République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Hassiba Benbouali de Chlef
Faculté de Technologie
Département d'Electronique

Polycopié de Cours

Domaine : Sciences et Technologies
Niveau : Deuxième Année Master
Filière : Automatique
Spécialité : Automatique et Informatique Industrielle

FPGA et programmation VHDL

Réalisé par :
Dr. BABA-AHMED Mohammed Zakarya
MCA à l'université de Chlef

Année Universitaire 2020-2021

Cours : FPGA et programmation VHDL

Semestre : 1

Unité d'enseignement : UEF 2.1.2

Matière 1 : FPGA et programmation VHDL

Crédits : 4

Coefficient : 2

Public cible : Master 2 Automatique option : Automatique et Informatique Industrielle.

Objectifs de l'enseignement :

Ce module enseigne les différentes technologies des circuits numériques, les méthodologies de conception des circuits à haute densité d'intégration VLSI ainsi que les outils de développement nécessaires à la description matérielle telle que les outils de CAO (Conception Assistée par Ordinateur) et les langages de haut niveau de description matérielle.

Connaissances préalables recommandées :

1. Le codage des nombres.
2. Les circuits combinatoires.
3. Les circuits séquentiels.

Année universitaire : 2017-2021

Table des matières

Avant-propos

Ce manuscrit résulte de Quatre années de cours du module FPGA et programmation VHDL répartie selon le canevas des master 2 Automatique (option : Automatique et informatique Industrielle) en 4 principaux axe à savoir la logique programmable et ses différentes technologies, les circuits FPGA (Field Programmable Gate Array) et ses différents composants, le langage de description matériel VHDL (pour VHSIC Hardware Description Language), et l'implémentation des circuits programmables sur des cartes FPGA.

Sachant que le but de ce cours et d'introduire nos lecteurs à la description et la manipulation des circuits numériques et les traduire en composant matériel en un temps record avec une possibilité de reconfiguration à volonté et des résultats d'exécution en temps réel.

Ce présent manuscrit est présenté comme suit :

Le chapitre 1 est dédié au langage de description matériel VHDL (VHSIC Hardware Description Language) avec ses différentes bibliothèques, son entité, son architecture et ses différentes configurations possibles. Les trois descriptions utilisées pour les instructions concurrentes et séquentielles et la partie simulation et synthèse

Le chapitre 2 permet d'englober les différents circuits numériques en partant des circuits standards, les ASIC (Application Specific Integration circuit) et en terminant avec les circuits programmables que ce soit les simples, les complexes PLD ou les FPGA ainsi qu'un bref retracement des différentes technologies utilisées dans la logique programmable que ce soit pour les circuits configurables ou pour les circuits reconfigurables.

Le chapitre 3 permet de détailler la structure interne des circuits FPGA, voir aussi son architecture ainsi que celles de ses composants avec des exemples sur les entrées sorties et les blocs logiques utilisés dans la référence Spartan.

Le chapitre 4 est dédié aux Méthodes de conception : la conception des circuits à faibles densité d'intégration, la conception des circuits à haute densité d'intégration. Les outils de développement : les outils de CAO, les différentes approches de description d'un circuit, les langages de description. Présentation des compilateurs qui contient les outils de CAO.

Le chapitre 5 permet la représentation des nombres relatifs : binaire décalé, signe et valeur absolue, complément à 1, complément à 2. Représentation à virgule fixe. Représentation à virgule flottante. Additionneurs. Multiplieurs. Diviseurs. Comparateurs.

Le chapitre 6 sera consacré à la partie manipulation et simulation de quelques circuits logiques : combinatoire et séquentielles avec un exemple d'implémentation d'un circuit sur une carte FPGA : Spartan 3E Réf : XC3S500E de la famille Xilinx.

Chapitre 1.
Le langage VHDL

I. Le HDL (Hardware Description Language)

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

Il peut généralement : Décrire le fonctionnement du circuit, Décrire sa structure, Et servir à vérifier sa fonctionnalité par simulation ou preuve formelle.

Ils avaient pour but de modéliser, donc de simuler, mais aussi de concevoir, des outils informatiques permettant de traduire automatiquement une description textuelle en dessins de transistors. [1]

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

Les classes de langages dont la seule caractéristique est de décrire un circuit par une hiérarchie de blocs interconnectés est appelée une netlist. [2]

Un synthétiseur logique permet de transformer un circuit décrit dans un langage de description de matériel en une netlist. [1]

II. But du HDL

Le but des HDL est double :

1. La simulation

L'un des objectifs des HDL est d'aboutir à une représentation exécutable d'un circuit, soit sous forme autonome, soit à l'aide d'un programme externe appelé simulateur. Cette forme exécutable comporte :

- Une description du circuit à simuler,
- Un générateur de stimuli (vecteurs de test),
- Un dispositif implémentant la sémantique du langage et l'écoulement du temps.

Il existe deux types de simulateurs :

1/ à temps discret, généralement pour le numérique,

2/ à temps continu pour l'analogique.

Remarque : Des HDL existent pour ces deux types de simulations.

2. La synthèse

En n'utilisant qu'un sous-ensemble d'un HDL, un programme spécial appelé synthétiseur peut transformer une description de circuit en une netlist de portes logiques ayant le même comportement que le circuit de départ. Le sous-ensemble du langage utilisé à ce propos est alors dit synthétisable.

La sémantique synthétisable ignore typiquement toutes les constructions ayant un rapport avec le temps. [1]

III. Les Principaux Langages HDLs

Il existe différents langages de description matériel, assimilable aux langages informatisés standards mais dans le but de décrire des circuits électroniques, nous citant parmi eux :

- **ABEL** "Advanced Boolean Expression Language", un langage propriétaire développé par Data I/O Corporation, maintenant possédé par Lattice;
- **AHDL** ("Altera HDL", langage propriétaire d'Altera pour la programmation de leurs FPGA) langage propriétaire essentiellement structurel proche d'ABEL ;
- **Verilog** qui mélange description structurelle et algorithmique ;
- **VHDL** légèrement plus abstrait que Verilog qui est inspiré de ADA ;
- **SystemC** utilisant le C++ et qui permet de modéliser les interactions logiciel/matériel ;
- **Confluence**, langage déclaratif GPL pouvant générer du Verilog, du VHDL, des modèles exécutables en C et des modèles de vérification formelle ;
- **CUPL** langage propriétaire de Logical Devices;
- **VHDL-AMS** : extension de VHDL destinée aux circuits analogiques ou mixtes. Les langages mixtes, qui sont souvent des extensions des précédents. Ils permettent la modélisation des systèmes à l'aide d'équations différentielles. [2]

IV. VHDL (VHSIC Hardware Description Language)

Le programme VHSIC (Very High Speed Integrated Circuits), impulsé par le département de la défense des Etats-Unis dans les années 1970-1980, a donné naissance à un langage : VHSIC-HDL, connu sous le nom de VHDL . [3]

Le langage de description VHDL est ensuite devenu une norme IEEE 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. [4]

Le but d'un langage de description matériel tel que le VHDL est de faciliter le développement d'un circuit numérique en fournissant une méthode rigoureuse de description du fonctionnement et de l'architecture du circuit désirée.

L'étape suivante consiste à synthétiser cette description matérielle pour obtenir un composant réalisant les fonctions désirées, à l'aide d'éléments logiques concrets (portes logiques, bascules ou registres). Ceux-ci seront implémentés, selon la technologie utilisée, soit directement en transistors (dans le cas d'un ASIC), ou en se basant sur les éléments programmables des FPGA.

Le VHDL ayant une double fonction (simulation et synthèse), une partie seulement du VHDL est synthétisable, l'autre existant uniquement pour faciliter la simulation (écriture de modèles comportementaux et de test benches).

Les avantages du langage VHDL

- ✓ La portabilité : des descriptions VHDL, c'est-à-dire, possibilité de cibler une description VHDL dans le composant ou la structure que l'on souhaite en utilisant l'outil que l'on veut (en supposant, bien sûr, que la description en question puisse s'intégrer dans le composant choisi et que l'outil utilisé possède une entrée VHDL) ;
- ✓ La conception de haut niveau : c'est-à-dire qui ne suit plus la démarche descendante habituelle (du cahier des charges jusqu'à la réalisation et le calcul des structures finales) mais qui se "limite" à une description comportementale directement issue des spécifications techniques du produit que l'on souhaite obtenir.
- ✓ La possibilité de décrire des systèmes très complexes en quelques lignes de code.
- ✓ De plus, le VHDL :
    - Peut-être simulé,
    - Peut être traduit en schéma de portes logiques. [5]

Environnement du VHDL

- ✓ Plateforme : PC/Station
- ❖ Éditeurs de texte, compilateur pour voir les erreurs, la simulation et la synthèse.
- ❖ Dans la partie du test Bench ; Interfaces graphiques (courbes stimulis & résultats).
- ✓ Notion de projets
- ❖ Un analyseur (transformation du code VHDL en éléments logiques de base).
- ❖ Un Place & Route (placement et connection des éléments logiques dans le composant).
- ✓ Bibliothèque communes (IEEE) et Paquetages IEEE, standards et personnalisé par l'utilisateurs (WORK).

Remarque : Beaucoup de logiciels prennent en charge toutes ces fonctionnalités :

Xillinx ISE, Altera Quartus, Lattice ISP Lever, Altium Designer, etc...

Une description VHDL : est un ensemble de fichiers d'extension. Vhd permettant : Édition des fichiers VHDL, Faire référence aux bibliothèques et paquetages nécessaires (IEEE, WORK), Compilation, Simulation, Synthèse et implémentation sur les ASICs ou les FPGAs.

Les unités de compilation VHDL

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

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

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

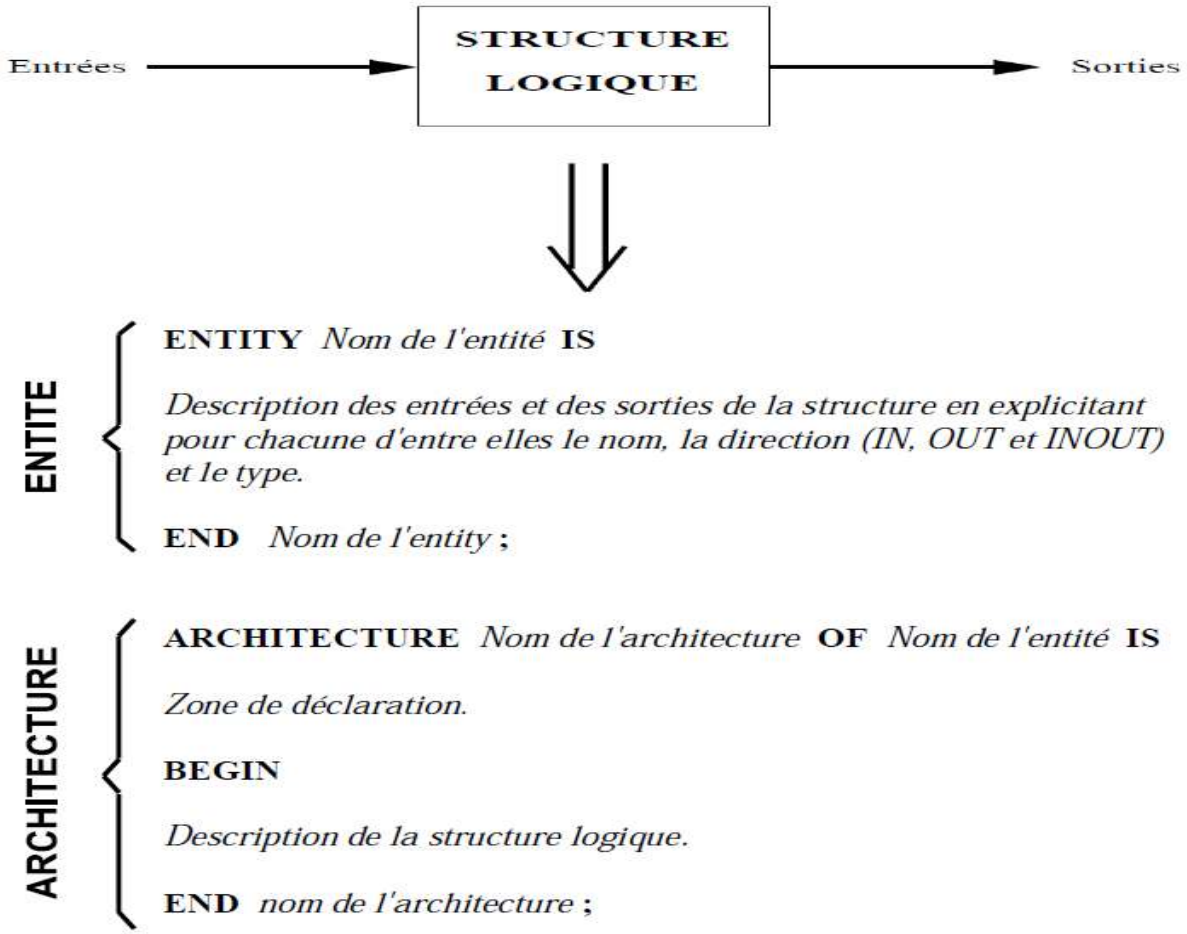


Figure 1 : Structure logique d'un programme VHDL [4]

✓ **Syntaxe de l'entité**

entity mon_circuit **is**

port (les signaux d'entrée **: in (type des signaux);**

les signaux de sortie**: out (type des signaux));**

end mon_circuit**;**

✓ **Syntaxe de l'architecture**

architecture exemple of mon_circuit **is**

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

begin

Corps de l'architecture.

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

end exemple **;**

- ✓ **Syntaxe de la configuration :** La déclaration de configuration est utilisée pour lier des entités et des architectures ensemble. Si votre conception a plusieurs entités et architectures multiples pour une logique similaire. Ensuite, vous pouvez utiliser la configuration pour choisir quelle entité vous allez prendre pour votre conception et ensuite lier cette entité avec une architecture de votre choix.

Syntaxe 1:

for label_1 {,label_2} :nom_du_composant **use entity** nom_d_entité{(nom_d_architecture)**}**

{generic map (correspondance_des_paramètres_génériques)**}**

{port map (correspondance_des_ports)**};**

Syntaxe 2:

for label_1 {, label_2 } :nom_du_composant **use configuration** nom_de_la_configuration

{generic map (correspondance_des_paramètres_génériques)**}**

{port map (correspondance_des_ports)**};**

Remarque : Par défaut, si vous avez plusieurs architectures écrites dans un fichier VHDL et que vous ne spécifiez pas la déclaration de configuration, le VHDL choisit la dernière architecture qui est écrite dans votre conception par elle-même.

- ✓ **Syntaxe du paquetage**

package PKG1 **is**

constant C1 **: integer;**

procedure cvt(I **: in std_logic_VECTOR;** o**: out integer);**

function cvt(I **: in std_logic_VECTOR) return POSITIVE);**

component CMP

generic (N : integer);

port (A, B **: in std_logic;** Y **: out std_logic);**

end component;

end PKG1**;**

Remarque : Et il faut faire la référence du paquage au début du fichier x.vhd qui utilise les éléments du paquage.

Exemple : **use work.**PKG1**.all;**

- ✓ **Syntaxe corps de paquetage**

package body PKG1 **is**

constant C1 **: integer** := 16**;**

procedure cvt(I **: in STD_LOGIC_VECTOR;** o**: out integer) is**

begin

o := cvt(I);

end cvt**;**

function cvt(I **: in STD_LOGIC_VECTOR) return integer is**

begin

return (cvt**_std_logic_vector_**int(I) **);**

end cvt**;**

end PKG1 ; [4]

La sémantique du langage VHDL

- Règles à respecter en langage VHDL
  - ✓ Ils commencent par une lettre,
  - ✓ Ils contiennent des lettres, des chiffres et des traits bas (_),
  - ✓ Ils ne doivent pas contenir deux traits bas consécutifs (_ _),
  - ✓ Ils ne doivent ni commencer, ni finir par un trait bas,
  - ✓ Un identificateur ne doit pas excéder une ligne de programme,
  - ✓ La différence minuscule majuscule n'est pas significative,
  - ✓ Ils doivent être différents des mots clés.
- ✓ Les commentaires :
  - ✓ 2 traits consécutifs indiquent que le reste de la ligne est du commentaire.
  - ✓ Ils sont liés à la ligne
  - ✓ Ils sont nécessaires à la compréhension du modèle mais totalement ignorés par le compilateur.
- ✓ Le langage VHDL exploite 4 objets différents :
  - ✓ **Les signaux :** ils portent les informations des liaisons d'entrée, des liaisons de sortie et des liaisons internes.
  - ✓ **Les constantes :** elles reçoivent leurs valeurs au moment de leurs déclarations.
  - ✓ **Les variables :** elles ne sont déclarées et utilisées que dans les processus, les fonctions et les procédures. L'assignation initiale est facultative.

- ✓ **Les ports :** ils 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.

✓ **Les types de données :**

✓ Les types énumérés

- Type BIT: ce type peut prendre deux valeurs : '0' ou '1'. Ces deux valeurs ne sont pourtant pas suffisantes pour satisfaire les besoins de la synthèse logique : par exemple, l'état haute impédance n'est pas envisagé !
- Type BIT_VECTOR: ce type permet de manipuler des grandeurs résultant de l'association d'éléments de type BIT (i.e. des vecteurs de bits ou mot).
- Type BOOLEAN: ce type ne peut prendre que deux valeurs différentes : TRUE (vrai) ou FALSE (faux)
- Types STD_LOGIC et STD_LOGIC_VECTOR: ces types sont inclus dans la bibliothèque IEEE 1164,
- STD_LOGIC : 9 valeurs décrivant tous les états d'un signal logique

  'U' : non initialisé**

  'X' : niveau inconnu, forçage fort**

  '0' : niveau 0, forçage fort

  '1' : niveau 1, forçage fort

  'Z' : haute impédance *

  'W' : niveau inconnu, forçage faible**

  'L' : niveau 0, forçage faible

  'H' : niveau 1, forçage faible

  '-' : quelconque (don't care) *

- STD_LOGIC_VECTOR : vecteur de STD_LOGIC
  Pour utiliser ces types il faudra inclure les directives suivantes dans le code source.
  library ieee;
  use ieee.std_logic_1164.all;
- Types CHARACTER et STRING :
  Les variables de type CHARACTER prennent 95 valeurs différentes correspondant à une partie du jeu de caractères ASCII. Le type STRING résulte de l'association de plusieurs éléments de type CHARACTER.
- Type SEVERITY_LEVEL :
  Ce type est employé avec une fonction spécifique aux programmes de tests et permet de caractériser le niveau de gravité programmable lors d'incidents de simulation. Les variables de ce type peuvent prendre les valeurs suivantes : NOTE, WARNING, ERROR, FAILURE.

✓ Les types numériques non énumérés

- Type INTEGER: valeurs signées codées sur 32 bits.

- Type REAL: l'étendue des valeurs dépend du compilateur VHDL et la norme VHDL impose des valeurs aux puissances de 10 au moins comprises entre -38 et +38.
- Sous-types NATURAL: ce type correspond en fait à une restriction du type INTEGER dans on conserve toutes les valeurs supérieures ou égales à 0.
- Sous-types POSITIVE: De la même façon, le sous-type POSITIVE est une restriction du type INTEGER dont seules les valeurs supérieures à 0 sont conservées.
- Types UNSIGNED et SIGNED: ces types UNSIGNED et SIGNED utilisent pour base des vecteurs de bits. L'éventail des valeurs que peut couvrir une variable ou un signal de type UNSIGNED ou SIGNED dépendra du nombre de bits qui lui sera attribué lors de la déclaration.

✓ Les types utilisateurs
- Types énumérés définis par l'utilisateur :
  Il est possible de créer de nouveaux types de données.
- Types non énumérés définis par l'utilisateur :
  Pour la création de type non énuméré, le VHDL dispose du mot clé RANGE faisant office de directive et permettant de spécifier l'intervalle des valeurs valides pour le type créé.
- Enregistrement : RECORD:
  Lorsque des informations de toute nature doivent être réunies, VHDL permet la création d'un type spécifique qui autorisera la manipulation de toutes ces données sous un même identificateur, appelé enregistrement. On utilise alors le mot clé record.
- Types vectoriels :
  Les types utilisateurs qui viennent d'être décrits sont des scalaires. On peut aussi créer des types vectoriels dont on fixe le nombre d'éléments dans la déclaration du type ou lors de la déclaration de l'instance. Ces types reposent sur le mot clé array.

✓ **Les opérateurs :**

✓ **Opérateurs d'affectation :**
Le VHDL utilise deux opérateurs d'affectations :
- Affectation à un signal : pour réaliser une affectation à un signal, on utilise l'opérateur <=
- Affectation à une variable : l'affectation à une variable exploite l'opérateur **:=**

✓ **Opérateurs relationnels :**
Ils sont valables avec des signaux et des variables scalaires et vectoriels.
=, /= (différent), <, <=, >, >=
La différence entre l'opérateur relationnel <= et l'opérateur d'affectation <= est effectuée par le compilateur en fonction de son contexte d'utilisation.

✓ **Opérateurs logiques :**
Ils s'appliquent à des signaux ou des variables de types booléens, bits et dérivés : BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, BOOLEAN.
- AND, OR, NAND, NOR et XOR.
- AND n'est pas prioritaire par rapport à OR (utiliser des parenthèses)

- Dans le cas d'opérandes de types STD_LOGIC_VECTOR ou BIT_VECTOR, l'opération logique s'effectue BIT à BIT (cela implique des opérandes de tailles identiques pour les opérateurs binaires).

✓ **Opérateurs arithmétiques :**
Ils sont définis pour les types entiers et réels.
+ , -, *, / (division entière), ** (exponentiation), mod (modulo), rem (reste de la division), abs (valeur absolue).

✓ **Opérateur de concaténation :**
L'opérateur de concaténation & permet de manipuler des tableaux de tout type et de différentes tailles pour en réaliser l'association sous un autre label de dimension plus i. Exemple : A <= "1100";
B <= "0011";
C <= A & B; C <= "11000011". [4]

❖ **Les instructions**

✓ **Instructions concurrentes :**
- ✓ L'ordre d'écriture des instructions n'a pas d'importance (c'est le parallélisme).
- ✓ Une instruction concurrente décrit une opération qui porte sur des signaux (entrée, interne) pour produire d'autres signaux (interne, sortie).
- ✓ Tous les signaux mis en jeu dans l'architecture sont disponibles au même moment.
- ✓ Le corps d'architecture est décrit dans un ou plusieurs styles.
- ✓ Le style n'est pas imposé par le type de logique (combinatoire ou séquentielle) ou le type de traitement des données (parallèle ou séquentiel).
- ✓ Trois styles peuvent coexister au sein d'une même architecture.
- ✓ Le VHDL est un langage concurrent ➠ le système à modéliser peut-être diviser en plusieurs blocs agissant en parallèle.
- ✓ L'instruction concurrente est exécutée quand un évènement sur certains de ses signaux apparait :
  - Instanciation de composants
    Création d'un composant dont le modèle est défini ailleurs (hiérarchie).
    Exemple de syntaxe
    Architecture structurelle **of** adder **is**
    .....
    Begin
    C1 : PORTE_ET **port map** (A1, A2, S1) ;
    C2 : PORTE_OU **port map** (A1, S1, S2) ;
    C3 : INVERSEUR **port map** (S2, S3) ;
    ...
    .....
    End structurelle ;
  - Process
    Ne contient que des instructions séquentielles
    Un process "vit" toujours ➠ cyclique
    Doit être controlé par sa liste de sensibilité ou des synchronisations internes (wait).
    Process (liste de sensibilité) ou **process**

Déclarations **begin** instructions séquentielles ..... **end process;**	Déclarations **begin** **wait** on (liste de signaux) instructions séquentielles ..... **end process;**

Remarque : Ne pas mettre une **liste de sensibilité** et un (ou des) **wait**.

- Affectation de signaux

Affectation simple:

Signal **<=** forme d'onde { délai }

Affectation conditionnelle:

Signal <= forme d'onde 1 **when** condition 1 **else** forme d'onde 2

Affectation sélective:

with expression **select**

Signal <= forme d'onde 1 **when** choix 1,
forme d'onde 2 **when** choix 2,

- Appel de procédure

Dans le cas d'un appel concurrent les paramètres sont des signaux ou des constantes sensible à un événement sur les paramètres d'entrée.

{label:} nom_de_la_procedure **(liste des paramètres)**;

- Assertion

Surveille une condition et génère un message si la condition est fausse

On y associe un niveau d'erreur qui peut être pris en compte par le simulateur.

{label:} **assert condition** { report message } { **severity niveau_d'erreur** };

- Bloc

Similaires à un process mais contiennent des instructions concurrentes

Le "block" permet :

– la hiérarchisation des instructions concurrentes et des déclarations locales

– de synchroniser les instructions du bloc avec l'extérieur de ce dernier

label : **block** { condition de garde }

-- déclarations locales

begin

-- instructions concurrentes

end block label;

- Génération

Raccourci d'écriture ➠ élaboration itérative ou conditionnelle d'instructions concurrentes

forme conditionnelle :

➥les instructions sont générées si la condition est vraie

{label :} **if** condition_booléenne **generate**

.....

suite d'instructions concurrentes

end generate {label} ;

forme itérative :

➥génération de structures répétitives indicées

{label :} **for** nom_de_l'index **in** intervalle_discret **generate**
..... -- l'index n'a pas besoin d'être déclaré
instructions concurrentes
end generate {label} ;

- ✓ **Instructions séquentielles :**
  - ✓ Elles sont utilisées dans les process ou les corps de sous-programmes.
  - ✓ Elles permettent des descriptions comportementales :
    - Affectation de variables
      Exemple :
      resul := 4;
      resul := resul + 1;
    - Affectation séquentielles de signaux
      Exemple :
      S <= 4 **after** 10 ns;
      S2 <= S **after** 20 ns;
    - Instruction de synchronisation
    - Instructions de contrôle
    - Appels de procédure
      Process
      begin
      ....
      test_valeur (A,B);
      ....
      end Process; -- Appel en séquentiel
    - Assertion
      assert S_OUT /= "0000"
      report "S_OUT est nul "
      severity warning; -- testé en séquentiel
    - Null
      case DIVI **is**
      when '0' => ..... ;
      when '1' => **null**;
      end case;
- ✓ **Programmation modulaire**
  - ✓ **Les fonctions**

    Une fonction retourne au programme appelant une valeur unique, elle a donc un type.
    - **Déclaration :**

    function nom [(liste de paramètres formels)]
    return nom_de_type ;
    - **Corps de la fonction :**

    function nom [(liste de paramètres formels)]
    return nom_de_type is
    [déclarations]
    begin
    instructions séquentielles

end [nom] ;
Le corps d'une fonction ne peut pas contenir d'instruction wait, les variables locales, déclarées dans la fonction, cessent d'exister dès que la fonction se termine.
Utilisation :
nom (liste de paramètres réels)
Lors de son utilisation, le nom d'une fonction peut apparaître partout, dans une expression, où une valeur du type correspondant peut être utilisée.

- ✓ **Les procédures**
  Une procédure, comme une fonction, peut recevoir du programme appelant des arguments : constantes, variables ou signaux. Donc une procédure peut renvoyer un nombre quelconque de valeurs au programme appelant.
  - **Déclaration :**

  procedure nom [(liste de paramètres formels)];
  - **Corps de la fonction :**

  procedure nom [(liste de paramètres formels)] is
  [déclarations]
  begin
  instructions séquentielles
  end [nom] ;
  Dans la liste des paramètres formels, la nature des arguments doit être précisée :
  procedure exemple (signal a, b: in bit; signal s : out bit) ;
  Le corps d'une procédure peut contenir une instruction wait, les variables locales, déclarées dans la procédure, cessent d'exister dès que la procédure se termine.
  Utilisation :
  Nom (liste de paramètres réels)

- ✓ **Les paquetages**
  Un paquetage permet de rassembler des déclarations et des sous programmes, utilisés fréquemment dans une application, dans un module qui peut être compilé à part, et rendu visible par l'application au moyen de la clause use.
  - ✓ **Les paquetages prédéfinis.**
    Un compilateur VHDL est toujours assorti d'une librairie, décrite par des paquetages, qui offre à l'utilisateur des outils variés.
  - ✓ **Les paquetages créés par l'utilisateur.**
    L'utilisateur peut créer ses propres paquetages. Cette possibilité permet d'assurer la cohérence des déclarations dans une application complexe, évite d'avoir à répéter un grand nombre de fois ces mêmes déclarations et donne la possibilité de créer une librairie de fonctions et procédures adaptée aux besoins des utilisateurs.

- ✓ **Les librairies**
  Une librairie est une collection de modules VHDL qui ont déjà été compilés. Ces modules peuvent être des paquetages, des entités ou des architectures.

Une librairie par défaut, work, est systématiquement associée à l'environnement de travail de l'utilisateur. Ce dernier peut ouvrir ses propres librairies par la clause library:
library nom_de_la_librairie ;
La façon dont on associe un nom de librairie à un, ou des, chemins, dans le système de fichiers de l'ordinateur, dépend de l'outil de développement utilisé. [4]

Modélisation en VHDL

- ✓ Les 3 niveaux de description :
  - ✓ **La description en flot de données.**
    - Utiliser des instructions concurrentes d'assignation de signal.
    - Description de la manière dont les données circulent de signal en signal, ou d'une entrée vers une sortie
    - Trois types d'instructions :

étiquette : … <= …

étiquette : … <= … when … else …

étiquette : with … select … <= … when …

L'ordre d'écriture des instructions d'assignation de signaux est quelconque (le parallélisme).

  - ✓ **La description comportementale.**
    - Utiliser des instructions concurrentes d'appel de processus.
    - Certains comportements peuvent être décrits de façon algorithmique ; il faut alors les définir comme des processus.
    - Dans l'architecture utilisatrice, un processus est considéré comme une instruction concurrente.
    - Instruction d'appel de processus :

étiquette: process
déclarations
begin
Instructions séquentielles
end process;

    - L'ordre d'écriture des instructions d'appel de processus est quelconque.
    - Un processus contient des instructions séquentielles qui ne servent qu'à traduire simplement et efficacement, sous forme d'un algorithme, le comportement d'un sous-ensemble matériel.
    - À l'intérieur d'un processus, trois types d'instruction d'assignation de signaux et deux types d'instruction d'itération

… <= … cas d'un signal interne
if … then … else …
case … when …
for … loop …
while … loop …
L'ordre d'écriture des instructions à l'intérieur du processus est déterminant.

- ✓ La description structurelle**.**
  - Instructions concurrentes d'instanciation de composant.
  - Interconnexion de composants (component), à la manière d'un schéma, mais sous forme d'une liste.
  - Dans l'architecture utilisatrice, un composant peut être considéré comme une boîte noire.
  - Un composant est instancié à l'aide d'une instruction d'appel de composant :

    Étiquette : nom_composant **port map** (liste des entrées et sorties);

    L'ordre d'écriture des instructions d'instanciation de composants est quelconque. Mais la liste doit suivre l'ordre des E/S déclaré auparavant.

  - Par définition, un composant est aussi un système logique (un sous-système) ; à ce titre, il doit aussi être décrit par un couple (entité, architecture) dans lequel sont définis ses entrées-sorties et son comportement.
  - Le corps d'architecture du composant (le comportement) est décrit selon un ou plusieurs Descriptions ; par exemple, un composant peut faire appel à d'autres composants.
  - Un composant peut être rangé dans une bibliothèque. [5]

❖ Simulations

VHDL est à la fois un langage de synthèse et de simulation.

- ✓ Un testbench VHDL est un code VHDL destiné à la vérification, par simulation, du bon fonctionnement d'un système, lui-même décrit en VHDL.
- ✓ Dans l'industrie, les testbenches jouent un rôle très important ; ils sont intégrés dans les spécifications d'un système.
- ✓ Simulation directe (à éviter, sauf si vous avez des simulations déjà tester auparavant) :
  - Charger le composant à simuler dans le simulateur.
  - Faire varier entrée par entrée à chaque pad de simulation.
  - Si on a 1 circuit : horloge + reset + 10 entrées !
    Pour un test exhaustif : $2^{11}+1$ possibilités.
    Si 1 possibilité = 5 sec.
    Alors il faut ~ 170 min. = ~ 3heures juste pour fixer les entrées.
- ✓ Simulation avec un fichier de test (testbench) est recommandée : Méthodique et un Gain de temps énorme.
  - En VHDL :
    Les entrées : lecture seulement
    Les sorties : écriture seulement
  - Compilation
  - Chargement dans le simulateur
  - Choix des signaux à observer
  - Lancement de la simulation
  - Résultat sur la fenêtre des courbes

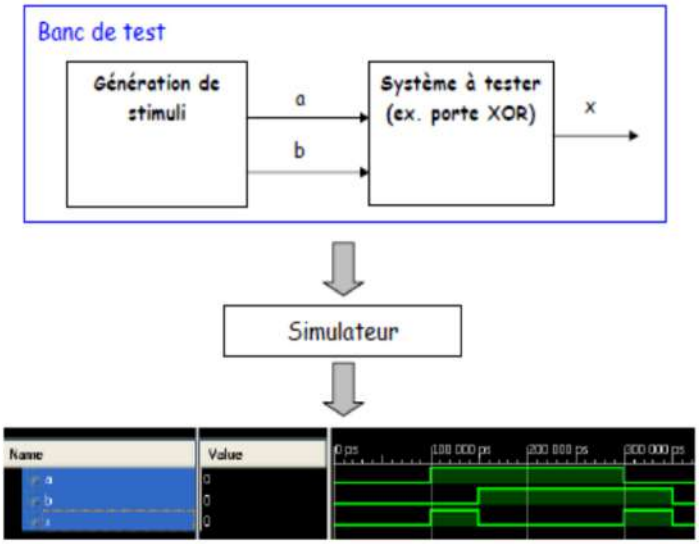


Figure 2 : Procédure de test

- ✓ Le code VHDL d'un banc de test
  - Structure du fichier pour le système à tester

Entité (Entrés, sorties)
Architecture
Déclarations
Instructions

  - Structure du fichier pour le banc de test :

Entité (ni entrée, ni sortie)
Architecture
Déclaration de composant du système à tester
Déclaration des signaux de test
Instruction d'instanciation
Instructions de génération de stimuli (entrées).

V. Conclusion

L'urgence de traiter l'information en temps réel à laisser les fabriquent des circuits intégrer à satisfaire les clients et les équipements en lui offrant une interface VHDL qui permet de traduire n'importe quel circuit en quelques lignes de code et sur diverses descriptions de composants ou de fonctions annexes constituant son application [6].

Des bibliothèques contiennent des unités pour la compréhension des opérations arithmétiques [7], ses unités utilisent des architectures de circuits performantes qui sont optimisées pour la synthèse et la conception à base de cellules (cellules standard, porte d'entrée, FPGA à grain fin).

Le VHDL reste pour le moment le langage de description matériel le plus utiliser pour les circuits FPGA pour son traitement parallèle et rapides des données, malgré l'utilisation des autres HDL comme le Verilog qui est son premier concurrent.

Chapitre 2.
Les circuits numériques

I. Introduction

Un circuit logique programmable, ou autrement dit un réseau logique programmable, est un circuit intégré qui fonctionne de manière logique et qui permet ça reprogrammation à volonté même après sa fabrication. Il n'y a pas de programmation en logiciel (contrairement à un microprocesseur). On utilise plutôt le terme « reconfiguration » plutôt que « reprogrammation » (car ses connexions ou ses composants, sont à base des portes logiques et des bascules). Le terme programmation est utilisé fréquemment, pour personnaliser les réseaux logiques reconfigurables et modifiables. Nous allons voir dans ce chapitres les différents circuits numériques et logiques programmables.

II. Architectures classiques des circuits numériques

Le rôle de la technologie micro-électronique est la réalisation et l'intégration des transistors nécessaires à la réalisation des circuits électroniques.

Cette technologie d'intégration se présente sous formes de plusieurs possibilités :

- ✓ Utiliser des circuits standards.
- ✓ Concevoir des circuits spécifiques à l'application (ASIC).
- ✓ Utiliser des circuits programmables (PLD).

Les systèmes électroniques modernes sont de plus en plus complexes, Les contraintes de taille, de puissance dissipée et de performances sont de plus en plus sévères (téléphonie mobile, ordinateurs, traitement du signal, de l'image, etc.…).

Accroissement spectaculaire des densités d'intégrations :

- 1964 Intégration à petite échelle (SSI de 1 à 10 transistors).
- 1968 Intégration à moyenne échelle (MSI de 10 à 500 transistors).
- 1971 Intégration à grande échelle (LSI de 500 à 20 000 transistors).
- 1980 Intégration à très grande échelle (VLSI plus de 20 000 à 1000 000 transistors).
- 1984 Intégration à très grande échelle (ULSI plus de 1000000 transistors). [8]

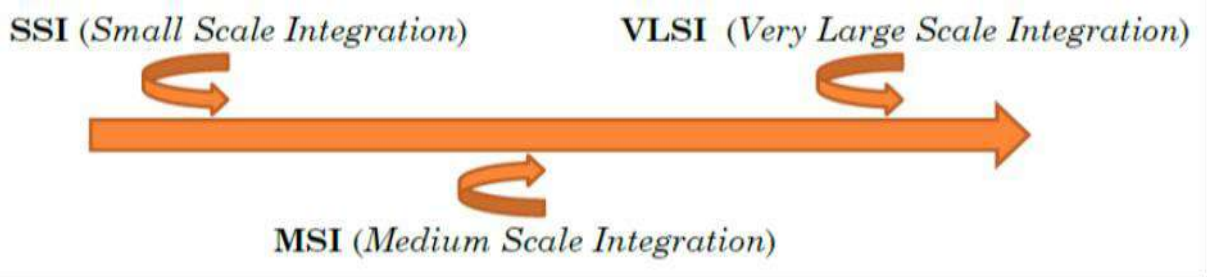


Figure 3 : les différentes échelles d'intégrations

III. Classification des circuits numériques

Les circuits numériques se réparties sur trois grandes familles des circuits bien distincte, comme le montre la figure suivante :

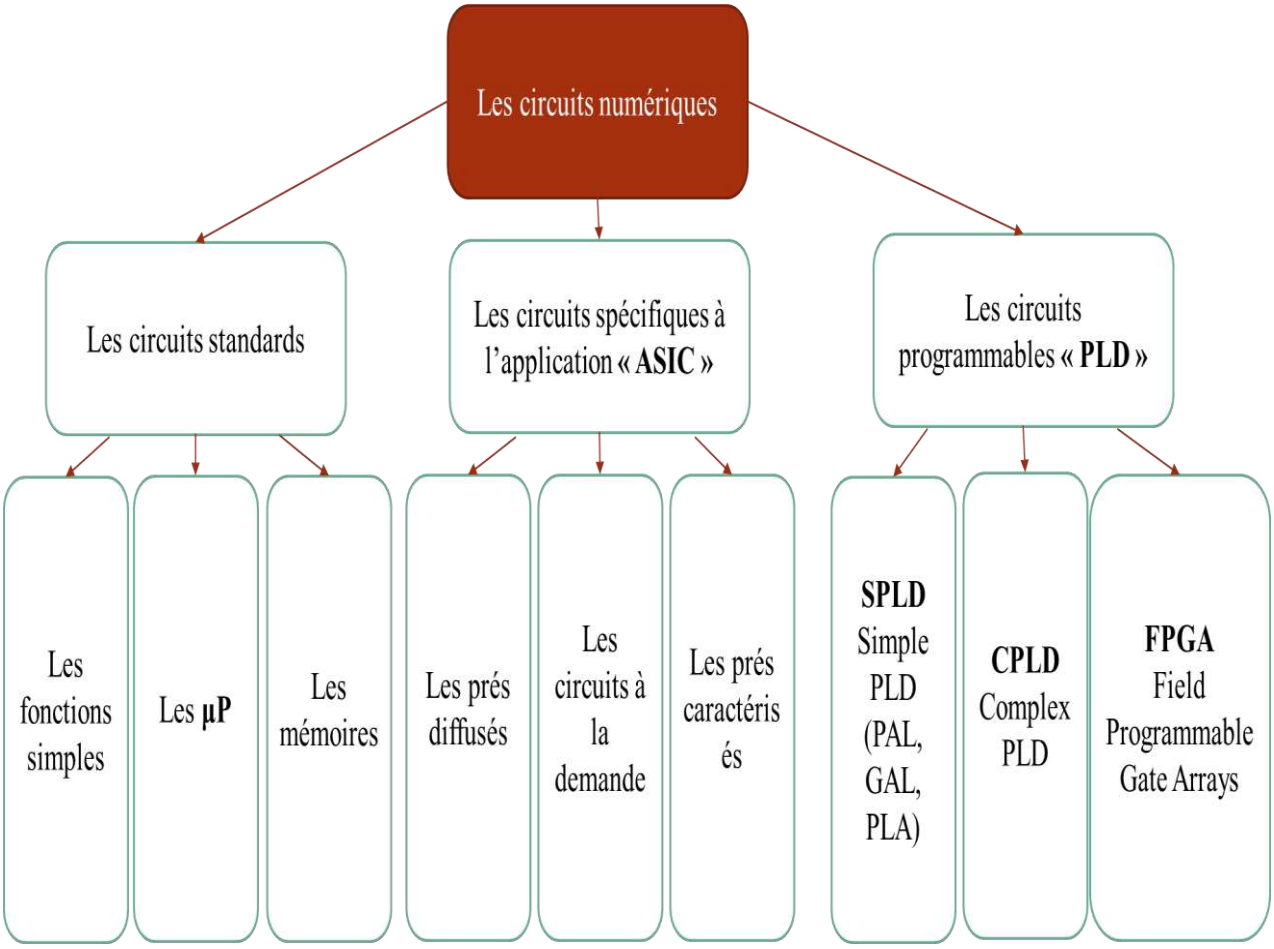


Figure 4 : classification des circuits numériques

1. Les circuits standards

Des fabricants de circuits tels que MATRA, MOTOROLA, SGS THOMSON, … proposent des composants standards ayant des fonctions plus ou moins complexes. L'association de ces composants sur un circuit imprimé permet de réaliser un système numérique.

Ces circuits standards se présente sous forme de trois critères à savoir :

- ✓ **Les fonctions simples.**
  - Certains circuits combinatoires de moyenne complexité MSI (Medium Scale Integration) sont considérés comme des circuits standards de base.
  - On peut trouver aussi les circuits SSI (Small Scale Integration), qui réalisent des fonctions combinatoires ou séquentielles élémentaires.
  - Les fonctions correspondantes réalisées en circuits intégrés se retrouvent comme composants dans les bibliothèques (librairies) de conception des circuits logique programmables.
- ✓ **Les microprocesseurs**.
  - *Processeur à usage général :* Pour minimiser l'impact du coût de conception et de fabrication des circuits intégrés les plus complexes :
    Créer un circuit de traitement numérique dont l'usage final (l'application) n'est pas connu à la fabrication.

Réaliser un circuit intégré ayant quelques ressources de traitement assez génériques (addition de deux nombres, stockage d'un nombre en mémoire, lecture d'un nombre d'une mémoire...). Avec de tels circuits l'augmentation de complexité des applications est gérée simplement par l'augmentation de la taille des programmes.

- *DSP (Digital Signal Processor) :*
  Le DSP est un microprocesseur optimisé, pour exécuter des applications de traitement numériques du signal (filtrage, extraction de signaux, etc) le plus rapidement possible.
  Les DSP sont utilisés dans la plupart des applications du traitement numérique du signal en temps réel.
  On les trouve dans les Modems (RTC, ADSL), les téléphones mobiles, les appareils multimédia (lecteur MP3), les récepteurs GPS, Etc…

✓ **Les mémoires.**

Une mémoire est un dispositif permettant de stocker puis de restituer une information. On distingue deux classes de mémoires à semi-conducteur :

- *Les mémoires vives :* sont des mémoires volatiles, car on peut perdre l'information en cas de coupure d'alimentation électrique. Elles peuvent être lues et écrites.
- *Les mémoires mortes* : sont des mémoires qui conservent l'information même en absence de l'alimentation. Donc on peut les considérées comme un circuit logique programmable.

2. Les circuits spécifiques à l'application ASIC (Application Specific Integration circuit)

- Les circuits ASIC constituent la troisième génération des circuits intégrés, apparu au début des années 80 après les bipolaires et les MOSFET (Metal Oxyde Semi-conducteur) ou dite à effet de champs pour FET. L'ASIC présente une personnalisation de son fonctionnement, selon l'utilisation :
  - Une réduction du temps de développement
  - Une augmentation de la densité d'intégration et de la vitesse de fonctionnement.

- L'ASIC est un circuit intégré qui permet un câblage direct des applications spécifiques sur le silicium. Il se présente sous trois catégories : les prés diffusés, les circuits à la demande et les prés caractérisés.

✓ **Les prés diffusés (Gate Array) :**

Les prés diffusés sont une approche de la conception et de la fabrication des circuits intégrés spécifiques à une application (ASIC) utilisant une puce préfabriquée avec des composants qui sont ensuite interconnectés dans des dispositifs logiques (par exemple des portes NAND, des bascules, etc.) selon un ordre personnalisé en : ajout de couches d'interconnexion métalliques en usine.
Une "mer" de portes est routée. Les éléments logiques existent déjà physiquement sur le circuit, seules les connexions peuvent être définies

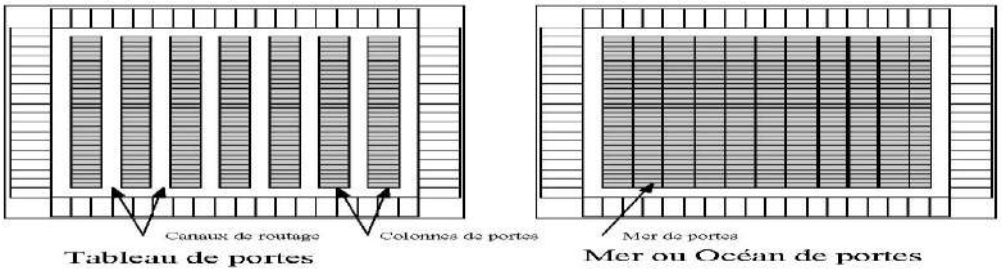


Figure 5 : ASIC (Près diffusés) [9]

✓ **Les circuits à la demande (Full-Custom) :**

Tous les éléments utilisés sont développés par le concepteur ; le fabricant réalise tous les niveaux de masque. Tout est modifiable : transistors (type, caractéristiques), connexions…

L'Intérêt du Full Custom :

- Création de circuits standards, de circuits programmables, de bibliothèques de cellules standards ou d' IPs.
- Utilisation des technologies les plus récentes.
- Contraintes de performances trop fortes pour les composants existants.

✓ **Les prés caractérisés (Standard Cell) :**

Une cellule standard (le prés caractérisé) est un groupe de transistors et de structures d'interconnexion fournissant une fonction logique booléenne (par exemple, ET, OU, XOR, XNOR, inverseurs) ou une fonction de stockage (bascule ou verrou). Les cellules les plus simples sont des représentations directes de la fonction booléenne élémentaire NAND, NOR et XOR, bien que des cellules d'une complexité beaucoup plus grande soient couramment utilisées (comme un additionneur complet à 2 bits ou une bascule binaire à entrer D multiplexée).

La logique booléenne de la cellule fonction est appelée par sa vue logique, le comportement fonctionnel est capturé sous la forme : d'une table de vérité ou d'une équation d'algèbre de Boole (pour la logique combinatoire), ou d'une table de transition d'état (pour la logique séquentielle). Dans les ASIC prés caractérisés :

- Le circuit est un assemblage de cellule placées/routées.
- Les éléments logiques sont choisis dans une bibliothèque de portes, les connexions sont libres.

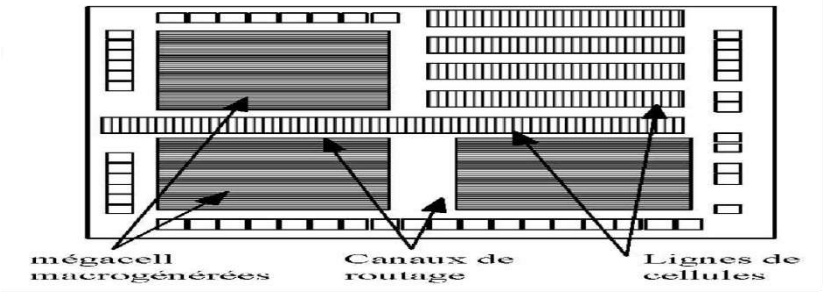


Figure 6 : ASIC (prés caractérisés) [9]

3. Les circuits programmables PLD (Programmble Logic Device)

Nom générique donné à l'ensemble des circuits monolithiques formés de cellules logiques (comportant des fusibles, des antis fusibles ou de la mémoire) qui peuvent être programmés et parfois reprogrammés par l'utilisateur.

Il existe trois principaux circuits programmables :

3-1: Les SPLD (Simple PLD)

Les Simple Programmable Logic Device sont des circuits programmables élémentaires, constitué d'un ensemble de portes « ET » sur lesquelles viennent se connecter les variables d'entrée et leurs compléments. Ainsi qu'un ensemble de portes « OU » sur lesquelles les sorties des opérateurs « ET » sont connectées les variables d'entrée. Comme le montre la figure suivante :

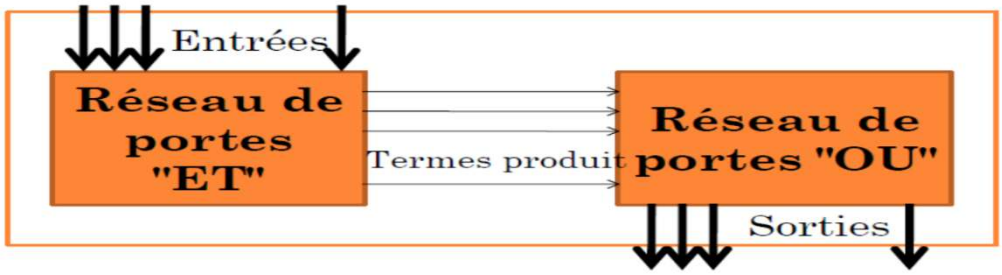


Figure 7 : Architecture globale d'un SPLD

Il existe trois types des SPLD :

- **PAL (Programmable Array Logic) :** réseaux logiques programmables
  - ✓ Développés au début des années 70 par MMI (ex-AMD).
  - ✓ La programmation se fait par destruction de fusibles.
  - ✓ Aucun fusible n'est grillé à l'achat de la PAL. [9]

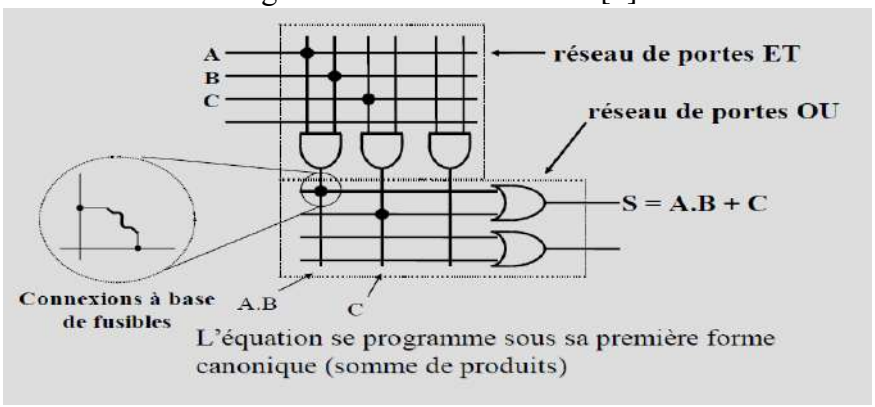


Figure 8 : PAL Architecture [9]

Remarque : Les fonctions ET sont programmables

- **GAL (Généric Array Logic)**

L'inconvénient majeur des PAL est qu'ils ne sont programmables qu'une seule fois. Ce qui a ramené la firme LATTICE a pensé de remplacer les fusibles irréversibles

des PAL par des transistors MOS FET qui peuvent être régénérés. Ceci a donné naissance aux GAL « Réseau Logique Générique ». Ces circuits peuvent donc être reprogrammés à volonté, la figure 9 présente une implémentation d'un circuit GAL.

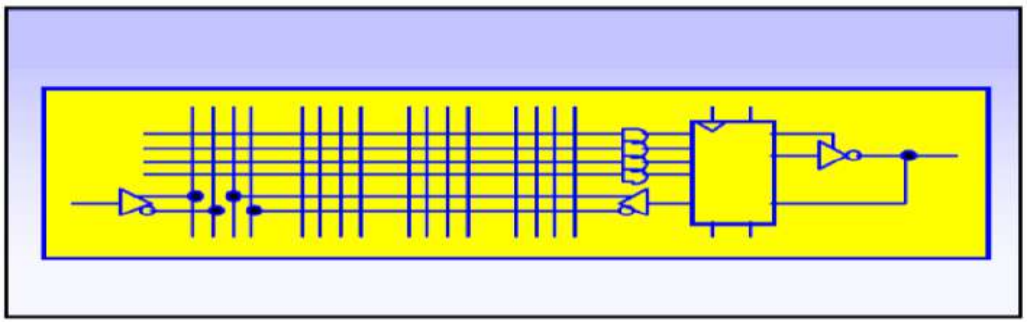


Figure 9 : Implémentation ET-OU-bascule D d'une cellule de base d'un circuit GAL [10]

- **PLA (Programmable Logic Arrays)**

Les PLA diffèrent des appareils logiques programmables (PAL et GAL) dans la mesure où les fonctions de porte ET et OU sont programmables. Comme le montre la figure suivante :

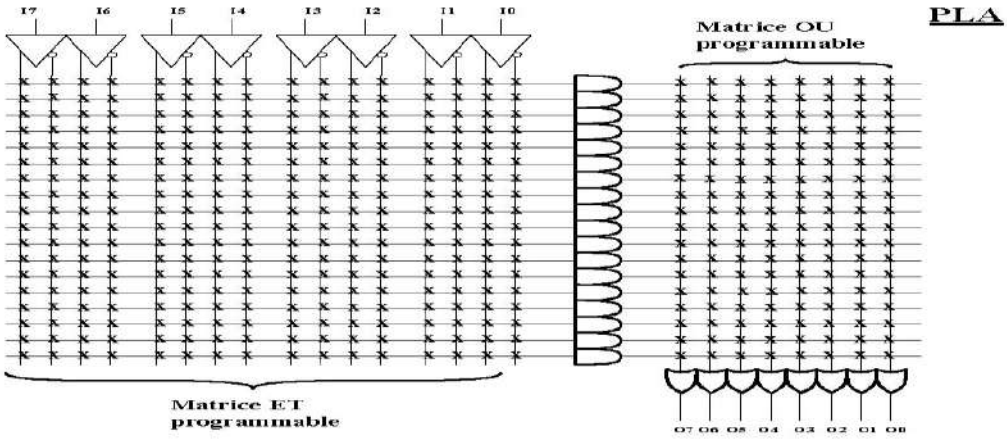


Figure 10 : Circuit PLA programmable des deux portes ET et OU [10]

Le tableau suivant va récapituler les trois technologies des circuits SPLD

Type	Plan ET	Plan OU	Technologie	Utilisation classique
PAL	Programmable	Fixe	Bipolaire	Décodage, machine à état
GAL	Reprogrammable	Fixe	CMOS	Décodage, machine à état
PLA	Programmable	Programmable	CMOS	Fonctions logiques complexes

Tableau 1 : Comparaison entre les SPLD

- **LIMITATIONS DES SPLD**
  - Impossibilité d'implémenter des fonctions multi-niveaux,
  - Impossibilité de partager des produits entre fonctions,
  - Avec les CPLDs on peut maintenant non seulement programmer la fonctionnalité mais aussi l'interconnexion entre 2 cellules.
  - Contrairement aux FPGAs, il n'y a qu'un seul chemin entre 2 points,
  - Les CPLDs perdent en flexibilité mais gagnent en prédictibilité.
  -

3-2 : Les CPLD (Complex PLD)

Les CPLD peuvent être vu comme une intégration de plusieurs PLD simples (SPLD) dans une structure à deux dimensions, ils sont composés de blocs logiques répartis autour d'une matrice d'interconnexion PI (Programmable Interconnect). Comme le montre la figure 11.

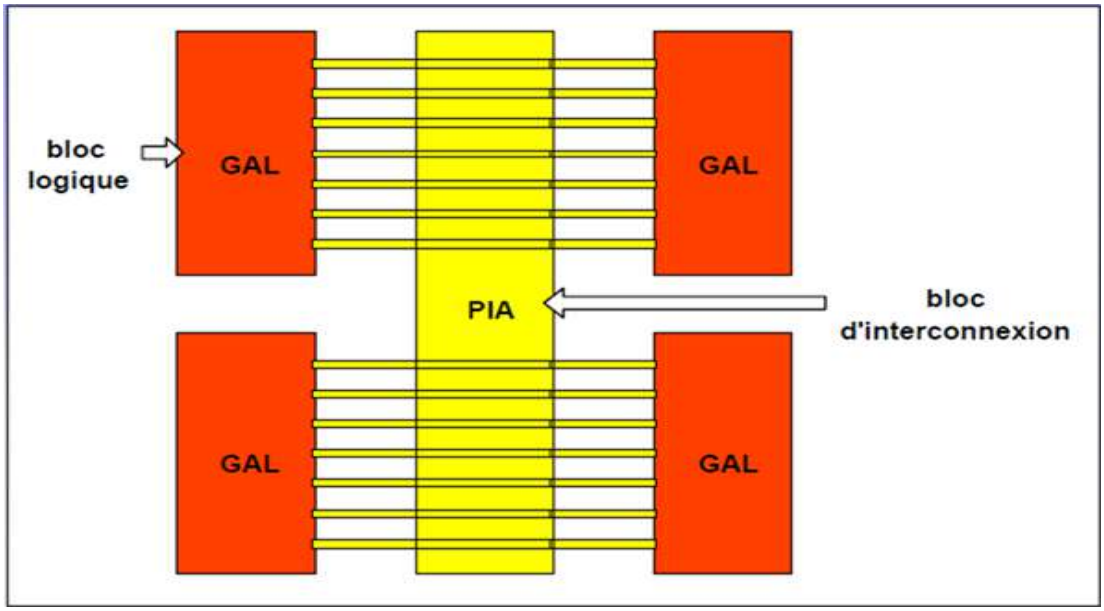


Figure 11 : Circuit CPLD [10]

Ça structure interne (Macro-cellule) est composée d'une zone de portes logiques et une bascule.

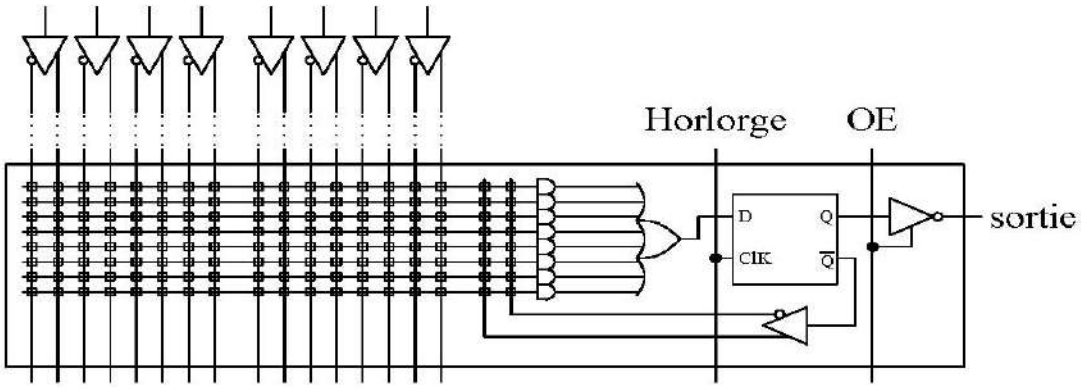


Figure 12 : Macro-cellule Programmable [10]

Dans les années 80, Altera sont les premiers à fournir une solution utilisable de CPLD : Utilisé pour le prototypage rapide, Un problème restait pourtant au niveau de la matrice d'interconnexion entre les SPLD, Limitant la taille des designs à prototyper

En 1984 Xilinx, lance le premier Field Gate Array (FPGA), le XC2064. La principale différence est sa flexibilité sans perte de performances et son inconvénient est le manque de prédictibilité des temps d'interconnexion.

- **Ressemblance entre CPLD et FPGA**
  - Les cellules logiques sont placées dans une topologie donnée, et reliées par une infrastructure d'interconnexion.
  - Leur fonction est programmable,
  - Les chemins entre 2 cellules sont multiples et les temps ne sont connues qu'après le routage.
  - Les cellules externes (IO cells) ne sont pas programmable fonctionnellement mais en :Direction,Voltage, Bufferisation.
  - Avantage principal : Le temps de conception.

- **Différence entre CPLD et FPGA**
  - FPGA contient jusqu'à 100 000 de minuscules blocs logiques, tandis que CPLD ne contient que quelques blocs de logique allant jusqu'à quelques milliers.
  - En termes d'architecture, les FPGA sont considérés comme des dispositifs à « grain fin » alors que les CPLD sont des « grains grossiers ».
  - Les FPGA sont parfaits pour des applications plus complexes, tandis que les CPLD sont mieux pour les plus simples.
  - Les FPGA sont constitués de minuscules blocs logiques tandis que les CPLD sont constitués de blocs plus gros.
  - FPGA est une puce logique numérique basée sur la RAM, tandis que CPLD est basé sur EEPROM.
  - Normalement, les FPGA sont plus coûteuses alors que les CPLD sont beaucoup moins chers.
  - Les délais sont beaucoup plus prévisibles dans les CPLD que dans les FPGA.

3-3: Les FPGA (Field Programmable Gate Arrays)

Circuit programmable composé d'un réseau de blocs logiques élémentaires (plusieurs milliers de portes), de cellules d'entrée-sortie et de ressources d'interconnexion totalement flexibles.

Ce circuit, qui nécessite un outil de placement-routage, est caractérisé par son architecture, sa technologie de programmation et les éléments de base de ses blocs logiques.

La puissance de ces circuits est telle qu'ils peuvent être composés de plusieurs milliers voire millions de portes logiques et de bascules. Les dernières générations de FPGA intègrent même de la mémoire vive (RAM). Les deux plus grands constructeurs de FPGA sont XILINX et ALTERA.

Les FPGA se situent entre les réseaux logiques programmables et les circuits logiques près diffusés. Les réseaux logiques programmables sont des composants qui ne nécessitent aucune

étape technologique supplémentaire pour être personnalisés, ce sont des circuits standards, programmables par l'utilisateur grâce aux différents outils de développement et qui incluent un grand nombre de solutions basées sur les variantes de l'architecture des portes ET et OU. Les prés diffusés sont des circuits intégrés basés sur l'utilisation des réseaux de cellules dont les blocs ont été préalablement diffusés, il faut créer les connexions entre ces blocs [14].

IV. Généralité sur les technologies des éléments programmables

On trouve les éléments programmables dans les blocs logiques des PLDs, afin de leur donner une fonctionnalité, mais aussi dans les matrices d'interconnexions entre ces blocs. Un élément programmable peut être considéré comme un interrupteur. Afin de respecter les contraintes imposées à l'ingénieur, les éléments programmables doivent posséder plusieurs qualités :

- Ils doivent occuper une surface la plus petite possible (Ce point s'explique pour des raisons évidentes de coût. Ceci est d'autant plus vrai que l'on désire en disposer d'un grand nombre).
- Ils doivent posséder une résistance de passage faible et une résistance de coupure très élevée.
- Ils doivent apporter un minimum de capacité parasite.

Les deux derniers points s'expliquent quant à eux pour des raisons de performance en termes de fréquence de fonctionnement du PLD. Plus la résistance et la capacité sur le chemin d'un signal sont faibles, plus la fréquence de ce signal peut être élevée (RC effet). [11]

V. Les technologies à fusibles

Les fusibles sont un dispositif intégré pour protéger l'équipement électrique en cas de surtension, il présente les caractéristiques suivantes :

- Circuits de faible densité.
- La programmation permet de supprimer la connexion.
- Toutes les connexions sont établies à la fabrication.
- Son principe est d'appliquer une tension de 12 à 25v (tension de claquage) aux bornes du fusible.
- La programmation dans cette technologie est irréversible.

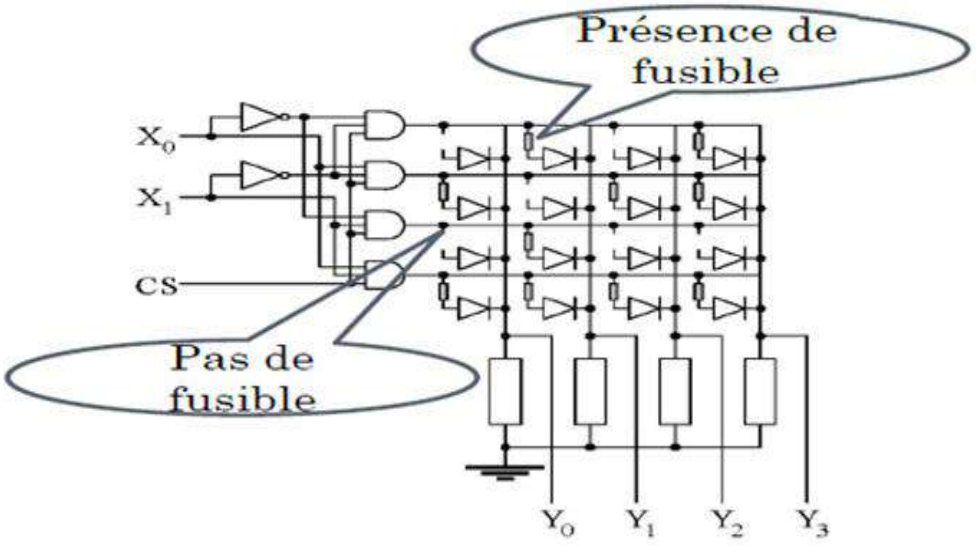


Figure 13 : les Fusibles [9]

Remarque :

Cette technologie à fusible est devenue obsolète pour des raisons de manque de fiabilité. Car le fait de "griller" les fusibles provoque des perturbations qui peuvent affecter le reste du circuit. Et en controversant le principe de la re-programmabilité cette programmation est irréversible.

VI. Les technologies à anti fusible et SRAM

Malgré qu'on n'entende plus le terme de la reprogrammation dans les circuits FPGA, il existe deux différentes classifications de FPGA soit à base d'anti fusible soit à celle d'une mémoire, la figure suivante nous montre la classification de ses dernières.

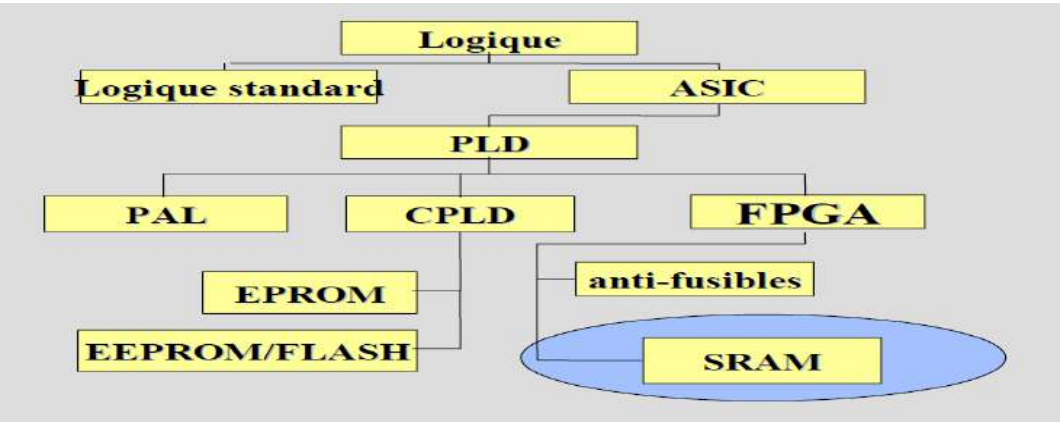


Figure 14 : Classification des technologies [9]

✓ **Les Anti-Fusibles**

Elaborer par la société ACTEL, La technique de l'anti-fusible consiste à isoler deux lignes de connexions par une fine couche d'oxyde.

Si une impulsion de haute tension (une vingtaine de volts) est appliquée à cet anti-fusible, la couche d'oxyde est trouée et les deux lignes sont reliées.

La programmation permet d'établir la connexion mais dans cette technologie est irréversible. Par déduction cette technique est non reprogrammable.

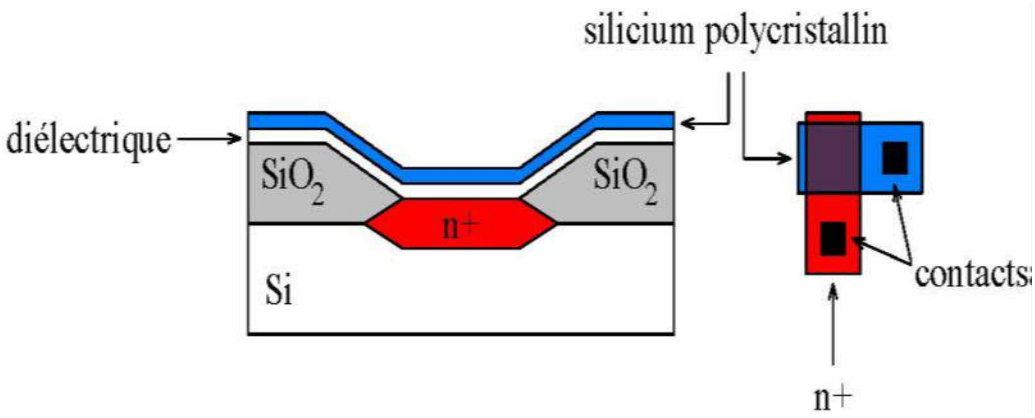


Figure 15 : Technologie de l'anti fusible [10]

✓ **Les SRAM (Static Random-Access Memory)**

Les circuits FPGA SRAM deviennent des solutions de remplacement avantageuses pour les systèmes numériques à haute intégration comme les circuits FPGA. La spécificité de cette technologie réside dans les cycles de développement et de prototypage (test et vérification en conditions réelles) qui sont accélérés ou même confondus (figure 16).

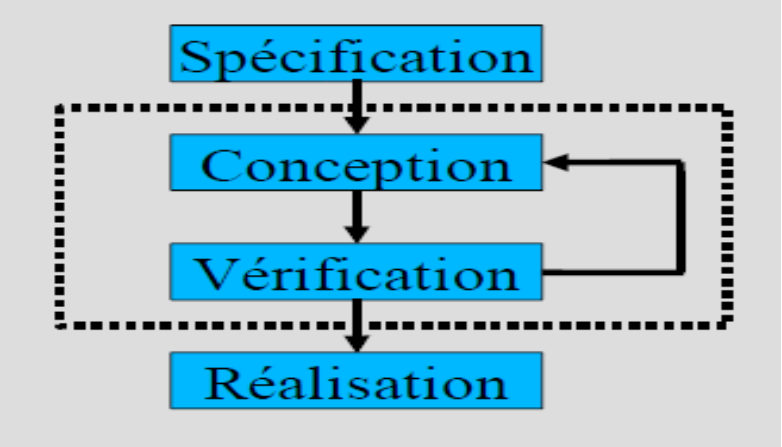


Figure 16 : technologie de FPGA SRAM [10]

VII. Les technologies à EPROM/FLASH

La mémoire flash est une mémoire de masse à semi-conducteurs réinscriptible, c'est-à-dire une mémoire possédant les caractéristiques d'une mémoire vive mais dont les données ne disparaissent pas lors d'une mise hors tension. La mémoire flash stocke dans des cellules de mémoire les bits de données qui sont conservées lorsque l'alimentation électrique est coupée.

La mémoire flash est un type d'EEPROM qui permet la modification de plusieurs espaces mémoires en une seule opération. La mémoire flash est donc plus rapide lorsque le système doit écrire à plusieurs endroits en même temps.

Il existe deux variantes de l'EPROM,

- Erasable Programmable Read Only Memory classique (EPROM)

- Electrically Erasable Programmable Read Only Memory (EEPROM).

La technologie EEPROM

- Les PLD à EPROM se programment électriquement et s'effacent aux UV, Par contre Les PLD à EEPROM se programment quasi instantanément, et gardent la configuration jusqu'à une nouvelle programmation (même en l'absence de tension)
- La technologie EEPROM est Facile et rapide à programmer, la configuration disparaît sans alimentation. [11]

VIII. Technologies utilisées par les différents fabricants

Dans les circuits programmables, il existe plusieurs fabricants qui utilisent une ou plusieurs technologies pour fonctionnées leurs propres circuits, dans le cadre de notre cours nous nous intéressons au circuits FPGA qui sont représenter par deux grande famille Altera et Xilinx mais afin de voir les différents fabriquant : Le tableau 2, nous donne les différents fabricants dans le monde qui utilise ces technologies des éléments programmables.

FABRICANTS	TECHNOLOGIES UTILISEES
ACTEL	Anti fusible, SRAM
ALTERA	EPROM, EEPROM, SRAM
AMD	EEPROM
ATMEL	SRAM
LATTICE	EPROM, EEPROM
XILINX	SRAM, Anti fusible, EPROM, EEPROM

Tableau 2 : Technologies utilisées par les différents fabricants.

Chapitre 3.
Les réseaux logiques reconfigurable FPGA

I. Introduction

Comme nous l'avons déjà évoqué dans les chapitres précédents, les éléments programmables sont à la base des caractéristiques des circuits FPGA.

Les FPGA (Field Programmable Gate Array) sont inventés par la société Xilinx en 1985. Xilinx fut précurseur du domaine en lançant le premier circuit FPGA commercial, le XC2000. Ce composant avait une capacité maximum de 1500 portes logiques. La technologie utilisée était alors une technologie aluminium à 2 micro-mètre avec 2 niveaux de métallisation. Xilinx ne sera suivi qu'un peu plus tard, et jamais lâchée, par son plus sérieux concurrent Altera qui lança en 1992 la famille de FPGA FLEX 8000 dont la capacité maximum atteignait 15 000 portes logiques.

Les réseaux logiques FPGA sont programmables et reprogrammables ou dite reconfigurable ce qui permet de leurs donné de la valeur en industrie ou chez les particuliers afin d'éviter des échecs de fabrications irréversibles.

Dans ce chapitre nous allons voir les différentes facettes des circuits FPGA à savoir son architecture, ses éléments programmables et la manière dont on programme ces éléments logiques.

II. Structure des FPGA

Le choix d'un FPGA ou des réseaux logiques programmables en générale dépendra de :

• La densité d'intégration.

• De la rapidité de fonctionnement.

• De la facilité de mise en œuvre (programmation, reprogrammation…).

• De la possibilité de maintien de l'information. [9]

A la différence des CPLD, les FPGA sont assimilables à des ASIC programmables par l'utilisateur. La puissance de ces circuits est telle qu'ils peuvent être composés de plusieurs milliers voire des millions de portes logiques et de bascules. Les dernières générations de FPGA intègrent même de la mémoire vive (RAM).

Ils sont composés de blocs logiques élémentaires (plusieurs milliers de portes) qui peuvent être interconnectés.

Critère de choix : vitesse de fonctionnement plus élevées pour les CPLD H.

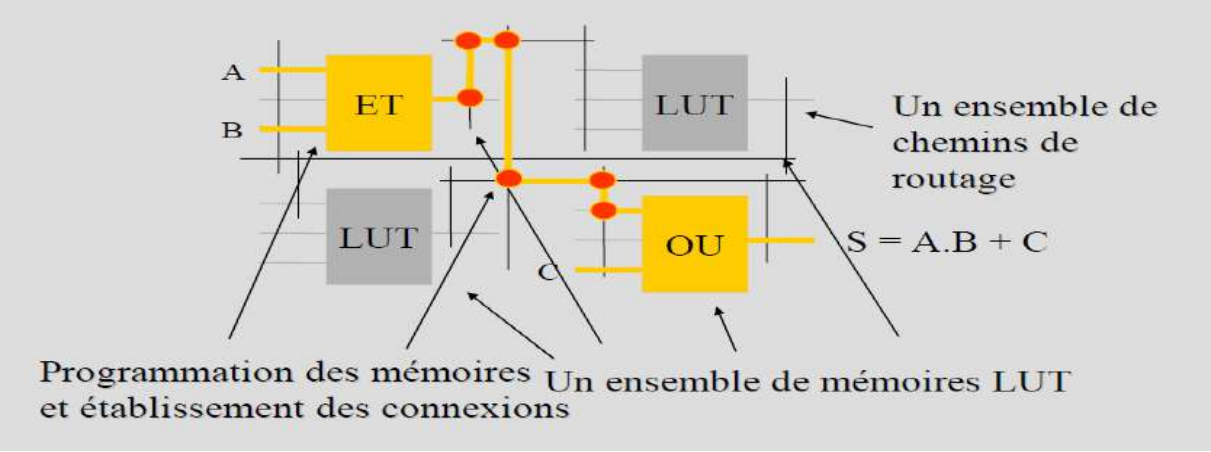


Figure 17 : Structure d'un circuit FPGA

La structure des LUT est caractérisée par une table de vérité ainsi qu'une mémoire pour router les différentes interconnexions entre les entrées et sorties des circuits FPGA.

Nous allons voir trois types de circuit FPGA à deux dimensions :

1. **Architecture de type Mer de portes**

   Elle est composée hiérarchiquement et le routage est de type logarithmique. Ce type de topologie fut utilisé par Xilinx pour sa série 6000. Mais ces composants n'ont pas eu de succès (Commercialement parlant) par manque d'outils de CAO (Conception Assistée par Ordinateur) capables de les exploiter correctement. Celle-ci n'est plus utilisée.

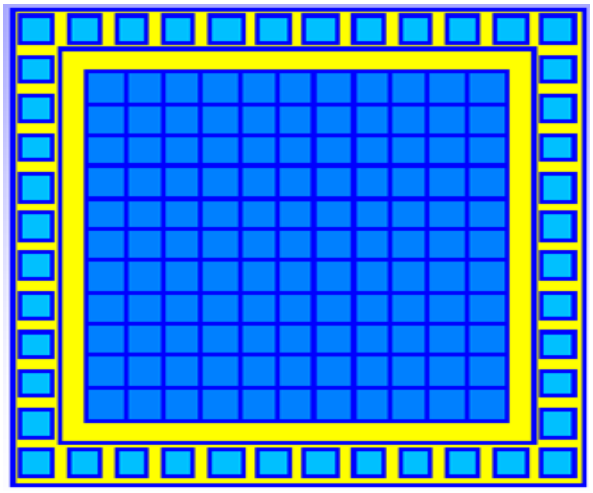


Figure 18 : Architecture Mer de porte [11]

2. **Architecture de type îlots de calcul**

   Ce type d'architecture est celui choisi dès le départ par Xilinx. Dans ce cas, le FPGA est constitué d'une matrice pleine d'éléments Ces éléments (que l'on détaillera dans la suite) constituent les ressources logiques et de routages programmables des FPGAs.

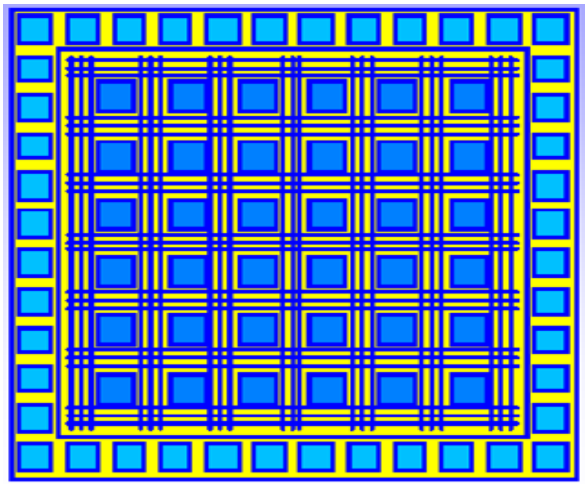


Figure 19 : Architecture îlots de calcul [11]

3. Architecture de type Hiérarchique

Cette fois il existe plusieurs plans dans le FPGA, mais ces plans ne sont pas physiques, ils correspondent aux niveaux de hiérarchie logique. C'est à dire qu'un élément d'un niveau logique peut contenir des éléments de niveau logique inférieur, d'où la notion de hiérarchie.

Chaque niveau logique reprend la topologie d'une architecture du type îlots de calcul avec un routage dédié pour chaque niveau.

Le FPGA est présenté pour la 1er fois par XILINX avec des structures :

• blocs logiques configurables.

• blocs d'I/O configurables.

• des interconnexions entre bloc configurables. [11]

La figure 20, Nous montre un exemple d'un circuit FPGA Virtex II, créer par la société XILINX.

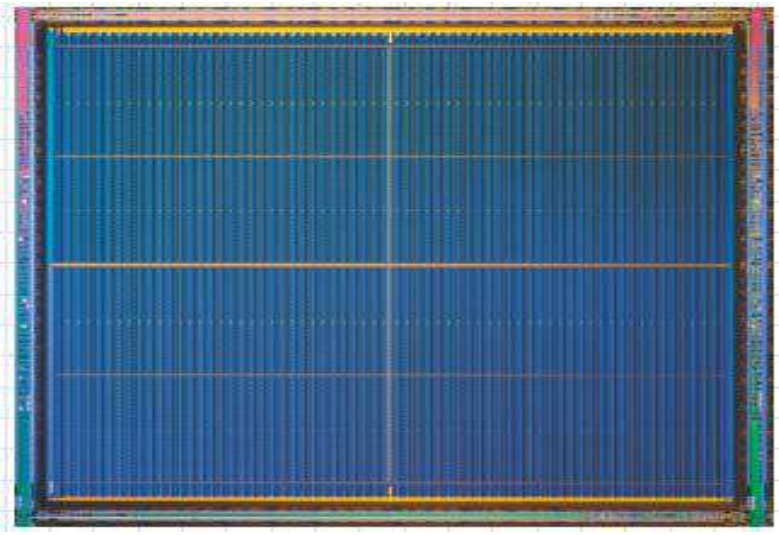


Figure 20 : Architecture de type Hiérarchique [11]

III. Architecture des circuits FPGA

1. Les blocs principaux

Un FPGA est un réseau (matrice) inspiré de l'architecture des ASIC et en complément aux circuits grossier et limiter qui sont les CPLD.

L'architecture des circuits FPGA se modélise globalement comme suit :

- Des blocs logique combinatoires et séquentiels (CLB).
- Des blocs d'entrée/sortie (IOB) sont associés aux broches du circuit.
- Les CLB et IOB sont interconnectés entre eux par des dispositifs variés.
- Les matrices s'organisent de 8x8 à 128x120. [9]

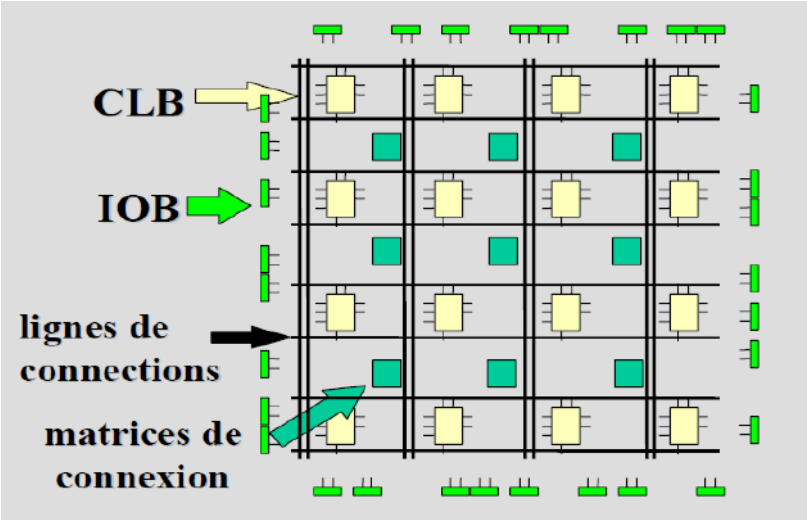


Figure 21 : Architecture générale d'un FPGA [9]

Nous allons à présent détaillé les différents éléments représenter dans la figure précédente.

✓ **Les Interconnexions**

Il existe trois type d'interconnections entre les différents blocs des circuits FPGA

- Interconnexion directe : entre les différents blocs logiques.
- Interconnexions par le biais d'une matrice.
- Interconnexion par les grandes lignes relient tous les CLB dans les extrémités des circuits FPGA.

La figure 22 nous donne un récapitulatif des trois méthodes de connexion.

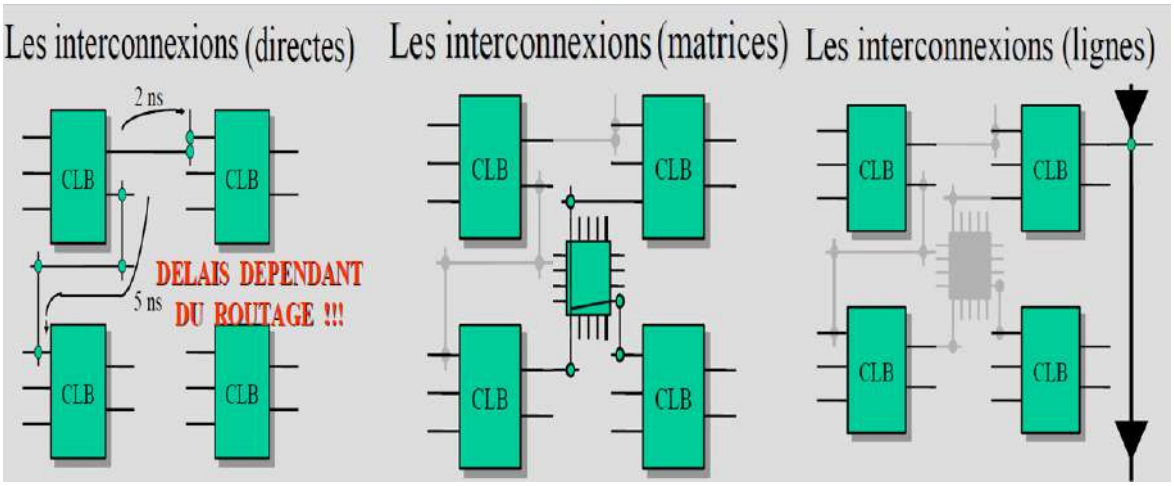


Figure 22 : Les différentes méthodes d'interconnexions entrent blocs logiques [9]

- ✓ **Structure des CLB (CONFIGURABLE LOGIC BLOCS)**

Une table de transcodage combinatoire (LUT) pouvant implanter :

– Deux fonctions indépendantes à 4 variables.

– Une fonction complète à 5 variables.

– Une fonction incomplète à 6 variables.

Elle permet aussi d'implanter, deux cellules séquentielles (bascules D) et des multiplexeurs de configuration. [9]

La figure 23, nous montre un exemple d'un CLB SPARTRAN avec ses différents composants.

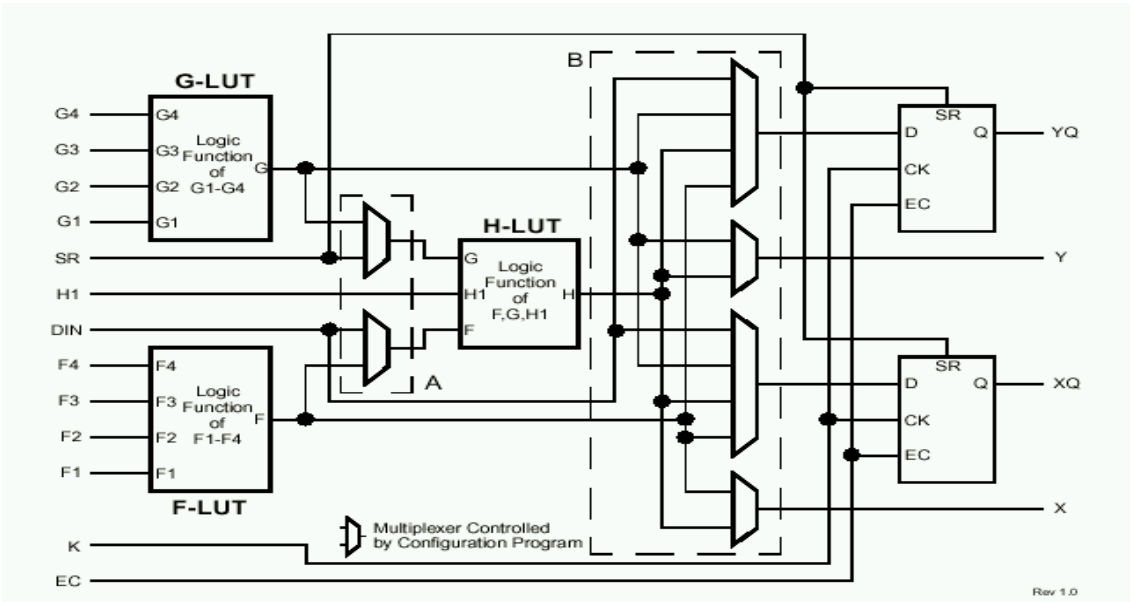


Figure 23 : Structure d'un CLB SPARTAN [9]

✓ **Structure des IOB (INPUT/OUTPUT BLOCKS)**

Les Ports d'entrées/sortie des circuits FPGA sont totalement programmables, et :

• Le seuil d'entrée est soit TTL ou CMOS.

• Le slew-rate est programmable (La vitesse de balayage ou Slew rate représente la vitesse de variation maximale que peut reproduire un amplificateur).

• Le buffer de sortie est programmable en haute impédance

• Les entrées et sorties sont directes ou mémorisées

• L'inverseur est aussi programmable. [9]

La figure 24, nous montre un exemple des IOB SPARTRAN avec ses différents composants.

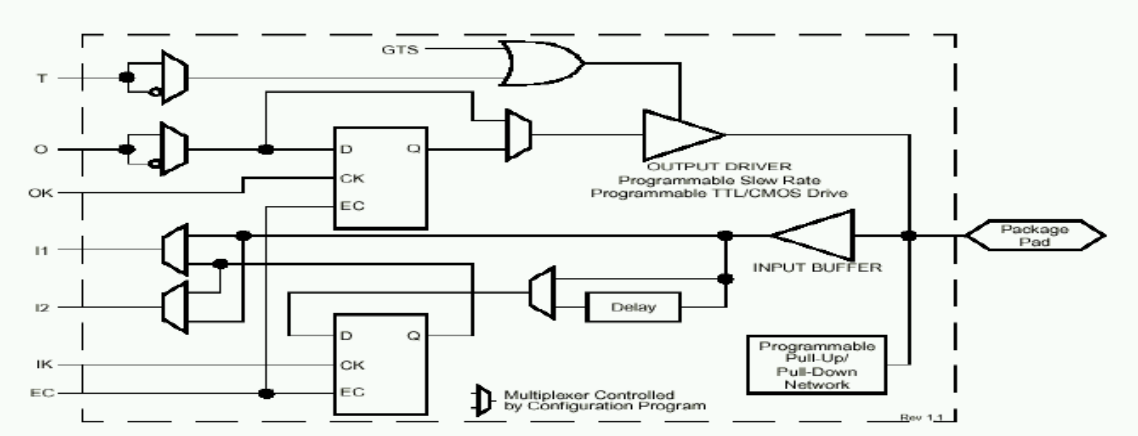


Figure 24 : Structure des IOB SPARTAN [9]

✓ **Canaux de routage**

Les canaux de routage sont présentés soit sous formes de 8 pour les interconnexions en matrices soit sous forme de doublé pour les interconnexions directes soit sous forme de 3 pour les grandes lignes comme le montre la figure suivante

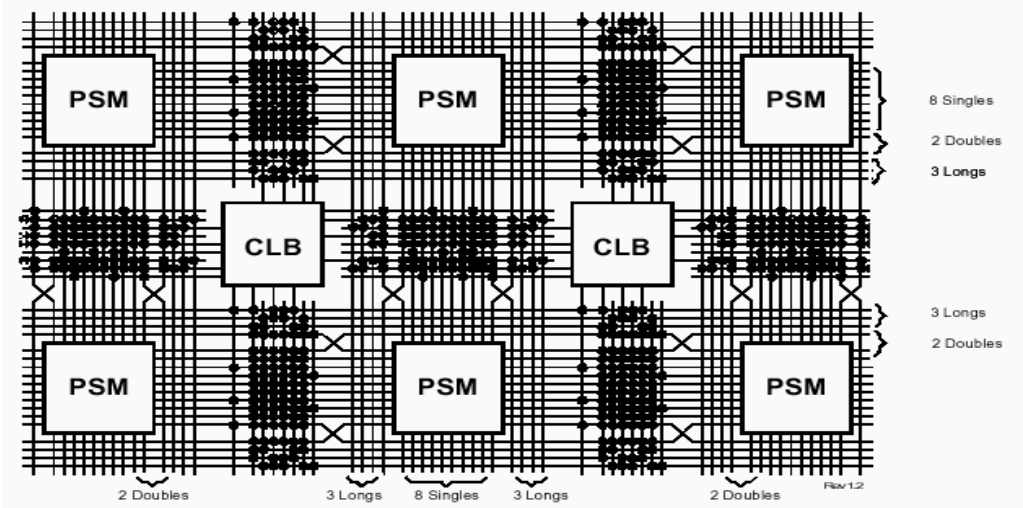


Figure 25 : canaux de routage dans un FPGA [9]

2. Les blocs à usage spécifique

En plus des blocs principaux décrits dans le paragraphe précédent, les FPGA possèdent souvent ce que l'on appelle des ressources additionnelles. La composition détaillée d'un circuit FPGA (Figure 26), illustre la disposition de ces blocs additionnels sur un FPGA. Le fonctionnement et le placement de ces blocs diffèrent d'une référence de FPGA à une autre.

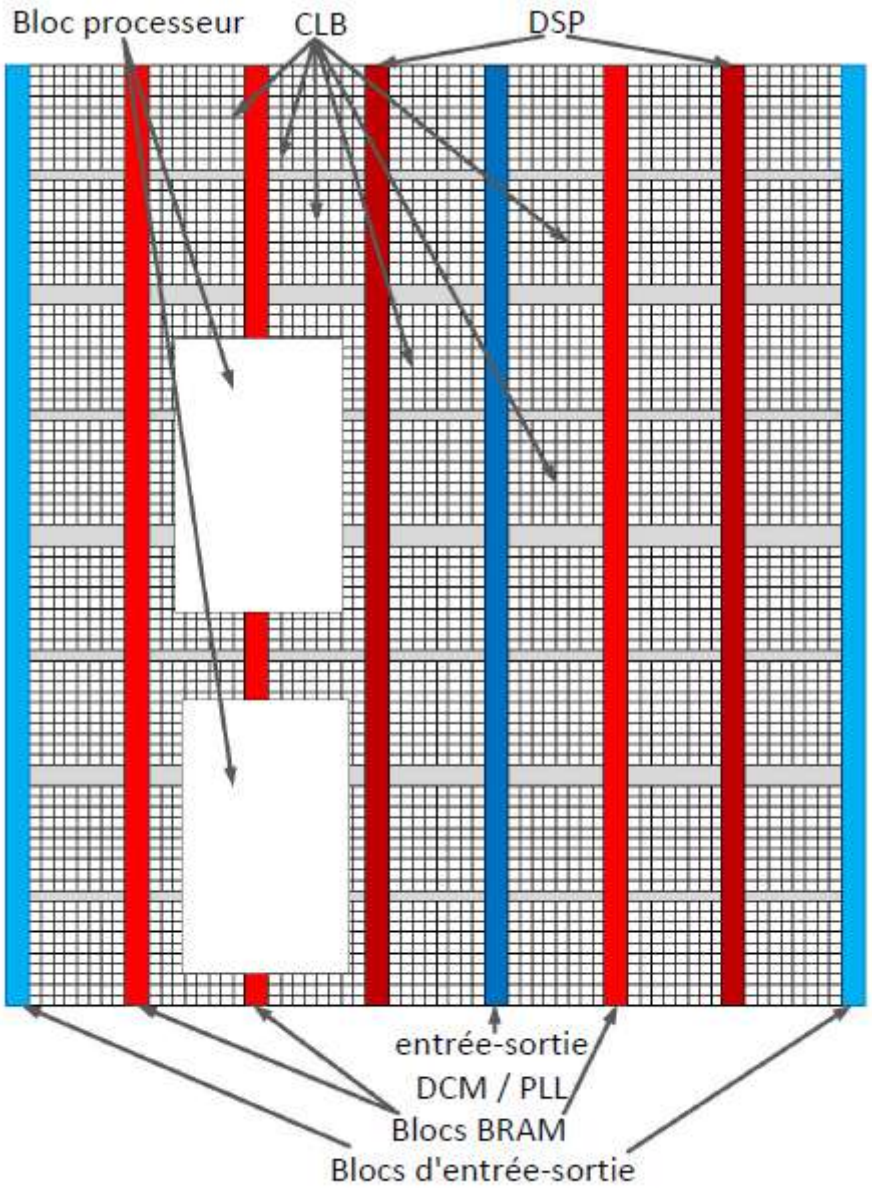


Figure 26 : Ressources d'un circuit FPGA. [14]

✓ Les blocs RAM (BRAM)

Les blocs RAM sont des mémoires définies par l'utilisateur, embarquées sur le circuit intégré FPGA, elles servent à stocker des ensembles de données. En fonction de la catégorie de FPGA, la mémoire RAM embarquée est configurable en blocs de 16 ou de 32 kilo-octets. Les mémoires RAM sont de type double ports, elles permettent une écriture et une lecture indépendante sur chaque port avec une horloge différente. Ceci est très utile, le composant peut produire (écrire) des données à une fréquence différente d'un autre composant qui consomme (lire) les données.

- ✓ **Digital Signal Processing (DSP)**

Les « Digital Signal Processing » sont des blocs qui permettent des conceptions plus complexes, qui peuvent consister soit en traitement numérique du signal ou seulement certains assortiments de multiplication, addition et soustraction. Comme pour les BRAM, il est possible de mettre en œuvre ces blocs grâce au CLB, mais il est plus efficace en termes de performances, et de consommation d'énergie d'intégrer plusieurs de ces composants au sein du FPGA. Un bloc DSP permet de réaliser un multiplicateur, un accumulateur, un additionneur, et des opérations logiques (AND, OR, NOT, et NAND) sur un bit. Il est possible de combiner les blocs DSP pour effectuer des opérations plus importantes, telles que l'addition avec virgule flottante simple, la soustraction, la multiplication, la division, et la racine carrée. Le nombre de blocs DSP est dépendant du dispositif.

- ✓ **Les processeurs embarqués**

Les processeurs embarqués enfouis sont l'un des ajouts les plus importants pour le FPGA. Beaucoup de conceptions nécessitent l'utilisation d'un processeur embarqué. Souvent, le choix d'un dispositif de FPGA avec un processeur embarqué (comme le Virtex de Xilinx 5) permet de simplifier grandement le processus de conception tout en réduisant l'utilisation des ressources et la consommation d'énergie. Le PowerPC IBM 405 et 440 processeurs sont des exemples de deux processeurs inclus dans le Virtex 4 et 5 de Xilinx. Ce sont des processeurs RISC classiques qui mettent en œuvre un jeu d'instructions PowerPC.

- ✓ **Le gestionnaire d'horloge numérique (DCM)**

Un gestionnaire d'horloge numérique permet d'avoir des périodes d'horloge différentes qui sont générées à partir d'une horloge de référence unique. La plupart des systèmes disposent d'une horloge externe unique qui produit une fréquence d'horloge fixe. Cependant, il y a un certain nombre de raisons pour lesquelles un concepteur peut avoir besoin de fonction logique fonctionnant à des fréquences différentes. L'avantage d'utiliser un DCM est que les horloges générées auront moins de gigue [14].

IV. Les éléments des FPGA

- ✓ L'horloge :
  Un élément essentiel pour le bon fonctionnement d'un système électronique. Les circuits FPGA sont prévus pour recevoir une ou plusieurs horloges.
  Des entrées peuvent être spécialement réservées à ce type de signaux. Ainsi que des ressources de routage spécialement adaptées au transport d'horloges sur de longues distances.
  L'horloge est un oscillateur à quartz : Placé dans un angle de la puce, il peut être activé lors de la phase de programmation pour réaliser un oscillateur.
  Il utilise deux IOB voisins, pour réaliser l'oscillateur. Cet oscillateur ne peut être réalisé que dans un angle de la puce où se trouve l'amplificateur prévu à cet effet.
  Il est évident que si l'oscillateur n'est pas utilisé, les deux IOB sont utilisables au même titre que les autres IOB.
- ✓ Les Critères De Choix :

- Coût de développement et fabrication : C'est le coût des dépenses engagées pour concevoir le système et réaliser les outils nécessaires à sa fabrication et son test.

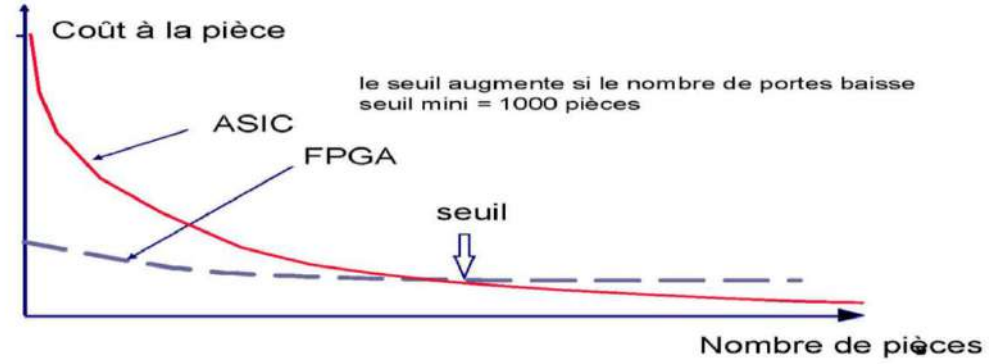


Figure 27 : Coût de développement et fabrication des technologies [9]

- Temps de développement : Le temps de fabrication à l'aide d'un circuit FPGA se résume à sa programmation ce qui est négligeable.
  SPEC = Standard Performance Evaluation Corporation (licence ou cahier de charge des systèmes informatiques).

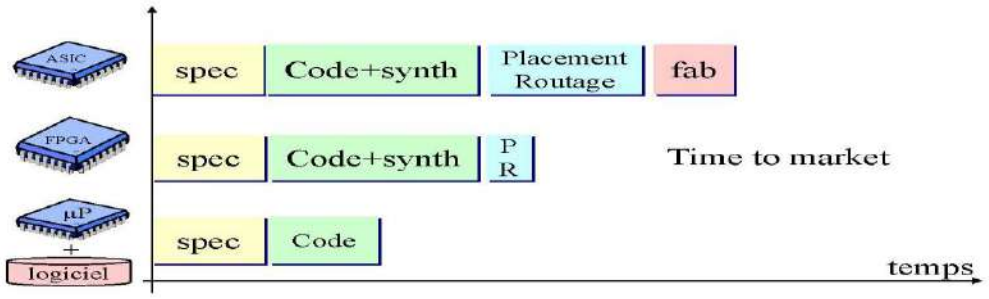


Figure 28 : temps de développement de différentes technologies [9]

- Taille : Il y a une forte dépendance entre la taille du système et la densité d'intégration. L'augmentation de la densité d'intégration produit des systèmes de taille réduite.

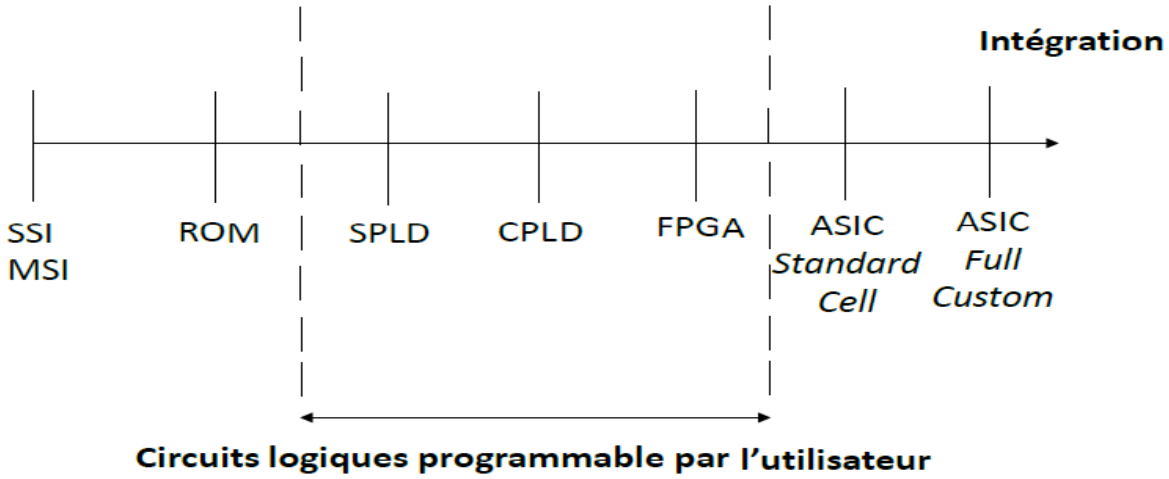


Figure 29 : Taille des différentes technologies [9]

- Souplesse d'utilisation : Favorise les circuits programmables (SPLD, CPLD, FPGA) dont on peut modifier plus facilement des fichiers que des circuits.
- Consommation : Critère particulièrement sensible dans les applications possédant une alimentation autonome. Il conduit à favoriser des solutions ASIC.
- Vitesse de fonctionnement : Les CPLD sont des composants pour la plupart reprogrammables électriquement ou à fusibles, peu chers et très rapides (fréquence de fonctionnement élevée) mais avec une capacité fonctionnelle moindre que les FPGA.
- Capacité mémoire : Les FPGA à SRAM contiennent des mémoires pour stocker leur configuration. La plupart des familles récentes offrent à l'utilisateur la possibilité d'utiliser certaines de ces mémoires en tant que telles. [9]

V. Domaines d'applications

Les applications spécifiques d'un FPGA comprennent :

- ✓ Le traitement du signal numérique,
- ✓ La bio-informatique,
- ✓ Les contrôleurs de périphériques,
- ✓ La radio logiciel restreinte (SDR),
- ✓ La logique aléatoire,
- ✓ Le prototypage ASIC,
- ✓ L'imagerie médicale,
- ✓ L'émulation de matériel informatique,
- ✓ L'intégration de plusieurs SPLD,
- ✓ La reconnaissance vocale,
- ✓ La cryptographie,
- ✓ Le filtrage et le codage de communication et bien d'autres.

Habituellement, les FPGA sont conservés pour des applications verticales particulières où le volume de production est faible. Pour ces applications à faible volume, les principales entreprises paient des coûts de matériel par unité. Aujourd'hui, la nouvelle dynamique de performance et le coût ont élargi la gamme des applications viables.

Quelques applications FPGA plus courantes sont utilisées pour : l'aérospatiale et la défense, l'électronique médicale, le prototypage des ASIC, l'audio, l'automobile, la diffusion, l'électronique grand public, les systèmes monétaires distribués, les centres de données, le calcul haute performance, les instruments industriels, domaines médicaux et scientifiques, les systèmes de sécurité, traitement d'image et de la vidéo, communications filaires et sans fil. [15]

Chapitre 4.
Méthodologie de la conception

I. Introduction

La méthodologie de conception fait référence au développement d'un système ou d'une méthode pour une situation unique. Aujourd'hui, le terme est le plus souvent appliqué aux domaines technologiques en référence à la conception de sites web, de logiciels ou de systèmes d'information [16].

Un système numérique est un assemblage sur une carte de différents composants discrets représentant chacun une fonction particulière plus ou moins complexe.

Si une erreur de conception était faite on peut soit ajouter des fils entre les composants pour certain système soit refaire une carte pour régler le problème. Et plus le système numérique est complexe, plus ces composants sont nombreux, plus la carte est chère et plus les perturbations électromagnétiques sont importantes. Un besoin existait donc de pouvoir modifier la logique sans modifier les cartes c'est la raison pour laquelle les chercheurs se sont basés sur la logique programmable dans la méthodologie de conception.

II. Méthodes de conception

La clé de la méthodologie de conception est de trouver la meilleure solution pour chaque situation de conception, que ce soit dans le design industriel, l'architecture ou la technologie. La méthodologie de conception met l'accent sur l'utilisation du brainstorming pour encourager les idées innovantes et la réflexion collaborative pour travailler sur chaque idée proposée et arriver à la meilleure solution. Répondre aux besoins et aux désirs de l'utilisateur final est la préoccupation la plus critique. La méthodologie de conception utilise également des méthodes de recherche de base, telles que l'analyse et les tests.

Le travail est dans sa première étape ; il s'agit de définir et d'évaluer les différentes phases permettant la mise en place d'un flot de conception des circuits numériques [17], pour cela on distingue deux méthodes de conception pour leurs développements :

1. La conception des circuits à faibles densités :

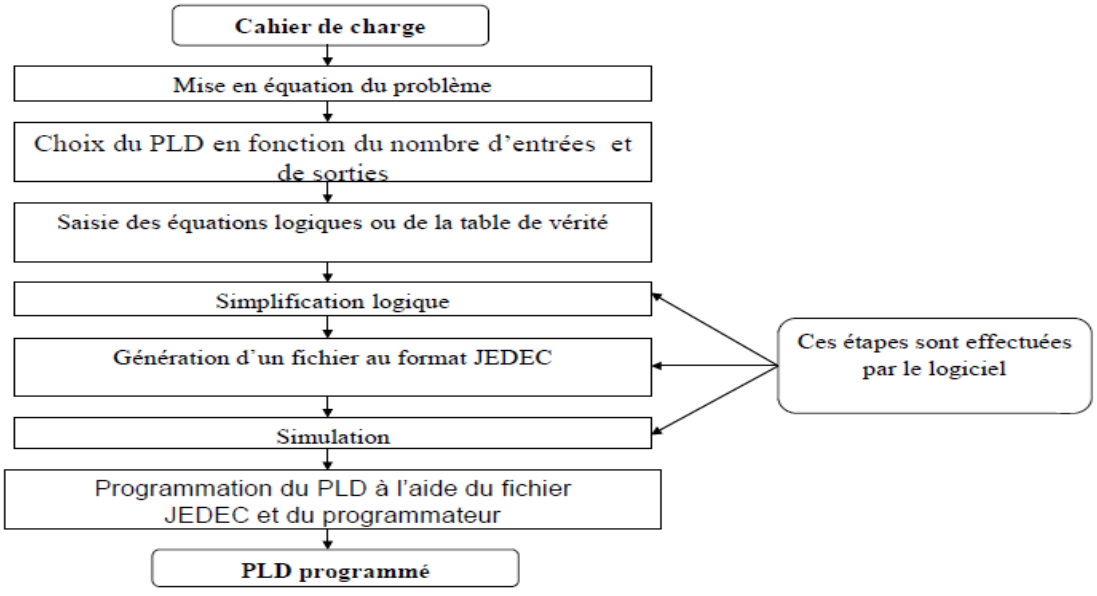


Figure 30 : Organigramme de conception des circuits à faibles densités

2. La conception des circuits à hautes densités :

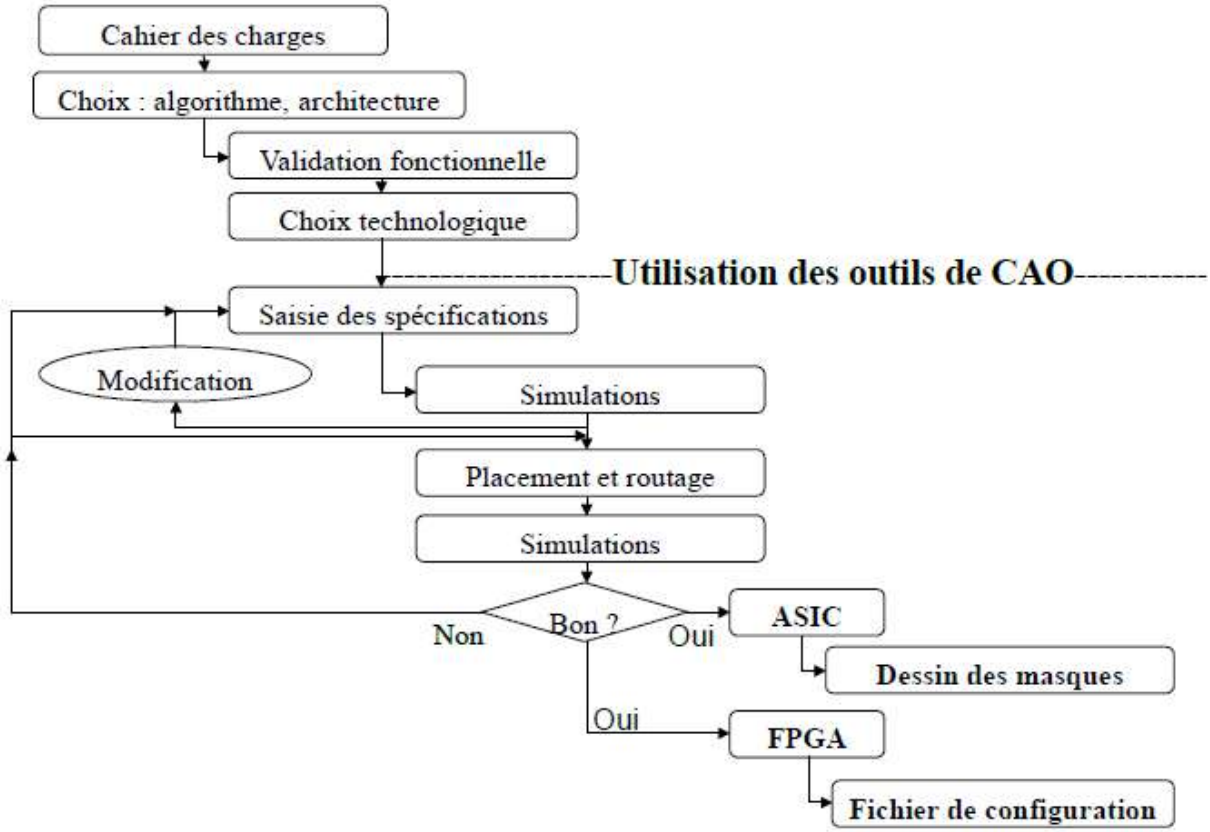


Figure 31 : Organigramme de conception des circuits à hautes densités

III. Méthodologie de conception en technologie

1) Le cahier des charges

Première Étape majeure dans la méthodologie de conception est la spécification de ce que nous devons faire et de faire un compromis entre ce qui est voulu et ce qui peut être fait.

Requières une bonne expérience pour faire des bons compromis permet :

- ✓ Les spécifications détaillées qui doivent être approuvées par l'ensemble participants au projet.
- ✓ Des modifications majeures pendant la conception qui peuvent entraîner des retards considérables.

2) La validation fonctionnelle

Bien que la méthodologie de conception soit utilisée dans de nombreuses industries, elle est couramment appliquée dans les domaines technologiques, y compris ceux qui utilisent Internet, le développement de logiciels et de systèmes d'information. Plusieurs approches méthodologiques de conception se sont développées dans l'industrie technologique. Chacun était une réaction à un type de problème différent. Certaines méthodologies de conception technologique courantes comprennent :

- ✓ **Conception Top-down (descendante) ou raffinement par étapes** : Cela commence à partir de la solution finale et fonctionne à l'envers, en affinant chaque étape en cours de route.
- ✓ **Conception Bottom-up (ascendante) :** Cette méthodologie commence par une fondation et évolue vers une solution.
- ✓ **Conception structurée :** il s'agit d'une norme de l'industrie. La technique commence par identifier les entrées et les sorties souhaitées pour créer une représentation graphique.
- ✓ **Analyse structurée et technique de conception** : Cette approche utilise un diagramme pour décrire la hiérarchie des fonctions d'un système.
- ✓ **Développement de systèmes structurés de données :** La structure des données détermine la structure du système dans cette méthodologie.
- ✓ **Conception orientée objet :** Cette méthodologie est basée sur un système d'objets en interaction.

Le concepteur commence par écrire un modèle comportemental (ou fonctionnel) du circuit en choisissant un algorithme et une architecture convenable. Le but de cette étape est de valider la partie fonctionnelle du cahier des charges en respectant toujours les contraintes du temps et de surface.

a) Conception Top – down :

Pour concevoir un modèle descendant dans une méthodologie en technologie, nous devons faire passer par les étapes suivantes :

- ✓ Choix de l'algorithme (optimisation)
- ✓ Choix l'architecture (optimisation)
- ✓ Définition des modules fonctionnels
- ✓ Définition de la hiérarchie du circuit ou système
- ✓ Diviser en petits blocs
  - o Diviser en sous blocs
- ✓ Définir les besoins en opérateurs (additionneurs, machine d'état, etc.)
- ✓ Transposer dans la technologie choisie (synthèse, schématique, layout)
- ✓ Outils de simulation comportementale (changer algorithme ou architecture si problème de vitesse ou taille puce)

b) Conception Bottom-up :

Pour concevoir un modèle ascendant dans une méthodologie en technologie, nous devons faire passer par les étapes suivantes :

- ✓ Faire des portes avec une technologie.
- ✓ Faire les blocs de base avec les portes.
- ✓ Faire des modules génériques (réutilisables)
- ✓ Assembler les modules
- ✓ Espérer que l'architecture est raisonnable
- ✓ Outils de simulation bas niveau (portes).

3) Le choix technologique

Le choix de la manière de décrire un circuit dépend de plusieurs facteurs :

- Fabriquer ce circuit avec un ASIC (Application Specific integrated circuit),
- Implémenter ce circuit sur un FPGA (Field Programmable Gate Arrays).
- La synthèse :
  - ✓ Indépendante de la technologie du circuit cible.
  - ✓ Contient la saisie de l'application dans un outil de CAO (Conception Assistée par Ordinateur).
  - ✓ Trois modes de description qui facilitent la conception de ce circuit.
    - i. Une description schématique.
    - ii. Une description HDL.
    - iii. Une description mixte.
  - ✓ La synthèse aboutit à une description du circuit au niveau logique appelée : "netlist".
  - ✓ La synthèse est validée toujours par une simulation logique.
- La première simulation :
  - ✓ Cette simulation peut être direct ou avec un banc d'essai (test bench).
  - ✓ La simulation par un fichier de test (test bench) consiste à créer un composant B sans entrées –sorties en déclarant le composant A à simuler et les signaux internes.
  - ✓ Ce banc d'essai est un module écrit en VHDL qui permet de faire varier les signaux internes connectés aux entrées afin de visualiser les résultats sur une fenêtre des courbes.
- Le placement et le routage : passe par plusieurs phases à savoir :
  - ✓ Placement des blocs.
  - ✓ Définition des régions pour mettre les rangées des cellules.
  - ✓ Placement des cellules.
  - ✓ Placement des Plots d'entrées-Sorties.
  - ✓ Routage global.
  - ✓ Alimentation, horloge, reset.
  - ✓ Routage détaillé.
  - ✓ Autres signaux.
  - ✓ Horloge (clock tree).
- Après Placement (cellules) et après routage

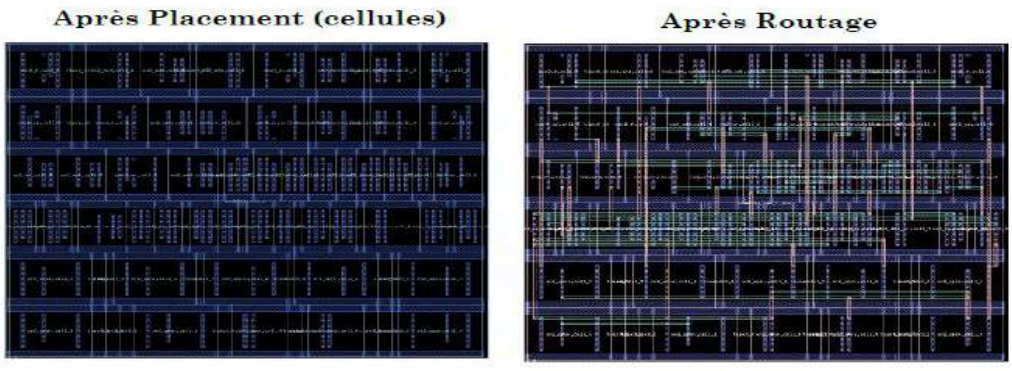


Figure 32 : Après Placement des cellules et routage [9]

- La deuxième simulation : A l'issue de ces opérations, il est possible :
  - ✓ D'extraire les retards apportés par les connexions.
  - ✓ De modifier la netlist en conséquence.
  - ✓ Cette opération s'appelle la rétro-annotation.
  - ✓ Une simulation post-routage utilisant les retards réels permet de vérifier si le circuit définitif répond bien aux contraintes du cahier des charges.
    - i. Sinon, il faut refaire la synthèse en tenant compte des résultats de la rétro-annotation.
    - ii. Si les résultats de la rétro-annotation sont parfaits, on va établir l'étape suivante !
- La fabrication ou la programmation du circuit :
  - ✓ Si c'est un ASIC:
    - i. On réalise le dessin des masques en respectant les règles du dessin afin d'établir la production du circuit.
  - ✓ Si c'est un FPGA:
    - i. Un fichier de configuration, contenant l'ensemble des informations relatives à l'implémentation, est produit.
    - ii. Celui-ci peut alors être utilisé pour configurer le circuit programmable,
    - iii. Soit à l'aide d'une mémoire non-volatile associée au circuit,
    - iv. Soit directement à partir d'une interface externe (processeur, bus, liaison parallèle, etc…).
- Un fichier de contraintes ou l'outil "PACE" de ISE 9.2i:

#PACE: Start of PACE I/O Pin Assignments

NET "CLK" **LOC** = "B8" ;

NET "RESET" **LOC** = "B18" ;

NET "D" **LOC** = "D18" ;

NET "G" **LOC** = "E18" ;

NET "SYNC_H" **LOC** = "T4" ;

NET "SYNC_V" **LOC** = "U3" ;

NET "JEU<0>" **LOC** = "U5" ;

NET "JEU<1>" **LOC** = "U4" ;

NET "JEU<2>" **LOC** = "N8" ;

NET "JEU<3>" **LOC** = "P8" ;

NET "JEU<4>" **LOC** = "P6" ;

NET "JEU<5>" **LOC** = "R9" ;

NET "JEU<6>" **LOC** = "T8" ;

NET "JEU<7>" **LOC** = "R8" ;

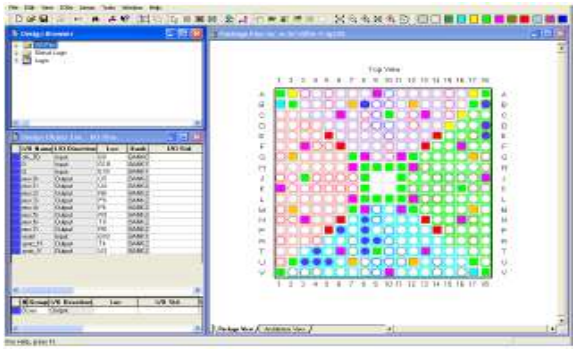


Figure 33: Start of PACE I/O Pin Assignments [9]

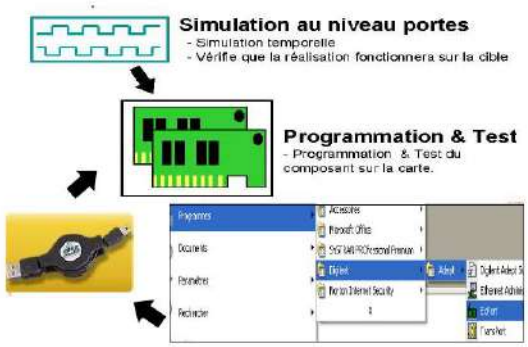


Figure 34: Simulation et synthèse [9]

IV. Les outils de développement

Les outils de développement servent à simplifier l'implémentation d'un nouveau système dans une solution existante ou en pleine création. Il s'agit d'un outil indispensable pour faciliter les tâches qui doivent être assurées pour la méthodologie de conception.

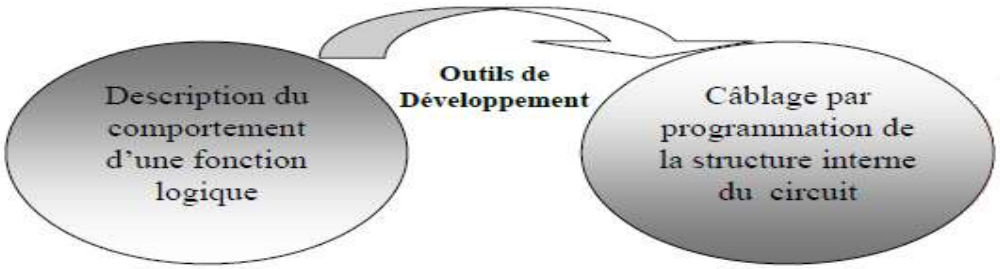


Figure 35 : Outils de développement

Il existe trois principaux outils de développement :

- ✓ Les outils de CAO (Conception Assistée par Ordinateur).
- ✓ Les différentes approches de description d'un circuit.
- ✓ Les langages de description **(déjà détaillé dans le chapitre 1, les différents HDLs).**

Remarque : Les concepteurs réalisent et testent les circuits sur ordinateur avant de lancer la fabrication.

1) Les outils de CAO (Conception Assistée par Ordinateur)

On peut distinguer trois types d'outils de CAO selon les modes de descriptions :

- ✓ Un outil de CAO utilisé que pour la synthèse et la simulation (comme **MODELSIM**).
- ✓ Un outil de CAO qui utilise le mode schématique et textuel (comme **ISE** de **XILINX**).
- ✓ Un outil de CAO qui peut utiliser le mode schématique, textuel et permet même le dessin du masque (layout) (comme **CADENCE**).

Les outils de synthèse et simulation sont représenter par 3 objectifs :

- ✓ Minimiser la surface de silicium.
- ✓ Minimiser les temps de propagation.
- ✓ Minimiser la consommation.

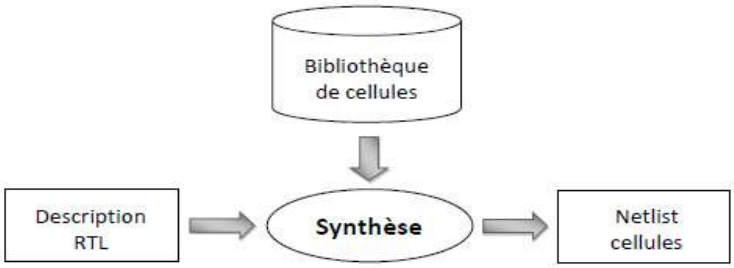


Figure 36 : Outils de synthèse et simulation [9]

Les outils de placement et routage est quant à eux représenter par 2 objectifs :

- ✓ Minimiser la surface globale du silicium.
- ✓ Minimiser les longueurs d'interconnexions.

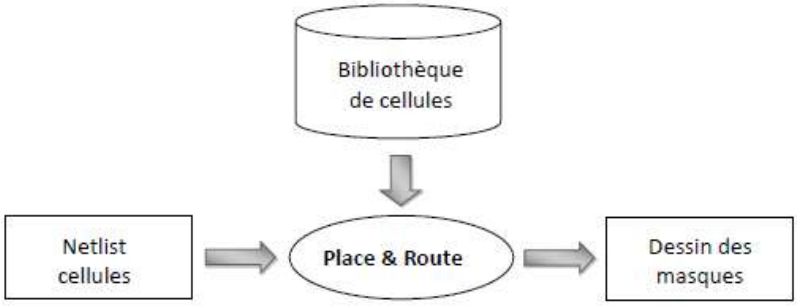


Figure 37 : Outils de placement et routage [9]

2) Les différentes approches de description d'un circuit

Il existe trois différentes approches de description d'un circuit :

i. Description physique (Dessin au micron)

Dans la première époque, la conception de circuits intégrés était un métier manuel. On dessinait les composants (transistors) à la main, sur un papier spécial (Mylar) avec des crayons de couleur. C'est ce qu'on appelle le dessin au micron.

Une telle technique limitait naturellement la complexité des dispositifs conçus, mais avec l'apparition des différents outils de CAO tel que CADENCE, le dessin des circuits complexes est devenu plus facile.
Pour cela, il y a un certain nombre de contraintes à respecter lors du dessin : les règles de dessin comme le montre la figure 38.

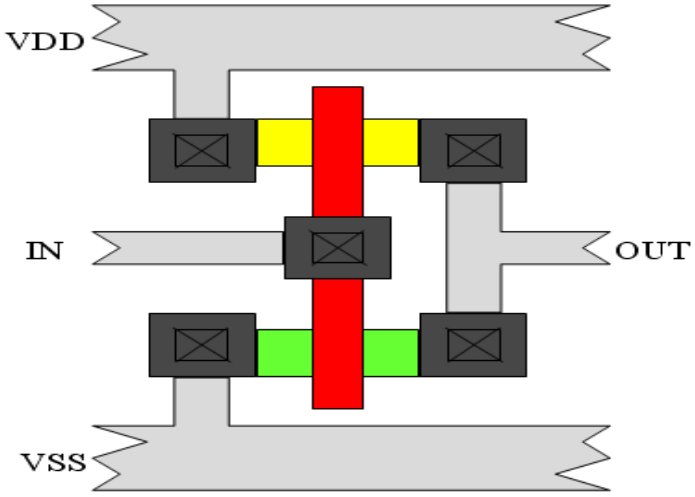


Figure 38 : Dessin au micron [9]

ii. Description schématique

L'apparition des interfaces graphiques et donc des éditeurs de schémas font la naissance d'une nouvelle description dans la réalisation des circuits qui est le schématique.

On peut comprendre facilement la description schématique du composant que son dessin de masque.

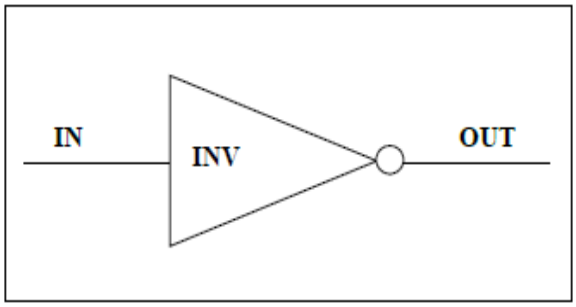


Figure 39 : Dessin schématique

iii. Description textuelle

Grâce aux nouvelles possibilités de description à un niveau d'abstraction plus élevé, les langages fonctionnels (ou comportementaux) de description de matériel ont répondu à des besoins fondamentaux des concepteurs de circuits intégrés :

- La réduction des temps de conception.
- L'accélération des simulations qui devenaient prohibitives avec l'accroissement de complexité des dispositifs.

- La normalisation des échanges : le monde de l'électronique a souffert pendant longtemps de la multiplicité des langages, des formats, des outils de CAO utilisés.
- La fiabilité : les langages HDL sont conçus pour limiter en principe les risques d'erreur.
- La portabilité : les langages normalisés sont très largement portables.
- La réutilisabilité : les modifications et adaptations sont rendues plus simples donc moins risquées et moins coûteuses.

```
Entity INV is
port (IN   : in   STD_LOGIC;
      OUT: out STD_LOGIC);
End INV;
Architecture A_INV of INV is
Begin
OUT <= not (IN);
End A_INV;
```

Figure 40 : Dessin Textuelle

V. Méthodologie Actuelle

La méthodologie actuelle repose essentiellement sur deux axes :

1) Les bibliothèques.
2) Les outils CAO.

Pour les bibliothèques, il existe trois types de composants :

- ✓ Portes logiques pré caractérisées (conçues pour placement/routage automatique).
- ✓ Blocs réguliers macro-générés (mémoires, opérateurs arithmétiques, PLA, …).
- ✓ Composants virtuels : IP core (cœurs de microprocesseurs, coprocesseurs spécialisés, ..).

Pour les outils CAO ou Composants virtuels :

- ✓ Soft Core : Description RTL synthétisable.
- ✓ Firm Core : Netlist de porte logique.
- ✓ Hard Core : Dessin des masques.

Remarque : Les bibliothèques de composants virtuels sont un élément essentiel de Concurrence entre les fondeurs de silicium.

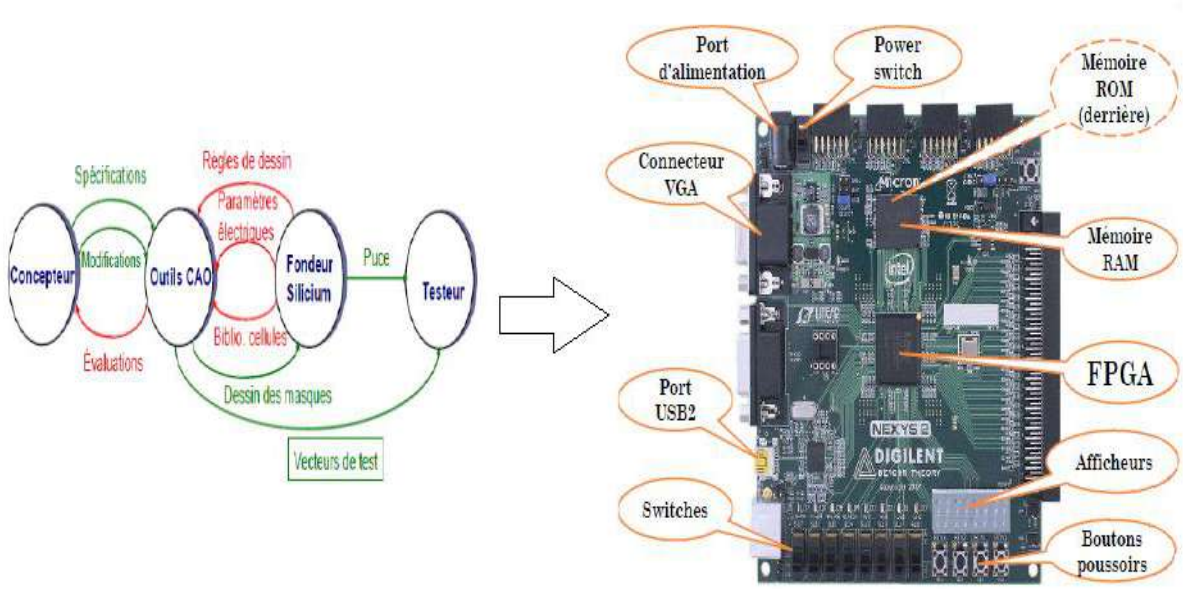


Figure 41 : Méthodologie & acteurs de la conception [9]

Chapitre 5.
Les opérateurs câblés

I. Introduction

Quelle que soit la nature de l'information traitée par un ordinateur (image, son, texte, vidéo), elle l'est toujours représentée sous la forme d'un ensemble de nombres binaires

Une information élémentaire correspond à un chiffre binaire (0 ou 1) appelé bit. Le terme bit signifie « binary digit ».

Le codage de l'information permet d'établir une correspondance entre la représentation externe de l'information et sa représentation binaire.

Les machines numériques utilisent le système binaire. C'est facile de représenter ces deux symboles (0 et 1) dans les machines numériques. Le 0 et le 1 sont représentés par deux tensions.

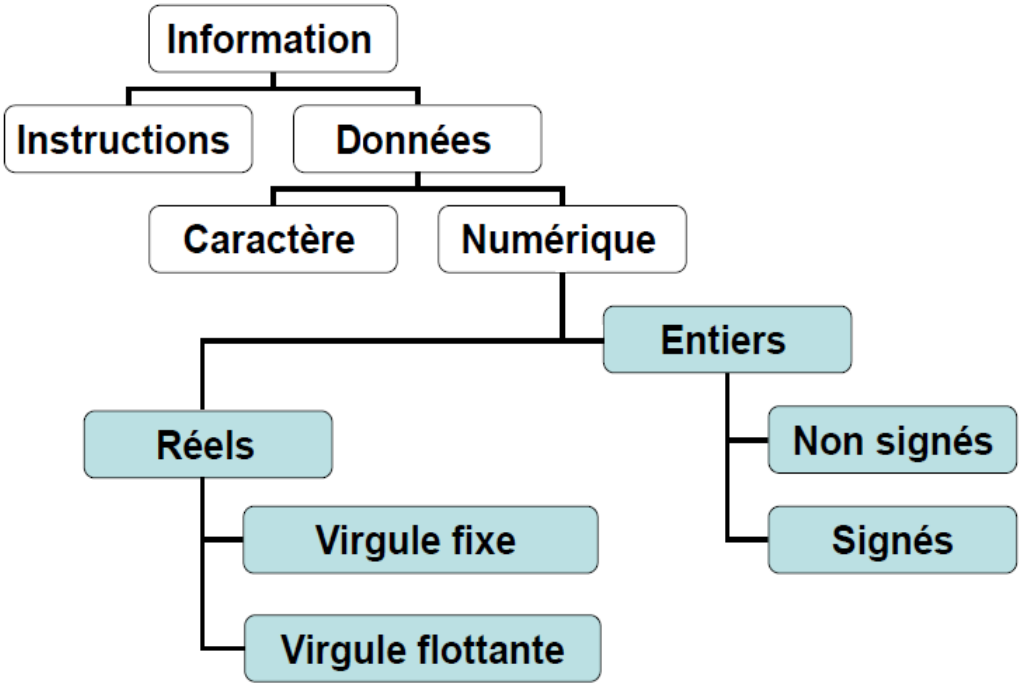


Figure 42 : Représentation de l'information

II. Représentation des nombres entiers

Il existe deux types d'entiers en informatique :

- ✓ Les entiers non signés (positifs). (Prérequis)
- ✓ Les entiers signés (positifs ou négatifs).

Le problème qui se pose est : comment indiquer à la machine qu'un nombre est négatif ou positif ?

Il existe 3 méthodes pour représenter les nombres négatifs :

- ✓ Signé/ valeur absolue.
- ✓ Complément à 1(complément restreint).
- ✓ Complément à 2 (complément à vrai).

1) Représentation signé/valeur absolue

Une valeur signée peut avoir une valeur positive ou négative et pour distinguer entre ses deux valeurs, nous procédons comme suite :

Si nous avons n bits, alors le bit du poids fort est utilisé pour indiquer le signe :

- ✓ 1 : signe négatif.
- ✓ 0 : signe positif.

Les autres bits (n-1) désignent la valeur absolue du nombre.

La figure 43, nous montre un exemple de représentation signé/valeur absolue.

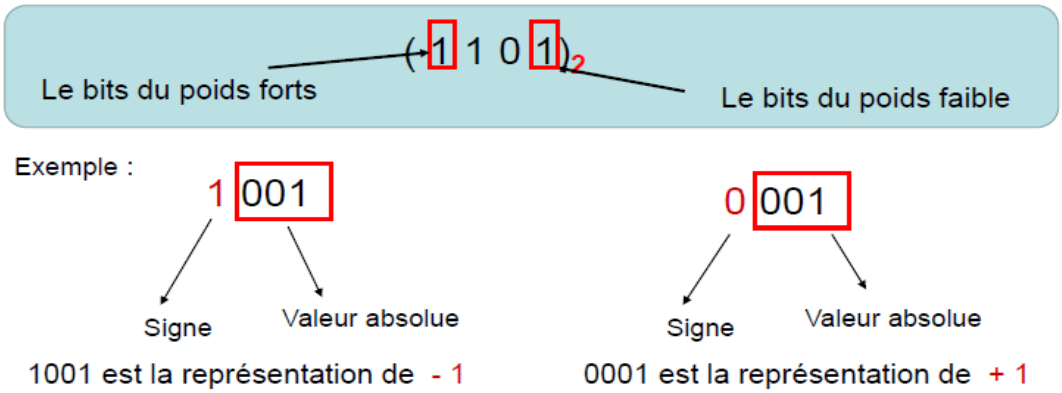


Figure 43 : Représentation Signé / Valeur absolue

Sur 3 bits, Les valeurs sont comprises entre -3 et +3 comme le montre le tableau 3.

$-3 \leq N \leq +3$

$-(4-1) \leq N \leq +(4-1)$

$-(22-1) \leq N \leq +(22-1)$

$-(2(3-1)-1) \leq N \leq +(2(3-1)-1)$

Signe	Valeur Absolue	Valeur
0	00	+0
0	01	+1
0	10	+2
0	11	+3
1	00	-0
1	01	-1
1	10	-2
1	11	-3

Tableau 3 : Intervalle des valeurs sur 3 bits signés

Sur n bits, l'intervalle des valeurs qu'on peut représenter en système en valeur absolue :

$-(2(n-1)-1) \leq N \leq +(2(n-1)-1)$.

Avantages et inconvénients :

- ✓ Représentation assez simple.
- ✓ Le zéro possède deux représentations +0 et -0 ce qui conduit à des difficultés au niveau des opérations arithmétiques.
- ✓ Pour les opérations arithmétiques il nous faut deux circuits : l'un pour l'addition et le deuxième pour la soustraction.

L'idéal est d'utiliser un seul circuit pour faire les deux opérations, puisque : X-Y = X + (-Y)

2) Représentation en complément à un

On appelle le complément à un d'un nombre N un autre nombre N'tel que : N+N'=2n -1

n : est le nombre de bits de la représentation du nombre N.

Exemple :

Soit N=1010 sur 4 bits donc son complément à un de N :

N'= (24- 1) - N

$N' = (16\text{-}1)_{10} - (1010)_2 = (15)_{10} - (1010)_2 = (1111)_2 - (1010)_2 = 0101$

1 0 1 0

+ 0 1 0 1

= 1 1 1 1

Sur n bits, l'intervalle des valeurs qu'on peut représenter en complément à 1 (CA1) :

$-(2(n-1) - 1) \leq N \leq +(2(n-1) - 1)$

Sur 3 bits, Dans cette représentation (Tableau 4), le bit du poids fort nous indique le signe : 0 : positif, 1 : négatif).

Valeur en CA1	Valeur en binaire	Valeur en décimal
000	000	+0
000	001	+1
000	010	+2
000	011	+3
100	-011	-3
101	-010	-2
110	-001	-1
111	-000	-0

Tableau 4 : Complément à 1 sur 3 bits signés

Remarque :

Dans cette représentation le zéro possède aussi une double représentation (+ 0 et – 0).

Exemple 1 :

Quelle est la valeur décimale représentée par la valeur 101010 en complément à 1 sur 6 bits ?

Solution :

- ✓ Le bit poids fort indique qu'il s'agit d'un nombre négatif.
- ✓ Valeur = - CA1(101010) = - (010101)2= - (21)10

3) Représentation en complément à 2

On représente un complément à 2 (CA2) par le principe suivant :

- ✓ Soit X un nombre sur n bits alors :

$X + 2^n = X$ modulo 2^n

Le résultat sur n bits est la même valeur que X :

$X + 2^n = X$

Exemple : soit X = 1001 sur 4 bits

$2^4 = 10000$

1 0 0 1

+ 1 0 0 0 0

= 1 1 0 0 1

Donc Si on prend le résultat sur 4 bits on trouve la même valeur de X = 1001.

- ✓ Si on prend deux nombres entiers X et Y sur n bits, on remarque que la soustraction peut être ramener à une addition :

$X - Y = X + (-Y)$ ce qui équivaux à trouver une valeur équivalente à -Y ?

$X - Y = (X + 2^n) - Y = X + (2^n - 1) - Y + 1$

On a $Y + CA1(Y) = 2^n - 1$ donc $CA1(Y) = (2^n - 1) - Y$ On obtient : $X - Y = X + CA1(Y) + 1$.

La valeur CA1(Y)+1 s'appelle le complément à deux de b : CA1(Y)+1 = CA2(Y)

Et enfin on va obtenir : X -Y = X + CA2(Y) ce qui implique de transformer la soustraction en une addition.

- ✓ Si on travaille sur 3 bits : on remarque que les valeurs sont comprises entre -4 et +3 comme le montre le tableau 5 :

$-4 \leq N \leq +3$

$-4 \leq N \leq +(4-1)$

$-22 \leq N \leq +(22-1)$

$-2(3-1) \leq N \leq +(2(3-1)-1)$

Valeur en CA2	Valeur en binaire	Valeur en décimal
000 001 010 011	000 001 010 011	+0 +1 +2 +3
100 101 110 111	-100 -011 -010 -001	-4 -2 -1 -0

Tableau 5 : Complément à 2 sur 3 bits signés

- ✓ Si on travaille sur n bits, l'intervalle des valeurs qu'on peut représenter en CA2 : $-(2^{(n-1)}) \leq N \leq +(2^{(n-1)}-1)$

Dans cette représentation, le bit du poids fort nous indique le signe. On remarque que le zéro n'a pas une double représentation.

Exemple 2 :

Quelle est la valeur décimale représentée par la valeur 101010 en complément à deux sur 6 bits ?

Le bit poids fort indique qu'il s'agit d'un nombre négatif.

Valeur = **-CA2(101010)** = -**(010101 + 1)** = -**(010110)$_2$= -(22)**

Exemple 3 :

Effectuer les opérations suivantes sur 5 Bits.

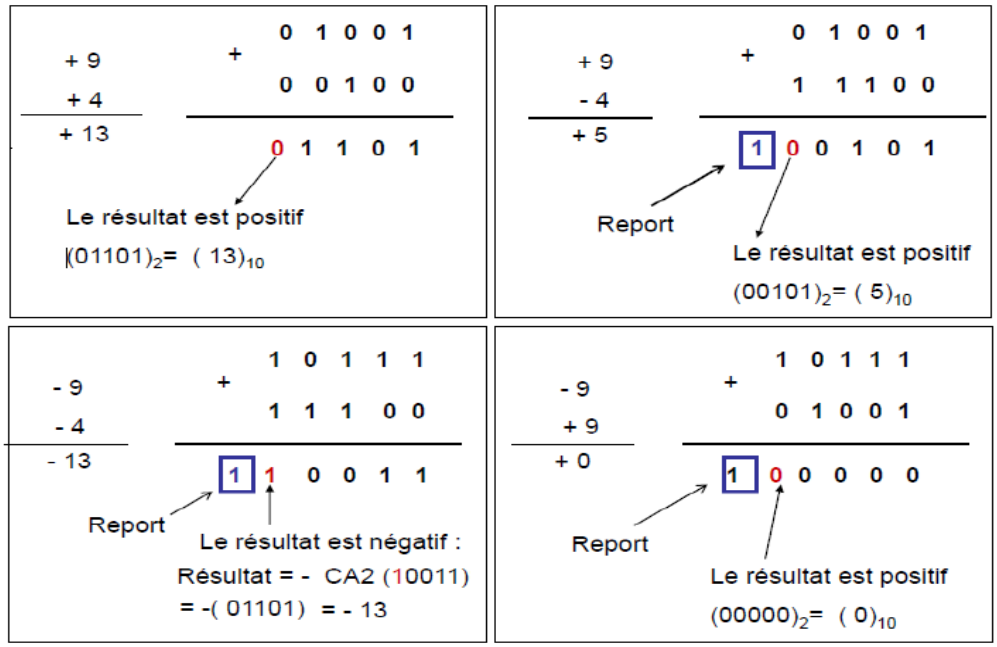


Figure 44 : Exemple de complément à 2 sur 5 bits signés

4) La retenue et le débordement

- ✓ On dit qu'il y a une retenue si une opération arithmétique génère un report
- ✓ On dit qu'il y a un débordement (Over Flow) ou dépassement de capacité : si le résultat de l'opération sur n bits et faux.
  - ❖ Le nombre de bits utilisés est insuffisant pour contenir le résultat.
  - ❖ Autrement dit le résultat dépasse l'intervalle des valeurs sur les n bits utilisés.

Cas de débordement :

- ✓ Si la somme de deux nombres positifs donne un nombre négatif.
- ✓ Ou la somme de deux nombres négatifs donne un Nombre positif.
- ✓ Il n'y a jamais un débordement si les deux nombres sont de signes différents.

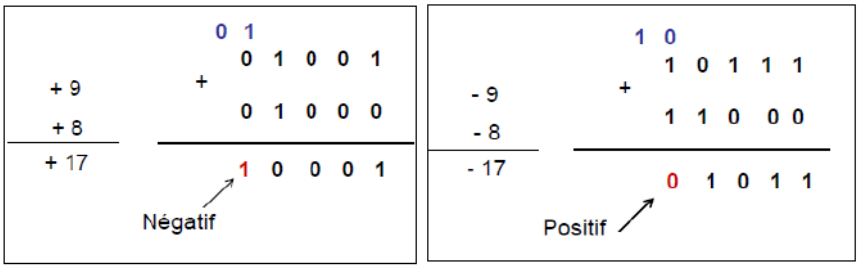


Figure 45 : Exemple de Débordement

III. Représentation des nombres réels

Un nombre réel est constitué de deux parties : la partie entière et la partie fractionnelle (les deux parties sont séparées par une virgule)

Le problème qui se pose est : comment indiquer à la machine la position de la virgule ?

Il existe deux méthodes pour représenter les nombres réels :

- ✓ Virgule fixe : la position de la virgule est fixe.
- ✓ Virgule flottante : la position de la virgule change (dynamique).

1) Représentation en virgule fixe

Dans cette représentation la partie entière est représentée sur n bits et la partie fractionnelle sur p bits, en plus un bit est utilisé pour le signe.

•Exemple : si n=3 et p=2 on va avoir les valeurs suivantes représenter par le tableau 6.

Signe	Partie entière	Partie fractionnelle	Valeur
0	000	00	+0,0
0	000	01	+0,25
0	000	10	+0,5
0	000	11	+0,75
0	001	.00	+1,0
.	.	.	.
.	.	.	.

Tableau 6 : Représentation en virgule fixe

Remarque : Dans cette représentation les valeurs sont limitées et nous n'avons pas une grande précision.

2) Représentation en virgule flottante

✓ Chaque nombre réel peut s'écrire de la façon suivante :

$N = \pm M * b^e$

Avec **M : M**antisse**, b :** la **b**ase **et e :** l'**e**xposant

Exemple :

$13,11 = 0,1311 * 10^{+2}$

$-(110,101)_2 = -(0,110101)_2 * 2^{+3}$

$(0,00101)_2 = (0,101)_2 * 2^{-2}$

Remarque :

On dit que la mantisse est normalisée, si le **premier chiffre après la virgule** est différent de **0** et **le premier chiffre avant la virgule** est égale à **0**.

✓ Dans cette représentation sur n bits :
- ❖ La mantisse est sous la forme signe/valeur absolue :
  - **1 bit** pour le signe.
  - **K bits** pour la valeur.
- ❖ L'exposant (positif ou négatif) est représenté sur p bits.

Signe mantisse	Exposant	Mantisse normalisée
1 bit	p bits	k bits

✓ Pour la représentation de l'exposant on utilise :
- ❖ Le complément à deux.
- ❖ Exposant décalé ou biaisé.

a) Représentation de l'exposant en complément à deux

Par exemple on veut représenter les nombres $(0,015)_8$ et $-(15,01)_8$ en virgule flottante sur une machine ayant le format suivant :

Signe mantisse	Exposant en CA2	Mantisse normalisée
1 Bit	4 Bits	8 Bits

$(0,015)_8 = (0,000001101)_2 = 0,1101 * 2^{-5}$

Signe mantisse : positif (0)

Mantisse normalisée : 0,1101

Exposant = -5 pour utiliser le complément à deux pour représenter le -5 sur 4 bits
CA2(0101)=1011

0	1011	11010000
1 BIT	4 BITS	8BITS

$-(15,01)_8 = -(001101,000001)_2 = -0,1101000001 * 2^4$

Signe mantisse : négatif (1)

Mantisse normalisée : 0,1101000001

Exposant= 4, en complément à deux il garde la même valeur (0100)

On remarque que la mantisse est sur 10 bits (1101 0000 **01**), et sur la machine **seulement 8 bits** sont utilisés pour la mantisse. Dans ce cas on va prendre **les 8 premiers bits de la mantisse**

0	0100	11010000
1 BIT	4 BITS	8 BITS

Remarque :

Si la mantisse est sur k bits et si elle est représentée sur la machine sur k' bits tel que k> k', alors la mantisse sera tronquée : on va prendre uniquement k' bits perdre dans la précision.

b) Représentation de l'exposant décalé (biaisé)

✓ En complément à 2, l'intervalle des valeurs qu'on peut représenter sur p bits :

$-2^{(p-1)} \leq N \leq 2^{(p-1)} - 1$

Si on rajoute la valeur $2^{(p-1)}$ à tous les termes de cette inégalité :

$-2^{(p-1)} + 2^{(p-1)} \leq N + 2^{(p-1)} \leq 2^{(p-1)} - 1 + 2^{(p-1)}$

$0 \leq N + 2^{(p-1)} \leq 2^p - 1$

✓ On pose $N' = N + 2^{(p-1)}$ donc : $0 \leq N' \leq 2^p - 1$

Dans ce cas on obtient des valeur positives.

✓ La valeur 2^{p-1} s'appelle le biais ou le décalage.

✓ Avec l'exposant biaisé on a transformé les exposants négatifs à des exposants positifs en rajoutons à l'exposant la valeur 2^{p-1}.

Exposant Biaisé = Exposant réel + Biais.

Exemple

On veut représenter les nombres $(0,015)_8$ et $-(15, 01)_8$ en virgule flottante sur une machine ayant le format suivant :

Signe mantisse	**Exposant décalé**	**Mantisse normalisée**
1 BIT	4 BITS	11 BITS

$(0,015)_8 = (0,000001101)_2 = 0,1101 * 2^{-5}$

Signe mantisse : positif (0).

Mantisse normalisée : 0,1101**.**

Exposant réel = -5.

Calculer le biais : $b= 2^{4-1} = 8$.

Exposant Biaisé = $-5 + 8 = +3 = (0011)_2$.

0	0011	11010000000
1 BIT	**4 BITS**	**11 BITS**

$-(15,01)_8=-(001101,000001)_2= -0,1101000001 * 2^4$

Signe mantisse : négatif (1)

Mantisse normalisée : 0,1101000001

Exposant réel = + 4

Calculer le biais : $b= 2^{4-1}= 8$

Exposant Biaisé = $4 + 8 = +12 = (1100)_2$

1	1100	11010000010
1 BIT	**4 BITS**	**11 BITS**

Chapitre 6.

Etude d'un exemple de FPGA - SPARTAN3E

I. Introduction

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

Le schéma ci-dessous représente un exemple de descriptions VHDL ou de blocs fonctionnels implantés dans un FPGA. 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 est réalisée par le logiciel placement/routage, au cours de laquelle les entrées et sorties seront affectées à des numéros de broches.

On peut remarquer sur le schéma la fonction particulière du bloc VHDL N°5. En effet dans la description fonctionnelle d'un FPGA on a souvent besoin d'une fonction qui sert à cadencer le fonctionnement de l'ensemble, celle-ci est très souvent réalisée par une machine d'états synchronisée par une horloge.

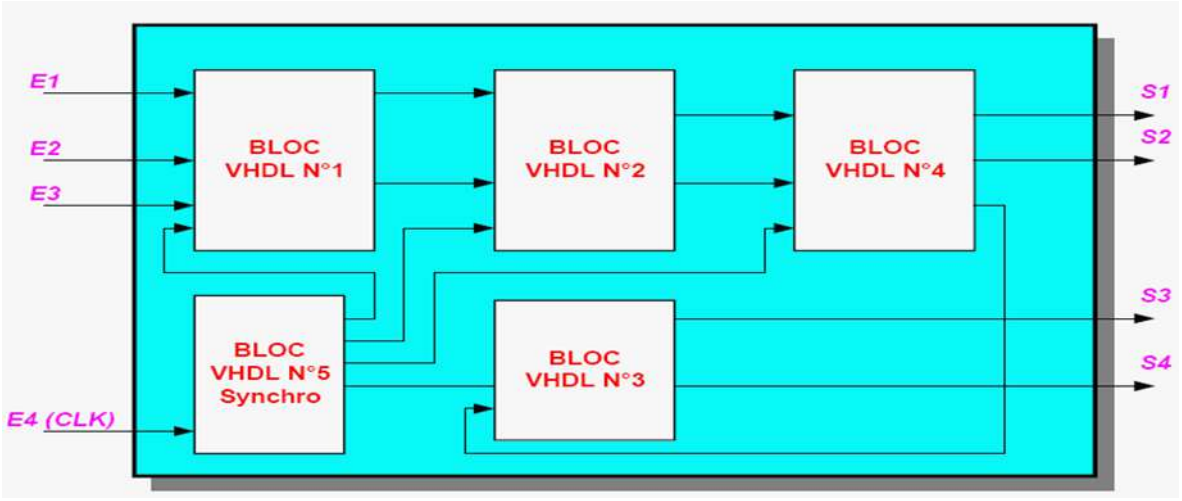


Figure 28 : Relation entre VHDL et FPGA [12]

II. Les différentes portes logiques

Réaliser les opérateurs de base de la logique combinatoire (les portes : Inverseur, ET, OU, NON-ET, NON-OU).

✓ Porte Inverseur :

Pour créer la **porte inverseur** vous aller cliquer dans la rubrique **Sources** sur la référence **xc3s1200e-5fg320** du projet combinatoire déjà créé auparavant ensuite dans la rubrique **Processes** vous aller cliquer sur **Create New Source** ensuite sur **VHDL Module** pour créer le fichier VHDL de la **porte_inverseur** nommé dans **file name** comme le montre la figure 28.

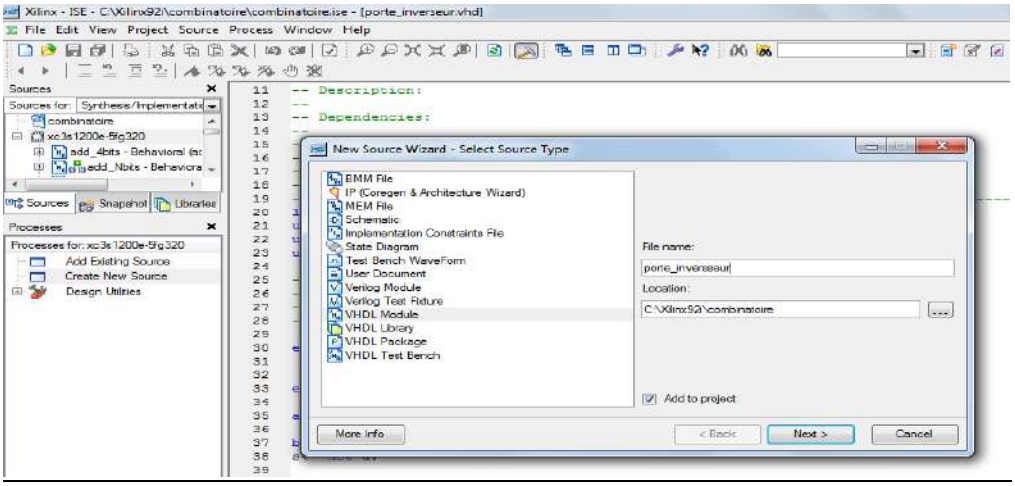


Figure 29 : première étape de la création d'un fichier VHDL

Ensuite vous aller nommé **les entrées/sorties** de la **porte_inverseur** (on choisit la direction de l'**entrée a** en **in** et la **sortie s** en **out**), le nom de l'entité (elle prendra par défaut le **nom du fichier**) et le nom de l'architecture (par défaut c'est **Behavioral**).

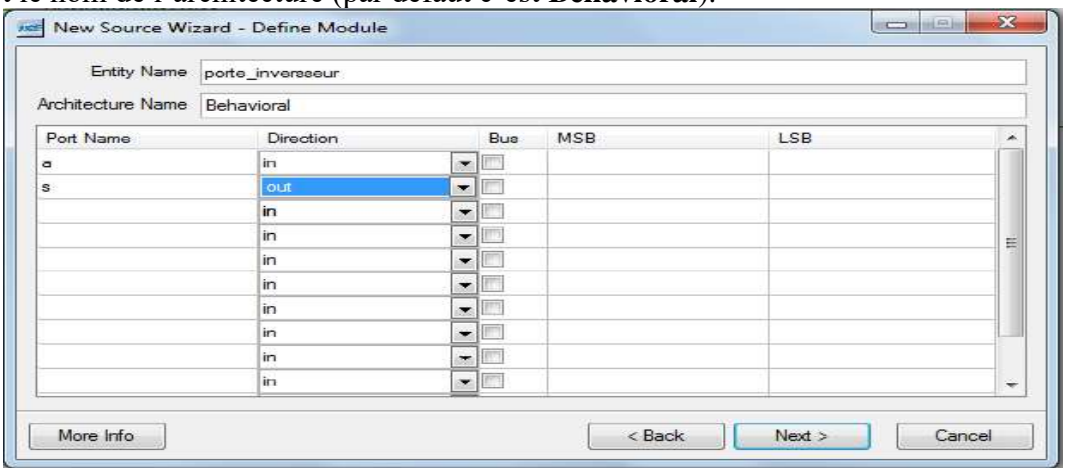


Figure 30 : deuxième étape de la création d'un fichier VHDL

Alors le programme de l'inverseur est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity porte_inverseur is
    Port ( a : in  STD_LOGIC;
           s : out  STD_LOGIC);
end porte_inverseur;
-- Déclaration de l'architecture avec une affectation simple.
architecture Behavioral of porte_inverseur is
begin
s<= not a;
end Behavioral;
```

Pour créer le **Test de la porte inverseur** vous aller cliquer dans la rubrique **Sources** sur la porte **porte_inverseur** du projet combinatoire déjà créé auparavant ensuite dans la rubrique **Processes** vous aller cliquer sur **Create New Source** ensuite sur **VHDL Test Bench** pour créer le fichier Test du VHDL : **test_porte_inverseur** nommé dans **file name** comme le montre la figure 31.

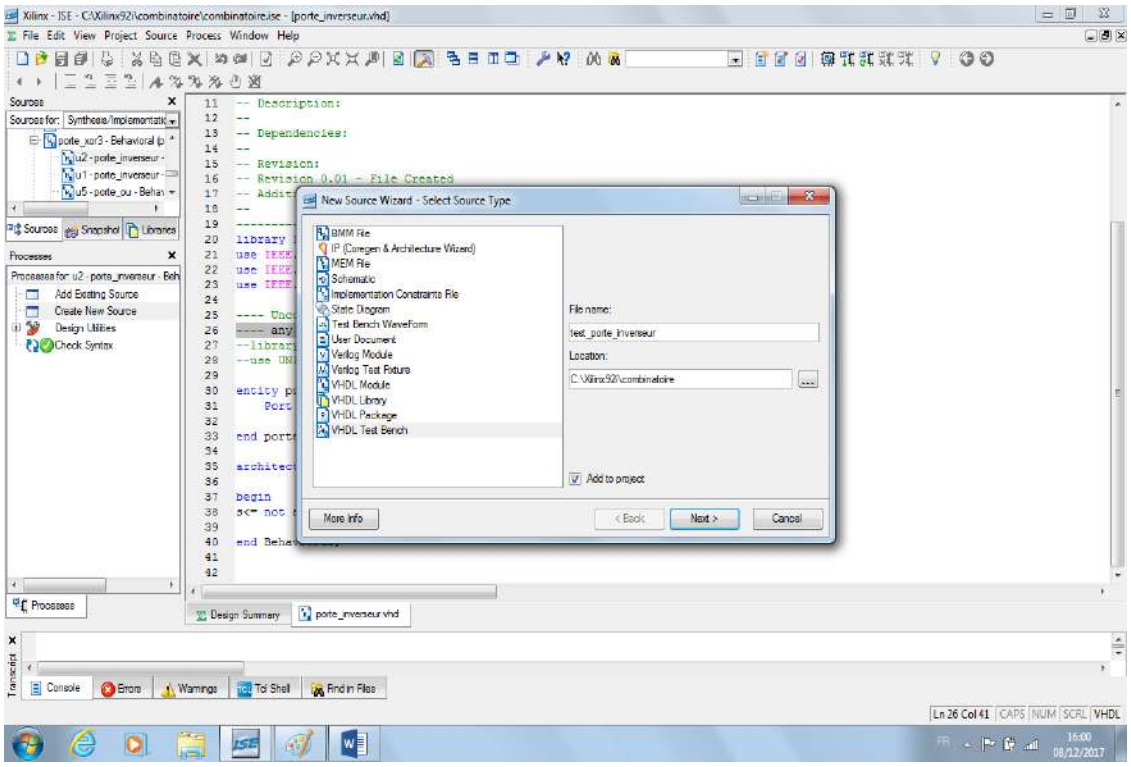


Figure 31 : la création d'un fichier test bench VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity test_port_invers_vhd is
end test_port_invers_vhd;

architecture behavior of test_port_invers_vhd is
        component porte_inverseur
        port(
                a : in std_logic;
                s : out std_logic);
        end component;

        --inputs
        signal a :  std_logic := '0';

        --outputs
        signal s :  std_logic;
```

```
begin
        -- instantiate the unit under test (uut)
        uut: porte_inverseur port map(
                a => a,
                s => s
        );
        tb : process
        begin

        a<='0';wait for 20 ns;
        a<='1';wait for 20 ns;
        wait;
        end process;
end;
```

Remarque : Que ce soit pour le **VHDL module** ou le **VHDL test bench** il faut cliquer sur **Check Syntax** dans la rubrique **Processes** pour vérifier s'il n'y a pas d'erreur de compilation**.**

Et pour voir le schéma de la porte décrite, Donner la priorité à la **porte_inverseur** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

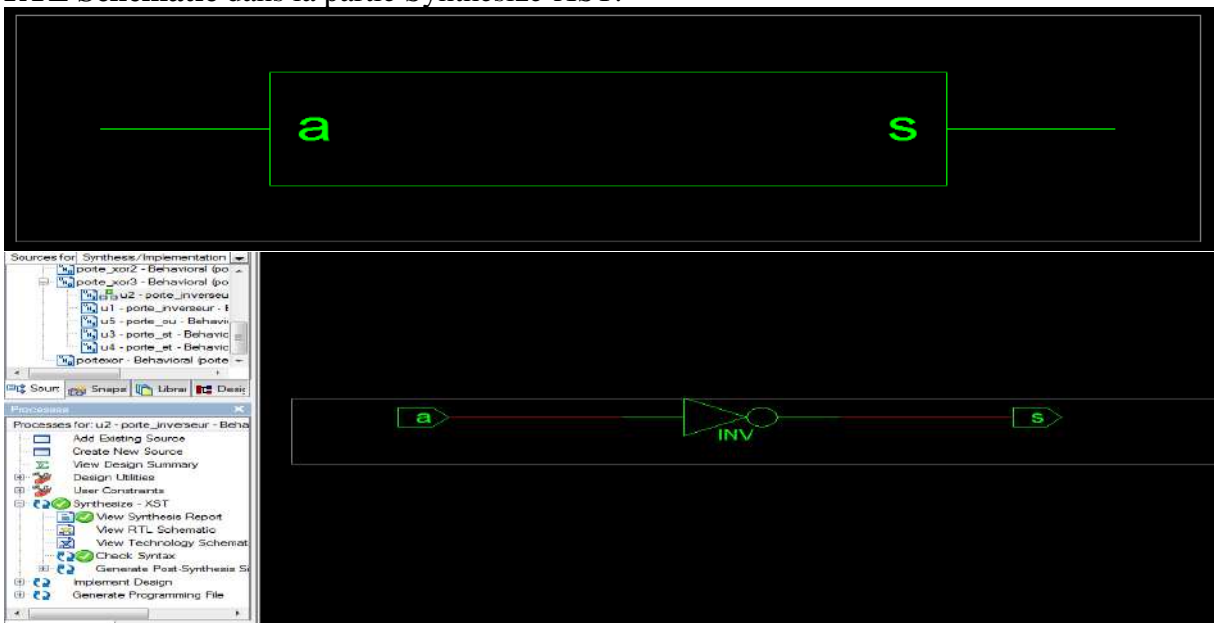


Figure 32 : le schéma de la porte_inversseur

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte en donnant des valeurs pour l'entrée **a** et voir la sortie **S** :

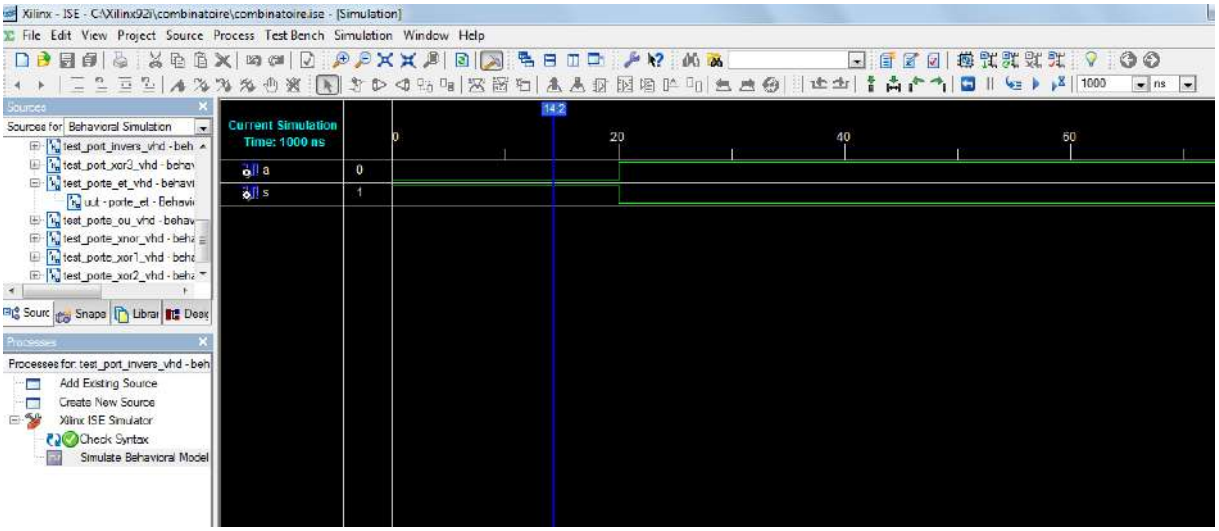


Figure 33 : la simulation de la porte_inverseur

De la même manière que dans la porte inverseur nous allons créer les autres portes :

- ✓ Porte AND :

Alors le programme de la porte_et est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity porte_et is
   Port ( a : in  STD_LOGIC;
          b: in  STD_LOGIC;
          s : out  STD_LOGIC);
end porte_et;
-- Déclaration de l'architecture avec une affectation simple.
architecture Behavioral of porte_et is
begin
s<=  a and b;
end Behavioral;
```

Le fichier Test du VHDL : **test_porte_et** nommé dans **file name** on change juste la partie du **tb :process**

```
tb : process
begin
a<= '0'; b<= '0'; wait for 20 ns;
a<= '1'; b<= '0'; wait for 20 ns;
a<= '0'; b<= '1'; wait for 20 ns;
a<= '1'; b<= '1'; wait for 20 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **porte_et** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

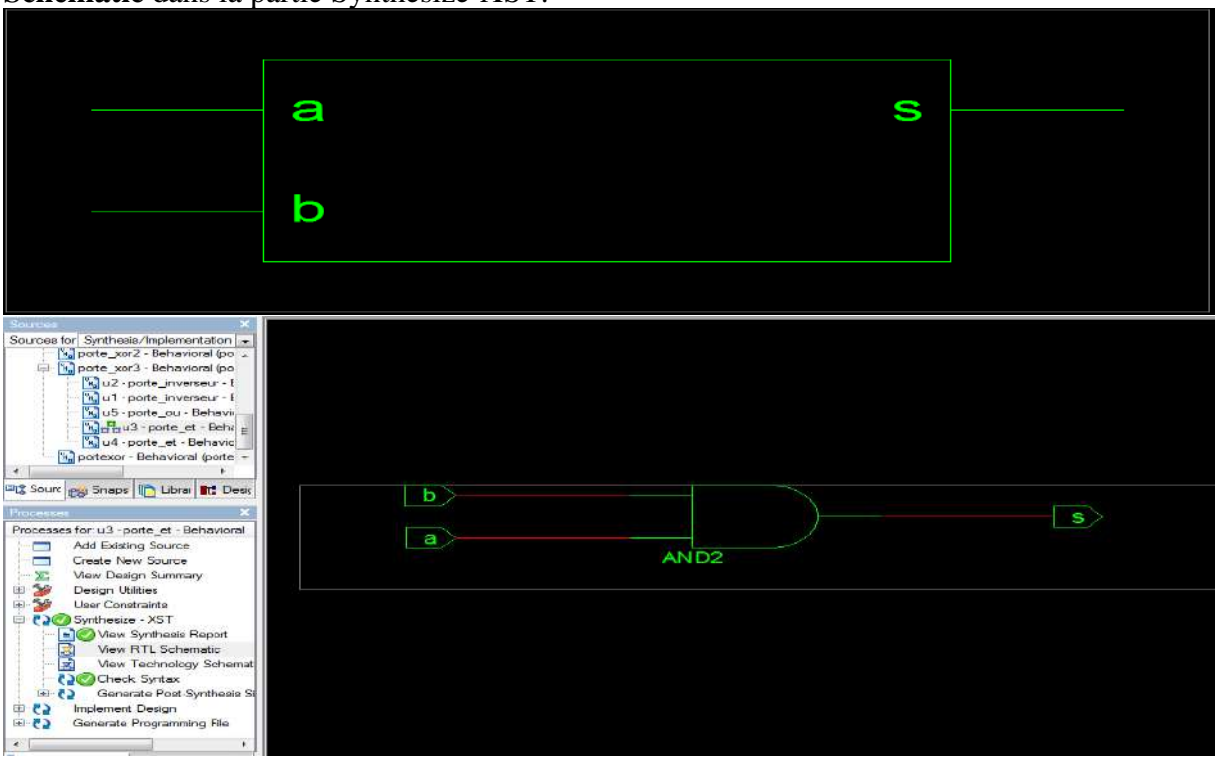


Figure 34 : le schéma de la porte_et

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte en donnant des valeurs pour l'entrée **a** et **b** pour voir la sortie **S** :

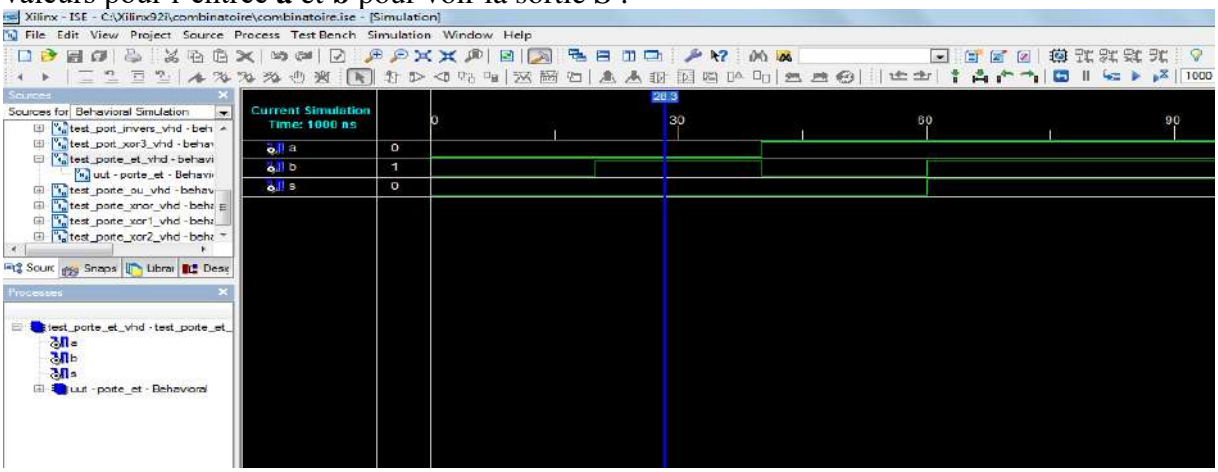


Figure 35 : la simulation de la porte_et

✓ Porte OR :

Alors le programme de la porte_ou est comme suite :

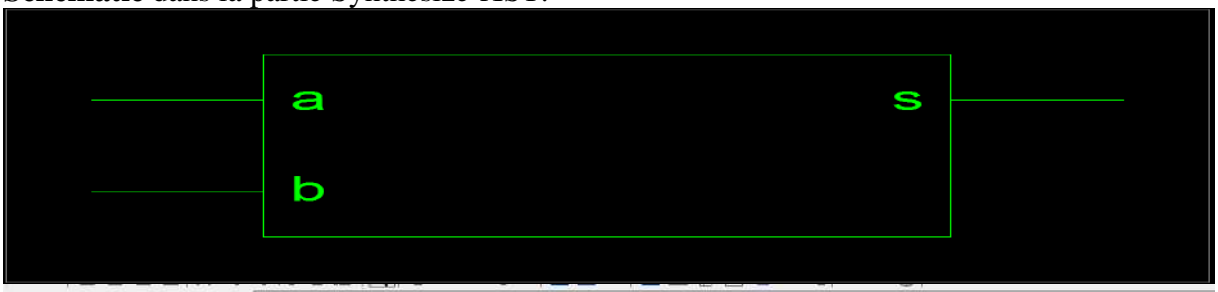
```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity porte_ou is
    Port ( a : in  STD_LOGIC;
           b: in  STD_LOGIC;
           s : out  STD_LOGIC);
end porte_ou;
-- Déclaration de l'architecture avec une affectation simple.
architecture Behavioral of porte_ou is
begin
s<=  a or b;
end Behavioral;
```

Le fichier Test du VHDL : **test_porte_ou** nommé dans **file name** on change juste la partie du **tb :process**

```
tb : process
begin
a<= '0'; b<= '0'; wait for 20 ns;
a<= '1'; b<= '0'; wait for 20 ns;
a<= '0'; b<= '1'; wait for 20 ns;
a<= '1'; b<= '1'; wait for 20 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **porte_ou** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.



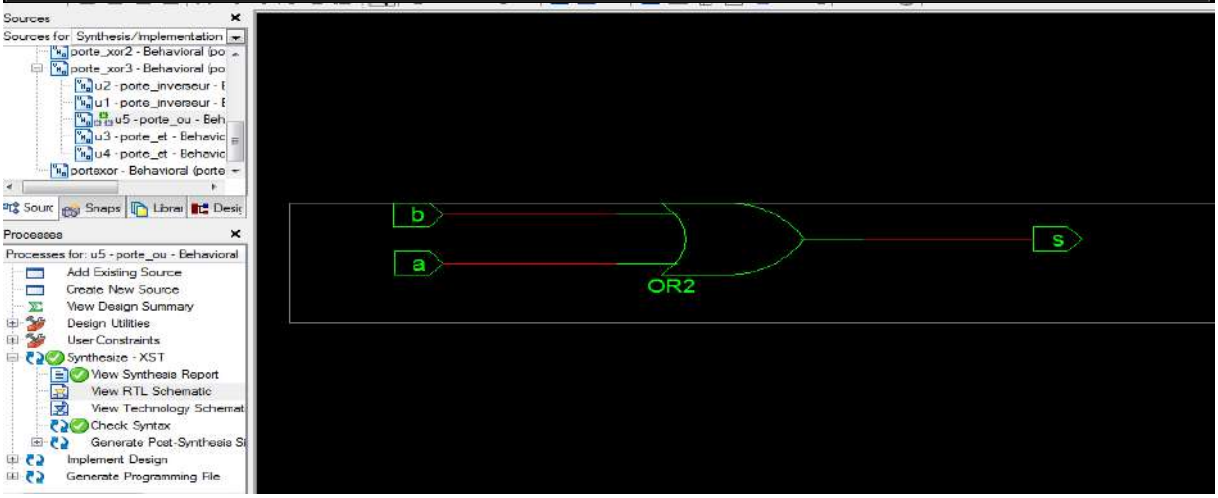


Figure 36 : le schéma de la porte_ou

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte en donnant des valeurs pour l'entrée **a** et **b** pour voir la sortie **S** :

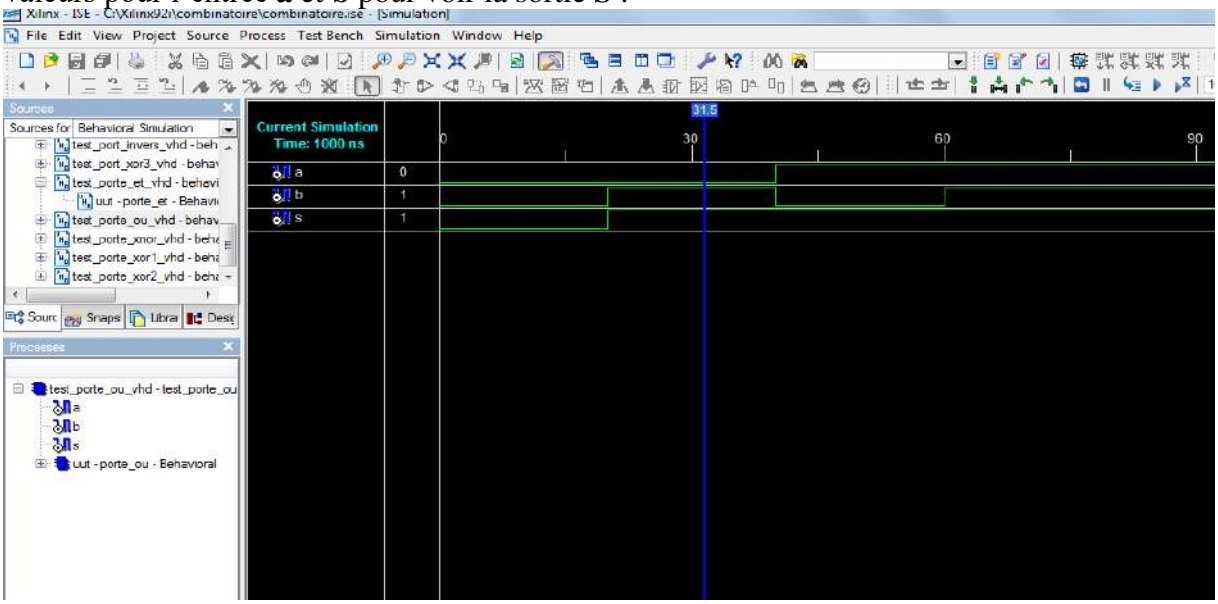


Figure 37 : la simulation de la porte_ou

✓ Porte NAND :

Alors le programme de la porte_non_et est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity porte_non_et is
    Port ( a : in  STD_LOGIC;
           b: in  STD_LOGIC;
           s : out  STD_LOGIC);
end porte_non_et;
-- Déclaration de l'architecture avec une affectation simple.
architecture Behavioral of porte_non_et is
begin
s<=  a nand b;
end Behavioral;
```

Le fichier Test du VHDL : **test_porte_non_et** nommé dans **file name** on change juste la partie du **tb :process**

```
tb : process
begin

a<= '0'; b<= '0'; wait for 20 ns;
```

```
        a<= '1'; b<= '0'; wait for 20 ns;
      a<= '0'; b<= '1'; wait for 20 ns;
        a<= '1'; b<= '1'; wait for 20 ns;
        wait;
        end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **porte_non_et** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

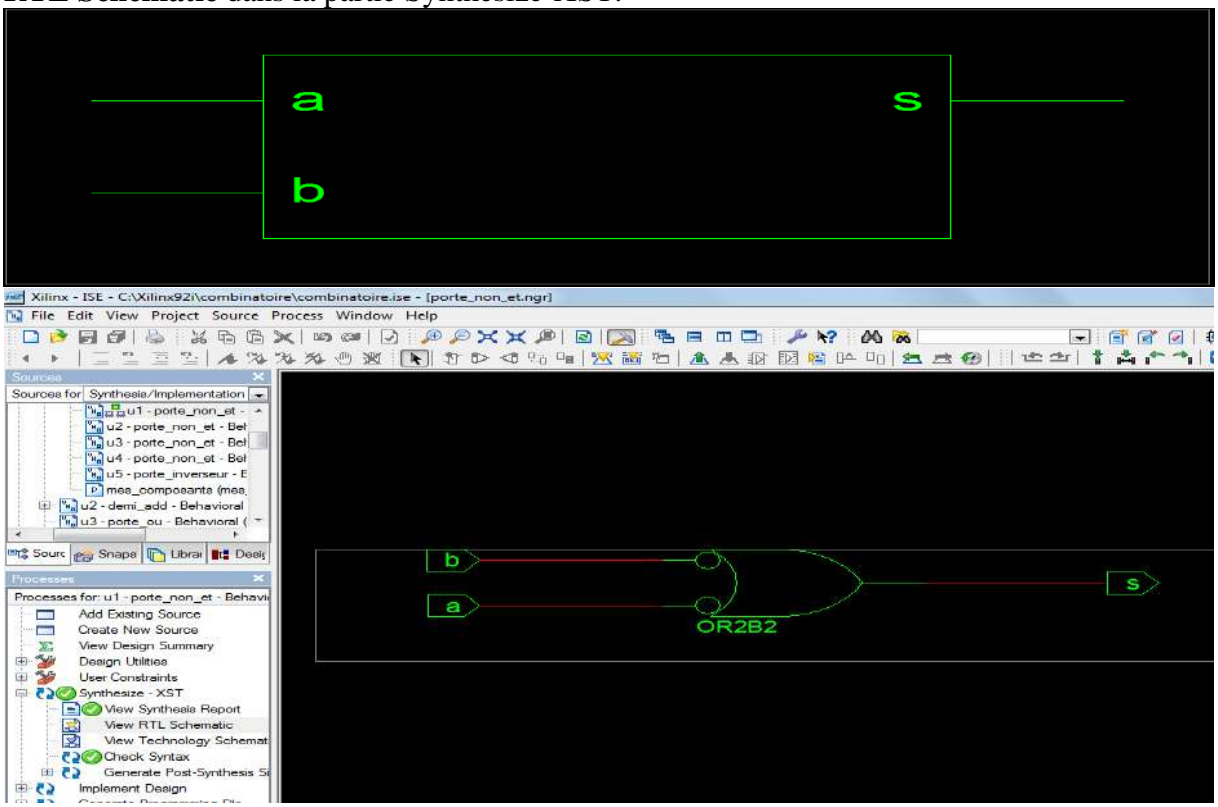


Figure 38 : le schéma de la porte_non_et

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte en donnant des valeurs pour l'entrée **a** et **b** pour voir la sortie **S** :

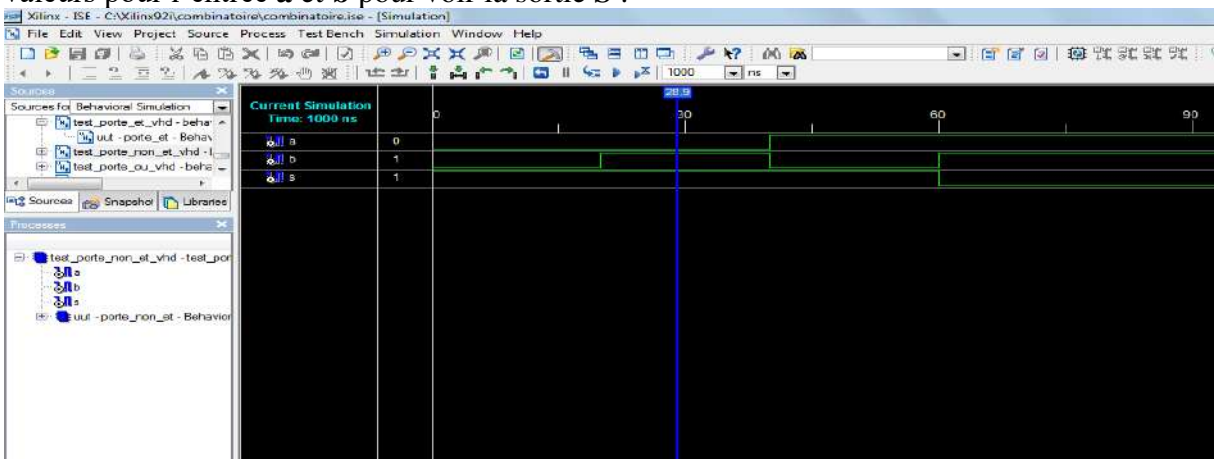


Figure 39 : la simulation de la porte_non_et

✓ Porte NOR :

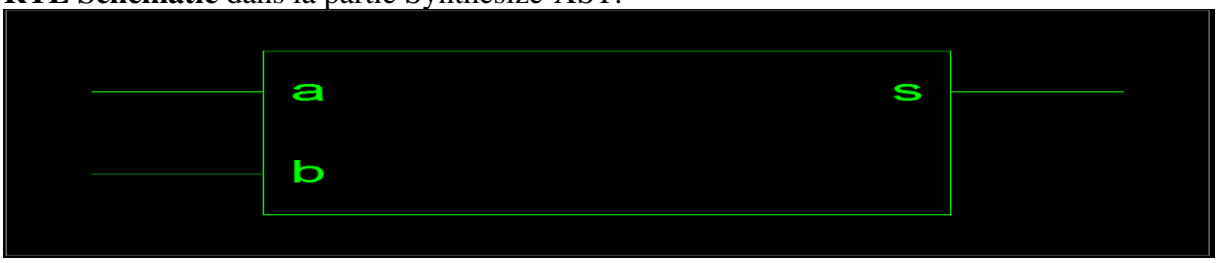
Alors le programme de la porte_non_ou est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity porte_non_ou is
   Port ( a : in  STD_LOGIC;
          b: in  STD_LOGIC;
          s : out  STD_LOGIC);
end porte_non_ou;
-- Déclaration de l'architecture avec une affectation simple.
architecture Behavioral of porte_non_ou is
begin
s<=  a nor b;
end Behavioral;
```

Le fichier Test du VHDL : **test_porte_non_ou** nommé dans **file name** on change juste la partie du **tb :process**

```
tb : process
begin
a<= '0'; b<= '0'; wait for 20 ns;
a<= '1'; b<= '0'; wait for 20 ns;
a<= '0'; b<= '1'; wait for 20 ns;
a<= '1'; b<= '1'; wait for 20 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **porte_non_ou** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.



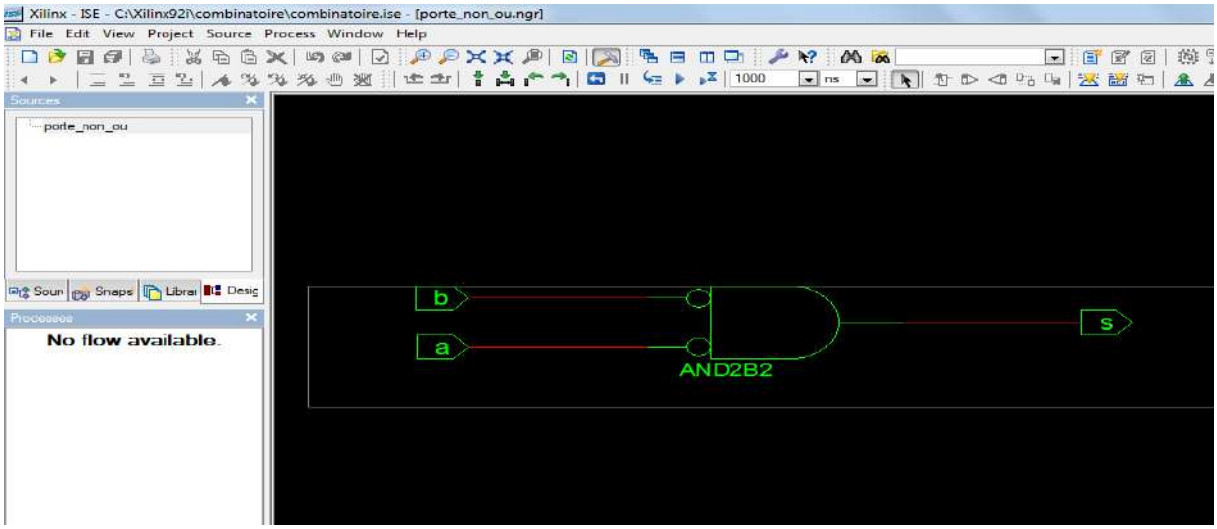


Figure 40 : le schéma de la porte_non_ou

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte en donnant des valeurs pour l'entrée **a** et **b** pour voir la sortie **S** :

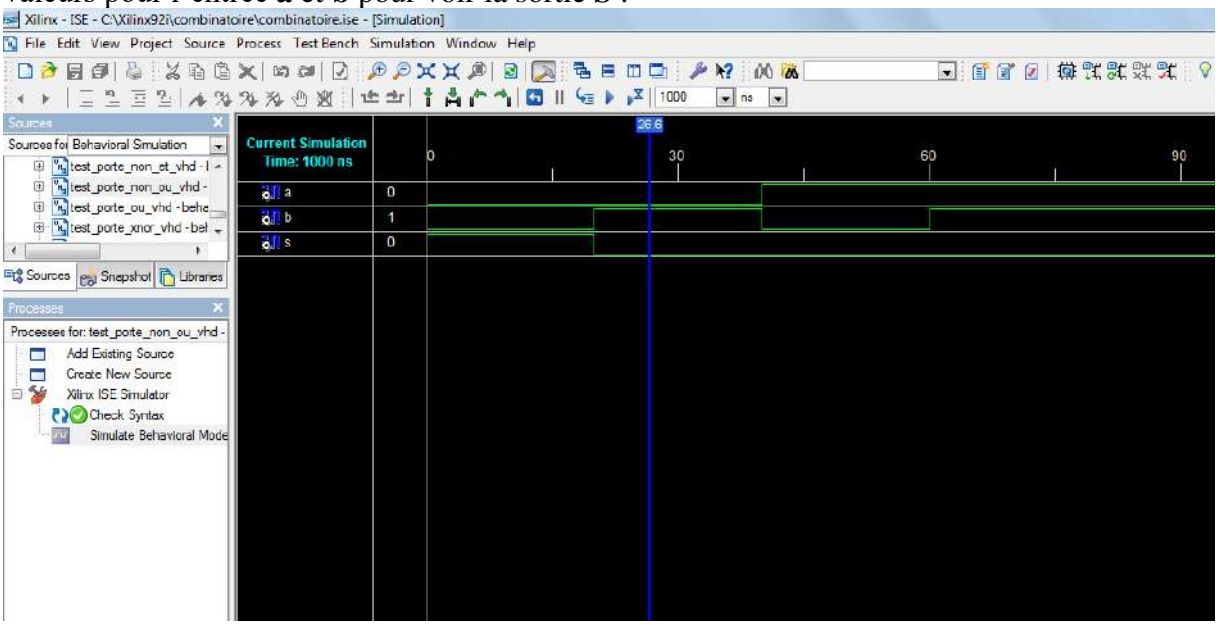


Figure 41 : la simulation de la porte_non_ou

1- Réaliser la porte XOR en utilisant les trois descriptions (flot de données, comportementale, structurelle) en faisant appel aux composants déjà réalisé en utilisant : **COMPONENT**.

✓ Porte XOR1 : flot de données

Alors le programme de la porte_xor1 est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity porte_xor1 is
   Port ( a : in  STD_LOGIC;
          b: in  STD_LOGIC;
```

```
            s : out  STD_LOGIC);
    end porte_xor1;
    -- Déclaration de l'architecture avec une affectation conditionnelle.
    architecture Behavioral of porte_xor1 is
    begin
    s<= '0' when a=b else '1';
    end Behavioral;
```

✓ Porte XOR2 : comportemental

Alors le programme de la porte_xor2 est comme suite :

```
    -- Déclaration de la bibliothèque et des paquetages.
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
   -- Déclaration de l'entité avec les entrées/sorties correspondants.
    entity porte_xor2 is
       Port ( a : in  STD_LOGIC;
              b: in  STD_LOGIC;
              s : out  STD_LOGIC);
    end porte_xor2;
    -- Déclaration de l'architecture avec une structure de test.
    architecture Behavioral of porte_xor1 is
    begin
    process (a,b)
    begin
    if a=b then s<= '0';
    else s<= '1';
    end if;
    end process;
    end Behavioral;
```

✓ Porte XOR3 : structurelle

Alors le programme de la porte_xor3 est comme suite :

```
    -- Déclaration de la bibliothèque et des paquetages.
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
   -- Déclaration de l'entité avec les entrées/sorties correspondants.
    entity porte_xor3 is
       Port ( a : in  STD_LOGIC;
              b: in  STD_LOGIC;
              s : out  STD_LOGIC);
    end porte_xor3;
    -- Déclaration de l'architecture en structurelle.
    architecture Behavioral of porte_xor3 is
    signal s1,s2,s3,s4:std_logic;
    component porte_inverseur
      port (a:in std_logic;
            s:out std_logic);
```

```
end component;
component porte_et
  port (a:in std_logic;
        b:in std_logic;
        s:out std_logic);
end component;
component porte_ou
  port (a:in std_logic;
        b:in std_logic;
        s:out std_logic);
end component;
begin
u2:porte_inverseur port map (b,s2);
u1:porte_inverseur port map (a,s1);
u5:porte_ou port map (s3,s4,s);
u3:porte_et port map (s1,b,s3);
u4:porte_et port map (a,s2,s4);
end Behavioral;
```

Le fichier Test du VHDL : **test_porte_xor 1,2 et 3** c'est le même on change juste la partie du **tb :process**

```
tb : process
begin
a<= '0'; b<= '0'; wait for 20 ns;
a<= '1'; b<= '0'; wait for 20 ns;
a<= '0'; b<= '1'; wait for 20 ns;
a<= '1'; b<= '1'; wait for 20 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **porte_xor3** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

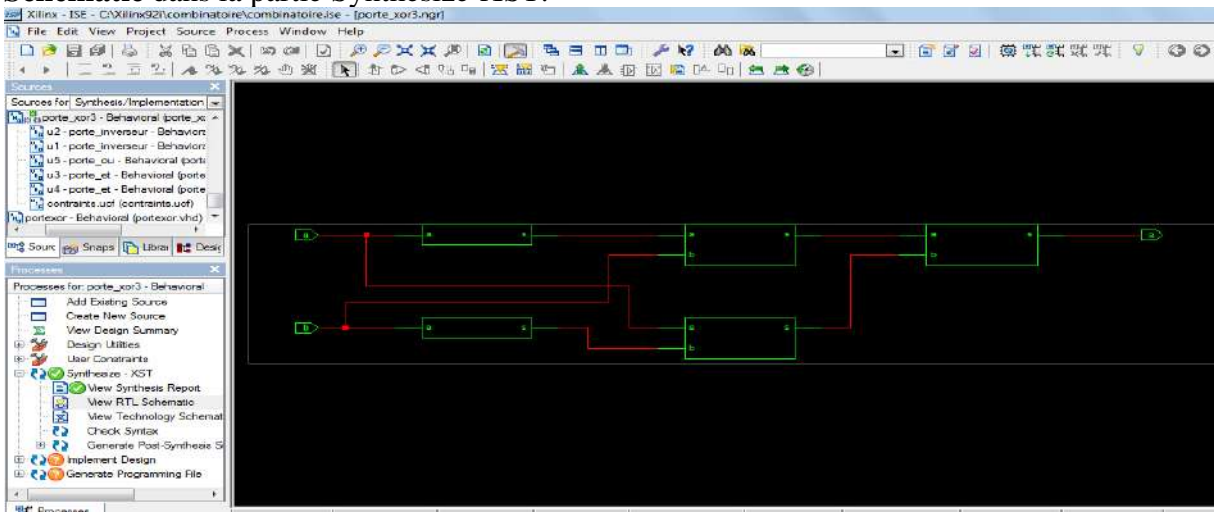


Figure 42 : le schéma de la porte_xor3

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte on donnant des valeurs pour l'entrée **a** et **b** pour voir la sortie **S** :

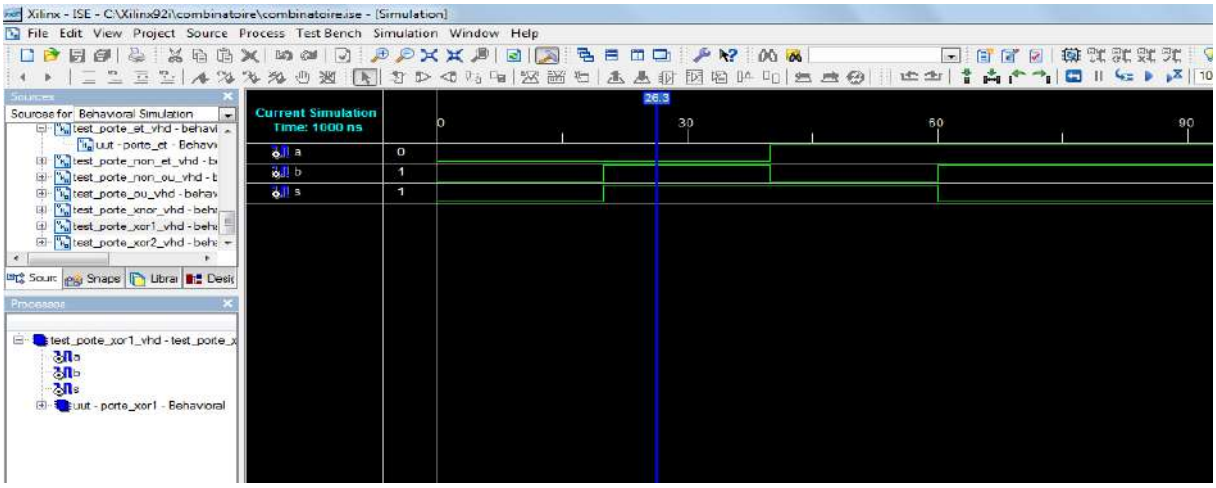


Figure 43 : la simulation de la porte_xor1,2 et 3

III. Multiplexeur

Réaliser un multiplexeur (2 vers 1) en utilisant les trois descriptions (flot de données, comportementale, structurelle).

✓ Porte Multiplexeur 1 : flot de données

Alors le programme de la porte_mux1 est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity porte_mux1 is
   Port ( in1 : in  STD_LOGIC;
          In2: in  STD_LOGIC;
          sel : in  STD_LOGIC;
          s : out  STD_LOGIC);
end porte_mux1;
-- Déclaration de l'architecture avec une affectation conditionnelle.
architecture Behavioral of porte_mux1 is
begin
s<= in1 when sel = '0';
          else s<= in2;
end Behavioral;
```

✓ Porte Multiplexeur 2 : comportemental

Alors le programme de la porte_mux2 est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity porte_mux2 is
   Port ( in1 : in  STD_LOGIC;
          In2: in  STD_LOGIC;
```

```
            sel : in  STD_LOGIC;
            s : out  STD_LOGIC);
    end porte_mux2;
    -- Déclaration de l'architecture avec une structure de test.
    architecture Behavioral of porte_mux2 is
    begin
    process (in1,in2, sel)
    begin
    if sel = '0' then s <=in1;
    else s<= in2;
    end if;
    end process;
    end Behavioral;
```

✓ Porte Multiplexeur 3 : structurelle

Alors le programme de la porte_mux3 est comme suite :

```
    -- Déclaration de la bibliothèque et des paquetages.
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
    -- Déclaration du paquetage spécifique pour appeler les components.
    use work.mes_composants.all;
   -- Déclaration de l'entité avec les entrées/sorties correspondants.
    entity porte_mux3 is
       Port ( in1 : in  STD_LOGIC;
              In2: in  STD_LOGIC;
              sel : in  STD_LOGIC;
              s : out  STD_LOGIC);
    end porte_mux3;
    -- Déclaration de l'architecture en structurelle.
    architecture Behavioral of porte_mux3 is
    signal s1,s2,s3:std_logic;
    begin
    u1:porte_inverseur port map (sel,s1);
    u4:porte_ou port map (s2,s3,s);
    u2:porte_et port map (in1,s1,s2);
    u3:porte_et port map (in2,sel,s3);
    end Behavioral;
```

Le fichier Test du VHDL : **test_porte_xor 1,2 et 3** c'est le même on change juste la partie du **tb :process**

```
    tb : process
    begin
in1<='0'; in2<='0'; sel<='0'; wait for 20 ns;
in1<='0'; in2<='0'; sel<='1'; wait for 20 ns;
in1<='0'; in2<='1'; sel<='0'; wait for 20 ns;
in1<='0'; in2<='1'; sel<='1'; wait for 20 ns;
in1<='1'; in2<='0'; sel<='0'; wait for 20 ns;
in1<='1'; in2<='0'; sel<='1'; wait for 20 ns;
in1<='1'; in2<='1'; sel<='0'; wait for 20 ns;
```

```
in1<='1'; in2<='1'; sel<='1'; wait for 20 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **porte_mux3** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

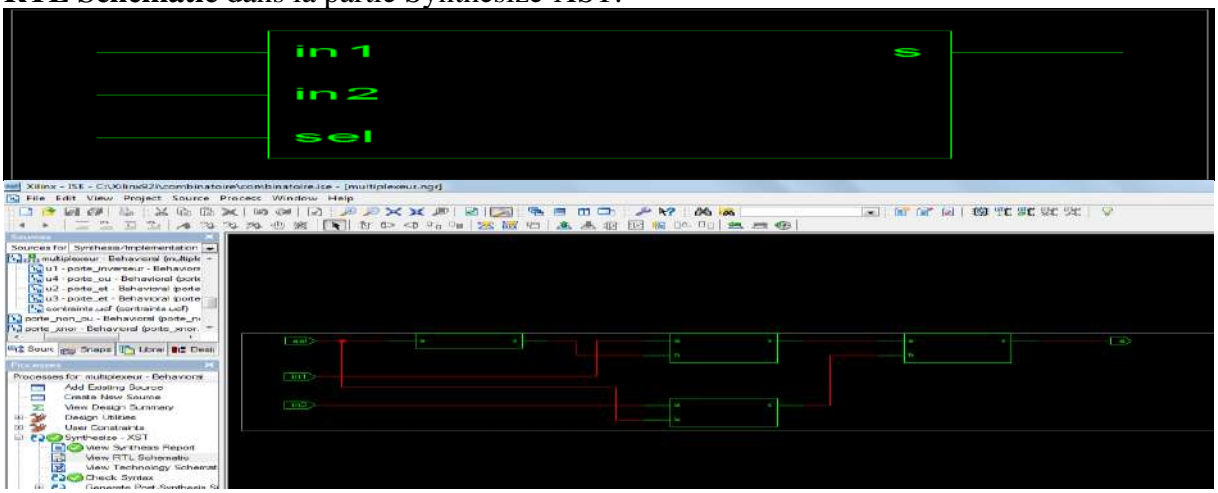


Figure 44 : le schéma de la porte_mux3

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte en donnant des valeurs pour l'entrée **in1, in2** et **sel** pour voir la sortie **S** :

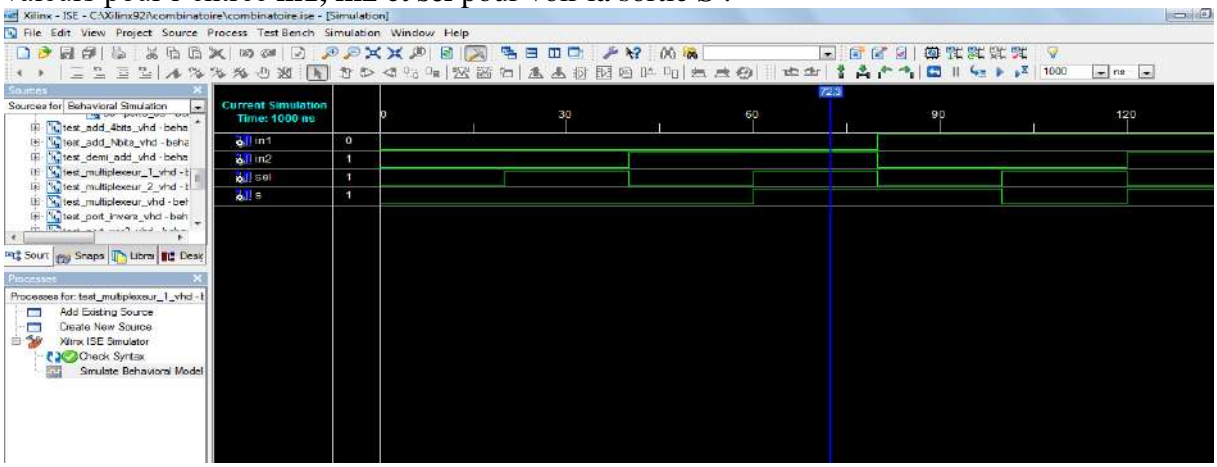


Figure 45 : la simulation de la porte_mux1,2 et 3

IV. Demi-additionneur

Réaliser un demi-additionneur en utilisant les portes NAND (NON-ET) et l'inverseur dans une description structurelle.

✓ Demi-additionneur : structurelle

Alors le programme du demi_add est comme suite :

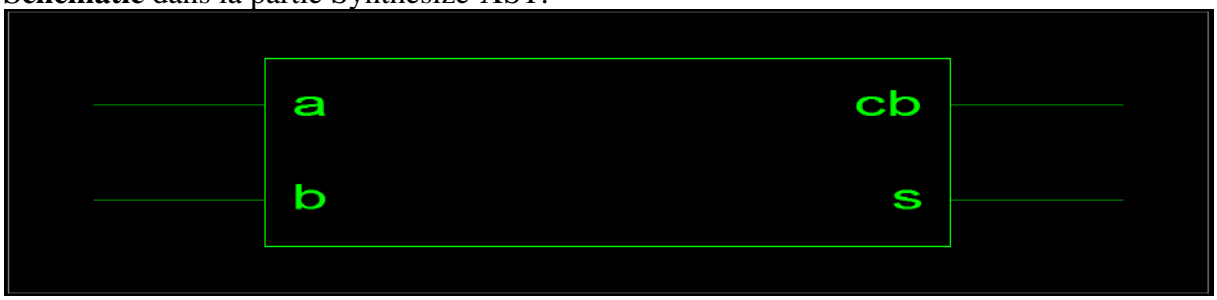
```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration du paquetage spécifique pour appeler les components.
```

```
use work.mes_composants.all;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity demi_add is
   Port ( a : in  STD_LOGIC;
          b: in  STD_LOGIC;
          s : out  STD_LOGIC;
          cb : out  STD_LOGIC);
end demi_add;
-- Déclaration de l'architecture en structurelle.
architecture Behavioral of demi_add is
signal s1,s2,s3:std_logic;
begin
u1:porte_non_et port map (a,b,s1);
u2:porte_non_et port map (a,s1,s2);
u3:porte_non_et port map (s1,b,s3);
u4:porte_non_et port map (s2,s3,s);
u5:porte_inverseur port map (s1,cb);
end Behavioral;
```

Le fichier Test du VHDL : **test_demi_add** c'est le même on change juste la partie du **tb :process**

```
tb : process
begin
a<= '0'; b<= '0'; wait for 20 ns;
 a<= '1'; b<= '0'; wait for 20 ns;
a<= '0'; b<= '1'; wait for 20 ns;
 a<= '1'; b<= '1'; wait for 20 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **demi_add** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.



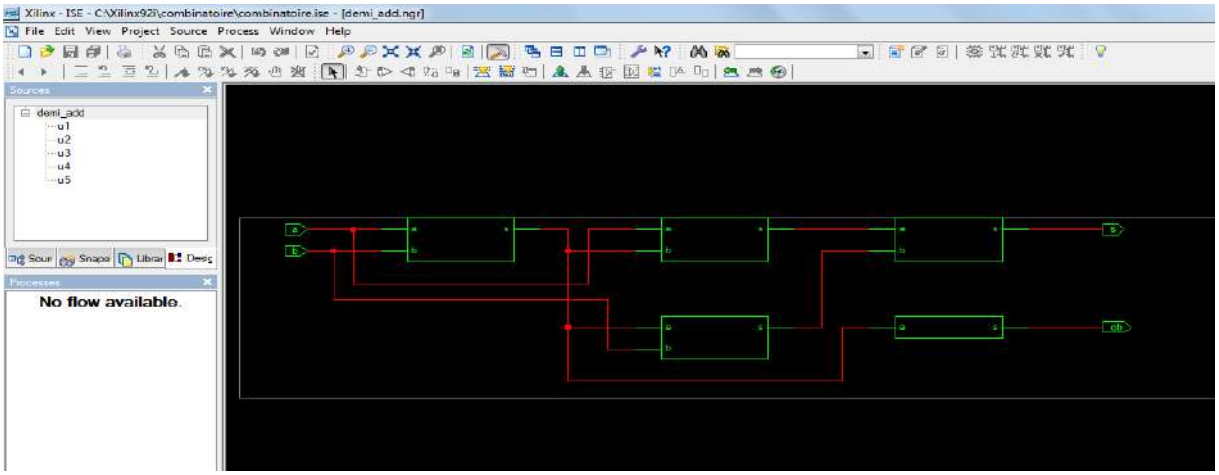


Figure 46 : le schéma du demi_add

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte en donnant des valeurs pour l'entrée **a** et **b** pour voir la sortie **S et cb** :

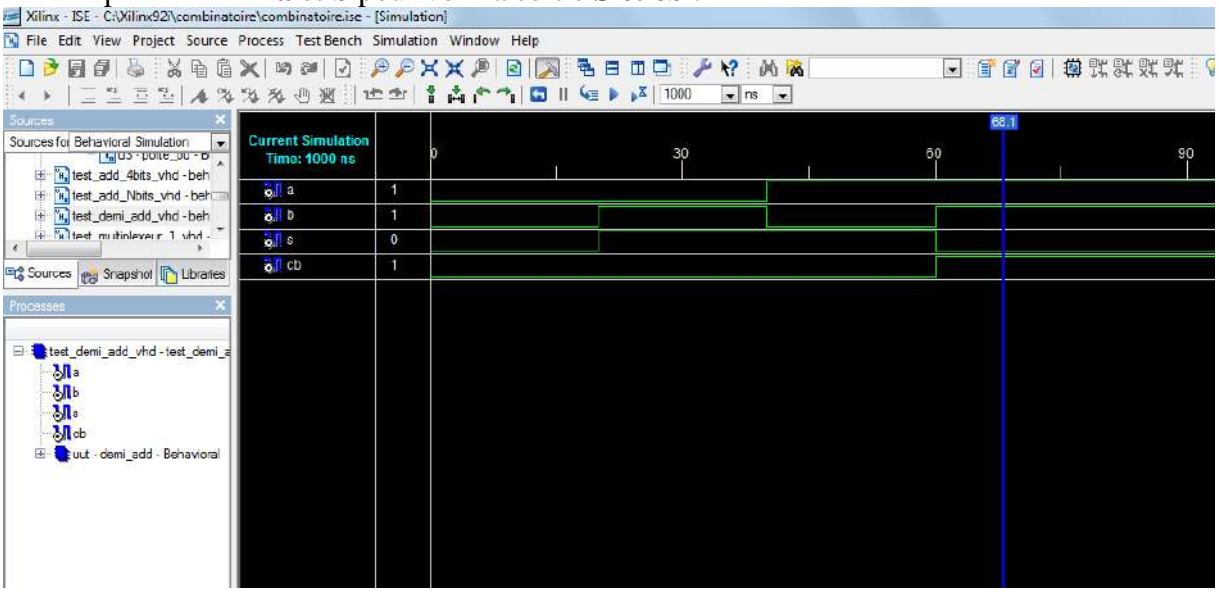


Figure 47 : la simulation du demi_add

V. Additionneur complet

Réaliser un additionneur complet de 1 bit en utilisant le demi_additionneur de 1bit ainsi qu'une retenue à l'entrée et sortie du système.

✓ Additionneur complet 1bit: structurelle

Alors le programme du add_1bit est comme suite :

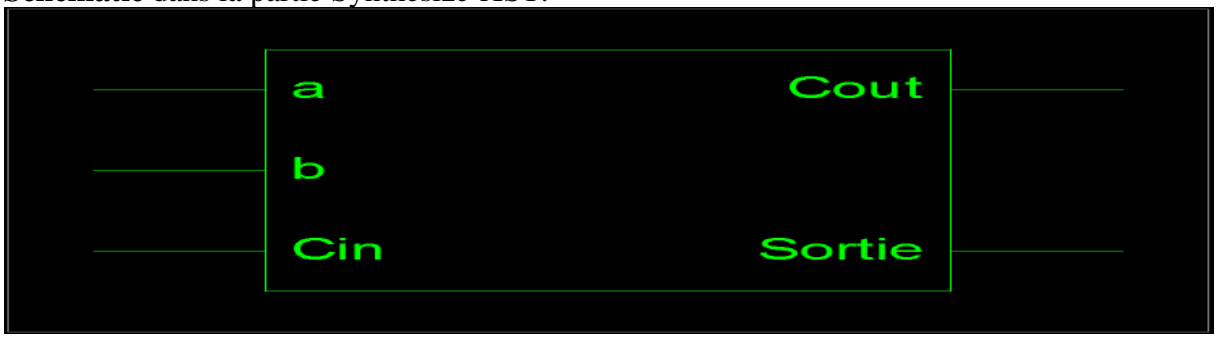
```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration du paquetage spécifique pour appeler les components.
use work.mes_composants.all;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity add_1bit is
```

```
        Port ( a : in  STD_LOGIC;
               b: in  STD_LOGIC;
               Cin : in  STD_LOGIC;
               Sortie : out  STD_LOGIC;
               Cout : out  STD_LOGIC);
    end add_1bit;
    -- Déclaration de l'architecture en structurelle.
    architecture Behavioral of add_1bit  is
    signal s1,s2,s3:std_logic;
    begin
    u1:demi_add port map (a,b,s1,S2);
    u2:demi_add port map (Cin,s1,sortie,s3);
    u3:porte_ou port map (s3,s2,Cout);
    end Behavioral;
```

Le fichier Test du VHDL : **test_ add_1bit** c'est le même on change juste la partie du **tb :process**

```
    tb : process
    begin
a<='0'; b<='0'; Cin<= '0'; wait for 20 ns;
a<='0'; b<='0'; Cin<= '1'; wait for 20 ns;
a<='0'; b<='1'; Cin<= '0'; wait for 20 ns;
a<='0'; b<='1'; Cin<= '1'; wait for 20 ns;
a<='1'; b<='0'; Cin<= '0'; wait for 20 ns;
a<='1'; b<='0'; Cin<= '1'; wait for 20 ns;
a<='1'; b<='1'; Cin<= '0'; wait for 20 ns;
a<='1'; b<='1'; Cin<= '1'; wait for 20 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **add_1bit** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.



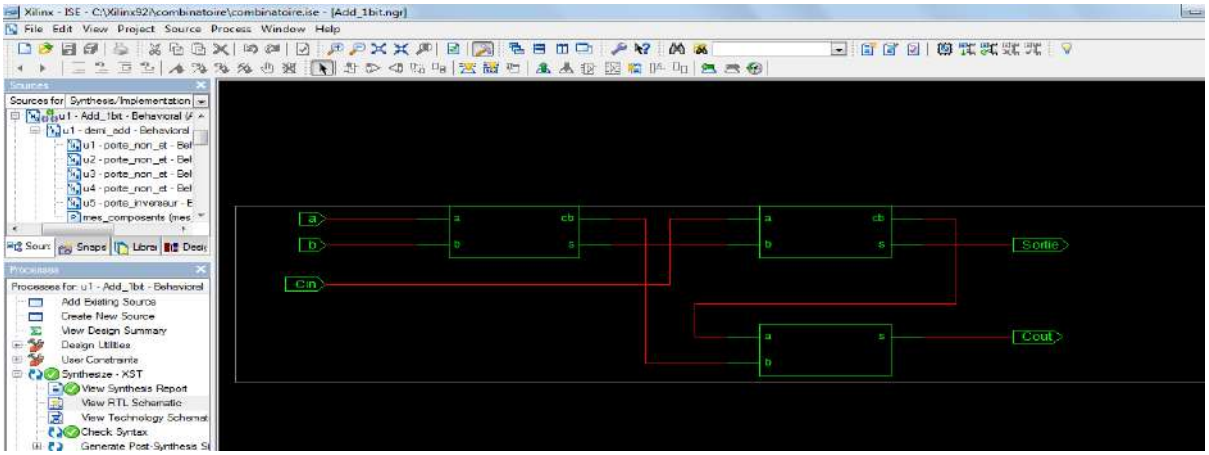


Figure 48 : le schéma d'add_1bit

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte en donnant des valeurs pour l'entrée **a, b** et **Cin** pour voir la sortie **Sortie et Cout** :

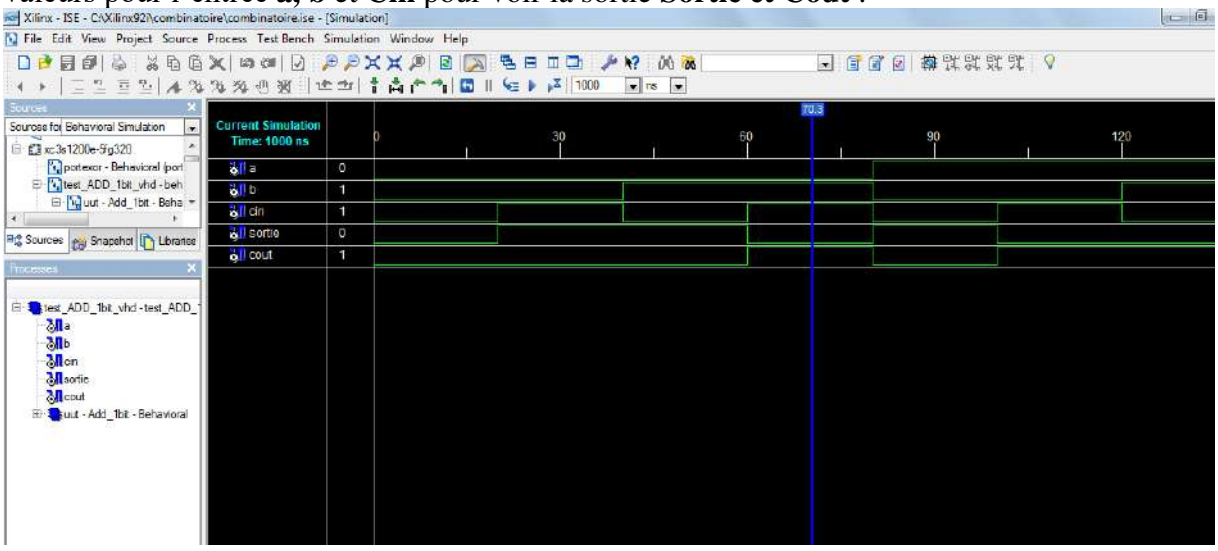


Figure 49 : la simulation d'add_1bit

1- Réaliser un additionneur complet de 4 bits en utilisant l'additionneur complet de 1bit. une description structurelle ainsi qu'un paquetage spécifique sont utilisés.

✓ Additionneur complet 4bits: structurelle

Nommé **les entrées/sorties** de l'**additionneur 4bits** (on choisit la direction de l'**entrée a** en **in** et la **sortie s** en **out**), le nom de l'entité (elle prendra par défaut le **nom du fichier**) et le nom de l'architecture (par défaut c'est **Behavioral**).

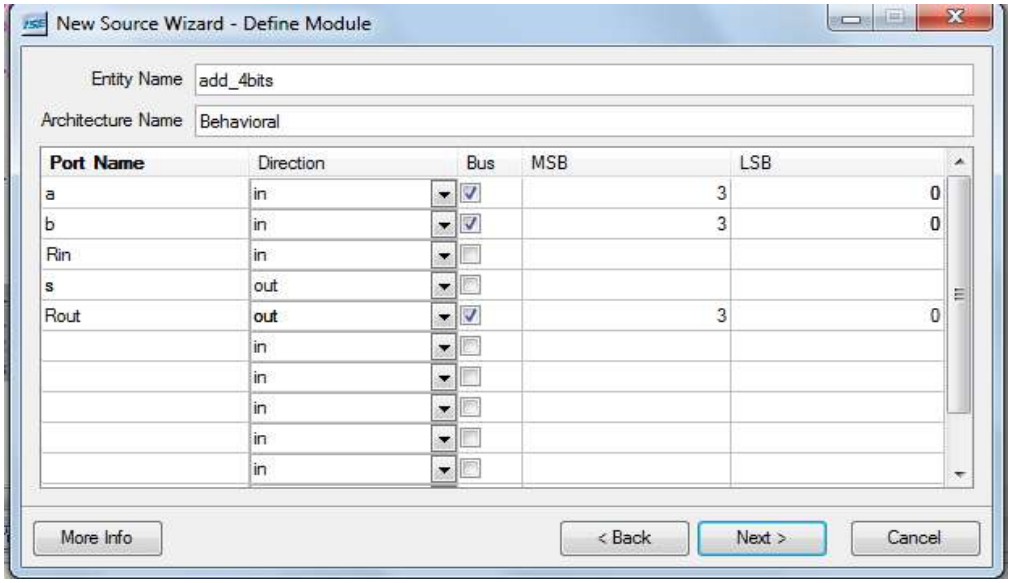


Figure 50 : la création d'un fichier VHDL des bits vecteurs

Alors le programme du add_4bits est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration du paquetage spécifique pour appeler les components.
use work.mes_composants.all;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity add_4bits is
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b: in  STD_LOGIC_VECTOR (3 downto 0);
          Rin : in  STD_LOGIC;
          s : out  STD_LOGIC_VECTOR (3 downto 0);
          Rout : out  STD_LOGIC);
end add_4bits;
-- Déclaration de l'architecture en structurelle.
architecture Behavioral of add_4bits  is
signal T1,T2,T3:std_logic;
begin
u1:Add_1bit port map (a(0),b(0),Rin,S(0),T1);
u2:Add_1bit port map (a(1),b(1),T1,S(1),T2);
u3:Add_1bit port map (a(2),b(2),T2,S(2),T3);
u4:Add_1bit port map (a(3),b(3),T3,S(3),Rout);
end Behavioral;
```

Le fichier Test du VHDL : **test_add_4bits** c'est le même on change juste la partie du **tb :process**

```
tb : process
begin
a<="0000"; b<="0000"; Rin<= '0'; wait for 20 ns;
a<="0100"; b<="0101"; Rin<= '1'; wait for 20 ns;
a<="1010"; b<="0110"; Rin<= '0'; wait for 20 ns;
```

```
a<="1100"; b<="0111"; Rin<= '1'; wait for 20 ns;
a<="0101"; b<="0111"; Rin<= '0'; wait for 20 ns;
a<="1000"; b<="0101"; Rin<= '1'; wait for 20 ns;
a<="0010"; b<="1011"; Rin<= '0'; wait for 20 ns;
a<="1110"; b<="0110"; Rin<= '1'; wait for 20 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **add_4bits** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

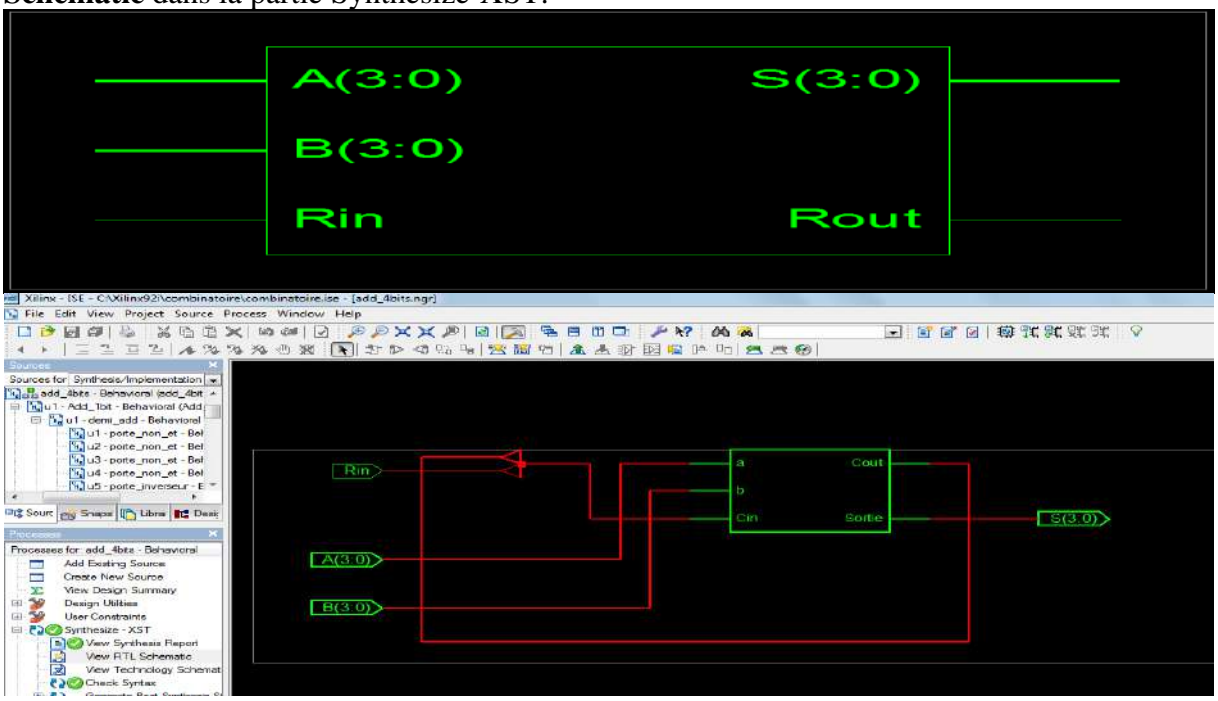


Figure 51 : le schéma d'add_4bits

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte on donnant des valeurs pour l'entrée **a,b** et **Rin** pour voir la sortie **S et Rout** :

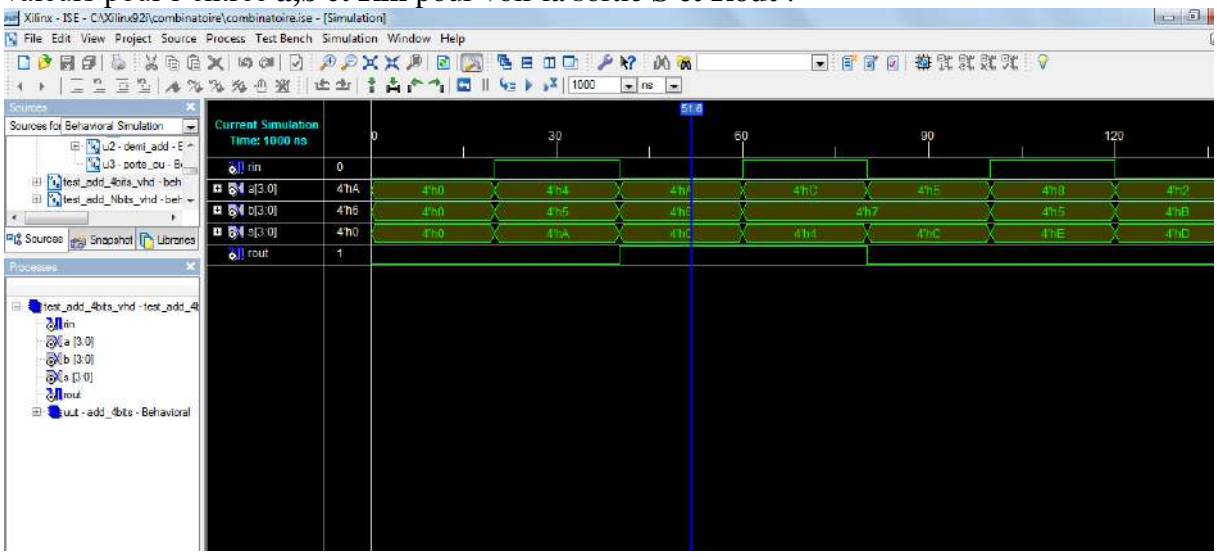


Figure 52 : la simulation d'add_4bits

Réaliser un additionneur complet de N bits en instanciant l'additionneur complet de 1bit. On doit déclarer le paramètre générique N dans l'entité et utiliser l'instruction GENERATE dans le programme principal de N bits.

✓ Additionneur complet Nbits: structurelle

Alors le programme du add_Nbits est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration du paquetage spécifique pour appeler les components.
use work.mes_composants.all;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity add_Nbits is
generic (n:natural:=8); -- (8 ou 16 ou AUTRES)
   Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
          b: in  STD_LOGIC_VECTOR (N-1  downto 0);
          Rin : in  STD_LOGIC;
          s : out  STD_LOGIC_VECTOR (N-1  downto 0);
          Rout : out  STD_LOGIC);
end add_Nbits;
-- Déclaration de l'architecture en structurelle.
architecture Behavioral of add_Nbits  is
signal t:std_logic_vector (n downto 0);
begin
u1:for i in 0 to n-1 generate
u2: add_1bit port map (a(i), b(i), t(i), s(i), t(i+1));
end generate;
t(0)<= Rin;
Rout <= t(n);
end Behavioral;
```

Le fichier Test du VHDL : **test_add_Nbits,** ICI N=8, la partie du **tb :process**

```
tb : process
begin
a<=x"FE"; b<=x"0D"; Rin<= '0'; wait for 20 ns;
a<=x"01"; b<=x"93"; Rin<= '1'; wait for 20 ns;
a<=x"A5"; b<=x"07"; Rin<= '0'; wait for 20 ns;
a<=x"C8"; b<=x"FF"; Rin<= '1'; wait for 20 ns;
wait;
end process;
end;
```

Et enfin dans la rubrique **behavioral Simulation**, on simule notre porte on donnant des valeurs pour l'entrée **a,b** et **Rin** pour voir la sortie **S** :

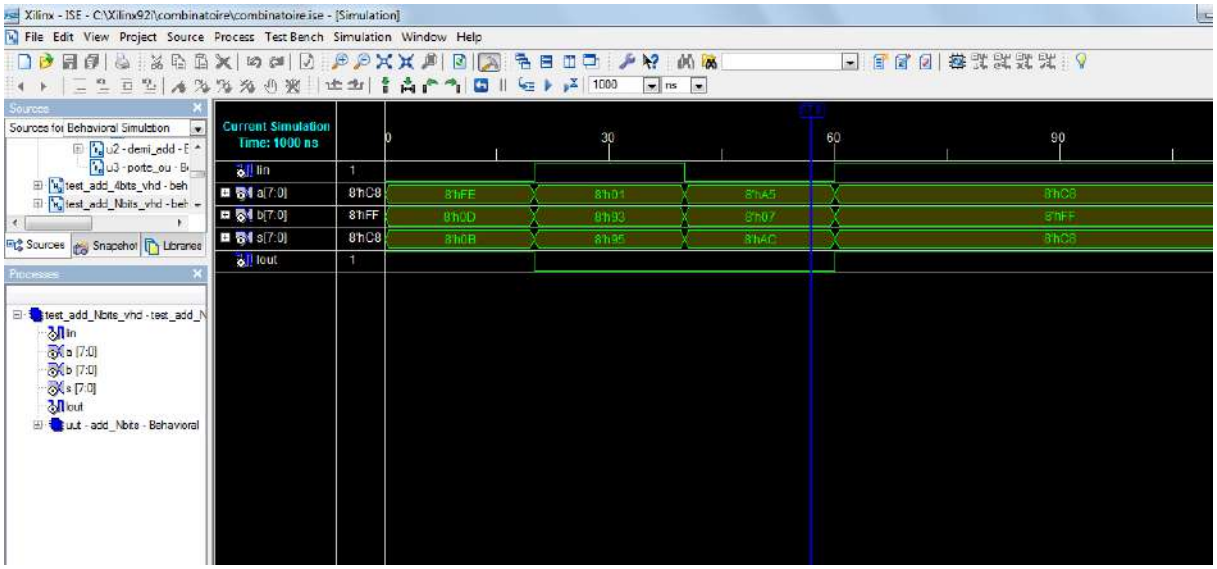


Figure 53 : la simulation d'add_Nbits (N=8)

VI. Bascule D

Réaliser la bascule D en utilisant un signal de remise à zéro synchrone active à l'état haut (Reset=1) et sensible au front montant de l'horloge Clk.

✓ Bascule D:

Alors le programme de la bascule_d est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity bascule_d is
    Port ( reset : in  STD_LOGIC;
           D: in  STD_LOGIC;
           Clk : in  STD_LOGIC;
           Q : out  STD_LOGIC;
           Qb : out  STD_LOGIC);
end bascule_d;
-- Déclaration de l'architecture avec une structure de test en séquentielle.
architecture Behavioral of bascule_d  is
begin
process (clk)
begin
if clk='1' and clk'event then
  if reset= '1' then Q<='0'; Qb<='1';
        else Q<= D; Qb<= not D;
end if;
end if;
end process;
end Behavioral;
```

Le fichier Test du VHDL : **test_bascule_d,** et le meme sauf pour la partie du **tb et tm process :**

```
tm: process
begin
clk<='0';wait for 10 ns;
clk<='1';wait for 10 ns;
end process;
tb : process
begin
reset<= '1'; D<= '0'; wait for 15 ns;
reset<= '1'; D<= '1'; wait for 15 ns;
reset<= '0'; D<= '0'; wait for 15 ns;
reset<= '0'; D<= '1'; wait for 15 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité à la **bascule_d** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

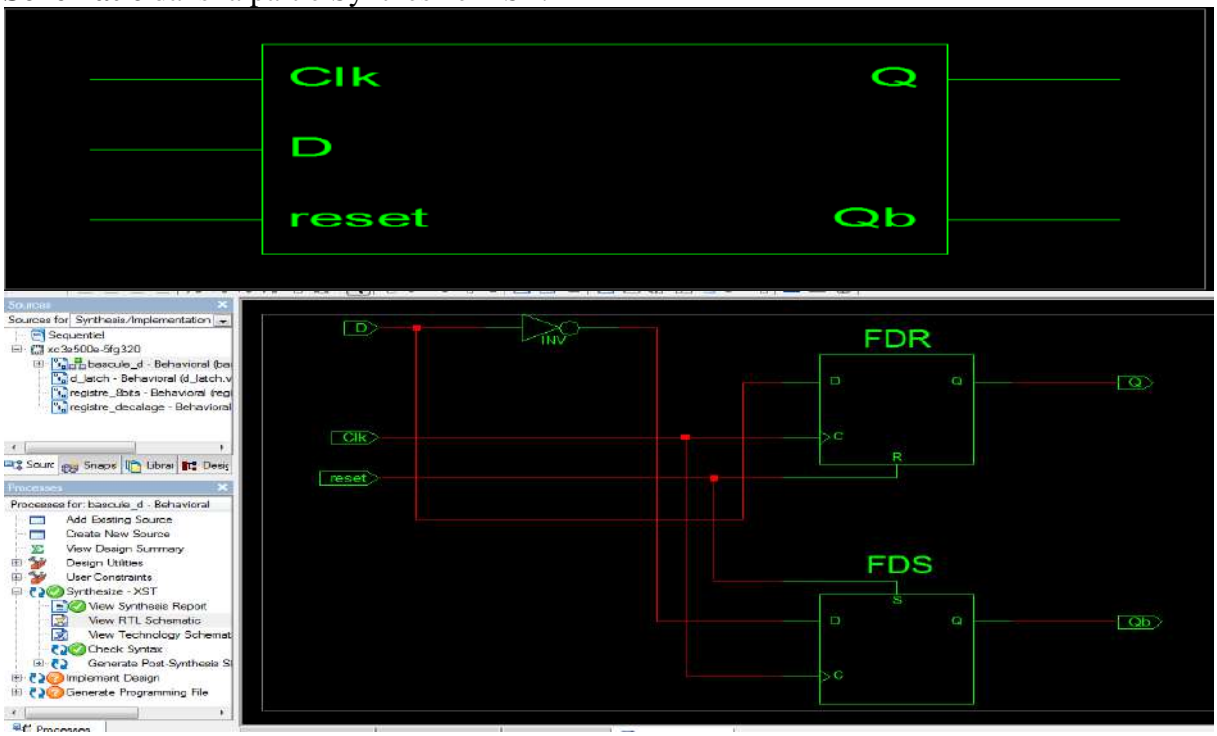


Figure 54 : le schéma de la bascule d

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

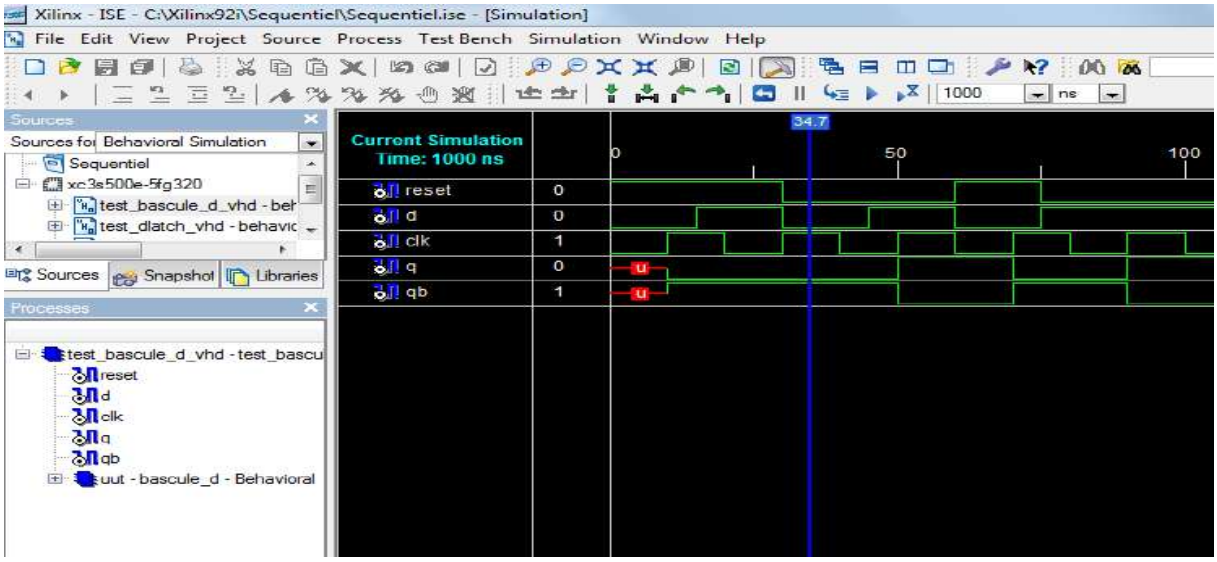


Figure 55 : la simulation de la bascule d

VII. REGISTRE

Réaliser un registre de 8 bits qui est constitué de bascules D avec un signal Reset asynchrone actif à l'état haut, le circuit sera sensible au front montant de l'horloge Clk.

✓ Registre à 8 bits:

Alors le programme d'un registre_8bits est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity registre_8bits is
   Port ( reset : in  STD_LOGIC;
          D: in  STD_LOGIC _VECTOR (7 downto 0);
          Clk : in  STD_LOGIC;
          Q : out  STD_LOGIC _VECTOR (7 downto 0);
          Qb : out  STD_LOGIC _VECTOR (7 downto 0));
end registre_8bits;
-- Déclaration de l'architecture avec une structure de test en séquentielle.
architecture Behavioral of registre_8bits  is
signal s,sb:std_logic_vector(7 downto 0);
begin
process (clk,reset,d)
begin

if reset= '1' then s<="00000000"; sb<="11111111";
else if clk='1' and clk'event then s<=d; sb<=not d;
else s<= s; sb<= sb;
end if;
end if;
end process;
q<=s; qb<=sb;
end Behavioral;
```

Le fichier Test du VHDL : **test_ registre_8bits,** et le même sauf pour la partie du **tb et tm process :**

```
tm: process
begin
clk<='0';wait for 10 ns;
clk<='1';wait for 10 ns;
end process;
tb : process
begin
reset<= '1'; D<= "01001001"; wait for 15 ns;
reset<= '1'; D<= "11111000"; wait for 15 ns;
reset<= '1'; D<= "00010101"; wait for 15 ns;
reset<= '0'; D<= "11001111"; wait for 15 ns;
reset<= '0'; D<= "00000111"; wait for 15 ns;
reset<= '0'; D<= "10101010"; wait for 15 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité au **registre_8bits** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

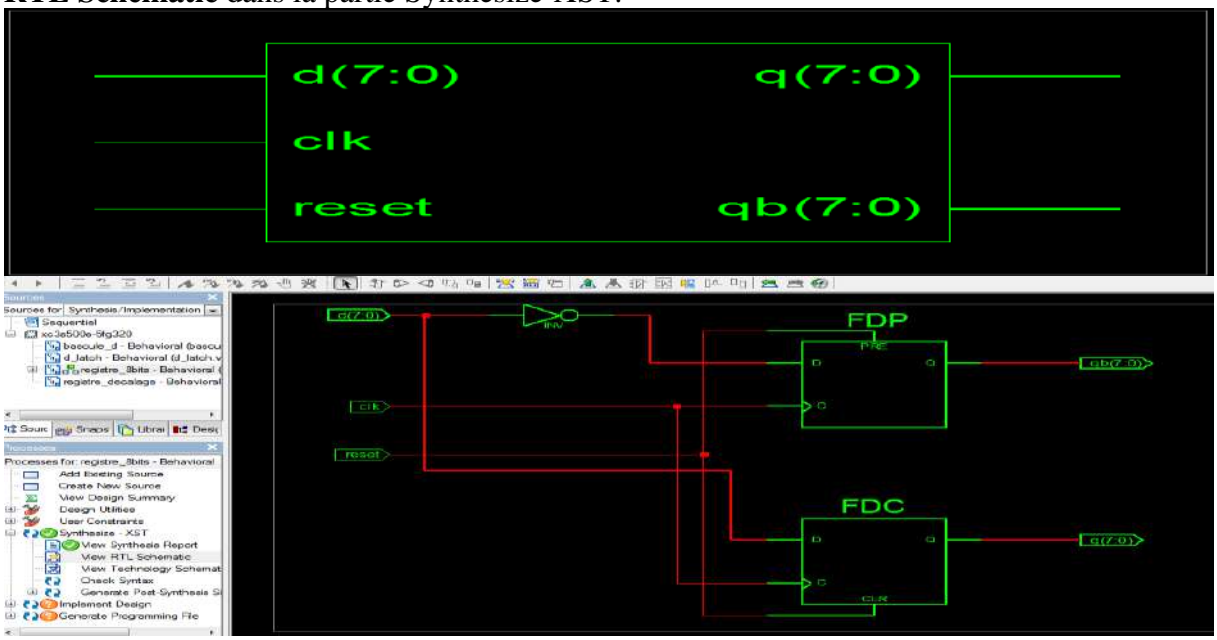


Figure 56 : le schéma du registre à 8bits

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

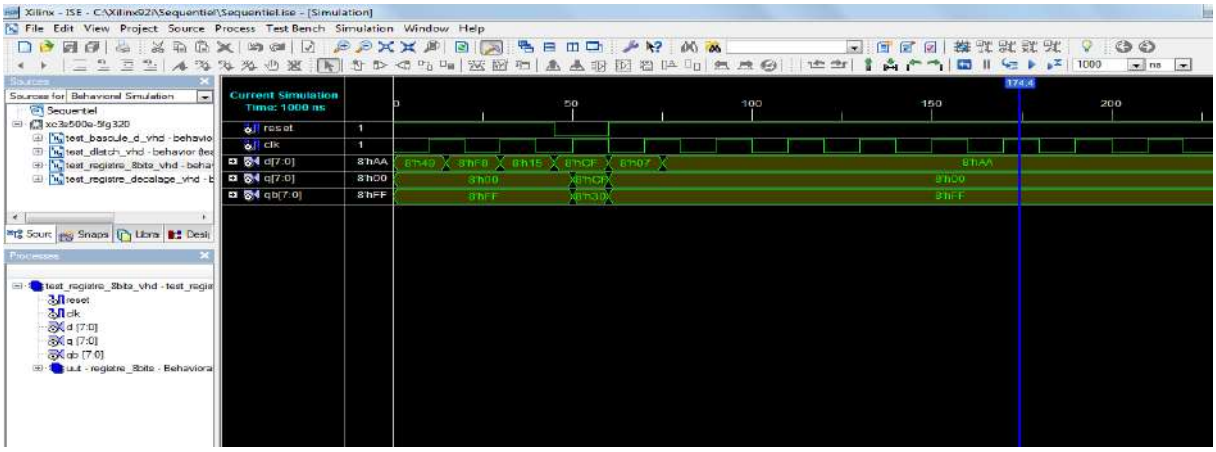


Figure 57 : la simulation du registre à 8bits

Réaliser un registre de N bits qui est constitué de bascules D avec un signal Reset asynchrone actif à l'état haut, le circuit sera sensible au front montant de l'horloge Clk.

✓ **Registre à N bits:**

Alors le programme d'un registre_Nbits est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity registre_Nbits is
   generic (n: natural:=16);
   Port ( reset : in  STD_LOGIC;
          D: in  STD_LOGIC _VECTOR (N-1 downto 0);
          Clk : in  STD_LOGIC;
          Q : out  STD_LOGIC _VECTOR (N-1 downto 0);
          Qb : out  STD_LOGIC _VECTOR (N-1 downto 0));
end registre_Nbits;
-- Déclaration de l'architecture avec une structure de test en séquentielle.
architecture Behavioral of registre_Nbits  is
signal s,sb:std_logic_vector(n-1 downto 0);
begin
process (clk,reset,d)
begin
if reset= '1' then s<=(others=>'0'); sb<=(others=>'1');
else if clk='1' and clk'event then s<=d; sb<=not d;
else s<= s; sb<= sb;
end if;
end if;
end process;
q<=s; qb<=sb;
end Behavioral;
```

Le fichier Test du VHDL : **test_ registre_Nbits,** et le même sauf pour la partie du **tb et tm process :**

tm: process

```
begin
clk<='0';wait for 10 ns;
clk<='1';wait for 10 ns;
end process;
tb : process
begin
reset<= '0'; D<= x"FF1E"; wait for 15 ns;
reset<= '0'; D<= x"2C35"; wait for 15 ns;
reset<= '1'; D<= x"1D6E"; wait for 15 ns;
reset<= '1'; D<= x"1123"; wait for 15 ns;
reset<= '0'; D<= x"097A"; wait for 15 ns;
reset<= '1'; D<= x"DE5A"; wait for 15 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité au **registre_Nbits** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

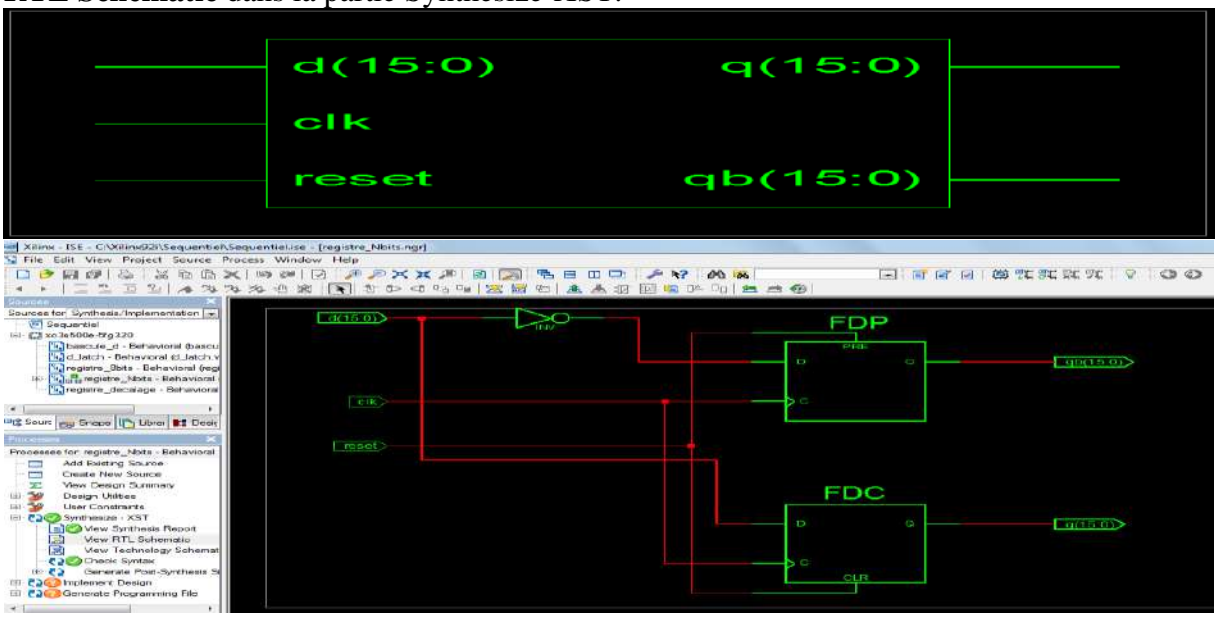


Figure 58 : le schéma du registre à Nbits (N=16)

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

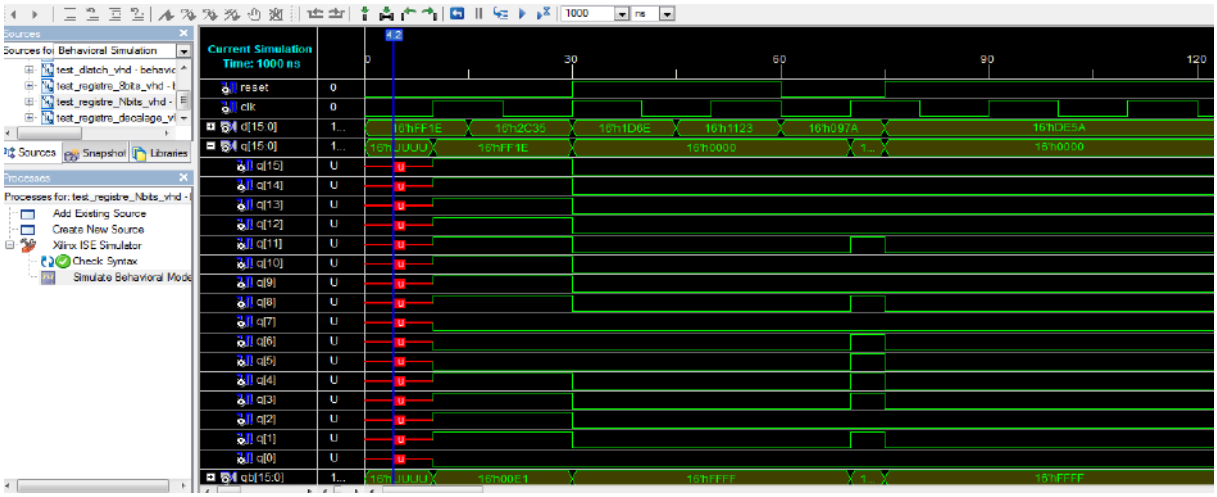


Figure 59 : la simulation du registre à Nbits (N=16)

VIII. Compteur/Décompteur

Réaliser un circuit compteur/décompteur de N bits (N est un paramètre générique) avec une remise à zéro asynchrone active à l'état haut. Le circuit sera sensible au front montant de l'horloge Clk, il fonctionnera comme compteur à l'état haut de Ud et comme décompteur à l'état bas de Ud.

✓ **Compteur/ décompteur à N bits:**

Alors le programme d'un compteur_decompteur_Nbits est comme suite :

```
-- Déclaration de la bibliothèque et des paquetages.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Déclaration de l'entité avec les entrées/sorties correspondants.
entity compteur_decompteur_Nbits is
   generic (n: natural:=8);
   Port ( reset : in  STD_LOGIC;
          Ud: in  STD_LOGIC;
          Clk : in  STD_LOGIC;
          Q : out  STD_LOGIC _VECTOR (N-1 downto 0));
end compteur_decompteur_Nbits;
-- Déclaration de l'architecture avec une structure de test en séquentielle.
architecture Behavioral of compteur_decompteur_Nbits  is
signal s:std_logic_vector(n-1 downto 0);
begin
process (clk,reset,Ud)
begin
if reset= '1' then s<=(others=>'0');
else if clk='1' and clk'event then
if ud = '1' then S <= S+1;
else S <= S-1;
end if; end if; end if;
end process;
q<=s;
```

```
end Behavioral;
```

Le fichier Test du VHDL : **test_ compteur_decompteur_Nbits,** et le même sauf pour la partie du **tb et tm process :**

```
tm: process
begin
clk<='0';wait for 10 ns;
clk<='1';wait for 10 ns;
end process;
tb : process
begin
reset<= '0'; ud<= '1'; wait for 15 ns;
reset<= '1'; ud<= '1'; wait for 15 ns;
reset<= '0'; ud<= '1'; wait for 15 ns;
reset<= '1'; ud<= '0'; wait for 15 ns;
reset<= '0'; ud<= '0'; wait for 15 ns;
reset<= '1'; ud<= '1'; wait for 15 ns;
wait;
end process;
end;
```

Et pour voir le schéma de la porte décrite, Donner la priorité au **compteur_decompteur_Nbits** en cliquons dessus, bouton droit (**set as Top module**) ensuite dans la rubrique **Processes** cliquer sur **View RTL Schematic** dans la partie Synthesize-XST.

Figure 60 : le schéma du compteur_decompteur_Nbits (N=8)

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

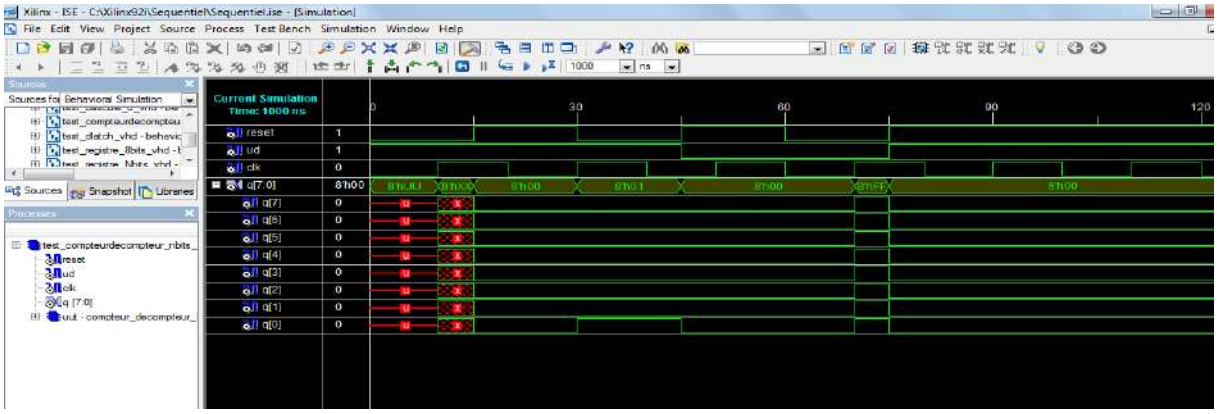
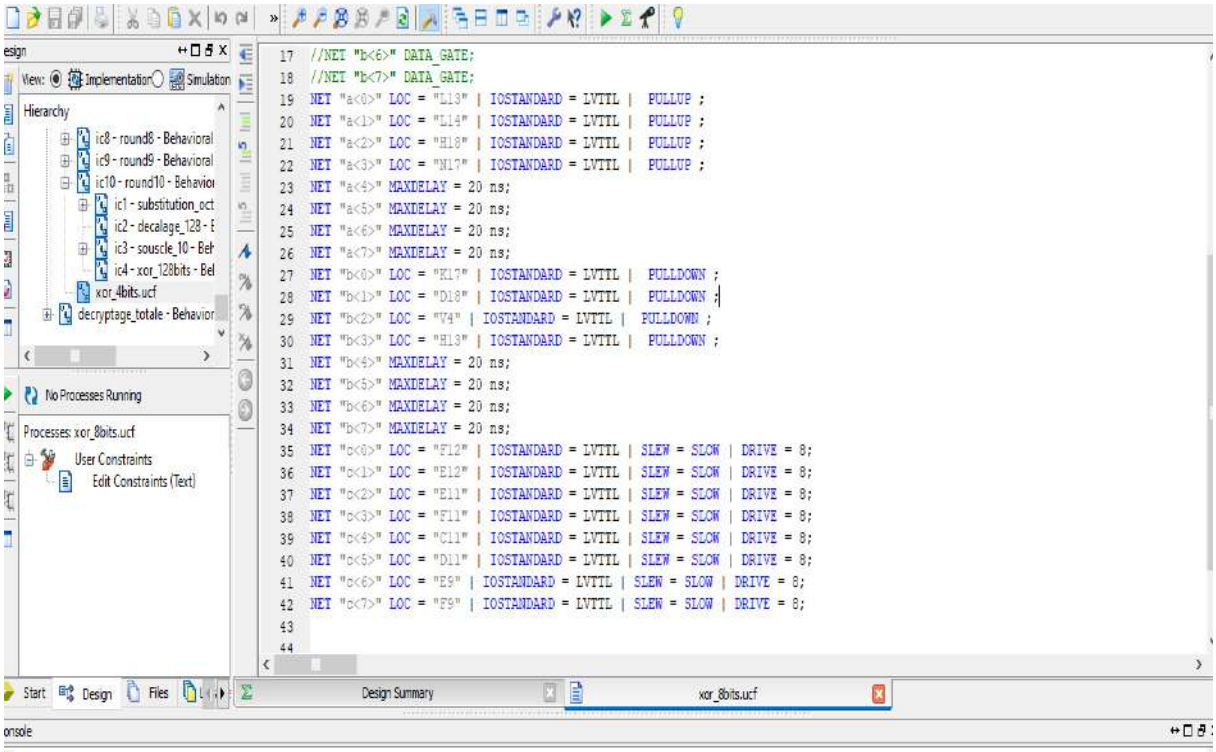


Figure 61 : la simulation du compteur_decompteur_Nbits (N=8)

IX. Exemple d'une implémentation de la porte XOR

Après avoir fait les mêmes étapes vu précédemment pour le VHDL module et le VHDL test Bench pour la porte Xor à 4 bit, nous allons à présent créer un fichier UCF (Impléméntation Constraints File) [13] afin de correspondre les entrées sorties du programme VHDL avec les différents Pins de la carte FPGA Spartan 3E XC3S500E.

La figure 62, nous montre comment faire la correspondance entre les entrées A et B à 4 bits avec les Pins des 4 switches et des 4 boutons poussoirs et la Sortie C à 4 bits aussi avec les Pins des 4 LEDs.



```
//NET "b<6>" DATA_GATE;
//NET "b<7>" DATA_GATE;
NET "a<0>" LOC = "L13" | IOSTANDARD = LVTTL |  PULLUP ;
NET "a<1>" LOC = "L14" | IOSTANDARD = LVTTL |  PULLUP ;
NET "a<2>" LOC = "H18" | IOSTANDARD = LVTTL |  PULLUP ;
NET "a<3>" LOC = "N17" | IOSTANDARD = LVTTL |  PULLUP ;
NET "a<4>" MAXDELAY = 20 ns;
NET "a<5>" MAXDELAY = 20 ns;
NET "a<6>" MAXDELAY = 20 ns;
NET "a<7>" MAXDELAY = 20 ns;
NET "b<0>" LOC = "K17" | IOSTANDARD = LVTTL |  PULLDOWN ;
NET "b<1>" LOC = "D18" | IOSTANDARD = LVTTL |  PULLDOWN ;
NET "b<2>" LOC = "V4" | IOSTANDARD = LVTTL |  PULLDOWN ;
NET "b<3>" LOC = "H13" | IOSTANDARD = LVTTL |  PULLDOWN ;
NET "b<4>" MAXDELAY = 20 ns;
NET "b<5>" MAXDELAY = 20 ns;
NET "b<6>" MAXDELAY = 20 ns;
NET "b<7>" MAXDELAY = 20 ns;
NET "c<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "c<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "c<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "c<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "c<4>" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "c<5>" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "c<6>" LOC = "E9" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "c<7>" LOC = "F9" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
```

Figure 62 : Fichier UCF pour XOR 4 bits

L'étape suivante consiste à faire une synthèse pour voir si le circuit est synthétisable ou pas. La figure 63 et 64 nous montre les étapes de synthèse et de l'implémentation du fichier UCF en binaire dans la carte FPGA.

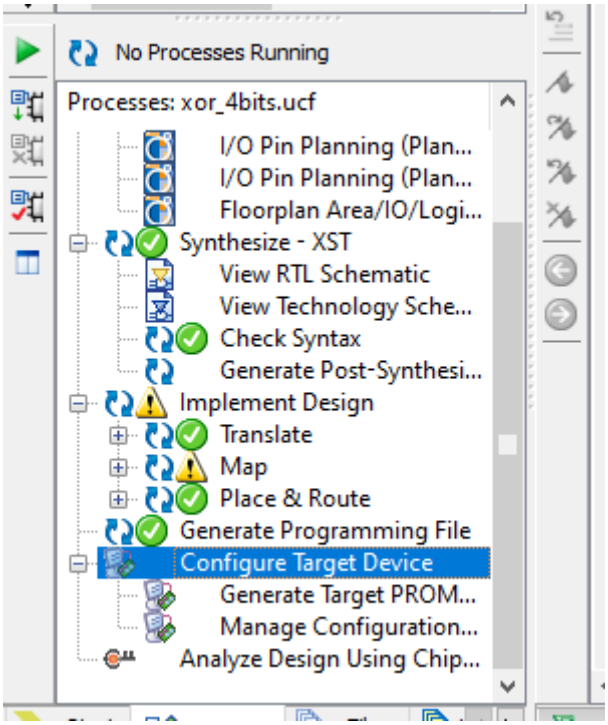


Figure 63 : La synthèse du XOR 4 bit

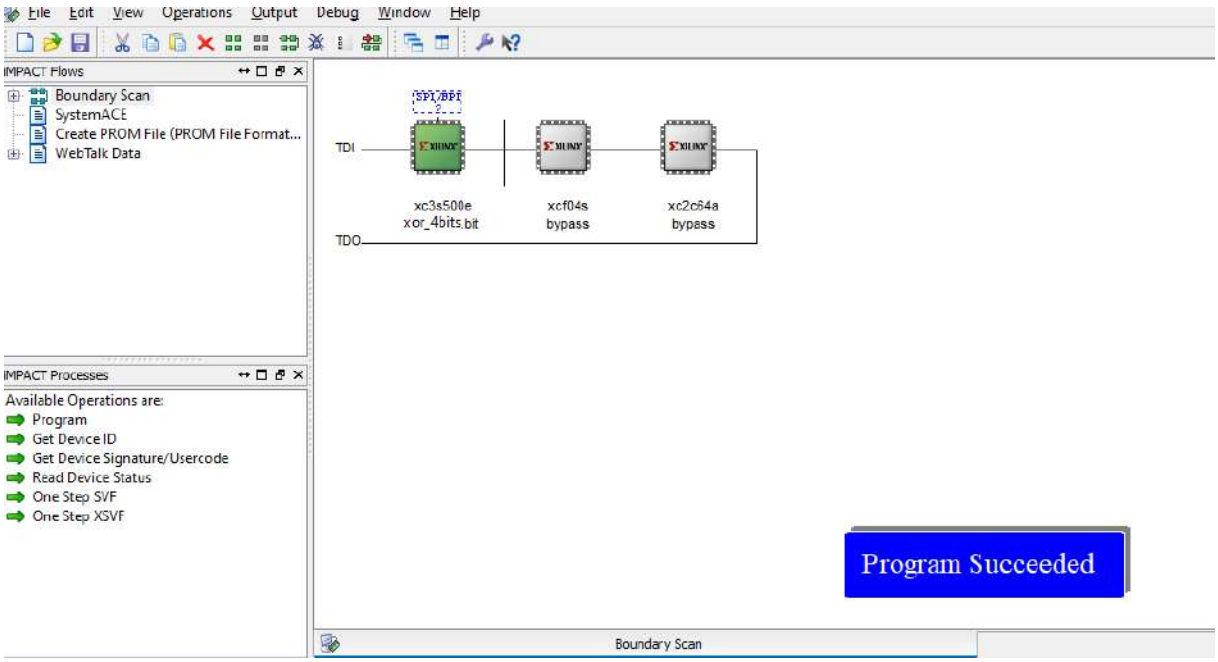


Figure 64 : Implémentation du fichier XOR 4 bits en bit sur la carte FPGA

La figure 65, nous montre le résultat final du programme VHDL pour le XOR 4 bits converti en binaire et traduisant les entrées/sorties en IOB grâce au fichier UCF et implémentant le fichier Xor_4bit en binaire dans les CLB des cartes FPGA.

L'exemple pris ici est 0000 représentant les boutons poussoir additionner en **XOR** avec les switches en position 0011 donc le résultat afficher est 0011 pour les 4 premiers LED.

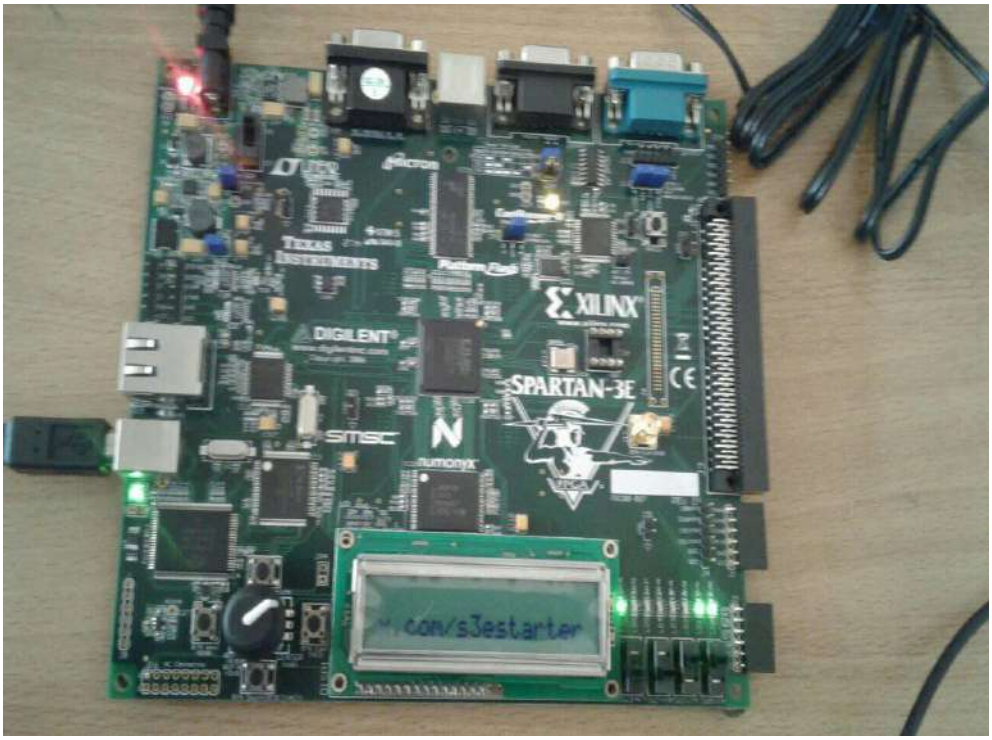


Figure 65 : Résultat de l'implémentation d'une porte XOR à 4bits sur une carte FPGA Spartan 3E XC3S500E.

Bibliographies

[1] : Alexis Polti, « Les HDL », COMELEC 2005-2014, http://hdl.telecom-paristech.fr/

[2] : GOLZE, Ulrich. VLSI chip design with the hardware description language VERILOG: An introduction based on a large RISC processor design. Springer Science & Business Media, 2013.

[3] : Jacques Weber, Sébastien Moutault, Maurice Meaudre, "Le langage VHDL : du langage au circuit, du circuit au langage", DUNOD, 2007.

[4]: T. BLOTIN, « le langage de description VHDL », Lycée Paul-Eluard 93206 SAINT-DENIS.

[5]: Remacle Matthieu, Schmitz Thomas, Pierlot Vincent, « Le language VHDL », Microélectronique, 24 février 2016.

[6] : AIRIAU, Roland, BERGÉ, Jean-Michel, OLIVE, Vincent, et al. VHDL: langage, modelisation, synthese. PPUR presses polytechniques, 1998.

[7]: Zimmermann, Reto. "VHDL library of arithmetic units." Proc. First Int. Forum on Design Languages (FDL'98), Lausanne, Switzerland. 1998.

[8]: Anderson, J., Shafer, L., Nahavandi, S., Zobrist, G., Arnold, G. W., Jacobson, D., & Malik, O. P. IEEE Press Editorial Board 2013.

[9] : François Verdier, « Les circuits FPGA Concepts de base, architecture et applications », Université de Cergy-Pontoise Laboratoire ETIS - UMR CNRS 8051.

[10] : Philip Simpson, La conception de systèmes avec FPGA - Bonnes pratiques pour le développement collaboratif Poche, Dunod, 2014.

[11]: Fabrice CAIGNET, « Etude des circuits logiques programmables Les FPGA », LAAS – CNRS.

[12] : AMINE B. CHOUKRI ADEL C ; "Implémentation sur FPGA des méthodes MPP "P&O" et "floue optimisée par les Algorithmes Génétiques" ; école national polytechnique Alger ; Algérie juin 2006.

[13] : Xilinx, "Spartan-3E FPGA Starter Kit Board User Guide" UG230 (v1.2) January 20, 2011. www.xilinx.com.

[14] : Nicolas MARQUES. « Méthodologie et architecture adaptative pour le placement efficace de taches matérielles de tailles variables sur des partitions reconfigurables » soutenue de thèse de doctorat de l'Université de Lorraine (Spécialité systèmes électroniques) le 26 novembre 2012.

[15]: BABU, Praveenkumar et PARTHASARATHY, Eswaran. Reconfigurable FPGA Architectures: A Survey and Applications. Journal of The Institution of Engineers (India): Series B, 2020, p. 1-14.

[16]: CROSS, Nigel. Science and design methodology: a review. Research in engineering design, 1993, vol. 5, no 2, p. 63-69.

[17] : Noëlle Lewis. « Méthodes de conception des circuits intègres analogiques et mixtes - Perspectives sur les systèmes électroniques en interaction avec le vivant ». Electronique. Université Bordeaux 1, 2010. fftel-01015800f.