectés ensemble, ce système logique peut être utilisé pour détecter des erreurs de conception.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

- BOOLEAN : Vrai, Faux.
- INTEGER : entiers 32 bits (de 2 147 483 647 à 2 147 483 647).
- NATUREL : Entiers non négatifs (de 0 à þ2 147 483 647).
- REAL : nombres réels allant de 1.0E38 à þ1.0E38. Non synthétisable.

Littéraux physiques : Utilisés pour informer des quantités physiques, comme l'heure, la tension, etc. Utile dans les simulations. Non synthétisable.
Littéraux de caractères : caractère ASCII unique ou une chaîne de ces caractères. Pas synthétisable.

- **SIGNED et UNSIGNED :**

Types de données définis dans le package *std_logic_arith* de la bibliothèque *ieee*. Ils ont l'apparence de STD_LOGIC_VECTOR, mais acceptent les opérations arithmétiques, qui sont typiques des types de données INTEGER (SIGNED et UNSIGNED seront discutés en détail dans la section 3.6).
Exemples :

```
x0 <= '0'; -- valeur bit, std_logic ou std_ulogic '0'
x1 <= "00011111"; -- bit_vector, std_logic_vector,
- std_ulogic_vector, signé ou non signé
x2 <= "0001_1111"; -- le soulignement a permis de faciliter
la visualisation
x3 <= "101111" -- représentation binaire du nombre décimal 47
x4 <= B "101111" -- représentation binaire du nombre décimal
47
x5 <= O "57" -- représentation octale du nombre décimal 47
x6 <= X "2F" -- représentation hexadécimale du décimal 47
n <= 1 200; -- entier
m <= 1_200; -- entier, trait de soulignement autorisé
IF ready THEN ... -- Booléen, exécuté si prêt = TRUE
y <= 1,2E-5; -- réel, non synthétisable
q <= d après 10 ns; -- physique, non synthétisable
```

Exemple : Opérations légales et illégales entre des données de différents types.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR (7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL e: ENTIER GAMME 0 À 255;
...
a <= b (5); -- légal (même type scalaire: BIT)
b (0) <= a; -- légal (même type scalaire: BIT)
c <= d (5); -- légal (même type scalaire: STD_LOGIC)
d (0) <= c; -- légal (même type scalaire: STD_LOGIC)
a <= c; -- illégal (non-concordance de type: BIT x STD_LOGIC)
b <= d; -- illégal (non-concordance de type: BIT_VECTOR x -
STD_LOGIC_VECTOR)
e <= b; -- illégal (non-concordance de type: INTEGER x
BIT_VECTOR)
e <= d; -- illégal (non-concordance de type: INTEGER x
        -- STD_LOGIC_VECTOR)
```

4.3.2 Types de données définis par l'utilisateur

VHDL permet également à l'utilisateur de définir ses propres types de données. Deux catégories de types de données définis par l'utilisateur sont présentées ci-dessous : entier et énuméré.

- Types d'entiers définis par l'utilisateur :

```
TYPE integer IS RANGE -2147483647 TO +2147483647;
-- This is indeed the pre-defined type INTEGER.
TYPE natural IS RANGE 0 TO +2147483647;
-- This is indeed the pre-defined type NATURAL.
TYPE my_integer IS RANGE -32 TO 32;
-- A user-defined subset of integers.
TYPE student_grade IS RANGE 0 TO 100;
-- A user-defined subset of integers or naturals.
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

- Types énumérés définis par l'utilisateur :

```
TYPE bit IS ('0', '1');
-- This is indeed the pre-defined type BIT
TYPE my_logic IS ('0', '1', 'Z');
-- A user-defined subset of std_logic.
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;
-- This is indeed the pre-defined type BIT_VECTOR.
-- RANGE <> is used to indicate that the range is
unconstrained.
-- NATURAL RANGE <>, on the other hand, indicates that the
only
-- restriction is that the range must fall within the NATURAL
-- range.
TYPE state IS (idle, forward, backward, stop);
-- An enumerated data type, typical of finite state machines.
TYPE color IS (red, green, blue, white);
-- Another enumerated data type.
```

4.3.3 Sous-types

Un Sous-types est un type avec une contrainte. La principale raison d'utiliser un sous-type plutôt que de spécifier un nouveau type est que, bien que les opérations entre des données de différents types ne soient pas autorisées, elles sont autorisées entre un sous-type et son type de base correspondant.

Exemples : Les sous-types ci-dessous sont dérivés des types présentés dans le précédent exemple.

```
SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER'HIGH;
-- As expected, NATURAL is a subtype (subset) of INTEGER.
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO 'Z';
-- Recall that STD_LOGIC=('X','0','1','Z','W','L','H','-').
-- Therefore, my_logic=('0','1','Z').
SUBTYPE my_color IS color RANGE red TO blue;
-- Since color=(red, green, blue, white), then
-- my_color=(red, green, blue).
SUBTYPE small_integer IS INTEGER RANGE -32 TO 32;
-- A subtype of INTEGER.
Example: Legal and illegal operations between types and
subtypes.
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO '1';
SIGNAL a: BIT;
SIGNAL b: STD_LOGIC;
SIGNAL c: my_logic;
...
b <= a; -- illegal (type mismatch: BIT versus STD_LOGIC)
b <= c; -- legal (same "base" type: STD_LOGIC)
```

4.3.4 Tableaux

Les tableaux sont des collections d'objets du même type. Ils peuvent être unidimensionnels (1D), bidimensionnels (2D) ou unidimensionnels par unidimensionnels (1Dx1D).

Ils peuvent également être de dimensions plus élevées, mais ils ne sont généralement pas synthétisables.

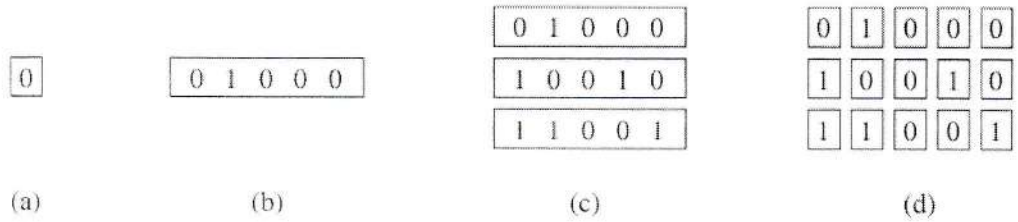


Figure 70. Construction de tableaux de données

La figure 70 illustre la construction de tableaux de données. Une valeur unique (scalaire) est affichée dans (a), un vecteur (tableau 1D) dans (b), un tableau de vecteurs (tableau 1Dx1D) dans (c) et un tableau de scalaires (tableau 2D) dans (d)

En effet, les types de données VHDL prédéfinis (vus dans la section 3.1) n'incluent que les catégories scalaire (bit unique) et vecteur (tableau unidimensionnel de bits). Les types synthétisables prédéfinis dans chacune de ces catégories sont les suivants :

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

- Scalaires : BIT, STD_LOGIC, STD_ULOGIC et BOOLEAN.
- Vecteurs: BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR,

INTEGER, SIGNED, and UNSIGNED.

Comme on peut le voir, il n'y a pas de tableaux 2D ou 1Dx1D prédéfinis, qui, si nécessaire, doivent être spécifiés par l'utilisateur. Pour ce faire, le nouveau TYPE doit d'abord être défini, puis le nouveau SIGNAL, VARIABLE ou CONSTANT peut être déclaré à l'aide de ce type de données. La syntaxe ci-dessous doit être utilisée.

Pour spécifier un nouveau type de tableau :

```
TYPE type_name IS ARRAY (specification) OF data_type;
```

Pour utiliser le nouveau type de tableau :

```
SIGNAL signal_name: type_name [:= initial_value];
```

4.4 Opérateurs et attributs

Le but de cette section, avec les sections précédentes, est de jeter les bases de base de VHDL, donc dans le prochain chapitre, nous pouvons commencer à traiter des conceptions de circuits réels. Il est en effet impossible - ou peu productif, du moins - d'écrire un code de manière efficace sans entreprendre d'abord le sacrifice de bien comprendre les types de données, les opérateurs et les attributs.

Les opérateurs et les attributs constituent une liste relativement longue des constructions VHDL généraux, qui sont souvent examinées que peu. Nous avons recueilli l'ensemble dans une section spécifique afin de fournir une vue complète et plus cohérente.

4.4.1 Opérateurs

VHDL fournit plusieurs types d'opérateurs prédéfinis :

- Opérateurs d'affectation
- Opérateurs logiques
- Opérateurs arithmétiques
- Opérateurs relationnels
- Opérateurs de décalage
- Opérateurs de concaténation

Chacune de ces catégories est décrite ci-dessous.

```
y <= NOT a AND b; -- (a'.b)
y <= NOT (a AND b); -- (a.b)'
y <= a NAND b; -- (a.b)'
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Elles sont :

<= Utilisé pour attribuer une valeur à un SIGNAL.

: = Utilisé pour affecter une valeur à une VARIABLE, CONSTANT ou GENERIC. Utilisé également pour établir les valeurs initiales.

=> Utilisé pour attribuer des valeurs à des éléments vectoriels individuels ou avec OTHERS.

Exemple : considérez les déclarations de signaux et de variables suivantes :

```
SIGNAL x : STD_LOGIC;
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNTO 0); -- Leftmost bit is
MSB
SIGNAL w: STD_LOGIC_VECTOR(0 TO 7); -- Rightmost bit is MSB
x <= '1'; -- '1' is assigned to SIGNAL x using "<="
y := "0000"; -- "0000" is assigned to VARIABLE y using ":="
w <= "10000000"; -- LSB is '1', the others are '0'
w <= (0 =>'1', OTHERS =>'0'); -- LSB is '1', the others are
'0'
```

- **Opérateurs logiques**

Utilisé pour effectuer des opérations logiques. Les données doivent être de type BIT, STD_LOGIC ou STD_ULOGIC (ou, bien entendu, leurs extensions respectives, BIT_VECTOR, STD_LOGIC_VECTOR ou STD_ULOGIC_VECTOR). Les opérateurs logiques sont :

- NOT
- AND
- OR
- NAND
- NOR
- XOR
- XNOR

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

- **Opérateurs arithmétiques**

Utilisé pour effectuer des opérations arithmétiques. Les données peuvent être de type INTEGER, SIGNED, UNSIGNED ou REAL (rappelez-vous que la dernière ne peut pas être synthétisée directement). De plus, si le package std_logic_signed ou Std_logic_unsigned de la bibliothèque ieee est utilisé, alors STD_LOGIC_VECTOR peut également être utilisé directement dans les opérations d'addition et de soustraction

- + Addition
- - Soustraction
- * Multiplication
- / Division
- ** Exponentiation
- MOD Module
- REM Reste
- ABS valeur absolue

Il n'y a pas de restrictions de synthèse concernant l'addition et la soustraction, et il en va généralement de même pour la multiplication. Pour la division, seule la puissance de deux diviseurs (opération de décalage) est autorisée. Pour l'exponentiation, seules les valeurs statiques de base et d'exposant sont acceptées. En ce qui concerne les opérateurs mod et rem, y mod x renvoie le reste de y / x avec le signal de x, tandis que y rem x renvoie le reste de y / x avec le signal de y. Enfin, abs renvoie la valeur absolue. En ce qui concerne les trois derniers opérateurs (mod, rem, abs), il y a généralement peu ou pas de support de synthèse.

- **Opérateurs de comparaison**

Utilisé pour faire des comparaisons. Les données peuvent être de l'un des types énumérés ci-dessus. Les opérateurs relationnels (de comparaison) sont :

- = Égal à
- /= Différent de
- < Inférieur à
- > Supérieur à
- <= Inférieur ou égal à

- <= Supérieur ou égal à

- **Attributs de données**

Les attributs de données prédéfinis et synthétisables sont les suivants :

- d'LOW : renvoie l'index du tableau le plus bas
- d'HIGH : renvoie l'index du tableau supérieur
- d'LEFT : renvoie l'index du tableau le plus à gauche
- d'RIGHT : renvoie l'index du tableau le plus à droite
- d'LENGTH : renvoie la taille du vecteur
- d'RANGE : renvoie la plage de vecteurs
- d'REVERSE_RANGE : renvoie la plage vectorielle dans l'ordre inverse

Exemple 1 : considérez le signal suivant :

Alors :

```
SIGNAL d : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Exemple 2 : considérez le signal suivant :

```
d'LOW=0, d'HIGH=7, d'LEFT=7, d'RIGHT=0, d'LENGTH=8,
d'RANGE=(7 downto 0), d'REVERSE_RANGE=(0 to 7).
```

Ensuite, les quatre instructions LOOP ci-dessous sont synthétisables et équivalentes.

```
SIGNAL x: STD_LOGIC_VECTOR (0 TO 7);
```

Si le signal est de type énuméré, alors :

```
FOR i IN RANGE (0 TO 7) LOOP ...
FOR i IN x'RANGE LOOP ...
FOR i IN RANGE (x'LOW TO x'HIGH) LOOP ...
FOR i IN RANGE (0 TO x'LENGTH-1) LOOP ...
```

- d'VAL (ligne, colonne): renvoie la valeur à la position spécifiée; etc.

Il existe peu ou pas de prise en charge de la synthèse pour les attributs de type de données énumérés.

- **Attributs de signal**

Considérons un signal s alors :

- s'EVENT : renvoie vrai lorsqu'un événement se produit sur s
- S'STABLE : Renvoie vrai si aucun événement ne s'est produit sur s
- s'ACTIVE : renvoie vrai si s = '1'
- S'QUIET <time> : Renvoie vrai si aucun événement ne s'est produit pendant le temps spécifié
- s'LAST_EVENT : Renvoie le temps écoulé depuis le dernier événement
- s'LAST_ACTIVE : Renvoie le temps écoulé depuis les dernières s ='1'
- S'LAST_VALUE : Renvoie la valeur de s avant le dernier événement ; etc.

Bien que la plupart des attributs de signal soient uniquement à des fins de simulation, les deux premiers de la liste ci-dessus sont synthétisables, s'EVENT étant le plus souvent utilisé de tous.

Exemple : les quatre affectations ci-dessous sont synthétisables et équivalentes. Ils renvoient TRUE lorsqu'un événement (un changement) se produit sur clk, ET si cet événement est ascendant (en d'autres termes, lorsqu'un front montant se produit sur clk).

```
IF (clk'EVENT AND clk='1')...          -- EVENT attribute used
                                       -- with IF
IF (NOT clk'STABLE AND clk='1')...     -- STABLE attribute used
                                       -- with IF
WAIT UNTIL (clk'EVENT AND clk='1');    -- EVENT attribute used
                                       -- with WAIT
IF RISING_EDGE(clk)...                 -- call to a function
```

4.4.2 Attributs définis par l'utilisateur

Nous avons vu ci-dessus les attributs de type HIGH, RANGE, EVENT, etc.

Ceux-ci sont tous prédéfinis dans VHDL. Cependant, VHDL permet également la construction d'attributs définis par l'utilisateur.

Pour utiliser un attribut défini par l'utilisateur, il doit être déclaré et spécifié. La syntaxe est la suivante :

Déclaration d'attribut :

4.5.1 Concurrent versus séquentiel

Nous commençons ce chapitre en passant en revue les différences fondamentales entre la logique combinatoire et la logique séquentielle, et en les opposants aux différences entre le code concurrent et le code séquentiel.

المركز الجامعي نور البشير البيض
المجلس العلمي
معهد العلوم

- **Logique combinatoire vs séquentielle**

Par définition, la logique combinatoire est celle dans laquelle la sortie du circuit dépend uniquement des entrées de courant figure 71. Il est alors clair que, en principe, le système ne nécessite aucune mémoire et peut être implémenté à l'aide de portes logiques classiques. En revanche, la logique séquentielle est définie comme celle dans laquelle la sortie dépend des entrées précédentes figure 72. Par conséquent, des éléments de stockage sont nécessaires, qui sont connectés au bloc logique combinatoire via une boucle de rétroaction, de sorte que maintenant les états stockés (créés par les entrées précédentes) affecteront également la sortie du circuit. Une erreur courante est de penser que tout circuit qui possède des éléments de stockage (bascules) est séquentiel. Une RAM (Random Access Memory) est un exemple. Une RAM peut être modélisée comme dans les figures 71 et 72. Notez que les éléments de stockage apparaissent dans un chemin avant plutôt que dans une boucle de rétroaction. L'opération de lecture en mémoire ne dépend que du vecteur d'adresse actuellement appliqué à l'entrée RAM, la valeur récupérée n'ayant rien à voir avec les accès mémoire précédents.

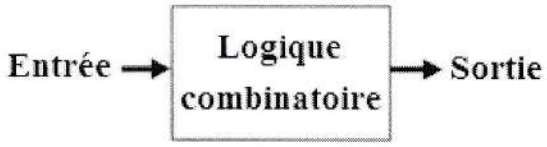


Figure 71. Schéma synoptique de la logique combinatoire

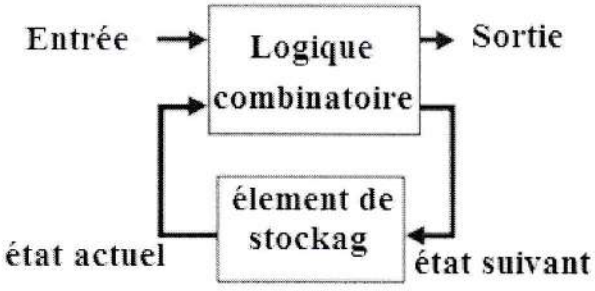


Figure 72. Schéma synoptique de la logique séquentiel

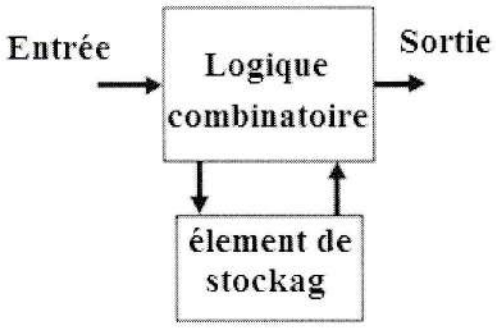


Figure 73. Schéma synoptique de la logique combinatoire (RAM)

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

- **Programmation concurrente et séquentiel**

Le code VHDL est intrinsèquement concurrent (parallèle). Seules les instructions placées dans un PROCESS, FUNCTION ou PROCEDURE sont séquentielles. Pourtant, bien que dans ces blocs l'exécution soit séquentielle, le bloc, dans son ensemble, est concurrençant toutes les autres instructions (externes). Le code simultané est également appelé code de flux de données.

À titre d'exemple, considérons un code avec trois instructions simultanées (stat1, stat2, stat3). Ensuite, l'une des alternatives ci-dessous rendra le même circuit physique :

stat1 stat3 stat1
stat2 ≡ stat2 ≡ stat3 ≡ etc.
stat3 stat1 stat2

Il est alors clair que, puisque l'ordre n'a pas d'importance, le code purement concurrent ne peut pas être utilisé pour implémenter des circuits synchrones (la seule exception est quand un GUARDED BLOCK est utilisé). En d'autres termes, en général, nous ne pouvons construire que des circuits de logique combinatoire avec du code concurrent. Pour obtenir des circuits logiques séquentiels, un code séquentiel doit être utilisé. En effet, avec ce dernier, nous pouvons mettre en œuvre à la fois des circuits séquentiels et combinatoires. Nous discuterons du code concurrent, c'est-à-dire que nous étudierons les instructions qui ne peuvent être utilisées qu'en dehors des PROCESSUS, FUNCTIONS ou PROCEDURES. Il s'agit de l'instruction WHEN et de l'instruction GENERATE. Outre eux, des affectations utilisant uniquement des opérateurs (logiques, arithmétiques, etc.) peuvent évidemment également être utilisées pour créer des circuits combinatoires. Enfin, un type spécial d'instruction, appelé BLOCK, peut également être utilisé.

En résumé, dans le code simultané, les éléments suivants peuvent être utilisés :

- ✓ Les opérateurs;
- ✓ L'instruction WHEN (WHEN / ELSE ou WITH / SELECT / WHEN);
- ✓ L'instruction GENERATE;
- ✓ L'instruction BLOCK.

Chacun de ces cas est décrit ci-dessous :

4.5.2 Utilisation des opérateurs

Il s'agit de la manière la plus élémentaire de créer du code simultané. Opérateurs (AND, OR, +,-, *, sll, sra, etc.) ont été résumé dans le tableau 5 ci-dessous.

Les opérateurs peuvent être utilisés pour implémenter n'importe quel circuit combinatoire. Cependant, comme cela apparaîtra plus tard, les circuits complexes sont généralement plus faciles à écrire en utilisant un code séquentiel, même si le circuit ne contient pas de logique séquentielle. Dans l'exemple qui suit, une conception utilisant uniquement des opérateurs logiques est présentée.

Type d'opérateur	Opérateurs	Types de données
Logique	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmétique	+, -, *, /, ** (mod, rem, abs)	INTEGER, SIGNED, UNSIGNED
Comparaison	=, /=, <, >, <=, >=	
Changement	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Enchaînement	&, (,,,)	Identique aux opérateurs logiques, plus SIGNED et UNSIGNED

Table 5. Differents types d'operateur

المركز الجامعي البيض
المجلس العلمي
معهد العلوم

Exemple 4.1 : Multiplexeur

La figure 74 montre un multiplexeur à 4 entrées, un bit par entrée. La sortie doit être égale à l'entrée sélectionnée par les bits de sélection, s1-s0.

Son implémentation, en utilisant uniquement des opérateurs logiques, peut se faire comme suit :

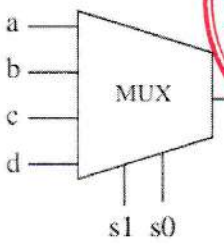


Figure 74. Multiplexeur 4x1

```
1 ---------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY mux IS
6 PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7 y: OUT STD_LOGIC);
8 END mux;
9 ---------------------------------------
10 ARCHITECTURE pure_logic OF mux IS
11 BEGIN
12 y <= (a AND NOT s1 AND NOT s0) OR
13 (b AND NOT s1 AND s0) OR
14 (c AND s1 AND NOT s0) OR
15 (d AND s1 AND s0);
16 END pure_logic;
17 ---------------------------------------
```

4.5.3 WHEN (simple et sélectionné)

Comme mentionné ci-dessus, WHEN est l'une des instructions concurrentes fondamentales (avec les opérateurs et GENERATE). Il apparaît sous deux formes : WHEN / ELSE (simple WHEN) et WITH / SELECT / WHEN (sélectionné WHEN). Sa syntaxe est affichée au-dessous :

- **WHEN / ELSE :**

```
affectation WHEN condition ELSE
affectation WHEN condition ELSE
...;
```

- **WITH / SELECT / WHEN :**

```
WITH identifiant SELECT
affectation WHEN valeur,
affectation WHEN valeur,
...;
```

Chaque fois que WITH / SELECT / WHEN est utilisé, toutes les permutations doivent être testées, le mot-clé OTHERS est donc souvent utile. Un autre mot clé important est UNAFFECTED, qui doit être utilisé lorsqu'aucune action ne doit avoir lieu.

Exemple :

```
------ With WHEN/ELSE -------------------------
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
        "001" WHEN ctl='1' ELSE
        "010";
---- With WITH/SELECT/WHEN --------------------
WITH control SELECT
output <= "000" WHEN reset,
          "111" WHEN set,
UNAFFECTED WHEN OTHERS;

-----------------------------------------------
```

Un autre aspect important lié à l'instruction WHEN est que la «WHEN valeur» indiquée dans la syntaxe ci-dessus peut en effet prendre trois formes :

```
WHEN value                  -- valeur unique
WHEN value1 to value2       -- plage, pour les types de données
                            -- énumérés uniquement
WHEN value1 | value2 |...   - valeur 1 ou valeur 2 ou ...
```

Exemple 4.2 : Multiplexer 2

Cet exemple montre l'implémentation du même multiplexeur de l'exemple 4.1, mais avec une représentation légèrement différente pour l'entrée sel figure 75. Cependant, dans celui-ci WHEN a été utilisé au lieu d'opérateurs logiques. Deux solutions sont présentées : l'une utilisant WHEN / ELSE (simple WHEN) et l'autre avec WITH / SELECT / WHEN (sélectionné WHEN).

```
1 ------- Solution 1: with WHEN/ELSE ------
--
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
--
5 ENTITY mux IS
6 PORT ( a, b, c, d: IN STD_LOGIC;
7 sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8 y: OUT STD_LOGIC);
9 END mux;
10 ---------------------------------------
---
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13 y <= a WHEN sel="00" ELSE
14 b WHEN sel="01" ELSE
15 c WHEN sel="10" ELSE
16 d;
17 END mux1;
18 ---------------------------------------
---
1 --- Solution 2: with WITH/SELECT/WHEN ---
--
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
--
5 ENTITY mux IS
6 PORT ( a, b, c, d: IN STD_LOGIC;
7 sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8 y: OUT STD_LOGIC);
9 END mux;
10 ---------------------------------------
---
11 ARCHITECTURE mux2 OF mux IS
12 BEGIN
13 WITH sel SELECT
14 y <= a WHEN "00", -- notice "," instead
of ";"
15 b WHEN "01",
16 c WHEN "10"
```

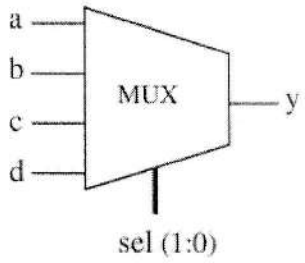


Figure 75. MUX 2b

Dans les solutions ci-dessus, sel aurait pu être déclaré comme entier (INTEGER), auquel cas le code serait le suivant :

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

```
1 ------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY mux IS
6 PORT ( a, b, c, d: IN STD_LOGIC;
7 sel: IN INTEGER RANGE 0 TO 3;
8 y: OUT STD_LOGIC);
9 END mux;
10 ---- Solution 1: with WHEN/ELSE ---------------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13 y <= a WHEN sel=0 ELSE
14 b WHEN sel=1 ELSE
15 c WHEN sel=2 ELSE
16 d;
17 END mux1;
18 -- Solution 2: with WITH/SELECT/WHEN --------
19 ARCHITECTURE mux2 OF mux IS
20 BEGIN
21 WITH sel SELECT
22 y <= a WHEN 0,
23 b WHEN 1,
24 c WHEN 2,
25 d WHEN 3; -- here, 3 or OTHERS are equivalent,
26 END mux2; -- for all options are tested anyway
27 ------------------------------------------------
```

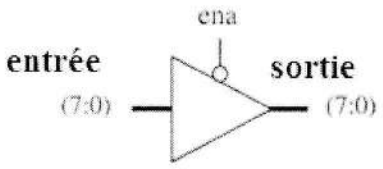


Figure 76. Tampon à trois états

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Exemple 5.3 : Tampon à trois états

Ceci est un autre exemple qui illustre l'utilisation de WHEN. Le tampon à 3 états de la figure 75 doit fournir une sortie ¼ d'entrée lorsque ena (activer) est à l'état bas, ou une sortie = «« ZZZZZZZZ » (haute impédance) dans le cas contraire.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ---------------------------------------------
4 ENTITY tri_state IS
5 PORT ( ena: IN STD_LOGIC;
6 input: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7 output: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
8 END tri_state;
9 ---------------------------------------------
10 ARCHITECTURE tri_state OF tri_state IS
11 BEGIN
12 output <= input WHEN (ena='0') ELSE
13 (OTHERS => 'Z');
14 END tri_state;
15 ---------------------------------------------
```

4.5.4 GENERATE

GENERATE est une autre instruction simultanée (avec les opérateurs et WHEN). Elle équivaut à l'instruction séquentielle LOOP en ce sens qu'elle permet à une section de code d'être répétée un certain nombre de fois, créant ainsi plusieurs instances des mêmes affectations. Sa forme régulière est la construction FOR / GENERATE, avec la syntaxe indiquée ci-dessous. Notez que GENERATE doit être étiqueté.

- FOR / GENERATE :

```
label: FOR identifier IN range GENERATE
(concurrent assignments)
END GENERATE;
```

Une forme irrégulière est également disponible, qui utilise IF/GENERATE (avec un équivalent IF; rappelez-vous qu'à l'origine IF est une instruction séquentielle). Ici, ELSE n'est pas autorisé. De la même manière que IF / GENERATE peut être imbriqué dans FOR/GENERATE (syntaxe ci-dessous), l'inverse peut également être fait.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

- IF / GENERATE imbriqué dans FOR / GENERATE:

```
label1: FOR identifier IN range GENERATE
...
label2: IF condition GENERATE
(concurrent assignments)
END GENERATE;
...
END GENERATE;
```

Exemple :

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (7 DOWNTO 0);
...
G1: FOR i IN x'RANGE GENERATE
z(i) <= x(i) AND y(i+8);
END GENERATE;
```

Une remarque importante à propos de GENERATE est que les deux limites de la plage doivent être statiques.

À titre d'exemple, considérons le code ci-dessous, où le choix est un paramètre d'entrée (non statique). Ce type de code généralement n'est pas synthétisable.

4.5.5 BLOCK

Il existe deux types d'instructions BLOCK : simples et sécurisées.

- **BLOCK simple**

L'instruction BLOCK, dans sa forme simple, ne représente qu'un moyen de partitionner localement le code. Il permet à un ensemble d'instructions simultanées d'être regroupées en un BLOC, dans le but de rendre le code global plus lisible et plus gérable (ce qui peut être utile lorsqu'il s'agit de codes longs). Sa syntaxe est indiquée ci-dessous.

```
label: BLOCK
[declarative part]
BEGIN
(concurrent statements)
END BLOCK label;
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Par conséquent, l'aspect général d'un code «blocked» est le suivant :

```
------------------------
ARCHITECTURE example ...
BEGIN
...
block1: BLOCK
BEGIN
...
END BLOCK block1
...
block2: BLOCK
BEGIN
...
END BLOCK block2;
...
END example;
------------------------
```

Exemple :

```
b1: BLOCK
SIGNAL a: STD_LOGIC;
BEGIN
a <= input_sig WHEN ena='1' ELSE 'Z';
END BLOCK b1;
```

Un BLOC (simple ou protégé) peut être imbriqué dans un autre BLOCK. La syntaxe correspondante est indiquée ci-dessous.

```
label1: BLOCK
[declarative part of top block]
BEGIN
[concurrent statements of top block]
label2: BLOCK
[declarative part nested block]
BEGIN
(concurrent statements of nested block)
END BLOCK label2;
[more concurrent statements of top block]
END BLOCK label1;
```

- **BLOCK sécurisés (protégé).**

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Un BLOCK protégé (sécurisé) est un type spécial de BLOCK, qui comprend une expression supplémentaire, appelée expression de garde. Une instruction protégée dans un BLOCK protégé est exécutée uniquement lorsque l'expression de garde est TRUE.

```
label: BLOCK (guard expression)
[declarative part]
BEGIN
(concurrent guarded and unguarded statements)
END BLOCK label;
```

Comme l'illustrent les exemples ci-dessous, même si seules des instructions concurrentes peuvent être écrites dans un BLOCK, avec un BLOCK protégé, même des circuits séquentiels peuvent être construits. Ceci, cependant, n'est pas une approche de conception habituelle.

Exemple 5.7 : Verrou implémenté avec un BLOC protégé

L'exemple présenté ci-dessous implémente un verrou transparent. Dans ce document, clk = '1' (ligne 12) est l'expression de garde, tandis que q <= GUARDED d (ligne 14) est une instruction gardée. Par conséquent, q <= d ne se produira que si clk = '1'.

```
1  --------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------
5  ENTITY latch IS
6  PORT (d, clk: IN STD_LOGIC;
7  q: OUT STD_LOGIC);
8  END latch;
9  --------------------------------
10 ARCHITECTURE latch OF latch IS
11 BEGIN
12 b1: BLOCK (clk='1')
13 BEGIN
14 q <= GUARDED d;
15 END BLOCK b1;
16 END latch;
17 --------------------------------
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Ici, une bascule de type D sensible au front positif, avec réinitialisation synchrone, est conçue. L'interprétation du code est similaire à celle de l'exemple ci-dessus. Dans celui-ci, clk'EVENT AND clk = '1' (ligne 12) est l'expression de garde, tandis que q <= GUARDED '0' WHEN rst = '1' (ligne 14) est une instruction protégée. Par conséquent, q <= "0" se produira lorsque l'expression de garde est vraie et que rst est "1".

```
1 -------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -------------------------------
5 ENTITY dff IS
6 PORT ( d, clk, rst: IN STD_LOGIC;
7 q: OUT STD_LOGIC);
8 END dff;
9 -------------------------------
10 ARCHITECTURE dff OF dff IS
11 BEGIN
12 b1: BLOCK (clk'EVENT AND clk='1')
13 BEGIN
14 q <= GUARDED '0' WHEN rst='1' ELSE d;
15 END BLOCK b1;
16 END dff;
17 -------------------------------
```

4.6 Programmation séquentiel

Le code VHDL est intrinsèquement concurrent. PROCESSES, FONCTIONS et PROCÉDURES sont les seules sections de code qui sont exécutées séquentiellement. Cependant, dans l'ensemble, n'importe lequel de ces blocs est toujours en même temps que toute autre instruction placée en dehors de celui-ci.

Un aspect important du code séquentiel est qu'il n'est pas limité à la logique séquentielle. En effet, avec elle, nous pouvons construire des circuits séquentiels ainsi que des circuits combinatoires. Le code séquentiel est également appelé code comportemental.

Les instructions décrites dans cette section sont toutes séquentielles, c'est-à-dire autorisées uniquement à l'intérieur des PROCESSES, FUNCTIONS ou PROCEDURES. Ce sont : IF, WAIT, CASE et LOOP.

Les VARIABLES sont également restreintes pour être utilisées dans le code séquentiel uniquement (c'est-à-dire à l'intérieur d'un PROCESS, FUNCTION ou PROCEDURE). Ainsi, contrairement à un SIGNAL, une VARIABLE ne peut jamais être globale, donc sa valeur ne peut pas être passée directement.

Nous allons nous concentrer sur les PROCESSUS ici. Les FONCTIONS et PROCÉDURES sont très similaires, mais sont destinées à la conception au niveau du système, étant donc vues dans la partie II de ce livre.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

4.6.1 PROCESS

Un PROCESS est une section séquentielle du code VHDL. Il est caractérisé par la présence de IF, WAIT, CASE ou LOOP, et par une liste de sensibilité (sauf lorsque WAIT est utilisé). Un PROCESS doit être installé dans le code principal et est exécuté chaque fois qu'un signal dans la liste de sensibilité change (ou que la condition liée à WAIT est remplie). Sa syntaxe est indiquée ci-dessous.

```
[label:] PROCESS (sensitivity list)
[VARIABLE name type [range] [:= initial_value;]]
BEGIN
(sequential code)
END PROCESS [label];
```

Les VARIABLES sont facultatives. S'ils sont utilisés, ils doivent être déclarés dans la partie déclarative du PROCESSUS (avant le mot BEGIN, comme indiqué dans la syntaxe ci-dessus). La valeur initiale n'est pas synthétisable, étant uniquement prise en compte dans les simulations. L'utilisation d'une étiquette est également facultative. Son objectif est d'améliorer la lisibilité du code. Pour construire un circuit synchrone, la surveillance d'un signal (horloge, par exemple) est nécessaire. Un moyen courant de détecter un changement de signal est au moyen de l'attribut EVENT . Par exemple, si clk est un signal à surveiller, alors clk'EVENT retourne TRUE quand un changement sur clk se produit (front montant ou descendant). Un exemple, illustrant l'utilisation de EVENT et PROCESS, est illustré ci-dessous.

Exemple :

Une bascule de type D (figure 77) est le bloc de construction le plus élémentaire des circuits logiques séquentiels. Dans celui-ci, la sortie doit copier l'entrée à la transition positive ou négative du signal d'horloge (front montant ou descendant). Dans le code présenté ci-dessous, nous utilisons l'instruction IF (abordée dans la section 6.3) pour concevoir un DFF avec réinitialisation asynchrone.

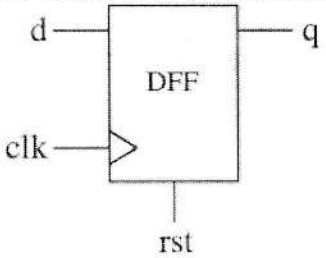


Figure 77. Bascule D

Si rst= «1», alors la sortie doit être q = '0' (lignes 14 à 15), quel que soit l'état de clk. Sinon, la sortie doit copier l'entrée (c'est-à-dire q = d) sur le front montant de clk (lignes 16 à 17). L'attribut EVENT est utilisé à la ligne 16 pour détecter une transition d'horloge. Le PROCESS (lignes 12–19) est exécuté à chaque fois que l'un des signaux apparaissant dans sa liste de sensibilité (clk et rst, ligne 12) change.


```
1 ------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------
5 ENTITY dff IS
6 PORT (d, clk, rst: IN STD_LOGIC;
7 q: OUT STD_LOGIC);
8 END dff;
9 ------------------------------------
10 ARCHITECTURE behavior OF dff IS
11 BEGIN
12 PROCESS (clk, rst)
13 BEGIN
14 IF (rst='1') THEN
15 q <= '0';
16 ELSIF (clk'EVENT AND clk='1') THEN
17 q <= d;
18 END IF;
19 END PROCESS;
20 END behavior;
21 ------------------------------------
```

4.6.2 Signaux et variables

Les signaux et les variables seront étudiés en détail dans la prochaine section. Cependant, il est impossible de discuter du code séquentiel sans connaître au moins ses caractéristiques les plus élémentaires.

VHDL à deux façons de passer des valeurs non statiques : au moyen d'un SIGNAL ou au moyen d'une VARIABLE. Un SIGNAL peut être déclaré dans un PACKAGE, ENTITY ou ARCHITECTURE (dans sa partie déclarative), tandis qu'une VARIABLE ne peut être déclarée qu'à l'intérieur d'un morceau de code séquentiel (dans un PROCESS, par exemple). Par conséquent, alors que la valeur de la première peut être globale, la seconde est toujours locale.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

La valeur d'une VARIABLE ne peut jamais être transmise directement du PROCESSUS; si nécessaire, il doit être affecté à un SIGNAL. En revanche, la mise à jour d'une VARIABLE est immédiate, c'est-à-dire que l'on peut compter rapidement sur sa nouvelle valeur dans la ligne de code suivante. Ce n'est pas le cas avec un SIGNAL (lorsqu'il est utilisé dans un PROCESSUS), car sa nouvelle valeur n'est généralement garantie d'être disponible qu'après la conclusion de l'exécution actuelle du PROCESSUS.

Enfin, rappelez-vous que l'opérateur d'affectation pour un SIGNAL est « <= » (ex.: Sig <= 5), alors que pour une VARIABLE, il est «: = » (ex.: Var: =5).

4.6.3 IF

Comme mentionné précédemment, IF, WAIT, CASE et LOOP sont les instructions destinées au code séquentiel. Par conséquent, ils ne peuvent être utilisés que dans un PROCESS, une FONCTION ou une PROCÉDURE.

La tendance naturelle est que les gens utilisent IF plus que tout autre énoncé.

Bien que cela puisse, en principe, avoir une conséquence négative (parce que l'instruction IF / ELSE pourrait déduire la construction d'un décodeur de priorité inutile), le synthétiseur optimisera la structure et évitera le matériel supplémentaire. La syntaxe de IF est indiqué ci-dessous.

```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

Exemple

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
```

4.6.4 WAIT

Le fonctionnement de WAIT est parfois similaire à celui de IF. Cependant, plus d'une forme de WAIT est disponible. De plus, contrairement à l'utilisation de IF, CASE ou LOOP, le PROCESS ne peut pas avoir de liste de sensibilité lorsque WAIT est utilisé. Sa syntaxe (il existe trois formes de WAIT) est indiquée ci-dessous.

```
WAIT UNTIL signal_condition;
```

```
WAIT ON signal1 [, signal2, ... ];
```

```
WAIT FOR time;
```

L'instruction WAIT UNTIL n'accepte qu'un seul signal, ce qui est plus approprié pour le code synchrone qu'asynchrone. Puisque le PROCESS n'a pas de liste de sensibilité dans ce cas, WAIT UNTIL doit être la première instruction du PROCESS. Le PROCESSUS sera exécuté chaque fois que la condition est remplie.

Exemple :

```
PROCESS -- no sensitivity list
BEGIN
WAIT UNTIL (clk'EVENT AND clk='1');
IF (rst='1') THEN
output <= "00000000";
ELSIF (clk'EVENT AND clk='1') THEN
output <= input;
END IF;
```

L'instruction WAIT UNTIL n'accepte qu'un seul signal, ce qui est plus approprié pour le code synchrone qu'asynchrone. Puisque le PROCESS n'a pas de liste de sensibilité dans ce cas, WAIT UNTIL doit être la première instruction du PROCESS. Le PROCESSUS sera exécuté chaque fois que la condition est remplie. ...

Exemple :

```
PROCESS
BEGIN
WAIT ON clk, rst;
IF (rst='1') THEN
output <= "00000000";
ELSIF (clk'EVENT AND clk='1') THEN
output <= input;
END IF;
END PROCESS;
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

4.6.5 CASE

CASE est une autre instruction destinée exclusivement au code séquentiel (avec IF, LOOP et WAIT). Sa syntaxe est indiquée ci-dessous.

```
CASE identifier IS
WHEN value => assignments;
WHEN value => assignments;
...
END CASE;
```

Exemple :

```
CASE control IS
WHEN "00" => x<=a; y<=b;
WHEN "01" => x<=b; y<=c;
WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

L'instruction CASE (séquentielle) est très similaire à WHEN (combinatoire). Ici aussi, toutes les permutations doivent être testées, le mot-clé OTHERS est donc souvent utile. Un autre mot clé important est NULL (l'équivalent de UNAFFECTED), qui doit être utilisé lorsqu'aucune action ne doit avoir lieu. Par exemple, WHEN OTHERS => NULL cependant, CASE autorise plusieurs affectations pour chaque condition de test (comme indiqué dans l'exemple ci-dessus), tandis que WHEN n'en autorise qu'une. Comme dans le cas de WHEN, ici aussi «WHEN value» peut prendre trois formes :

```
WHEN value                  -- single value
WHEN value1 to value2       -- range, for enumerated data types
                            -- only
WHEN value1 | value2 |.  -- value1 or value2 or ...
```

4.6.6 LOOP

Comme son nom l'indique, LOOP est utile lorsqu'un morceau de code doit être instancié plusieurs fois. Comme IF, WAIT et CASE, LOOP est destiné exclusivement au code séquentiel, il ne peut donc être utilisé qu'à l'intérieur d'un PROCESS, FUNCTION ou PROCEDURE.

Il existe plusieurs façons d'utiliser LOOP, comme indiqué dans les syntaxes ci-dessous :

- FOR / LOOP : La boucle est répétée un nombre fixe de fois.

```
[label:] FOR identifier IN range LOOP
(sequential statements)
END LOOP [label];
```

- WHILE / LOOP: La boucle est répétée jusqu'à ce qu'une condition ne soit plus remplie.

```
[label:] WHILE condition LOOP
(sequential statements)
END LOOP [label];
```

- EXIT : Utilisé pour terminer la boucle.

```
[label:] EXIT [label] [WHEN condition];
```

- NEXT : Utilisé pour sauter des étapes de boucle.

```
[label:] NEXT [loop_label] [WHEN condition];
```

4.6.7 CASE ou IF

Bien qu'en principe la présence de ELSE dans l'instruction IF / ELSE puisse déduire la mise en œuvre d'un décodeur de priorité (ce qui ne se produirait jamais avec CASE), cela ne se produira généralement pas. Par exemple, lorsque IF (une instruction séquentielle) est utilisé pour implémenter un circuit entièrement combinatoire, un multiplexeur peut être déduit à la place. Par conséquent, après optimisation, la tendance générale est qu'un circuit synthétisé à partir d'un code VHDL basé sur IF ne diffère pas de celui basé sur CASE.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

	WHEN	CASE
Type de déclaration	Concurrent	Séquentiel
Usage	Uniquement en dehors des PROCESSUS, FONCTIONS ou PROCÉDURES	Uniquement à l'intérieur des PROCESSUS, FONCTIONS ou PROCÉDURES
Toutes les permutations doivent être testées	Oui pour WITH / SELECT / WHEN	Oui
Max. # d'affectations par test	1	quelconque
Mot-clé sans action	UNAFFECTED	NULL

Table 6. Type de déclaration

Exemple :

```
---- With WHEN: ----------------
WITH sel SELECT
x <= a WHEN "000",
b WHEN "001",
c WHEN "010",
UNAFFECTED WHEN OTHERS;
---- With CASE: ----------------
CASE sel IS
WHEN "000" => x<=a;
WHEN "001" => x<=b;
WHEN "010" => x<=c;
WHEN OTHERS => NULL;
END CASE;
--------------------------------
```

4.6.8 CASE versus WHEN

CASE et WHEN sont très similaires. Cependant, alors que l'un est simultané (WHEN), l'autre est séquentiel (CASE). Leurs principales similitudes et différences sont résumées dans le tableau 6.1.

Exemple :

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

```
---- With IF: --------------
IF (sel="00") THEN x<=a;
ELSIF (sel="01") THEN x<=b;
ELSIF (sel="10") THEN x<=c;
ELSE x<=d;
---- With CASE: ------------
CASE sel IS
WHEN "00" => x<=a;
WHEN "01" => x<=b;
WHEN "10" => x<=c;
WHEN OTHERS => x<=d;
END CASE;
----------------------------
```

4.7 Signaux et Variables

VHDL fournit deux objets pour traiter les valeurs de données non statiques : SIGNAL et VARIABLE. Il fournit également des moyens pour établir des valeurs par défaut (statiques) : CONSTANT et GENERIC. Le dernier d'entre eux (l'attribut GENERIC) a déjà été vu dans la section précédente. SIGNAL, VARIABLE et CONSTANT seront étudiés ensemble dans ce chapitre.

CONSTANT et SIGNAL peuvent être globaux (c'est-à-dire vus par le code entier) et peuvent être utilisés dans l'un ou l'autre type de code, simultané ou séquentiel. Une VARIABLE, par contre, est locale, car elle ne peut être utilisée qu'à l'intérieur d'un morceau de code séquentiel (c'est-à-dire dans un PROCESS, FUNCTION ou PROCEDURE) et sa valeur ne peut jamais être transmise directement.

Comme on le verra, le choix entre un SIGNAL ou une VARIABLE n'est pas toujours facile, donc une section entière et plusieurs exemples seront consacrés à ce sujet.

De plus, une discussion sur le nombre de registres déduits par le compilateur, basée sur les assignations SIGNAL et VARIABLE, sera également présentée.

4.7.1 CONSTANT

CONSTANT sert à établir les valeurs par défaut. Sa syntaxe est indiquée ci-dessous.

```
CONSTANT name : type := value;
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد الرياضيات

Exemple :

```
CONSTANT set_bit : BIT := '1';
CONSTANT datamemory : memory := (('0','0','0','0'),
                                 ('0','0','0','1'),
                                 ('0','0','1','1'));
```

Un CONSTANT peut être déclaré dans un PACKAGE, ENTITY ou ARCHITECTURE. Lorsqu'il est déclaré dans un package, il est vraiment global, car le package peut être utilisé par plusieurs entités. Lorsqu'il est déclaré dans une entité (après PORT), il est global à toutes les architectures qui suivent cette entité. Enfin, lorsqu'il est déclaré dans une architecture (dans sa partie déclarative), il est global uniquement au code de cette architecture. Les endroits les plus courants pour trouver une déclaration CONSTANTE sont dans une ARCHITECTURE ou dans un PACKAGE.

4.7.2 SIGNAL

SIGNAL sert à transmettre des valeurs dans et hors du circuit, ainsi qu'entre ses unités internes. En d'autres termes, un signal représente des interconnexions de circuits (fils). Par exemple, tous les PORTS d'une ENTITY sont des signaux par défaut. Sa syntaxe est la suivante :

```
SIGNAL name : type [range] [:= initial_value];
```

Exemple :

```
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

La déclaration d'un SIGNAL peut être faite aux mêmes endroits que la déclaration d'un CONSTANT (décrit ci-dessus).

Un aspect très important d'un SIGNAL, lorsqu'il est utilisé à l'intérieur d'une section de code séquentiel (PROCESS, par exemple), est que sa mise à jour n'est pas immédiate. En

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

d'autres termes, il ne faut pas s'attendre à ce que sa nouvelle valeur soit prête avant la conclusion du PROCESSUS, FONCTION ou PROCÉDURE correspondant.
Rappelons que l'opérateur d'affectation pour un SIGNAL est " <= " (Ex .: count <= 35;). De plus, la valeur initiale de la syntaxe ci-dessus n'est pas synthétisable, étant uniquement prise en compte dans les simulations.
Un autre aspect qui pourrait affecter le résultat est lorsque plusieurs affectations sont faites au même SIGNAL. Le compilateur peut se plaindre et quitter la synthèse, ou peut déduire le mauvais circuit (en ne considérant que la dernière affectation, par exemple).
Par conséquent, l'établissement des valeurs initiales, comme à la ligne 15 de l'exemple ci-dessous, doit être effectué avec une VARIABLE.

4.7.3 VARIABLE

Contrairement à CONSTANT et SIGNAL, une VARIABLE ne représente que des informations locales. Il ne peut être utilisé qu'à l'intérieur d'un PROCESS, FUNCTION ou PROCEDURE (c'est-à-dire en code séquentiel) et sa valeur ne peut pas être transmise directement. D'autre part, sa mise à jour est immédiate, de sorte que la nouvelle valeur peut être rapidement utilisée dans la ligne de code suivante.
Pour déclarer une VARIABLE, la syntaxe suivante doit être utilisée :

```
VARIABLE name : type [range] [:= init_value];
```

Exemple :

```
VARIABLE control: BIT := '0';
VARIABLE count: INTEGER RANGE 0 TO 100;
VARIABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) := "10001000";
```

Puisqu'une VARIABLE ne peut être utilisée qu'en code séquentiel, sa déclaration ne peut être faite que dans la partie déclarative d'un PROCESS, FUNCTION ou PROCEDURE.
Rappelez-vous que l'opérateur d'affectation pour une VARIABLE est " : = " (Ex : count : = 35;). De plus, comme dans le cas d'un SIGNAL, la valeur initiale dans la syntaxe ci-dessus n'est pas synthétisable, n'étant prise en compte que dans les simulations.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد الحقوق

4.7.4 SIGNAL ou VARIABLE

Comme déjà mentionné, choisir entre un SIGNAL ou une VARIABLE n'est pas toujours simple. Leurs principales différences sont résumées dans le tableau 6.

	SIGNAL	VARIABLE
Affectation	<=	:=
Utilité	Représente les interconnexions de circuits (fils)	Représente des informations locales
Portée	Peut-être global (vu par le code entier)	Local (visible uniquement à l'intérieur du PROCESSUS, FONCTION, ou PROCÉDURE)
Comportement	La mise à jour n'est pas immédiate en séquence code (nouvelle valeur généralement disponible uniquement à l'issue du PROCESSUS, FONCTION ou PROCÉDURE)	Mis à jour immédiatement (la nouvelle valeur peut être utilisé dans la prochaine ligne de code)
Usage	Dans un PACKAGE, ENTITY ou ARCHITECTURE. Dans une ENTITY, tous Les PORTS sont des SIGNAUX par défaut	Uniquement en code séquentiel, c'est-à-dire dans un PROCESSUS, FONCTION ou PROCÉDURE

Table 7. Signal et variable

Nous voulons souligner à nouveau qu'une affectation à une VARIABLE est immédiate, mais ce n'est pas le cas avec un SIGNAL. En général, la nouvelle valeur d'un SIGNAL ne sera disponible qu'à la fin de l'exécution en cours du PROCESSUS correspondant. Bien que ce ne soit pas toujours le cas, il est prudent de le considérer comme tel.

L'exemple présenté ci-dessous illustreront davantage ceci et d'autres différences entre SIGNAUX et VARIABLES.

Exemple :

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Dans cet exemple, nous implémenterons le même multiplexeur de l'exemple 4.2 (répété sur la figure 75). C'est, en effet, un exemple classique concernant le choix d'un SIGNAL par rapport à une VARIABLE.

```
1 -- Solution 1: using a SIGNAL (not ok) --
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY mux IS
6 PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7 y: OUT STD_LOGIC);
8 END mux;
9 ----------------------------------------
10 ARCHITECTURE not_ok OF mux IS
11 SIGNAL sel : INTEGER RANGE 0 TO 3;
12 BEGIN
13 PROCESS (a, b, c, d, s0, s1)
14 BEGIN
15 sel <= 0;
16 IF (s0='1') THEN sel <= sel + 1;
17 END IF;
18 IF (s1='1') THEN sel <= sel + 2;
19 END IF;
20 CASE sel IS
21 WHEN 0 => y<=a;
22 WHEN 1 => y<=b;
23 WHEN 2 => y<=c;
24 WHEN 3 => y<=d;
25 END CASE;
26 END PROCESS;
27 END not_ok;
28 ----------------------------------------
1 -- Solution 2: using a VARIABLE (ok) ----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY mux IS
6 PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7 y: OUT STD_LOGIC);
8 END mux;
9 ----------------------------------------
10 ARCHITECTURE ok OF mux IS
11 BEGIN
12 PROCESS (a, b, c, d, s0, s1)
13 VARIABLE sel : INTEGER RANGE 0 TO 3;
14 BEGIN
15 sel := 0;
16 IF (s0='1') THEN sel := sel + 1;
17 END IF;
18 IF (s1='1') THEN sel := sel + 2;
19 END IF;
20 CASE sel IS
21 WHEN 0 => y<=a;
22 WHEN 1 => y<=b;
23 WHEN 2 => y<=c;
24 WHEN 3 => y<=d;
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Chapitre 5. Applications : Implémentation de quelques circuits logiques dans les circuits FPGA

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

5.1 Exemple d'implantation

Dans Ce dernier chapitre deux simples exemples de concept de projet basé sur une implémentation de FPGA. Nous abordons comment implanter n'importe quel circuit (combinatoire ou séquentiel) dans un FPGA en utilisant le logiciel Xilinx, nous traitons la porte OU et le demi-additionneur.

5.1.1 La porte « OU »

Pour créer une porte OU nous allons suivre les étapes suivantes :

1. Après l'exécution du logiciel Xilinx
2. Créer un nouveau projet : **File + New project**

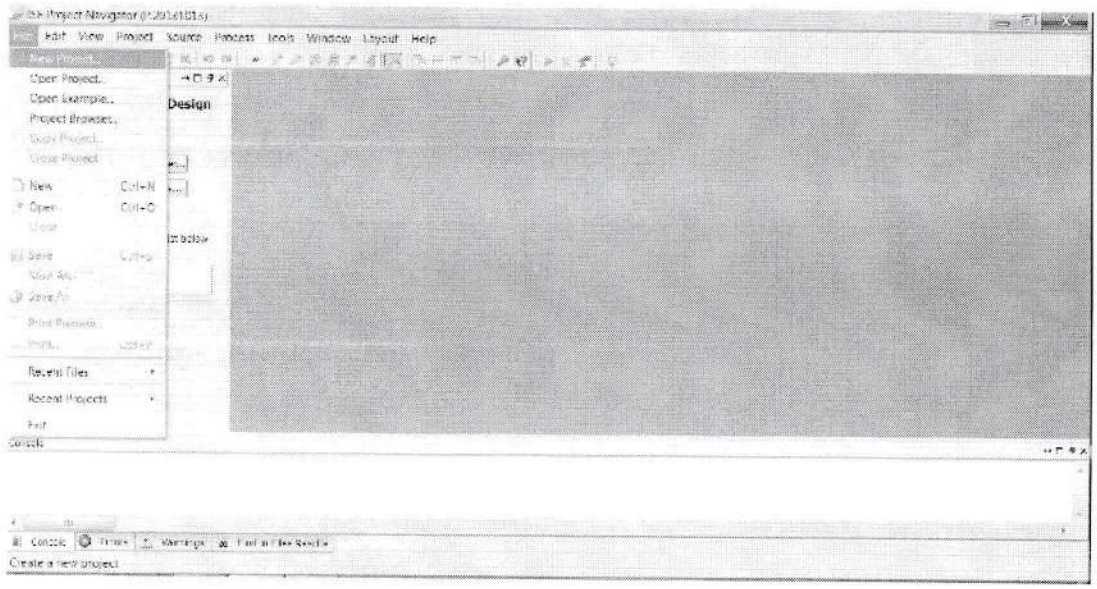


Figure 78. Création d'un nouveau projet

3. On introduit le nom et l'emplacement du projet comme indique la figure 79.

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

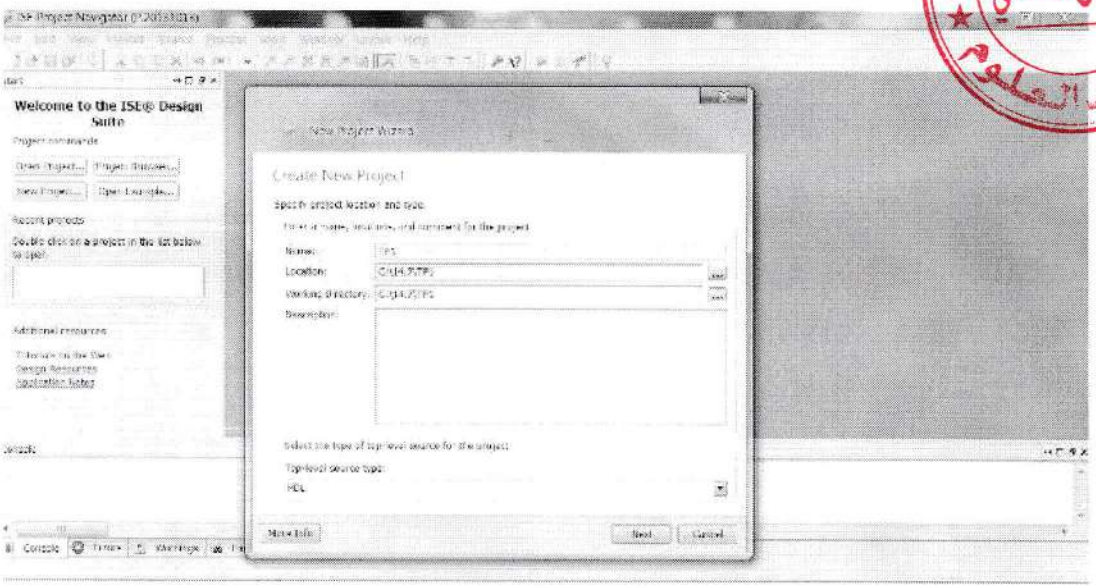


Figure 79. Fenêtre pour introduire le nom et l'emplacement du nouveau projet

4. On cliquant sur **Next** une deuxième fenêtre apparaitre pour introduire d'autres paramètres pour le projet.

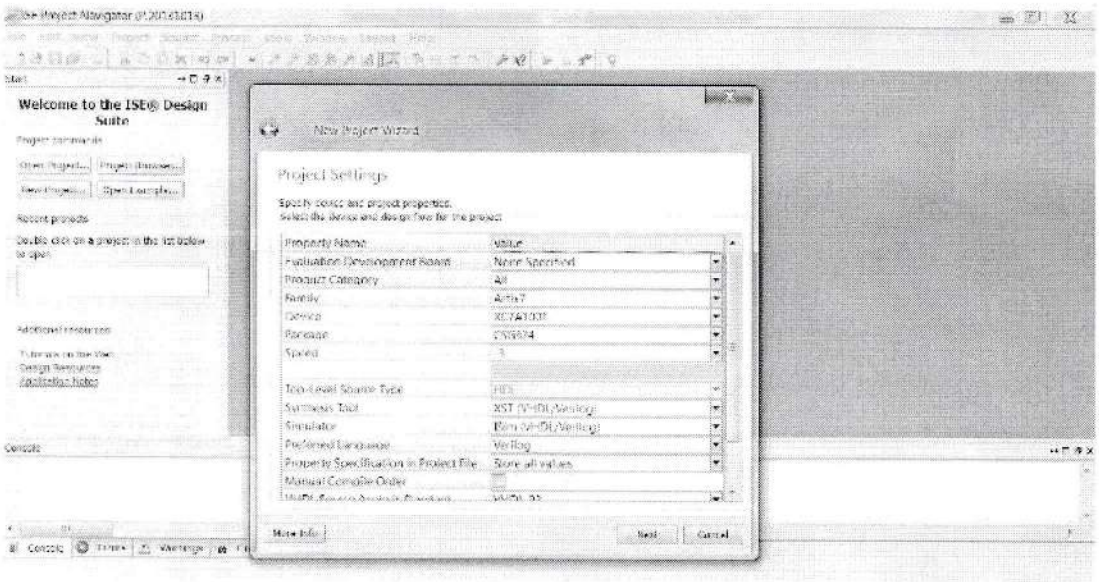


Figure 80. Fenêtre pour les paramètres du projet

5. On cliquant sur **Next** une troisième fenêtre apparaitre qui résume les différentes options du projet (figure 81).

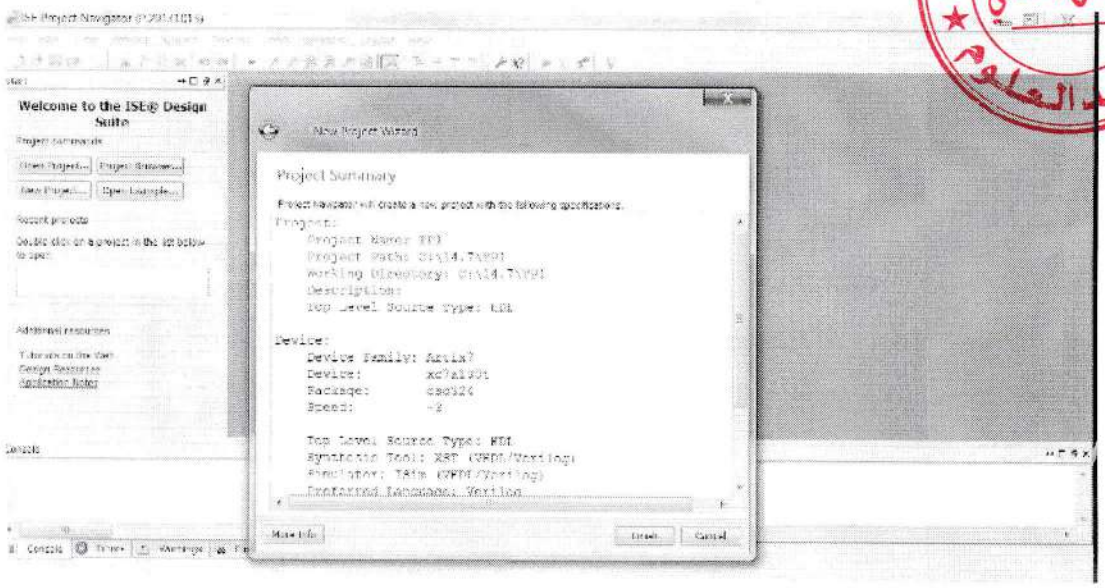


Figure 81. Fenêtre résumant les différentes options du projet

6. Cliquez sur **Project + New source** apparaitre une fenêtre pour choisir le type de programmation

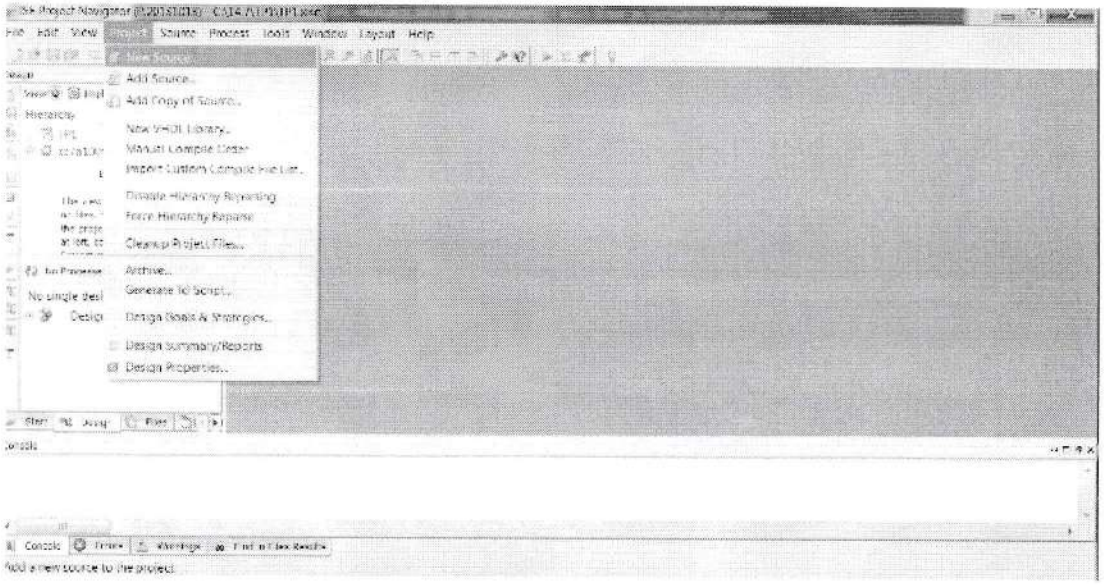


Figure 82. La création de type de programmation

7. La figure 83 montre les différents types de programmation (source), on choisit **VHDL Module** et on introduit le nom de fichier source (portand) et on clique sur **Next**.

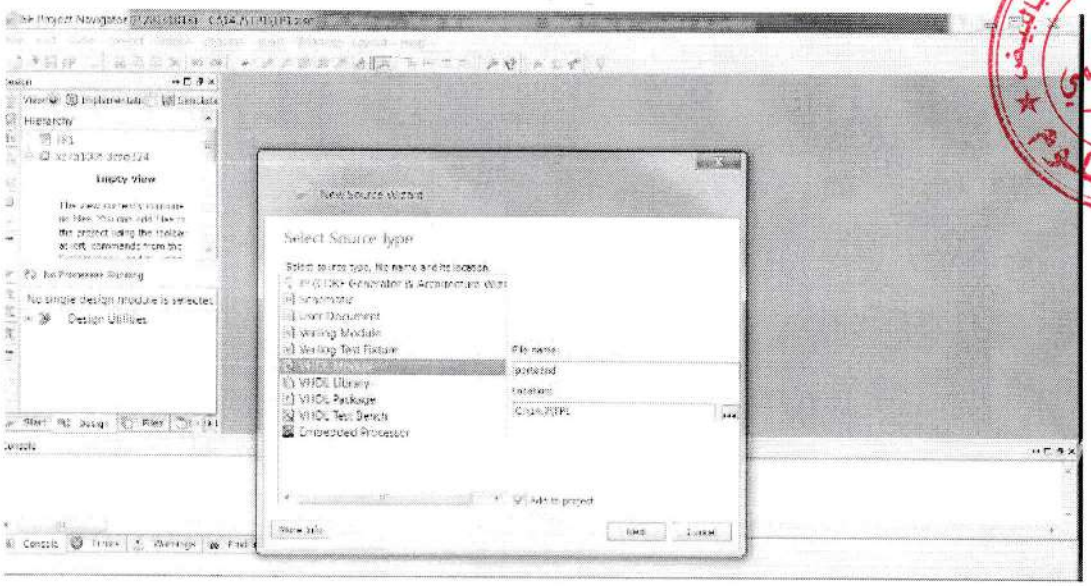


Figure 83. Fenêtre montre les différents types de source (programmation)

8. La figure 84 montre l'étape suivante, (**Define Module**) qui permet de nommé les entrées et les sorties

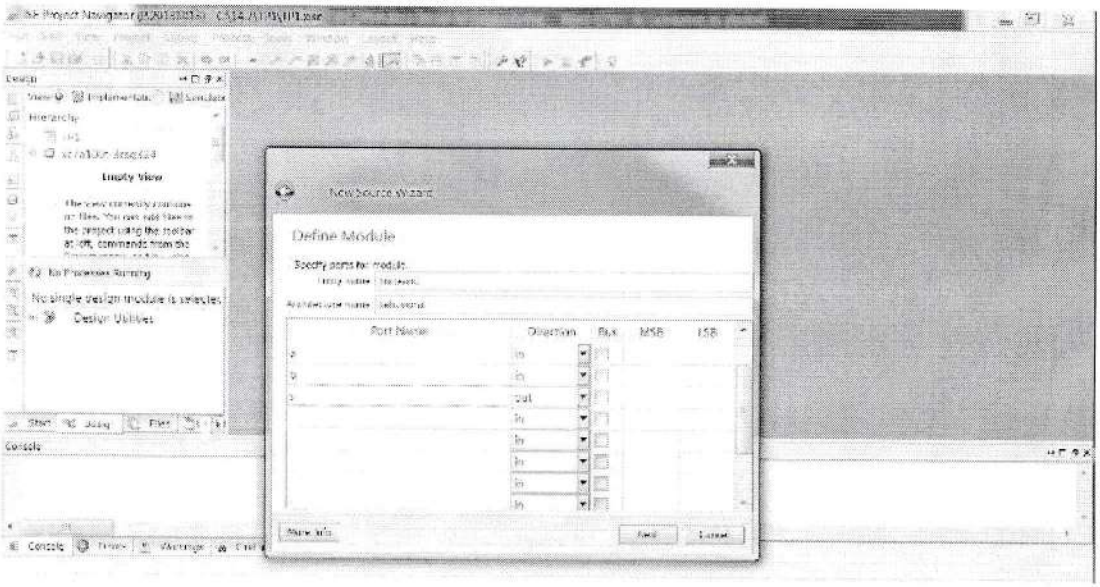


*Figure 84. Introduction des entrées et sorties dans la fenêtre (**Define Module**)*

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

9. Cliquez sur **Next**
10. Récapitulation des entrées et sorties est illustré dans la figure 85

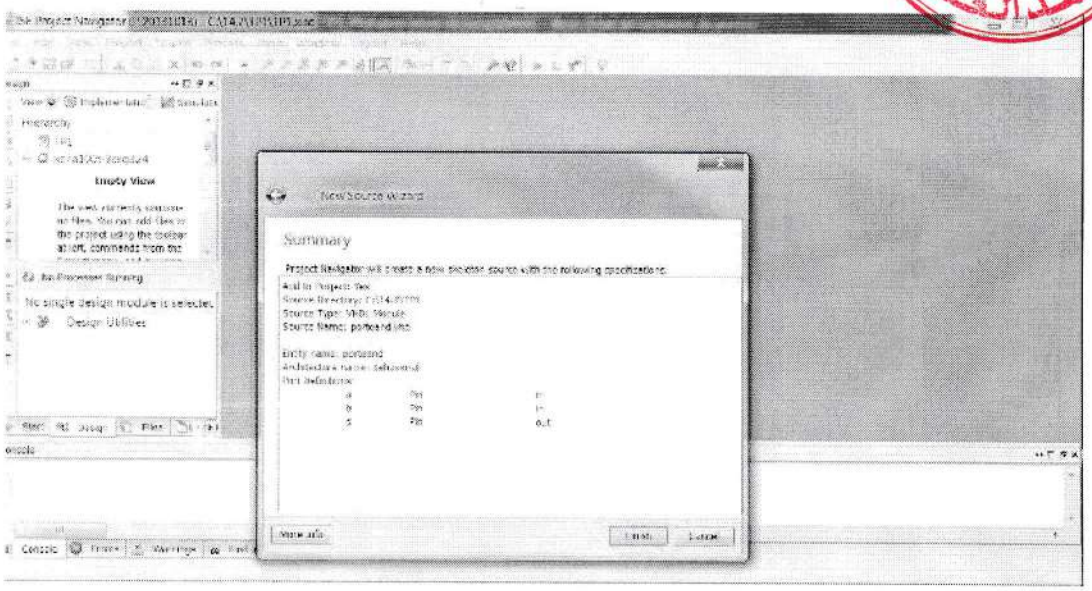


Figure 85. Récapitulation des entrées et sorties

11. Le fichier **porteand.vhd** est créé après de cliquer sur **Finish**

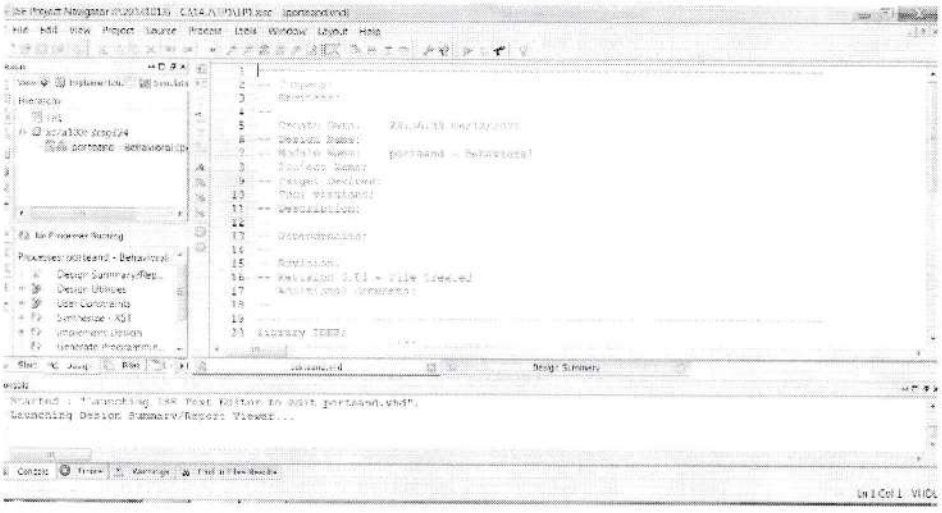


*Figure 86. La création du fichier **porteand.vhd***


```
----------------------
-- Company:
-- Engineer:
--
-- Create Date:    23:56:33 06/12/2021
-- Design Name:
-- Module Name:    porteand - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if
-- instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity porteand is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           s : out  STD_LOGIC);
end porteand;

architecture Behavioral of porteand is

begin

end Behavioral;
```

*Table 8. Le fichier **porteand.vhd***

12. Le fichier **porteand.vhd** est incomplet, il faut le compléter dans la partie architecture (Table 5).
13. Table 6 montre la partie architecture compléter par l'instruction **s <= a and b ;** qui est l'équation d'une porte AND

```
architecture Behavioral of porteand is
begin
s <= a and b;
end Behavioral;
```

Table 9. Partie architecture

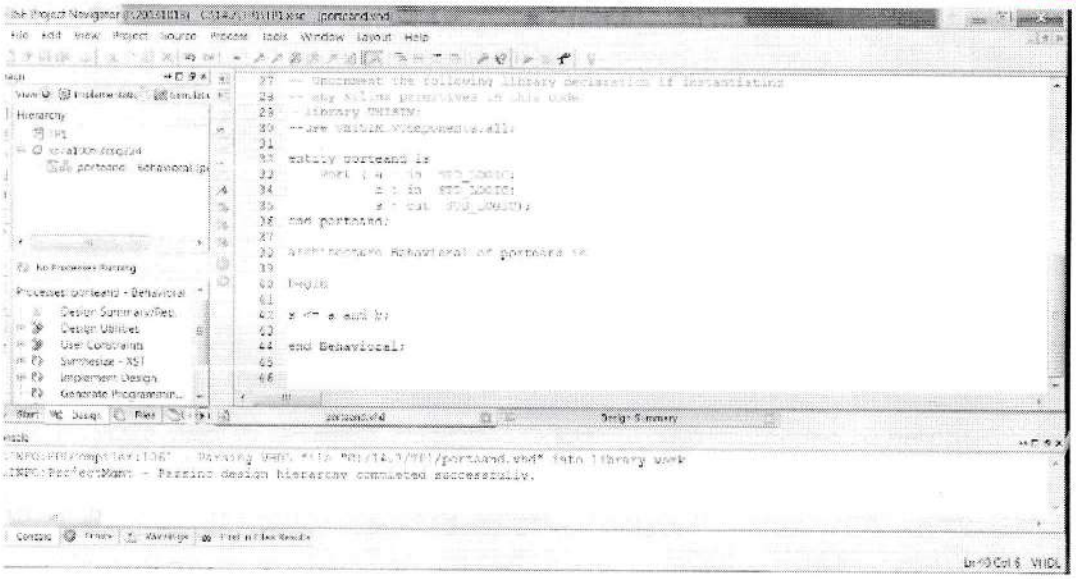


*Figure 87. Le fichier **porteand.vhd** compléter*

14. La figure 87 montre le fichier **porteand.vhd** compléter
15. Double clic sur **Check Syntax** on vérifie s'il y a d'erreur

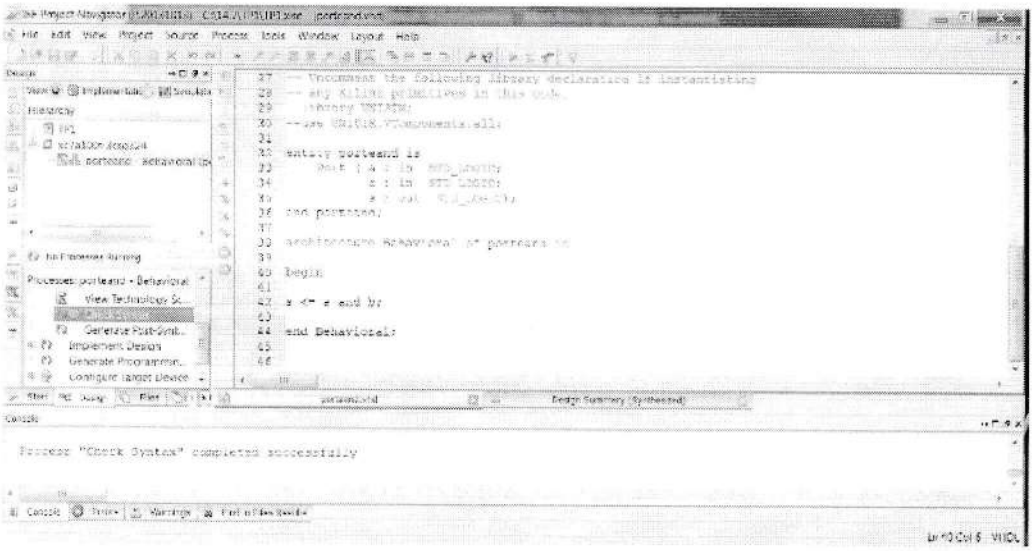


*Figure 88. Le fichier **porteand.vhd** après **Check Sytax***

14. Double clic sur **Synthesize- XST**, on vérifie toujours s'il y a d'erreurs

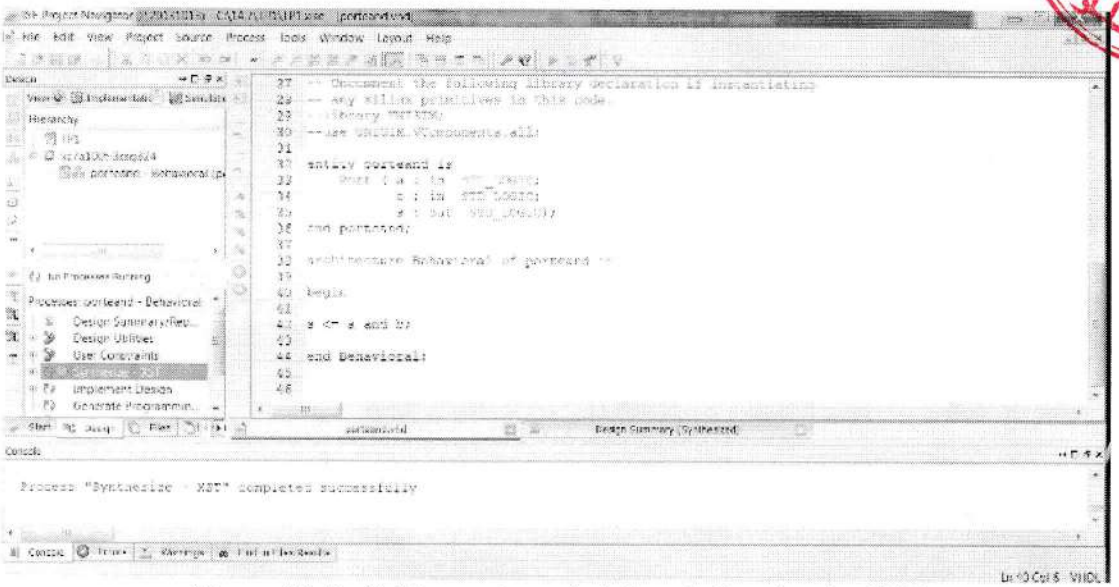


*Figure 89. Le fichier **porteand.vhd** après **Synthesize-XST***

15. L'étape suivante on va créer un deuxième fichier de type **VHDL Test Bench,** on cliquant sur **Project + New souce,** on nomme le fichier **porte_and** et on cliquant sur **Next**

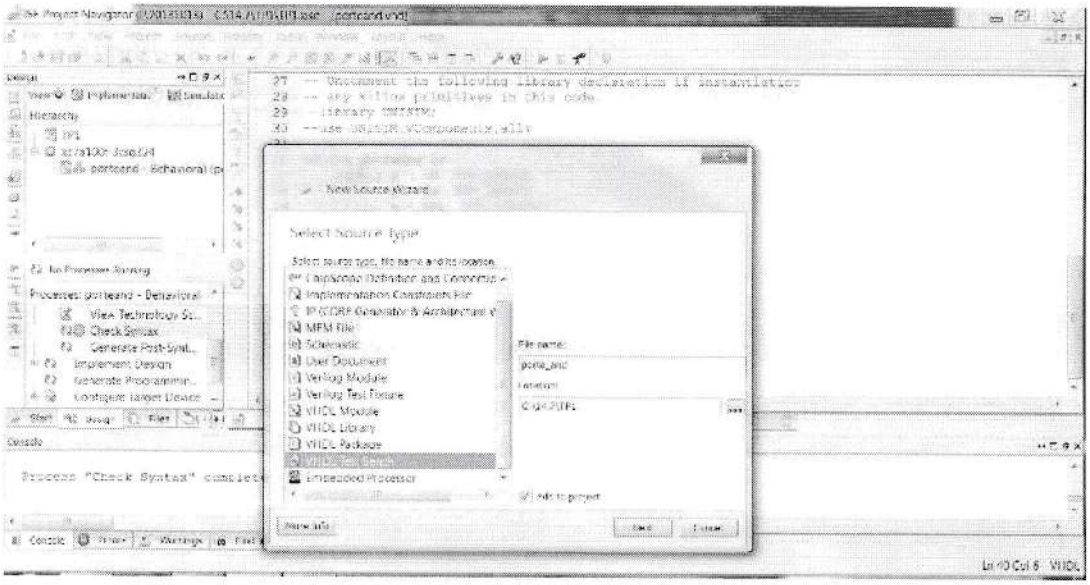


*Figure 90. Création de fichier de type **VHDL Test Bench***

16. Une fenêtre apparaitre confirme association du nouveau fichier **porte_and.vhd** avec l'ancien fichier **porteand.vhd** (figure 91)

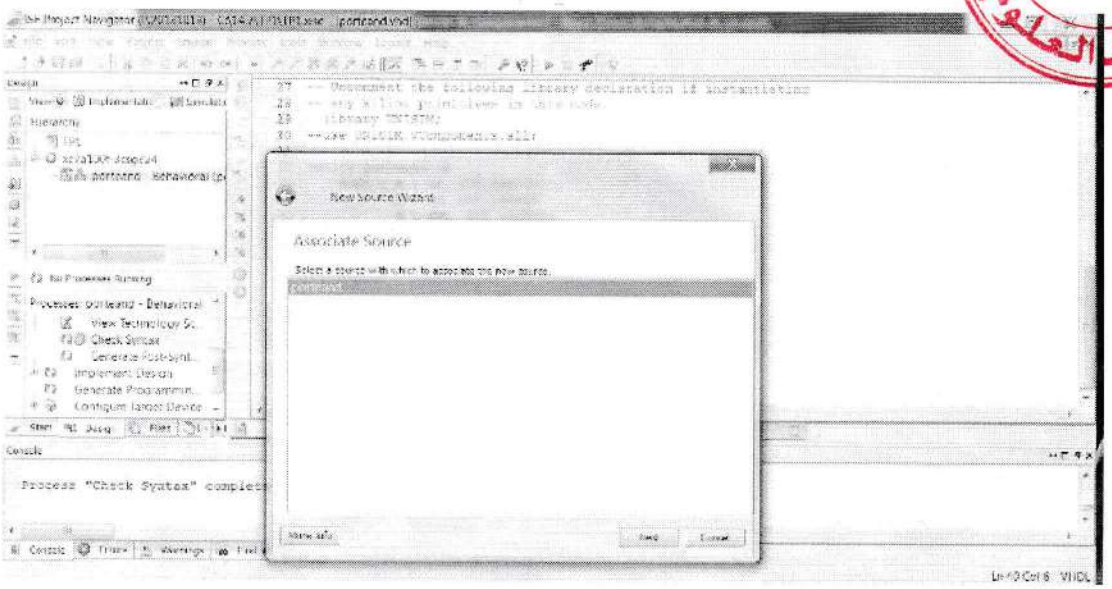


Figure 91. Fenêtre confirmant association des deux fichiers

17. La figure 92 résume les associations des deux fichiers VHDL

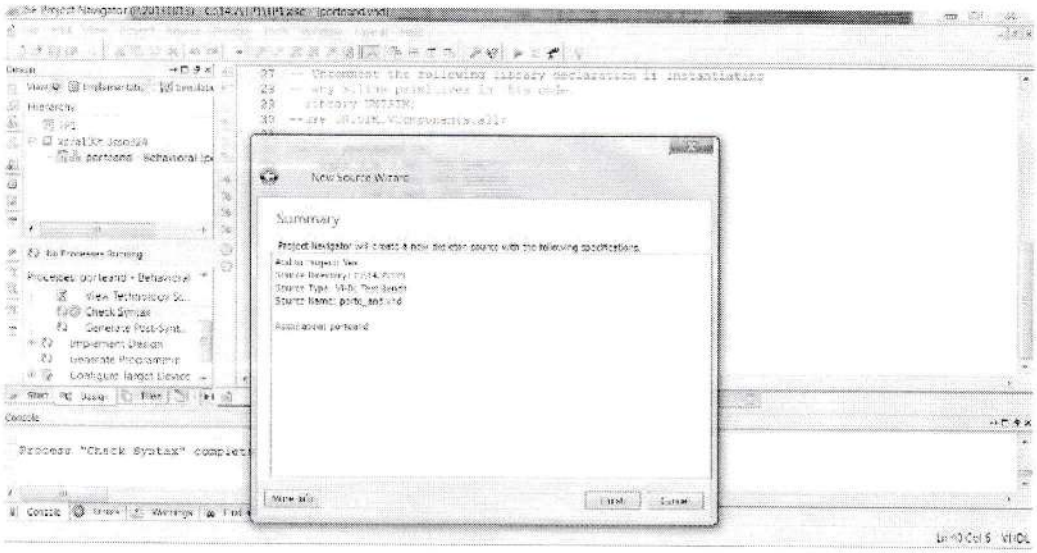


Figure 92. Récapitulation de l'association des deux fichiers

18. La fenêtre 76 confirme la création du fichier **VHDL Test Bench** avec succès.

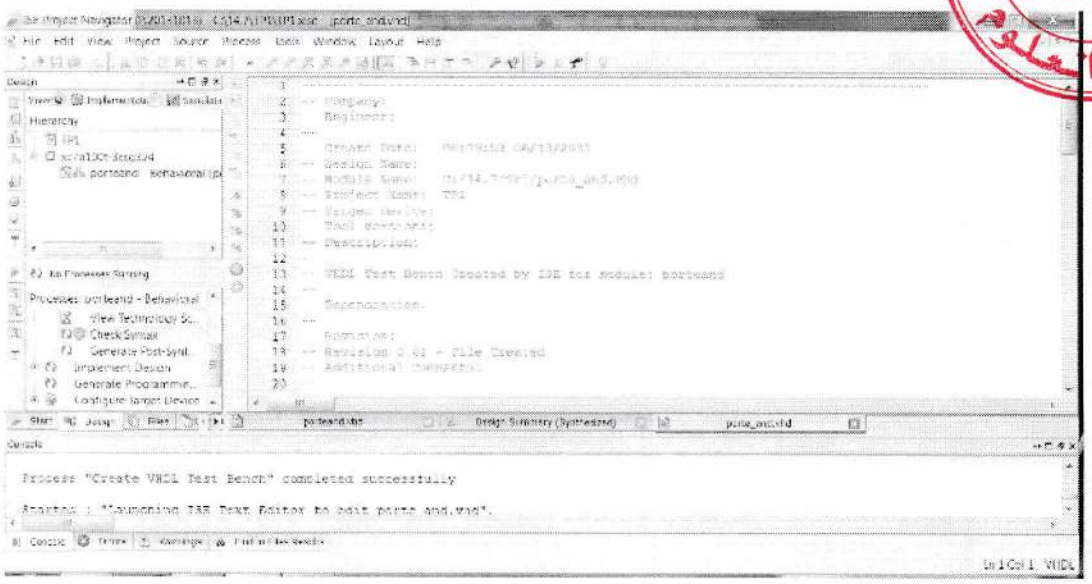


*Figure 93. Le fichier **VHDL Test Bench** est créé avec succès*

19. On modifie et compléter le fichier **porte_and.vdh** comme suit :

```
--------------------------------------------------------------------
-------------------
-- Company:
-- Engineer:
--
-- Create Date:    00:09:53 06/13/2021
-- Design Name:
-- Module Name:    C:/14.7/TP1/porte_and.vhd
-- Project Name:   TP1
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: porteand
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using
types std_logic and
-- std_logic_vector for the ports of the unit under test.
Xilinx recommends
```

```
-- that these types always be used for the top-level I/O of a
design in order
-- to guarantee that the testbench will bind correctly to the
post-implementation
-- simulation model.
--------------------------------------------------------------
------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY porte_and IS
END porte_and;

ARCHITECTURE behavior OF porte_and IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT porteand
    PORT(
         a : IN  std_logic;
         b : IN  std_logic;
         s : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal a : std_logic := '0';
   signal b : std_logic := '0';

                                                --Outputs
   signal s : std_logic;
   -- No clocks detected in port list. Replace <clock> below
with
   -- appropriate port name



BEGIN

                                                -- Instantiate the
Unit Under Test (UUT)
   uut: porteand PORT MAP (
          a => a,
          b => b,
          s => s
        );
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

```
   -- Clock process definitions


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
        wait for 100 ns; a<= '0'; b<= '0';
        wait for 100 ns; a<= '0'; b<= '1';
        wait for 100 ns; a<= '1'; b<= '0';
        wait for 100 ns; a<= '1'; b<= '1';
   wait for 100 ns;
   end process;



      -- insert stimulus here

END;
```

*Table 10. Fichier **porte_and.vhd***

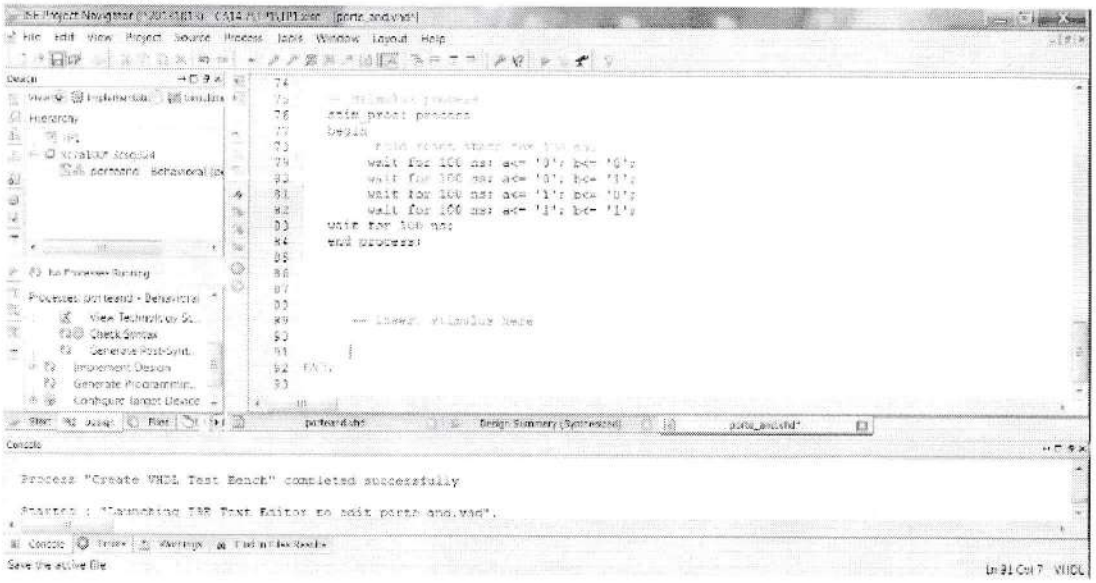


*Figure 94. Le fichier **porte_and.vhd** après modification*

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

20. Après cette étape on change le mode en Simulation

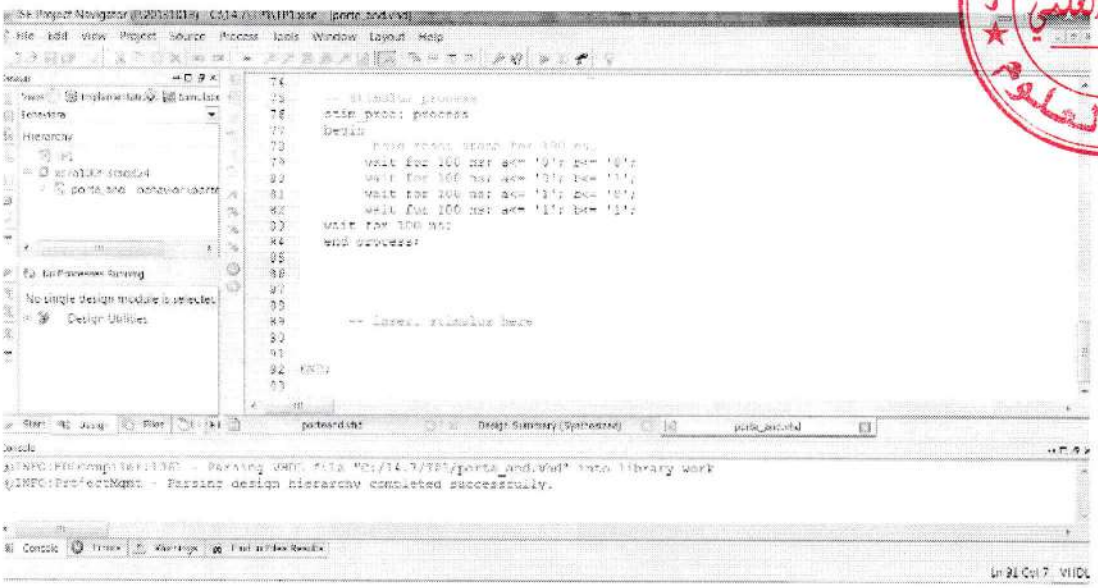


Figure 95. Changement en mode Simulation (la case Simulation coché)

21. Dans la fenêtre **Hierarchy** on clique sur **porte_and behavior** suivant la figure 96

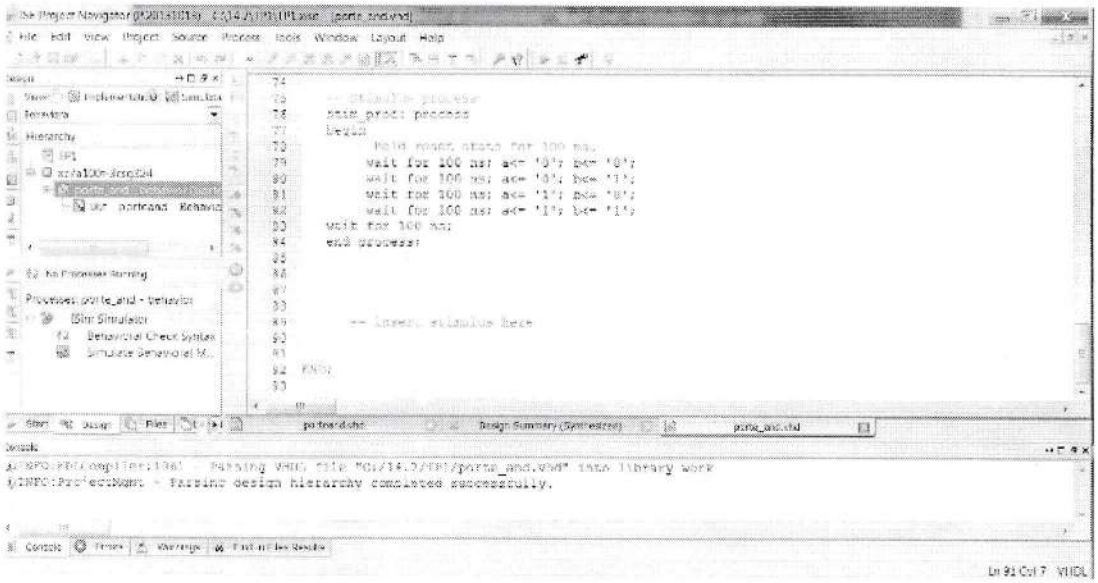


*Figure 96. Le fichier **porte_and.vhd** compléter*

22. On clique sur **ISim Simulator**

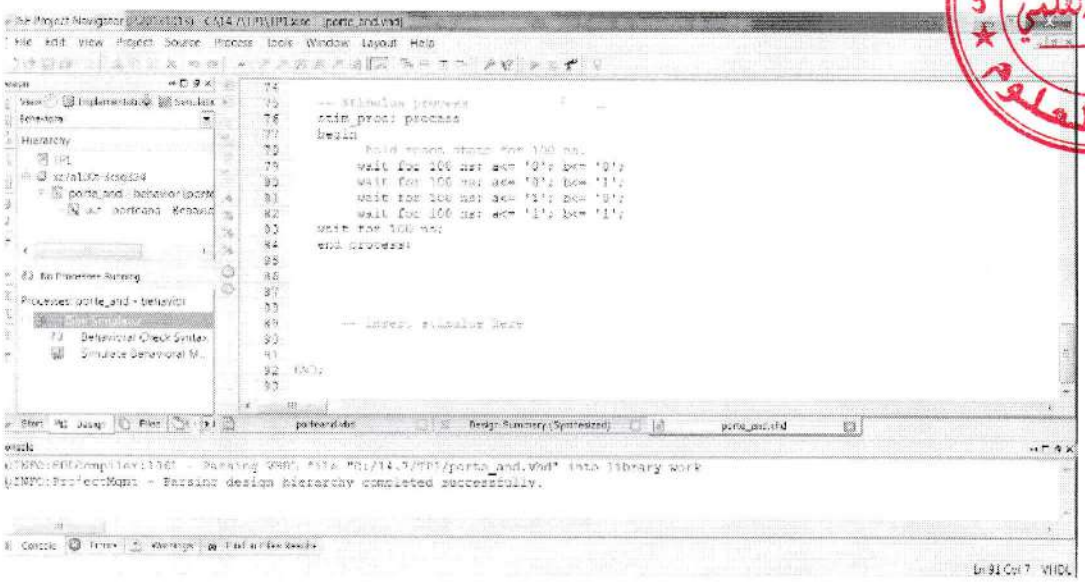


*Figure 97. **ISim Simulator** sélectionné*

23. Double clic sur **Behavior Check Syntax** pour trouver les erreurs possible

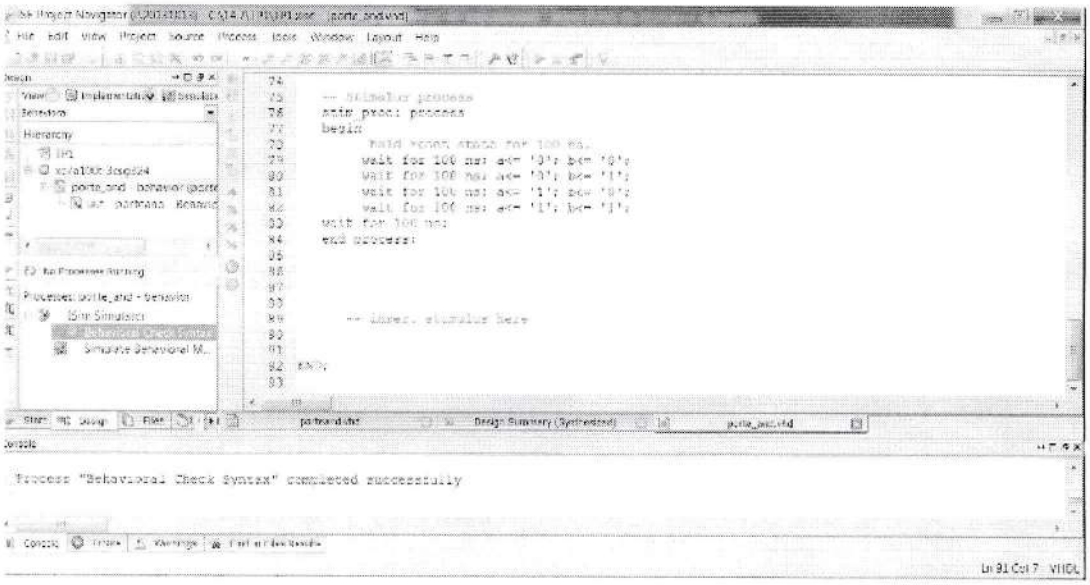


Figure 98. Fenêtre montrant qu'il n'y a pas erreurs (verte)

24. L'étape suivante, on clique sur **Simulate Behavioral Model** comme la montre la figure 99, on verifie toujours dans la fentre console que : **process « Simulate Behavioral Model » completed successfully**

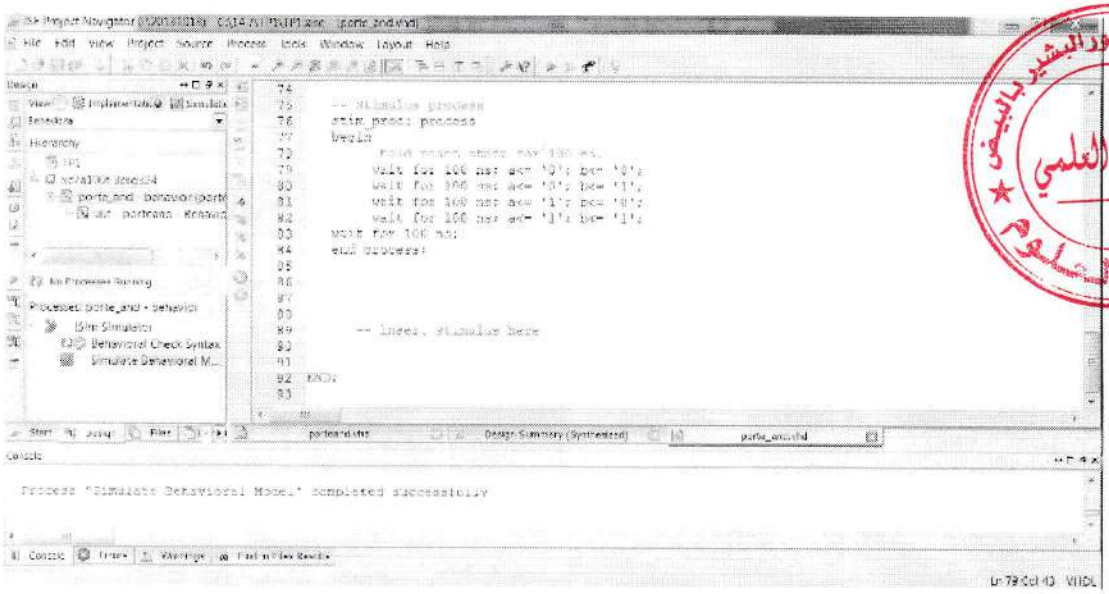


Figure 99. Le model de simulation est compléter

25. Finalement le chronogramme est affiché dans la figure 100

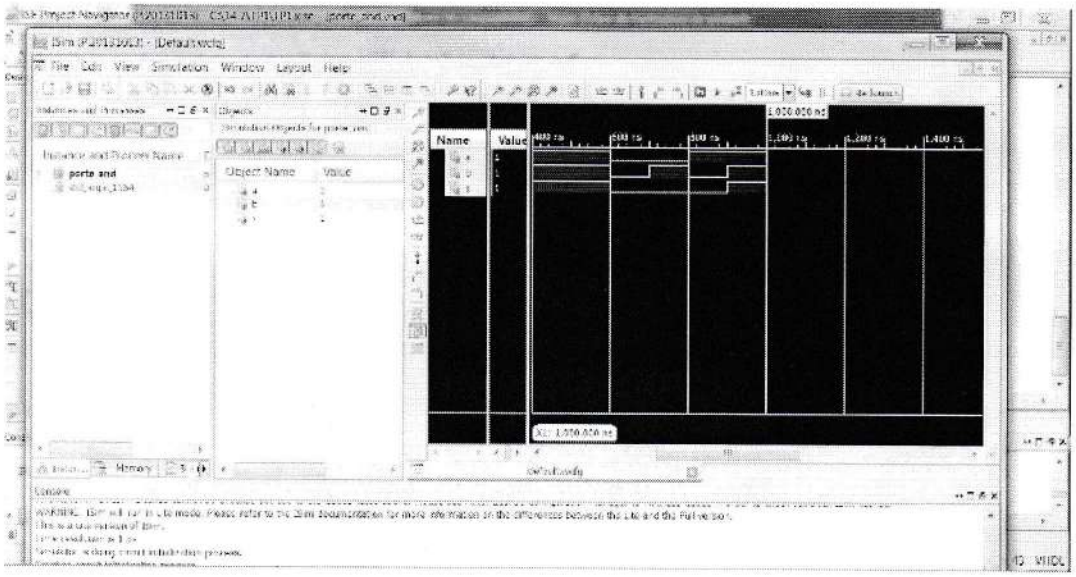


Figure 100. Chronogramme des entrées et sorties

5.1.2 Demi additionneur

Le deuxième exemple est de réaliser un demi-additionneur en suivant les étapes définit précédemment :

1. La première étape est de créer un nouveau projet on le nomme TP2, on cliquant sur **File+New Project**

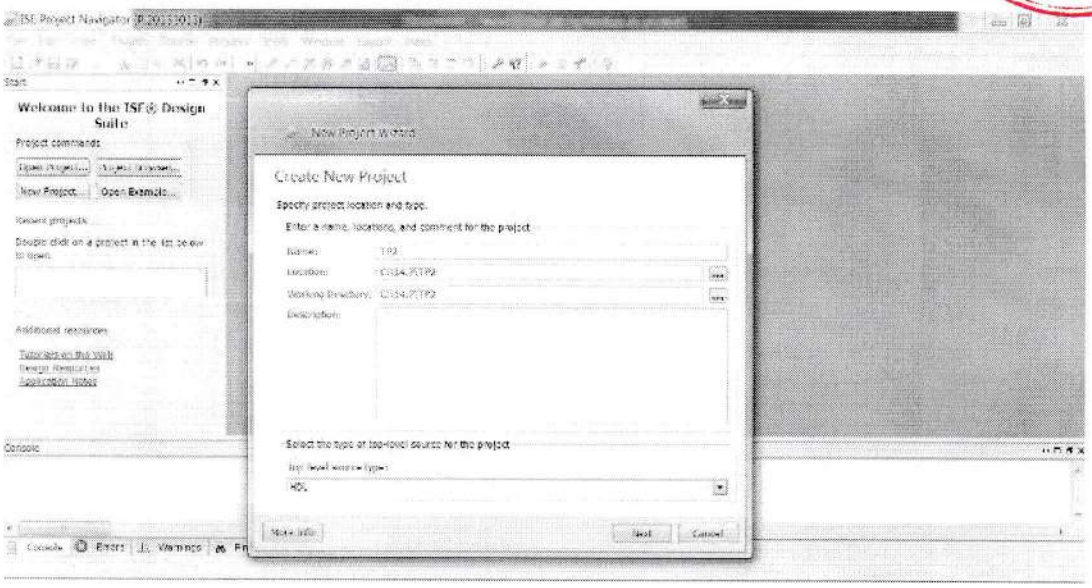


Figure 101. Création d'un nouveau projet TP2

2. La figure 102 indique la configuration du nouveau projet

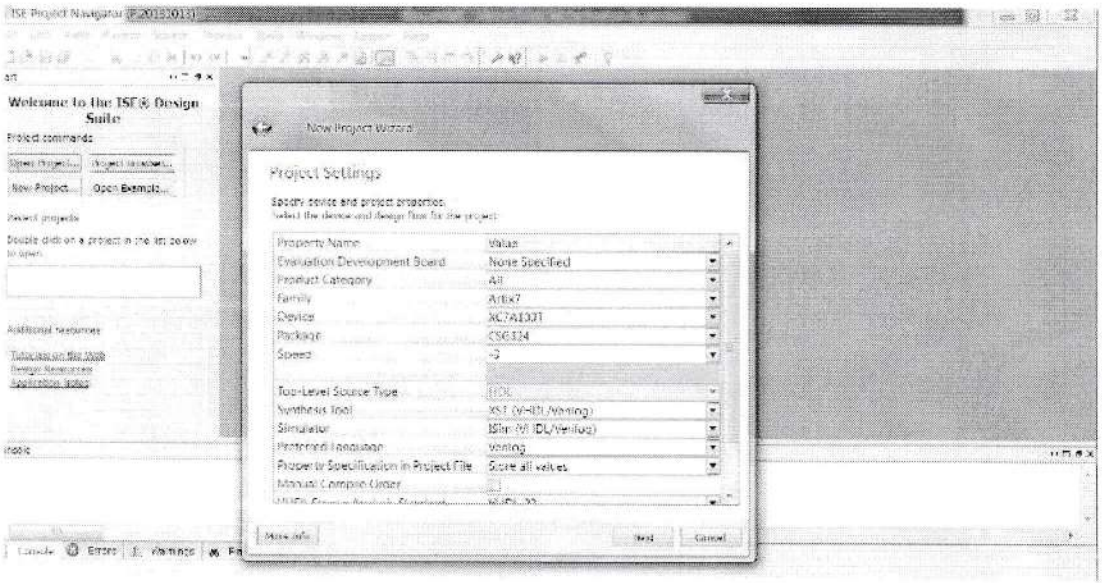


Figure 102. Les différentes configurations du nouveau projet

3. Après d'avoir choisi les différentes configurations on clique sur **Next**
4. La figure 103 montre les différentes configurations du nouveau projet et cliqué sur **Finish**

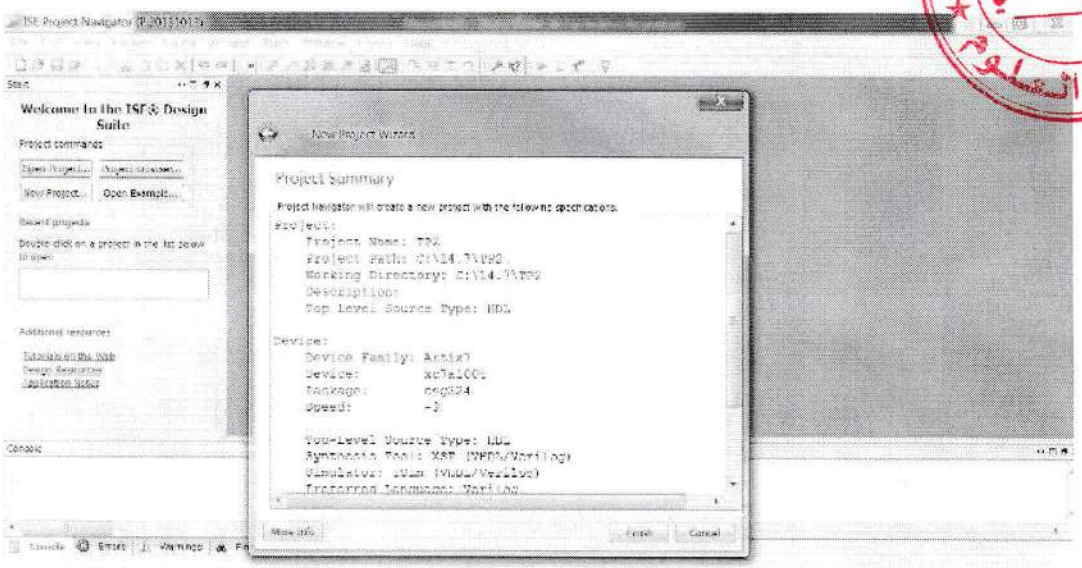


Figure 103. Résumé pour les différentes configurations

5. Après le clic sur Finish on trouve la fenêtre illustré par la figure 104

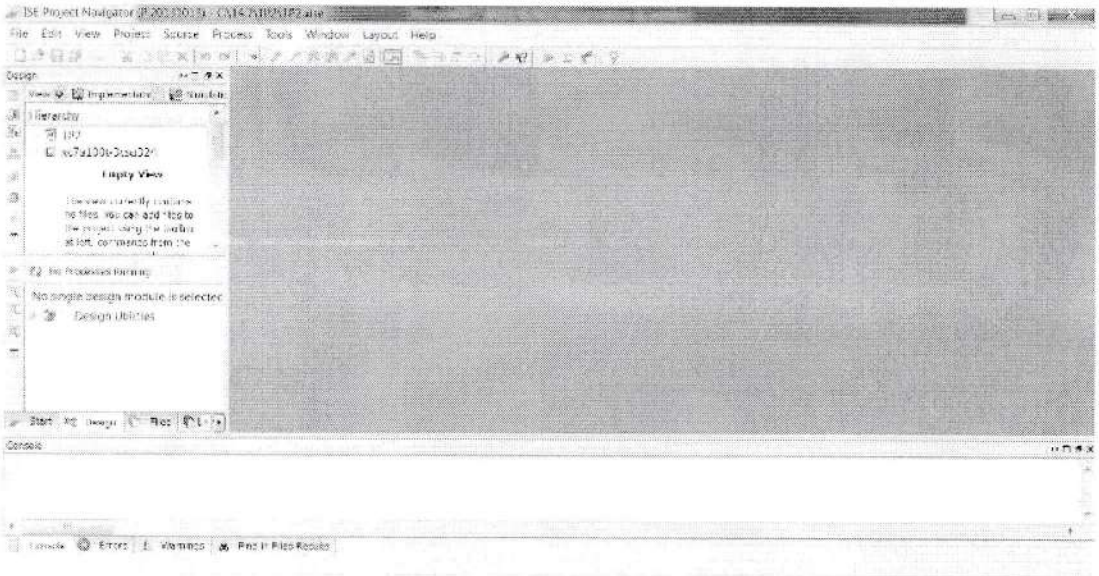


Figure 104. Création du nouveau projet

6. Après cette étape on va créer une nouvelle source de programmation, on cliquant sur **Project+New Source**

المركز الجامعي نور البشير بالبيض
المجلس العلمي

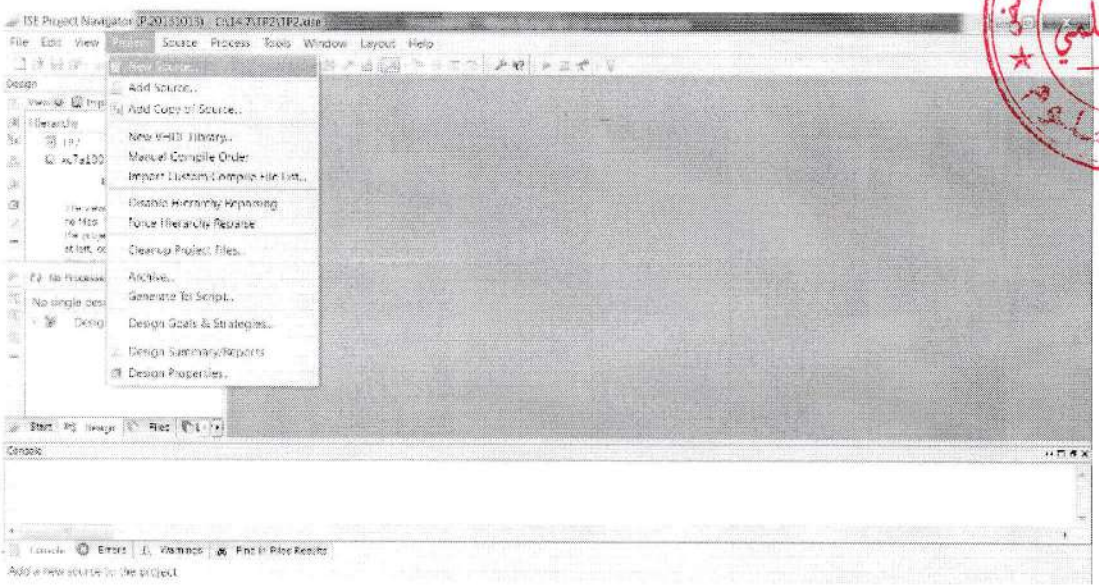


Figure 105. Création d'une nouvelle source de programmation

7. Une fenêtre apparaitre pour introduire le nom le type de source, on choisit **VHDL Module** comme type source et **demiadd.vhd** comme nom.

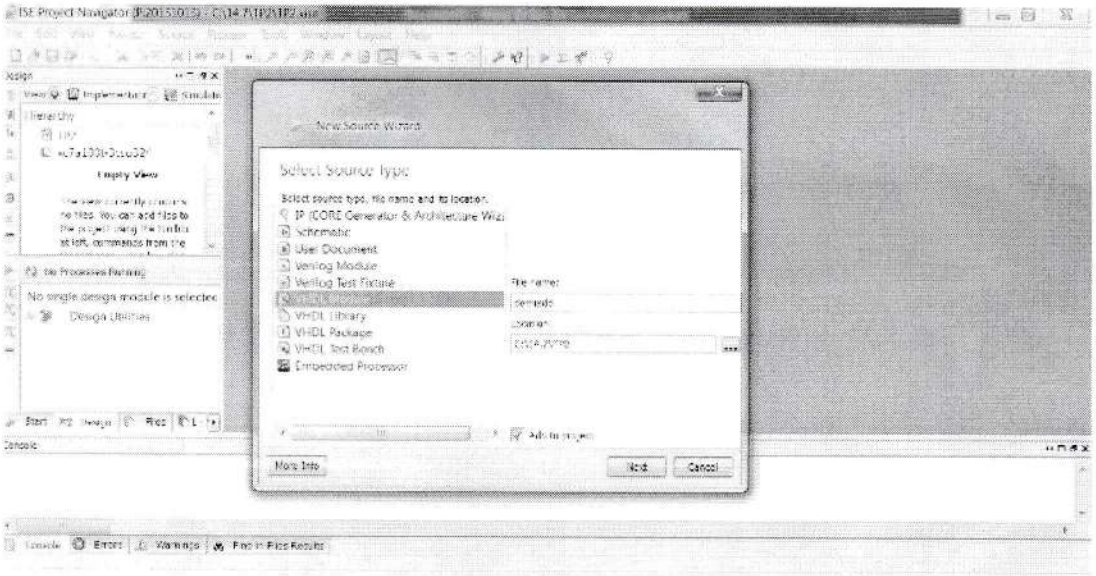


Figure 106. Choix de type de source

8. Après le choix de type de source et d'introduire le nom, on clique sur **Next**
9. La figure 107 montre la fenêtre suivante pour configurer les entrées et les sorties

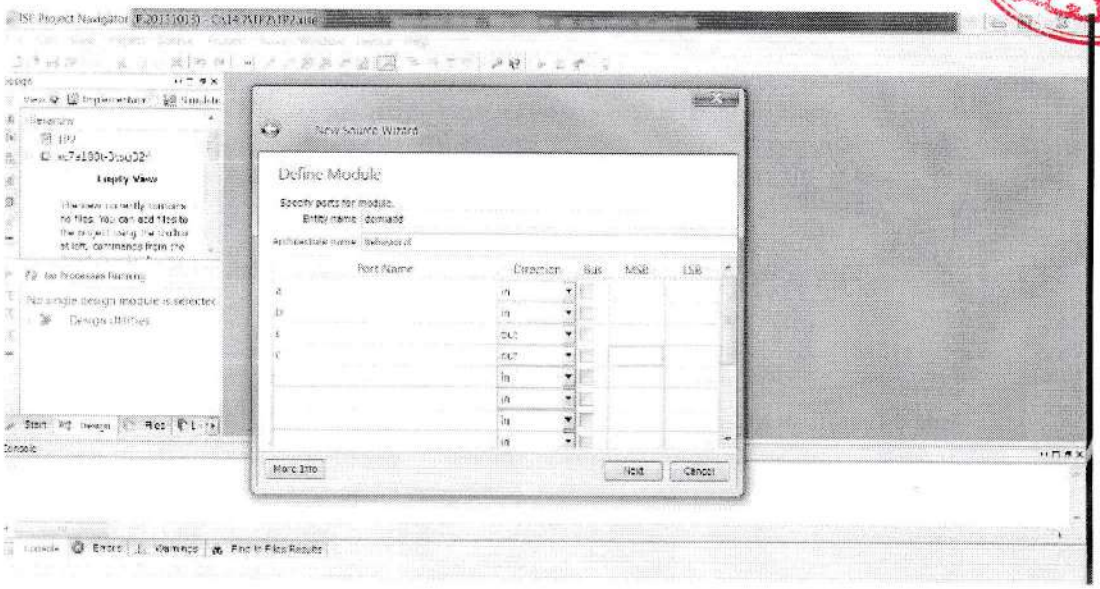


Figure 107. Configuration des entrées et sorties

10. Après la configuration des entrées et sorties, on clique sur **Next**
11. Une autre fenêtre qui résume la configuration des entrées et sorties.
12. On clique sur **Finish** pour finaliser la création de fichier VHDL

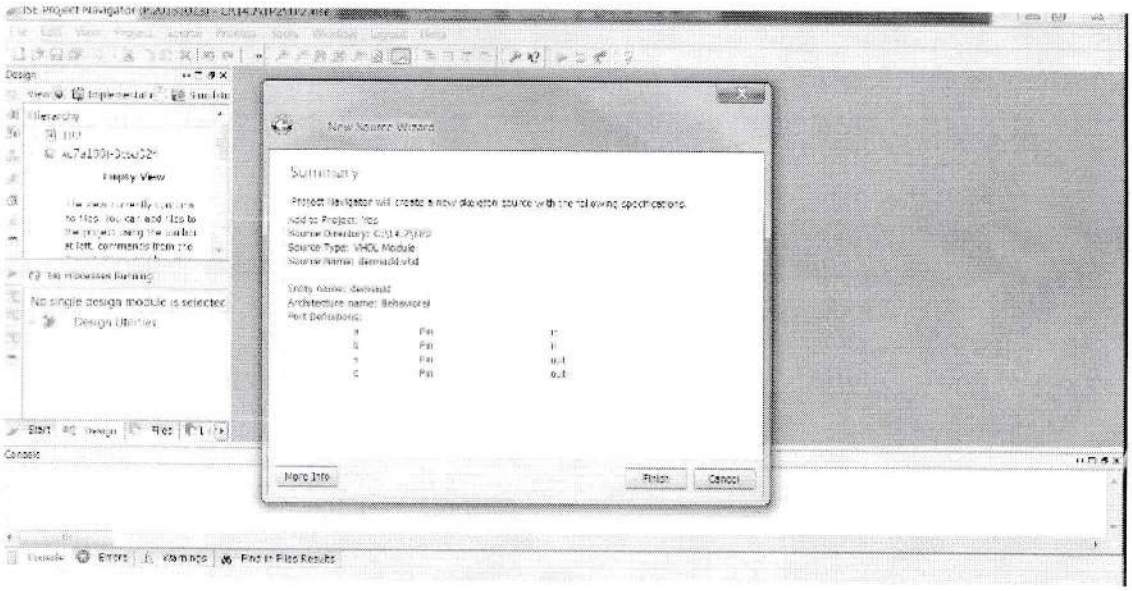


Figure 108. Récapitulation du fichier VHDL

13. La création du fichier est achevée, mais il reste à compléter la programmation

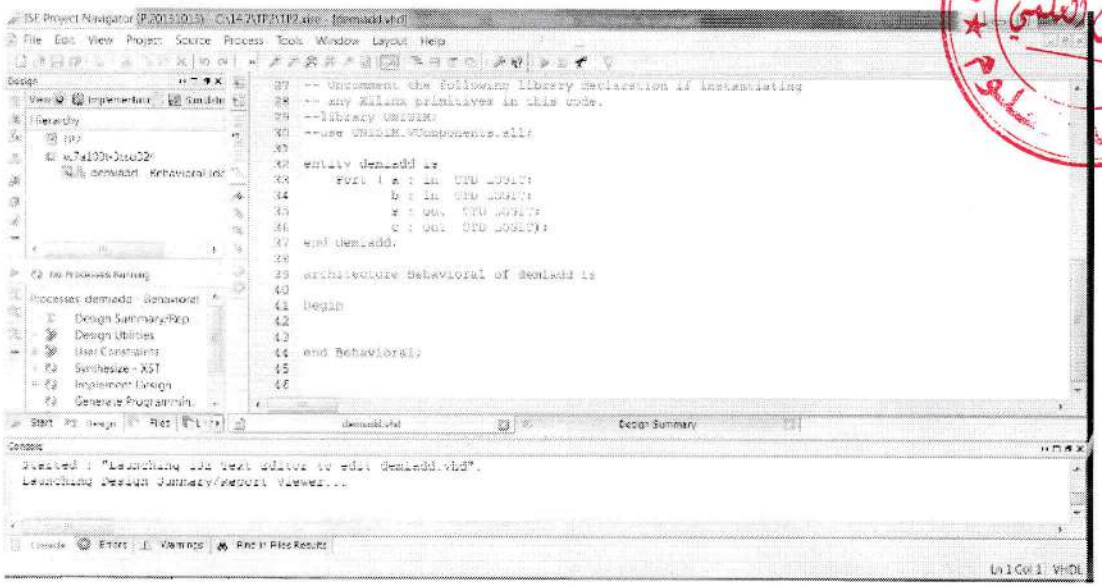


Figure 109. Le fichier ***demiadd.vhd***

14. La table suivante montre le fichier **demiadd.vhd** (incomplet)

```
----------------------
-- Company:
-- Engineer:
--
-- Create Date:     10:50:26 07/12/2021
-- Design Name:
-- Module Name:     demiadd - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
----------------------------------------------------------
----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if
-- instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

```
--use UNISIM.VComponents.all;

entity demiadd is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           s : out  STD_LOGIC);
end demiadd;

architecture Behavioral of demiadd is

begin

end Behavioral;
```

*Table 11. Le fichier **demiadd.vhd** (incomplet)*

15. Maintenant reste à compléter le fichier **demiadd.vhd** par les équations entre les entrées et sorties

16. La table 9 la partie à compléter :

```
architecture Behavioral of demiadd is
begin
s <= a xor b;
c <= a and b;
end Behavioral;
```

Table 12. La partie du fichier est compléter

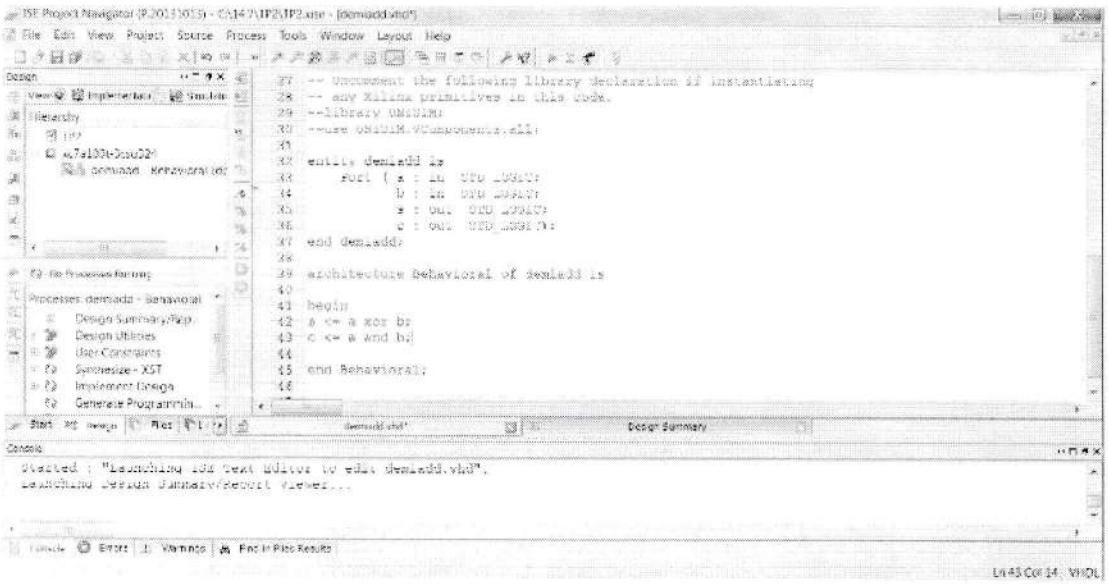


*Figure 110. Le fichier **demiadd.vhd** est compléter*

المركز الجامعي نور البشير
المجلس العلمي
معهد

17. Etape suivante est de vérifier le programme en cliquant sur **Synthesize-XST**, on doit vérifier que le programme a été bien vérifié avec succès

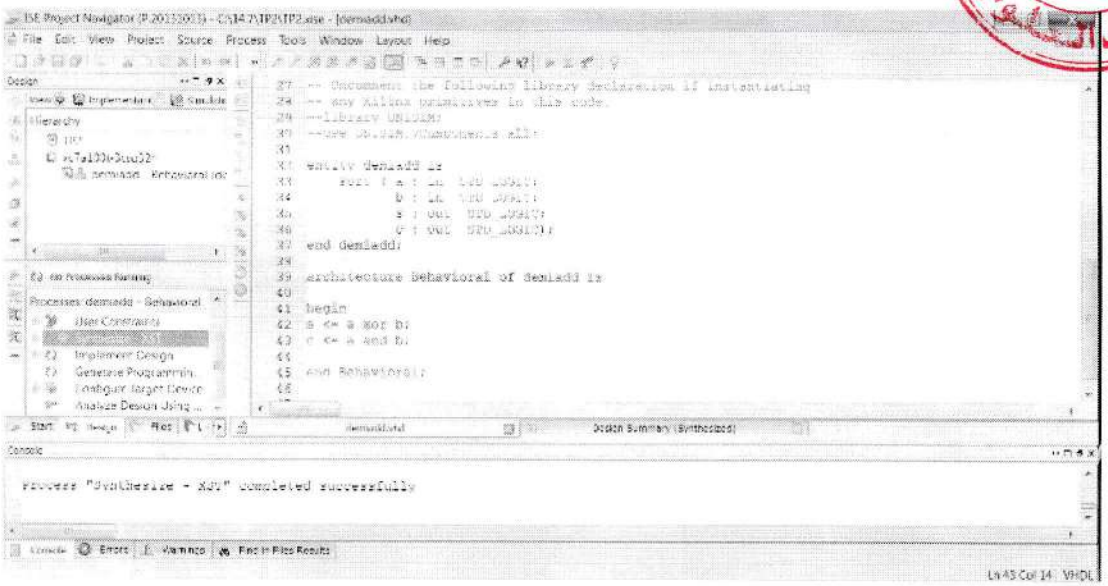


Figure 111. La vérification du programme

18. Après on va exécuter **Check Syntax** et vérifier s'il est exécuter avec succès

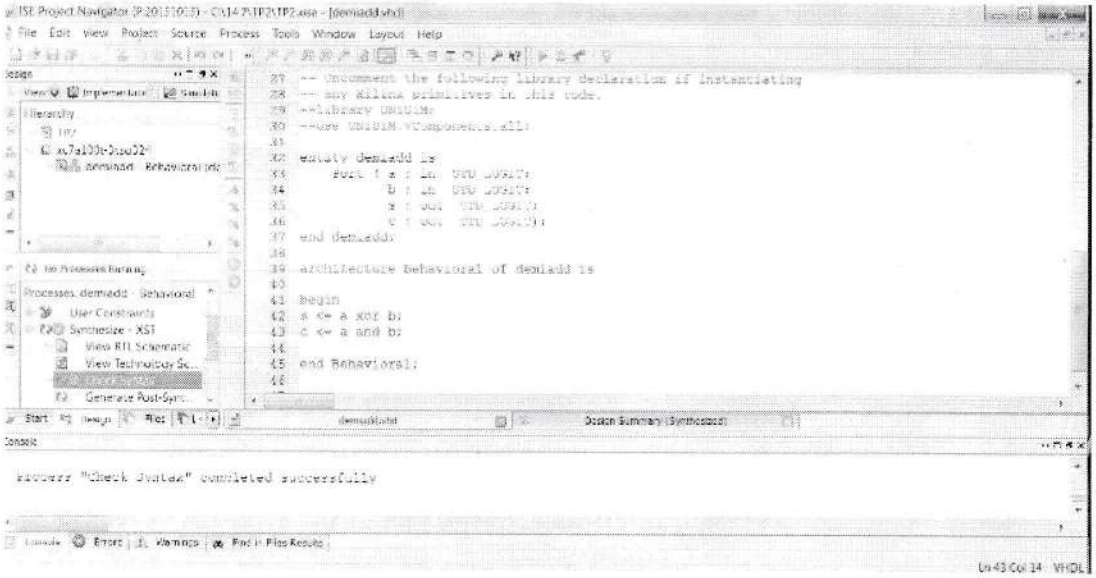


*Figure 112. Exécution de **Check Syntax***

المركز الجامعي نور البشير البيض
المجلس العلمي
معهد العلوم

19. L'étape qui va suivre c'est créer une autre source de programme en cliquant sur **Project+New Source**

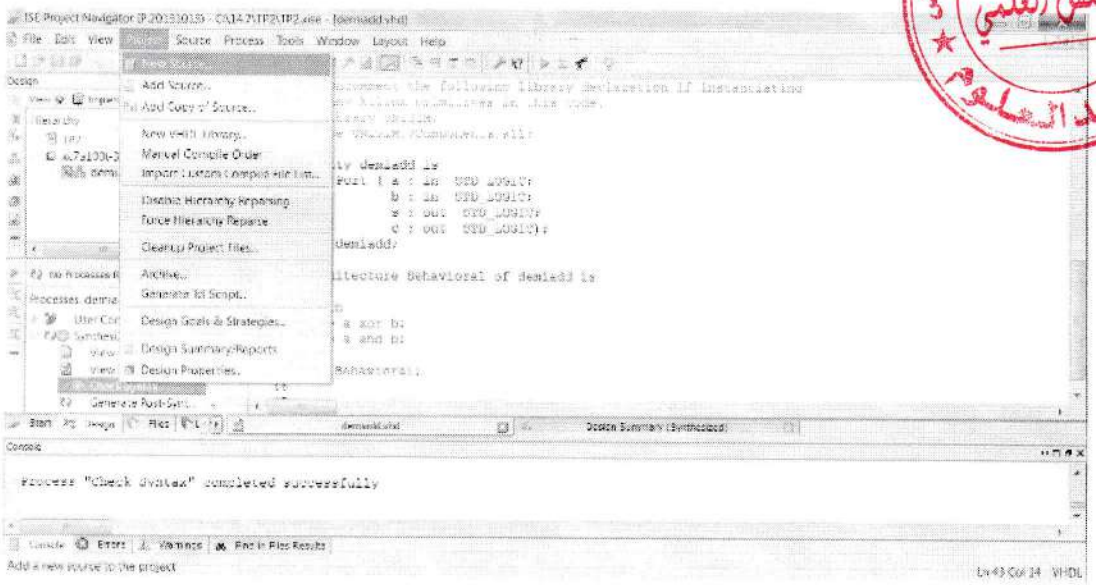


Figure 113. La création d'une autre source

20. La figure 114 montre plusieurs types de source de programme, on choisit **VHDL Test Bench** et on introduit le nom du fichier, on choisit **demi_add** et on clique sur **Next**

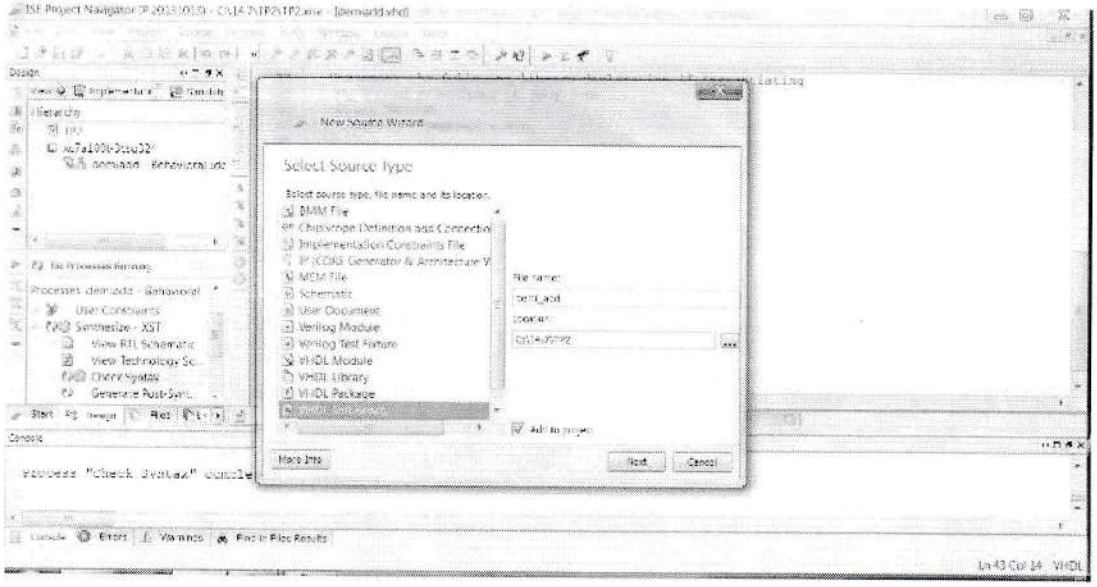


*Figure 114. Création du fichier **VHDL Test Bench***

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

21. L'étape suivante est d'associer les deux fichiers on cliquant sur **Next**

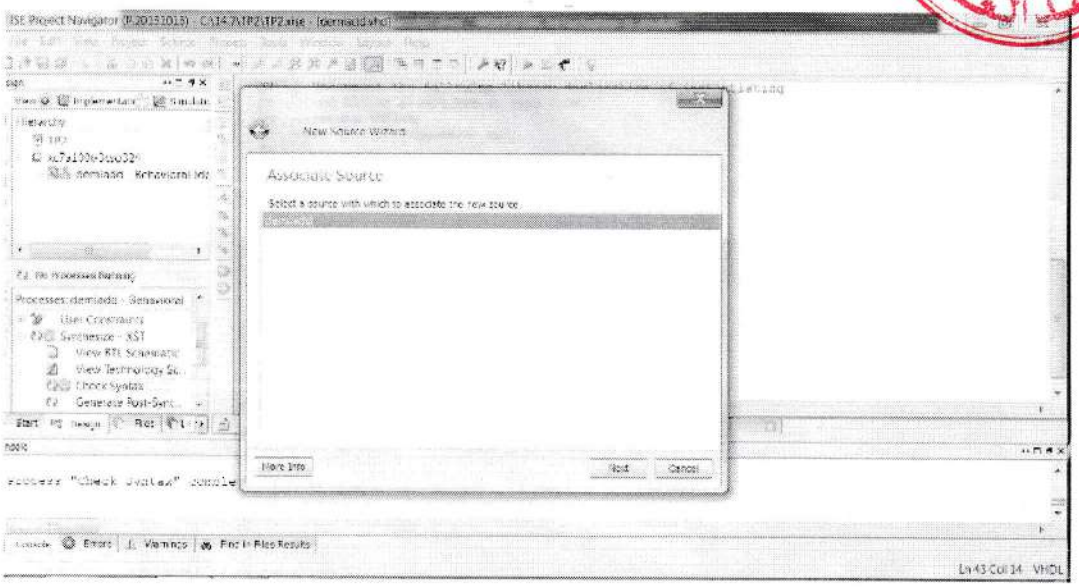


Figure 115. Association des deux fichiers

22. Récapitulation de la création du fichier **demi_add.vhd** après on clique sur **Finish**

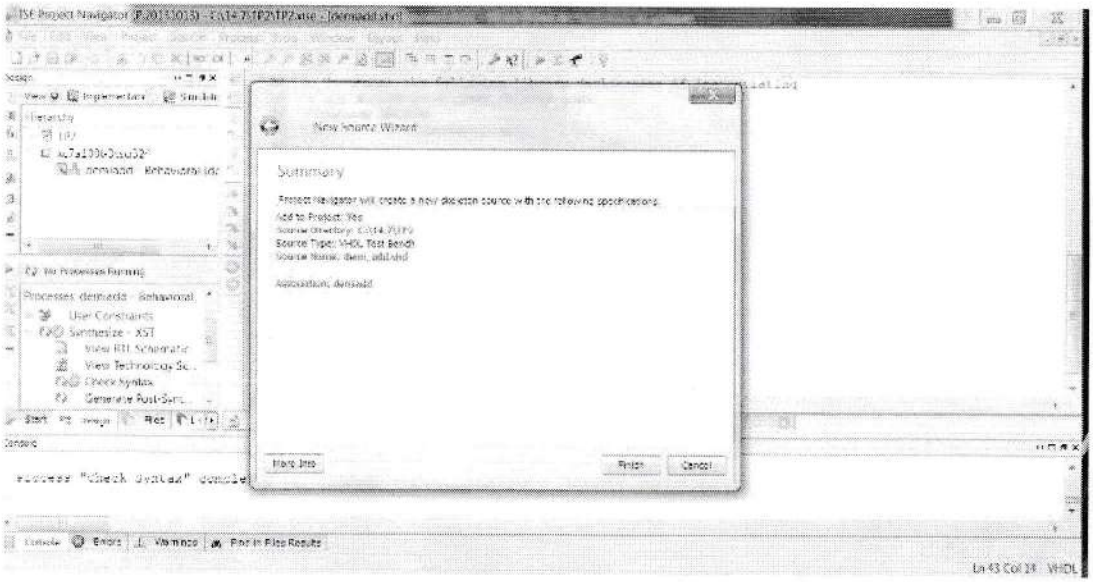


*Figure 116. Récapitulation du fichier **demi_add.vhd***

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

23. La figure 117 montre que le fichier a été bien créé et on doit le compléter (programmer)

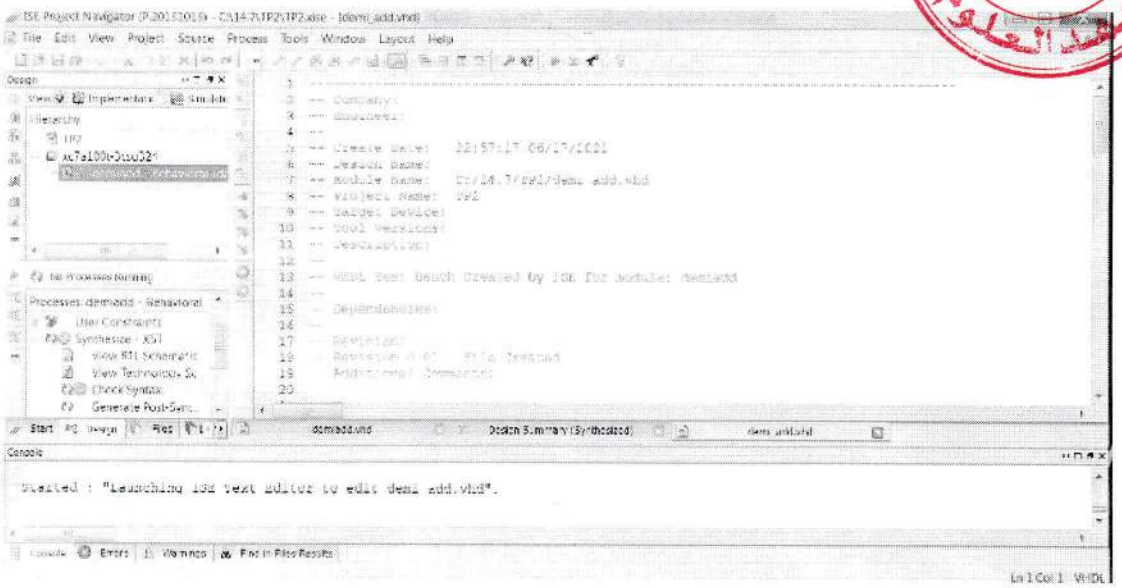


Figure 117. Création du fichier ***demi_add.vhd***

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    15:11:50 08/15/2021
-- Design Name:
-- Module Name:    C:/14.7/TP1/demi_add.vhd
-- Project Name:   TP2
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: porteand
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using
types std_logic and
-- std_logic_vector for the ports of the unit under test.
Xilinx recommends
```

```
-- that these types always be used for the top-level I/O of a
design in order
-- to guarantee that the testbench will bind correctly to the
post-implementation
-- simulation model.
------------------------------------------------------------
------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY demi_add IS
END demi_add;

ARCHITECTURE behavior OF demi_add IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT demiadd
    PORT(
         a : IN  std_logic;
         b : IN  std_logic;
         s : OUT  std_logic
         c : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal a : std_logic := '0';
   signal b : std_logic := '0';

                                                --Outputs
   signal s : std_logic;
          c : std_logic;
   -- No clocks detected in port list. Replace <clock> below
with
   -- appropriate port name



BEGIN
                                                -- Instantiate the
Unit Under Test (UUT)
   uut: demiadd PORT MAP (
          a => a,
          b => b,
          s => s
```

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

```
            c => c
        );

    -- Clock process definitions


    -- Stimulus process
    stim_proc: process
    begin
        -- hold reset state for 100 ns.
          wait for 100 ns; a<= '0'; b<= '0';
          wait for 100 ns; a<= '0'; b<= '1';
          wait for 100 ns; a<= '1'; b<= '0';
          wait for 100 ns; a<= '1'; b<= '1';
    wait for 100 ns;
    end process;



        -- insert stimulus here

END;
```

*Table 13. Le fichier **demi_add.vhd***

24. La table 10 affiche le fichier **demi_add.vhd** créer et modifié
25. La figure 118 montre le fichier **demi_add.vhd** programmé
26. La case **Simulation** doit être coché

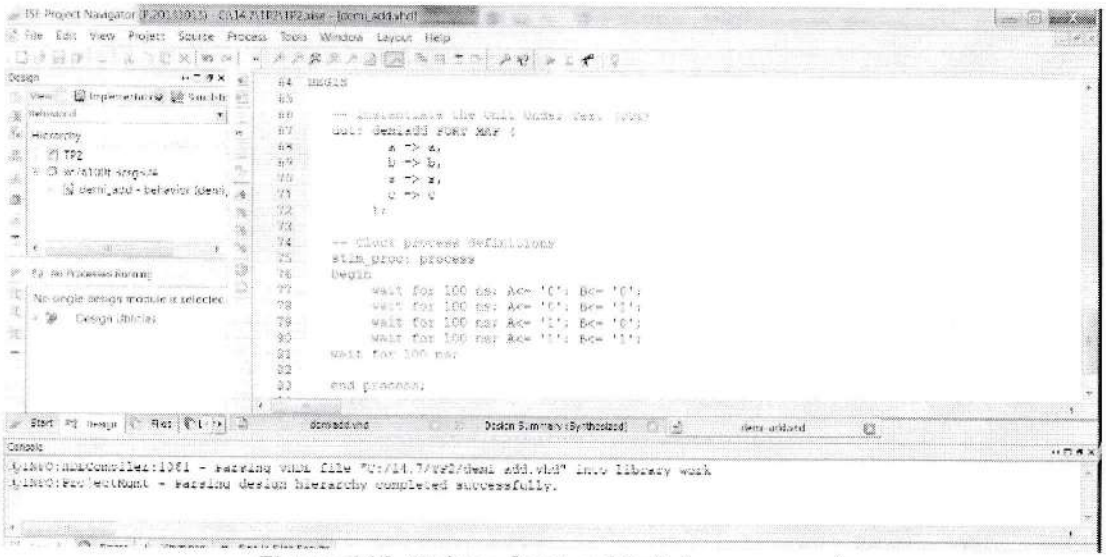


*Figure 118. Fichier **demi_add.vhd** programmé*

27. L'étape qui suivre on doit vérifier le programme on cliquant sur **Behavioral check Syntax**

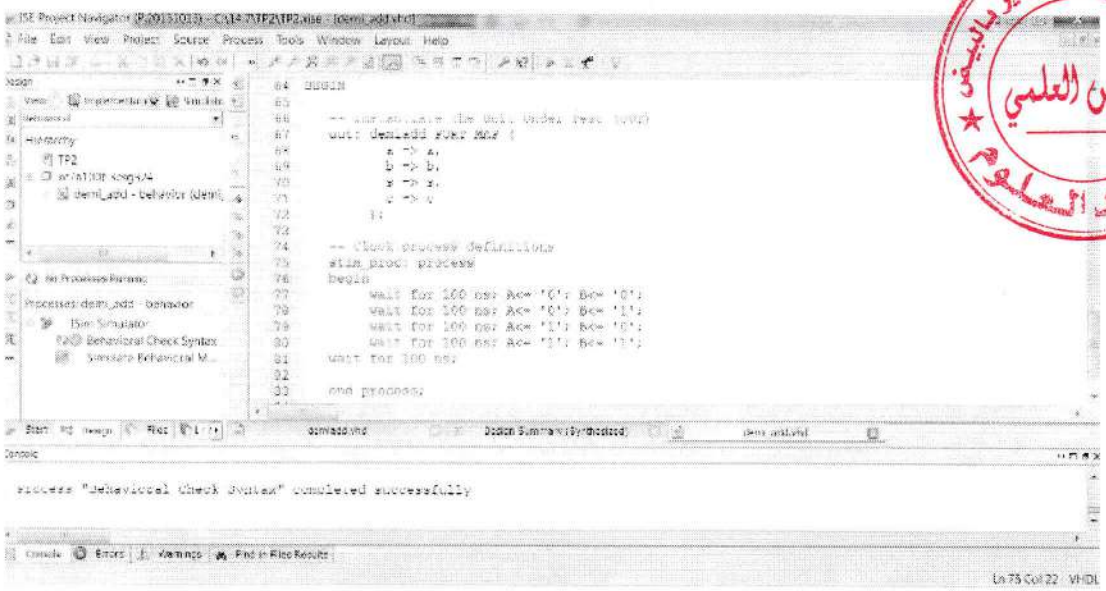


*Figure 119. Vérification **Behavioral Check Syntax***

28. La vérification a été parfaite (case verte) figure 119

29. La dernière étape est de simuler la programmation en cliquant sur **Simulate Behavioral Model**

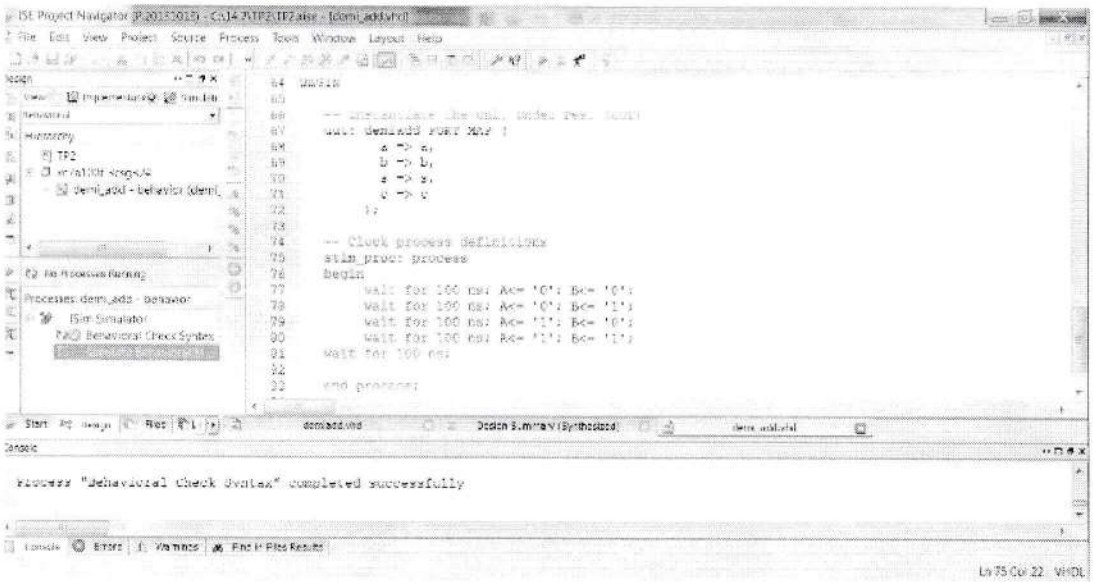


Figure 120. Le model de simulation est compléter pour le demi additionneur

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

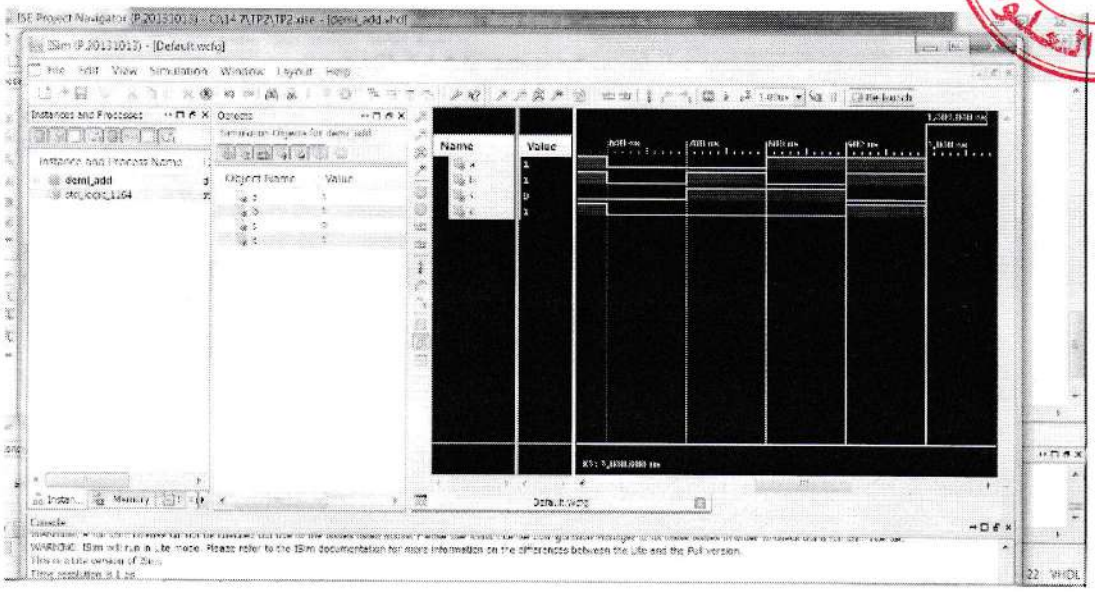


Figure 121. Le chronogramme d'un demi-additionneur

30. La figure 121 montre le chronogramme d'un demi-additionneur

المركز الجامعي نور البشير بالبيض
المجلس العلمي
معهد العلوم

Bibliographie

[1] J.-P. Deschamps, G.D. Sutter, E. Cantó, Guide to FPGA implementation of arithmetic functions, Springer Science & Business Media, 2012.

[2] P.P. Chu, FPGA Prototyping by VHDL Examples: Xilinx MicroBlaze MCS SoC, John Wiley & Sons, 2017.

[3] V.A. Pedroni, Circuit design and simulation with VHDL, MIT press, 2010.

[4] B.J. LaMeres, Introduction to logic circuits & logic design with VHDL, Springer, 2019.