COURS VHDL

```
ENTITY comparateur IS
  GENERIC (delai: TIME := 7 ns); -- delai par defaut, modifiable
  PORT (vecteur1, vecteur2 : IN bit_vector (3 DOWNTO 0);
        pg, egal, pp : IN bit;
        v1pgv2, v1egalv2, v1ppv2 : OUT bit);
END comparateur;

ARCHITECTURE comportementale OF comparateur IS
BEGIN
  PROCESS (vecteur1, vecteur2, pg, egal, pp)
  BEGIN

    v1pgv2 <= '0';      -- initialisation obligatoire,
    v1egalv2 <= '0';    -- sinon des bascules sont
    v1ppv2 <= '0';      -- rajoutees pour synchroniser les
                        -- signaux

    IF vecteur1 > vecteur2 THEN
        v1pgv2 <= '1' AFTER delai;
        v1egalv2 <= '0' AFTER delai;
        v1ppv2 <= '0' AFTER delai;
    ELSIF vecteur1 = vecteur2 THEN
        v1pgv2 <= pg AFTER delai;
        v1egalv2 <= egal OR NOT(pg OR pp) AFTER delai;
        v1ppv2 <= pp AFTER delai;
    ELSIF vecteur1 < vecteur2 THEN
        v1pgv2 <= '0' AFTER delai;
        v1egalv2 <= '0' AFTER delai;
        v1ppv2 <= '1' AFTER delai;
    END IF;
  END PROCESS;
END comportementale;
```

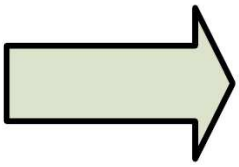




TABLE DES MATIERES

MODELISATION VHDL SONAR

1) Présentation page N°3
2) Introduction page N°4
3) Principe transmission série page N°4
4) Schéma page N°6

NOTION DE BASE LANGAGE VHDL

1) Introduction page N°7
2) Structure d'une description VHDL page N°8
3) Les instructions de base mode concurrent page N°12

4) Les instructions du mode séquentiel process page N°16

5) Description structurelle page N°21

MODELISATION VHDL MACHINE ETAT page N°23

1) notion automate page N°23
2) Machine d'états page N°23
3) Le diagramme d'état ou de transition page N°24
4) modélisation VHDL page N°25

MODELISATION COMPORTEMENTALE RS232

1) principe page N°26
2) machine d'état page N°26
3) questions page N°29

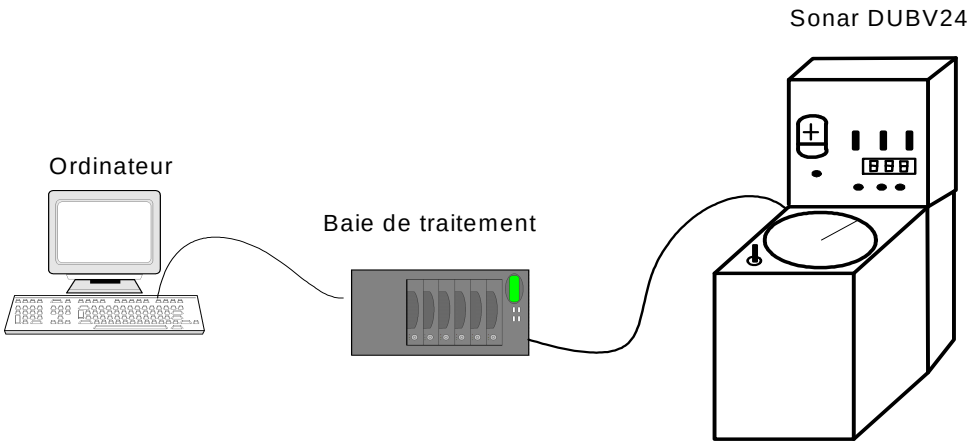
Utilisation Du logiciel ISPLEVER de Lattice page N°34

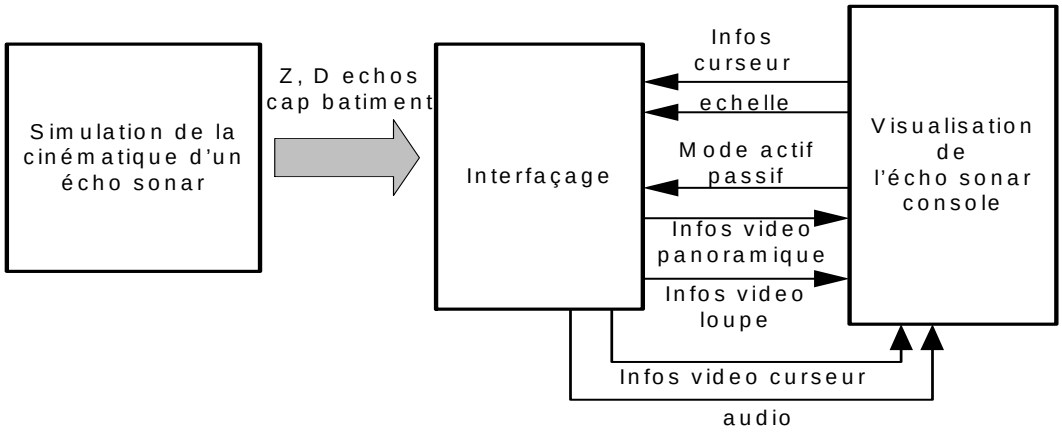
CPLD de type ISPLSI

Modélisation VHDL d'une UART pour un SIMULATEUR SONAR DUBV24

1) présentation

Le simulateur SONAR doit permettre d'assurer la gestion d'une console de visualisation afin d'assurer la formation des opérateurs sonar.
Il devra assurer un rendu audio et une visualisation d'un écho sonar généré par le PC. Le matériel est le suivant :





Rôle ordinateur	Rôle : interface	Rôle console
Simuler la cinématique d'une cible et transmettre un azimut distance correspondant vers l'interface.	Récupérer un azimut distance de l'écho simulé en provenance du PC. Récupérer les réglages affichés sur la console Générer les trames de balayage vidéo des scopes panoramique et loupe avec écho simulé. Recopie des informations azimut distance sur la console Distribuer l'audio correspondante.	Visulisation et reglages des paramètres de fonctionnement

2) Introduction

Les informations azimuth distance de l'écho envoyées du PC vers l'interface circulent sur une liaison série du type RS232
Cette étude à pour but de réaliser l'interface permettant de communiquer avec le port série du PC. La liaison série du PC est à la norme RS232. Cependant, les spécifications de la norme RS232 ne seront pas toutes implémentées :

- 1 bit de Start
- 8 bits de données
- 1 bit de stop
- pas de bit de parité
- Fréquence de transfert : 9600 bps

Pour concevoir une fonction à la norme RS232, il faut programmer en vhdl ce que l'on appelle une UART. Une UART (universal asynchronous receiver and transmitter) permet une communication asynchrone avec des périphériques externes à l'ordinateur comme un modem, un autre ordinateur (via un câble null-modem). De plus, une UART permet la communication avec un périphérique sans signal de synchronisation. L'UART sert généralement d'interface de communication entre un périphérique et le port série de l'ordinateur. L'UART sera implémenté dans un circuit programmable CPLD (voir schéma structurel page N°6)

3) Principe d'une transmission série

Une UART est constitué de deux fonctions distinctes à savoir la réception et l'émission de données. Le schéma ci-dessous illustre le protocole de communication entre deux UARTs. Pour commencer les données séries sont transmises par blocs de 10 bits. En effet, un octet de 8 bits est encapsulé entre un bit de Start à l'état bas et un bit de stop à l'état haut.

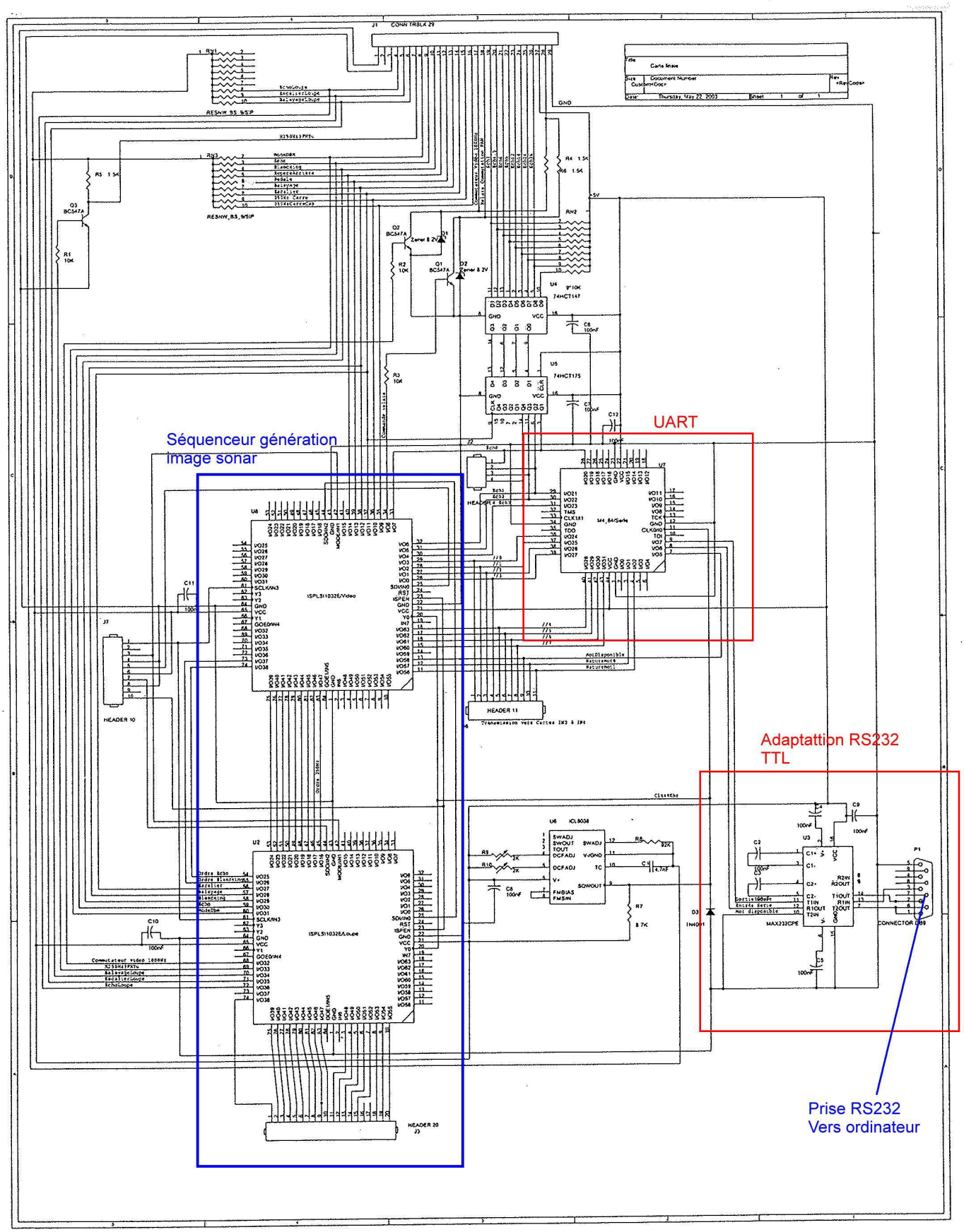


Après émission du bit de Start, vient le bit de poids faible, la fin de la transmission se terminant par le bit de poids fort. Enfin entre deux transmissions successives, la ligne de transmission est à l'état haut. Nous devons aussi implémenter des signaux de communications avec le séquenceur.

Objectifs :

A l'issue de ce cours et des séances de TP, l'étudiant devra être capable :

- De coder toutes fonctions combinatoires ou séquentielles à l'aide du langage VHDL
- D'élaborer un diagramme d'état afin de répondre à l'application envisagée
- D'établir une modélisation VHDL à partir du diagramme d'état.
- D'implémenter le code VHDL dans un circuit programmable grâce aux outils de développement de la société Lattice
- De mettre en œuvre des méthodes de simulation ou de test afin de vérifier si la fonction réalisée répond bien à l'application envisagée.


Séquenceur génération
image sonar
UART
Adaptattion RS232
TTL
Prise RS232
Vers ordinateur

Notion de base sur le langage VHDL
Pour la synthèse logique

I) Introduction.

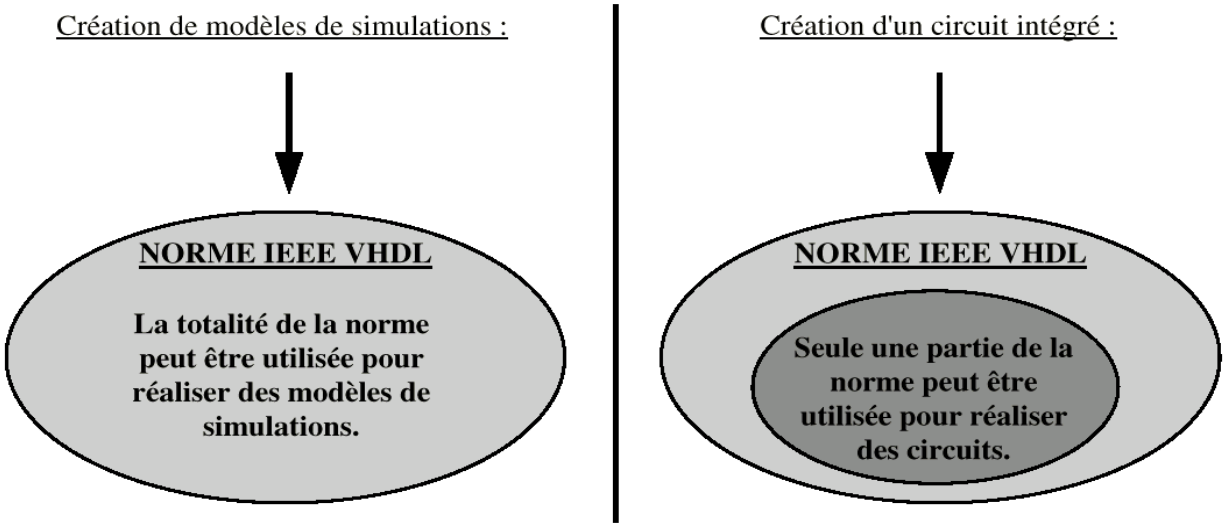
VHDL est un langage de description de structures électroniques (V.H.D.L. very Hardware Description language) VHDL ?

Ce langage a été écrit dans les années 70 pour réaliser la simulation de circuits électroniques. On l'a ensuite étendu en lui rajoutant des extensions pour permettre la conception (synthèse) de circuits logiques programmables (P.L.D. Programmable Logic Device).

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

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

Les sociétés de développement et les ingénieurs ont voulu s'affranchir des contraintes technologiques des circuits. Ils ont donc créé des langages dits de haut niveau à savoir VHDL et VERILOG. Ces deux langages **font abstraction des contraintes technologies des circuits PLDs**. Ils permettent au code écrit d'être **portable**, c'est à dire qu'une description écrite pour un circuit peut être facilement utilisée pour un autre circuit. La norme qui définit la syntaxe et les possibilités offertes par le langage de description VHDL est très importante. une partie de cette norme est utilisée pour la réalisation de circuits programmable à cause de la limite actuelle de la synthèse logique. Dans ce polycopié seul sera étudié une partie de la norme du langage VHDL, celle qui permet la synthèse logique d'un circuit programmable.
On donne en annexe 1 les différentes méthodes de développement de circuit programmable.



II) Structure d'une description VHDL simple.

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

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

Exemple : Un démultiplexeur 2 vers 4.

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

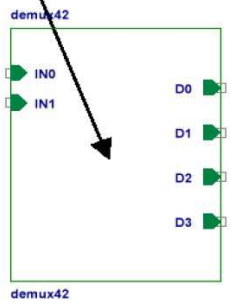
Déclaration des bibliothèques

Commentaires, en VHDL ils commencent par - -

```
-- démultiplexeur / décodeur
-- Deux vers quatre
```

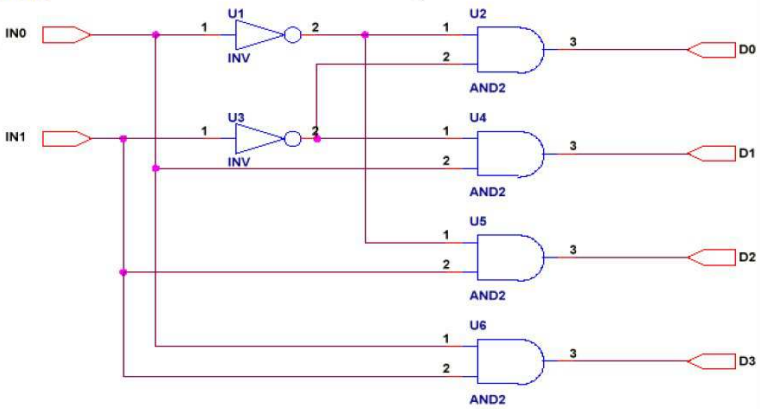
Déclaration de l'entité du démultiplexeur
Correspondance schématique

```
entity DEMUX2_4 is
      port(IN0, IN1: in std_logic;
           D0, D1, D2, D3: out std_logic);
end DEMUX2_4;
```



Déclaration de l'architecture du démultiplexeur
Correspondance schématique

```
architecture DESCRIPTION of DEMUX2_4 is
begin
      D0 <= (not(IN1) and not(IN0));
      D1 <= (not(IN1) and IN0);
      D2 <= (IN1 and not(IN0));
      D3 <= (IN1 and IN0);
end DESCRIPTION;
```



- « l'ENTITY » est la boîte avec ses entrées et ses sorties physiques. « L'architecture » contient la description du fonctionnement de la boîte et possède son propre nom.

II.1) Déclaration des bibliothèques.

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

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

-- cette dernière bibliothèque est souvent utilisée pour l'écriture de compteurs car elle permet de transformer un signal du type entier en un signal du type bit et réciproquement.
La directive *Use* permet de sélectionner les bibliothèques à utiliser.

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

Elle permet de définir le *NOM* de la description VHDL ainsi que les entrées et sorties utilisées, l'instruction qui les définit c'est *port* :
Syntaxe:
entity NOM_DE_L_ENTITE is
port (Description des signaux d'entrées /sorties …);
end NOM_DE_L_ENTITE;

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

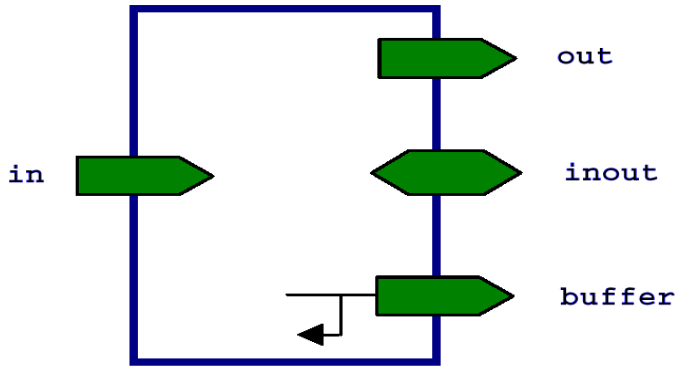
Remarque : Après la dernière définition de signal de l'instruction *port* il ne faut jamais mettre de point virgule.
On doit définir pour chaque signal : le *NOM_DU_SIGNAL*, le *sens* et le *type*.

II.2.1) Le NOM_DU_SIGNAL.

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

II.2.2) Le *SENS* du signal.

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



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

II.2.3) Le *TYPE*.

Le *TYPE* utilisé pour la synthèse des signaux d'entrées / sorties est :
- le *std_logic* pour un signal.
- le *std_logic_vector* pour un bus composé de plusieurs signaux.

Par exemple un bus bidirectionnel de 5 bits s'écrira :
LATCH : inout std_logic_vector (4 downto 0) ;
Où *LATCH(4)* correspond au MSB et *LATCH(0)* correspond au LSB.

Les valeurs que peuvent prendre un signal de type *std_logic* sont :
- *'0' ou 'L'* : pour un niveau bas.
- *'1' ou 'H'* : pour un niveau haut.
- *'Z'* : pour état haute impédance.
- *'x'* : Quelconque, c'est à dire n'importe quelle valeur.

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

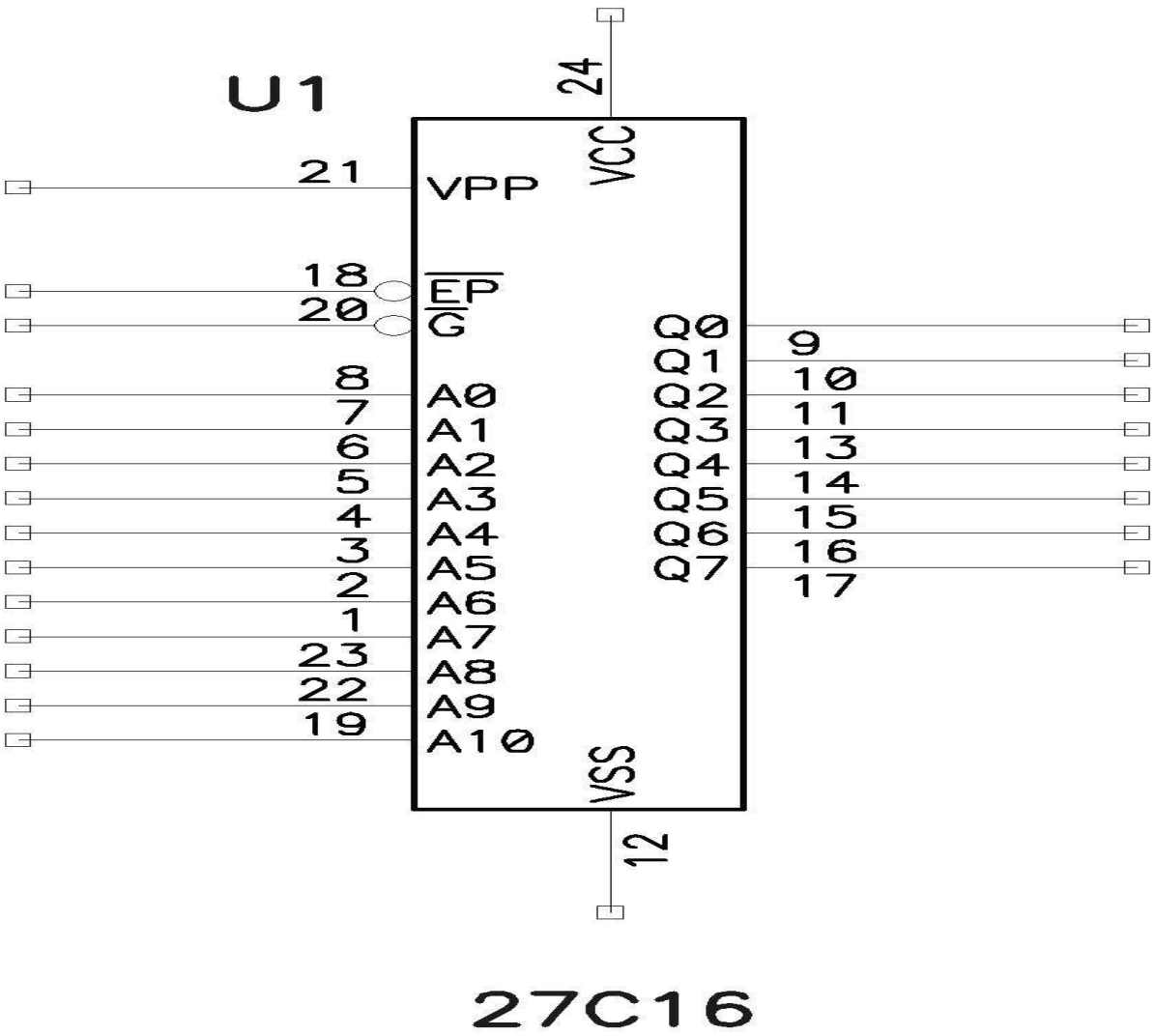
L'architecture **décrit le fonctionnement souhaité pour un circuit** ou une partie du circuit.
En effet le fonctionnement d'un circuit est généralement décrit par plusieurs modules VHDL. Il faut comprendre par module le couple *ENTITE/ARCHITECTURE*. Dans le cas de simples PLDs on trouve souvent un seul module.
L'architecture établit à travers les instructions les relations entre les entrées et les sorties. On peut avoir un fonctionnement purement combinatoire, séquentiel voire les deux séquentiel et combinatoire.

II.4) Exemple

Soit le schéma d'une mémoire, complétez le code VHDL suivant :

Library ieee;
Use ieee.std_logic_1164.all;
entity is
PORT (

End ;



III) Les instructions de base (mode « concurrent »), logique combinatoire.

Qu'est ce que le mode « *concurrent* » ? **Pour une description VHDL toutes les instructions sont évaluées et affectent les signaux de sortie en même temps.** L'ordre dans lequel elles sont écrites n'a aucune importance. En effet la description génère des structures électroniques, c'est la grande différence entre une description VHDL et un langage informatique classique.
Dans un système à microprocesseur, les instructions sont exécutées les unes à la suite des autres.

III.1) Les opérateurs.

III.1.1) L'affectation simple : <=

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

Exemple avec des portes logiques : *S1 <= E2 and E1 ;*
- Pour les signaux composés de plusieurs bits on utilise les guillemets " … "
- Les bases numériques utilisées pour les bus peuvent être :

BINAIRE, exemple : *BUS <= "1001" ; -- BUS = 9 en décimal*
HEXA, exemple : *BUS <= X"9" ; -- BUS = 9 en décimal*
OCTAL, exemple : *BUS <= O"11" ; -- BUS = 9 en décimal*

III.1.2) Opérateurs logiques.

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

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

III.1.3) Opérateurs arithmétiques.

Opérateur	VHDL
ADDITION	+
SOUSTRACTION	-
MULTIPLICATION	*
DIVISION	/

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

Use ieee.numeric_std.all ;
Use ieee.std_logic_arith.all ;

Exemples :
S1 <= A – 3 ; -- S1 = A – 3
-- On soustrait 3 à la valeur de l'entrée / signal A
S1 <= S1 + 1 ; -- On incrémente de 1 le signal S1

III.1.4) Opérateurs relationnels.

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

- *when … else …*
- *with …. Select ….*

Voir ci-dessous.

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

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

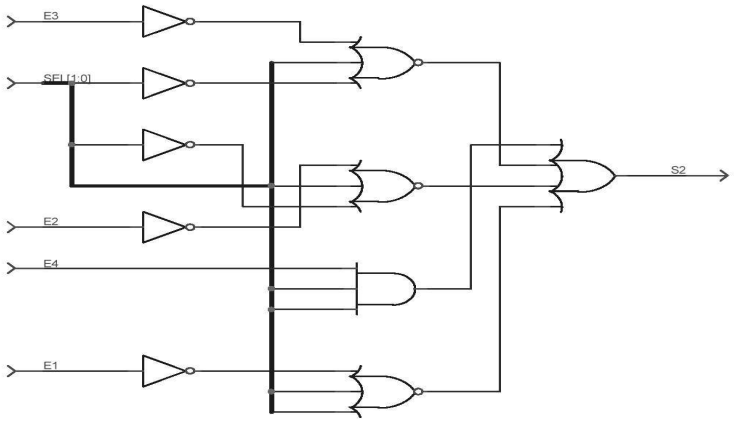
III.2.1) Affectation conditionnelle :

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

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

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

Schéma correspondant après synthèse:



III.2.2) Affectation sélective :

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

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

Exemple N°1 :
-- Multiplexeur 4 vers 1
with SEL select
S2 <= E1 when "00",
E2 when "01",

E3 when "10",
E4 when "11",
'0' when others;

Remarque: *when others* est nécessaire car il faut toujours définir les autres cas du signal de sélection pour prendre en compte toutes les valeurs possibles de celui-ci.
Le schéma de synthèse reste identique au cas précédent.

Exercice

Proposez une description VHDL du circuit suivant :
On additionne 2 mots A et B si le bit X est à 0 sinon si X est au '1' logique on réalise une soustraction A-B
A et B 2 octets

IV) Les instructions du mode séquentiel.

IV.1) Définition d'un PROCESS.

Un process est une partie de la description d'un circuit dans laquelle **les instructions sont exécutées séquentiellement c'est à dire les unes à la suite des autres**.
Il permet d'effectuer des opérations sur les signaux en utilisant les instructions standard de la programmation structurée comme dans les systèmes à microprocesseurs.
L'exécution d'un process **est déclenchée par un ou des changements d'états de signaux logiques**. Le nom de ces signaux est défini dans la liste de sensibilité lors de la déclaration du process.

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

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

L'exécution d'un process est concurrente même si les instructions qu'il contient sont séquentielles.
Règles de fonctionnement d'un process :

1) L'exécution d'un *process* a lieu à chaque changement d'état d'un signal de la liste de sensibilité.
2) Les instructions du *process* s'exécutent séquentiellement.
3) Les changements d'état des signaux par les instructions du *process* sont pris en compte à la fin du *process*.

IV.2) Les deux principales instructions utilisées dans un process.

L'assignation conditionnelle

```
if condition then
     instructions
[elsif condition then instructions]
[else instructions]
end if ;
Exemple:
if (RESET='1') then SORTIE <= "0000";
```

L'assignation sélective

```
case signal_de_slection is
when valeur_de_sélection => instructions
[when others => instructions]
end case;

Exemple:
case SEL is
          when "000" => S1 <= E1;
          when "001" => S1 <= '0';
          when "010" | "011" => S1 <='1';
          -- La barre | permet de réaliser
          -- un ou logique entre les deux
          -- valeurs "010" et "011"
          when others => S1 <= '0';
end case;
```

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

IV.3) Les boucles

*a) La boucle « **for..in..to..loop..end loop** »*

```
for N in X to Y loop
INSTRUCTION ;
end loop ;
```

Exécute les instructions pour N variant de X à Y

b) La boucle « while..loop..end loop »

```
while CONDITION loop
INSTRUCTION ;
end loop ;
```

Exécute les instructions tant que la condition est vraie

L'habitude de la programmation des microprocesseurs peut inciter à utiliser les instructions de boucles pour gérer le temps. Quoi de plus naturel, semble-t-il, que de créer un compteur, par exemple, au moyen dune structure de boucle? II n'en est rien. Rappelons que la boucle fondamentale qui permet de gérer le temps est celle que constitue le corps d'un processus lui-même ; **les instructions de boucles servent a créer des structures spatiales répétitives, à décrire par des algorithmes des opérations qui peuvent très bien être instantanées.**

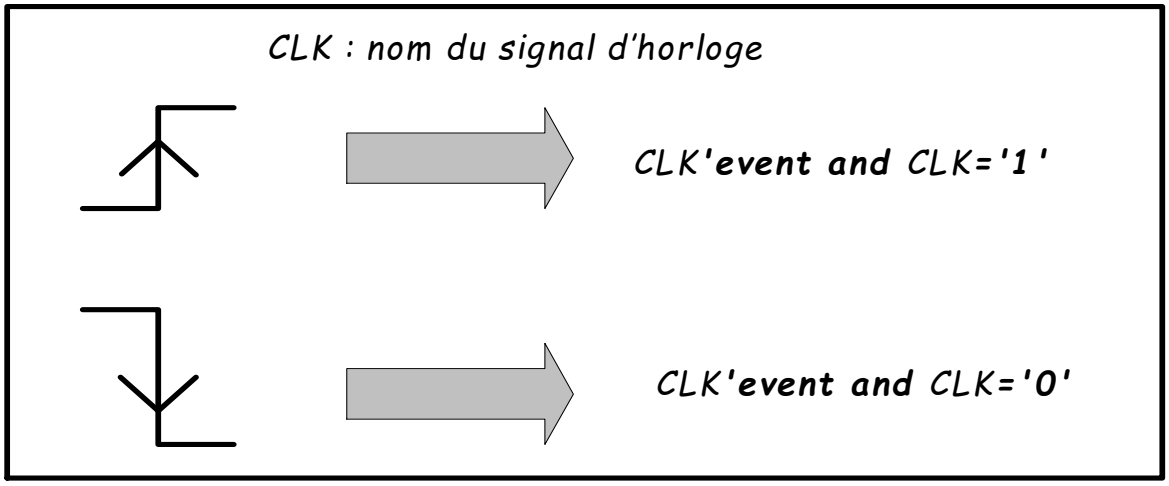
IV.4) La variable

variable : la variable est généralement utilisée comme index dans la génération de boucle et n'a habituellement pas de correspondance avec un signal physique. Elle ne peut être déclarée que dans un process. Théoriquement une variable **change de valeur dès l'affectation**, tandis qu'un signal ne change qu'en sortie d'un process. L'affectation d'une variable est l'opérateur :=

exemple :

```
Architecture parite of ouex is
Begin
Process (a)
Variable parite : bit;
Begin
Parite :='0';
-------
```

IV.5) Déclaration d'un front dans un process

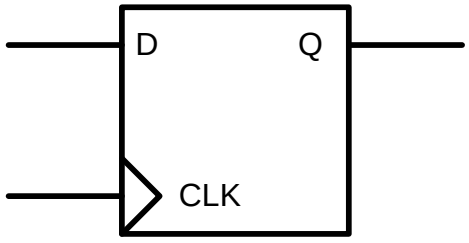


IV.6) Exemples de *process* :

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

Complétez le code VHDL suivant :

```
Library ieee;
Use ieee.std_logic_1164.all;
entity BASCULED is
port (
        : in std_logic;
         : out std_logic);
end BASCULED;
architecture DESCRIPTION of BASCULED is
begin
PRO_BASCULED : process (          )
begin
if (                                  ) then
       <=   ;
end if;
end process PRO_BASCULED;
end DESCRIPTION;
```



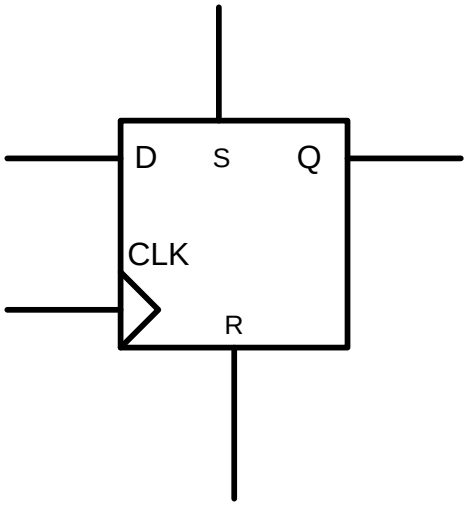
- Seul le signal *CLK* fait partie de la liste de sensibilité. D'après les règles de fonctionnement énoncées précédemment, seul un changement d'état du signal *CLK* va déclencher le process et par conséquent évaluer les instructions de celui-ci.

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

exercice

on rajoute à la bascule D précédentes 2 entrées synchrones :
- Une de remise à 0 R active au 1 logique.
- L'autre de mise à 1 S active au 1 logique.

Proposez une description VHDL pour ce type de bascule D (R prioritaire sur S)
Les 2 entrées ne sont plus synchrones mais asynchrones proposez une nouvelle description pour cette bascule D, R prioritaire sur S.



IV.7) Les compteurs :

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

IV.7.1) Compteur simple :

Complétez le code VHDL suivant :

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
entity Compteur is
PORT (
CLOCK : in std_logic;       -- Déclenche sur un front montant
Q : inout std_logic_vector(3 downto 0));
End compteur;
architecture DESCRIPTION of compteur is
begin
process (          )
begin
if (                        ) then
    <=          ;
end if;
end process;
end DESCRIPTION;
```

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

Cette description fort simple, appelle quelques commentaires :

- L'incrémentation du compteur est réalisée par l'opérateur + associé à la valeur 1. Cela est logique, mais elle l'est beaucoup moins pour les PLDs et les synthétiseurs, pourquoi ? La réponse est que les entrées et sorties ainsi que les signaux sont déclarés de type *std_logic* ou *std_logic_vector*, et par conséquent on ne peut pas leur associer de valeur entière décimale.

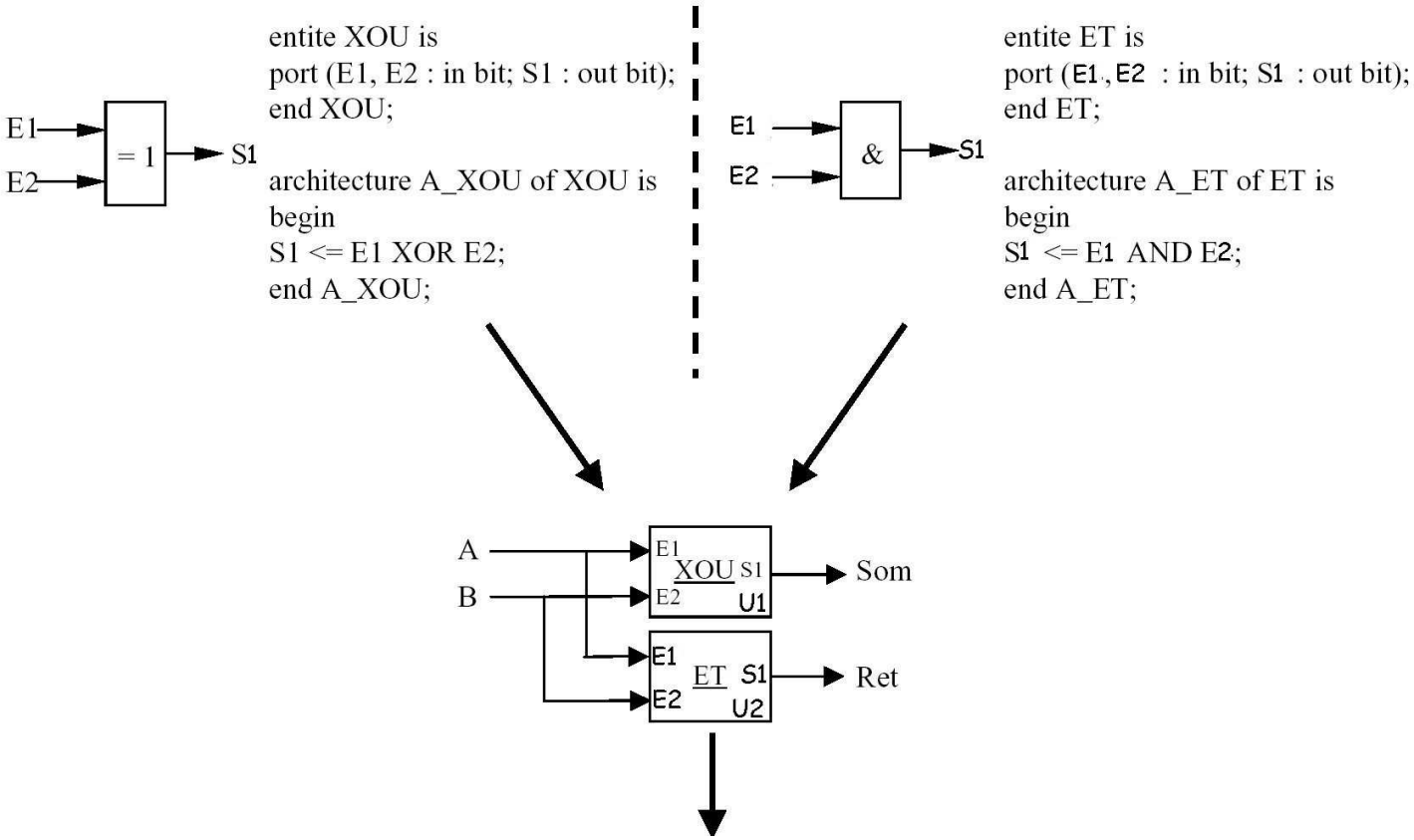
- Un signal peut prendre comme valeur les états '1' ou '0' et un bus n'importe quelle valeur, du moment qu'elle est écrite entre deux guillemets "1010" ou X"A" ou o"12", mais pas une valeur comme par exemple 1,2,3,4. Ces valeurs décimales sont interprétées par le synthétiseur comme des valeurs entières (integer), on ne peut pas par défaut additionner un nombre entier 1 avec un bus de type électronique (*st_logic_vector*), c'est pour cela que l'on rajoute dans la partie déclaration des bibliothèques les lignes :

Use ieee.numeric_std.all;
Use ieee.std_logic_unsigned.all;
Ces deux bibliothèques ont des fonctions de conversions de types et elles permettent d'associer un entier avec des signaux électroniques. Elles permettent d'écrire facilement des compteurs, décompteurs, additionneurs, soustracteurs, …..

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

- Le signal Q est déclaré dans l'entité de type *inout*, cela est logique car il est utilisé à la fois comme entrée et comme sortie pour permettre l'incrémentation du compteur.

V) Description structurelle



Donnez une description structurelle d'une bascule RS
Entity NOR is
Port (a ,b : in bit ; a_NOR_B : out bit) ;
End NOR ;

Déclaration du composant XOU.

Déclaration du composant ET.

Spécification de l'entité et de l'architecture ‡ utiliser pour les composants ET et XOU.
pas nécessaire sous ISPLEVER

Câblage des composants XOU et ET appelé U1 et U2.

```
Entité add is
port (A, B : in bit; Som, Ret : out bit);
end add;
architecture Arch_structurelle of add is
component XOU
port (E1, E2 : in bit; S : out bit);
end component;
component ET
port (E1, E2 : in bit; S : out bit);
end component;
for U1 : XOU use entity work.XOU(A_XOU);
for U2 : ET use entity work.ET(A_ET);
begin
U1 : XOU port map (A, B, Som);
U2 : ET port map (A, B, Ret);
end arch_structurelle;
```

```
architecture A_NOR of NOR
begin
a_NOR_b <= not (a or b) ;
end A_NOR ;
```

Modélisation VHDL d'une machine d'état

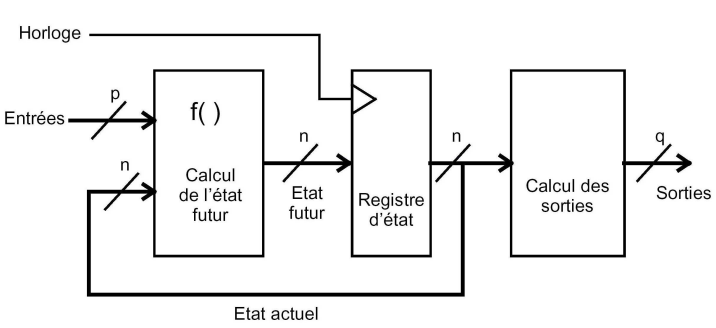
1) notion sur les automates

On peut utiliser un automate dans toutes les applications où la gestion d'un processus, l'utilisation d'une ressource, un traitement sur de données peuvent être décrits par une **chronologie** qui fait intervenir un **nombre fini** d'opérations distinctes conditionnées par des entrées extérieures.
Les automates, peuvent prendre différentes appélations :
automates finis (les théoriciens de l'automatique et des réseaux), **machines à nombre fini d'états** (les concepteurs de circuits numériques), **séquenceurs** (les concepteurs d'unités centrales d'ordinateurs et certains automaticiens) ces objets peuvent être réalisés par du matériel, des circuits, ou du logiciel, des programmes, ou par un savant mélange des deux ingrédients précédents.
Ces automates sont soit réalisées :
- **Logiciellement** , si la vitesse des processeurs le permet
- **Matériellement** par des circuits spécialisés si la vitesse de traitement demandée dépasse les capacités des processeurs.

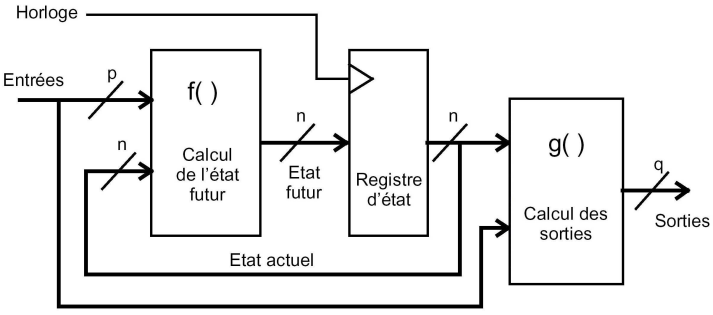
On se placera par la suite dans le rôle du concepteur de circuit numérique, l'automate sera alors considéré comme une machine d'état.

2) Les machines d'états

On distingue 2 types de machines d'état:



MACHINE DE MOORE



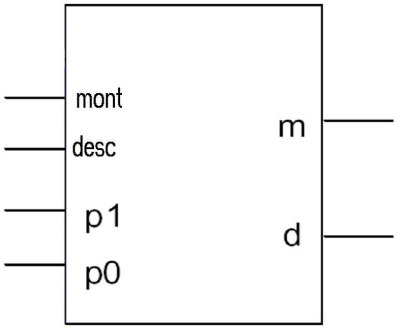
MACHINE DE MEALY

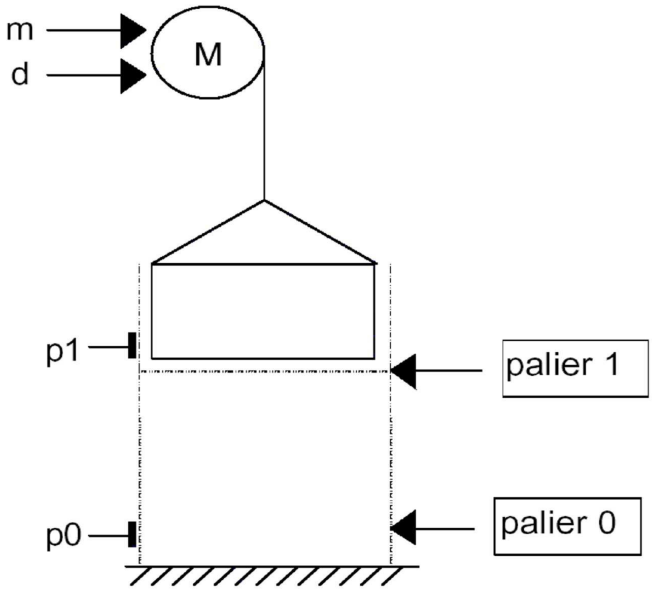
- Une machine d'état va résoudre la solution d'un problème **par une succession d'étapes** bien définies, bien séparées les unes des autres.
- Chaque étape est représentée par un ensemble **fini** de valeurs de ce que l'on appelle l'état (interne) du système. **Cet état est physiquement matérialisé par les valeurs contenues dans une mémoire (registre d'état)**. Cette mémoire représente l'état de l'automate, pas celui du système commandé (calcul des sorties).
- Les entrées du système provoquent des **transitions** entre les états (calcul de l'état futur).
- Si aucune des transitions issues d'un état n'est active, toutes les conditions étant fausses, le système reste dans l'état considéré, **par défaut il y a mémorisation de l'état présent**.

On peut modéliser les différents états d'une machine d'état grâce à un diagramme de transition.

3) Le diagramme d'état ou de transition

Prenons l'exemple d'un monte charge :





mont bouton de montée
desc bouton de descente
m, d : commandes du moteur
init : initialisation

Le moteur est commandé par deux signaux m et d pour montée et descente. La commande élabore ces commandes à partir de ses entrées mont, p1, p0 et desc suivant le cahier des charges suivant :

- desc provoque la descente du monte-charge jusqu'au palier p0. Ce signal est activé à la mise sous tension. On suppose qu'un plancher empêche le monte-charge d'être à un niveau inférieur à p0 .
- mont provoque la montée du monte-charge jusqu'au palier p1
- Quand la cabine est entre deux étages, la dernière commande (m ou d) est maintenue.

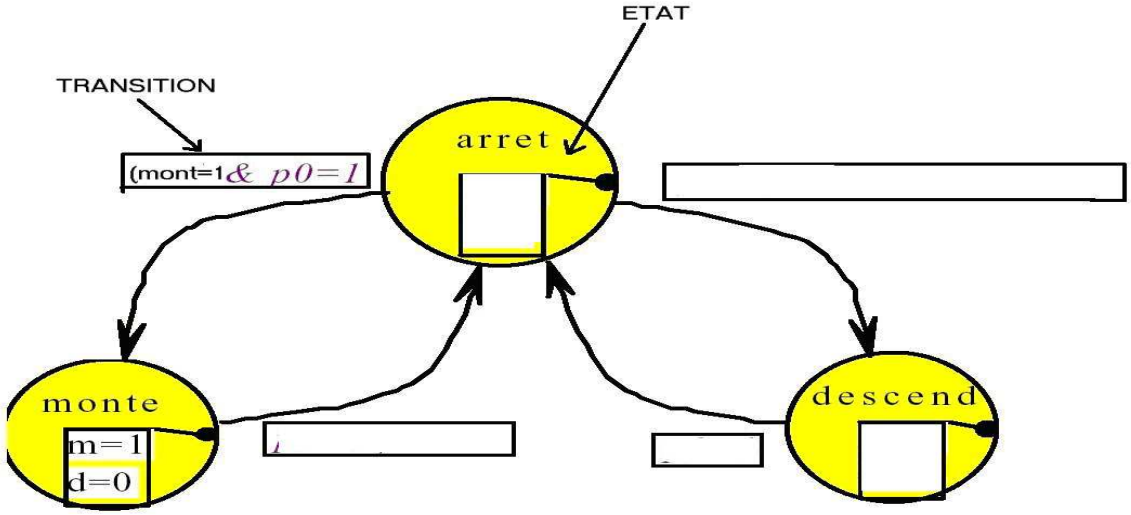


DIAGRAMME D'ETAT

Complétez le diagramme d'état afin qu'il réponde à l'application envisagée

4) Modélisation comportementale d'une machine d'état

Le processus qui décrit le fonctionnement d'une machine d'états comporte deux structures imbriquées : le traitement des transitions pour passer d'un état à un autre et le traitement de l'état de départ de chaque transition.

- **Les transitions**, compte tenu de leurs hiérarchies, se prêtent bien à une modélisation par des structures **if...elsif...else...end if**.
- **Les états** se prêtent bien à une modélisation en terme d'aiguillage, soit les structures **case...when...when others...end case**.
- Suivant le type de fonctionnement, primauté à l'état de départ ou primauté à la commande, on choisira l'ordre d'imbrication des deux structures correspondantes.

Pour décrire un diagramme de transitions en VHDL, une méthode simple consiste à traiter toutes les valeurs possibles de l'état de la machine, et pour chaque cas, analyser les entrées pour en déduire l'état suivant.

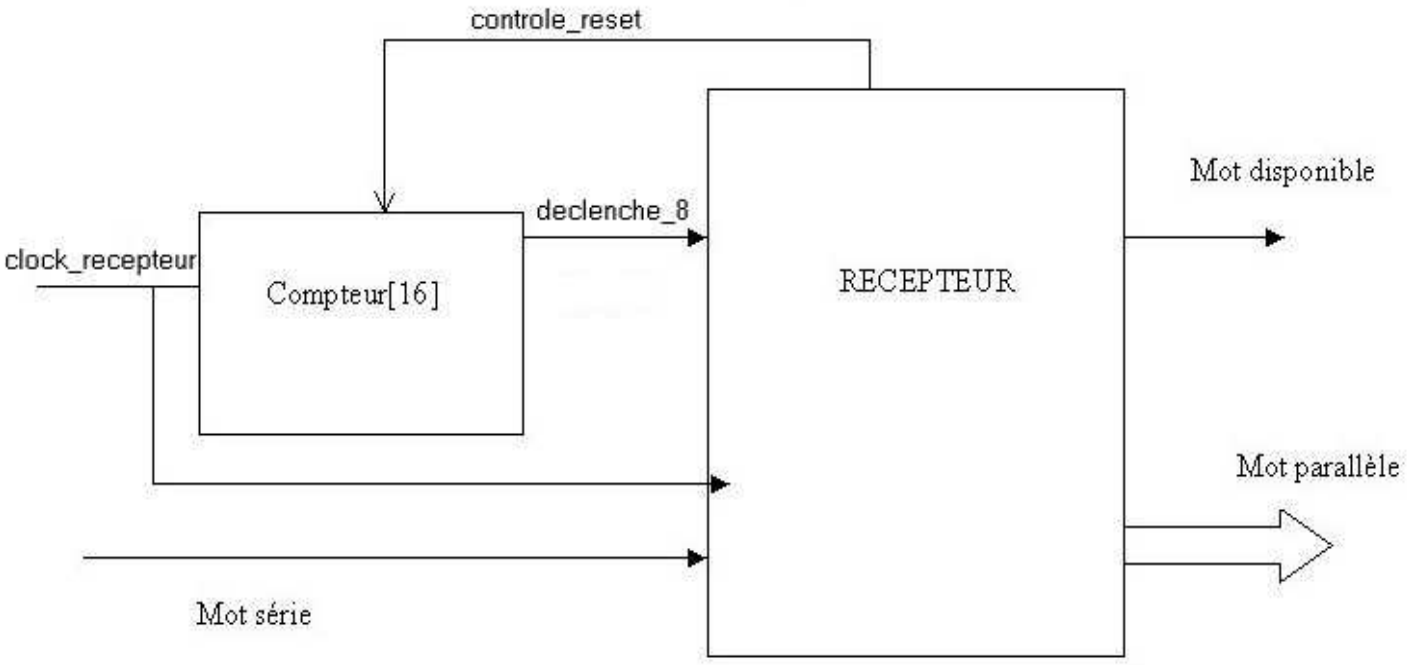
Complétez la modélisation VHDL de la machine d'état ci-dessous :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity montech is
port ( mont , desc, CLOCK, p0, p1: in std_logic ;
m , d: out std_logic) ;
end montech;
architecture comporte of montech is
signal etat: std_logic_vector(                    );
begin
d <=                ;
m <=          ;
process (clock)
begin
if (CLOCK ='1' and CLOCK'event) then
case etat is -- primauté a l'etat.
when "00" =>  if(                                 ) then etat <= "01" ;
     elsif(                                       )  then etat <= "10" ;
     else etat <= "00" ;
     end if ;
when "01" => if(p0='1') then etat <= "00" ;
     else etat <= "01";
     end if ;
when "10" => if(                    ) then etat <= "00" ;
     else etat <= "          ";
     end if ;
when others =>      etat <= "                     ";
end case ;
   end if ;
end process ;
end comporte ;
```

Modélisation comportementale du récepteur RS232
Simulateur sonar DUBV24

1) Principe

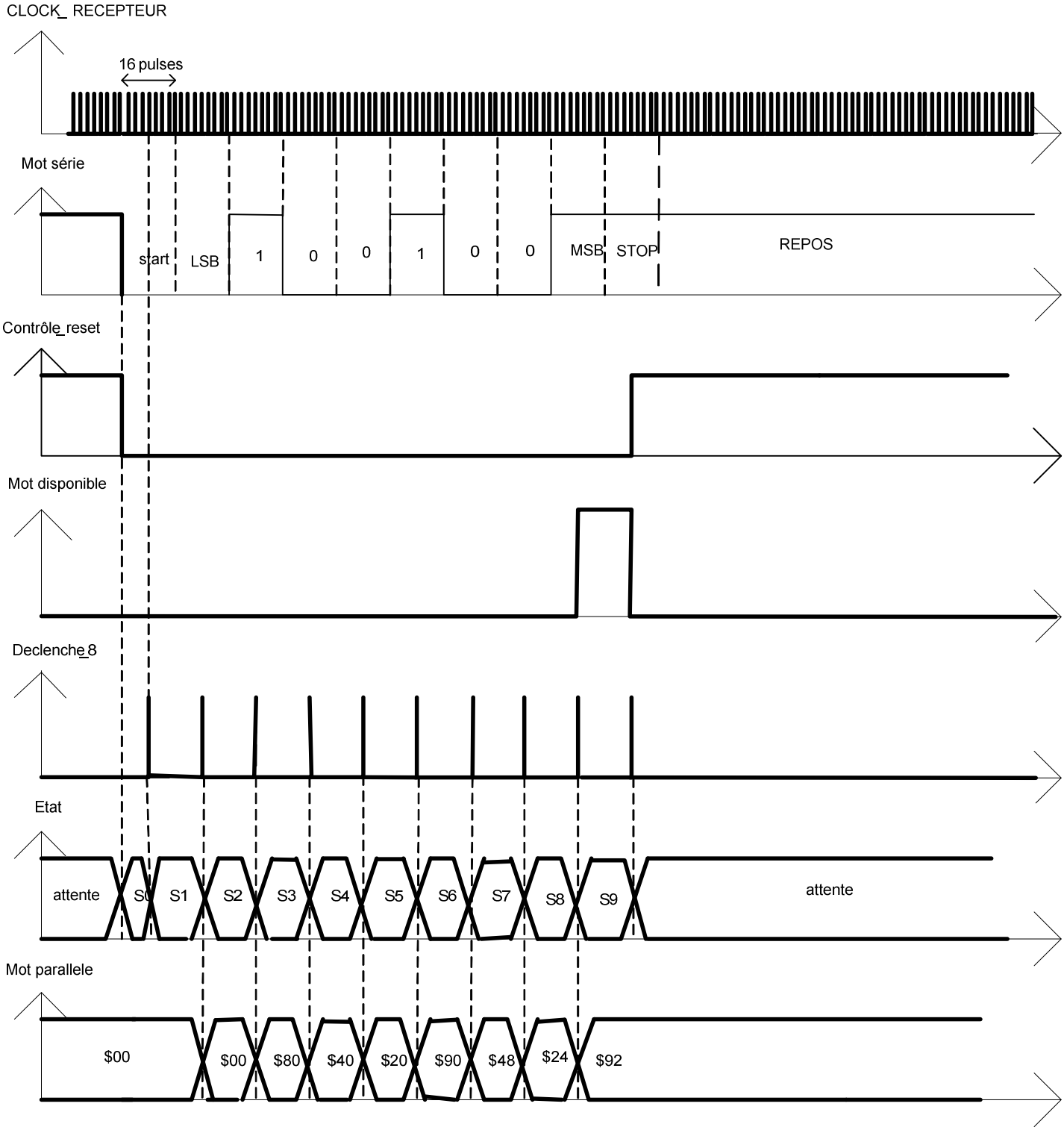
Pour la fonction de réception, nous avons échantillonné chaque bit au milieu de sa transmission. Pour cela, il fallait détecter un front descendant (bit de start sur mot série) sur la ligne puis synchroniser le système sur ce front. Une fonction de comptage (déclenchée par la détection du front descendant) positionne le moment d 'échantillonnage sur le bit.

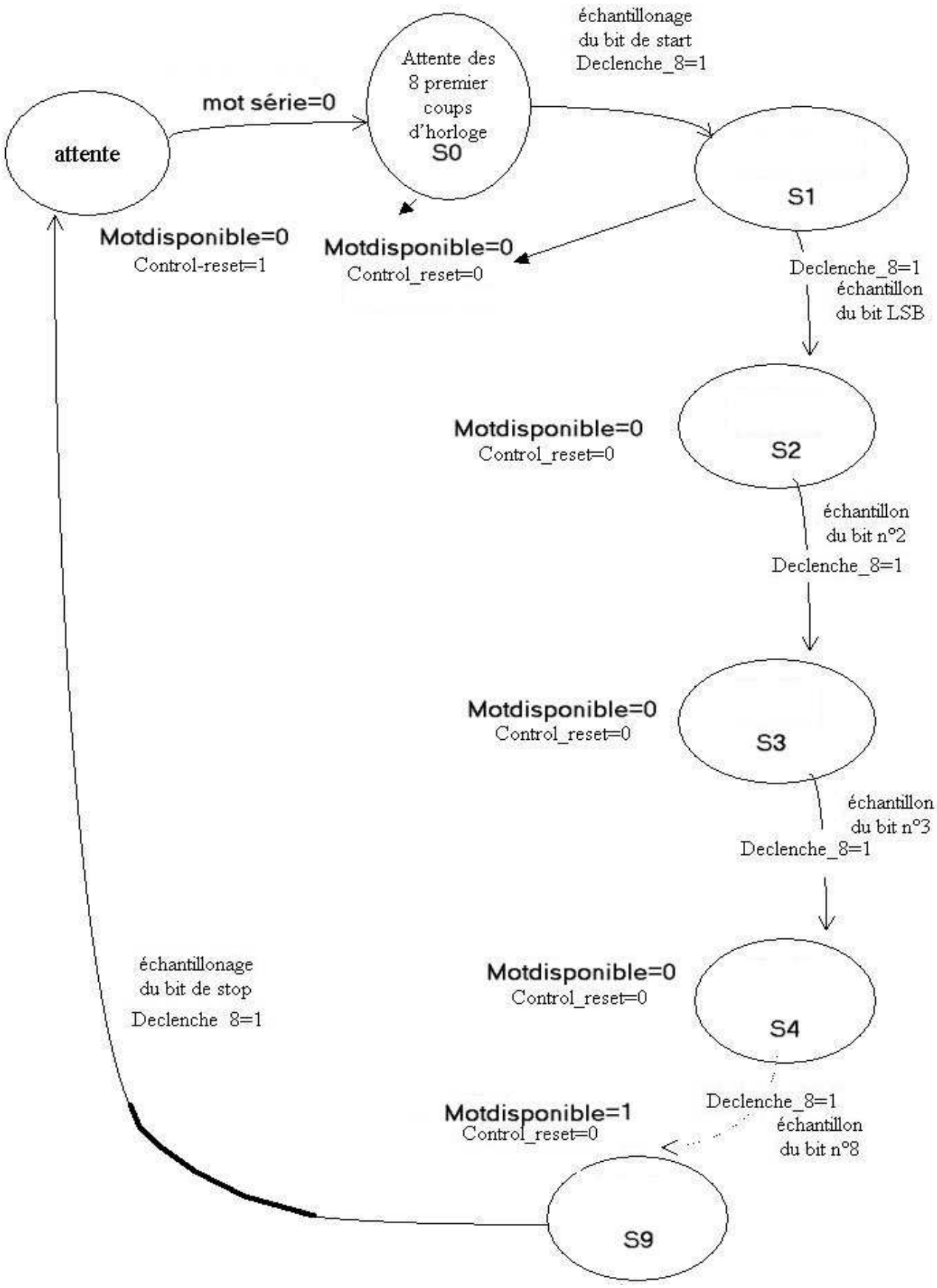


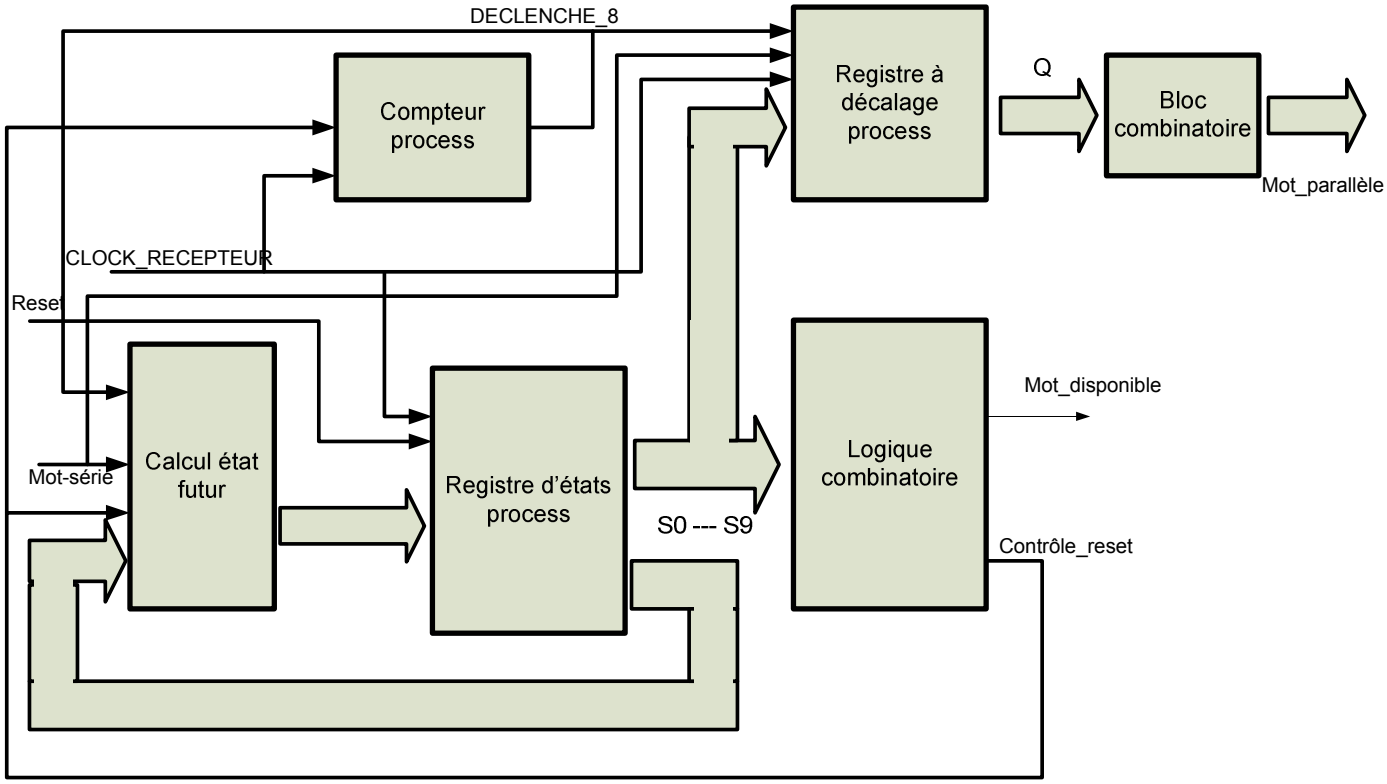
L'horloge d'entrée du récepteur est cadencée 16 fois plus vite que la vitesse de transmission de la ligne série. Un compteur modulo 16 nous permet, à chaque état à 8 de déclencher l'acquisition. En effet, le compteur va attendre 8 coups d'horloges pour échantillonner au milieu du bit de start. Puis 16 coups d'horloges plus tard, c'est à dire lorsque le compteur revient à l'état 8, nous re échantillonnons au milieu du bit de poids faible et ainsi de suite jusqu'au bit de stop.
Les chronogrammes page 27 nous donnent le principe de fonctionnement.

2) Machine d'état

Le récepteur est une machine d'état du type moore qui comprend 10 états (S0----S9, attente). Le fonctionnement est donné sous forme de chronogramme page suivante.
L'arrivée du bit de start (front descendant) permet de passer dans l'état S0 et le passage à 1 de declenche_8 permet de passer aux états suivants.
On donne page N° 28 le diagramme d'état de cette machine de moore :


CLOCK_ RECEPTEUR
16 pulses
Mot série
start
LSB
1
0
0
1
0
0
MSB
STOP
REPOS
Contrôle_reset
Mot disponible
Declenche_8
Etat
attente
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
attente
Mot parallele
$00
$00
$80
$40
$20
$90
$48
$24
$92


attente
mot série=0
Attente des 8 premier coups d'horloge S0
échantillonage du bit de start Declenche_8=1
S1
Motdisponible=0
Control-reset=1
Motdisponible=0
Control_reset=0
Declenche_8=1
échantillon du bit LSB
S2
Motdisponible=0
Control_reset=0
échantillon du bit n°2
Declenche_8=1
S3
Motdisponible=0
Control_reset=0
échantillon du bit n°3
Declenche_8=1
S4
Motdisponible=0
Control_reset=0
Declenche_8=1
échantillon du bit n°8
Motdisponible=1
Control_reset=0
S9
échantillonage du bit de stop Declenche 8=1



3) Question

3.1) Proposez un process en langage VHDL qui permet de décrire le fonctionnement de la machine d'état (On affectera pas pour l'instant les signaux Q, Motdisponible et contrôle_reset). Chaque état sera évaluée lors d'un front montant de clock_recepteur. On prendra en compte un signal reset asynchrone actif au 1 logique qui permettra de se positionner dans l'état d'attente au départ.
Pour vous aider on donne ci-dessous le début du programme VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;

entity recepteur is
port(
mot_serie: in std_logic;
declenche_8: in std_logic;
clock_recepteur: in std_logic;
reset: in std_logic;
control_reset: out std_logic;
mot_disponible: out std_logic;
mot_parallele: out std_logic_vector(7 downto 0)
);
```

```
end recepteur;

architecture archi_recepteur of recepteur is
TYPE liste_etat IS (attendre,s0,s1,s2,s3,s4,s5,s6,s7,s8,s9);
SIGNAL state : liste_etat;
signal Q : std_logic_vector(7 downto 0):="00000000";
begin
```

3.2) Proposez un process qui permet d'affecter la sortie Q en fonction du mot série et des différents états. Le changement d 'état Q sera synchroniser avec chaque front montant du signal clock_recepteur. Q est simplement une recopie du mot parallèle.

3.3) donnez les instructions concurrentes qui permettront d'affecter les signaux contrôle_reset, motdisponible, mot_parallele

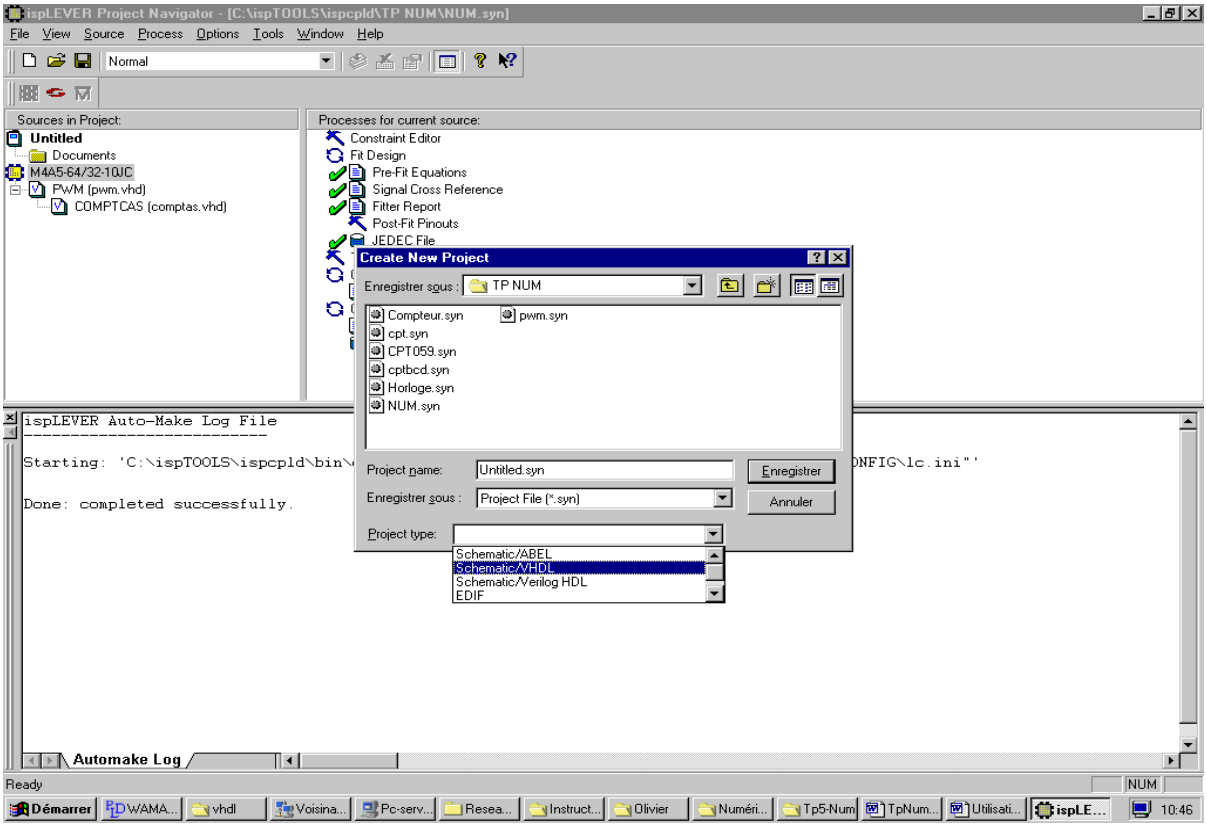
3.4) Donnez un process qui permet d'obtenir le signal declenche_8

```
library IEEE;
LIBRARY synth;
USE synth.vhdlsynth.all;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;

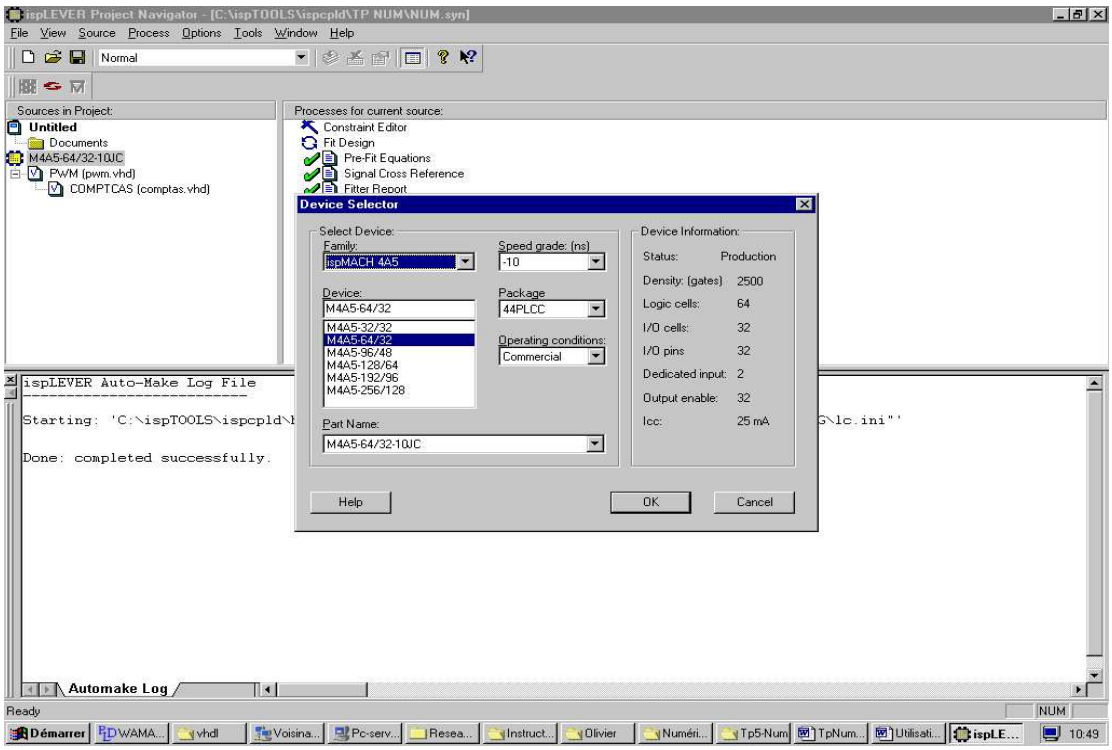
entity recepteur is
port(
mot_serie: in std_logic;
declenche_8: inout std_logic;
clock_recepteur: in std_logic;
reset: in std_logic;
control_reset: out std_logic;
mot_disponible: out std_logic;
mot_parallele: out std_logic_vector(7 downto 0)
);
end recepteur;
```

UTILISATION DE ISPLEVER

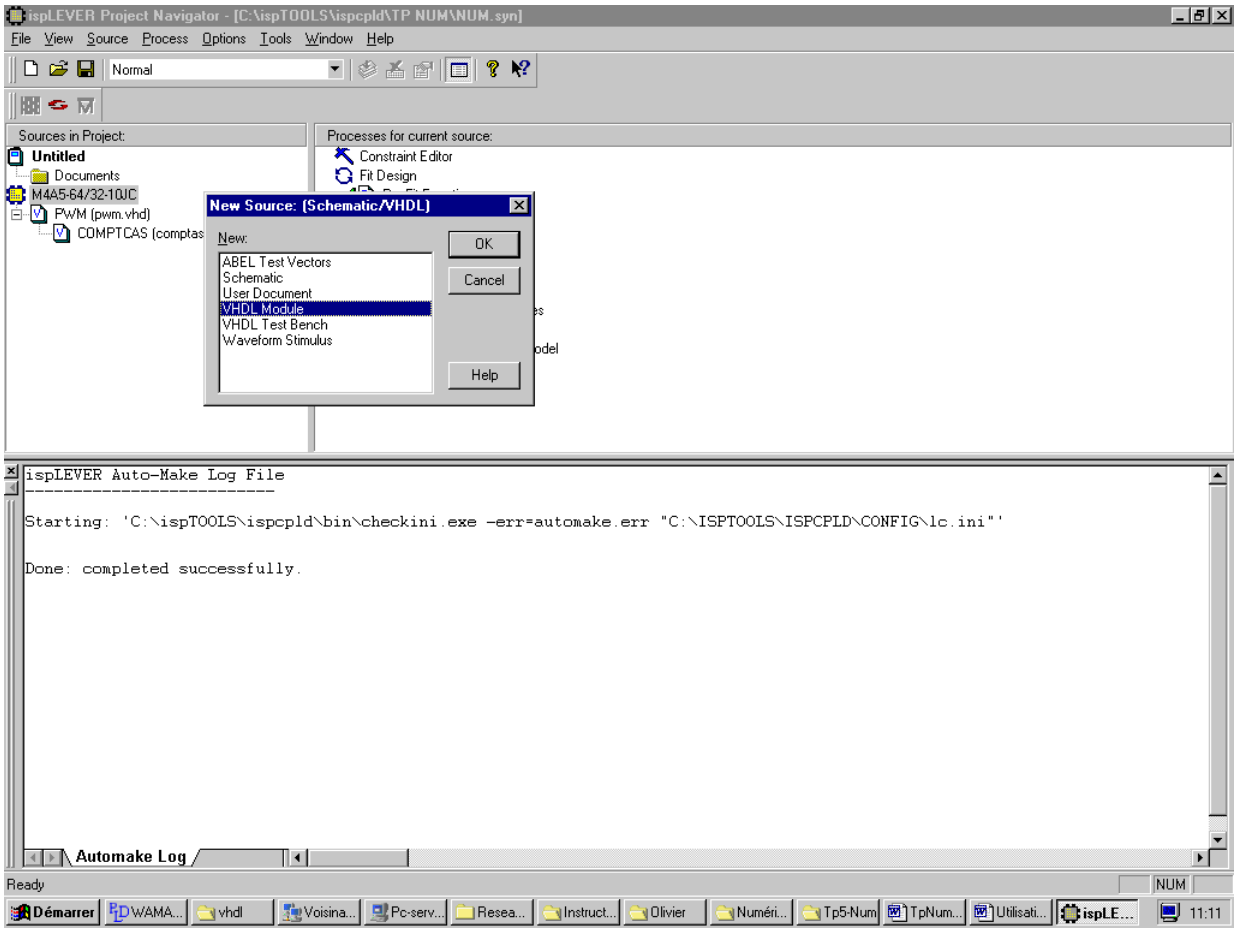
1. Ouvrir ISPLEVER puis créer un **NEW PROJECT.** Nommez votre projet puis spécifiez le type : **schematic/VHDL.**



2. Sélectionnez le composant utilisé en double-cliquant sur le composant dans la fenêtre « **Sources in Project »** puis sélectionnez un M4A5 64/32 –10JC.



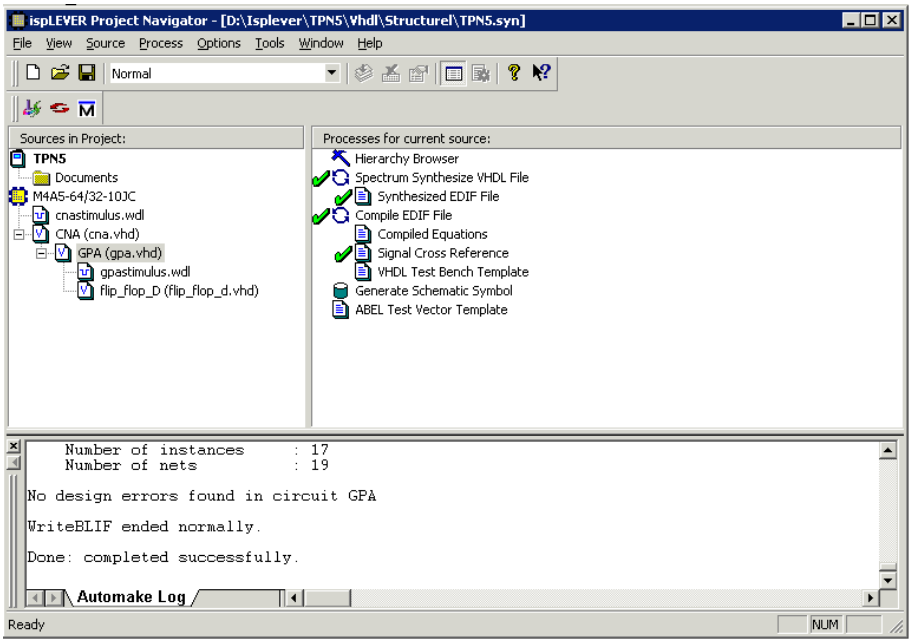
3. Cliquez droit sur le composant pour sélectionner **NEW SOURCE** puis **VHDL MODULE.**



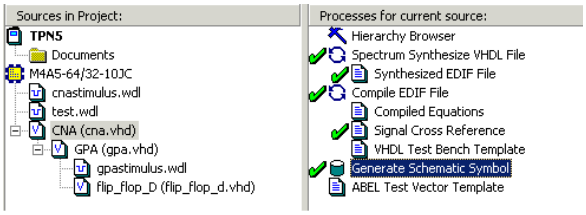
Remplir les champs « File Name », « Entity » et « Architecture » puis valider.
Vous pouvez alors saisir votre code source en VHDL.

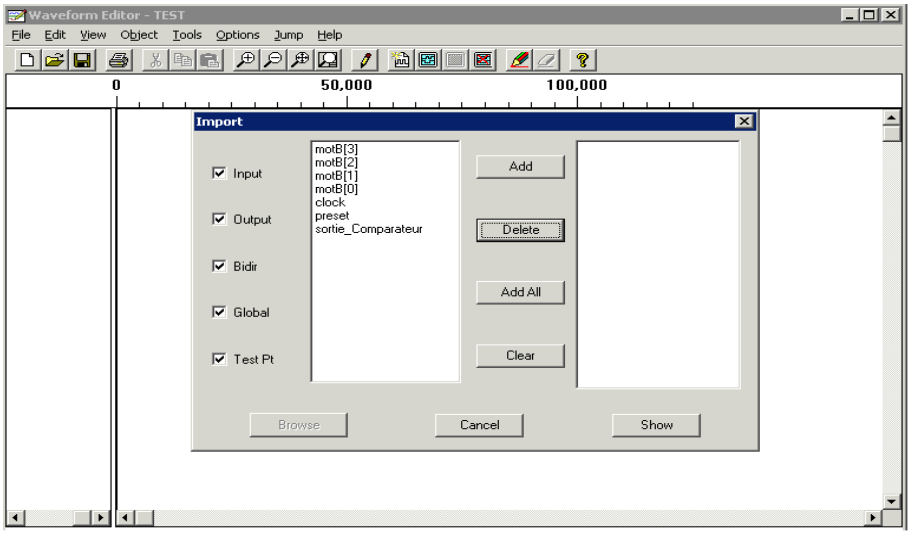
4. Votre programme étant écrit, sauvegardez le puis fermez l'éditeur de texte.

5. Sélectionnez votre fichier puis double-cliquez sur **compile EDIF file** pour que le compilateur vérifie la cohérence de votre programme ainsi que la syntaxe VHDL .

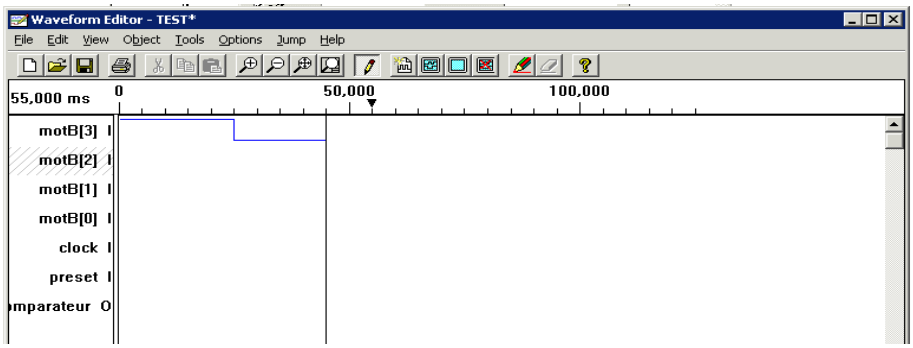


6. Afin de réaliser une simulation il vous est nécessaire d'éditer des « waveform stimulus » .Pour cela sélectionner soit le fichier VHDL concerné soit directement le composant puis en cliquant droit choisir new « waveform stimulus » puis choisir le fichier ou le composant auquel il doit être associé .Après avoir nommer votre nouveau fichier l'éditeur de waveform est lancé .Lancer alors la fonction « generate schematic symbol » associée à votre fichier VHDL vous pouvez alors importer les waveform associées à ce même fichier en cliquant sur « edit » puis « import wave dans l'éditeur de waveform ».

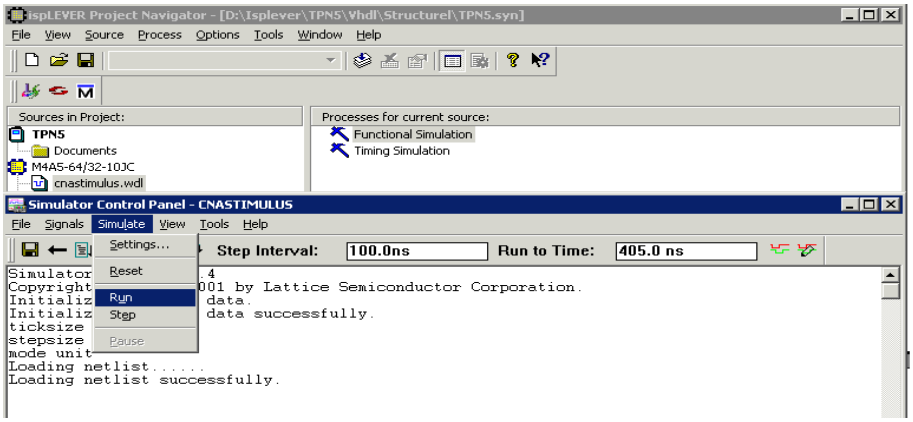




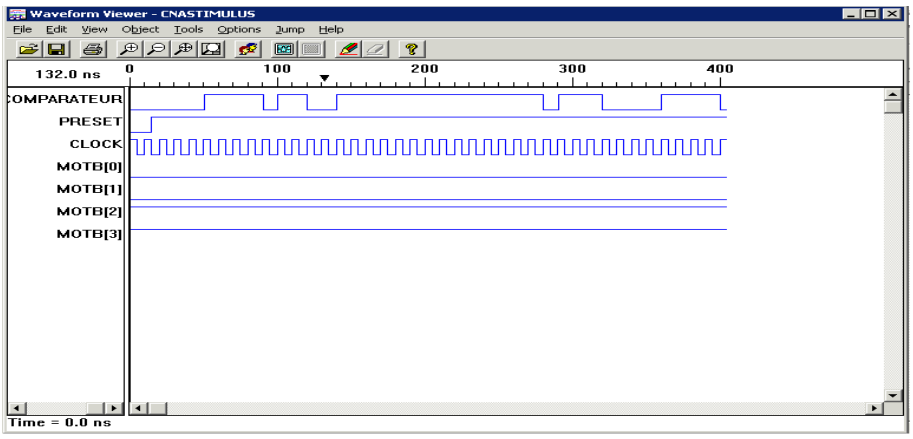
Il faut maintenant définir les chronogrammes associés aux entrées en sélectionnant l'entrée souhaitée puis en cliquant gauche chaque fois que l'on désire changer d'état Une fois tous vos signaux d'entrée tracés sauvegarder.



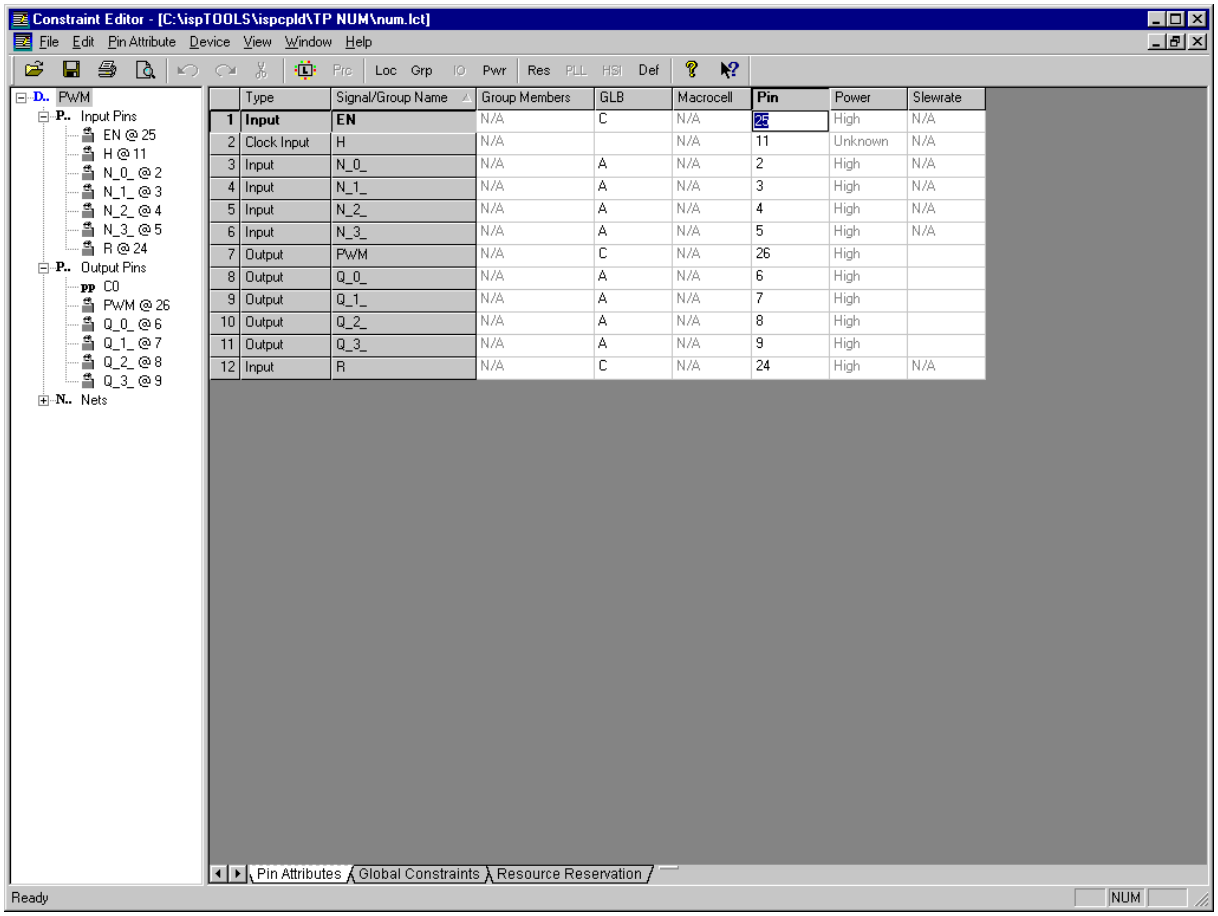
7. Sélectionnez le fichier *.wdl puis lancer la fonction « functionnel simulation ».Le « simulator control panel » s'ouvre ; en cas de succès vous pouvez alors effectuez la simulation en cliquant sur « simulate » puis « run ».



8. La fenêtre du « waveform viever » s'ouvre alors .Vous pouvez visualiser la ou les sorties préalablement retenues sous le « waveform editor ».



9. Ouvrez la fenêtre **constraint editor** puis affectez vos entrées et sorties aux broches que vous désirez.



10. Sélectionnez votre composant puis double-cliquez sur « JEDEC FILE » afin de « Fitter » le programme, c'est à dire de vérifier la cohérence du programme avec la structure interne du composants .Cela générera un fichier ***.jed.**

Le fichier ***.jed** sera ouvert à l'aide du programmateur