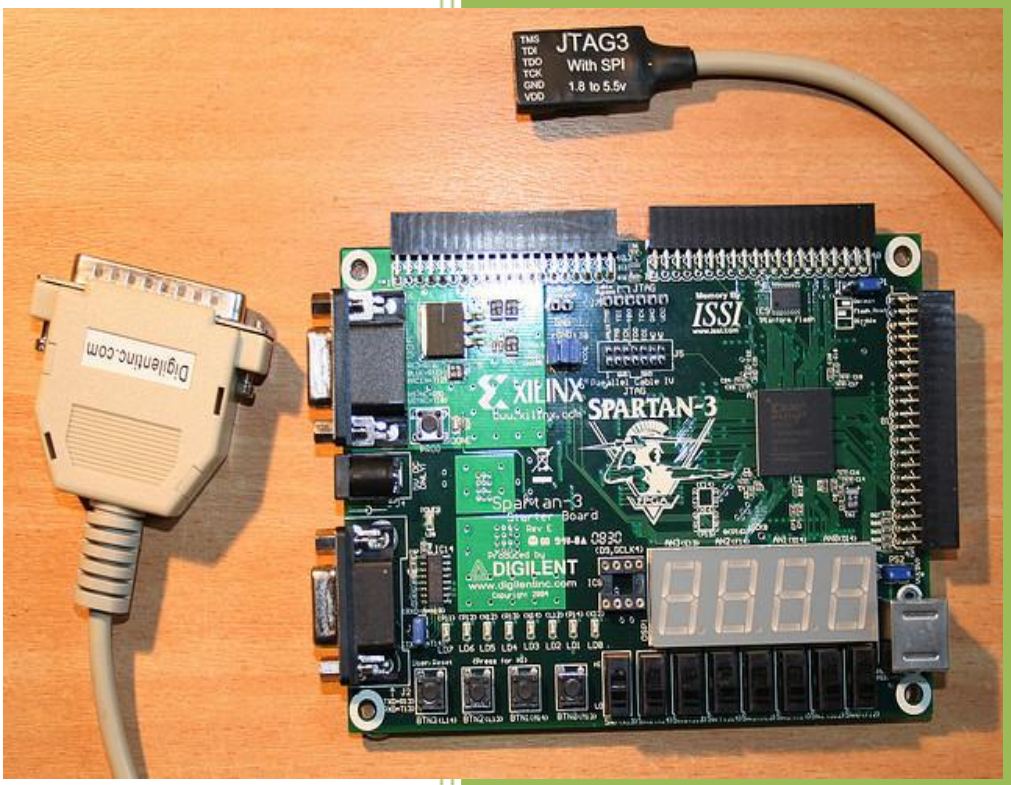
Année Universitaire 2014-2015

Ministères de l'enseignement supérieur
Institut Supérieur des Etudes Technologiques de Gabès

SUPPORT DE COURS FPGA



TAYARI LASSAAD
MAITRE TECHNOLOGUE A ISET GABES
E-mail :lassaad.tayari@isetn.rnu.tn

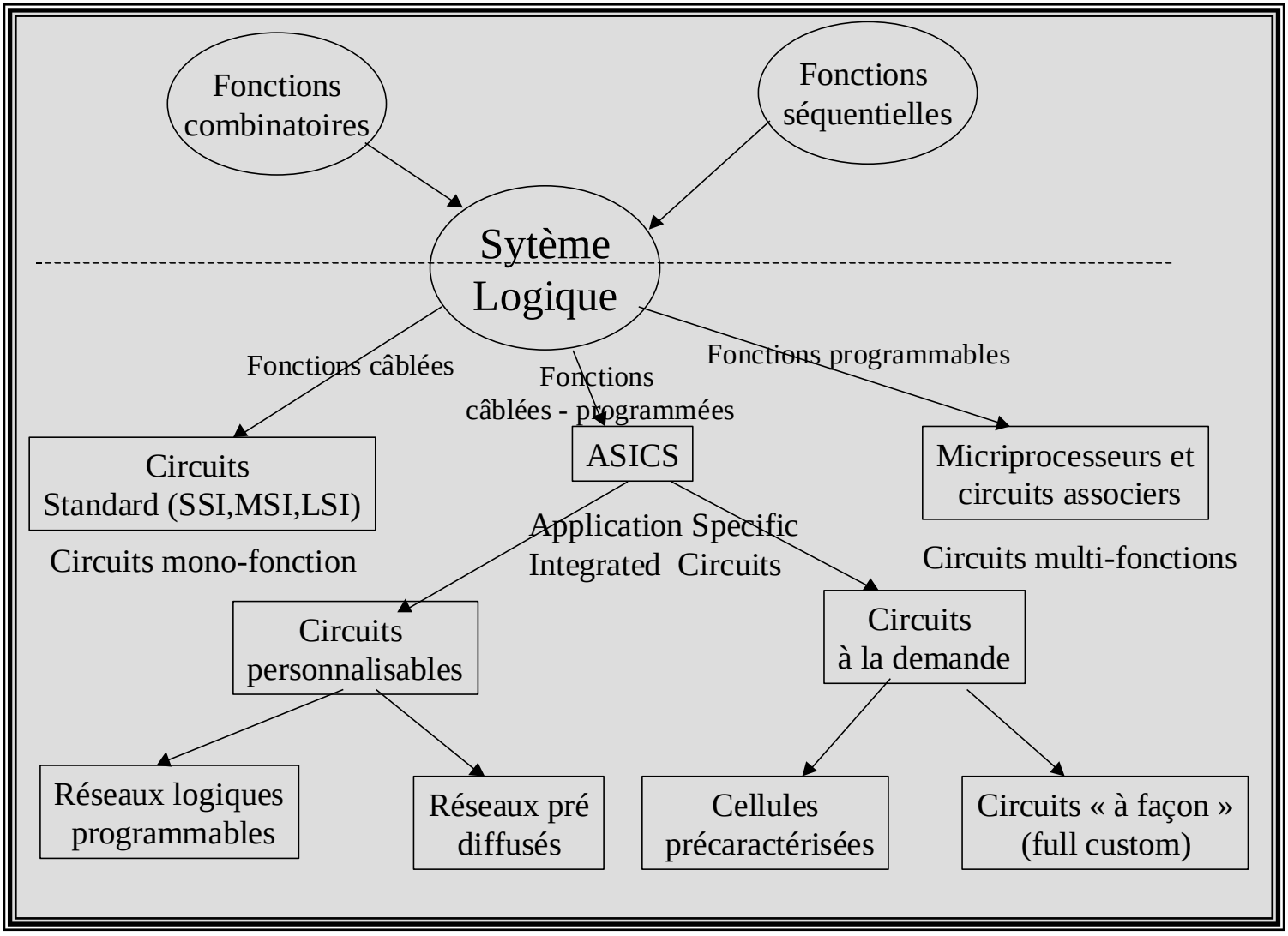
SOMMAIRE

Chapitre I

Classification des systèmes logiques

I. Introduction :

Un système logique utilise les fonctions combinatoires et séquentielles. Actuellement de multiples types de circuits coexistent dans ces systèmes. Le domaine de complexité de ces circuits s'étend de quelques opérateurs élémentaires (SSI) « Simple Scale Integration », à quelques dizaines (MSI) « Medium Scale Integration ». Pour atteindre des milliers (LSI) « Large Scale Integration » et même des centaines de milliers (VLSI).



On peut classer ces circuits en plusieurs familles compte tenu de leurs fonctionnalité :

II. Les circuits standard :

Ils donnent les fonctions fixes d'usage courant.

Il existe trois types de circuits standard :

a) Les circuits à faible degré d'intégration (SSI) (ET, OU, Bascules,… ,etc)
b) Les circuits à densité moyenne d'intégration (multiplexeurs, décodeurs, encodeurs, registres a décalage, compteurs, …,etc
c) Les circuits à grande densité d'intégration (LSI) (mémoires a semi-conducteurs (Ram, ROM, PROM, EPROM, …,etc)).

Ces circuits sont caractérisés par le fait qu'ils ne réalisent qu'une seule fonction.

III. Les circuits spécifiques : (ASIC : Application Specific Integrated Circuits)

Ce sont des circuits adaptés à des applications particulières. On distingue quatre familles :

a) 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.

La programmation de ce type de circuits s'effectue à l'aide d'un logiciel spécifique (PALASM, PLDesigner…etc) et d'un appareil adéquat appelé programmateur.

b) Les réseaux prédiffusés (Gate Array) (FPGA ou LCA)

A l'état vierge, un tel circuit comprend un grand nombre de cellules. Chaque cellule contient soit des portes logiques, soit des transistors et des résistances. Ces protes ou ces éléments ne sont pas interconnectés. La programmation de ce type de circuits revient à assurer la connexion entre ses différents composants.

Le développement de ce type de circuits nécessite l'utilisation d'un outil d'IAO (Interconnexion Assistée par ordinateur).

c) Les cellules pré caractérisées : (Cell Array ou Standard Cells)

Dans ce type de circuits il n'y a pas de diffusion préalable d'éléments actifs sur le silicium. La programmation de ces circuits se fait à l'aide d'un outil de CAO qui doit être très puissant.

d) les circuits à « façon » full custum

Ces circuits sont analogues aux cellules pré caractérisées mais qui sont beaucoup plus compliqués et qui représentent des circuits semi-fini au niveau physique. C'est à dire que leurs programmation se fait par gravure directe. Cette opération est faite par le constructeur.

IV. Les microprocesseurs et circuits associés

Ces sont des circuits multifonctions qui offrent une grande souplesse d'utilisation. Ils fonctionnent selon le principe de la logique programmée enregistrée. Le fonctionnement de ces circuits se fait en fonction d'une suite d'instructions enregistrée dans une mémoire. Le microprocesseur appelle ces instructions (actions) d'une façon séquentielle.

Chapitre II

Les réseaux logiques combinatoires programmables

I/ INTRODUCTION

La réalisation pratique d'un système logique consiste à utiliser les composants disponibles sur le marché. Le concepteur doit décomposer le système en blocs fonctionnels. Le problème consiste à optimiser le choix des fonctions utilisées par rapport à des critères de coût, performance, nombre de composants, complexité de connexion, … etc.

L'utilisation de circuits dont la fonction est adaptable (programmable) par l'utilisateur représente une solution élégante à ce problème.

On distingue trois principaux types de fonctions à circuits programmables :

1. Les fonctions combinatoires ou séquentielles programmables par fusible (Réseaux Logiques Programmables).
2. Les réseaux de cellules pré diffusées (Gate Array).
3. Les circuits à la demande (Cell Array et Full Custum)

II/ STRUCTURE DES RESEAUX LOGIQUES COMBINATOIRES

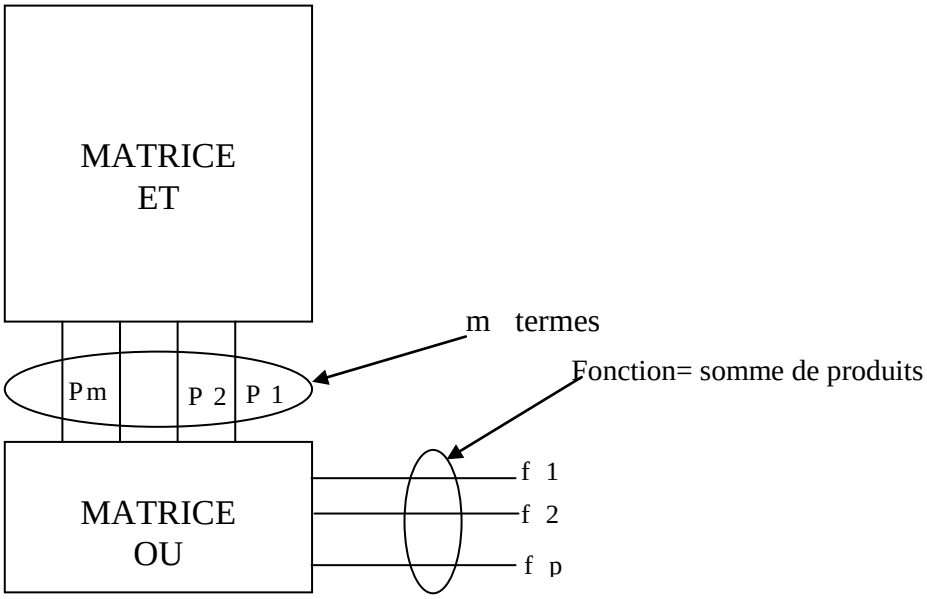
La plupart de ces composants sont constitués d'une matrice d'opérateurs ET qui génère les produits des variables d'entrées et de leurs compléments et d'une matrice d'opérateurs OU qui somme les produits.

Suivant le type de circuits, l'une ou l'autre ou les deux matrices sont programmables.

REMARQUE :

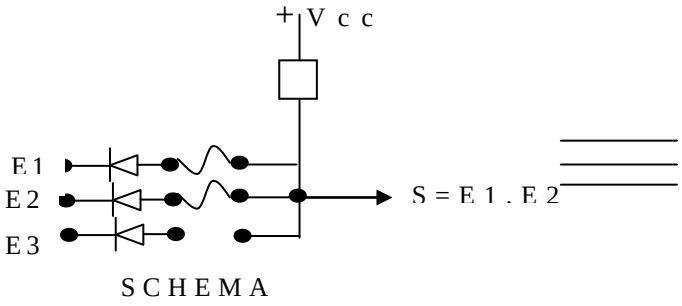
D'après le cours de systèmes logiques :

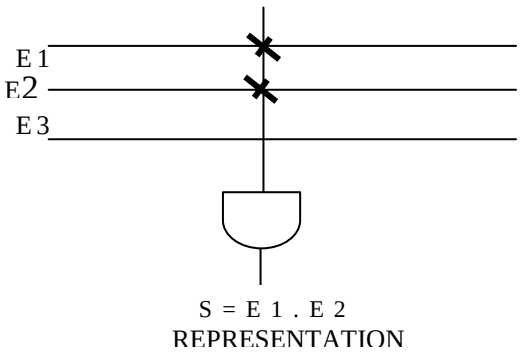
Toute fonction logique se représente sous forme d'une expression de **somme de produits** ou de **produit sommes**.

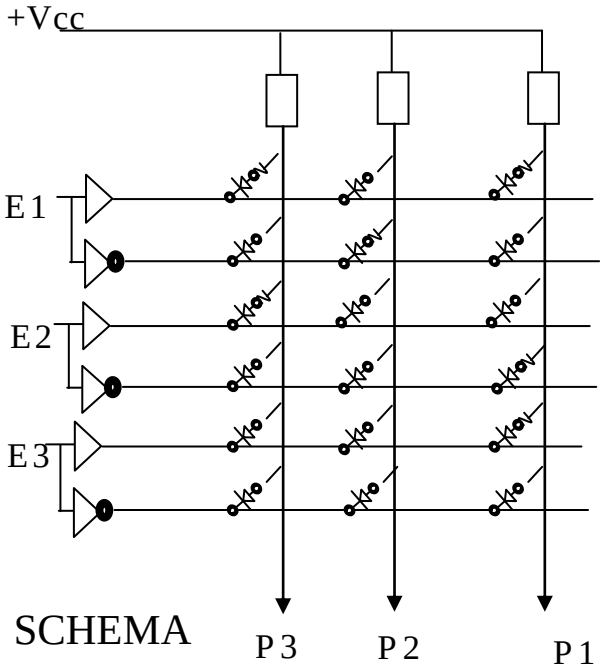


1/ Réalisation de la matrice ET

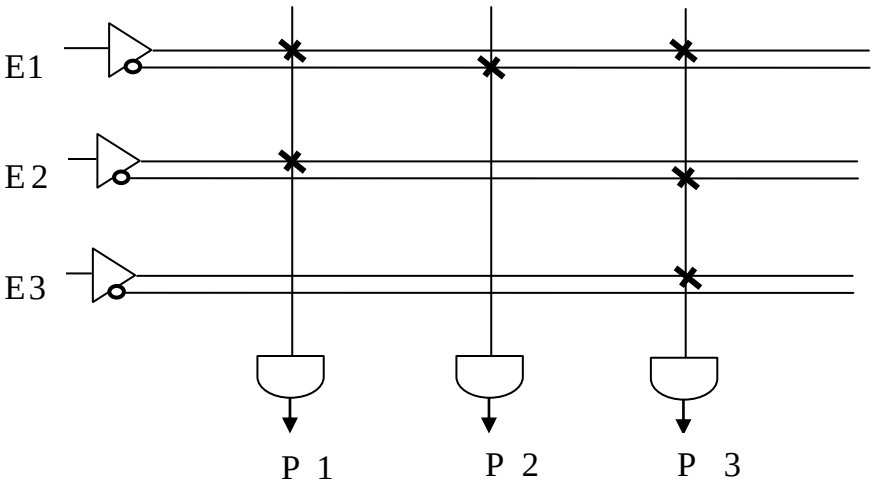
On utilise le principe de ports ET à diodes. La programmation s'effectue graçe à des fusibles placés en série avec des diodes.







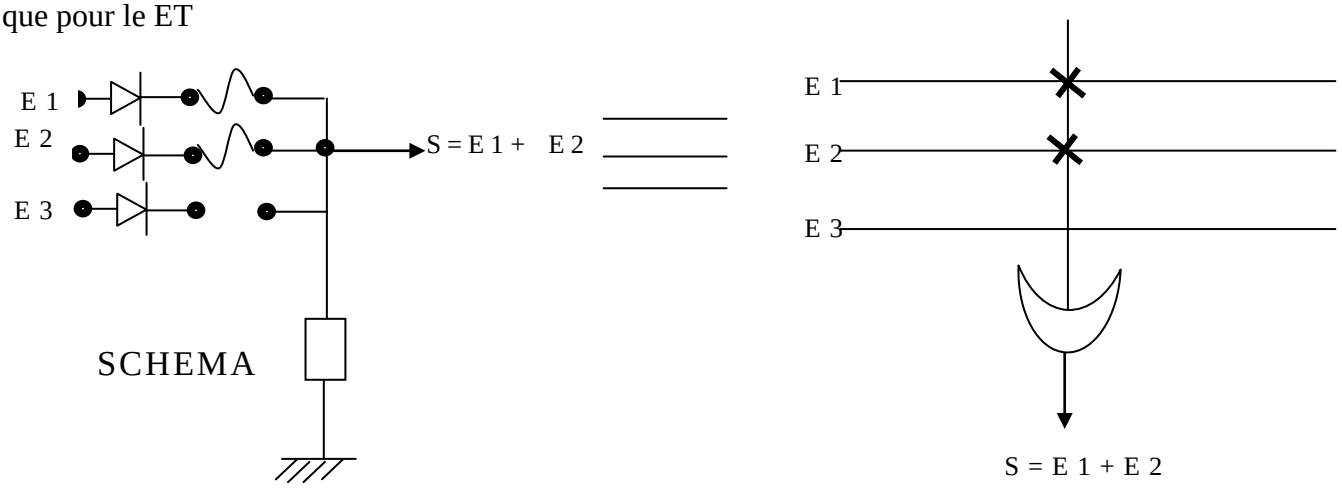
$$P1 = E1.E2$$
$$P2 = \overline{E1}$$
$$P3 = E1.\overline{E2}E3$$



REPRESENTATION

2/ Réalisation de la matrice OU

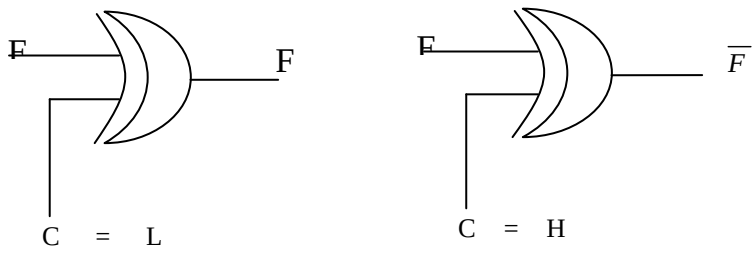
On utilise le principe d'une porte OU à diode. La programmation se fait de la même manière que pour le ET



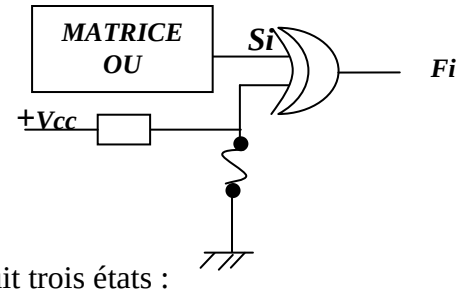
REPRESENTATION

3/ Les circuits de sortie

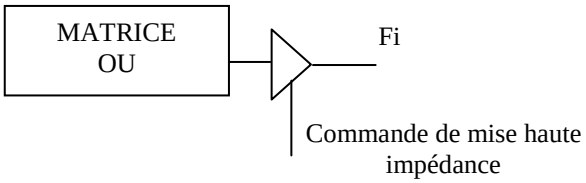
On adjoint un circuit de sortie à la matrice OU. Il peut s'agir d'un circuit d'inversion. dans ce cas , on utilise l'opérateur OU exclusif dont une entrée est utilisée comme commande d'inversion.



On aura donc :



Le circuit de sortie peut aussi être un circuit trois états :

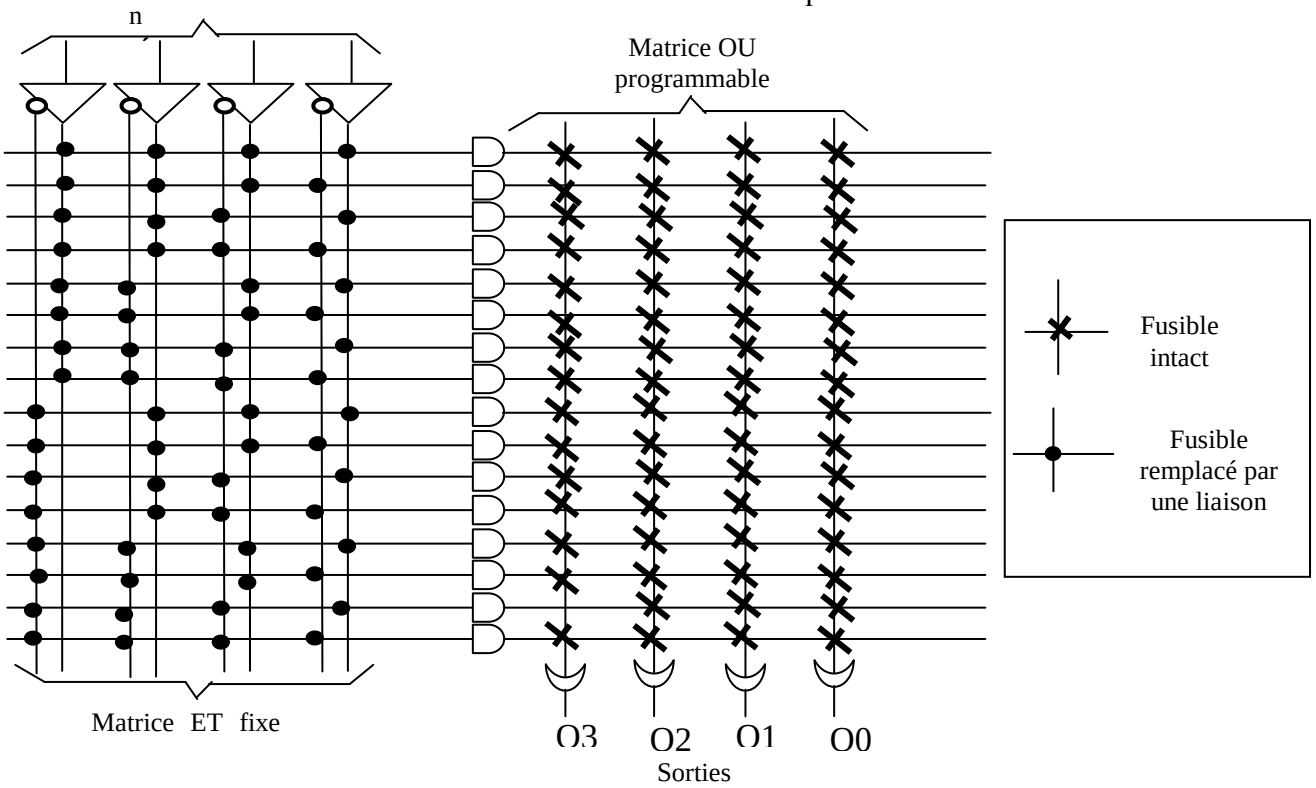


III/ CLASSIFICATION DES RESEAUX LOGIQUES COMBINATOIRES

1/ Les PLE ou PROM (Programmable Logic Elements ou Programmable Read Only Memory)

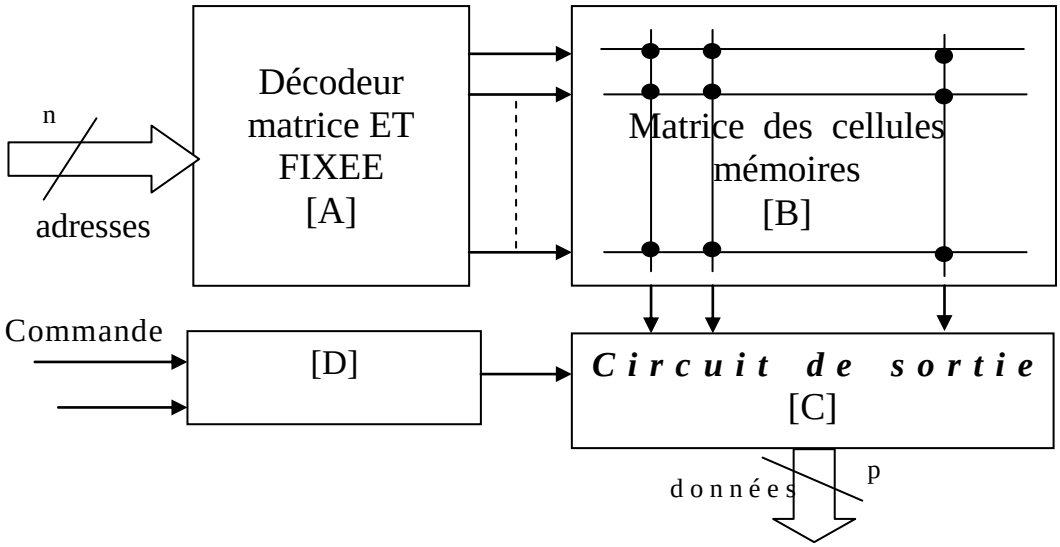
Ces circuits disposent :

- D'une matrice ET fixé et complète. Donc le nombre de termes produits est égal à 2^n pour n entrées. La matrice ET a la structure d'un décodeur binaire à n bits.
- D'une matrice OU programmable.
- D'un circuit de sortie trois états à commande unique.



Une fonction F est donc réalisée en programmant sa table de vérité. Donc en mémorisant la valeur de F pour toutes les combinaisons des entrées.

La forme générale est donc :

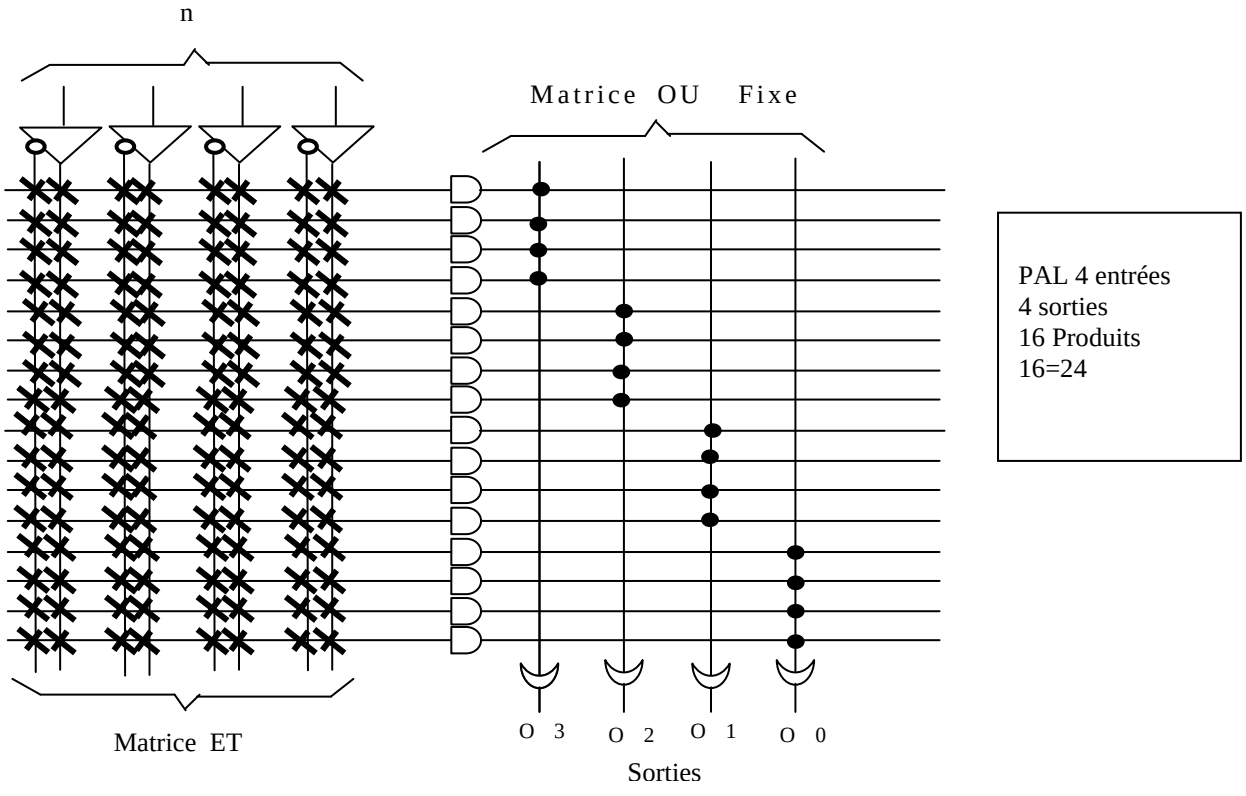


Les n fils d'adresse permettent de choisir une des 2^n lignes continues dans la matrice [B]. La sélection est réalisé par un décodeur [A] sous le contrôle d'un circuit logique [D]. Lorsqu'une adresse « a » est placée à l'entrée, la ligne « a » est sélectionnée. Les « p » informations de cette ligne sont transmises au circuit de sortie [C].

2/ Les PAL (Programmable Array Logic)

a- Structure de principe

La structure de base d'un PAL est l'opposée de celle d'une PROM. Il dispose d'une matrice ET programmable et d'une matrice OU figée.

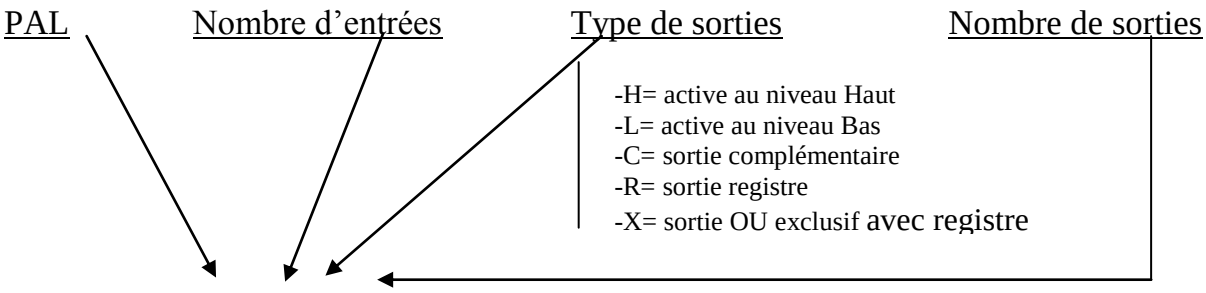


b/ Les différents types de PAL proposées par MMI

Les PAL proposées par MMI se différencient par :

- Le nombre d'entrées,
- le nombre de fonctions de sortie
- le nombre de termes produits par sortie
- la polarité des sorties.
- Les circuits à polarité de sortie directe :
- Il existe quatre : PAL 10H8,12H6, 14H4 et 16H2

Le nom du circuit indique sa constitution selon le schéma suivant :

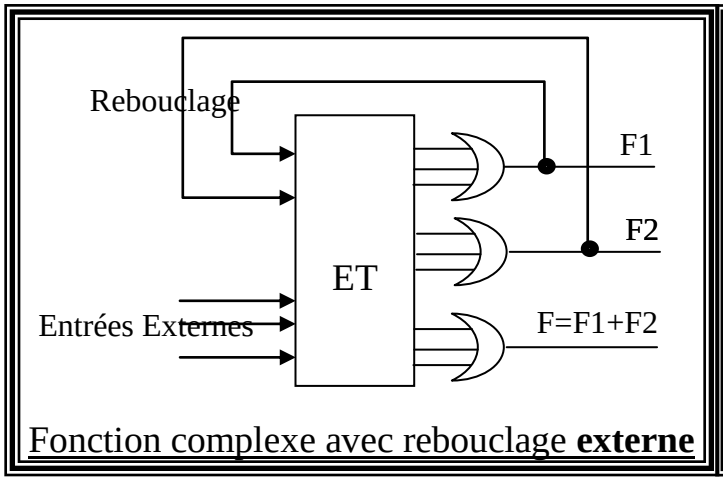


Exemple : **PAL 12 H 6**

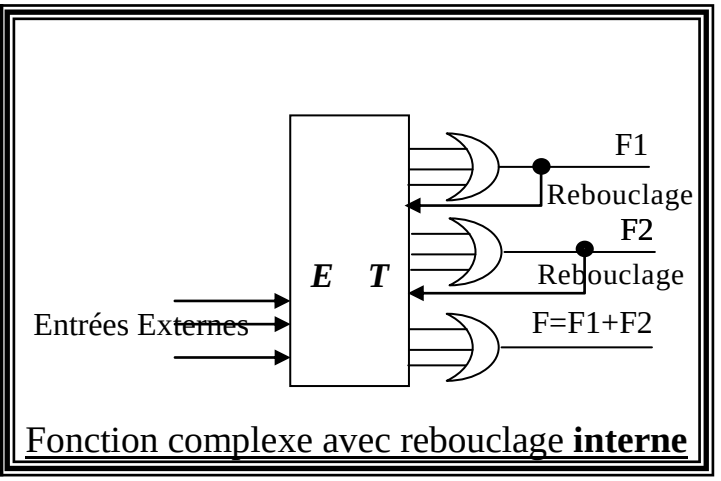
Le PAL 12H6 comprend 12 entrées, 6 sorties dont 4 possèdent deux termes produits et deux possèdent quatre termes produits.

La formule complète de ce PAL s'écrit 12H6 (4x2+2x4).

- Les circuits à polarité complémentaires : Ces circuits donnent une sortie directe et complémentaire. Il en existe deux : PAL 16C1 et 20C1.
- Les circuits à polarité inverse : Le circuit de sortie est un NOR qui donne une polarité inverse noté L. Ces circuits sont : PAL 10L8, 12L6, 14L4, 16L2, 12L10, 14L8, 16L6, 18L4 et 20L2.
- Les circuits avec rétroaction :Lorsqu'une fonction est trop complexe pour le nombre de termes produits associés à ne sortie, on calcule séparément des groupes de produits que l'on réunit en suite.



Fonction complexe avec rebouclage **externe**



Fonction complexe avec rebouclage **interne**

Remarque: Si l'on reboucle systématiquement les fonctions de sortie à l'intérieur du réseau de fusibles, on garde toutes les potentialités des entrées externes. Exemple les PAL 16L8 et 20L10.

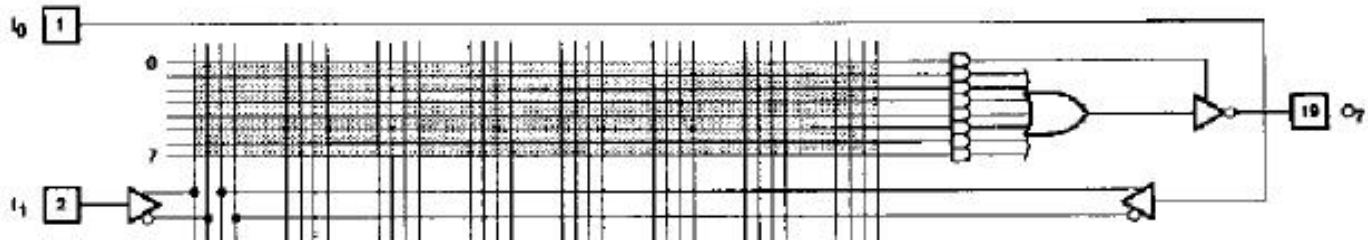
c/ Exemples de PAL combinatoires. le PAL 16L8.

Ce type de circuit est uniquement constitué de logique combinatoire. Il possède 20 broches agencées de la façon suivante :

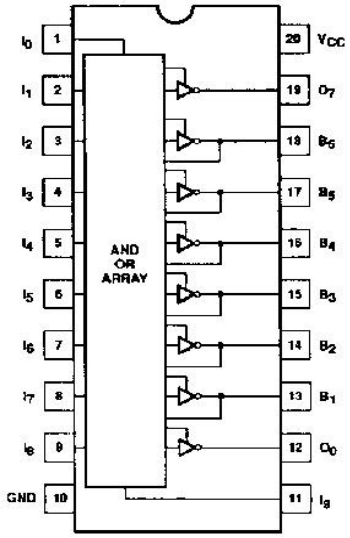
- 10 broches configurables uniquement en entrée
- 2 broches configurables uniquement en sortie
- 6 broches configurables en entrée et en sortie
- 2 broches d'alimentation.

L'ensemble des sorties provient de portes 3 états inverseuses. L'état « haute impédance » peut être commandée par l'ensemble des entrées.

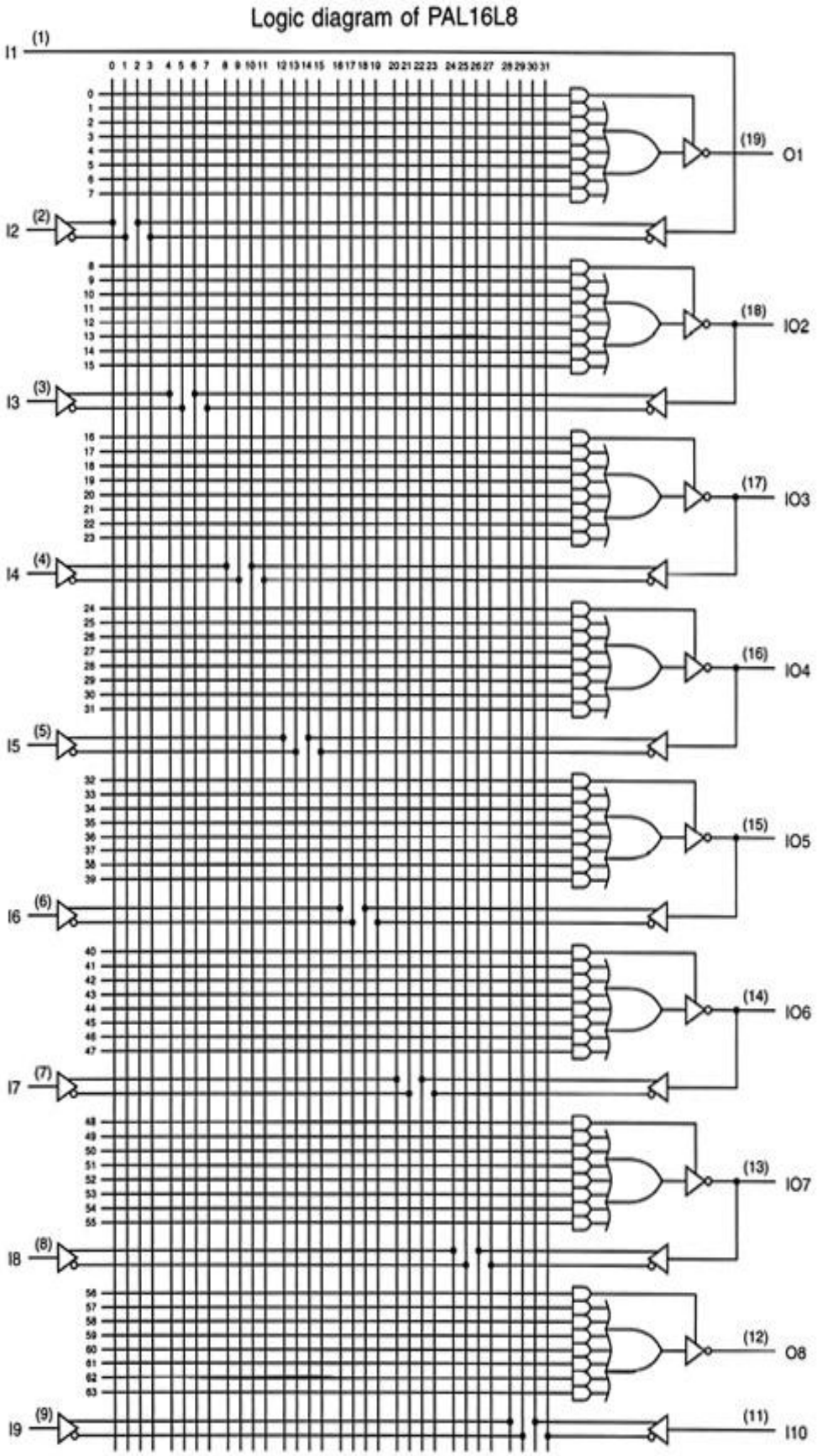
Chaque porte de la matrice « OU » possède 7 entrées. Ceci signifie que chaque sortie peut résulter, au maximum, d'une fonction « OU » entre 7 termes produits. Chaque porte de la matrice « ET » possède 32 entrées. Ceci signifie que chaque terme produit peut résulter, au maximum, d'une fonction « ET » entre 16 variables et leurs compléments.



Configuration partielle interne du PAL 16L8

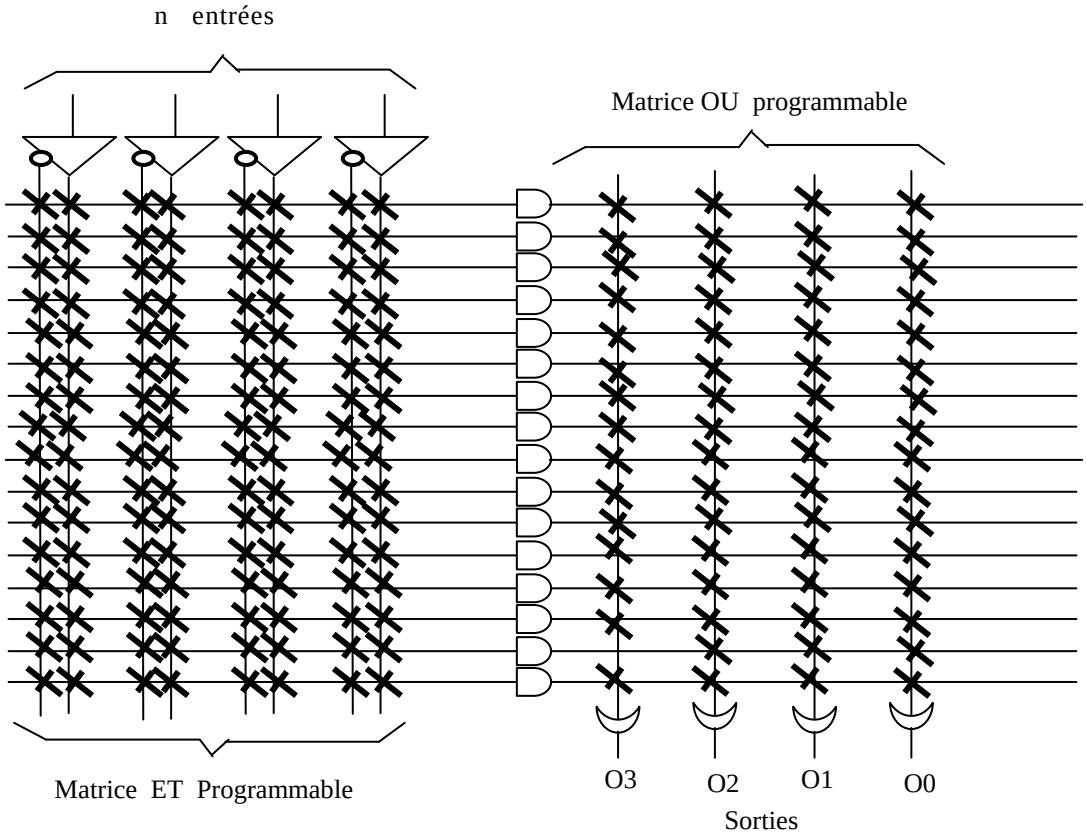


Brochage du PAL 16L8 -


Logic diagram of PAL16L8
I1 (1)
I2 (2)
I3 (3)
I4 (4)
I5 (5)
I6 (6)
I7 (7)
I8 (8)
I9 (9)
(19) O1
(18) IO2
(17) IO3
(16) IO4
(15) IO5
(14) IO6
(13) IO7
(12) O8
(11) I10

3/ Les PLA ou FPLA (Field Programmable Logic Array)

L'architecture des PLA est donnée ci-dessous :



FPLA 4 Entrées 4 Sorties 16 Produits

Les matrices ET et OU sont programmables indépendamment, ce qui offre une très grande souplesse. Ces circuits disposent également d'un circuit de sortie programmable (circuit d'inversion).

4/ Les circuits logiques Séquentiels programmables

Les constructeurs de PAL combinatoires ont développé des réseaux logiques programmables adaptés aux systèmes séquentiels. On en trouve deux types :

1. ***Les FPLS (Field Programmable Logic Sequenser)***
2. ***Les PAL séquentiels.***

4-1 LES FPLS :

Ils intègrent, en plus des matrices ET-OU classiques, des bascules dont la fonctionnalité est programmable.

4-2 LES PAL SÉQUENTIELS :

Ce sont des circuits PAL dont la matrice ET-OU est identique à celle des PAL combinatoires et qui disposent en sortie de bascules D (edge triggered) qui échantillonnent le résultat combinatoire à chaque front montant d'un signal d'horloge commun à toute les bascules.

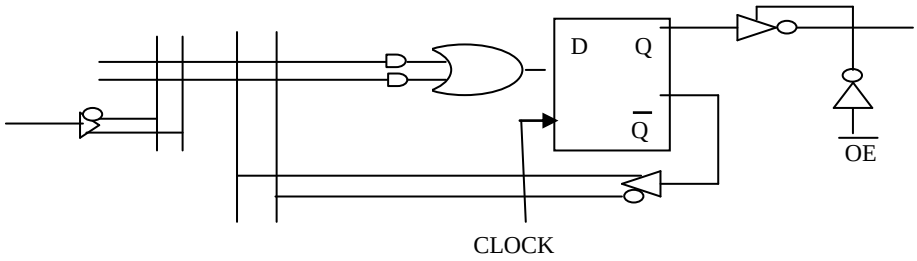
a- Les différents types de PAL séquentiels :

compte tenu de la structure de l'étage de sortie, on distingue trois types de PAL séquentiels :

1. les PAL de type R
2. les PAL de type x
3. les PAL de type V

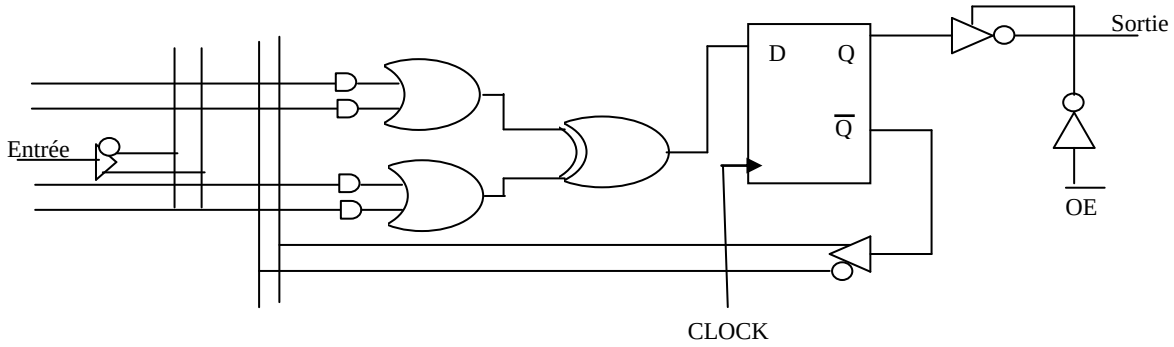
a-1- Les PAL de type R (configuration Registre) :

Ce sont des circuits pour lesquels la structure de l'étage de sortie est la suivante :



a-2- Les PAL de type X (configuration ou exclusif) :

Ce sont des circuits pour lesquels la structure de l'étage de sortie est la suivante :



a-3- Les PAL de type V (versable : macrocellule de sortie) :

Les PAL XXV8 sont des circuits pour lesquels la structure de l'étage de sortie permet l'émulation de tous les PAL combinatoires et séquentiels. La structure de la cellule de sortie de ces PAL est la suivante :

Les différentes configurations par un PAL versatile sont :

SG_0	SG_1	SG_{0X}	Type de PAL Emulé
0	1	0	16R8, 16R6, 16R4
0	1	1	16R6, 16R4
1	0	0	10H8, 12H6, 14H4, 16H2, 10L8, 12L6, 14L4, 16L2
1	0	1	12H6, 14H4, 16H2, 12L6, 14L4, 16L2
1	1	1	16L8

b-EXEMPLES DE PAL SEQUENTIELS. Le PAL 16R8.

Ce type de circuit est constitué de logique combinatoire et séquentielle. Il possède 20 broches (Doc. annexe) agencées de la façon suivante :

8 broches (n° 2 à 9) configurables uniquement en entrée

1 broche (n° 1) d'entrée d'horloge de l'ensemble des 8 bascules D

1 broche (n° 11) de validation des 8 sorties (output enable)

8 broches (n° 12 à 19) configurables en sortie et pouvant être réinjecter en entrée

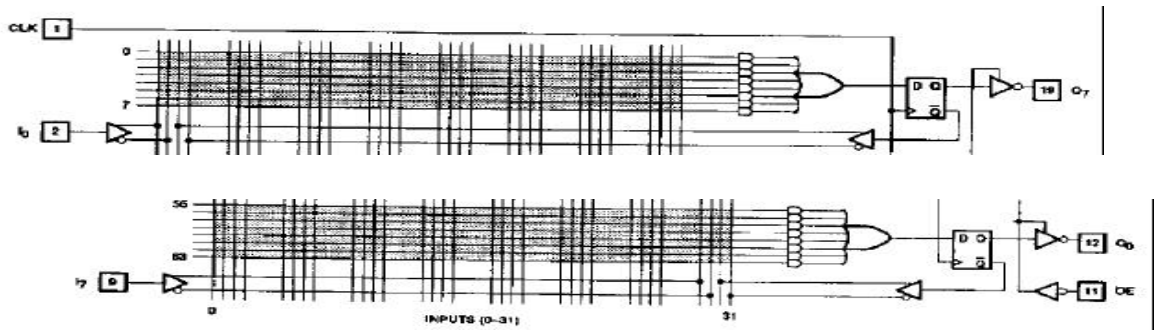
2 broches d'alimentation (n° 10 et 20).

L'ensemble des sorties provient de portes 3 états inverseuses provenant elles-mêmes de bascules D (figure 9). L'état haute impédance est commandée par l'entrée OE (broche n°11).

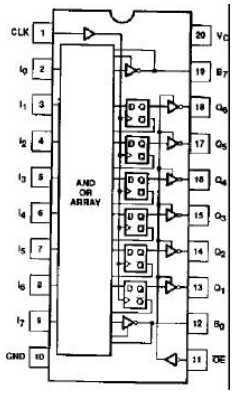
Chaque porte de la matrice « OU » possède 7 entrées. Ceci signifie que chaque sortie peut résulter, au maximum, d'une fonction « OU » entre 7 termes produits.

Chaque porte de la matrice « ET » possède 32 entrées. Ceci signifie que chaque terme produit peut résulter, au maximum, d'une fonction « ET » entre 16 variables et leurs compléments.

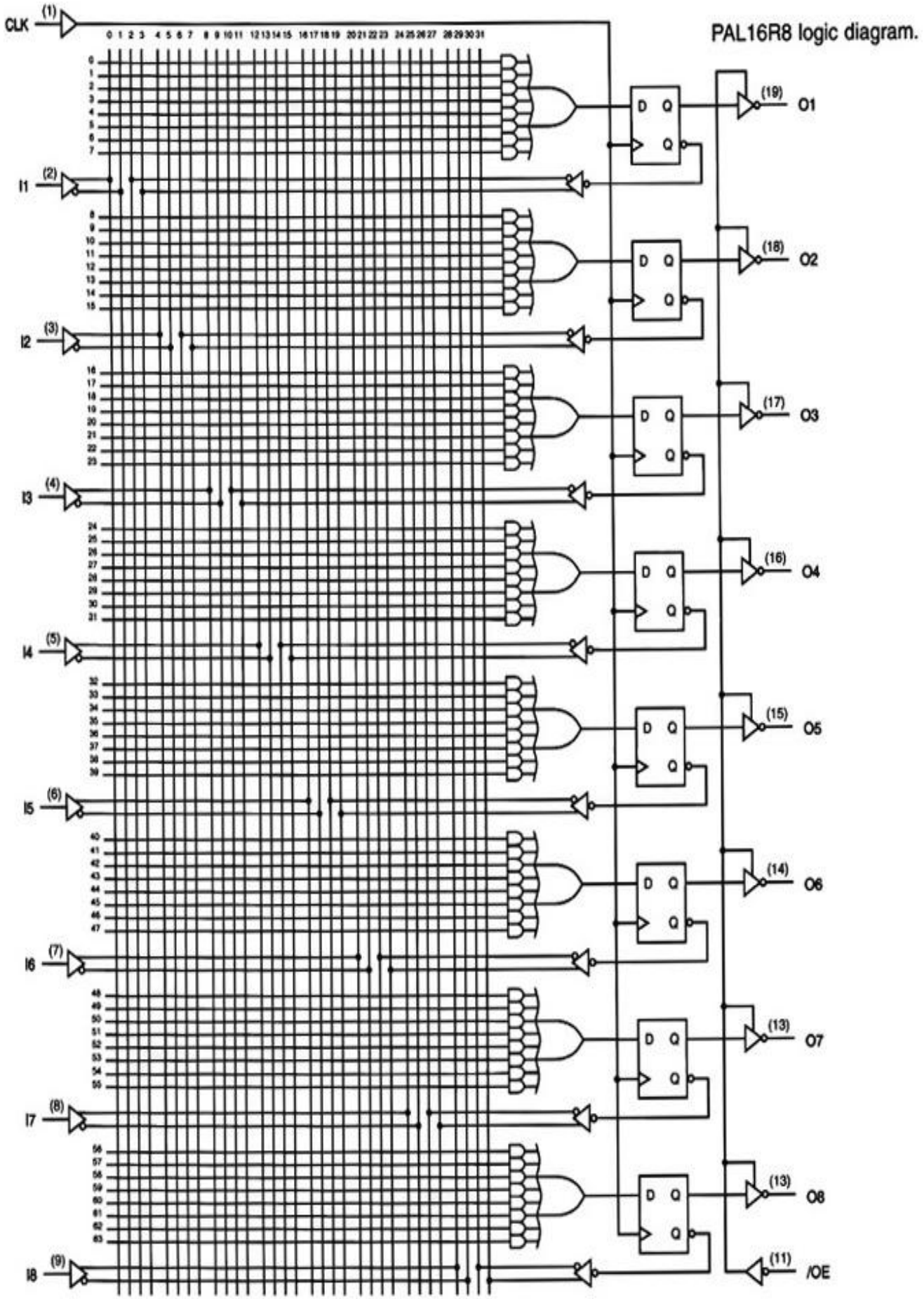
En résumé on peut dire que les huit sorties de ce circuit proviennent d'un registre trois états. La mise en haute impédance de ce dernier est commandée par la broche OE et la mémorisation est activée par les fronts montants de l'horloge CLK.



Configuration partielle interne du PAL 16R8 -



Brochage du PAL 16R6 –


PAL16R8 logic diagram.
CLK
(1)
I1
(2)
I2
(3)
I3
(4)
I4
(5)
I5
(6)
I6
(7)
I7
(8)
I8
(9)
D Q
Q
(19)
O1
(18)
O2
(17)
O3
(16)
O4
(15)
O5
(14)
O6
(13)
O7
(13)
O8
(11)
/OE

5/ LES GALS (GENERIC ARRAY LOGIC).

5-1 PRESENTATION.

L'inconvénient majeur des PALs est qu'ils ne sont programmables qu'une seule fois. Ceci impose un gaspillage important de ces circuits lorsqu'on veut développer un nouveau produit. La firme 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 appelant « PAL CMOS ».

Par soucis de remplacer les PALs, LATTICE a équipé la plupart de ses GALs de macro cellules programmables permettant d'émuler n'importe quel PAL. Ces structures de sortie sont donc du type « Versatile » (V).

5-2 PROTECTION CONTRE LA DUPLICATION.

Les GAL sont dotés d'un bit de sécurité qui peut être activé lors de la programmation empêchant ainsi toute lecture du contenu du circuit. Ce bit est remis à zéro seulement en effaçant complètement le GAL.

Il est aussi constitué d'un ensemble de huit octets, appelé signature électronique, pouvant contenir des informations diverses sur le produit.

5-3 REFERENCE DES GAL.

Le nombre de types de GAL est de 8. Les deux 2 derniers-nés présentent une structure plus particulière que nous n'aborderons pas dans ce document. Les six plus anciens sont différenciés par leurs nombres d'entrées et de sorties. Ils possèdent une structure de sortie soit du type « Versatile » soit du type « Registre asynchrone ».

Référence :

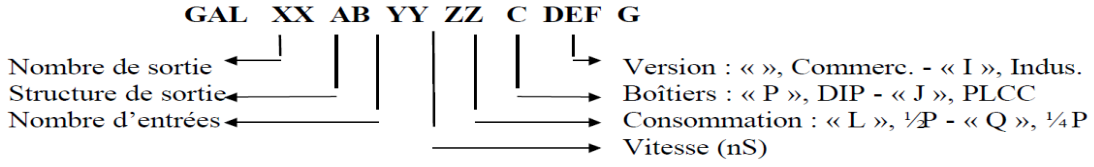


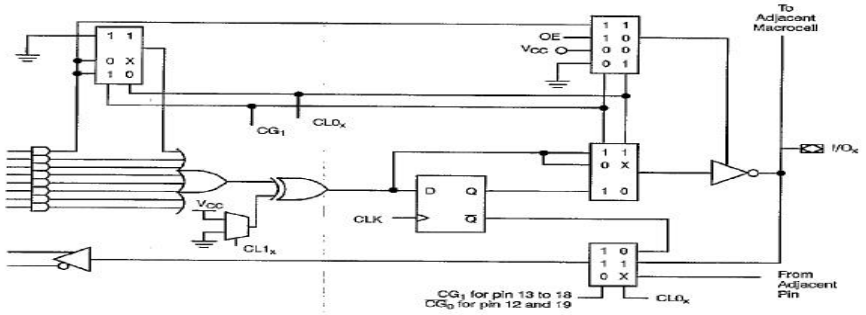
Tableau résumant les différents types de GAL :

Référence	Nombre de	Vitesse(nS)	Consommation(mA)	Remarque
GAL 16V8	20	10, 15 ou20	55 ou 115	Macro-cellule (1)
GAL 18V10	20	15 ou 20	115	//
GAL 20V8	24	10, 15 ou25	55 ou 115	//
GAL20RA10	24	15 ou 20	115	Registre asynchrone (1)
GAL 22V10	24	15, 20 ou25	130	Macro-cellule (1)
GAL 26V12	28	15 ou 20	130	//
GAL 6001	24	30 ou 35	150	Macro-cellule (1) - Type FPLA (2)
ispGAL16Z8	24	20 ou 25	90	Macro-cellule (1) - Programmable en

* (1) : structure de sortie.
* (2) : Matrices « OU » et « ET » programmables.
* (3) : Circuit reprogrammable à tout moment par liaison série.

5-4 MACRO CELLULE DE SORTIE (OLMC).

Comme cela a été spécifié auparavant, ces structures de sortie sont programmables et permettent d'émuler n'importe quelle autre structure de sortie. Elles possèdent en tout 2 bits de programmation communs à toutes les cellules (CG1 et CG0) et 2 bits spécifiques à chaque cellule (CL0x et CL1x).

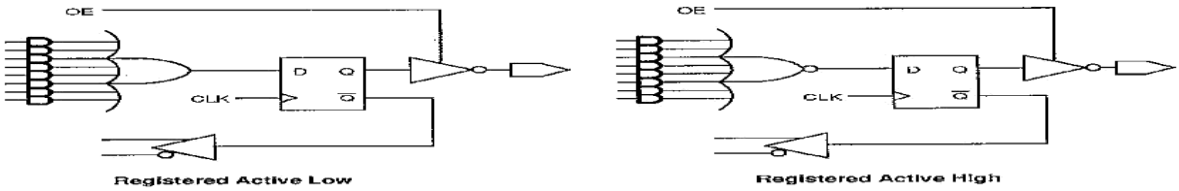


macro-cellule OLMC du GAL 16V8 -

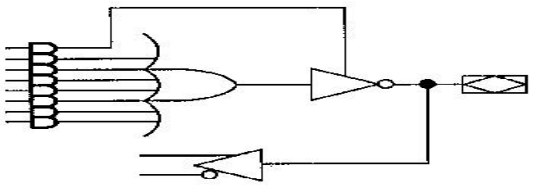
Configurations possibles de la macro-cellule pour le GAL 16V8 :

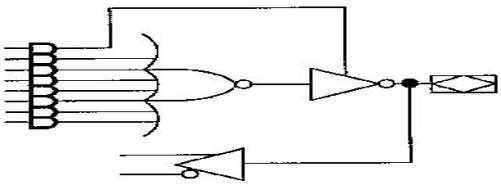
Configuration de la structure de sortie	Circuit PAL émulé
Registre synchrone - sortie 3 états (C1)	16R8
Entrée / Sortie combinatoire - sortie 3 états (C2)	16R4 - 16R6
Entrée et/ou Sortie combinatoire (C3)	10L8 - 12H6
Entrée combinatoire (C4)	12L6
Entrée / Sortie combinatoire - sortie 3 états (C2)	16L8 - 16H8

Remarque *:* En configuration « Registre » et en n'utilisant pas l'état haute impédance des portes 3 états, il faut relier la broche OE (n°11) à VCC.

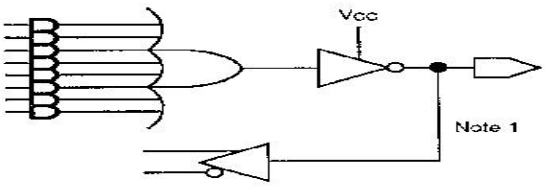


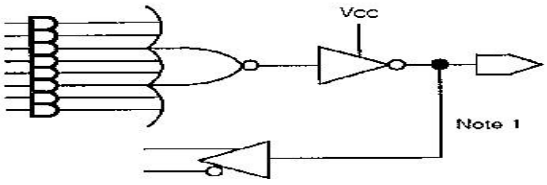
registre synchrone / sortie 3 états -





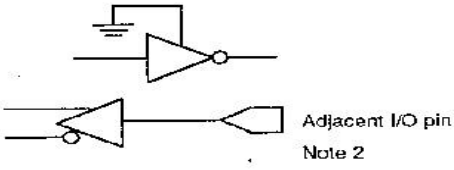
E/S combinatoire / sortie 3 états -





E/S Combinatoire -

Notes:
1. *Feedback is not available on pins 15 and 16 in the combinatorial output mode.*
2. *This configuration is not available on pins 15 and 16.*

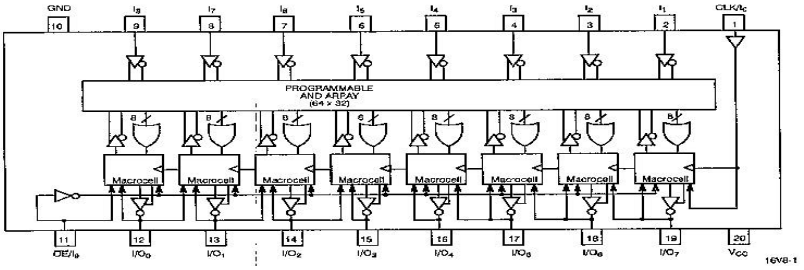


Entrée combinatoire -

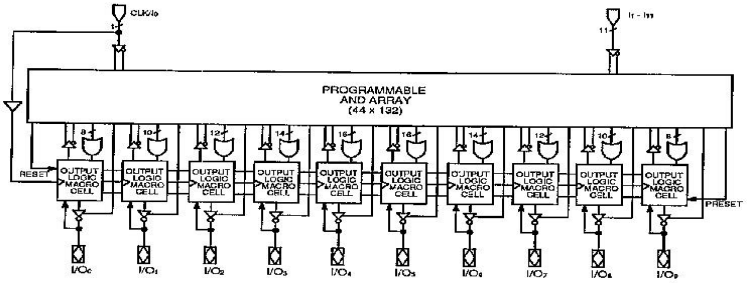
Remarques : La programmation des cellules de sortie est transparente pour l'opérateur. C'est le logiciel de développement qui, en fonction de certaines indications (sortie / entrée registre ou combinatoire), effectue la configuration des structures de sortie.

Pour le GAL 16V8, Les broches 15 et 16 ne peuvent pas être configurées en entrées combinatoires

5-5 EXEMPLE DE BROCHAGE DE CIRCUIT.



brochage du 16V8 -



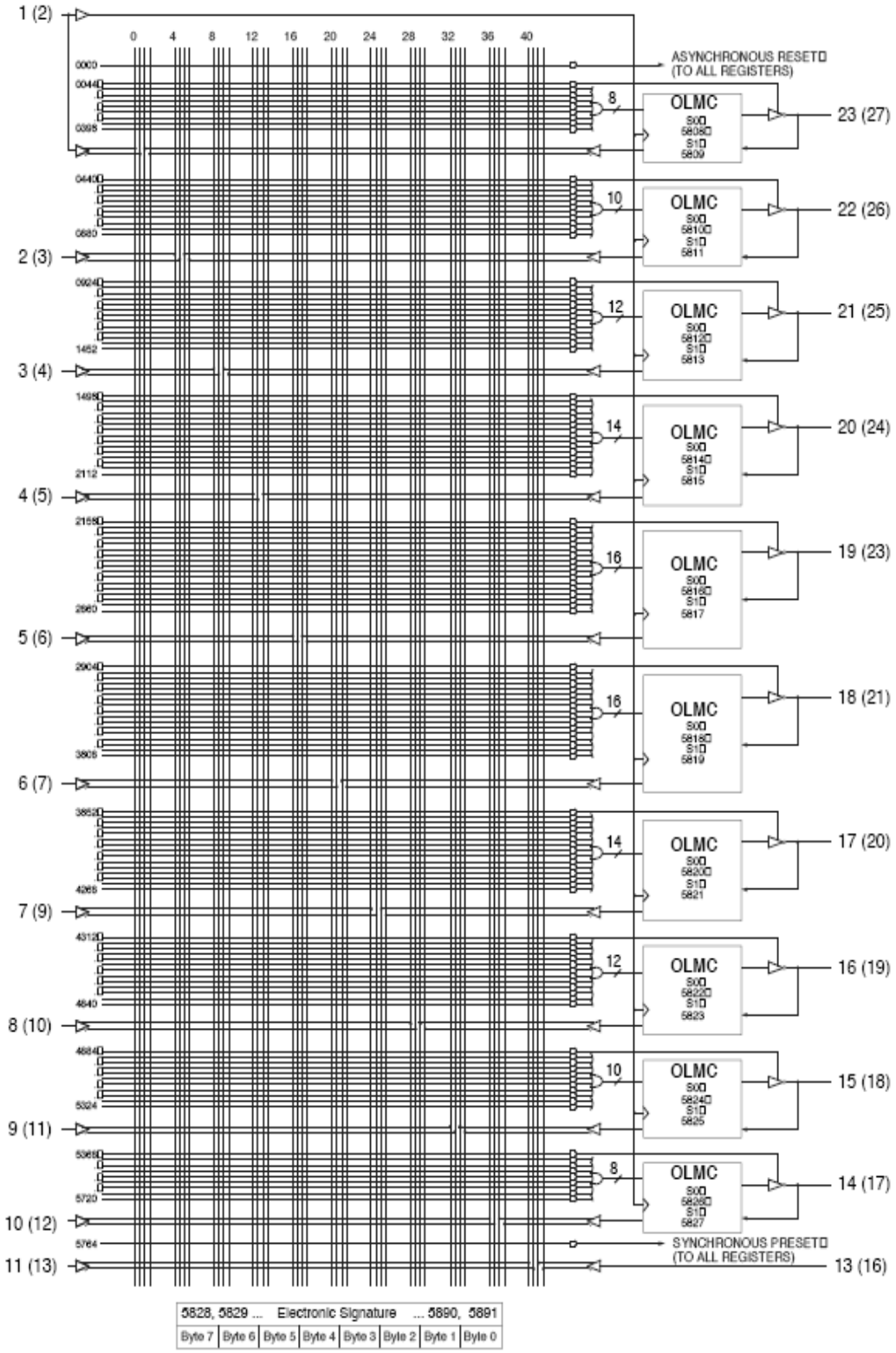
brochage du 22V10 –

Lattice Semiconductor Corporation

Specifications **GAL22V10**

GAL22V10 Logic Diagram / JEDEC Fuse Map

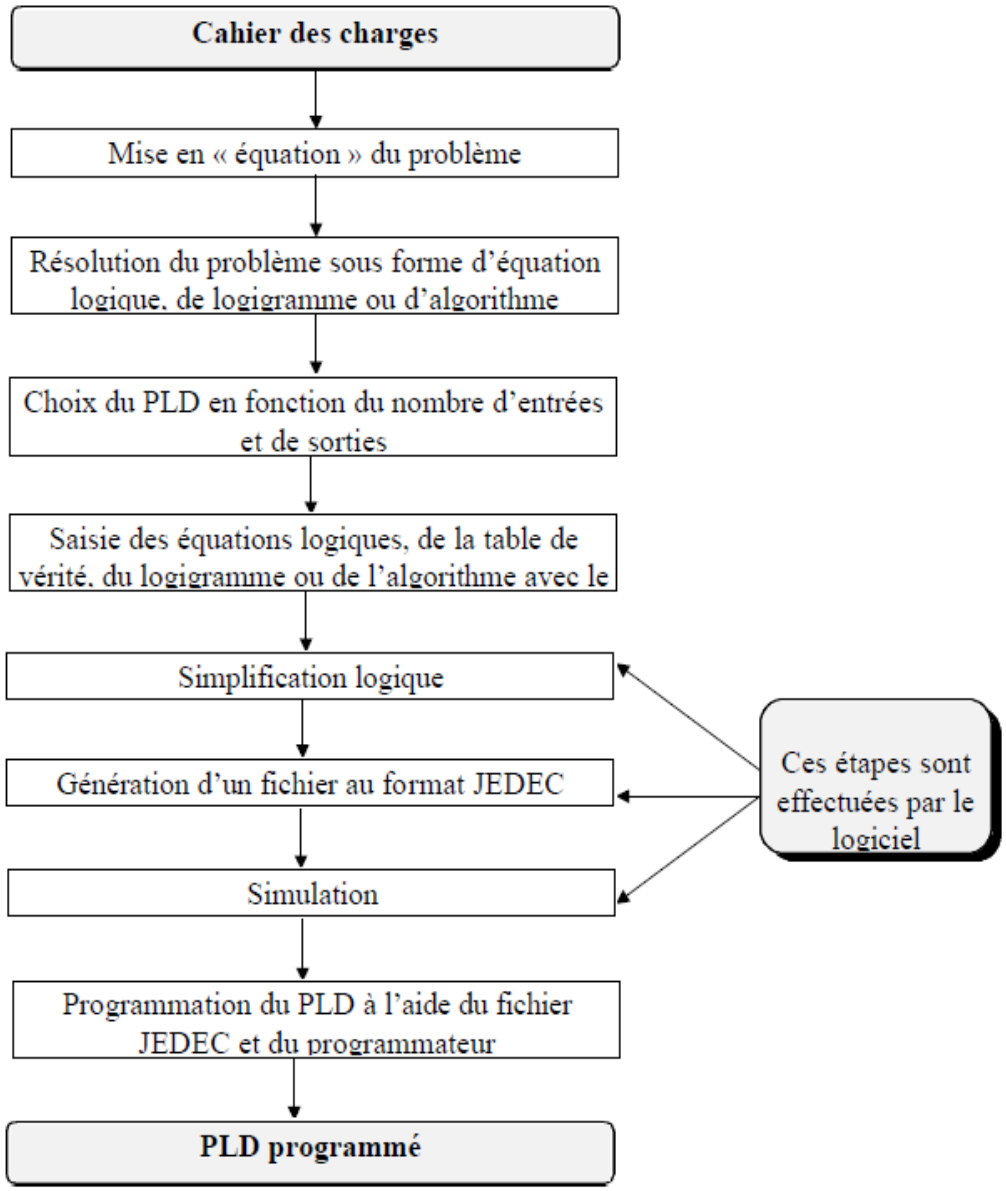
DIP (PLCC) Package Pinouts



5828, 5829 ... Electronic Signature ... 5890, 5891							
Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0

6/ PROGRAMMATION DES PLDS.

La programmation des PLDs nécessite un logiciel adapté pour le développement du programme et un programmateur permettant de « griller » le circuit. En outre il est conseillé de suivre la démarche décrite par l'organigramme suivant :



* Le logiciel de développement permet de simplifier les équations et de générer un fichier JEDEC à partir des données rentrées par l'opérateur. Il simule aussi le fonctionnement du PLD avec le programme obtenu.

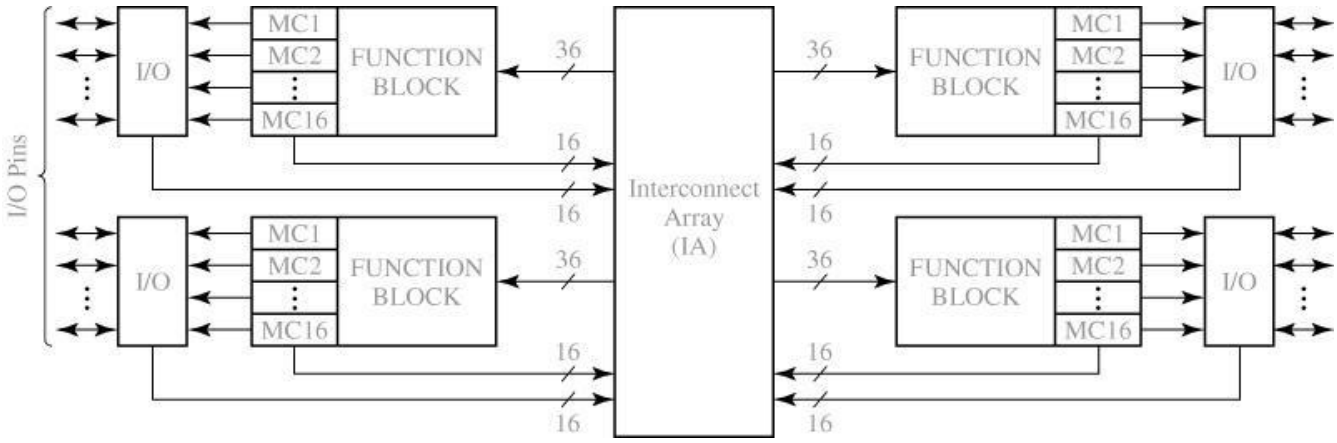
* Le fichier JEDEC est un ensemble de données binaires indiquant au programmateur les fusibles à « griller ».

* Le programmateur permet de « griller » les fusibles du PLD en fonctions des données du fichier JEDEC. Il est en général associé à un logiciel de pilotage. Les programmateurs utilisés sont les mêmes que ceux permettant la programmation des EPROM.

7/ Les CPLD (Complex Programmable Logic Devices).

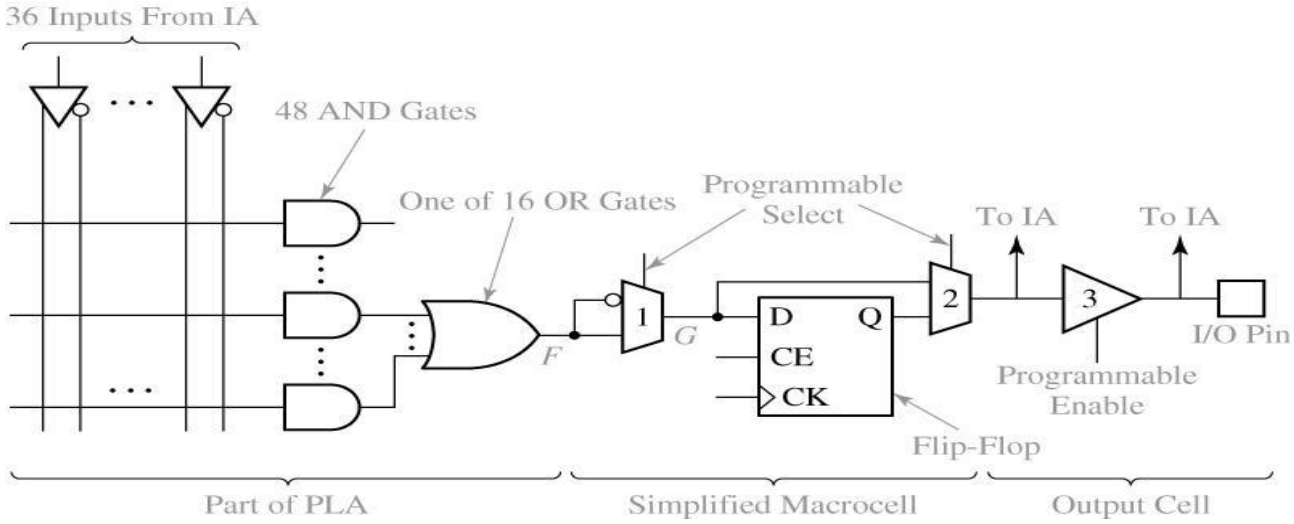
Les ROMs, PLAs, PALs et GALs sont parfois appelés des circuits logique programmable simples (*Simple Programmable Logic Devices* – SPLD).

Les CPLD sont une extension naturelle des circuits PAL. À mesure que leur complexité grandissait, la taille des PALs était limitée par le nombre de pattes pouvant être placées sur la puce. Un CPLD incorpore donc plusieurs PALs ou PLAs sur une seule puce avec un réseau d'interconnexions. Le réseau permet de relier les pattes de la puce à différents blocs internes, mais aussi à relier les blocs entre eux. Il est donc possible de composer des fonctions logiques très complexes incluant des machines à états et de petites mémoires.



architecture d'un CPLD de Xilinx

Le système comprend quatre blocs fonctionnels qui sont des PALs à 36 entrées et 16 sorties. Les sorties proviennent de macro-cellules contenant un élément programmable à mémoire. Le réseau d'interconnexions permet d'établir des connexions entre les blocs d'entrées-sorties, les blocs fonctionnels et les macro-cellules.



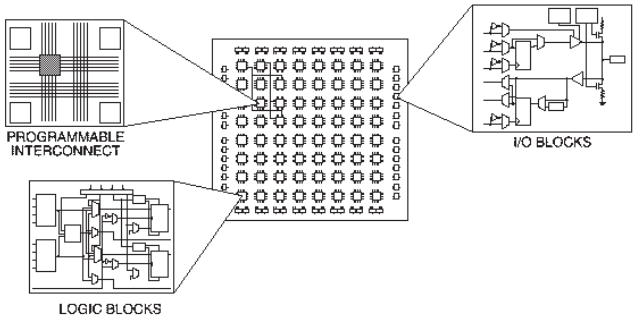
Détails d'un CPLD

On voit les 36 entrées en versions naturelle et complémentée qui alimentent 48 portes ET à connexions programmables. Les 48 portes ET alimentent 16 portes OU à connexions programmables. La macro-

cellule permet d'activer ou non la bascule avec un signal naturel ou complémenté. La cellule de sortie contient un tampon à trois états.

8/ 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.

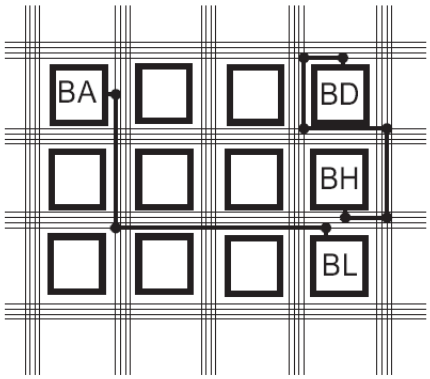


Structure d'une FPGA

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 terme de vitesse. Sur la figure suivante, pour illustrer 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.



L'utilisation optimale des potentialités d'intégration d'un FPGA conduira à un routage délicat et pénalisant en terme de vitesse. Il convient de noter que ces retards sont dus à l'interaction de la

résistance de la connexion et de la capacité parasite; cela n'a rien à voir avec un retard dû à la propagation d'un signal sur une ligne tel qu'on le voit en haute fréquence.

Ces composants permettent une forte densité d'intégration. La petitesse des blocs logiques autorise une meilleure utilisation des ressources du composant (au prix d'un routage délicat) Il devient alors possible d'implanter dans le circuit des fonctions aussi complexes qu'un micro-contrôleur. Ces fonctions sont fournies sous forme de programme (description VHDL du composant) par le constructeur du composant et appelées "megafunction" ou "core". Le terme générique classiquement utilisé pour les désigner est "propriété intellectuelle" ou IP (Intelectual Property).

Les FPGAs utilisent généralement les technologies SRAM ou antifusible, selon les fabricants, Xilinx utilisant la technologie SRAM et ACTEL tout comme ALTERA la technologie antifusible de préférence.

Chapitre III

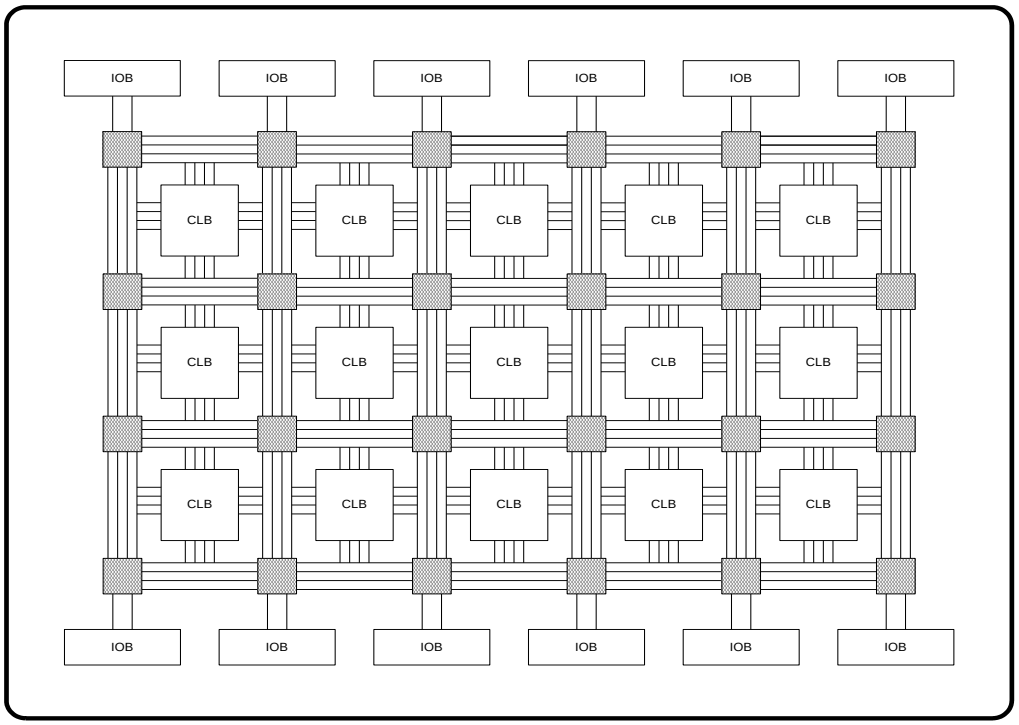
Réseaux pré-diffusés programmables (FPGA)

I/ Architecture générale

Les FPGA se sont imposés sur le marché des circuits logiques depuis la fin des années 1980. Alors qu'au début ils permettaient de remplacer quelques composantes discrètes de façon plus efficace que les PALs et PLAs, ils concurrencent de nos jours certains microprocesseurs et microcontrôleurs en complexité et en performance.

Un FPGA est composé à la base de :

- un réseau de blocs de logique programmable (Configurable Logic Block – CLB), chaque bloc pouvant réaliser des fonctions complexes de plusieurs variables, et comportant des éléments à mémoire;
- un réseau d'interconnexions programmables entre les blocs;
- des blocs spéciaux d'entrée et de sortie avec le monde extérieur (Input/Output Block – IOB).



modèle d'un FPGA

II/ Blocs de logique programmable

Les FPGAs se distinguent des CPLDs par le fait que les CLBs des FPGAs sont beaucoup plus nombreux et plus simples que les blocs fonctionnels des CPLDs. Comme ils sont présents en grand nombre sur une même puce (de plusieurs centaines à plusieurs milliers), ils favorisent la réalisation et l'intégration d'une multitude de circuits indépendants.

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. Pour ce faire, on exploite le fait qu'un multiplexeur génère effectivement tous les mintermes des signaux de contrôle S. Il ne reste qu'à spécifier la valeur qu'on veut propager quand un des ces mintermes est vrai. La procédure de conception consiste à écrire la table de vérité de la fonction en groupant les lignes par paires. À chaque paire de lignes correspond une valeur des lignes de sélection du multiplexeur. On assigne aux lignes de données l'une de quatre valeurs : 0, 1, α ou α', où α est la variable d'entrée la moins significative. Par exemple, considérons la fonction logique $F(x,y,z)=\sum m(0,5,6,7)$. On exprime la fonction sous la forme d'une table de vérité. On rajoute une colonne pour indiquer à quelle entrée Di chaque minterme correspond. Finalement, on indique la valeur à donner à chaque entrée Di en fonction de la sortie F désirée et la valeur de la variable z. Cet exemple est montré à la Figure 0-1.

#	*x*	*y*	*z*	*F*	**entrée D_i**	**valeur**
0	0	0	0	1	D_0	z'
1	0	0	1	0		
2	0	1	0	0	D_1	0
3	0	1	1	0		
4	1	0	0	0	D_2	z
5	1	0	1	1		
6	1	1	0	1	D_3	1
7	1	1	1	1		

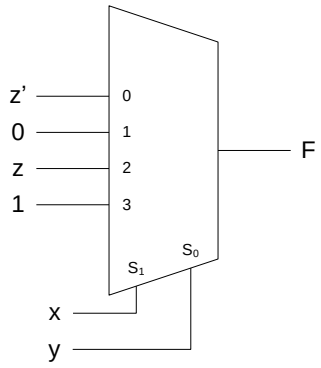


Figure 0-1 – implémentation d'une fonction logique par multiplexeur

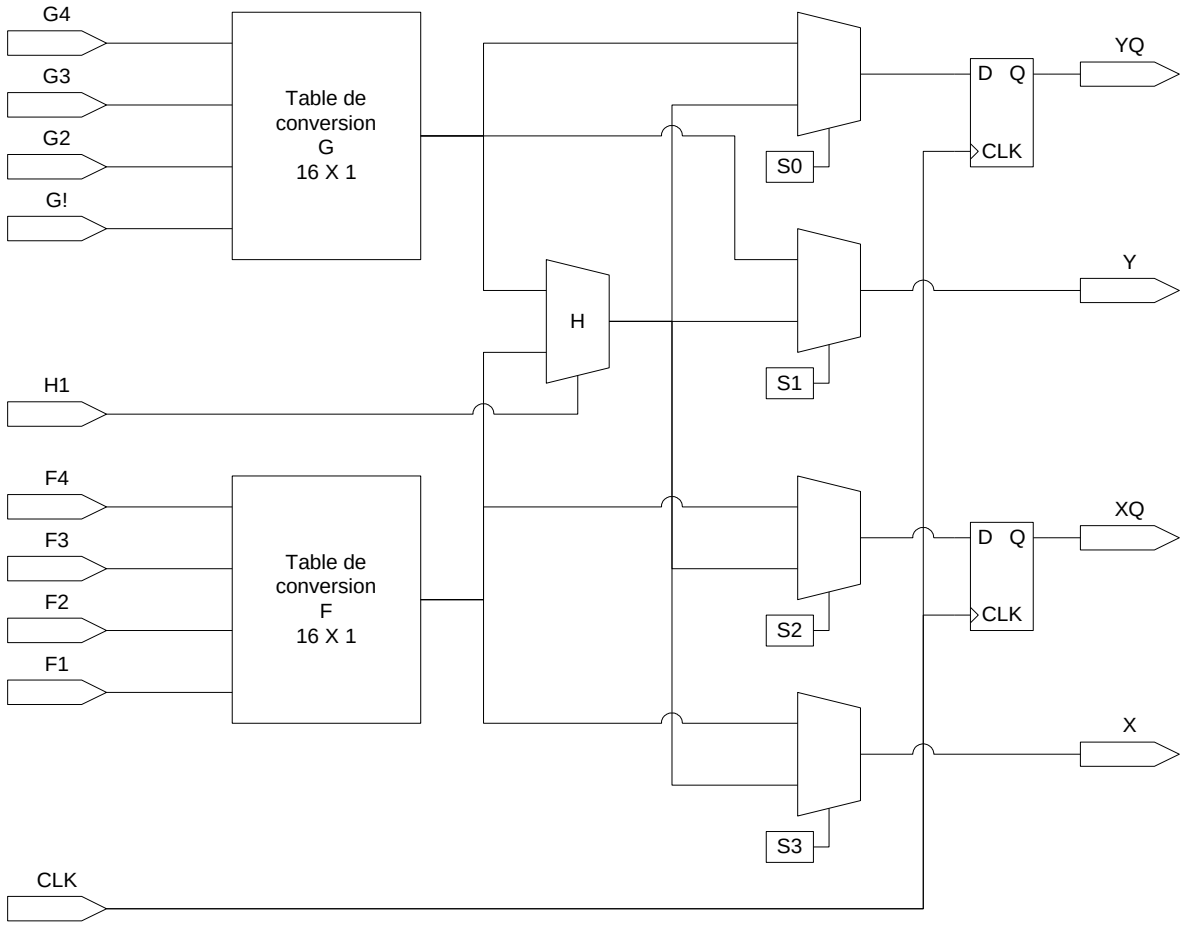
- Les CLBs basés sur les tables de conversion utilisent de petites mémoires programmables au lieu de multiplexeurs. Cette approche est similaire à l'approche par multiplexeurs, mais en supposant que les entrées du multiplexeur ne peuvent être que des constantes. Effectivement, il faut un multiplexeur deux fois plus gros pour réaliser la même fonction, mais le routage du circuit est plus simple. De plus, le circuit peut être plus rapide parce que les entrées du multiplexeur sont constantes.

Les FPGAs de la compagnie Altera étaient à l'origine basés sur une approche par multiplexeurs, alors que ceux de Xilinx utilisaient des tables de conversion. La plupart des FPGAs récents utilisent

des tables de conversion. Cela ne signifie pas que des multiplexeurs ne sont plus utilisés. Au contraire, ils sont essentiels pour router adéquatement les signaux à l'intérieur d'un CLB en choisissant différentes possibilités de configuration.

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.



bloc de logique programmable simplifié – Xilinx

Pour plusieurs familles de FPGA, les tables de conversion peuvent aussi être utilisées comme mémoire distribuée par l'ajout de signaux de contrôle. Des configurations très polyvalentes permettent de varier le nombre de mots ou leur largeur, et de supporter une mémoire à un ou deux ports de sortie. De façon similaire, les tables de conversion peuvent être utilisées comme registres à décalage.

III/ Terminologie : LE, LAB, ALM, slice, CLB

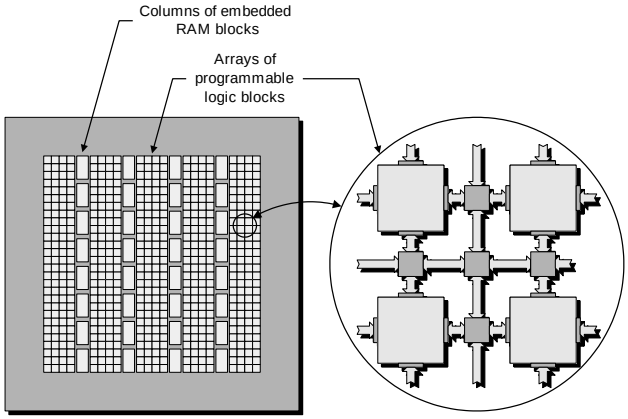
Pour des raisons internes aux différents manufacturiers, plusieurs termes sont utilisés pour parler de l'architecture interne des FPGAs.

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. Un Logic Array Bloc – LAB regroupe dix LEs. Pour la famille Stratix, Altera a remplacé les LEs par des blocs plus complexes, les 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. Un Configurable Logic Block – CLB regroupe deux ou quatre slices, selon la famille de FPGA.

IV/ Blocs de mémoire intégrée

Pendant les années 1990, alors que de plus en plus de transistors étaient disponibles pour intégration sur des puces, les fabricants de FPGA ont commencé à intégrer des modules de plus en plus complexes au tissu de base montré à la figure ci-dessous. 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.

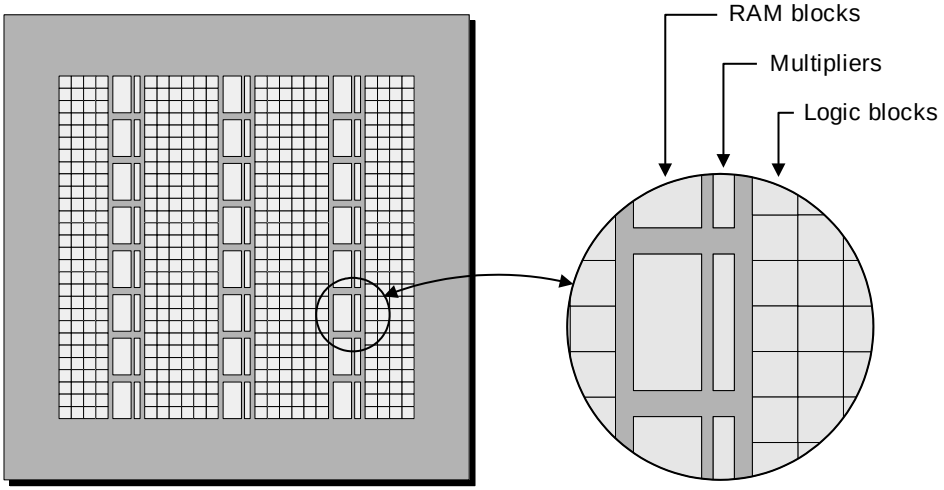


mémoire RAM intégrée

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.

V/ Fonctions arithmétiques avancées

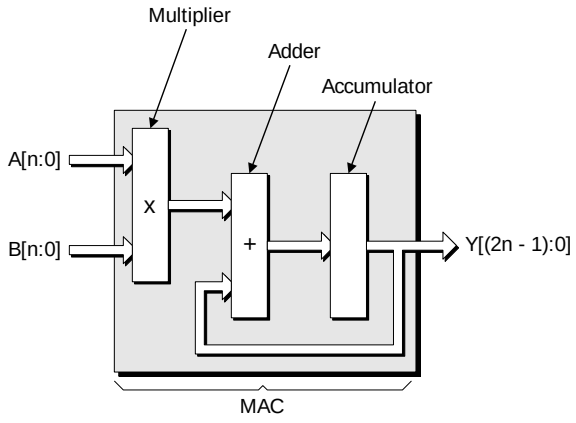
La multiplication est une opération fondamentale dans les applications de traitement du signal pour lesquelles les FPGAs sont très populaires. Les fabricants de FPGAs ont donc ajouté à leurs architectures des blocs spéciaux pour réaliser cette opération. Les multiplicateurs sont en général disposés près des blocs de mémoire RAM pour des raisons d'efficacité de routage des signaux et de disposition des ressources du FPGA sur une puce. La figure ci dessous illustre un tel arrangement.



multiplicateurs intégrés

Les paramètres d'opération des multiplicateurs varient selon les familles de FPGA. Un format populaire est un multiplicateur signé de 18 × 18 bits, ce qui est suffisant pour beaucoup d'applications de traitement du signal (le son est encodé avec 8 bits sur une ligne de téléphone et avec 16 bits par canal pour un CD audio). Pour les applications devant traiter des opérandes plus larges, il est possible de regrouper plusieurs multiplicateurs. Le nombre de multiplicateurs varie, mais on en retrouve facilement des dizaines même sur les FPGAs les plus modestes.

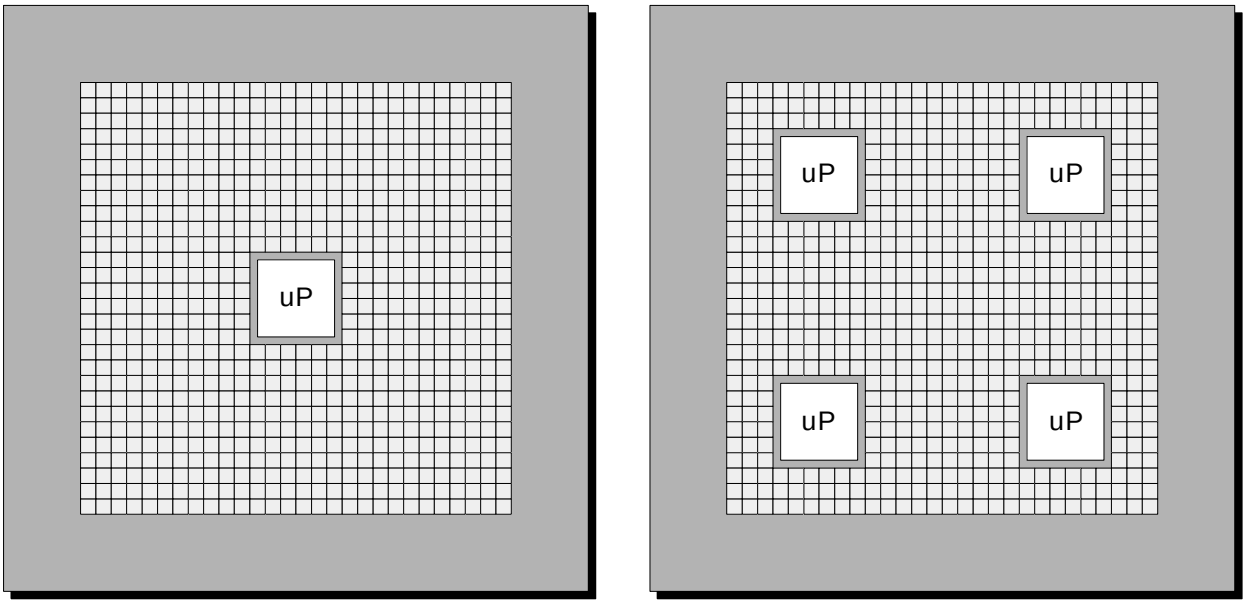
Une autre fonction populaire est la multiplication-accumulation (*Multiply-Accumulate* – MAC), qui consiste à accumuler le produit de plusieurs nombres. Cette opération est utilisée entre autres pour le calcul du produit scalaire. Afin d'accélérer cette opération, plusieurs fabricants intègrent des blocs MAC à leurs FPGAs.



multiplication-accumulation

VI/ Microprocesseurs fixes

Avec le nombre de transistors intégrables sur une puce doublant tous les 18 mois, et les besoins important en contrôle dans plusieurs applications, les fabricants de FPGA intègrent des processeurs fixes (*hard*) à plusieurs familles de FPGAs. La figure ci-dessous illustre deux variations, avec un et quatre processeurs fixes.



(a) One embedded core (b) Four embedded cores

Processeur intégré

Les microprocesseurs intégrés ne sont pas réalisés à l'aide de blocs de logique ou de mémoire du FPGA. Ce sont plutôt des régions de la puce optimisées comme si le microprocesseur était réalisé par lui-même dans un circuit intégré. En général, les constructeurs des FPGAs achètent des processeurs populaires dans l'industrie et déjà bien connus des consommateurs, comme par exemple l'architecture PowerPC de IBM.

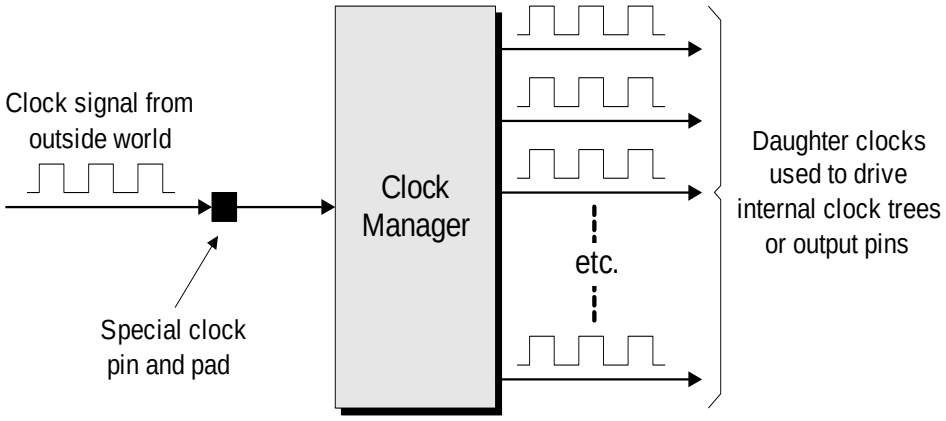
En intégrant ainsi un ou plusieurs microprocesseurs à un FPGA, on obtient un tissu de calcul d'une puissance très intéressante. En effet, une application peut être divisée efficacement entre des parties matérielles et logicielles; les parties matérielles sont réalisées avec les ressources configurables du FPGA, et les parties logicielles sont réalisées avec les microprocesseurs. Là où le parallélisme des calculs l'exige, les blocs de logique configurable peuvent accélérer les calculs par un facteur de 10×, 100× ou plus. Là où l'application risque d'être modifiée, ou si elle nécessite beaucoup de contrôle ou d'interface avec le monde extérieur ou entre des modules, une solution logicielle peut être obtenue plus facilement.

Avec les microprocesseurs entourés des ressources configurables du FPGA, on peut atteindre des taux d'échange de données très élevés. Tout d'abord, la proximité physique des dispositifs réduit les délais. Ensuite, il est possible de mettre en place des liens de communication très larges.

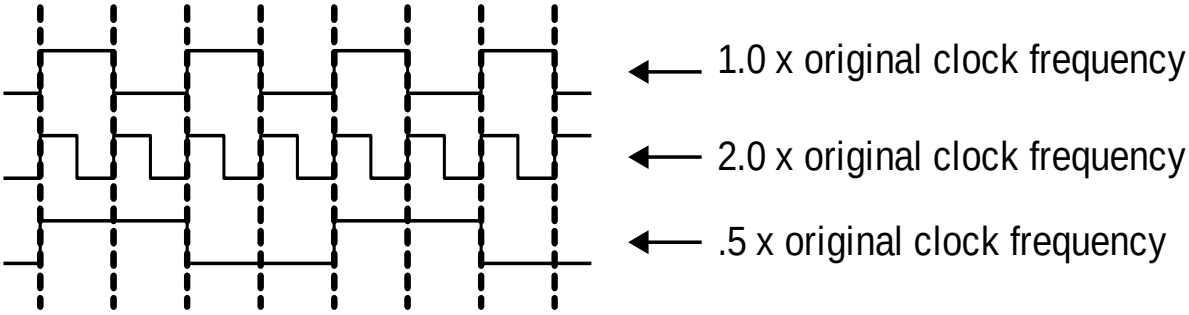
VII/ Génération et distribution d'horloge

Un problème important lors de la conception et réalisation d'un circuit numérique concerne la génération et la distribution du signal d'horloge. Dans une application de communications et de traitement du signal, un même circuit peut nécessiter une dizaine d'horloges de fréquences et de phases différentes.

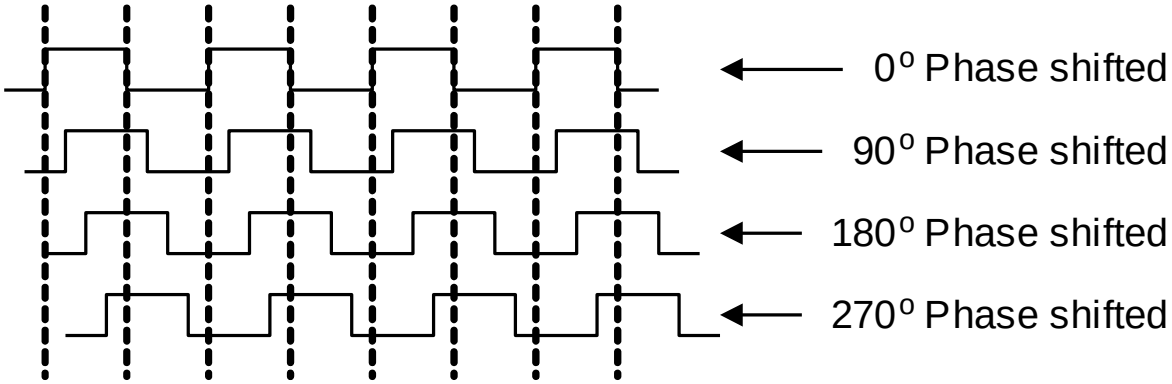
Pour faciliter la tâche des concepteurs, les FPGAs incluent maintenant des circuits spécialisés de génération, régulation et distribution de l'horloge. Ce type de circuit accepte en entrée une horloge externe et génère une ou plusieurs horloges internes. Les horloges internes peuvent avoir des fréquences et des phases différentes..



génération de signaux d'horloge à partir d'une référence externe

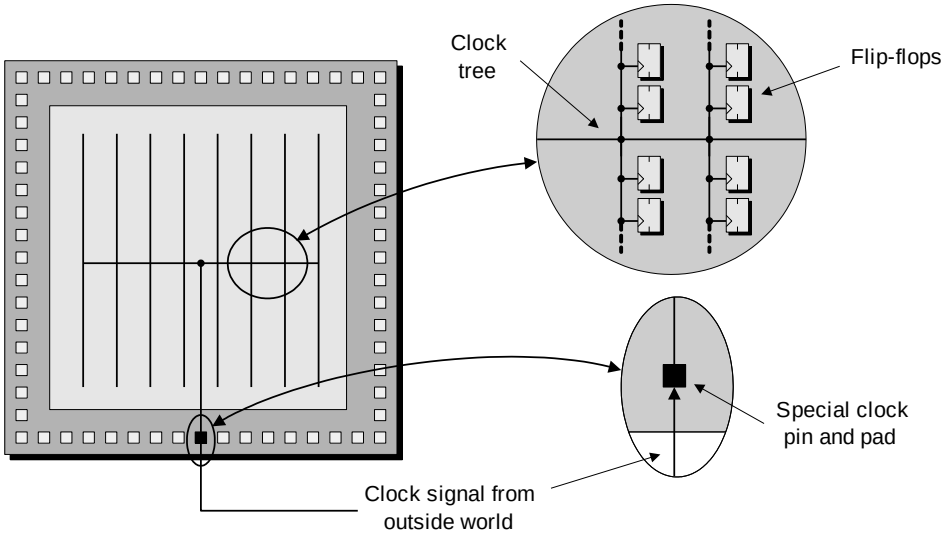


varier la fréquence d'horloge



varier la phase de l'horloge

Pour distribuer l'horloge à travers la puce tout en minimisant le déphasage d'horloge, on utilise un réseau en arbre dédié. Ce réseau est alimenté soit par une patte spéciale du FPGA à laquelle est associé un amplificateur dédié, ou bien par l'entremise de signaux internes pouvant être routés au même amplificateur. La **Erreur ! Source du renvoi introuvable.** illustre un tel arrangement.



arbre de distribution d'horloge

VIII/ Blocs d'entrées-sorties

Les éléments logiques à l'intérieur d'un FPGA sont forcément très importants; les interfaces avec le monde extérieur le sont tout autant. Un problème observé assez tôt lors du développement de FPGAs de plus en plus performants est que leur capacité de traitement excédait leur capacité à recevoir des données et transmettre des résultats. Les FPGAs incorporent donc des blocs d'entrées-sorties très performants.

Les blocs d'entrées-sorties doivent pouvoir supporter plusieurs normes en termes de débit d'information, de tensions et d'impédance. Ils incorporent en général une bascule pour minimiser les délais de propagation et augmenter le taux de transmission de l'information.

IX/ Comparaison d'équivalences en termes de portes logiques

La comparaison de deux FPGAs de familles différentes est difficile à faire, et spécialement si ce sont des FPGAs de deux constructeurs différents. Au début de l'ère FPGA, les constructeurs ont commencé à publier des métriques comme des « équivalent-portes » pour indiquer combien un FPGA donné pouvait contenir de portes logiques de base (fonction NON-OU ou NON-ET) à deux entrées. Cependant, avec la multitude de blocs spécialisés sur un FPGA aujourd'hui, il vaut mieux comparer les métriques de ressources en termes des ressources elles-mêmes, comme le nombre de tables de conversions, de bascules, de bits de mémoire ou de microprocesseurs embarqués.

Chapitre IIX

Le langage VHDL

I) Introduction.

L'abréviation **VHDL** signifie **VHSIC** **H**ardware **D**escription **L**anguage (**VHSIC** : **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit). Ce langage a été écrit dans les années 70 pour réaliser la simulation de circuits électroniques. On l'a ensuite étendu en lui rajoutant des extensions pour permettre la conception (synthèse) de circuits logiques programmables (**P.L.D. P**ro- grammable **L**ogic **D**evice).

Auparavant pour décrire le fonctionnement d'un circuit électronique programmable les techniciens et les ingénieurs utilisaient des langages de bas niveau (**ABEL, PALASM, ORCAD/PLD**,..) ou plus simplement un outil de saisie de schémas.

Actuellement la densité de fonctions logiques (portes et bascules) intégrée dans les **PLDs** est telle (plusieurs milliers de portes voire millions de portes) qu'il n'est plus possible d'utiliser les outils d'hier pour développer les circuits d'aujourd'hui.

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

Ils permettent au code écrit d'être portable, c'est à dire qu'une description écrite pour un circuit peut être facilement utilisée pour un autre circuit.

Il faut avoir à l'esprit que ces langages dits de haut niveau permettent de matérialiser les structures électroniques d'un circuit.

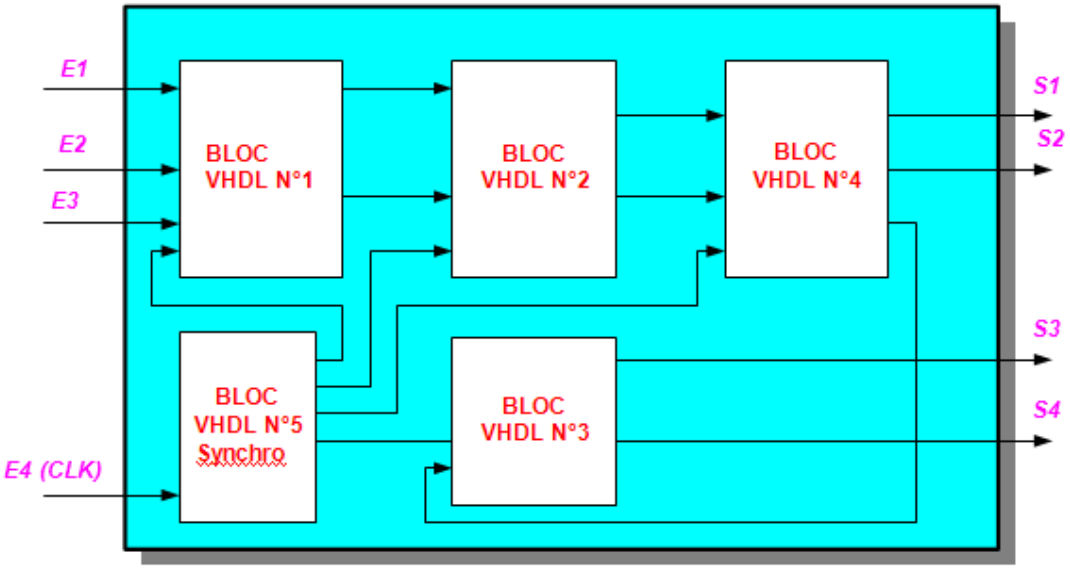
En effet les instructions écrites dans ces langages se traduisent par une configura- tion logique de portes et de bascules qui est intégrée à l'intérieur des circuits **PLDs**. C'est pour cela que je préfère parler de description **VHDL** ou **VERILOG** que de langage.

II) Relation entre une description VHDL et les circuits logiques programmables.

Décrire le fonctionnement d'un circuit logique programmable c'est bien, mais coment faire le lien avec la structure de celui-ci ?

L'implantation d'une ou de plusieurs descriptions VHDL dans un PLD va dépendre de l'affectation que l'on fera des broches d'entrées / sorties et des structures de base du circuit logique programmable.

II.1) Schéma fonctionnel d'implantation de descriptions VHDL dans un circuit logique programmable.



Ci-dessus le schéma représente un exemple d'implantation de descriptions VHDL ou de blocs fonctionnels implantés dans un PLD. Lors de la phase de synthèse chaque bloc sera matérialisé par des portes et/ou des bascules. La phase suivante sera d'implanter les portes et les bascules à l'intérieur du circuit logique.

Cette tâche sera réalisée par le logiciel placement/routage (« Fitter »), au cours de laquelle les entrées et sorties seront affectées à des numéros de broches.

II.2) L'affectation des broches d'entrées sorties.

Elle peut se faire de plusieurs manières :

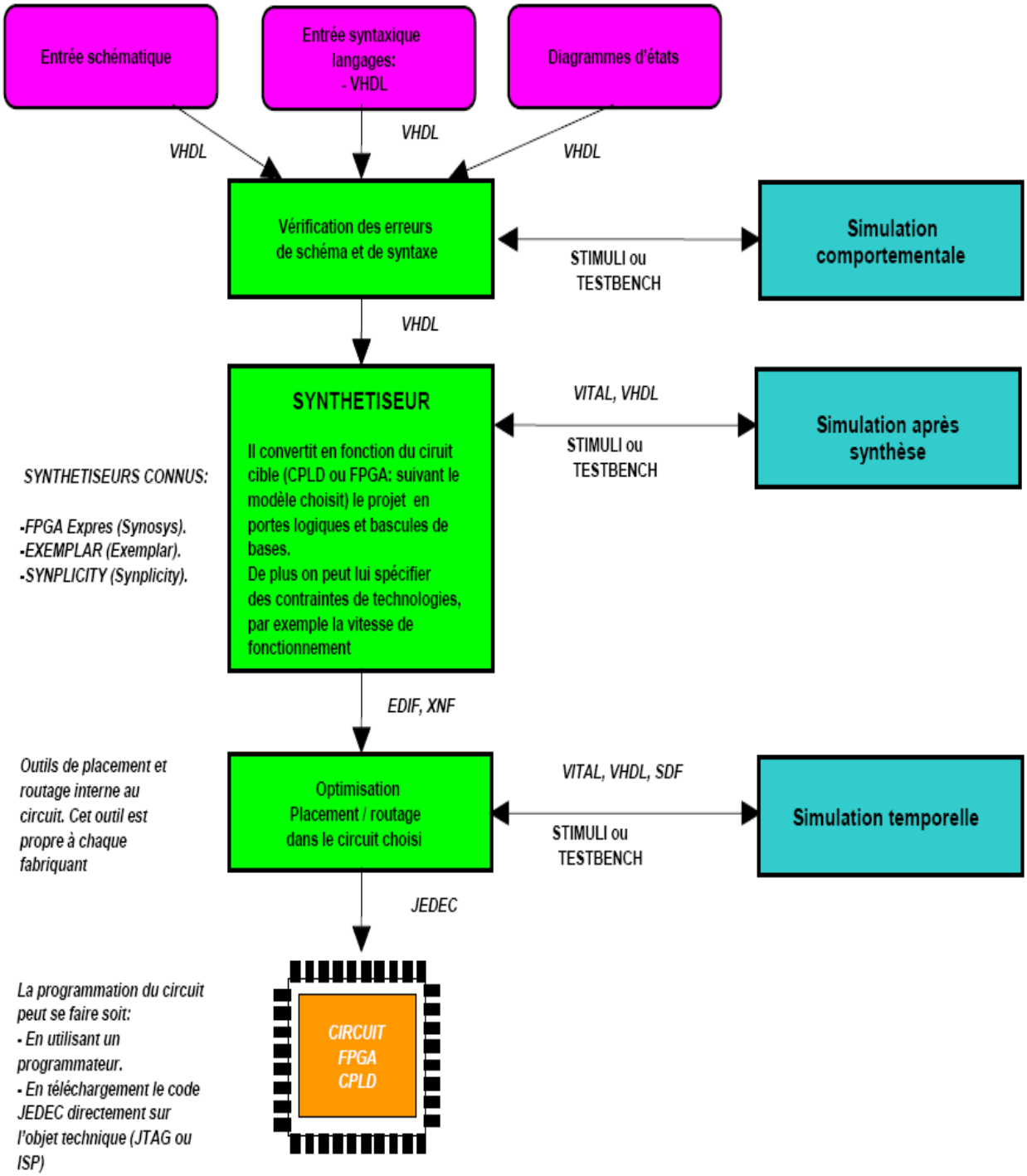
1) L'affectation automatique.

On laisse le synthétiseur et le Fitter du fabriquant (« Fondeur » : Xilinx, Lattice, Altéra, Cypress...») du circuit implanter la structure correspondant à la description VHDL. Les numéros de broches seront choisis de façon automatique.

2) L'affectation manuelle.

On définit les numéros de broches dans la description VHDL ou sur un schéma bloc définissant les liaisons entre les différents blocs VHDL ou dans un fichier texte propre au fondeur. Les numéros de broches seront affectés suivant les consignes données.

II.3) Organisation fonctionnelle de développement d'un PLD.



III) Structure d'une description VHDL simple.

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

- L'entité (ENTITY), elle définit les entrées et sorties.
- L'architecture (ARCHITECTURE), elle contient les instructions VHDL permettant de réaliser le fonctionnement attendu.

Exemple : Un décodeur 1 parmi 4.

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

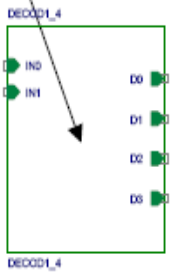
Déclaration des bibliothèques

Commentaires, en VHDL ils commencent par - -

```
-- décodeur
-- Un parmi quatre
```

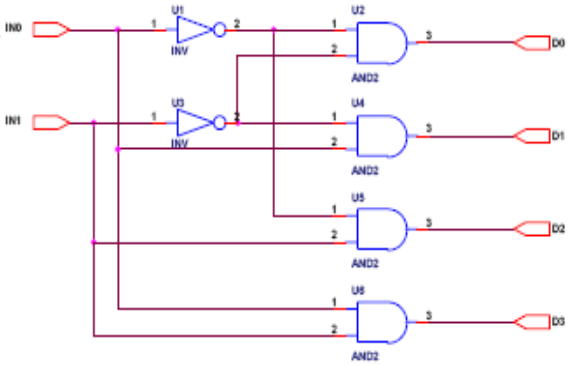
Déclaration de l'entité du décodeur
Correspondance schématique

```
entity DECOD1_4 is
      port(IN0, IN1: in std_logic;
           D0, D1, D2, D3: out std_logic);
end DECOD1_4;
```



Déclaration de l'architecture du décodeur
Correspondance schématique

```
architecture DESCRIPTION of DECOD1_4 is
begin
      D0 <= (not(IN1) and not(IN0));
      D1 <= (not(IN1) and IN0);
      D2 <= (IN1 and not(IN0));
      D3 <= (IN1 and IN0);
end DESCRIPTION;
```



III.1) Déclaration des bibliothèques.

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

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
-- cette dernière bibliothèque est souvent utilisée pour l'écriture de compteurs
```

La directive Use permet de sélectionner les bibliothèques à utiliser.

III.2) Déclaration de l'entité et des entrées / sorties (I/O).

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

Syntaxe:

```
entity NOM_DE_L_ENTITE is

port ( Description des signaux d'entrées /sorties …);

end NOM_DE_L_ENTITE;
```

Exemple :

```
entity SEQUENCEMENT is port (
CLOCK    : in std_logic;
RESET    : in std_logic;
Q        : out std_logic_vector(1 downto 0)
);
end SEQUENCEMENT;
```

Remarque : Après la dernière définition de signal de l'instruction port ne jamais mettre de point virgule.

L'instruction port .
Syntaxe: NOM_DU_SIGNAL : sens type; Exemple: CLOCK: in std_logic;
BUS : out std_logic_vector (7 downto 0);

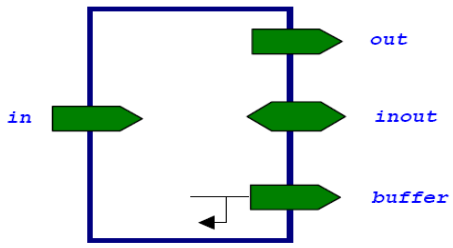
On doit définir pour chaque signal : le NOM_DU_SIGNAL, le sens et le type.

III.2.1) Le NOM_DU_SIGNAL.

Il est composé de caractères, le premier caractère doit être une lettre, sa longueur est quelconque, mais elle ne doit pas dépasser une ligne de code. VHDL n'est pas sensible à la « casse », c'est à dire qu'il ne fait pas la distinction entre les majuscules et les minuscules.

III.2.2) Le SENS du signal.

- in : pour un signal en entrée.
- out : pour un signal en sortie.
- inout : pour un signal en entrée sortie
- buffer : pour un signal en sortie mais utilisé comme entrée dans la description.



III.2.3) Le TYPE.

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

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

Par exemple un bus bidirectionnel de 5 bits s'écrira :

LATCH : inout std_logic_vector (4 downto 0) ;

Où LATCH(4) correspond au MSB et LATCH(0) correspond au LSB.

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

- '0' ou 'L' : pour un niveau bas.
- '1' ou 'H' : pour un niveau haut.
- 'Z' : pour état haute impédance.
- '-' : Quelconque, c'est à dire n'importe quelle valeur.

III.2.4) Affectation des numéros de broches en utilisant des attributs supplémentaires:

La définition des numéros de broches par la définition d'attributs supplémentaires dépend du logiciel utilisé pour le développement.

Exemples :

Avec Warp de Cypress ou ISE de Xilinx:

```
entity AFFICHAGE is port(
CLK, RESET, DATA: in std_logic;
S:buffer std_logic_vector(9 downto 0));
-- C'est la ligne ci-dessous qui définit les numéros de broches
attribute pin_numbers of AFFICHAGE:entity is "S(9):23 S(8):22 S(7):21
S(6):20 S(5):19 S(4):18 S(3):17 S(2):16 S(1):15 S(0):14";
end AFFICHAGE;
```

III.2.5) Exemples de description d'entités:

```
entity COUNT is
port(CLK, RST: in std_logic;
     CNT: inout std_logic_vector(2 downto 0));
end COUNT;


entity MAGNITUDE is
port( A, B: in std_logic_vector(7 downto 0); AGRB: buffer std_logic));
end MAGNITUDE;


entity 7SEG is port (
A    : in std_logic; B: in std_logic; C: in std_logic; D: in std_logic;
SA   : out std_logic;
SB   : out std_logic;
SC   : out std_logic; SD: out std_logic; SE : out std_logic; SF: out std_logic;
SG   : out std_logic );
end 7SEG;


entity IMAGE3 is
port (CLK : in std_logic;
S,I : out std_logic);
end IMAGE3;


entity COMP4BIT is
port (A,B :in std_logic_vector(3 downto 0); PLUS,MOINS,EGAL :out std_logic);
end COMP4BIT;


entity COMPT_4 is
```

```
port (H,R :in std_logic;
Q :out std_logic_vector(3 downto 0));
end COMPT_4;


entity DECAL_D is
port (H,R,IN_SERIE :in std_logic;
OUT_SERIE :out std_logic);
end DECAL_D;
```

III.3) Déclaration de l'architecture correspondante à l'entité : description du fonctionnement.

L'architecture décrit le fonctionnement souhaité pour un circuit ou une partie du circuit

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

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

Exemples :

```
-- Opérateurs logiques de base entity PORTES is
port (A,B :in std_logic;
Y1,Y2,Y3,Y4,Y5,Y6,Y7:out std_logic);
end PORTES;

architecture DESCRIPTION of PORTES is begin
Y1 <= A and B; Y2 <= A or B; Y3 <= A xor B; Y4 <= not A;
Y5 <= A nand B;
Y6 <= A nor B;
Y7 <= not(A xor B);
end DESCRIPTION;

-- Décodeurs 7 segments entity DEC7SEG4 is
port (DEC :in std_logic_vector(3 downto 0); SEG:out std_logic_vector(0 downto 6));
end DEC7SEG4;

architecture DESCRIPTION of DEC7SEG4 is begin
SEG <= "1111110" when DEC = 0
else "0110000" when DEC = 1 else "1101101" when DEC = 2 else "1111001" when DEC =
3 else "0110011" when DEC = 4 else "1011011" when DEC = 5 else "1011111" when DEC
= 6 else "1110000" when DEC = 7 else "1111111" when DEC = 8 else "1111011" when
DEC = 9 else "-------";
end DESCRIPTION;

-- Décodage d'adresses
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;

entity DECODAGE is port (
A15, A14, A13, A12, A11, A10 : in std_logic;
 RAM0: out std_logic;
```

```
 RAM1: out std_logic;
 RAM2: out std_logic;
 RAM3: out std_logic;
ROM: out std_logic;
INTER1: out std_logic;
INTER2: out std_logic;
INTER3: out std_logic);
end DECODAGE;

architecture DESCRIPTION of DECODAGE is

signal ADRESSE: std_logic_vector(15 downto 0);

begin

ADRESSE <= A15 & A14 & A13 & A12 & A11 & A10 & "----------";
-- definition du bus d'adresses

ROM   <= '0' when (ADRESSE >= x"E000") and (ADRESSE <= x"FFFF") else '1'; RAM0  <=
'0' when (ADRESSE >= x"0000") and (ADRESSE <= x"03FF") else '1'; RAM1  <= '0' when
(ADRESSE >= x"0400") and (ADRESSE <= x"07FF") else '1'; RAM2  <= '0' when (ADRESSE
>= x"0800") and (ADRESSE <= x"0BFF") else '1'; RAM3  <= '0' when (ADRESSE >=
x"0C00") and (ADRESSE <= x"0FFF") else '1';

INTER1 <= '0' when (ADRESSE >= x"8000") and (ADRESSE <= x"8001") else '1'; INTER2
<= '0' when (ADRESSE >= x"A000") and (ADRESSE <= x"A001") else '1'; INTER3 <= '0'
when (ADRESSE >= x"C000") and (ADRESSE <= x"C00F") else '1';

end DESCRIPTION;
```

IV) Les instructions de base (mode « concurrent »), logique combina- toire.

Pour une description VHDL toutes les instructions sont évaluées et affectent les signaux de sortie en même temps. L'ordre dans lequel elles sont écrites n'a aucune importance. En effet la description génère des structures électroniques, c'est la grande différence entre une description VHDL et un langage informatique classique.

Dans un système à microprocesseur, les instructions sont exécutées les unes à la suite des autres.

Avec VHDL il faut essayer de penser à la structure qui va être générée par le synthétiseur pour écrire une bonne description.

Exemple : Pour le décodeur 1 parmi 4, l'ordre dans lequel seront écrites les instructions n'a aucune importance.

```
architecture DESCRIPTION of DECOD1_4 is begin
D0 <= (not(IN1) and not(IN0));     -- première instruction
D1 <= (not(IN1) and IN0);          -- deuxième instruction
D2 <= (IN1 and not(IN0));          -- troisième instruction
```

D3 <= (IN1 and IN0);- - quatrième instruction

end DESCRIPTION;

L'architecture ci dessous est équivalente :

architecture DESCRIPTION of DECOD1_4 is begin

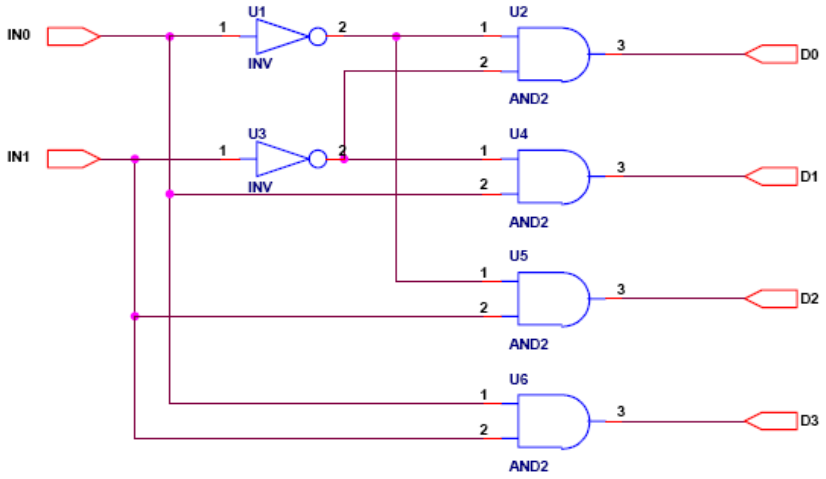
D1 <= (not(IN1) and IN0); -- deuxième instruction

D2 <= (IN1 and not(IN0)); -- troisième instruction

D0 <= (not(IN1) AND not(IN0)); -- première instruction

D3 <= (IN1 AND IN0); -- quatrième instruction end DESCRIPTION;

L'instruction définissant l'état de D0 à été déplacée à la troisième ligne, la synthèse de cette architecture est équivalente à la première.



IV.1) Les opérateurs.

IV.1.1) L'affectation simple : <=

Dans une description VHDL, c'est l'opérateur le plus utilisé. En effet il permet de modifier l'état d'un signal en fonction d'autres signaux et/ou d'autres opérateurs.

Exemple avec des portes logiques : S1 <= E2 and E1 ;

Les valeurs numériques que l'on peut affecter à un signal sont les suivantes :

- '1' ou 'H' pour un niveau haut avec un signal de 1 bit.
- '0' ou 'L' pour un niveau bas avec un signal de 1 bit.
- 'Z' pour un état haute impédance avec un signal de 1 bit.

- '-' pour un état quelconque, c'est à dire '0' ou '1'. Cette valeur est très utilisée avec les instructions : when … else et with …. Select ….

- Pour les signaux composés de plusieurs bits on utilise les guillemets

" … " , voir les exemples ci dessous :

- Les bases numériques utilisées pour les bus peuvent être :

BINAIRE, exemple : BUS <= "1001" ; -- BUS = 9 en décimal

HEXA, exemple : BUS <= X"9" ; -- BUS = 9 en décimal

OCTAL, exemple : BUS <= O"11" ; -- BUS = 9 en décimal

Remarque : La base décimale ne peut pas être utilisée lors de l'affectation de signaux. On peut seulement l'utiliser avec certains opérateurs, comme + et – pour réaliser des compteurs.

Exemple:

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

entity AFFEC is port (
E1,E2          : in std_logic;
BUS1,BUS2,BUS3 : out std_logic_vector(3 downto 0);
S1,S2,S3,S4    : out std_logic);
end AFFEC;
architecture DESCRIPTION of AFFEC is begin
S1 <= '1'; -- S1 = 1
S2 <= '0'; -- S2 = 0
S3 <= E1; -- S3 = E1
S4 <= '1' when (E2 ='1') else 'Z';-- S4 = 1 si E1=1 sinon S4
                                  -- prend la valeur haute impédance
BUS1 <= "1000";           -- BUS1 = "1000"
BUS2 <= E1 & E2 & "10";   -- BUS2 = E1 & E2 & 10
BUS3 <= x"A";             -- valeur en HEXA -> BUS3 = 10(déc)

end DESCRIPTION;
```

IV.1.2) Opérateur de concaténation : &.

Cet opérateur permet de joindre des signaux entre eux .

Exemple :

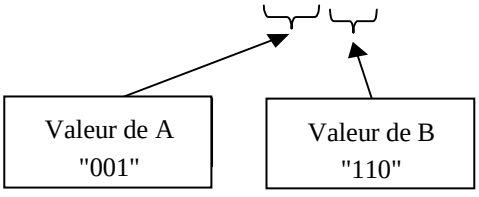
-- Soit A et B de type 3 bits et S1 de type 8 bits

-- A = "001" et B ="110"

S1 <= A & B & "01" ;

-- S1 prendra la valeur suivante après cette affectation

-- S1 = "001110"



IV.1.3) Opérateurs logiques.

Opérateur	VHDL
ET	and
NON ET	nand
OU	or
NON OU	nor
OU EXCLUSIF	xor
NON OU EXCLUSIF	xnor
NON	not
DECALAGE A	sll
DECALAGE A	srl
ROTATION A	rol
ROTATION A	ror

Exemples :

```
S1 <= A sll 2 ;  -- S1 = A décalé de 2 bits à gauche.
S2 <= A rol 3 ;  -- S2 = A avec une rotation de 3 bits à gauche
S3 <= not (R);   -- S3 = R
```

Remarque : Pour réaliser des décalages logiques en synthèse logique, il est préférable d'utiliser les instructions suivantes :

Décalage à droite :

```
-- Si A est de type std_logic_vector(7 downto 0)
S1 <= '0' & A(7 downto 1); -- décalage d'un bit à droite
S1 <= "000" & A(7 downto 3); -- décalage de trois bits à droite
```

Décalage à gauche :

```
-- Si A est de type std_logic_vector(7 downto 0)
S1 <= A(6 downto 0) & '0'; -- décalage d'un bit à gauche
S1 <= A(4 downto 0) & "000"; -- décalage de trois bits à gauche
```

IV.1.4) Opérateurs arithmétiques.

Opérateur	VHDL
ADDITION	**+**
SOUSTRACTION	**-**
MULTIPLICATION	*****
DIVISION	**/**

Remarque N°1 : Pour pouvoir utiliser les opérateurs ci-dessus il faut rajouter les bibliothèques suivantes au début du fichier VHDL:

Use ieee.numeric_std.all ;

Use ieee.std_logic_arith.all ;

Exemples :

S1 <= A – 3 ; -- S1 = A – 3

-- On soustrait 3 à la valeur de l'entrée / signal A

S1 <= S1 + 1 ; -- On incrémente de 1 le signal S1

Remarque N°2 : Attention l'utilisation de ces opérateurs avec des signaux comportant un nombre de bits important peut générer de grandes structures électroniques.

Exemples :

S1 <= A * B ;-- S1 = A multiplié par B : A et B sont codés sur 4 bits

S2 <= A / B ;-- S2 = A divisé par B : A et B sont codés sur 4 bits

VI.1.5) Opérateurs relationnels.

Ils permettent de modifier l'état d'un signal ou de signaux suivant le résultat d'un test ou d'une condition. En logique combinatoire ils sont souvent utilisés avec les instructions :

- when … else …
- with …. Select ….

Voir ci-dessous.

Opérateur	VHDL
Egal	=
Non égal	/=
Inférieur	<
Inférieur ou égal	<=
Supérieur	>
Supérieur ou égal	>=

IV.2) Les instructions du mode « concurrent ».

IV.2.1) Affectation conditionnelle :

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

```
SIGNAL <= expression when condition
[else expression when condition] [else expression];
```

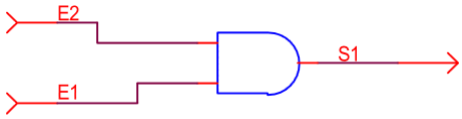
Remarque : l'instruction [else expression] n'est pas obligatoire mais elle for- tement conseillée, elle permet de définir la valeur du SIGNAL dans le cas où la condition n'est pas remplie.

On peut mettre en cascade cette instruction voir l'exemple N°2 ci-dessous

Exemple N°1 :

```
-- S1 prend la valeur de E2 quand E1='1' sinon S1 prend la valeur '0'
        S1 <= E2 when ( E1= '1') else '0';
```

Schéma correspondant :

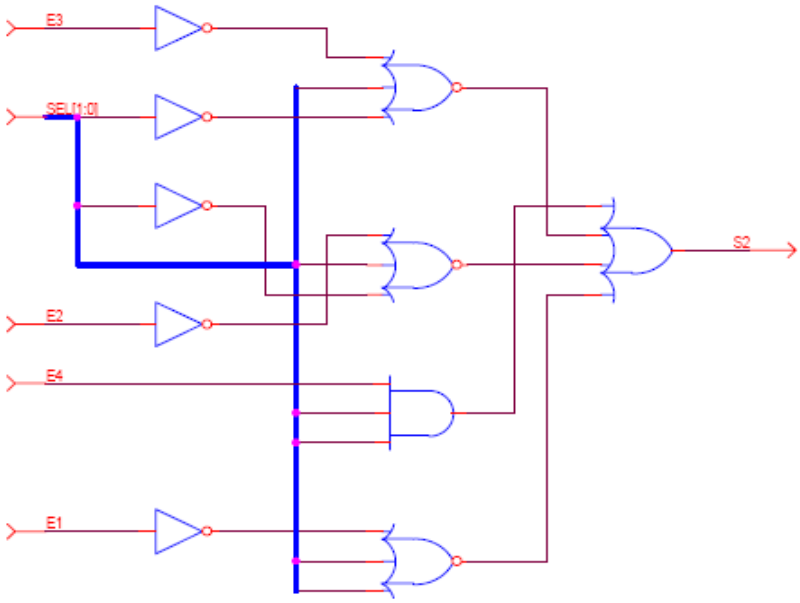


Exemple N°2 :

-- Structure évoluée d'un multiplexeur 4 vers 1

```
S2 <= E1 when (SEL="00" ) else
E2 when (SEL="01" ) else
E3 when (SEL="10" ) else
E4 when (SEL="11" )
else '0';
```

Schéma correspondant après synthèse:



IV.2.2) Affectation sélective :

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

```
with SIGNAL_DE_SELECTION select
SIGNAL <=          expression when valeur_de_selection,
[expression when valeur_de_selection,]
[expression when others];
```

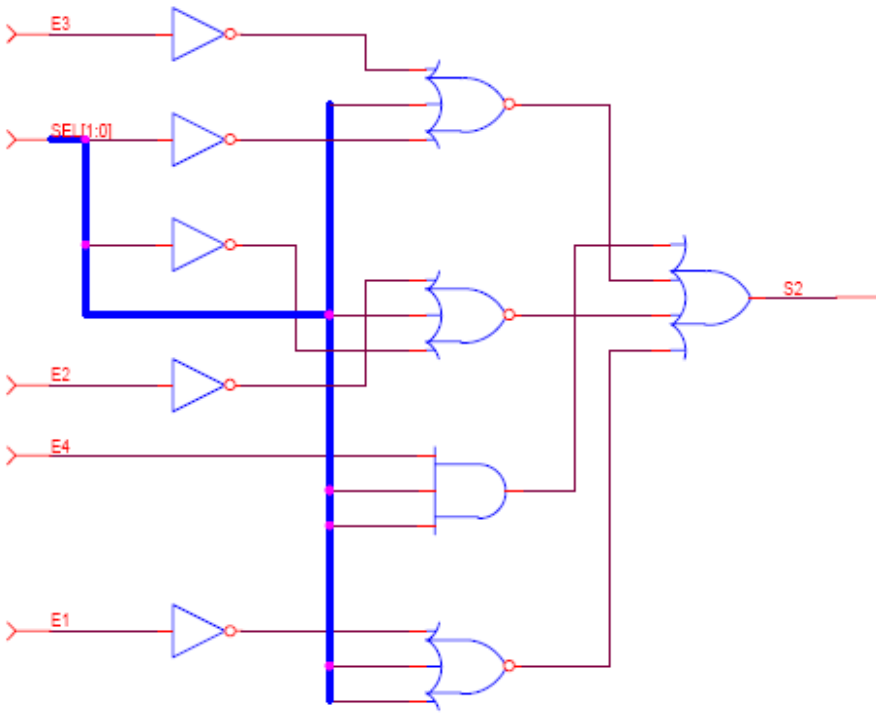
Remarque: l'instruction [expression when others] n'est pas obligatoire mais fortement conseillée, elle permet de définir la valeur du SIGNAL dans le cas où la condition n'est pas remplie.

Exemple N°1 :

```
-- Multiplexeur 4 vers 1
with SEL select
S2 <= E1 when "00",
E2 when "01",
E3 when "10",
E4 when "11",
'0' when others;
```

Remarque: when others est nécessaire car il faut toujours définir les autres cas du signal de sélection pour prendre en compte toutes les valeurs possibles de celui-ci.

Schéma correspondant après synthèse:



En conclusion, les descriptions précédentes donnent le même schéma.

L'étude des deux instructions montre toute la puissance du langage VHDL pour décrire un circuit électronique, en effet si on avait été obligé d'écrire les équations avec des opérateurs de base pour chaque sortie, on aurait eu les instructions suivantes :

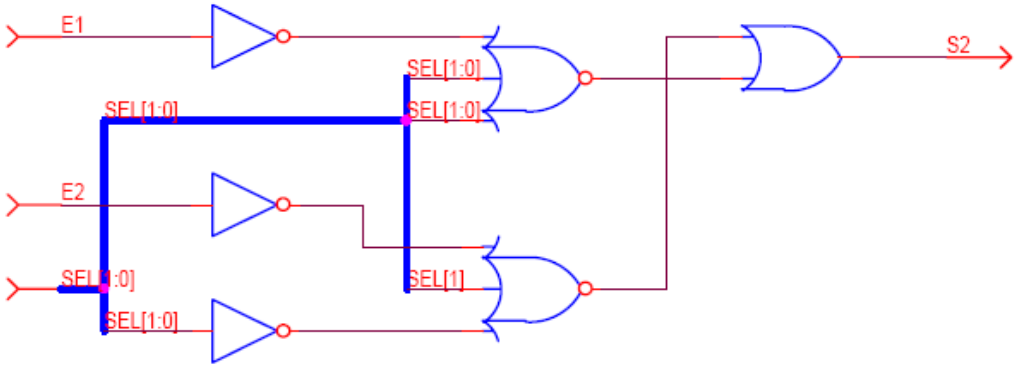
```
S2 <= (E1 and not(SEL(1)) and not(SEL(0))) or (E2 and  not SEL(1) and
(SEL(0)) or (E3 and SEL(1) and not(SEL(0))) or (E4 and SEL(1) and SEL(0));
```

L'équation logique ci-dessus donne aussi le même schéma, mais elle est peu compréhensible, c'est pourquoi on préfère des descriptions de plus haut niveau en utilisant les instructions VHDL évoluées.

Exemple N°2 : Affectation sélective avec les autres cas forcés à '0'.

```
Library ieee;
Use
ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity TABLE1 is
      port (
            E1,E2 : in std_logic;
            SEL    :  in  std_logic_vector(1  downto
            0); S2    : out std_logic);
end TABLE1;
architecture DESCRIPTION of TABLE1 is
begin
      with SEL select
            S2 <= E1  when "00",
                  E2  when "01",
                  '0' when others;   -- Pour les autres cas de SEL S2
                                     -- prendra la valeur 0 logique
end DESCRIPTION;
```

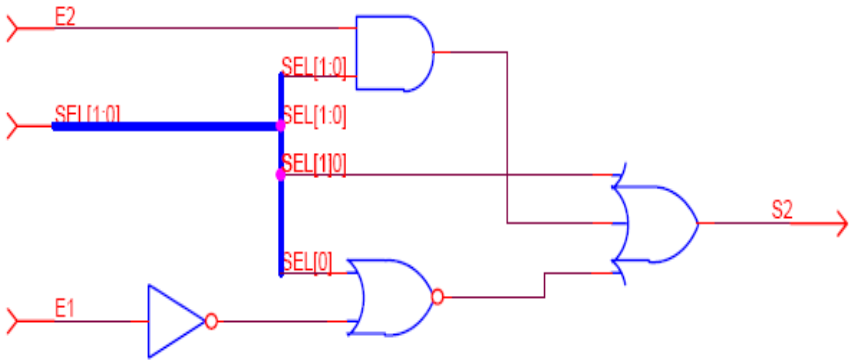
Schéma correspondant après synthèse:



Exemple N°3 : Affectation sélective avec les autres cas forcés à un '1'.

```
Library ieee;
Use
ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity TABLE2 is
      port (
            E1,E2 : in std_logic;
            SEL   : in std_logic_vector(1 downto 0);
            S2    : out std_logic);
end TABLE2;
architecture DESCRIPTION of TABLE2 is
begin
      with SEL select
            S2   <=   E1    when
                  "00",       E2
                      when
                  "01",
                  '1' when others;   -- Pour les autres cas de SEL S2
                                     -- prendra la valeur 1 logique
end DESCRIPTION;
```

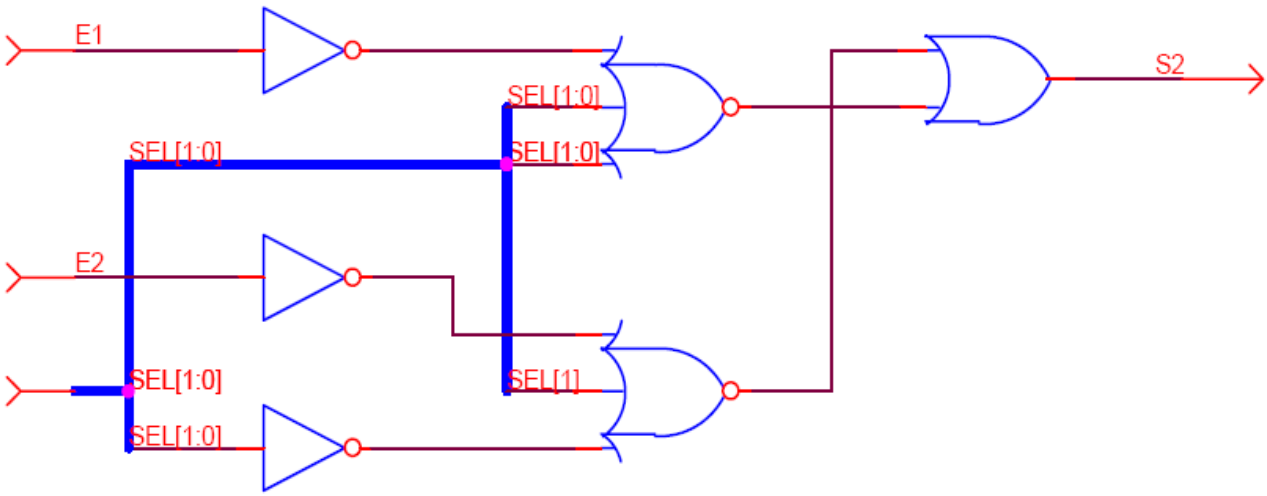
Schéma correspondant après synthèse:



Exemple N°4 :Affectation sélective avec les autres cas forcés à une valeur quelconque '-'.

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity    TABLE3
      is port (
            E1,E2 : in std_logic;
            SEL     :  in  std_logic_vector(1  downto
            0); S2    : out std_logic);
end TABLE3;
architecture DESCRIPTION of TABLE3 is
begin
      with SEL select
            S2 <= E1  when "00",
                  E2  when "01",
                  '-' when others;        -- Pour les autres cas de SEL S2
                                          -- prendra la valeur quelconque
end DESCRIPTION;
```

Schéma correspondant après synthèse:



V) Les instructions du mode séquentiel.

V.1) Définition d'un PROCESS.

Un process est une partie de la description d'un circuit dans laquelle les instructions sont exécutées séquentiellement c'est à dire les unes à la suite des autres.

Il permet d'effectuer des opérations sur les signaux en utilisant les instructions standards de la programmation structurée comme dans les systèmes à microprocesseurs.

L'exécution d'un process est déclenchée par un ou des changements d'états de signaux logiques. Le nom de ces signaux est défini dans la liste de sensibilité lors de la déclaration du process.

[Nom_du_process:]process(Liste_de_sensibilité_nom_des_signaux) Begin

-- instructions du process

end process [Nom_du_process] ;

Remarque: Le nom du process entre crochet est facultatif, mais il peut être très utile pour repérer un process parmi d'autres lors de phases de mise au point ou de simulations.

Règles de fonctionnement d'un process :

1) L'exécution d'un process a lieu à chaque changement d'état d'un signal de la liste de sensibilité.

2) Les instructions du process s'exécutent séquentiellement.

3) Les changements d'état des signaux par les instructions du process sont pris en compte à la fin du process.

V.2) Les deux principales structures utilisées dans un process.

L'assignation conditionnelle	L'assignation sélective
if condition then ***instructions*** ***[elsif condition then instructions]*** ***[else instructions]*** ***end if ;*** ***Exemple:*** ***if (RESET='1') then SORTIE <= "0000";*** ***end if ;***	***case signal_de_slection is*** ***when valeur_de_sélection => instructions*** ***[when others => instructions]*** ***end case;*** ***Exemple:*** ***case SEL is*** ***when "000" => S1 <= E1;*** ***when "001" => S1 <= '0';*** ***when "010" | "011" => S1 <='1';*** ***-- La barre | permet de réaliser*** ***-- un ou logique entre les deux*** ***-- valeurs "010" et "011"*** ***when others => S1 <= '0';*** ***end case;***

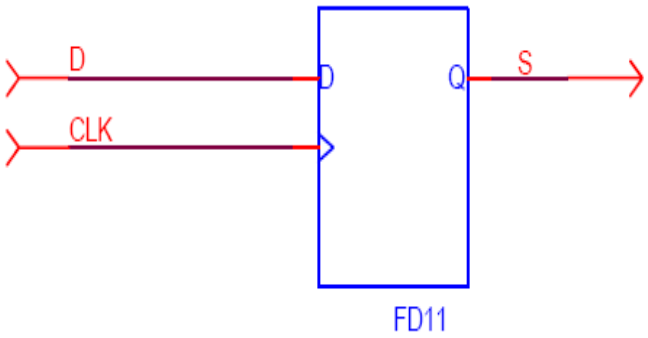
Remarque: ne pas confondre => (implique) et <= (affecte).

V.3) Exemples de *process* :

Exemple N°1 : Déclaration d'une bascule D.

```
Library ieee;
Use
ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity BASCULED is
      port
      (
            D,CLK : in std_logic;
            S     : out std_logic);
end BASCULED;
architecture  DESCRIPTION  of  BASCULED
is begin
      PRO_BASCULED : process (CLK)
      begin
            if (CLK'event and CLK ='1') then
                  S <= D;
            end if;
      end process PRO_BASCULED;
end DESCRIPTION;
```

Schéma correspondant après synthèse:



Commentaires

- Seul le signal CLK fait partie de la liste de sensibilité. D'après les règles de fonctionnement énoncées précédemment, seul un changement d'état du signal CLK va déclencher le process et par conséquent évaluer les instructions de celui-ci.
- L'instruction if (CLK'event and CLK='1') then permet de détecter un front montant du signal CLK. La détection de front est réalisée par l'attribut event appliqué à l'horloge CLK. Si on veut un déclenchement sur un front descendant, il faut écrire l'instruction suivante : if (CLK'event and CLK=

LK).

- Si la condition est remplie alors le signal de sortie S sera affecté avec la valeur du signal d'entrée D.

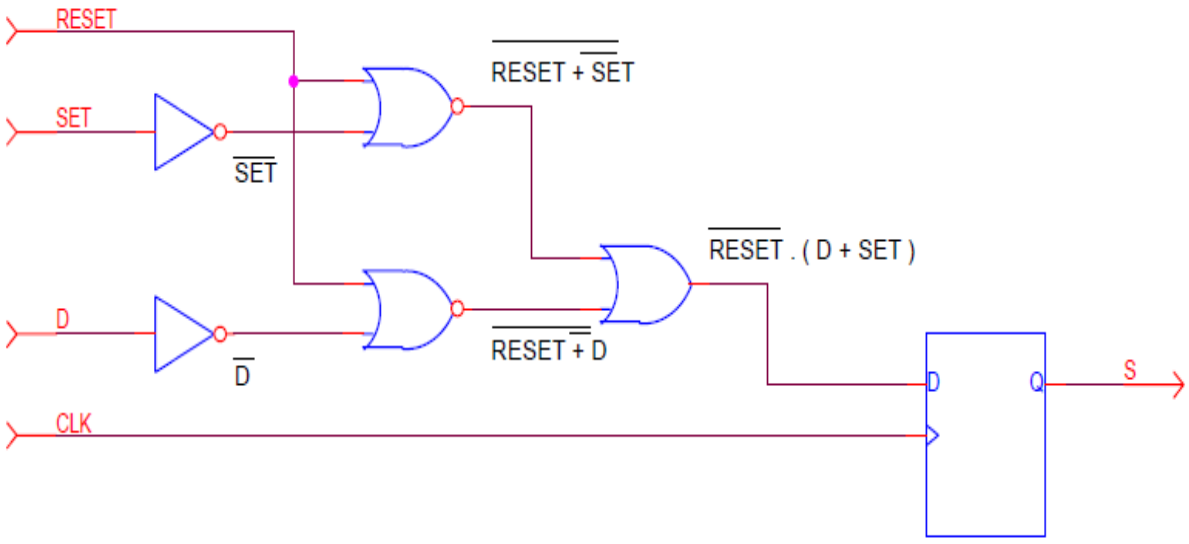
Exemple N°2 : Même exemple que précédemment mais avec des entrées de présélections de mise à zéro RESET prioritaire sur l'entrée de mise à un SET, toutes les deux sont synchrones de l'horloge CLK.

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

entity    BASCULEDSRS
      is port (
            D,CLK,SET,RESET   : in std_logic;
            S                 : out std_logic);
end BASCULEDSRS;

architecture  DESCRIPTION  of  BASCULEDSRS
is begin
      PRO_BASCULEDSRS : process (CLK)
      Begin
            if  (CLK'event  and  CLK  ='1')
            then if (RESET ='1') then
                        S      <=
                        '0';
                  elsif (SET ='1')then
                        S      <=
                        '1';
                        else
                        S <= D;
                    end
                    if;
            end if;
     end process PRO_BASCULEDSRS;
end DESCRIPTION;
```

Schéma correspondant après synthèse:



Exemple N°3 : Même exemple que précédemment mais avec des entrées de présélections,
de mise à zéro *RESET* prioritaire sur l'entrée de mise à un *SET*, toutes les deux sont asynchrones de l'horloge *CLK*.

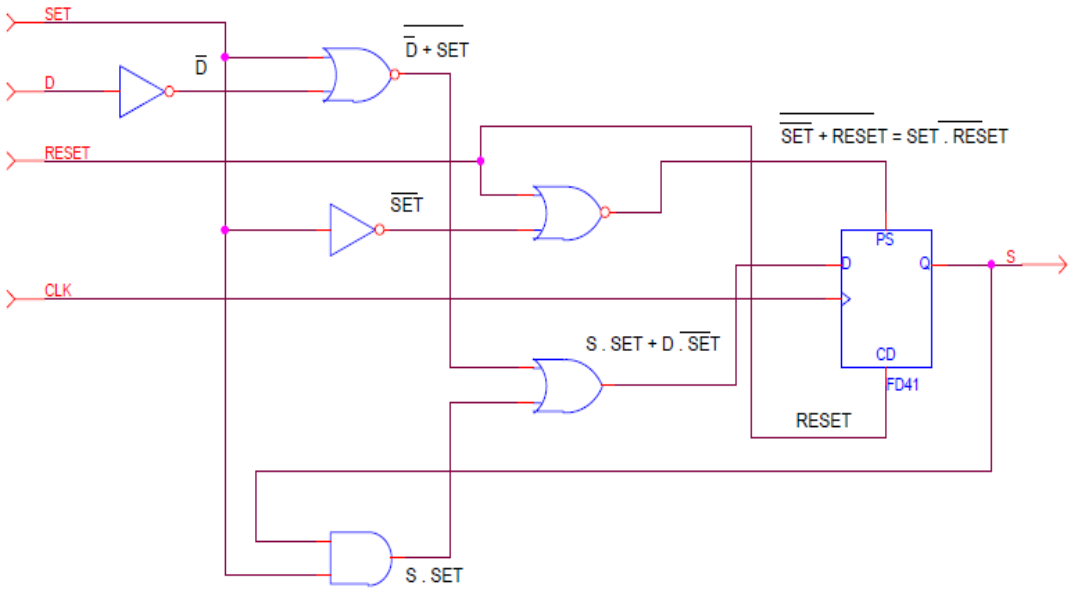
```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity BASCULEDSRA is
port (
D,CLK,SET,RESET : in std_logic;
S : out std_logic);
end BASCULEDSRA;
architecture DESCRIPTION of BASCULEDSRA is
begin
PRO_BASCULEDSRA : process (CLK,RESET,SET)
Begin
if (RESET ='1') then
S <= '0';
elsif (SET ='1')then
S <= '1';
elsif (CLK'event and CLK ='1') then
S <= D;
end if;
end process PRO_BASCULEDSRA;
end DESCRIPTION;
```

Schéma correspondant après synthèse:

Commentaire

L'entrée RESET est prioritaire sur l'entrée SET qui est à son tour prioritaire sur le front montant

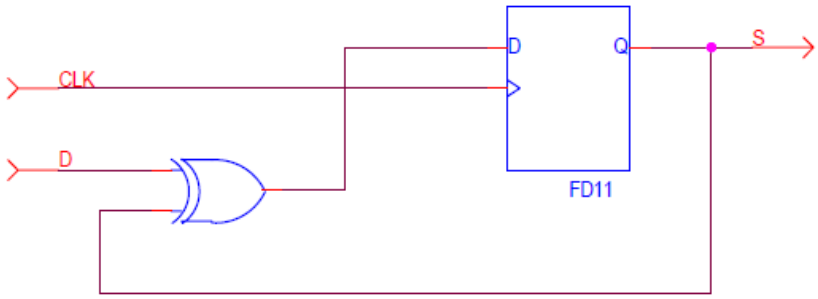
de l'horloge CLK. Cette description est asynchrone car les signaux d'entrée SET et RESET sont mentionnés dans la liste de sensibilité du process.



Exemple N°4 : Bascule T.

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

Schéma correspondant après synthèse:



Commentaires :La description ci-dessus fait appel à un signal interne appelé S_INTERNE, pourquoi faire appel à celui-ci ? La réponse est que le signal S est déclaré comme une sortie dans l'entité, et par conséquent on ne peut pas utiliser une sortie en entrée.

Pour contourner cette difficulté on utilise un signal interne qui peut être à la fois une entrée ou une sortie. Avec cette façon d'écrire, les signaux de sortie d'une description ne sont jamais utilisés comme des entrées. Cela permet une plus grande portabilité du code.

Si on ne déclare pas de signal interne, le synthétiseur renverra certainement une erreur du type : Error, cannot read output: s; [use mode buffer or inout].

Le synthétiseur signale qu'il ne peut pas lire la sortie S et par conséquent, celle-ci doit être du type buffer ou inout.

Remarque : Si on souhaite ne pas utiliser de variable interne on peut déclarer le signal S de type buffer ou inout.

Ce qui donne pour descriptions :

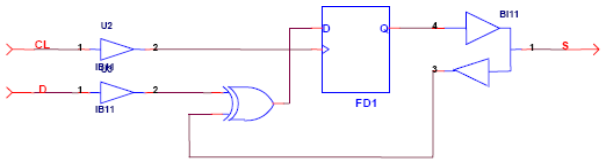
Avec *S* de type *inout*

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

entity   BASCULET   is
       port (
              D,CLK   : in std_logic;
              S       : inout std_logic);
end BASCULET;

architecture  DESCRIPTION  of  BASCULET  is
begin
       PRO_BASCULET  :  process  (CLK)
       Begin
              if (CLK'event and CLK='1') then if
                     (D='1') then
                            S <= not (S);
                     end if;
              end if;
       end process PRO_BASCULET;
end DESCRIPTION;
```

Schéma correspondant après synthèse:



Commentaires :

On peut constater que ***S*** est bien du type **inout**.

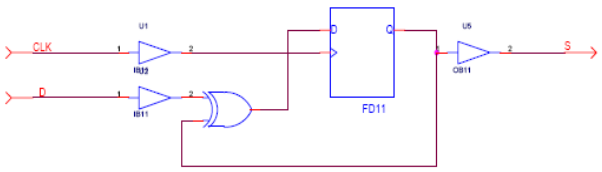
Avec S de type buffer

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

entity   BASCULET
       is port (
              D,CLK   : in std_logic;
              S       : buffer std_logic);
end BASCULET;

architecture  DESCRIPTION  of  BASCULET
is begin
       PRO_BASCULET    :    process
       (CLK) Begin
              if   (CLK'event   and   CLK='1')
                     then if (D='1') then
                            S <= not (S);
                     end
                     if;
              end if;
       end process PRO_BASCULET;
end DESCRIPTION;
```

Schéma correspondant après synthèse:



Commentaires :

On peut constater que ***S*** est bien du type ***buffer***

V.4) Les compteurs :

Ils sont très utilisés dans les descriptions VHDL. L'écriture d'un compteur peut être très simple comme très compliquée. Ils font appels aux process.

V.4.1) Compteur simple :

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity CMP4BITS is
PORT (
      CLOCK : in std_logic;
      Q     : inout std_logic_vector(3 downto 0));
end CMP4BITS;

architecture  DESCRIPTION  of  CMP4BITS
is begin
  process (CLOCK)
  begin
    if (CLOCK ='1' and CLOCK'event) then
      Q <= Q + 1;
    end
    if;
  end
  process;
end DESCRIPTION;
```

Commentaires :

La description ci-dessus est très simple. Le déclenchement du process se fe- ra sur un changement d'état du signal CLOCK, l'incrémentation de la sortie Q se fera sur le front montant de l'horloge CLOCK.

Cette description fort simple, appelle quelques commentaires :

- L'incrémentation du compteur est réalisée par l'opérateur + associé à la valeur 1. Cela est logique, mais elle l'est beaucoup moins pour les PLDs et les synthétiseurs, pourquoi ? La réponse est que les entrées et sorties ainsi que les signaux sont déclarés de type std_logic ou std_logic_vector, et par conséquent on ne peut pas leur associer de valeur entière décimale.
- Un signal peut prendre comme valeur les états '1' ou '0' et un bus n'importe quelle valeur, du moment qu'elle est écrite entre deux guillemets "1010" ou X"A" ou o"12", mais pas une valeur comme par exemple

1,2,3,4. Ces valeurs décimales sont interprétées par le synthétiseur comme des valeurs entières (integer), on ne peut pas par défaut addi- tionner un nombre entier 1 avec un bus de type électronique (st_logic_vector), c'est pour cela que l'on rajoute dans la partie déclaration des bibliothèques les lignes :

Use ieee.numeric_std.all;

Use ieee.std_logic_unsigned.all;

Ces deux bibliothèques ont des fonctions de conversions de types et

elles permettent d'associer un entier avec des signaux électroniques. Elles permettent d'écrire facilement des compteurs, décompteurs, additionneurs, soustracteurs, …..

Remarque : Il ne faut pas oublier de les rajouter dans les descriptions.

- Le signal Q est déclaré dans l'entité de type inout, cela est logique car il est utilisé à la fois comme entrée et comme sortie pour permettre l'incrémentation du compteur. Ce type d'écriture est peu utilisé car elle ne permet pas au code d'être portable, on préféra utiliser un signal interne, celui-ci peut être à la fois une entrée et une sortie.

Même description en utilisant un bus interne:

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

entity CMP4BITS is
PORT (
      CLOCK : in std_logic;
      Q     : out std_logic_vector (3 downto 0));
```

```
end CMP4BITS;

architecture DESCRIPTION of CMP4BITS is
signal Q_BUS_INTERNE : std_logic_vector(3 downto 0));
begin
  process (CLOCK)
  begin
    if (CLOCK ='1' and CLOCK'event) then
      Q_BUS_INTERNE <= Q_BUS_INTERNE + 1;
    end if;
  end process;

  Q <= Q_BUS_INTERNE;   -- affectation du bus interne au
                        -- signal de sortie Q
end DESCRIPTION;
```

V.4.2) Compteur mise à un SET et mise à zéro RESET :

V.4.2.1) Compteur 3 bits avec remise à zéro asynchrone.

```
Library ieee;
Use
ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity CMP3BITS is
PORT (
      CLOCK : in std_logic;
      RESET : in std_logic;
      Q     : out std_logic_vector(2 downto 0));
end CMP3BITS;
architecture  DESCRIPTION  of  CMP3BITS  is
signal  CMP:  std_logic_vector  (2  downto
0); begin
  process (RESET,CLOCK)
  begin
    if RESET ='1' then
      CMP <= "000";
    elsif (CLOCK ='1' and CLOCK'event) then
      CMP <= CMP + 1;
   end
   if;
  end
  process;   Q
  <= CMP;
end DESCRIPTION;
```

V.4.2.2) Compteur 3 bits avec remise à zéro synchrone.

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity CMP3BITS is
PORT (
      CLOCK      :      in
      std_logic;  RESET  :
      in std_logic;
      Q     : out std_logic_vector (2 downto 0));
end CMP3BITS;
architecture DESCRIPTION of CMP3BITS is
signal CMP: std_logic_vector (2 downto 0);
```

```
begin
  process (CLOCK)
  begin
    if (CLOCK ='1' and CLOCK'event) then
      if RESET ='1' then
         CMP        <=
         "000";
      else
         CMP  <=  CMP  +
         1;
      end if;
    end
    if;
  end process;
  Q <= CMP;
end DESCRIPTION;
```

Quelle différence entre les deux descriptions ? Dans la deuxième le signal RESET n'est plus dans la liste de sensibilité, par conséquence le process ne sera déclenché que par le signal CLOCK. La remise à zéro ne se fera que si un front montant sur CLOCK a lieu.

V.4.3) Compteur / Décompteur à entrée de préchargement :

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

entity    CMP4BITSLUD
is
PORT
(
      RESET, CLOCK, LOAD, UP: in std_logic;
            DATA  : in std_logic_vector (3 downto 0);
            Q     : out std_logic_vector (3 downto 0));
end
CMP4BITSLUD;

architecture  DESCRIPTION  of  CMP4BITSLUD
is signal CMP: std_logic_vector (3 downto
0); begin
  process (RESET,CLOCK)
  begin
    if RESET ='1' then
      CMP <= "0000";          -- Remise à zero asynchrone du compteur
      elsif  (CLOCK  ='1'  and  CLOCK'event)
      then  if (LOAD ='1') then
            CMP <= DATA;            -- Préchargement synchrone
            Else
            if (UP ='1')
              then CMP <= CMP + 1;   -- Incrémentation synchrone
              else
                 CMP <= CMP - 1;   -- Décrémentattion synchrone
            end if;
      end if;
    end
    if;
```

```
  end
  process;
  Q <= CMP;
end
DESCRIPTION;
```

Remarque : La mise à zéro des sorties du compteur passe par l'instruction :

CMP <= "0000";

Une autre façon d'écrire cette instruction est :

CMP <= (others => '0') ;

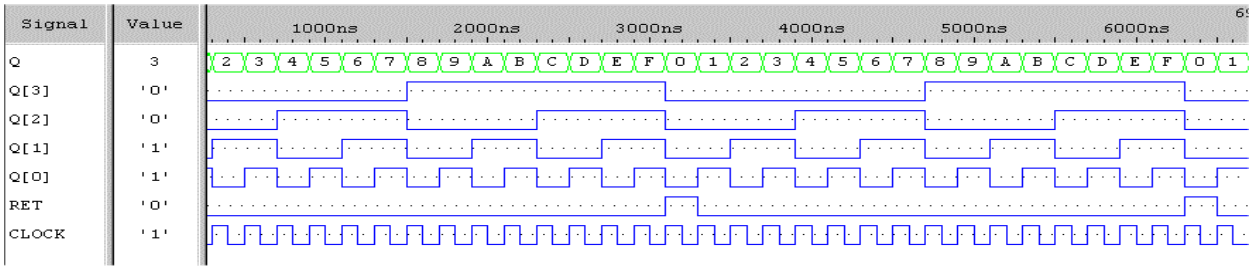
Cette dernière est très utilisée dans les descriptions car elle permet de s'affranchir de la taille du bus. En effet others=>'0' correspond à mettre tous les bits du bus à zéro quelque soit le nombre de bits du bus. De la même façon on peut écrire others=>'1' pour mettre tous les bits à un.

V.4.4) Les erreurs classiques avec l'utilisation de *process* :

Exemple : compteur avec retenue (fonctionnement incorrect).

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity CMP4BITSRET is
PORT (
      RESET, CLOCK      : in std_logic;
      RET               : out std_logic;
      Q                 : out std_logic_vector (3 downto 0));
end CMP4BITSRET;
architecture DESCRIPTION of CMP4BITSRET is
signal CMP: std_logic_vector (3 downto 0);
begin
  process (RESET,CLOCK)
  begin
    if RESET ='1' then
            CMP <= "0000";
    elsif (CLOCK ='1' and CLOCK'event) then
      CMP <= CMP + 1;
      if (CMP = "1111") then
          RET <= '1';
      else
      RET <= '0';
     End if;
    End if;
  end rocess;
  Q <= CMP;
          end DESCRIPTION
```

Signaux de simulation obtenus :



Les résultats de simulation appellent quelques commentaires :

On s'aperçoit que le signal de retenue RET passe au niveau logique haut quand la valeur du compteur vaut 0, pourquoi ?Il ne faut pas oublier la règle 3 des process (voir page N°21) qui dit que les valeurs de signaux à l'intérieur d'un process ne sont mis à jour qu'à la fin de celui-ci.

Dans notre cas, prenons l'état où CMP=14, au coup d'horloge suivant on incrémente le compteur CMP, mais la nouvelle valeur ne sera affectée à CMP qu'à la fin du process, donc quand le test pour valider le signal de retenue est effectué, la valeur de CMP est égale à 14, et celui-ci n'est pas valide.

Au coup d'horloge suivant CMP=15 et CMP est incrémenté donc prendra la valeur 0 à la fin du process., mais la condition CMP= "1111" sera valide et le signal de retenue RET passera au niveau logique un.

Comment faire pour pallier à ce problème ? deux solutions :

1) Il faut anticiper d'un coup d'horloge, on valide la retenue quand la valeur du compteur vaut 14, c'est à dire n-1.

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity CMP4BITSRET is
PORT (
      RESET, CLOCK      : in std_logic;
      RET               : out std_logic;
      Q                 : out std_logic_vector (3 downto 0));
end CMP4BITSRET;

architecture DESCRIPTION of CMP4BITSRET is
signal CMP: std_logic_vector (3 downto 0);
begin
  process (RESET,CLOCK)
   begin
     if RESET ='1' then
        CMP <= "0000";
    elsif (CLOCK ='1' and CLOCK'event) then
      CMP <= CMP + 1;
      if (CMP = "1110") then
            -- La retenue passera à un  quand CMP = 14 décimal
            RET <= '1';
```

```
            else
                RET <= '0';
          end if;
    end if;
   end
   process;   Q
   <= CMP;
end DESCRIPTION;
```

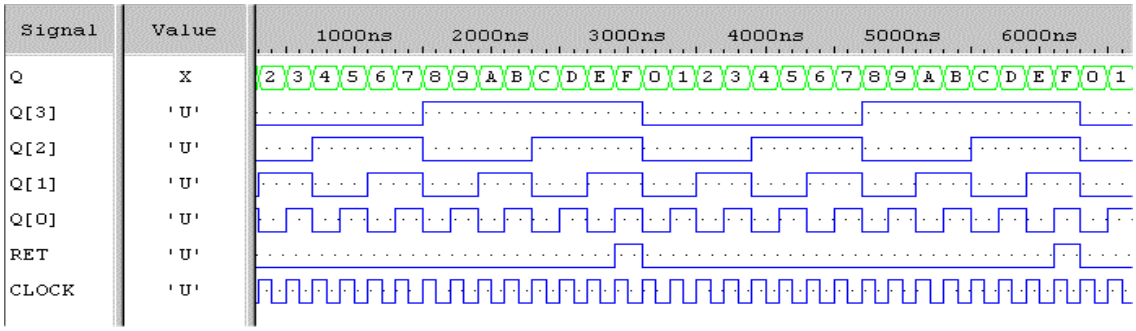
Remarque : Dans ce cas la validation de la retenue s'effectue de façon synchrone car elle est dans le process, mais la description est peu lisible.

2) Le test de validation de la retenue est effectuée en dehors du process.

```
Library ieee;
Use
ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity CMP4BITSRET is
PORT (
      RESET, CLOCK      : in std_logic;
      RET                        :  out
      std_logic;
      Q                 : out std_logic_vector (3 downto 0));
end CMP4BITSRET;
architecture DESCRIPTION of CMP4BITSRET is
signal CMP: std_logic_vector (3 downto 0);
begin
  process (RESET,CLOCK)
  begin
    if RESET='1' then
      CMP <= "0000";
    elsif (CLOCK ='1' and CLOCK'event) then
      CMP <= CMP + 1;
    end
    if;
  end process;
  Q <= CMP;
  -- Validation de la retenue
  RET <= '1' when (CMP = "1111") else '0';
end DESCRIPTION;
```

Remarque : Dans ce cas la validation de la retenue s'effectue de façon asynchrone car elle est en dehors du process, mais la description est lisible.

Signaux de simulation obtenus :



V.4.5) Compteur BCD deux digits :

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

entity CMP_BCD8 is
      PORT (
            RESET, CLK : in std_logic;
            Q     : out std_logic_vector (7 downto 0));
end CMP_BCD8;

architecture DESCRIPTION of CMP_BCD8 is
signal DIGIT_MSB, DIGIT_LSB: std_logic_vector (3 downto 0);
begin
  process (RESET,CLK)
  begin
    if RESET ='1' then
      DIGIT_LSB <= (others=>'0');
      DIGIT_MSB <= (others=>'0');
    elsif  (CLK  ='1'  and  CLK'event)
          then if DIGIT_LSB < 9 then
             DIGIT_LSB <= DIGIT_LSB + 1;
          els
          e
             DIGIT_LSB <= (others=>'0');
             if DIGIT_MSB < 9 then
                  DIGIT_MSB <= DIGIT_MSB + 1 ;
                      Else
                      DIGIT_MSB <= (others=>'0');
             end if;
          end if;
    end
    if;
  end
  process;
  Q <= DIGIT_MSB & DIGIT_LSB;
end DESCRIPTION;
```

Signaux de simulation obtenus :