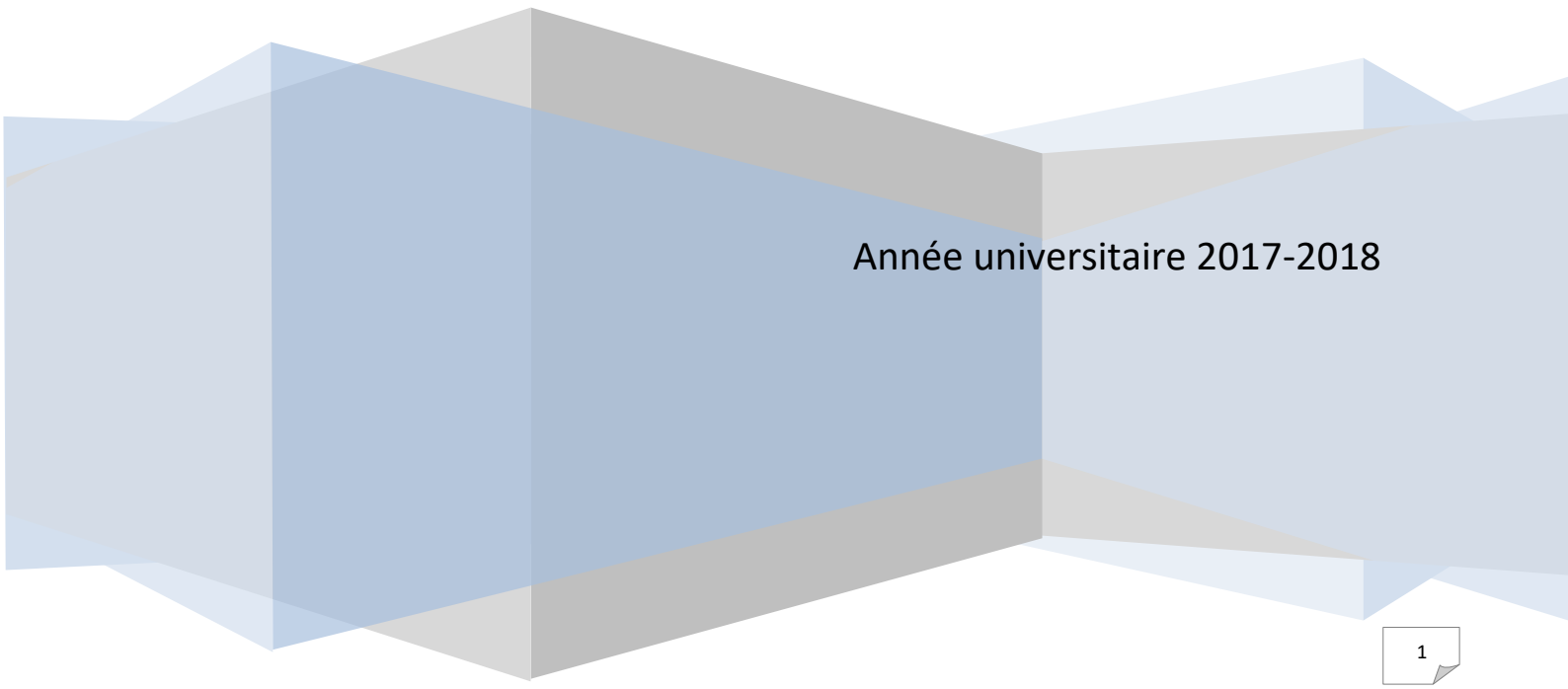
République Algérienne Démocratique et Populaire
Ministère de l'enseignement Supérieur et de la Recherche Scientifique
Université Abou Bekr Belkaïd – Faculté de Technologie

Département de Génie-Biomédical

Introduction à la technologie FPGA et à la logique programmable

Initiation au langage VHDL - Cours et TP

Mme Benseghir née Lazzouni Sihem Amel



Année universitaire 2017-2018

Avant propos

Il existe une loi empirique, appelée **loi de Moore**, qui dit que la densité d'intégration dans les circuits intégrés numériques à base de silicium double tous les 18 à 24 mois, à prix du circuit équivalent. La loi de Moore s'est révélée remarquablement exacte jusqu'à ce jour et elle explique en grande partie l'évolution de l'électronique numérique de ses origines à nos jours.

Au début de l'ère des circuits intégrés numériques, les fonctions logiques telles que les portes, les registres, les compteurs et les ALU, étaient disponibles en circuit TTL. On parlait de composants SSI (Small Scale Integration) ou MSI (Medium Scale Integration) pour un tel niveau d'intégration. Le nombre de transistors intégrés sur une puce de silicium n'a cessé d'augmenter, ce qui a poussé les fabricants à mettre sur le marché des composants de plus en plus sophistiqués comme les microprocesseurs, et les circuits à architecture programmable tels les FPGAs et les ASICs. Ces derniers nécessitent l'apprentissage d'un langage évolué de description matérielle comme le VHDL.

VHDL est l'acronyme de **VHSIC HDL (*Very High Speed Integrated Circuit Hardware Description Language*)**, c'est un langage de description matérielle qui a été créé dans les années 1980 à la demande du département de la défense américaine.

La première version du **VHDL** accessible au public a été publiée en 1985, et a fait l'objet d'une norme internationale en 1986 par l'institut des ingénieurs électriciens et électroniciens (**IEEE**).

De nos jours, le langage **VHDL** devient un outil indispensable pour la conception des systèmes électroniques intégrés, il est proposé par la grande majorité des sociétés de développement et la commercialisation d'**ASIC** et d'**FPGA** telle que la société américaine **Xilinx**. Avec un langage de description matérielle et un **FPGA** (***Field Programmable Gate Array***), un concepteur peut développer rapidement et simuler un circuit numérique sophistiqué, de l'implémenter sur une carte de prototypage, et de vérifier son fonctionnement. L'introduction de cette classe de circuits numériques dans l'enseignement s'est donc révélée être un choix pertinent.

Ce polycopié de Cours/TP correspond aux enseignements de licence et de Master en Génie biomédical et Génie électrique pour la partie « Logique programmable ». Le langage utilisé est le VHDL puisque l'on s'intéresse aux méthodologies de développement des FPGAs.

Sommaire

I. Technologies de logique programmable

I.1. Circuits SSI, MSI et LSI

Les premiers circuits numériques intégrés sont apparus sur le marché dans les années 1960. On les classifiait alors selon le nombre de transistors qu'ils intégraient. Les trois acronymes de base, SSI, MSI et LSI, référaient respectivement à *Small*, *Medium* et *Large Scale Integration*. Un circuit SSI contenait de l'ordre de 10^2 transistors, un circuit MSI de l'ordre de 10^3, et un circuit LSI de l'ordre de 10^4 transistors.
Une famille de circuits SSI/MSI très populaire jusqu'au début des années 1990 était la série 7400. Normalisés dans l'industrie, ils étaient manufacturés par plusieurs fournisseurs. Les deux derniers chiffres reflétaient la fonction logique réalisée et la position des signaux sur les pattes de la puce. Quelques exemples sont indiqués au Tableau 1-1, et une puce 7400 est illustrée à la Figure 1-1 avec son schéma interne.

numéro	fonction
7400	4 × NON-ET
7402	4 × NON-OU
7404	8 × NON
7411	3 × ET (3 entrées)
7473	2 × bascule JK avec reset

Tableau I.1 : Exemples de la série 7400

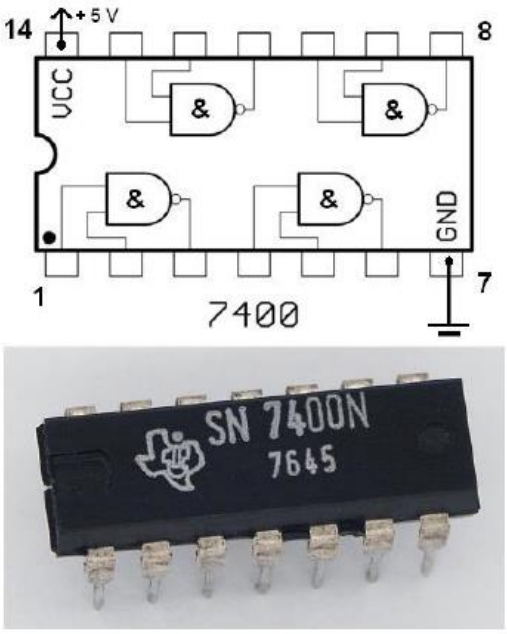


Figure I.1. Le Circuit intégré 7400

Les circuits MSI incluaient entre autres des multiplexeurs, décodeurs, registres et compteurs. Leur coût de fabrication n'était pas significativement supérieur à celui des composants SSI à cause de l'amélioration continue des processus de fabrication. Les circuits LSI incluaient entre autres des mémoires, des processeurs simples et les premiers microprocesseurs comme le 4004 d'Intel à plus de 2000 transistors.
À la fin des années 1970 et au début des années 1980, on a vu apparaître les circuits VLSI (pour *Very Large Scale Integration*) avec 10^5 transistors. L'acronyme ULSI (pour *Ultra*) a été peu utilisé, VLSI se généralisant pour tous les circuits intégrés très complexes.

L'utilisation des circuits logiques programmables change les méthodes de conception des systèmes logiques. Jusqu'à présent, la mise en œuvre des systèmes logiques consistait à utiliser les circuits logiques de fonctions standard (NON, ET, OU, NAND, Décodeurs, Multiplexeurs,…). Cette approche conduisait à adapter le système à concevoir aux circuits logiques existants. Elle se traduisait très souvent par des pertes à plusieurs niveaux.

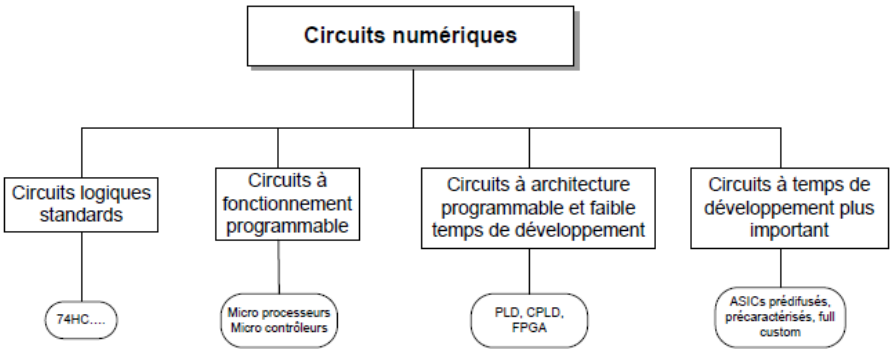
- Au niveau du circuit intégré : toutes les ressources du circuit ne sont pas utilisées. Par exemple, on n'utilise que deux portes NAND sur les quatre que le circuit intégré contient,
- Au niveau le la surface du circuit imprimé : c'est le corollaire de l'utilisation non optimale des circuits intégrés,
- Au niveau de l'évolution du produit : l'oubli ou le souhait de rajout d'une fonction conduit à la remise en question du système. Les schémas et circuits imprimés doivent être modifiés.

L'avènement des circuits logiques programmables permet d'adapter les circuits au système à concevoir. On obtient des systèmes plus compacts, plus souples à utiliser et à modifier. Un circuit logique programmable est connu sous le sigle PLD (Programmable Logic Device)

Le développement des mémoires utilisées en informatique fut à l'origine des premiers circuits logiques programmables (PLD). Ce type de produit peut intégrer dans un seul circuit plusieurs fonctions logiques programmables par l'utilisateur. Sa mise en œuvre se fait très facilement à l'aide d'un programmateur, d'un micro-ordinateur et d'un logiciel adapté.

A la fin des années 70, le monde des circuits numériques se répartit schématiquement en 4 grands groupes :

- Les fonctions standards sont utilisées pour les applications réalisées en logique câblée (Circuits TTL et CMOS)
- Les microprocesseurs, omniprésents dans les applications industrielles.
- Dans des applications trop complexes, on utilise des séquenceurs microprogrammés.
- Les circuits intégrés spécifiques (ASICs)



ASIC : Application specific Integrated Circuit (pré caractérisés = cellules standards / pré diffusés = il reste à déposer les interconnexions)

CPLD : Complex Programmable Logic Device

FPGA : Field Programmable Gate array

Figure I.2. Classes de circuits numériques

I.2. Mémoires mortes programmables

Une mémoire est un dispositif d'enregistrement, de conservation et de restitution de l'information. On distingue deux classes de mémoire à semi-conducteur :

- Les mémoires vives :
  Ce sont des mémoires pour lesquelles on ne peut pas garantir l'intégrité de l'information inscrite si entre temps il y a eu interruption de l'alimentation électrique. Lorsqu'elles sont alimentées, elles peuvent être lues et écrites.

- Les mémoires mortes :
  Ce sont des mémoires pour lesquelles on inscrit l'information dans leur structure matérielle. Elles conservent ainsi l'information même en l'absence de l'alimentation électrique. Cette classe est la plus adaptée pour être considérée comme **un circuit logique programmable**. Lorsqu'elles sont alimentées, elles ne sont généralement que lues. L'écriture appelée aussi **programmation** se fait à l'aide d'un dispositif spécial appelé programmateur.

L'élément de base d'une mémoire à semi-conducteur est le bit, stocké dans une cellule mémoire, appelée aussi point mémoire. La cellule élémentaire d'une mémoire morte peut être assimilée à un interrupteur à semi-conducteur constitué soit d'une diode soit de transistors. La cellule élémentaire d'une mémoire vive peut être assimilée à une bascule RS, JK ou D. Elle est généralement du type D.

Bus d'adresses

L'organisation d'une mémoire consiste en une association de plusieurs registres dont l'un d'entre eux peut être mis en relation avec l'extérieur, soit pour sa lecture (mémoire morte), soit pour sa lecture/écriture (mémoire vive).

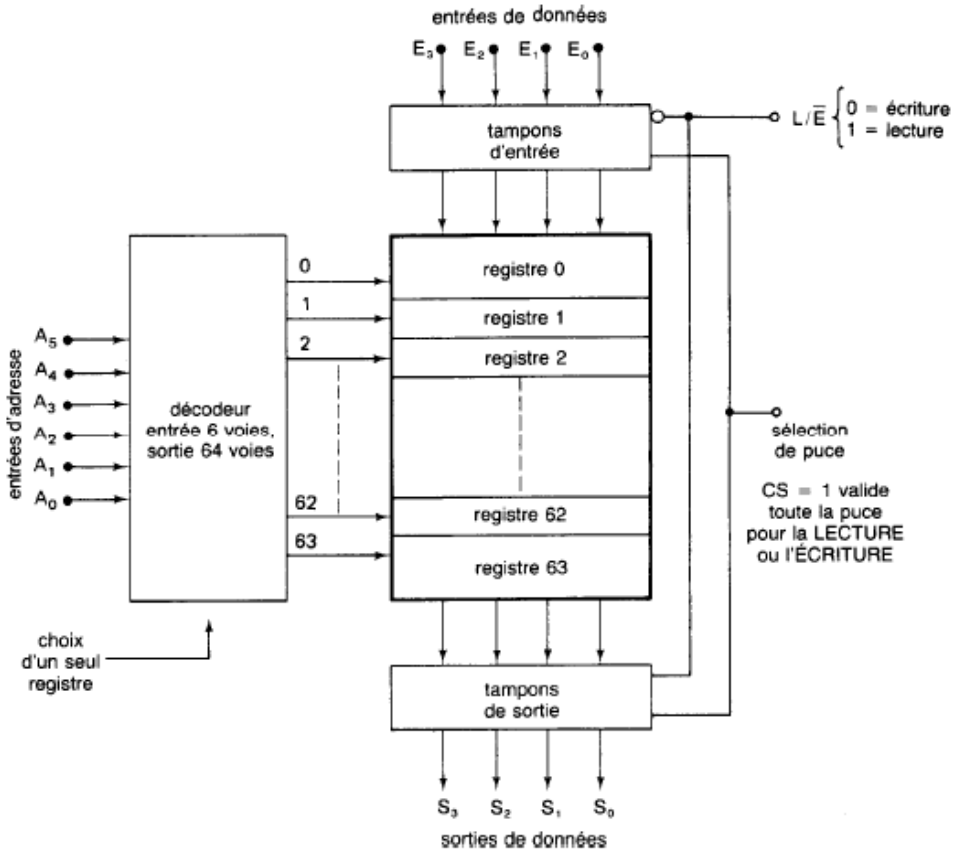


Figure I.3 : Organisation d'une mémoire

On ne peut donc accéder qu'à une case (registre) à la fois dont le format va de 1 bit à 64 bits et plus (sur l'exemple ci-dessus 4 bits sont accessibles simultanément dans chaque registre de données). La sélection d'un des registres se fait par n fils appelés **bus d'adresse.** (avec n fils d'adresse, on peut adresser 2^n cases d'adresse parmi les 2^n registres disponibles dans la mémoire). On parle de mémoire à X registre par Y données dans chaque registre ce qui nous donne la capacité totale X x Y bits (la capacité peut aussi être exprimée en octet : un octet = 8 bits). La lecture (ou l'écriture) dans un registre suppose que l'on sélectionne ce registre (A0, A1, A2, A3, A4, A5) et que l'on sélectionne le boîtier (CS) afin d'avoir accès aux données contenues dans le registre. Dans le cas de mémoire vive, on précisera aussi si l'on souhaite lire ou écrire dans ce registre (L/E).Les mémoires mortes ne sont quant à elles accessibles qu'en lecture.

I.3. Types de Mémoires mortes

1. **La ROM**

Une mémoire morte (i.e ROM : **R**ead **O**nly **M**emory) est une mémoire dont le contenu a été défini et réalisé une bonne fois pour toutes au moment de la fabrication. La fabrication de ROMs ne se conçoit que pour des séries importantes (>10.000 unités). Cette programmation est faite directement sur le wafer (galette de silicium) à l'aide des masques de programmation.

2. **PROM (Programmable ROM)**

Les premiers circuits programmables par l'utilisateur étaient des mémoires mortes programmables (*Programmable Read Only Memory* – PROM). Une PROM ne se programmait qu'une seule fois.
Les PROMs sont des mémoires non volatiles, dont le contenu comme dans le cas des ROMs, est défini une fois pour toutes. Toutefois, contrairement aux ROMs, elles sont programmables (1 seule fois) par l'utilisateur.
Il existe différents types de PROMs. Les noeuds de la matrice peuvent comprendre soit des fusibles soit des jonctions ayant une faible tension de claquage (anti-fusibles). Dans les deux cas, la programmation s'effectue en sélectionnant l'adresse désirée, en présentant la donnée sur les lignes de sortie, et en alimentant pendant un bref instant (quelques centaines de ms) le circuit avec une tension élevée (10 à 15 V suivant les cas), ce qui a pour effet de faire fondre le fusible (ouverture du circuit) ou de claquer la jonction (fermeture du circuit). Ce processus est évidemment irréversible.

Principe

L'exemple le plus simple de mémoire morte à fusibles 4x5 bits peut se schématiser de la façon suivante : un décodeur 2 vers 4 (74139) avec sorties actives à l'état bas permet de sélectionner une ligne parmi 4.
Exemple:
A l'adresse A1 A0 = 00, la ligne notée 00 est forcée à 0 et les autres lignes sont à 1. Dans ce cas les 5 bits de sortie ont la valeur 0. Avant programmation toutes les sorties sont donc à 0.

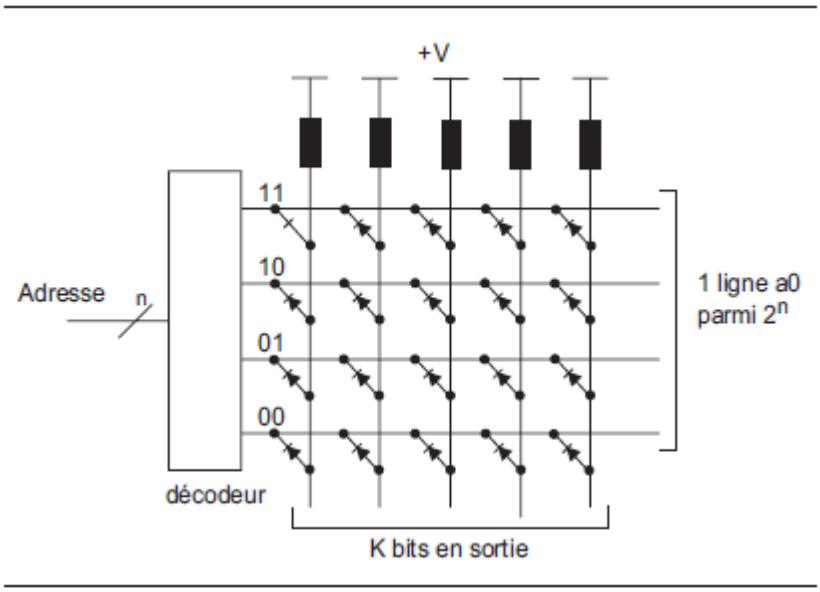


Figure I.4. : Schéma d'une PROM

Après programmation, c'est à dire après destruction de certains fusibles, on peut obtenir le schéma suivant:

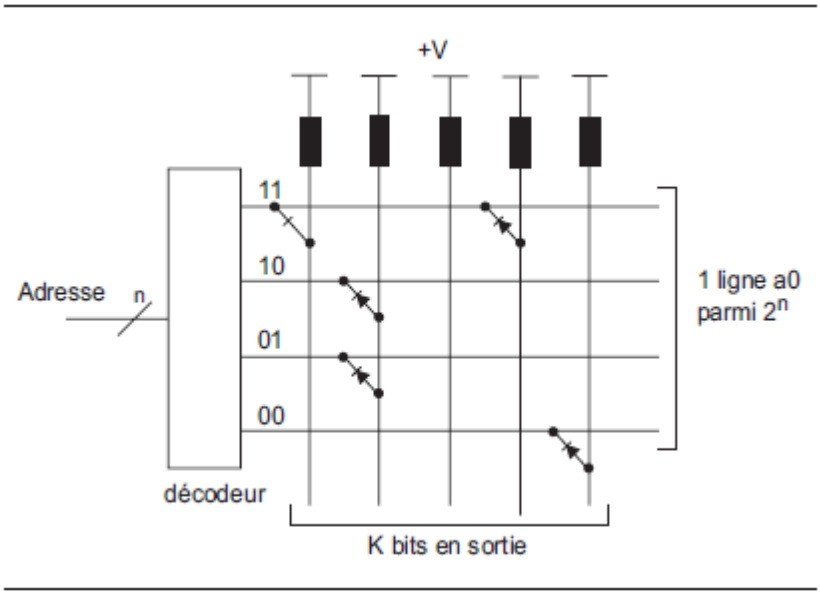


Figure I.5. : PROM programmée

Fonctionnement

Grâce à la résistance de tirage une ligne de sortie vaut 1 en l'absence de diode (liaison détruite) entre elle et le fil d'adresse sélectionnée. Si par contre, une diode est présente, elle ramène le potentiel de la ligne sortie à 0. Le contenu de cette mémoire 20 bits est alors :

adresse	$S_4S_3S_2S_1S_0$
1 1	0 1 1 0 1
1 0	1 0 1 1 1
0 1	1 0 1 1 1
0 0	1 1 1 1 0

3. EPROM (Erasable PROM)

Les EPROMs sont des mémoires non volatiles programmables électriquement puis effaçables par UV. Dans ce cas, les nœuds de la matrice sont constitués de transistors MOS à grille isolée (flottante). Cette grille peut être chargée par influence en appliquant une tension importante (10 à 15 V) entre le drain et le substrat. Une fois chargée, elle peut conserver sa charge de manière quasiment indéfinie, ce qui rend le transistor passant.
Le processus est réversible en irradiant la puce aux rayons ultraviolets pendant plusieurs minutes, ce qui décharge la grille par effet photoélectrique.

Principe

Chaque élément mémoire est composé d'un transistor MOS dont la grille est complètement isolée dans une couche d'oxyde. Par application d'une tension suffisamment élevée, qui est appelée tension de programmation, on crée des électrons ayant une énergie suffisante pour traverser la mince couche d'oxyde. Ces charges s'accumulent et se retrouvent piégées dans la grille, la couche d'oxyde entre la grille et le silicium étant trop épaisse pour que les électrons puissent la traverser. La cellule mémoire est alors programmée.

Si maintenant, on veut effacer la mémoire, on expose la puce aux rayons U.V. Les photons (ou particules d'énergie lumineuse) communiquent alors leur énergie aux électrons et leur font franchir la barrière isolante dans l'autre sens. La cellule mémoire est effacée.

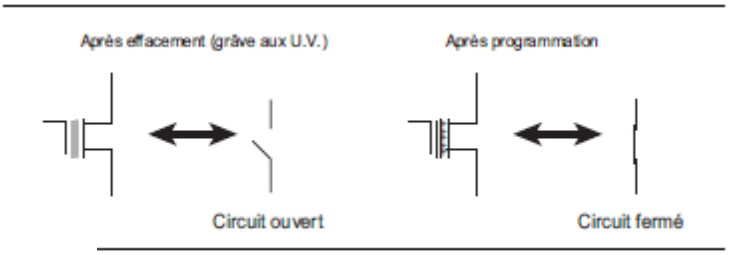


Figure I.6. : Principe de l'EPROM

Exemple :
La mémoire proposée ci-dessous est une mémoire EPROM 16 bits.

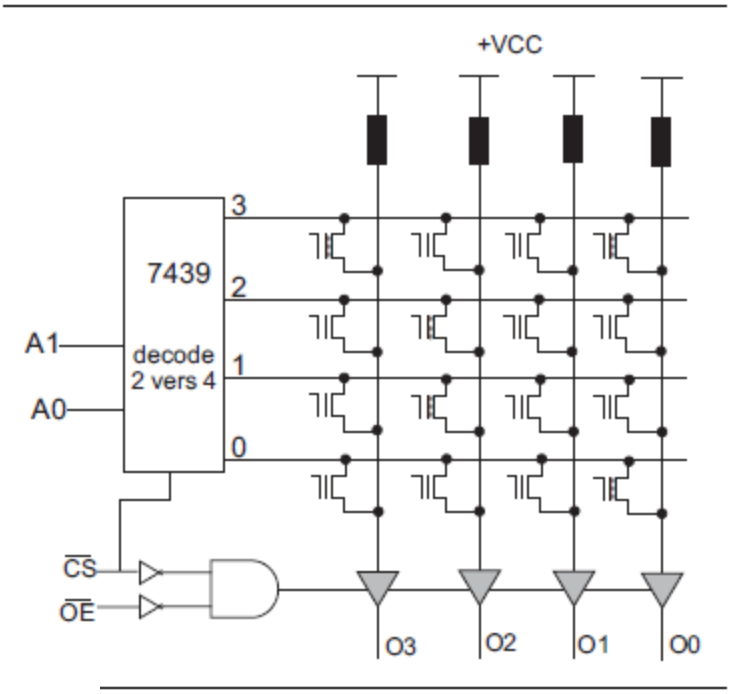


Figure I.7 : Mémoire EPROM 4x4 bits après programmation

On utilise un décodeur 2 vers 4 pour sélectionner une ligne parmi quatre. Si $\overline{CS}$ est à 0 alors le décodeur est actif et si par exemple A_1A_0 = 00, alors la ligne 0 est forcée à 0 (les autres lignes restent à 1) on obtient donc sur le bus de données $O_3O_2O_1O_0$ = 1110. En effet seul le transistor de la ligne 0 bit O_0 est programmé (grille flottante chargée).
On utilise un buffer 3 états pour déconnecter la mémoire du bus de données externe. Pour envoyer les données sur le bus de données il faudra donc activer ce buffer 3 états (conditions $\overline{CS}$ = 0 et $\overline{OE}$ =0).

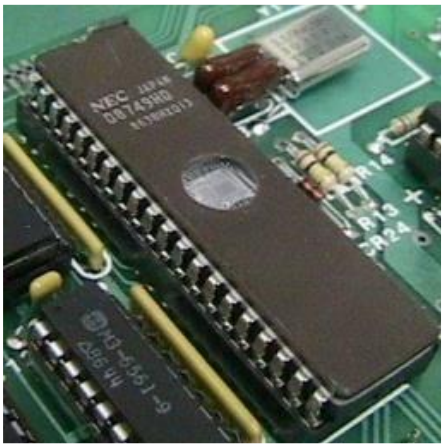


Figure I.8. : Le microcontrôleur NEC D8749 avec mémoire EPROM intégrée.

4. **Mémoire EEPROM (E2PROM) :**

C'est une EPROM effaçable électriquement, ayant donc les mêmes caractéristiques en lecture qu'une EPROM et pouvant même être programmée par un programmateur d'EPROM.

5. Les mémoires Flash

Les mémoires flash sont plus rapides que les mémoires E2PROM (en terme d'effacement et de programmation) mais ne permettent que l'effacement total de la mémoire. Les EEPROM flash utilisent comme pour les mémoires EPROM un seul transistor MOS par bit.
Les mémoires flash sont utilisées dans les PC (BIOS), ou dans les systèmes embarqués pour mémoriser les programmes importants. Les nouveaux appareils grand public (téléphones portables, cartes mémoire des appareils de photos et caméscopes...) demandent de plus en plus de capacité mémoire pour les interfaces graphiques et nouvelles fonctions. Les mémoires flash sont donc en pleine expansion.

Des structures programmables plus complexes sont également disponibles, elles permettent de mettre en relations, combinatoires et séquentielles, des entrées et des sorties : ce sont les PLD, les FPGA, etc…

II. Architectures des circuits logiques programmables

L'électronique moderne se tourne de plus en plus vers le numérique qui présente de nombreux avantages sur l'analogique : grande insensibilité aux parasites et aux dérives diverses, modularité et (re)configurabilité, facilité de stockage de l'information, etc...
Les circuits numériques nécessitent par contre une architecture plus lourde et leur mode de traitement de l'information met en œuvre plus de fonctions élémentaires que l'analogique d'où découle des temps de traitement plus longs. Aussi les fabricants de circuits intégrés numériques s'attachent à fournir des circuits présentant des densités d'intégration toujours plus élevée, pour des vitesses de fonctionnement de plus en plus grandes.
D'abord réalisées avec des circuits SSI (Small Scale Integration), les fonctions logiques intégrées se sont développées avec la mise au point du transistor MOS dont la facilité d'intégration a permis la réalisation de circuits MSI (Medium Scale Integration) puis LSI (Large Scale Integration) puis VLSI (Very Large Scale Integration). Ces deux dernières générations ont vu l'avènement des microprocesseurs et micro-contrôleurs. Bien que ces

derniers aient révolutionné l'électronique numérique par la possibilité de réaliser n'importe quelle fonction par programmation, ils traitent l'information de manière séquentielle, ne répondant pas toujours aux exigences de rapidité.
Au début des années 70 sont apparus les premiers composants (en technologie bipolaire) entièrement configurables par programmation. La nouveauté résidait dans le fait qu'il était maintenant possible d'implanter physiquement par simple programmation, au sein du circuit, n'importe quelle fonction logique, et non plus de se contenter de faire réaliser une opération logique par un microprocesseur dont l'architecture est figée.
D'abord dédiés à des fonctions simples en combinatoire (décodage d'adresse par exemple), ces circuits laissent aujourd'hui au concepteur la possibilité d'implanter des composants aussi divers qu'un inverseur et un microprocesseur au sein d'un même boîtier ; le circuit n'est plus limité à un mode de traitement séquentiel de l'information comme avec les microprocesseurs.

Le terme même de circuit programmable est ambigu, la programmation d'une FPGA ne faisant pas appel aux mêmes opérations que celle d'un microprocesseur, il serait plus juste de parler pour les PLD, CPLD et FPGA de circuits à architecture programmable ou encore de circuits à réseaux logiques programmables.
Ce domaine de l'électronique est aussi celui qui a certainement vu la plus forte évolution technologique ces dernières années :

1963 : premiers circuits TTL (10 portes logiques, 50 transistors)

1997 : technologie CMOS 0.25µm (250.000 portes logiques, 25.000.000 transistors)

2001 : technologie CMOS 0.15µm (2.000.000 portes logiques).

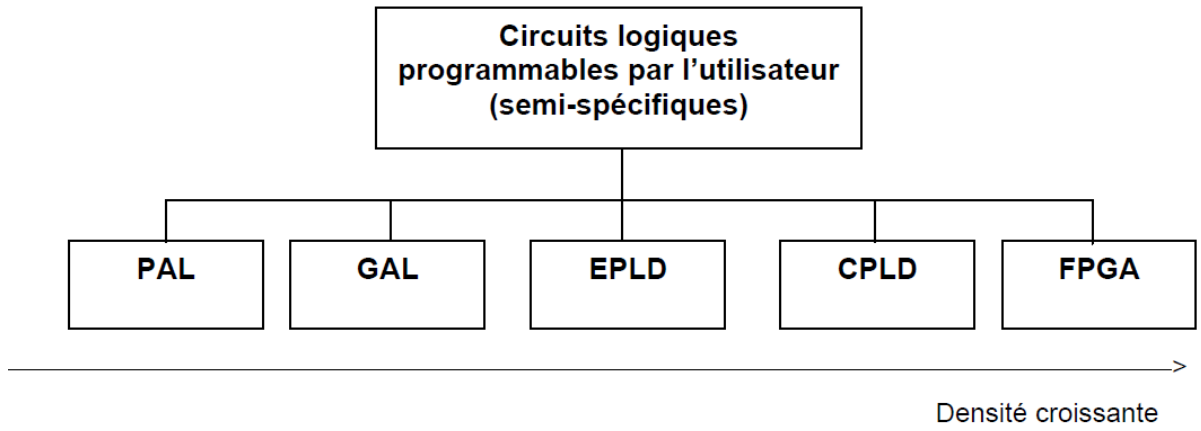


Figure II.1. Circuits à architecture programmable

- Les PALs sont les PLDs les plus anciens (1970). C'est une marque déposée qui est passée dans le langage courant (société MMI). La première génération de PALs était composée de PALs à fusibles programmables une seule fois (OTP :One Time Programmable) qui ne réalisaient que des fonctions combinatoires. Vinrent ensuite les PALs CMOS programmables et effaçables électriquement.

- Les GALs (Generic Array Logic) sont les premières PALs CMOS apparues sur le marché. Le terme GAL étant une marque déposée de la firme Lattice Semiconductor, les autres fabricants de PALs programmables et effaçables électriquement appellent leurs composants PALs CMOS.

- Les EPLDs sont des circuits logiques programmables électriquement et effaçables par rayonnement UV.

- Les CPLDs sont des boîtiers renfermant plusieurs structures de base PAL associées à une matrice d'interconnexion programmable.

- Au milieu des années 80 : apparition des composants de type FPGA (composants à architecture programmable). Ces composants permettent un degré d'intégration très élevé (jusqu'à 100.000 portes). Ces circuits ont amené toute la souplesse de la logique programmable par l'utilisateur et une densité d'intégration jusqu'alors réservée aux ASICs.

Parallèlement à cette évolution, s'est développée une architecture de composants spécifiques de très haute densité d'intégration : Les ASICs (Application Specific Integrated Circuits). Ce sont des composants à applications spécifiques limitées à des domaines de grande diffusion ou exigeant une grande confidentialité. La conception du composant est réalisée par le client. La réalisation finale du composant est confiée au fabricant : on parle de fondeurs.

II.1. Qu'est-ce qu'un circuit programmable ?

Un circuit programmable est un assemblage d'opérateurs logiques combinatoires et de bascules dans lequel la fonction réalisée n'est pas fixée lors de la fabrication. Il contient potentiellement la possibilité de réaliser toute une classe de fonctions, plus ou moins large suivant son architecture. ***La programmation*** du circuit consiste à définir une fonction parmi toutes celles qui sont potentiellement réalisables.

II.2. Codage d'une fonction logique

La base d'une fonction logique, est toujours une fonction combinatoire. Pour obtenir une fonction séquentielle, il suffira ensuite de réinjecter les sorties sur les entrées, ce qui donnera alors un système asynchrone. Si on souhaite un système synchrone, on intercalera avant les sorties, une série de bascules à front, dont l'horloge commune synchronisera toutes les données et évitera bien des aléas de fonctionnement.
Pour coder une fonction combinatoire, trois solutions sont classiquement utilisées.

a) Sommes de produits, produits de somme et matrice PLA (Programmable Logic Array)

N'importe quelle fonction peut être codée par une somme de produit, par un produit de somme ou un mélange des deux. On peut immédiatement en déduire une structure de circuits, appelé matrice PLA (Programmable Logic Array). La figure suivante représente une matrice PLA à 4 entrées et 4 sorties :

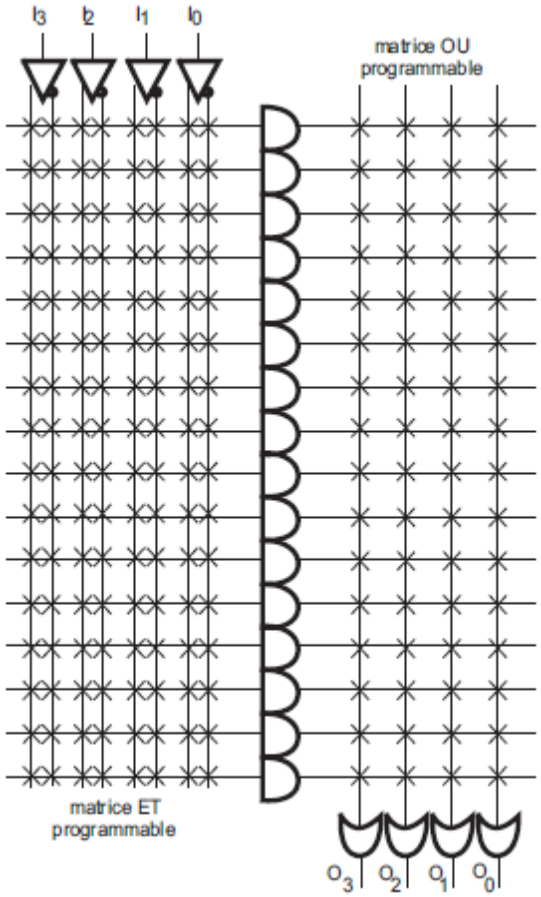


Figure II.2. Schéma d'une PLA à 4 entrées et 4 sorties

Chacune des 4 entrées et son complémentaire arrive sur une des 16 portes ET à 2x4=8 entrées. Afin de simplifier la représentation, les 8 lignes ont été représentées par une seule, chaque croix représentant une connexion programmable (un fusible par exemple). La figure suivante propose un principe de réalisation des fonctions de la matrice ET; la mise au niveau logique 0 (NL0) d'une des entrées I_X impose un NL0 en sortie :

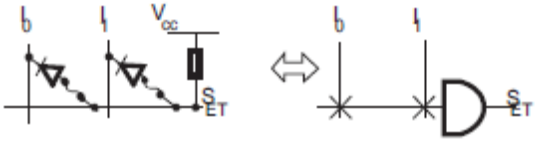


Figure II.3. Programmation de la matrice ET

et de la matrice OU, sur laquelle une entrée au NL1 impose un NL1 en sortie :

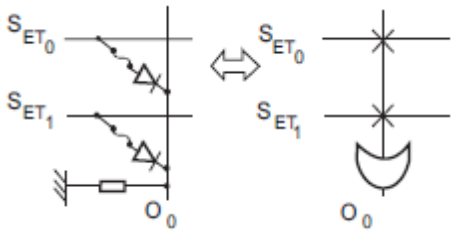


Figure II.4. Programmation de la matrice OU

Ce type de structure demande une densité d'intégration importante: en effet pour n variables en entrées, il faut 2^n fonctions ET à 2^n entrées et au moins un OU à 2^n entrées (il y a en effet 2^n

combinaisons possibles, chaque combinaison dépendant de l'entrée et de son complémentaire).
La plupart des applications n'exigent pas une telle complexité et on peut se contenter d'une matrice ET programmable et d'une matrice OU figée. De même, il est peu probable d'utiliser tous les termes produits et on peut alors limiter le nombre d'entrées de la fonction OU. C'est le principe utilisé par les circuits programmables, appelés au début PAL (Programmable Array Logic), mais plus communément désignés aujourd'hui sous le terme PLD (Programmable Logic Device).

b) Mémoires

Une fonction combinatoire associe à chacune de ces combinaisons d'entrée une valeur en sortie décrite par sa table de vérité. C'est le principe de la mémoire où pour chaque adresse en entrée, on associe une valeur en sortie, sur un ou plusieurs bits. La structure physique des mémoires fait appelle à une matrice PLA dont la matrice ET est figée et sert de décodeur d'adresse et dont la matrice OU est programmée en fonction de la sortie désirée.

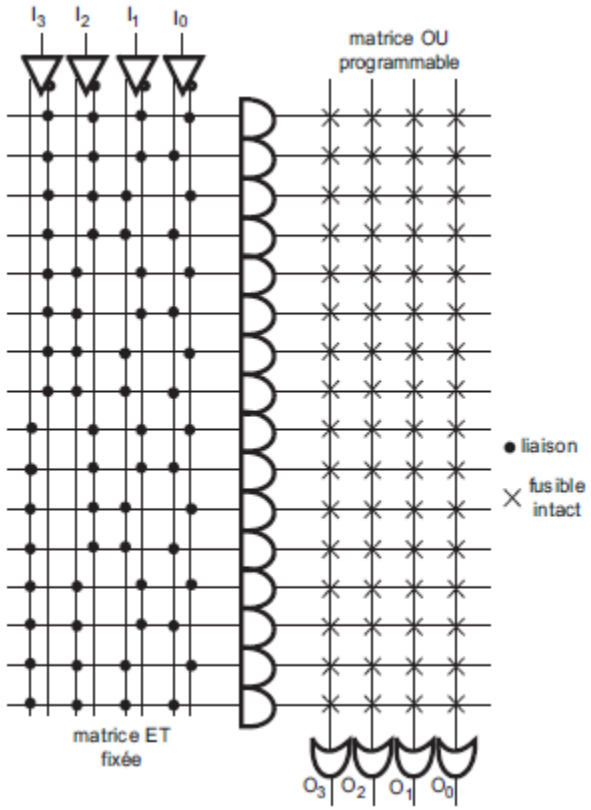


Figure II.5 Schéma d'une PROM

Lorsqu'une adresse est présentée, par exemple $I_3\ I_2\ I_1\ I_0 = 1111$, la porte ET concernée passe au NL1 (celle du bas dans l'exemple) et suivant les fusibles laissés intacts sur la matrice OU, on a un mot différent en $O_3\ O_2\ O_1\ O_0$ (0000 si tous les fusibles sont "grillés" par exemple).
Ce principe utilisé pour les mémoires, l'est aussi dans les CPLDs (Complex Programmable Logic Devices), mais surtout dans les FPGAs (Field Programmable Gate Arrays) sous le nom de LUT (Look Up Table).

c) Multiplexeur

Le multiplexeur permet également de coder une fonction combinatoire, comme le montre la figure suivante :

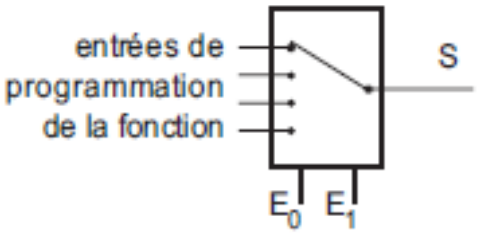


Figure II.6. Principe de fonctionnement d'un multiplexeur

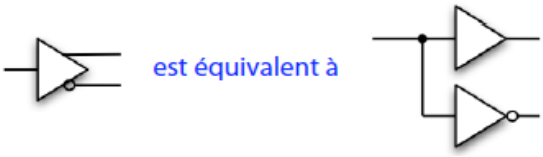
A chaque valeur des entrées E0 et E1 est associé un niveau logique défini dans la table de vérité pour la sortie S. Ce niveau logique est imposé sur l'entrée de programmation correspondante. Ce principe est utilisé dans les FPGA.

II.3. Structure des PLDs simples

Les PLD simples comportent les blocs suivants :

1. Le bloc d'entrée

Le bloc d'entrée permet de fournir au bloc combinatoire l'état de chaque entrée et de son complément.



2. Le Bloc combinatoire

Etant donné que toute fonction logique peut être exprimée comme une somme logique de produits, les circuits logiques programmables les plus communs sont formés par une matrice AND suivie d'une matrice OR. L'une des deux matrices, ou les deux, est *programmable.*

type de dispositif	réseau ET	réseau OU
ROM	fixe (tous les mintermes sont générés par un décodeur)	programmable
PLA	programmable (un nombre limité de mintermes peuvent être générés)	programmable
PAL	programmable (un nombre limité de mintermes peuvent être générés)	fixe (un nombre limité de mintermes peuvent être combinés)
GAL	comme PAL	comme PAL

Tableau II.1. Types de programmation des PLDs

3. Le bloc de sortie (Macro-cellule) et le bloc d'entrée-sortie

Le bloc de sortie est souvent appelé macro-cellule que l'on nomme OLMC (Output Logic Macro Cell qui signifie macro-cellule logique de sortie). Les différentes configurations de la macro-cellule et du bloc d'entrée-sortie associé donnent aux PLDs toute leur souplesse d'exploitation.

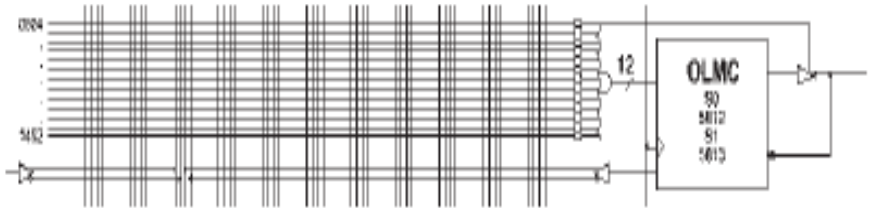


Figure II.7. La macro-cellule d'un PLD

La partie nommée OLMC (Output Logic MacroCell, dénomination Lattice) sur la figure peut être:
• combinatoire, une simple connexion relie alors la sortie du OU à l'entrée du buffer de sortie, dont la sortie est réinjectée sur le réseau programmable ;
• séquentielle, le bloc OLMC étant alors une simple bascule D ;
• versatile, il est alors possible par programmation de choisir entre les deux configurations précédentes.

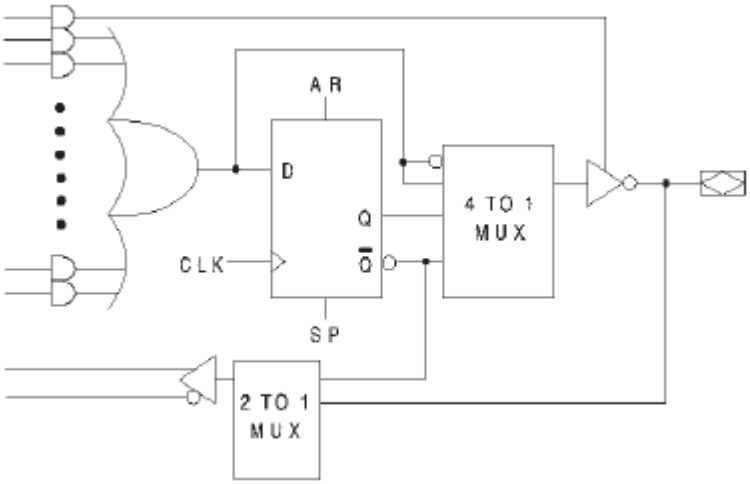


Figure II.8 : OLMC de type versatile

Le multiplexeur 4 vers 1 permet de mettre en circuit ou non la bascule D, en inversant ou pas les signaux. Le multiplexeur 2 vers 1 permet de réinjecter soit la sortie, soit l'entrée du buffer de sortie vers le réseau programmable. Le bloc d'entrée-sortie comporte une porte 3 états et une broche d'entrée-sortie.

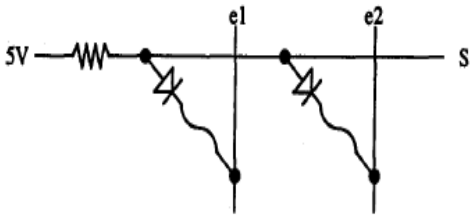
II.4. Technologies d'interconnexions

Il existe plusieurs technologies pour les dispositifs programmables. Passons en revue quelques technologies classiquement utilisées et leurs caractéristiques :

1. Connexions programmables une seule fois (OTP : One Time Programming)

a) Cellules à fusible

La technologie originale utilisée pour les premiers dispositifs programmables, les mémoires ROM, étaient des fusibles. Le dispositif inclut un fusible à chaque lien programmable. Le principe du fusible repose sur l'utilisation d'un métal conducteur qui fond et coupe le circuit lorsqu'il est chauffé par un courant électrique. Pour programmer le dispositif, il faut appliquer une tension élevée (typiquement 2 à 3 fois la tension nominale du dispositif) à des pattes choisies. Une fois les fusibles fondus, le circuit est programmé.



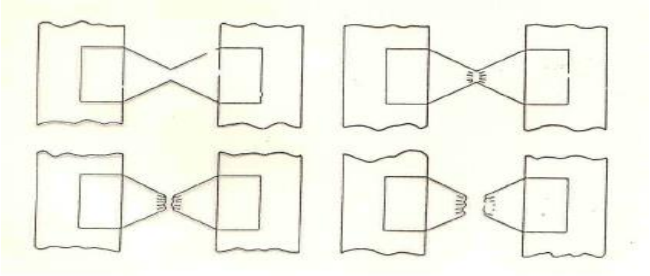


Figure II.9. Cellules d'interconnexion à fusibles

Les principaux inconvénients du fusible sont :

- On ne peut programmer le dispositif qu'une seule fois,
- La surface occupée par ces cellules qui est importante ($700\mu m^2$),
- Le matériau utilisé pour la zone fusible (mélange de titane et de tungstène qui doit pouvoir être facilement déposé lors du processus de fabrication et avoir les qualités de fusion requise),
- La pollution engendrée par le claquage du fusible (résidus),
- Et enfin la technologie utilisée dans ce type de circuit est une technologie à transistors bipolaires et donc gourmande en puissance.

Cette technologie a disparu au profit de technologies plus performantes.

b) Cellule à antifusible :

Depuis le milieu des années 80, la technique des antifusibles est utilisée. Il s'agit du contraire du fusible, c'est-à-dire que par défaut, les connexions n'existent pas dans le composant non programmé. La programmation s'opère par claquage d'un diélectrique disposé entre deux couches de silicium dopées. La géométrie d'une telle cellule est donnée à la figure suivante :

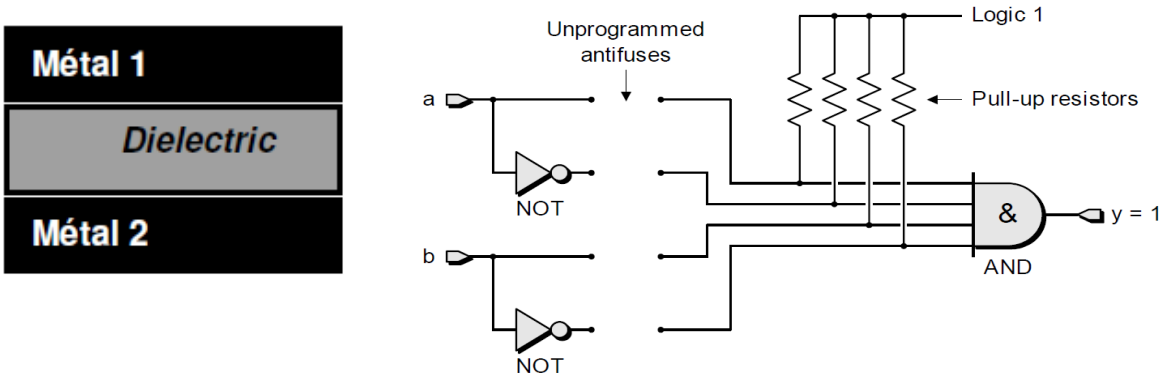


Figure II.10. Cellules à antifusibles

Cette technologie a plusieurs avantages :

- Le matériau utilisé est déjà présent dans les process de composants CMOS standards.
- Et la surface réduite occupée par cette cellule (1.8 μm^2).

La programmation consiste donc à claquer l'isolant (nitrure de silicium de 90Δ d'épaisseur) par application d'une tension de 16 V pendant une durée de l'ordre de la ms.

2. Connexions programmables et reprogrammables :

a) Cellule à transistor MOS à grille flottante et EPROM (Erasable Programmable Read Only Memory)

L'apparition du transistor MOS à grille flottante a permis de rendre le composant bloqué ou passant sans application permanente d'une tension de commande. Le principe consiste à piéger ou non (à l'aide d'une tension supérieure à la tension habituelle d'alimentation) des électrons dans la grille. L'extraction éventuelle des électrons piégés permet le retour à l'état initial. Plusieurs technologies EPROM sont en concurrence :

- ***UV-EPROM :***

Les connexions sont réinitialisables par une exposition à un rayonnement ultraviolet d'une vingtaine de minutes, une fenêtre étant prévue sur le composant. L'effacement, dont la mise en oeuvre est lourde, n'est pas sélectif et ne peut se faire sur site.

- ***EEPROM (Electrically EPROM) :***

L'effacement et la programmation se font cette fois électriquement avec une tension de 12 V et peuvent se faire de manière sélective (la reprogrammation de tout le composant n'est pas nécessaire). Une cellule demande toutefois 5 transistors pour sa réalisation, ce qui conduit à une surface importante (75 à 100 µm2 en CMOS 0,6 µm) et réduit la densité d'intégration possible. La programmation ou l'effacement d'une cellule dure quelques millisecondes.

- ***Flash EPROM***

L'utilisation de deux transistors par cellule uniquement (5 pour l'EEPROM) et une structure verticale permettent une densité d'intégration importante (25 µm2 par cellule en CMOS 0,6 µm) trois à quatre fois plus importante que l'EEPROM, mais quand même 10 fois moins que la technologie à antifusible. La tension de programmation et d'effacement

est de 12 V, avec un temps de programmation de quelques dizaines de µs pour un temps d'effacement de quelques millisecondes.

b) Cellules SRAM à transistors MOS classiques

Ce principe est classiquement choisi pour les FPGA. Le fait d'utiliser une mémoire de type RAM (donc volatile) impose la recharge de la configuration à chaque mise sous tension : une PROM série mémorise généralement les données.
Ce qui peut paraître un inconvénient devient un avantage si on considère l'aspect évolutif du système qui peut s'adapter à un environnement extérieur changeant et modifier sa configuration en fonction des besoins. On pourra d'autre part facilement intégrer de la mémoire RAM dans le circuit.
Le choix d'une cellule SRAM (Static Read Only Mémory) à 6 transistors permet de bénéficier d'un accès sélectif et rapide (quelques ns) en cours d'utilisation. La taille d'une cellule n'est que deux fois plus forte (50 µm2 par cellule) qu'avec une flash EEPROM. Cette technologie, utilisée pour les autres circuits VLSI (contrairement aux EEPROM et Flash EPROM et leurs transistors à grille flottante) permet de bénéficier directement des progrès importants réalisés dans ce domaine.

II.5. Architectures utilisées

1. Structure des CPLD

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. La Figure suivante illustre un CPLD de Xilinx.

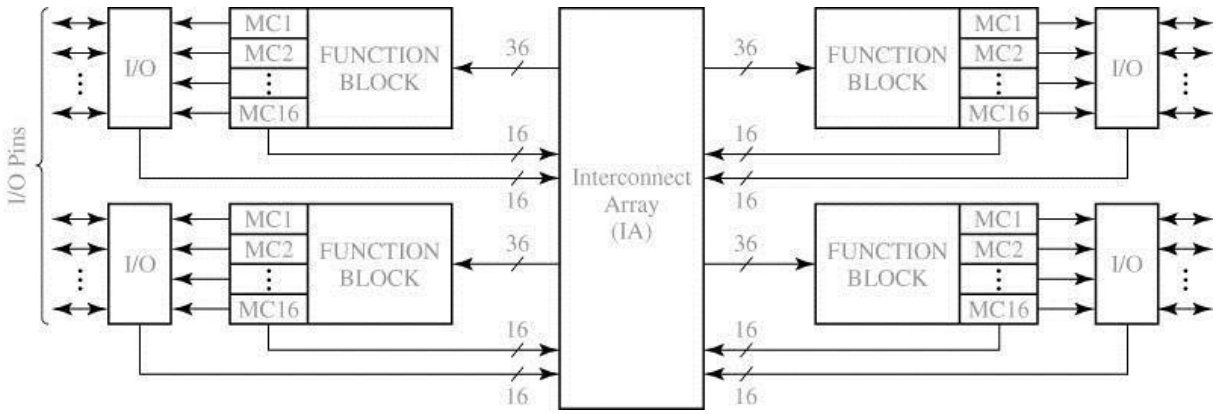


Figure II.11. Circuit logique programmable complexe

Contrairement aux PLD simples,

- La bascule de la macro-cellule d'un CPLD peut être configurée en bascule de type D, T, JK ou RS,
- L'horloge de commande de chaque bascule peut être globale ou individuelle :
    - Lorsqu'elle est globale, c'est une entée dédiée qui sert d'horloge. Certaines CPLD proposent plusieurs broches d'entrée pouvant servir d'horloges globales. On dit que l'horloge est synchrone
    - Lorsqu'elle est individuelle, un terme produit provenant de la matrice ET est utilisé pour commander l'entrée d'horloge. On dit que l'horloge est asynchrone.

Un CPLD est constitué :

- D'un bloc des entrées dédiées, de plusieurs blocs d'entrée-sortie,
- De plusieurs blocs de macro-cellules,
- D'un réseau d'interconnexion entre les blocs de macro-cellules.

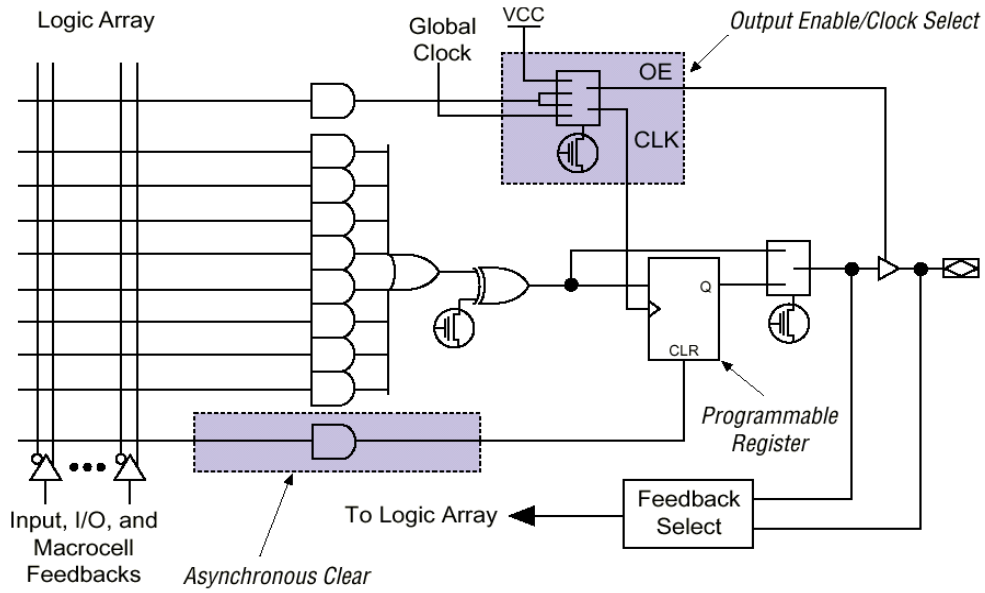


Figure II.12. Structure de la macro-cellule

Un bloc de macro-cellules comporte :

- Plusieurs macro-cellules, on compte généralement au moins 8 macro-cellules,
- Des blocs d'entrée-sortie. La tendance est à la réduction du nombre de broches d'entrée dédiée au profit de celles d'entrée-sortie,
- Une matrice ET pour toutes les macro-cellules du bloc,
- Un module d'optimisation du nombre de termes produit utilisés.

Un bloc de macro-cellule est appelé :

- Soit LAB (Logic Array Block),
- Soit FB (Functional Block)

2. Réseaux pré-diffusés programmables (FPGA)

Les FPGA se sont imposés sur le marché des circuits logiques depuis la fin des années 1980 sous l'initiative de l'entreprise Xilinx. 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. Ce sont des composants électroniques entièrement reconfigurables pouvant être reprogrammés afin d' accélérer certaines phases de calculs.

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; et,
- des blocs spéciaux d'entrée et de sortie avec le monde extérieur (*Input/Output Block – IOB*).

La Figure ci-dessous illustre un modèle de FPGA.

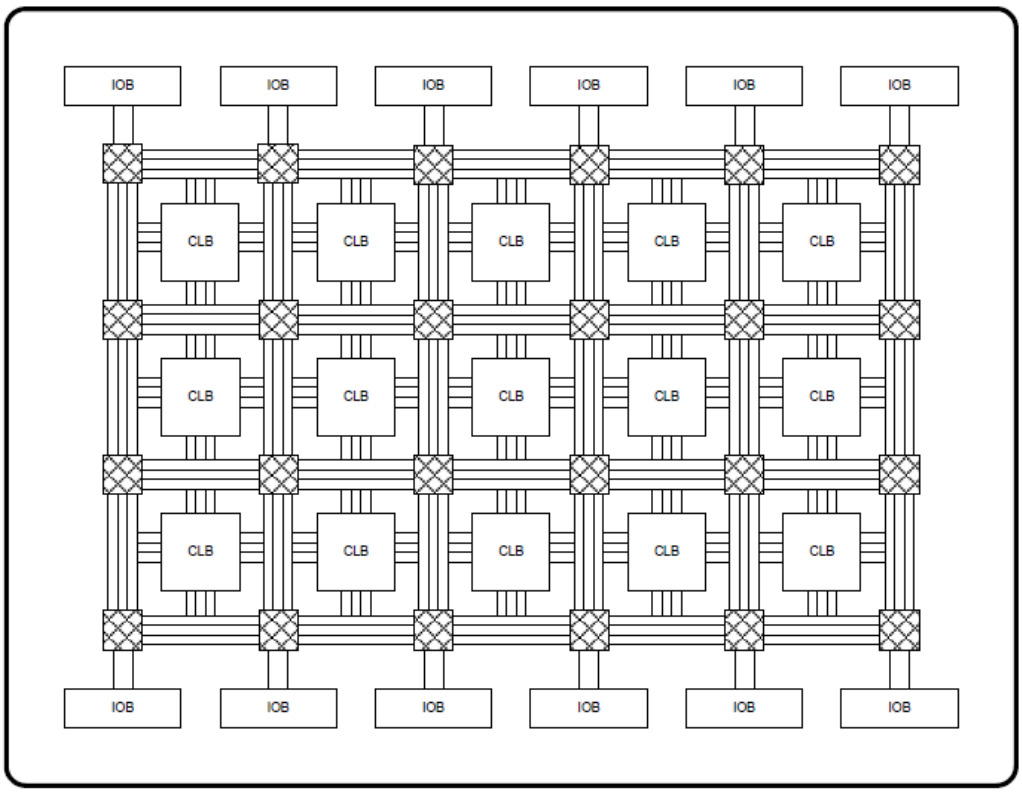


Figure II.13 : Architecture générale d'un FPGA

a. 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, comme en retrouve dans les processeurs.
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 design 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 $F(x, y, z) = \Sigma(0,5,6,7)$. 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 suivante :

#	*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		

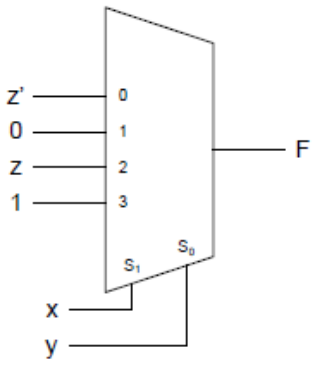


Figure II.14 : Codage d'une fonction par Multiplexeur

Les CLBs basés sur les tables de conversion (Look up Table ; LUT) 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.

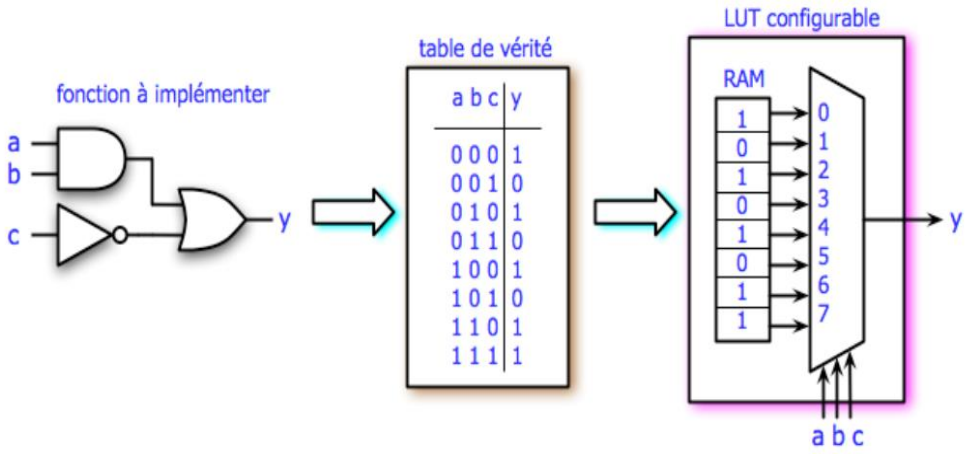


Figure II.15 : Table de conversion (LUT : Look Up Table)

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.

La Figure suivante illustre un CLB simplifié d'un FPGA de Xilinx. 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 verrous (latch).

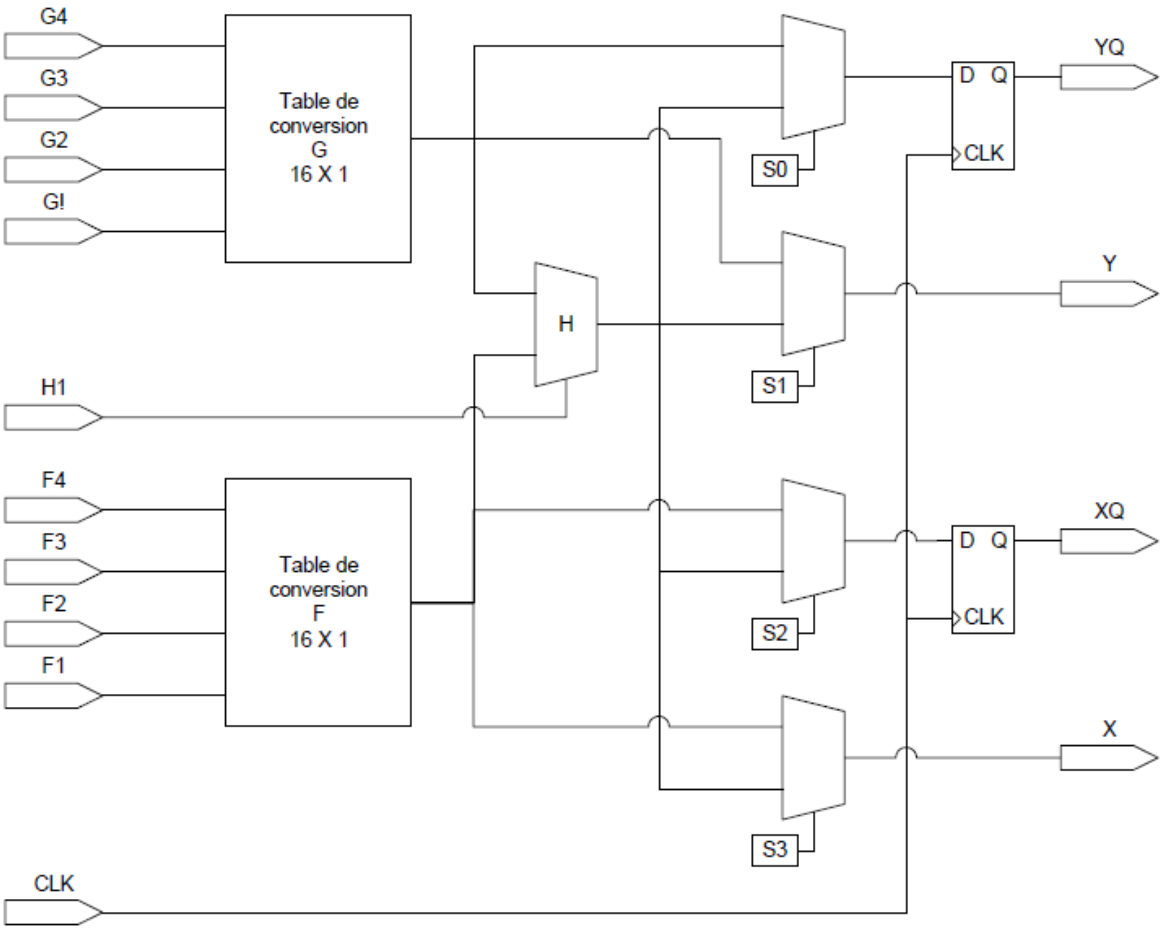


Figure II.16 : Bloc de Logique programmable simplifié – Xilinx

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.

b. 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* re-groupe 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 in-cluant deux tables de conversion, deux additionneurs et deux registres. Un *Configurable Logic Block – CLB* regroupe deux ou quatre *slices*, selon la famille de FPGA.

Remarque :

Un circuit logique programmable (CPLD ou FPGA) ne doit pas être confondu avec un micro-contrôleur. Le micro-contrôleur est une structure logique compliquée, **figée**, qui exécute des instructions rangées dans une ROM ou une RAM, les unes après les autres. L'utilisateur, via des langages de haut niveau (C, Pascal, JAVA…) ou de bas niveau (Assembleur), ne fait rien d'autre que remplir la ROM d'instructions élémentaires que le processeur exécute une à une.

I. Présentation du langage VHDL

I.1. Introduction

Le « **VHDL »** ou le Vhsic Hardware Description Langage (Vhsic = Very High Speed Integrated Circuit) a été développé dans les années 80 aux états-Unis. Ce langage de description est ensuite devenu une norme IEEE numéro 1076 en 1987. Révisée en 1993 pour supprimer quelques ambiguïtés et améliorer la portabilité du langage, cette norme est vite devenue un standard en matière d'outils de description de fonctions logiques. A ce jour, on utilise le langage VHDL pour :

- Concevoir des ASIC,
- Programmer des composants programmables du type PLD, CPLD et FPGA,
- Concevoir des modèles de simulations numériques ou des bancs de tests.

Le VHDL est un langage de programmation dont la vocation est de donner naissance à un circuit logique et non à un programme exécutable. Ce circuit peut se matérialiser dans un CPLD ou plus grand encore, dans un FPGA. Un circuit logique programmable est un ensemble de portes et de bascules élémentaires, intégrées dans une même puce, mais **déconnectées les unes des autres**. L'utilisateur, via un éditeur de schéma, ou un langage de description (Verilog ou VHDL) ne fait rien d'autre que relier des portes ou des bascules nécessaires, en vue d'obtenir **une structure logique souhaitée**.

I.2. Particularité du langage VHDL

La différence majeure entre un langage classique de programmation et le langage VHDL est l'introduction de la notion de « process » et de « signaux ». Une analogie directe est d'assimiler un signal à une équipotentielle d'un circuit électronique et comparer un process à un sous-programme d'interruption dont le signal d'interruption est un évènement survenu sur l'un des signaux déclarés dans la liste de sensibilité du process. Il est utilisé lorsque l'on a besoin de faire une exécution séquentielle dans une architecture.

```
--Déclaration de process
Nom du process : process (liste de sensibilité)
Begin
--Corps du process
End process nom du process
```

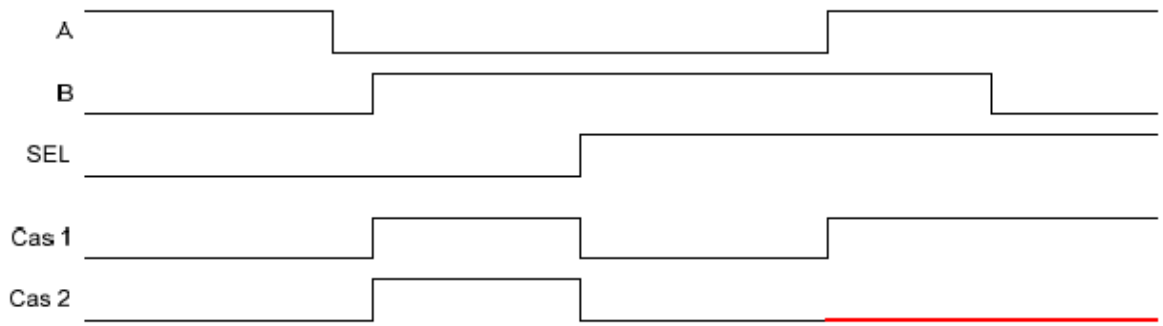
Exemple :

```
Entity MUX is
Port (
A, B, SEL : in std_logic;
OP : out std_logic
);
End MUX;

Architecture TEST of MUX is
Begin
MUXPROCESS : process (A, B, SEL)
Begin
        If (SEL = '1') then
            OP <= A;
        Else
            OP <= B;
        End if;
End process MUXPROCESS;

End TEST;
```

Un process peut être défini par un label (facultatif) et une liste de sensibilité (ce qui se trouve entre parenthèses). Lorsque l'un des signaux présents dans la liste de sensibilité va changer, le contenu du process est alors exécuté. Si l'on oublie de mettre un signal dans la liste de sensibilité, le résultat peut être complètement différent.

Dans le cas 1, on a la sortie OP avec la liste de sensibilité (A, B, SEL)

Dans le cas 2, on a la sortie OP avec une liste de sensibilité sans le signal A.



D'une manière générale, on met dans la liste de sensibilité tous les signaux qui seront scrutés dans le process.

- ***Règles d'écriture en VHDL***

En général, les instructions se terminent par « ; »

- ***Règles de dénominations (Les identificateurs) :*** Les noms choisis par le programmeur doivent respecter les règles suivantes :

- Ils sont constitués de caractères alphabétiques (26 lettres), numériques (10 chiffres) et du caractère souligné '_',
- Le premier caractère est une lettre,

- Il ne peut y avoir 2 »_ » de suite,
- L'identificateur ne peut se terminer par « _ »,
- Un identificateur ne peut pas être l'un des mots réservés du langage.

Les mots réservés de VHDL (selon la norme 1076-2002) sont :

abs	component	generic	next	record	use
access	configuration	guarded	nor	register	
after	constant		not	rem	variable
alias		if	null	report	
all	disconnect	in		return	wait
and	downto	inout	of		when
architecture		is	on	select	while
array	else		open	severity	with
assert	elsif	label	or	signal	
attribute	end	library	others	subtype	xor
	entity	linkage	out		
begin	exit	loop		then	
block			package	to	
body	file	map	port	transport	
buffer	for	mod	procedure	type	
bus	function		process		
		nand		units	
case	generate	new	range	until	

Les commentaires commencent par « -- » et s'arrêtent à la fin de la ligne.

Les principaux objets manipulés

Le langage VHDL exploite 4 objets différents.

1. ***Les constantes***

Une constante peut être assimilée à un signal interne (au circuit) auquel est associée une valeur fixe et définitive. La constante peut être de tout type.

2. ***Les ports***

Les ports sont les signaux d'entrées et de sorties de la fonction décrite ; ils représentent en quelques sortes les signaux des broches du composant VHDL en cours de description.

3. ***Les signaux***

En VHDL, le signal est une information qui existe physiquement dans le circuit. Il sert à modéliser les informations qui passent sur les fils, les bus ou, d'une manière générale, qui transitent entre les différents composants. En quelques sortes, les ports font la liaison avec l'extérieur et les signaux sont internes à la fonction décrite.

4. ***Les variables***

Une variable est capable de retenir une valeur pendant une durée limitée. Elle ne peut être employée qu'à l'intérieur d'un process. A l'opposé d'un signal, une variable n'est pas une liaison concrète et ne doit pas laisser de trace après synthèse.

NB) Pour assigner une valeur à un objet de la catégorie **variable**, on utilise le symbole ' **:=**'. Le symbole '<=' est réservé aux objets de la catégorie **signal**.

- ***Les types de données***

Le langage VHDL est un langage fortement typé, tout objet utilisé dans une description doit au préalable être déclaré et associé à un type. Cette contrainte autorise des vérifications à la compilation sur la compatibilité des objets mis en relation dans des expressions de test ou d'affectation. Un type définit l'ensemble des valeurs que peut prendre un objet ainsi que l'ensemble des opérations disponibles sur cet objet. Puisque VHDL s'applique au domaine logique, les valeurs '0' et '1' peuvent être considérées. L'ensemble de ces deux valeurs définit le type BIT. Il est possible de définir d'autres valeurs comme par exemple la valeur 'Z' désignant la valeur trois état (Haute impédance). Un type plus complet, englobant 9 valeurs logiques différentes décrivant tous les états d'un signal électronique numérique a été standardisé dans la librairie std_logic_1164 : le *std_logic* (Tableau).

Il existe 4 familles de type en VHDL :

- les types scalaires, dont la valeur est composée d'un seul élément (*integer*, *bit*, *std_logic*, *boolean*, etc.),
- les types composés, dont la valeur comprend plusieurs éléments (*bit_vector*, *std_logic_vector*),
- les types accès, qui sont les pointeurs des langages de programmation,
- les types fichiers, qui ne sont utilisés que pour les objets fichiers.

Les types les plus utilisés pour la description de systèmes numériques sont résumés dans le tableau suivant :

catégorie	type ou sous-type	source de la définition	valeurs
scalaires	`boolean`	type prédéfini	FALSE et TRUE
	`bit`	type prédéfini	'0' et '1'
	`character`	type prédéfini	256 caractères de la norme ISO 8859-1, avec des abréviations reconnues et certaines qui sont propres à VHDL Les 128 premiers sont les caractères ASCII.
	`integer`	type prédéfini	plage minimale de $-2^{31} + 1$ à $2^{31} - 1$
	`natural`	sous-type prédéfini	0 à $2^{31} - 1$
	`positive`	sous-type prédéfini	1 à $2^{31} - 1$
	`real`	type prédéfini	typiquement –1.7014111E±308 à 1.7014111E±308
	`std_logic`	Package `std_logic_1164`	'U' : valeur inconnue, pas initialisée 'X' : valeur inconnue forcée '0' : 0 forcé '1' : 1 forcé 'Z' : haute impédance (pas connecté) 'W' : inconnu faible 'L' : 0 faible 'H' : 1 faible '-' : peu importe (*don't care*)
composés	`bit_vector`	type prédéfini	tableau de `bit`
	`string`	type prédéfini	tableau de `character`
	`std_logic_vector`	Package `std_logic_1164`	tableau de `std_logic`
	`unsigned`	Package `numeric_std`	tableau de `std_logic`, interprété comme un nombre binaire non signé
	`signed`	Package `numeric_std`	tableau de `std_logic`, interprété comme un nombre binaire signé en complément à deux

Tableau 2-3 - types les plus communément utilisés pour la synthèse de circuits numériques

- ***Les Opérateurs***

Les opérateurs prédéfinis de VHDL sont décrits au tableau suivant :

catégorie	opérateurs	type de l'opérande de gauche	type de l'opérande de droite	type de l'expression
logique	and, or, nand, nor, xor, xnor, not	bit, boolean		
relation	=, /=, <, <=, >, >=	scalaire ou tableau		boolean
décalage	sll (déc. logique gauche), srl (déc. logique droite), sla (déc. arithmétique gauche), sra (déc. arithmétique droit), rol (rotation gauche), ror (rotation droite)	tableau de bit ou boolean	integer	comme l'opérande de gauche
arithmétique	+, –, *, /, abs (valeur absolue), mod (modulo), rem (reste)	type numérique	type numérique	type numérique
	** (exponentiation)		integer	
concaténation	&	tableau ou type énuméré		tableau

Tableau 2-4 – opérateurs prédéfinis en VHDL

I.3. Structure d'un fichier VHDL

La description d'un système numérique par le biais du langage VHDL passe par 3 étapes différentes :

- La déclaration des ressources externes (bibliothèques) : Cette phase est réalisée automatiquement pour les bibliothèques courantes. On retrouve en en-tête du fichier source les instructions suivantes :

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

- La description de **l'entité** du système, correspondant à la liste des entrées/sorties ;
- La description de l'**architecture** du système, correspondant à la définition des fonctionnalités du système.

L'ensemble est contenu dans un fichier source portant l'extension *.vhd

Les entités de conception (réunion d'une entité et d'une architecture) sont les briques d'une construction VHDL. Elles peuvent représenter un système complet, un sous-système, une carte, un circuit intégré, une porte logique. Elles présentent des ports d'entrée et de sortie bien définis et exécutent une fonction bien définie. Une configuration peut être utilisée pour décrire comment ces entités sont connectées entre elles pour former un système complet. Une

entité de conception est définie par une déclaration d'**entité** associée au corps d'**une architecture**.

1. La déclaration d'entité :

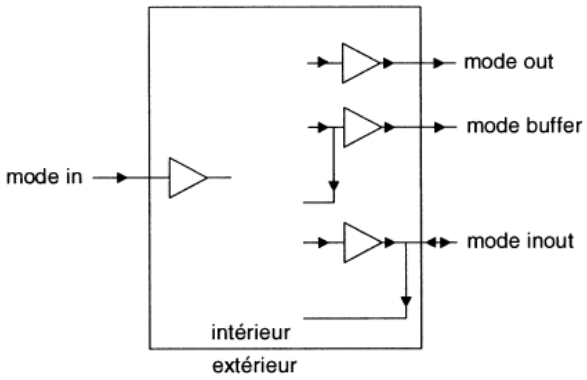
L'entité décrit la vue externe du modèle. Une déclaration d'entité définit les ports d'interface entre une entité de conception et son environnement extérieur. On y retrouve donc un nom d'entité, et la liste des ports, avec leurs caractéristiques (nom, mode, type). Les modes des ports couramment utilisés sont :

in : port d'entrée

out : port de sortie

inout : port d'entrée/sortie

Le mode **buffer** peut être utile, il permet de déclarer un port de sortie dont on peut relier la valeur à l'intérieur de l'unité de conception.



Exemple :

```
-- Déclaration d'entité pour une bascule D
Entity basculeD is
        Port (
                d, h    : in std_logic;
                q       : out std_logic
        ) ;
End entity basculeD;
```

2. Les architectures :

Une fois l'entité définie, il faut décrire son fonctionnement, c'est ce que l'on fait avec l'architecture.

L'architecture est divisée en deux parties : une zone déclarative et une zone d'instructions. La partie déclarative permet de définir des types, des objets (signaux, constantes) locaux au bloc considéré. Elle permet également de déclarer des noms d'objets

externes (composants d'une librairie, par exemple) utilisés dans le corps de l'architecture. Toutes les instructions qui décrivent le corps de l'architecture sont des instructions ***concurrentes*** qui s'exécutent de façon asynchrone les unes par rapport aux autres.

```
architecture nomArchitecture of nomEntité is
[déclarations sous-programmes, types, constantes,
signaux, alias, composants…]
begin
instructions concurrentes
end [architecture] [nomArchitecture];
```

Exemple :

```
-- Description d'une architecture pour une bascule D
architecture rtl of basculeD is
begin
        pBascD : process (h)
        begin
                if (h'event and h = '1') then
                        q       <= d;
                end if;
        end process pBascD;
end architecture rtl;
```

Le corps d'une architecture spécifie les relations entre les entrées et les sorties d'une entité de conception. Cette spécification peut être exprimée sous forme :

- ***Comportementale (behavioral)*** : se présente comme des blocs d'algorithmes séquentiels exécutés par des processus indépendants. Elle peut traduire un fonctionnement séquentiel ou combinatoire du circuit modélisé.
- ***flot de données* (Data flow)** : Les signaux passent à travers des couches d'opérateurs logiques qui décrivent les étapes successives qui font passer des entrées d'un module à sa sortie. Ce niveau de description est souvent appelé « logiques à transfert de registres » ou RTL (Register Transfer Logic).
- ***Structurelle (structural)*** : Utilise des composants supposés exister dans une librairie de travail. Le programme se contente alors d'instancier les composants nécessaires et de décrire leurs interconnexions.

NB) Les trois formes peuvent coexister à l'intérieur d'un même corps d'architecture.

- ***Déclaration d'un composant (Component)***

La déclaration d'un composant déclare les interfaces à une entité de conception virtuelle. Une configuration de composant peut être utilisée pour associer l'instance d'un composant avec une unité de conception présente en librairie.

Exemple :

```
-- Déclaration du composant basculeD
component basculeD is
        Port (
                d       : in std_logic;
                h       : in std_logic ;
                q       : out std_logic
        ) ;
end component  basculeD;
```

- ***Instanciation d'un composant***

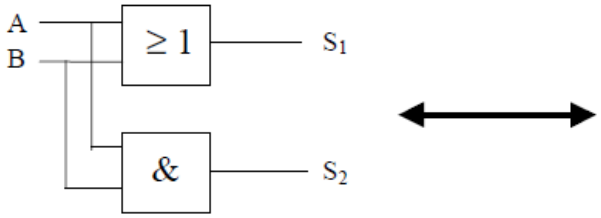
L'instanciation d'un composant dans la description d'une unité de conception définit un sous-composant de cette unité. L'instanciation associe des signaux ou des valeurs aux ports et des valeurs aux paramètres génériques de ce sous-composant.

```
-- instanciation d' un composant par son nom
architecture rtlStructural of regDecal is
signal sig1, sig2 : std_logic;
component basculeD is
        Port (
                h, d    : in std_logic;
                q       : out std_logic
        ) ;
end component  basculeD;
begin
--instantiation bascules 1, 2, 3
basculeD1: basculeD
        port map (h => h, d => d, q => sig1);
basculeD2: basculeD
        port map (h => h, d => sig1, q => sig2);
basculeD3: basculeD
        port map (h => h, d => sig2, q => q);
end architecture rtlStructural;
```

- ***Les Instructions Concurrentes et Séquentielles :***
-

1. *Instructions Concurrentes*

Lorsque l'on étudie une structure logique, il apparaît à l'évidence que les signaux qui la constituent évoluent de façon indépendante et parallèle. Prenons l'exemple suivant :




```
architecture ET_OU of ET_OU is
begin
        S2 <= A and B;
        S1 <= A or B;
end ET_OU;
```

Dans cette exemple, les signaux S_1 et S_2 évoluent de façon indépendante et parallèle. Il est donc impossible de décrire leur évolution de façon séquentielle. Il faut avoir recours à des

instructions de types concurrents qui s'exécutent en parallèle indépendamment de leur ordre d'écriture.

On peut dissocier 3 types d'instructions concurrentes :

- Les instructions d'affectation qui correspondent à l'écriture d'une équation booléennes (logiques),
- Les instructions conditionnelles du type « affectation **When** condition **else**… »,

Exemple :

```
architecture exemple of exemple is
begin
S  <= A when (Sel = "00") else
      B when (Sel = "01") else
      C when (Sel = "10") else
      '0';
end exemple;
```

- Les instructions conditionnelles du type « **With** signal select affectation **When** condition **else**… ».

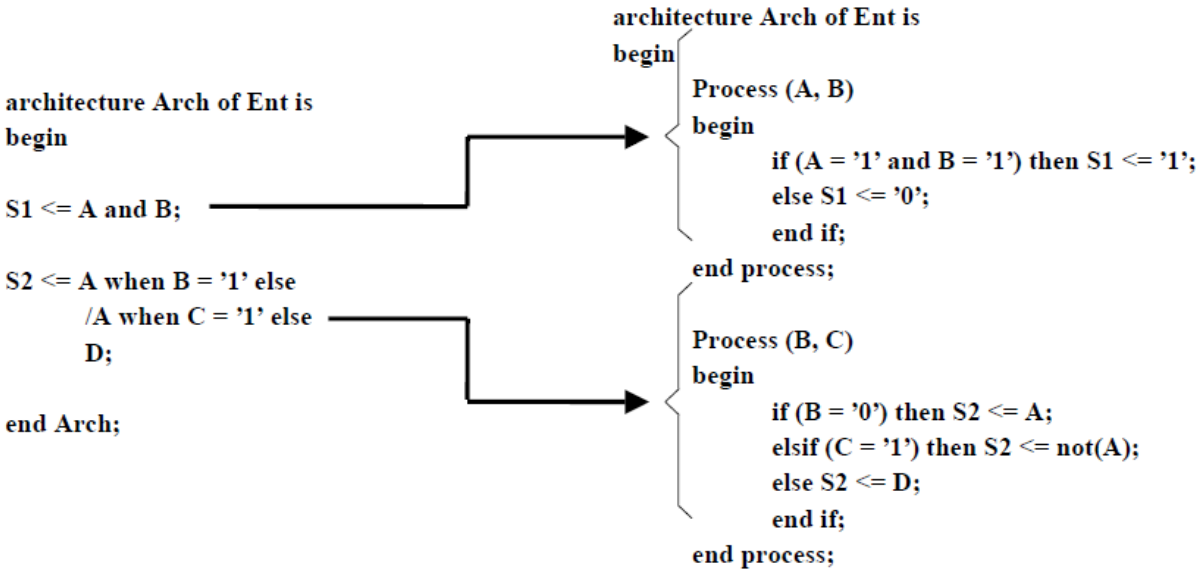
Exemple :

```
architecture exemple of exemple is
begin
With Sel select
S  <= A when "00",
      B when "01",
      C when "10",
      '0' when others;
end exemple;
```

2. Les instructions séquentielles

Par opposition aux instructions concurrentes présentées au paragraphe précédent, le langage VHDL propose un jeu très complet d'instructions séquentielles identiques à celles que l'on trouce dans les langages de programmation évolués (C++, Pascal,…). Mais ces types d'instructions ne peuvent pas être utilisés pour décrire les phénomènes que l'on rencontre dans les montages à base de composants logiques. Pour palier l'incompatibilité qui existe entre les instructions séquentielles et le fonctionnement des montages que l'on souhaite décrire, le langage VHDL propose une solution simple qui consiste à créer des ensembles appelés **process**, regroupant des instructions **séquentielles** et se comportant, d'un point de vue externe, comme des instructions **concurrentes**.

Voici un exemple d'équivalence entre deux descriptions, l'une à base d'instructions concurrentes et l'autre à base d'instructions séquentielles regroupées au sein d'un process :



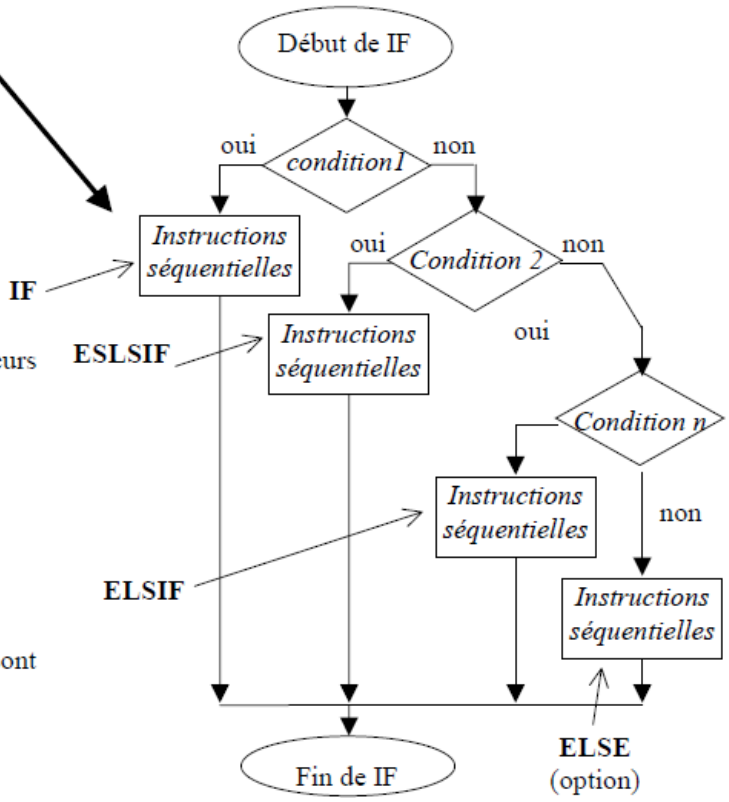
I.4. Les boucles et instructions : if, then, else, for, case…

Les instructions **séquentielles** du type if, then, else… sont très puissantes et permettent la réalisation de boucles conditionnelles intéressantes. De plus, la syntaxe qui leur est associée ne présente pas de difficultés particulières et permet une écriture des descriptions très lisible.

INSTRUCTION IF

IF *condition1* **THEN**
instructions séquentielles;
ELSIF *condition2* **THEN**
instructions séquentielles;
ELSIF *condition3* **THEN**
instructions séquentielles;
⋮
ELSE
instructions séquentielles;
END IF;

SI *condition1* **ALORS**
instructions séquentielles;
SINON SI *condition2* **ALORS**
instructions séquentielles;
SINON SI *condition3* **ALORS**
instructions séquentielles;
⋮
SINON
instructions séquentielles;
FIN DU SI;



Remarques :

❶ Il est possible d'imbriquer plusieurs boucles IF les unes dans les autres.

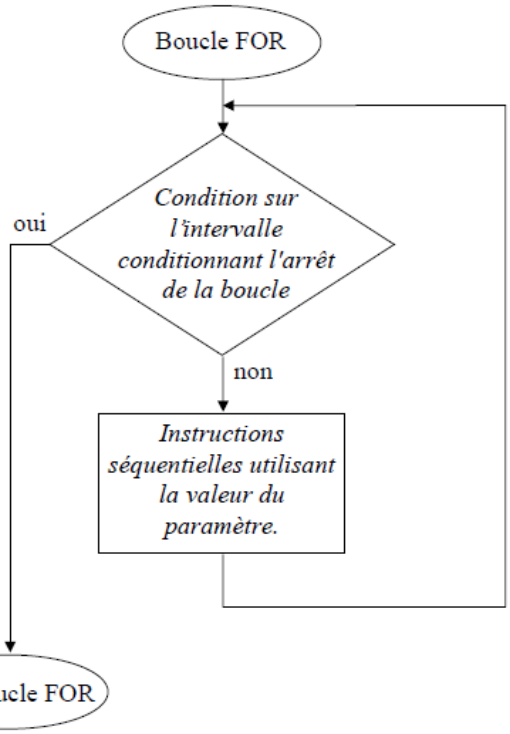
```
IF ... THEN
    IF ... THEN
    ELSIF ... THEN
    END IF;
ELSE
END IF;
```

❷ Les instructions ELSIF et ELSE ne sont pas obligatoires.

BOUCLE FOR

FOR *paramètre* **IN** *intervalle* **LOOP**
instructions séquentielles;
END LOOP;

POUR *le paramètre* **COMPRIS**
dans l'intervalle **EXECUTER**
instructions séquentielles;
FIN DE LA BOUCLE;



Exemple :

```
FOR i IN 0 to 3 LOOP
        IF (A = i) THEN
        S <= B;
        END IF;
END LOOP;
```

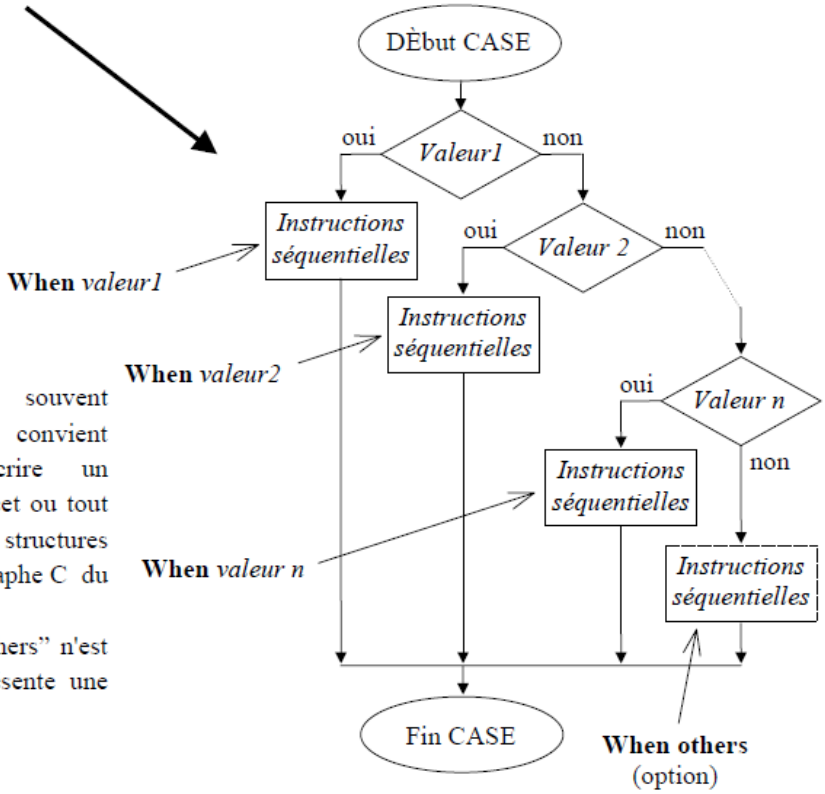
Exécuter, pour i = 0, i = 1, i = 2 puis i = 3
les instructions suivantes :

```
        SI (A = i) THEN
        S <= B;
        END IF;
FIN DE LA BOUCLE;
```

INSTRUCTION CASE

CASE *signal* **IS**
WHEN *valeur1* =>
instructions séquentielles;
WHEN *valeur2* =>
instructions séquentielles;
WHEN *valeur3* =>
instructions séquentielles;
WHEN OTHERS =>
instructions séquentielles;
END CASE;

CAS *possibles de l'expression* **EST**
LORSQUE *signal* = *valeur1* =>
instructions séquentielles;
LORSQUE *signal* = *valeur2* =>
instructions séquentielles;
LORSQUE *signal* = *valeur3* =>
instructions séquentielles;
LORSQUE *signal* = **AUTRES** =>
instructions séquentielles;
FIN DE CAS;



Remarques :

❶ L'instruction CASE, souvent appelée "switch case", convient parfaitement pour décrire un diagramme d'état, un grafcet ou tout autre formalisme de structures séquentielles (voir paragraphe C du chapitre VII).

❷ L'instruction "When others" n'est pas obligatoire, elle représente une facilité d'écriture.

I.5. Différences entre un langage de programmation et VHDL

VHDL n'est pas un langage de programmation comme le C, c'est un langage de description matériel. Il est important de bien comprendre la différence entre les deux :

- Un **langage de programmation** est destiné à être traduit en **langage machine** puis à être exécuté par un **microprocesseur.**
- Un **langage de description matériel** comme VHDL décrit une réalité matérielle, c'est-à-dire le fonctionnement d'un système numérique. Il va être traduit (synthétisé) en un ensemble de **circuits logiques combinatoires et séquentiels** qui vont être implémentés dans un **circuit intégré**. Il n'y a aucun microprocesseur pour exécuter la description VHDL dans un circuit intégré.

Introduction

Le logiciel Xilinx ISE est un outil de conception de circuit pour FPGA de Xilinx. Le logiciel ISE possède une version gratuite et téléchargeable du site de Xilinx (www.xilinx.com). Ce logiciel permet essentiellement d'effectuer les différentes étapes propres à la synthèse de circuits numériques sur FPGA. Il est alors possible d'en faire l'implémentation sur les différentes familles de puces fournies par Xilinx. Cet outil de Xilinx permet de créer des projets comportant plusieurs types de fichiers (HDL, schématique,…), de compiler, d'effectuer des designs rule check (DRC), de créer des contraintes de timings sur les horloges, de déterminer l'emplacement des broches, de créer des bancs d'essai de simulation (testbench) et de gérer efficacement les projets d'envergure.

Une fois le nouveau projet créé dans l'environnement ISE, le fichier VHDL vide décrivant la fonction désirée s'ouvre dans l'espace de travail. Ce fichier contient l'entité du module mais présente une architecture vide à compléter.

TP n°1 : Introduction au logiciel Xilinx ISE 9.2i

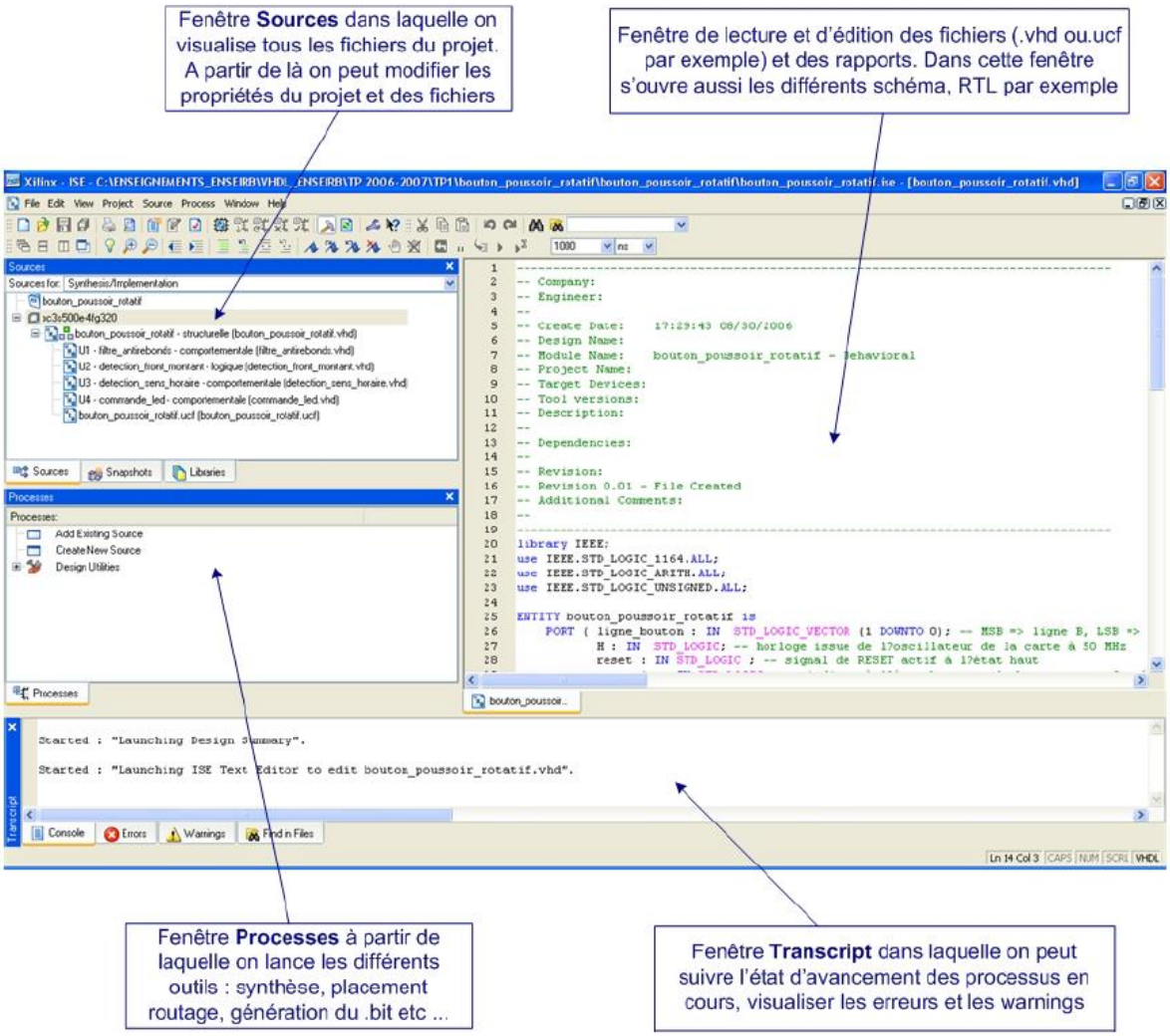
Objectif

Ce TP a pour but de permettre la prise en main des outils de CAO Xilinx sur un exemple simple : les portes combinatoires. Nous allons passer en revue toutes les phases du développement d'un design en VHDL.

Lancement du logiciel Project Navigator :

Pour lancer le navigateur ISE, il faut double-cliquer sur l'icône  , ou cliquer sur le menu **Démarrer > Programmes >Xilinx ISE 9.2i >Project Navigator.**

L'écran principal du navigateur est composé de plusieurs fenêtres qui sont montrées comme suit :

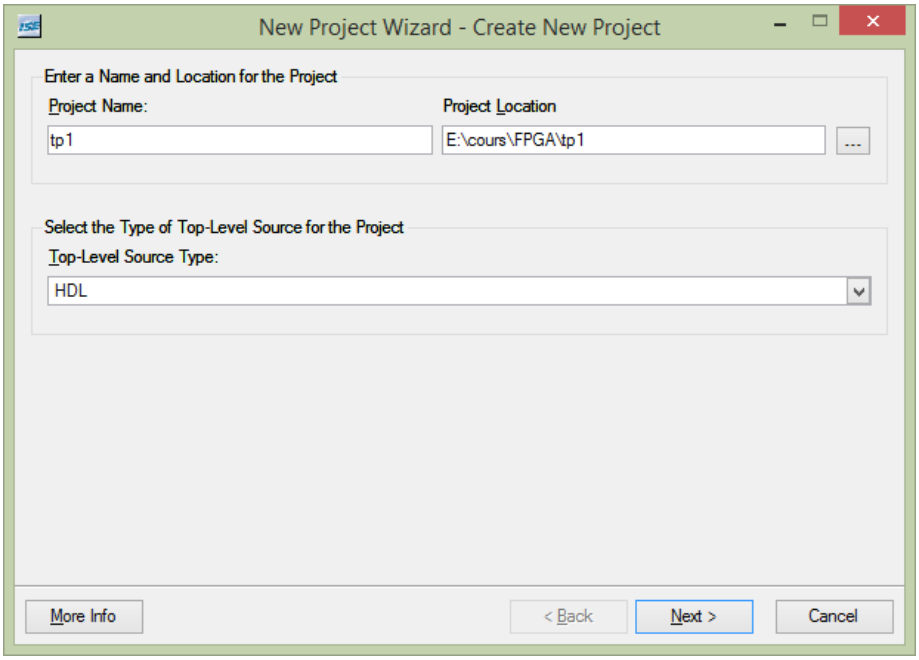


1. *Création du projet*

Cliquer sur le menu « File » et le sous-menu « New Project… ». La fenêtre de création du projet apparaît.

- Dans l'ordre, tapez le nom du projet tp1 dans le champ « Project Name »
- Sélectionnez le type HDL pour le design principal.

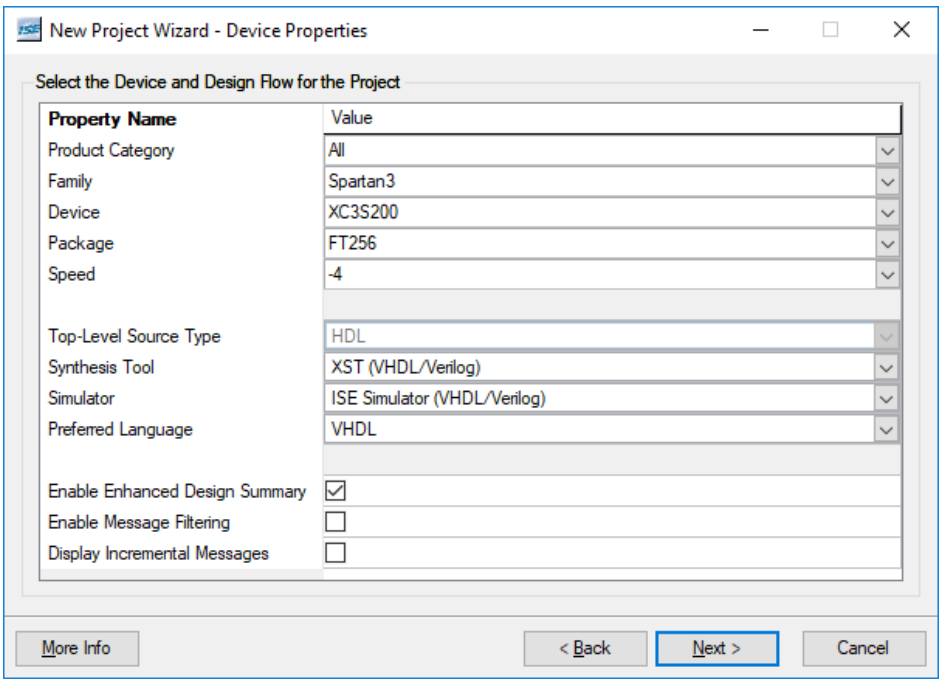
Vous devez finalement obtenir la fenêtre suivante avant de cliquer sur « Next »



Dans la fenêtre qui s'ouvre, sélectionnez :

- La famille de FPGA utilisée (Spartan3 dans le champ « Device Family »
- Le circuit utilisé (xc3s200 dans le champ « Device »)
- Le boîtier (ft256 dans le champ « Package »),
- La vitesse (-4 dans le champ « Speed Grade »),
- Les outils du flot de développement (synthétiseur : XST, simulateur : ISE simulator, langage : VHDL).

Vous devez finalement obtenir la fenêtre suivante avant de cliquer sur « Next »

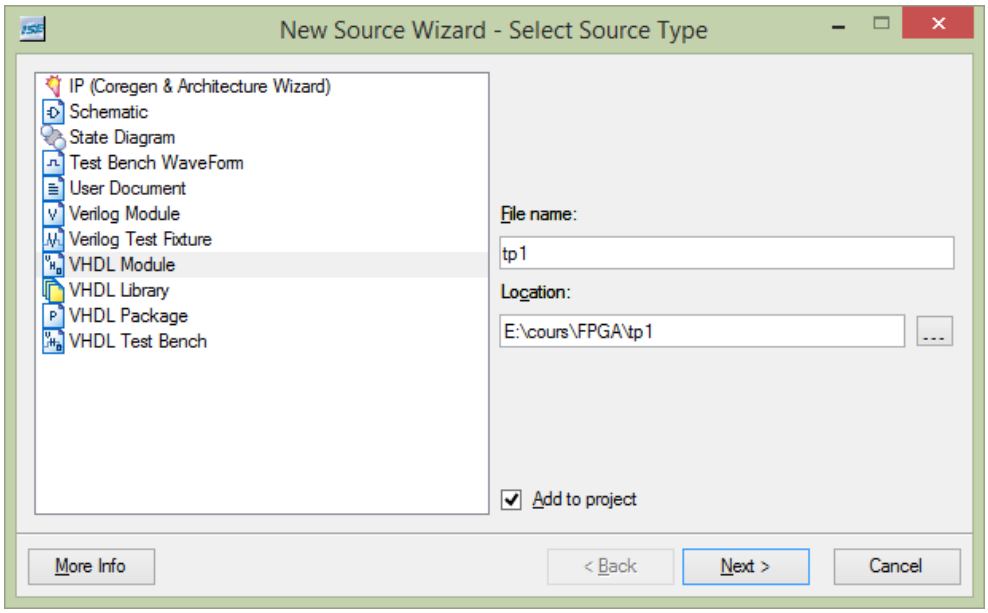


Cliquer sur « Next » dans les deux fenêtres suivantes, puis sur « finish » dans la fenêtre finale.

2. *Création du design VHDL*

Pour lancer l'éditeur, cliquer sur le menu « Project » puis sur le sous-menu « New source… ».

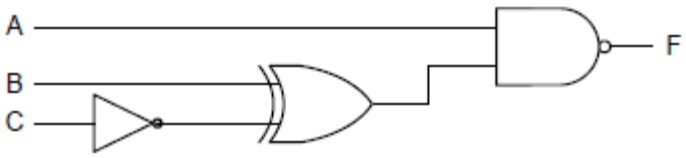
La fenêtre suivante s'ouvre. Sélectionnez « VHDL Module », tapez le nom du design tp1 dans le champ « File Name » puis cliquez sur « Next »



Cliquez sur « Next » dans la fenêtre suivante.

Travail demandé :

1- Réaliser la fonction représentée dans la figure ci-dessous en utilisant les 3 types d'instructions concurrentes (description en flot de données).

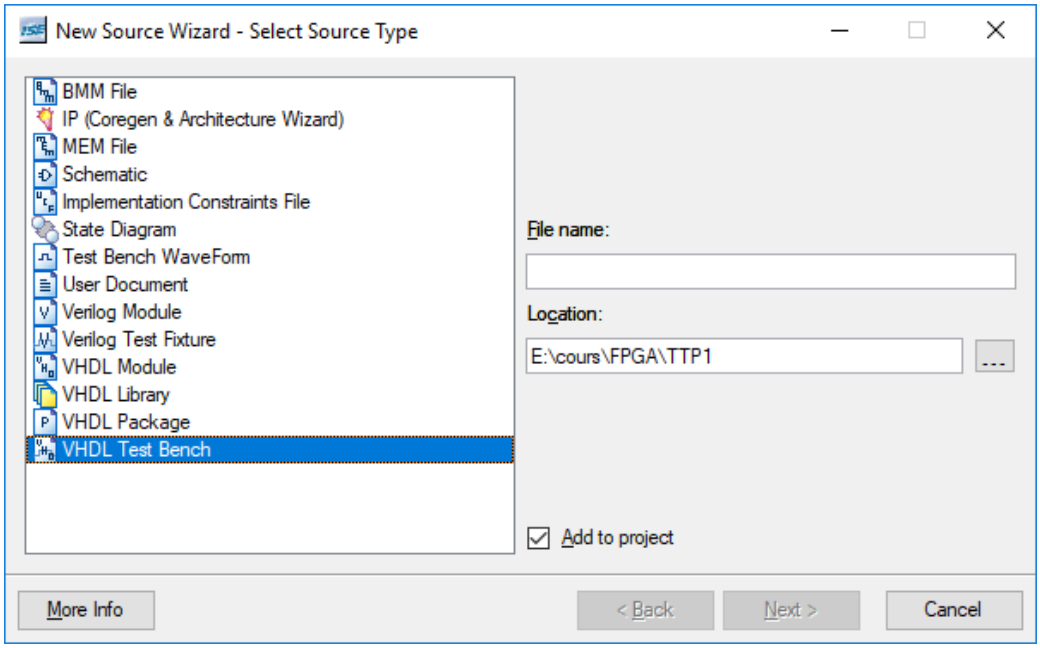


Structure d'un programme VHDL

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

entity mon_circuit is
     port (déclaration des
     entrées/sorties);
end mon_circuit;
```

```
architecture test of mon_circuit is
begin
corps de l'architecture
end test;
```

2. Ecrire un fichier de test (TEST Bench) en langage VHDL de la fonction F, en entrant des valeurs pour les stimuli (les signaux d'entrées) et observer le résultat sur la fenêtre des courbes. (créer une nouvelle source en sélectionnant VHDL Test Bench).



Travaux Pratiques n° 2 : Les Circuits Arithmétiques

But du T.P. :

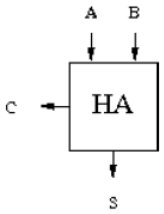
A l'aide de Xilinx ISE 9.2i, nous étudierons :

- les propriétés des circuits arithmétiques et en particulier les additionneurs.
- La programmation en VHDL d'un additionneur parallèle 4 bits en utilisant *la description structurelle* (assemblage de sous-systèmes).

Travail demandé :

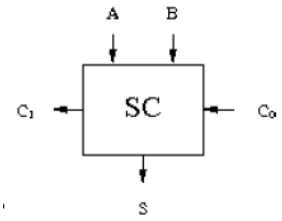
- Définir l'entité et l'architecture décrivant un demi-additionneur 1 bit.

A	B	s	c

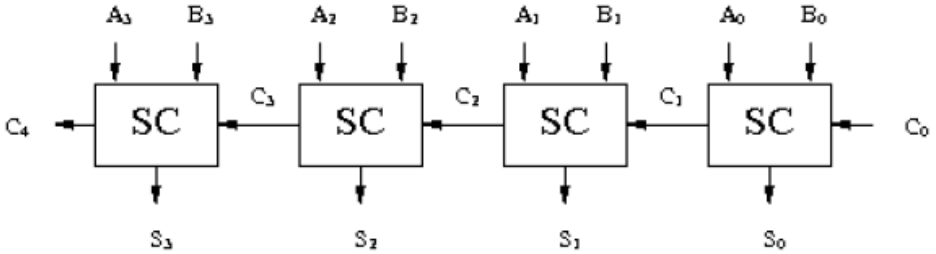


- Définir l'entité et l'architecture d'un additionneur complet 1 bit en utilisant la description en flot de données.

a_i	b_i	c_i	s_i	c_{i+1}



- Réaliser l'additionneur complet 1 bit à l'aide du demi-additionneur réalisé précédemment en utilisant la description structurelle.
- En déduire le programme VHDL d'un additionneur parallèle 4 bits.



Travaux Pratiques n° 3 : Les Bascules

But du T.P. :

A l'aide de Xilinx ISE 9.2i, nous étudierons :

- les propriétés des Bascules asynchrones et synchrones.
- La programmation en VHDL d'une bascule RS et d'une bascule D en utilisant les assignations concurrentes et séquentielles (process).

Modélisation d'éléments à mémoire en VHDL

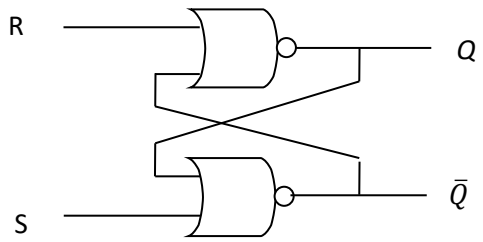
Pour modéliser une bascule, il est nécessaire de pouvoir décrire le fait que le changement d'état se produit sur une *transition* d'un signal d'horloge et non sur sa *valeur*. Pour ce faire, on peut utiliser les attributs d'événement (*event attribute*) définis en VHDL. L'Exemple suivant démontre l'utilisation de l'attribut event sur le signal CLK, dénoté par CLK'event.

Comportement désiré	option 1	option 2
front montant	`CLK'event and CLK = '1'`	`rising_edge(CLK)`
front descendant	`CLK'event and CLK = '0'`	`falling_edge(CLK)`

Cet attribut est vrai quand un changement de valeur se produit sur le signal auquel il est associé. Pour distinguer entre un front montant et un front descendant, on doit de plus spécifier la valeur du signal immédiatement après que l'événement ait eu lieu, soit '1' pour un front montant et '0' pour un front descendant.

<u>Travail demandé</u> :

- Définir l'entité et l'architecture de la bascule RS définie par le montage suivant :



- Ecrire le programme VHDL d'une bascule RS synchrone admettant une entrée horloge H.
- En utilisant les assignations séquentielles (process), écrire le programme d'une bascule D synchrone déclenchée par front montant du signal d'horloge CLK.

Résumé : (*La chaîne complète de conception en VHDL)*

La chaîne de développement complète utilisant l'outil ISE de la société Xilinx comprend les étapes suivantes.

1. Ecriture du modèle VHDL avec l'éditeur intégré du navigateur de projet ; il y a deux parties à écrire :
   - L'entité (entrées/sorties)
   - L'architecture (la description du fonctionnement)
2. Ecriture du testbench. Pour pouvoir simuler le modèle VHDL de notre design, il faut écrire des vecteurs (stimuli) de test qui forment le testbench. Cette écriture se déroule en deux phases :
   - Définition graphique des vecteurs de test,
   - Sauvegarde de la saisie graphique et du fichier équivalent en langage VHDL.
3. Simulation fonctionnelle (simulateur ModelSim). A partir des fichiers design.vhd (modèle VHDL du design) et design_tb.vhw (vecteurs de test en VHDL), nous pouvons effectuer la simulation fonctionnelle du design. Il y a quatre phases :
   - Compilation des fichiers,
   - Lancement du simulateur,
   - Exécution des vecteurs de test,
   - Vérification du fonctionnement à l'aide des chronogrammes.
4. Synthèse logique avec XST. La synthèse est l'opération qui consiste, à partir du fichier texte, à produire la netlist contenant le schéma électrique destiné aux outils de placement-routage (fichier au format NGC). Cette étape est particulièrement délicate dans le cas des FPGAs à cause de la complexité élevée (granularité) de la cellule de base du FPGA.
5. Implémentation
6. Simulation temporelle (avec Model Sim). Après le placement-routage, on peut obtenir un modèle VHDL réel du design (modèle VITAL) ainsi que le fichier de timings qui lui est associé (fichier SDF). On utilise le fichier de stimuli généré par HDL Bencher pour effectuer la simulation de timing du design en 4 phases :
   - Compilation des fichiers,
   - Lancement du simulateur
   - Exécution des vecteurs de test,
   - Vérification du fonctionnement à l'aide des chronogrammes.
7. Téléchargement : une fois le design entièrement simulé, nous pouvons télécharger le fichier de configuration du FPGA dans la maquette grâce au logiciel Impact :
   - Initialisation de la chaîne JTAG,
   - Association du fichier .bit avec le FPGA,
   - Téléchargement dans le FPGA.
8. Vérification sur la maquette FPGA. Un design simulé mais non testé matériellement c'est comme un programme en C que l'on aurait compilé mais pas exécuté.

Bibliographie

[1] J. Weber, M. Meaudre, « Circuits numériques et synthèse logique, un outil : VHDL », Edition Masson, 1995.

[2] A. Nketsa, « Circuits logiques programmables, mémoires,PLD,CPLD et FPGA », Edition ellipses, 1998.

[3] J. Weber, M. Meaudre, « Le langage VHDL, cours et exercices », 2ème Edition Dunod.