

# Avertissement

Au département GEII de l'IUT de Troyes le module a2i13 est composé de trois parties :

- apprentissage d'un logiciel de CAO (Eagle),
- approfondissement de l'introduction à VHDL du module a2i11.
- apprentissage de SPICE

Nous présentons ici le polycopié A2I13 concernant VHDL. Les 4 premiers TDs représentent 1h30 pendant 7 semaines et les deux derniers TDs ne sont pas réalisés faute de temps (pour être plus exact on ne dépasse jamais la page 21). A ceci s'ajoutent 1h30 de TP pendant 7 semaines. Les énoncés des TPs sont disponibles dans un fichier séparé.

La première version de ce polycopié était indépendante des compilateurs, mais il nous est apparu essentiel d'ajouter des exercices plus dépendants, utilisant en particulier les fichiers de rapport (pour obliger les étudiants à lire ces fichiers). Comme en TP nous utilisons Warp II de Cypress, il y aura un certain nombre de références à ce logiciel. Nous espérons malgré tout ne pas être allé trop loin dans ces références et qu'ainsi, ce document soit facilement adaptable à d'autres environnements.

Nous avons choisi ce logiciel car il était pour nous essentiel de programmer des petits composants comme les 20V8 et 22V10.

Certains énoncés d'exercices sont en anglais avec peut être quelques fautes ...

Ce document a été réalisé avec StarOffice 5.2 à l'origine puis modifié et transformé en pdf avec OpenOffice 1.1 le tout essentiellement sous LINUX.

Troyes Mars 2004  
Serge Moutou

## TD1 A2I13 : VHDL, tables de vérité, diagramme d'évolution

### I) Retour sur les styles de programmation VHDL

Lorsque nous avons présenté la programmation d'une table de vérité en VHDL (A2I11-TD7) nous avons utilisé la technique de concaténation « & » pour assembler deux signaux en un seul. Ici nous présentons la technique des BIT\_VECTOR. Imaginons que l'on ait la table de vérité (4 entrées 2 sorties) et l'entité correspondante :

a3	a2	a1	a0	s1	s0	
0	1	0	1	1	1	ENTITY demo IS PORT( a : in BIT_VECTOR(3 DOWNTO 0);-- 4 entrées s : out BIT_VECTOR(1 DOWNTO 0)); -- 2 sorties END demo;
0	1	1	0	0	1	
1	1	0	1	1	0	

(ce qui n'est pas mentionné correspond à 00 en sortie )

Le programme VHDL en style "with select when" s'écrit :

```

ARCHITECTURE mydemo OF demo IS
BEGIN
  WITH a SELECT
    s <= "11" WHEN "0101", -- premiere ligne
         "01" WHEN "0110", -- deuxieme ligne
         "10" WHEN "1101", -- troisieme ligne
         "00" WHEN OTHERS;
END mydemo;
```

On écrirait le même programme en style "when else" :

```

ARCHITECTURE mydemo OF demo IS
BEGIN
  -- style when else
  s <= "11" WHEN a="0101" ELSE -- premiere ligne
       "01" WHEN a="0110" ELSE -- deuxieme ligne
       "10" WHEN a="1101" ELSE -- troisieme ligne
       "00";
END mydemo;
```

Remarque : la structure "when else" ne nécessite pas des conditions mutuellement exclusives. Elle engendre alors une architecture avec priorité. Par exemple dans

```

j<= w when a='1' else
  x when b='1' else
  0;
```

les conditions ne sont pas mutuellement exclusives. On ne pourrait pas utiliser une structure "with select when" qui nécessite des conditions absolument exclusives.

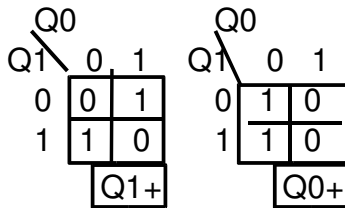
**Exercice 1**

Write a VHDL program for a one bit adder using "with select when" style.

**II) Le séquentiel**

**Le séquentiel simple (diagramme d'évolution) avec équations de récurrence**

État présent		État futur	
Q1	Q0	Q1+	Q0+
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0



Équations de récurrence :

$$Q_1^+ = Q_1 \text{ XOR } Q_0$$

$$Q_0^+ = \text{NOT } Q_0$$

```

ENTITY cmpt IS PORT (
  clk: IN BIT;
  q0,q1: INOUT BIT);
END cmpt;
ARCHITECTURE acmpt OF cmpt IS
BEGIN
  cmpt1 : PROCESS (clk) BEGIN
    IF (clk'EVENT AND clk='1') THEN
      q0 <= NOT q0;
      q1 <= q0 XOR q1;
    END IF;
  END PROCESS;
END acmpt;

```

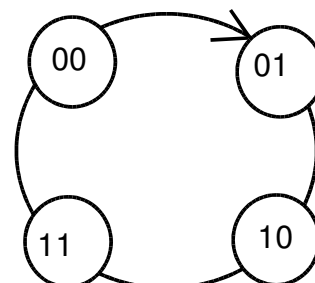
**Le séquentiel simple (diagramme d'évolution) sans équations de récurrence**

```

ENTITY demo IS PORT(
  clock : IN BIT;
  q : INOUT BIT_VECTOR(0 TO 1));
END demo;

ARCHITECTURE mydemo OF demo IS
  BEGIN
    PROCESS(clock) BEGIN
      IF clock'EVENT AND clock='1' THEN
        CASE q IS --style case when
          WHEN "00" => q <= "01";
          WHEN "01" => q <= "10";
          WHEN "10" => q <= "11";

```



```
        WHEN OTHERS => q <="00" ;  
    END CASE ;  
END IF ;  
END PROCESS ;  
END mydemo ;
```

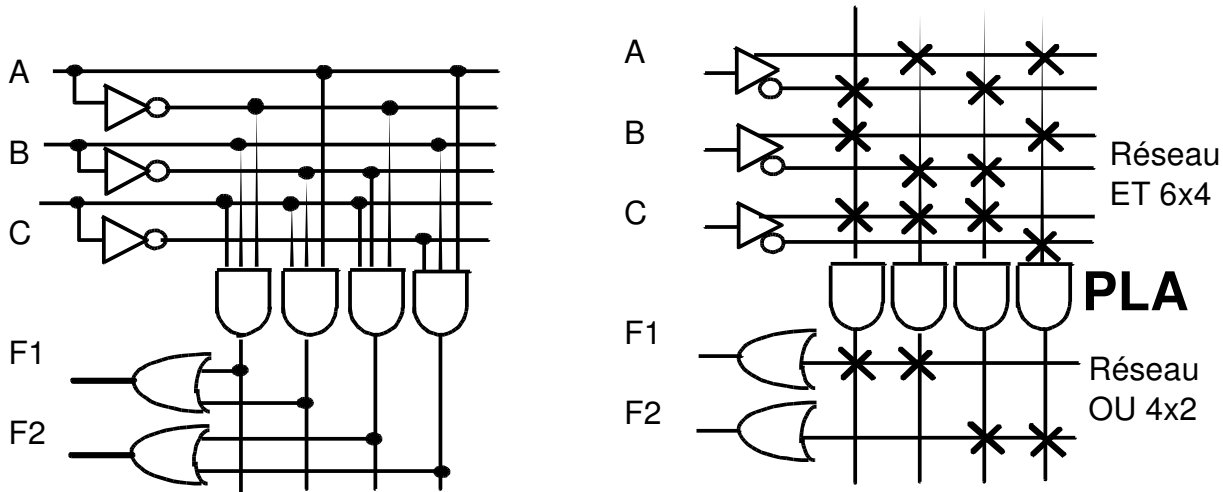
## **Exercice 2**

Réaliser un compteur GRAY sur 3 bits en utilisant ces deux méthodes

## TD2 A2I13 : VHDL et logique programmable

### I) Généralités

Conventions de la représentation des circuits programmables :

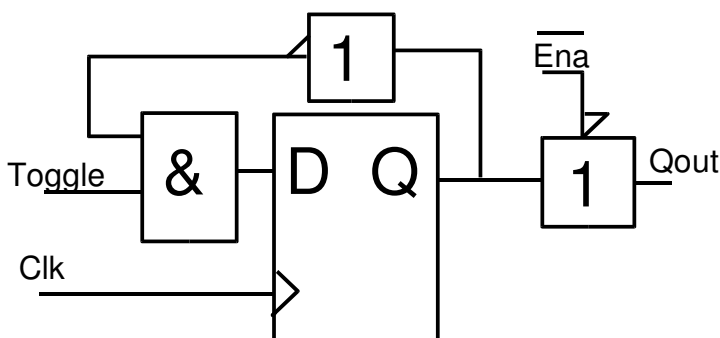


Comment relier les broches de CI aux entrées/sorties logiques ? Avec Warp2 on utilise un attribut :

```
ATTRIBUTE pin_numbers of mydesign:ENTITY IS
"x:1 y:2 clk:3 a(0):4 ";
```

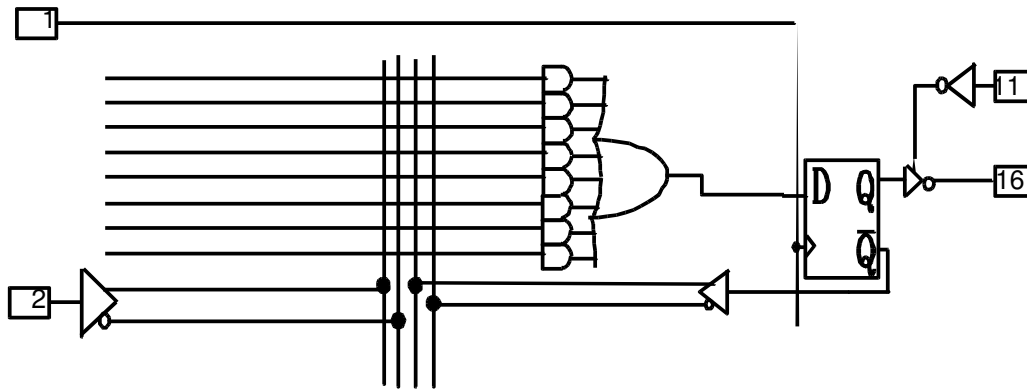
avant de fermer la déclaration d'entité (qui s'appelle dans ce cas "mydesign").

### Exercice 1

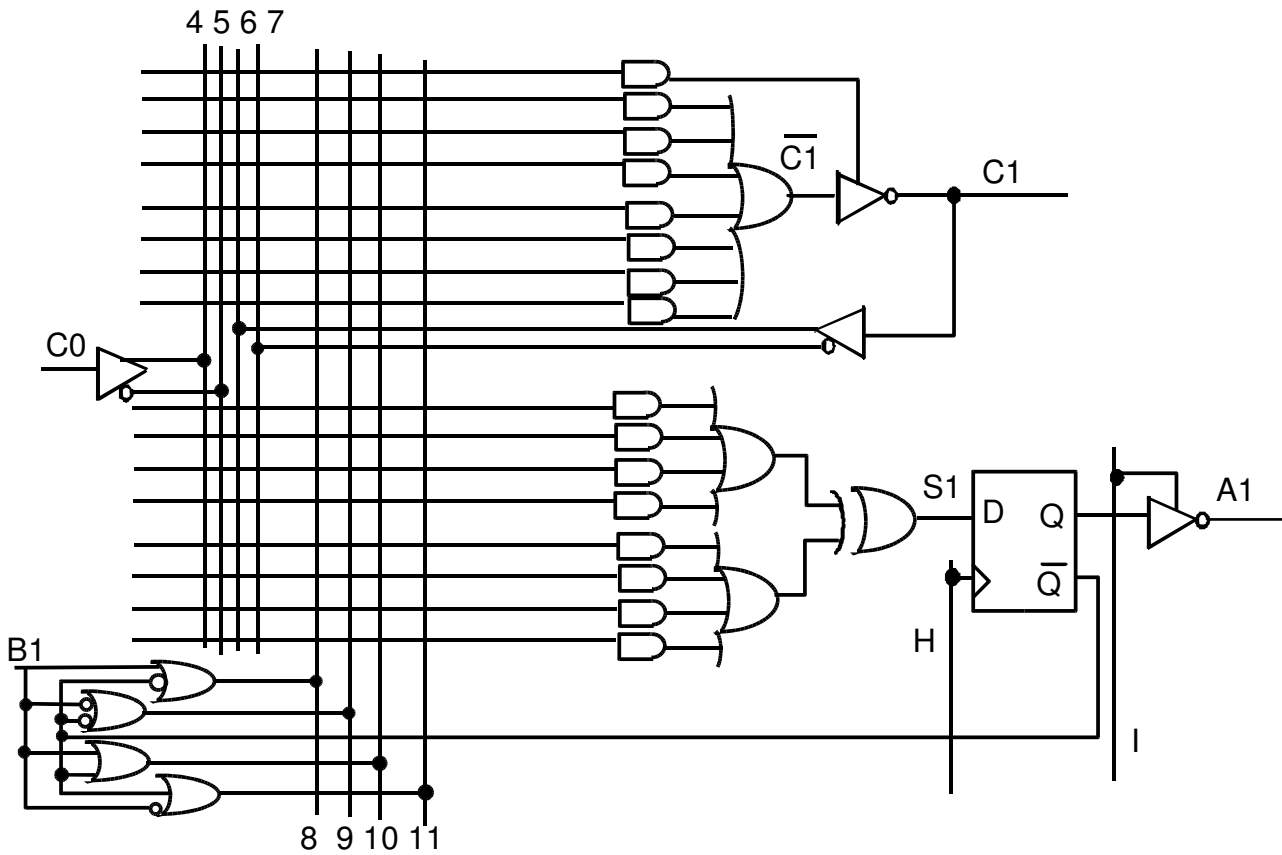


On veut réaliser le circuit schématisé ci-contre dans une PAL 16R8 dont le schéma partiel est présenté ci-dessous. Écrire le programme VHDL correspondant en remarquant que la variable Ena n'a pas besoin d'être déclarée dans l'entité. Placez les croix correspondantes au programme sur le schéma ci-après.

**Remarque** : même si cette structure (ci-dessous) avec OU à nombre d'entrées fixé est appelée PAL, on utilisera indifféremment dans la suite PAL ou PLA.



**Exercice 2** (PAL 16X4 spécialisé pour arithmétique)



1°) La cellule de base d'un PAL destiné à l'arithmétique est donnée ci-dessus. Repérer et exprimer les valeurs logiques des fils internes 8, 9, 10 et 11 en fonction de A1 et B1.

A quoi sert I ?

2°) On cherche à utiliser ce genre de cellule pour réaliser une addition 1 bit :  $(C1 A1^+)_2 = (A1)_2 + (B1)_2 + (C0)_2$ . On rappelle les équations de C1 et A1<sup>+</sup>

$$C1 = (A1 \text{ xor } B1).C0 + A1.C1$$

$$A1^+ = A1 \text{ xor } B1 \text{ xor } C0$$

En déduire les équations de S1 et de /C1.

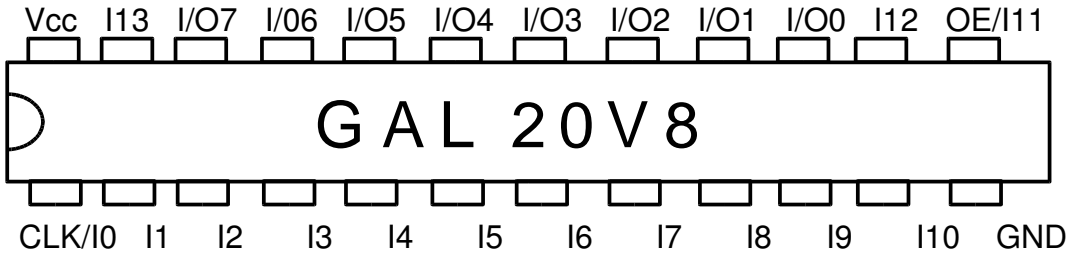
3°) Pour réaliser S1 il nous faut un deuxième ou exclusif, par exemple A1 xor B1. Exprimer ce ou exclusif en fonction de 8, 9, 10 et 11.

4°) Dessiner les fusibles non grillés sur la figure ci-dessus.

**Indication** : on rappelle que  $\bar{a} \cdot b = (\bar{a} + \bar{b}) \cdot (\bar{a} + b) \cdot (a + b)$   
 et encore  $a \text{ xor } b = (a + b) / (a + /b) = (9) \cdot (10)$  et même  $(a \text{ identité } b) = (8) \cdot (11)$

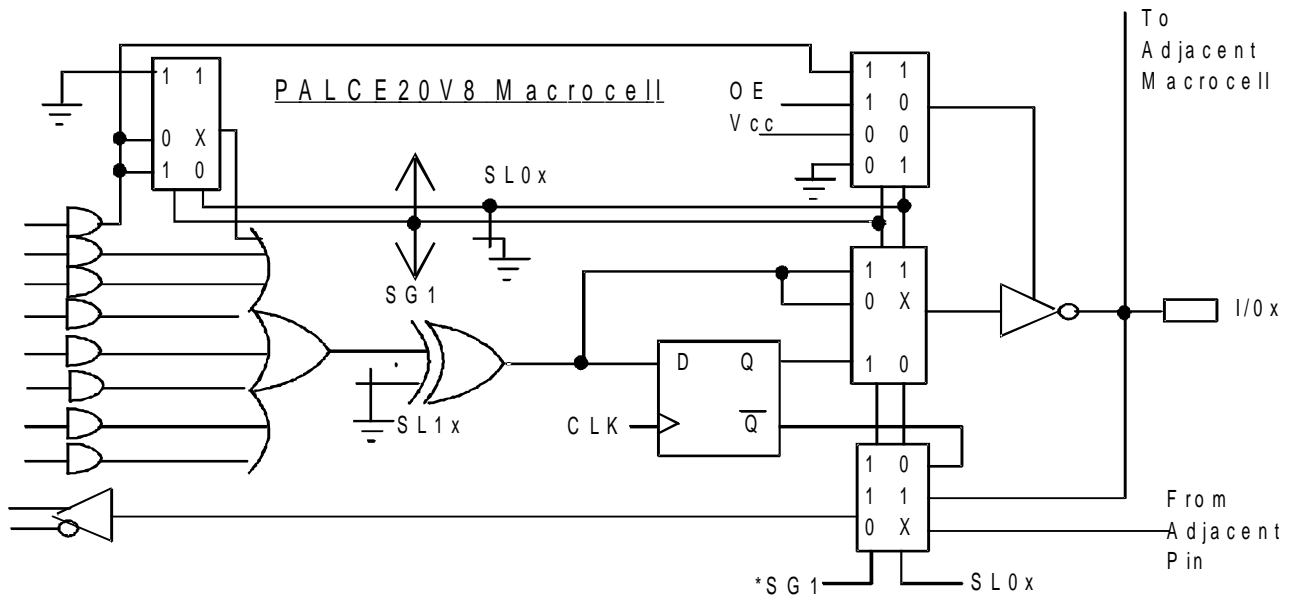
**I) La GAL 20V8**

On présente ci-dessous le schéma partiel de la GAL 20V8 (20 broches 8 sorties configurables).



**Exercice 3**

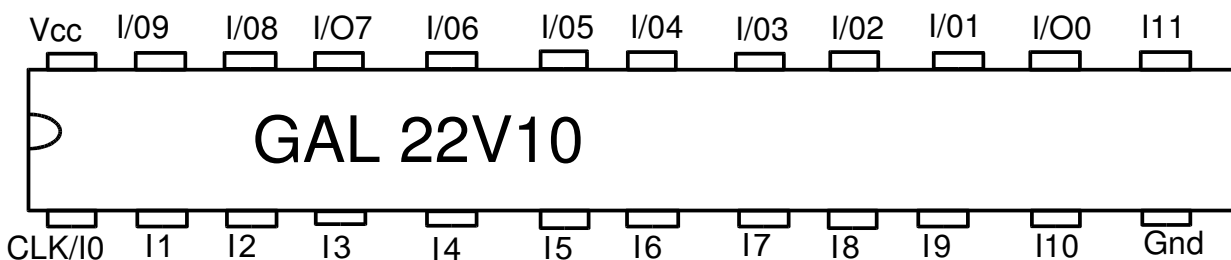
Donner tous les schémas équivalents de la macrocellule de sortie de la GAL 20V8 (présentée en ci-après).

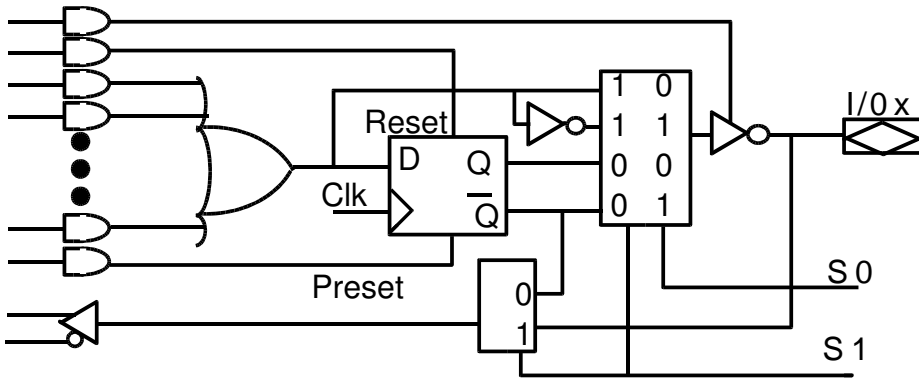


\*In macrocell MC0 and MC7, SG1 is replaced by /SG0 on the feedback multiplexer.

**III) La GAL 22V10**

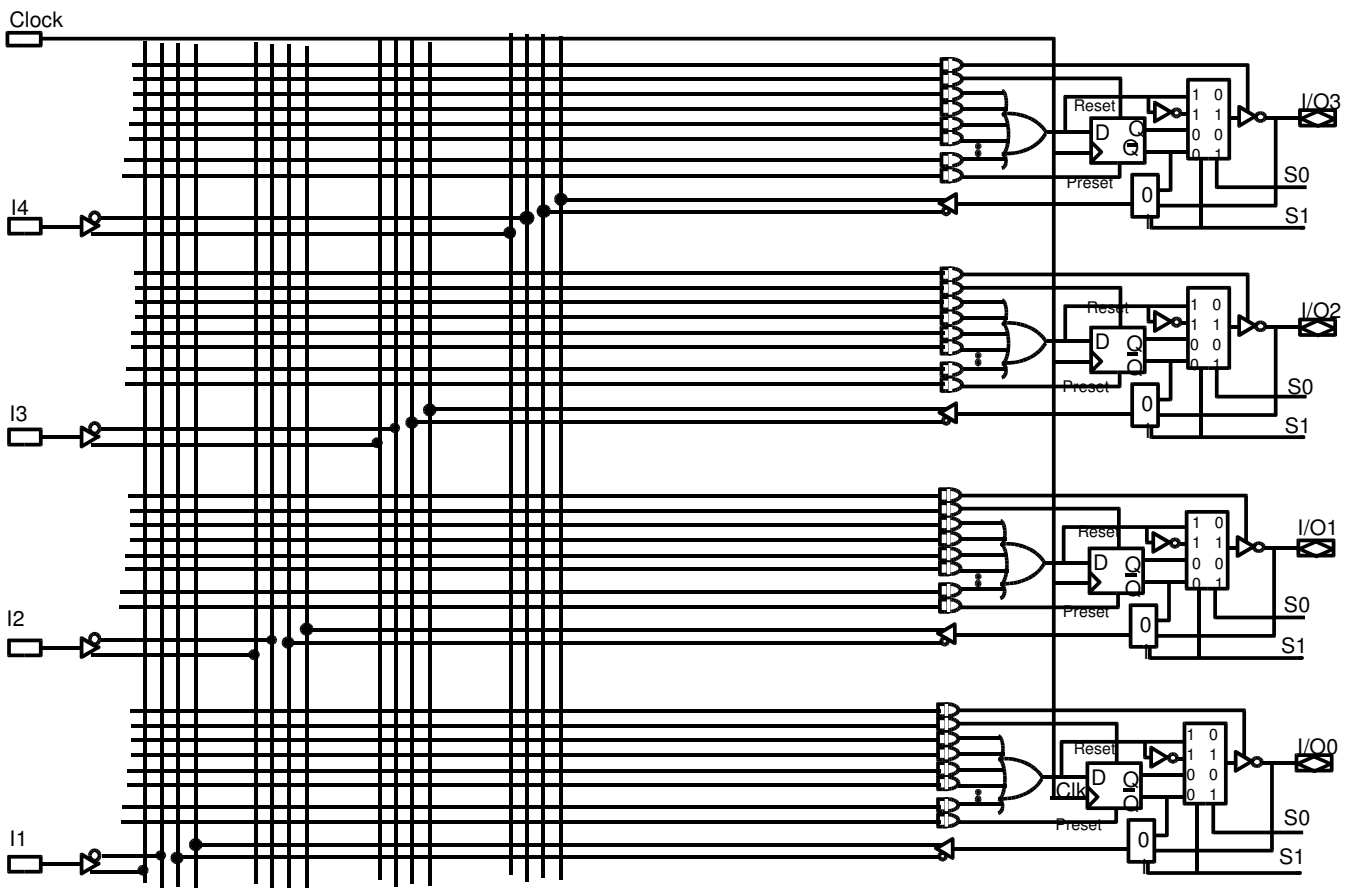
La GAL 22V10 est un peu plus complexe que la 20V8 : 10 sorties matrice ET 132x44.





**OLMC**

The 22V10 has variable number of product terms per OLMC : pins 14 and 23 have eight, pins 15 and 22 have ten, pins 21 and 16 have twelve, pins 17 and 20 have fourteen and pins 18 and 19 have sixteen product terms.



**Exercise 4**

The output polarity of each OLMC can be individually programmed to be true or inverting, in either combinational or registered mode.

- 1°) Which of the two bits (S0 and S1) determines the polarity ?
- 2°) Draw the four equivalent logic circuits for an OLMC.

n	BCD				Excess 3			
	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	E <sub>3</sub>	E <sub>2</sub>	E <sub>1</sub>	E <sub>0</sub>
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

**Exercice 5**

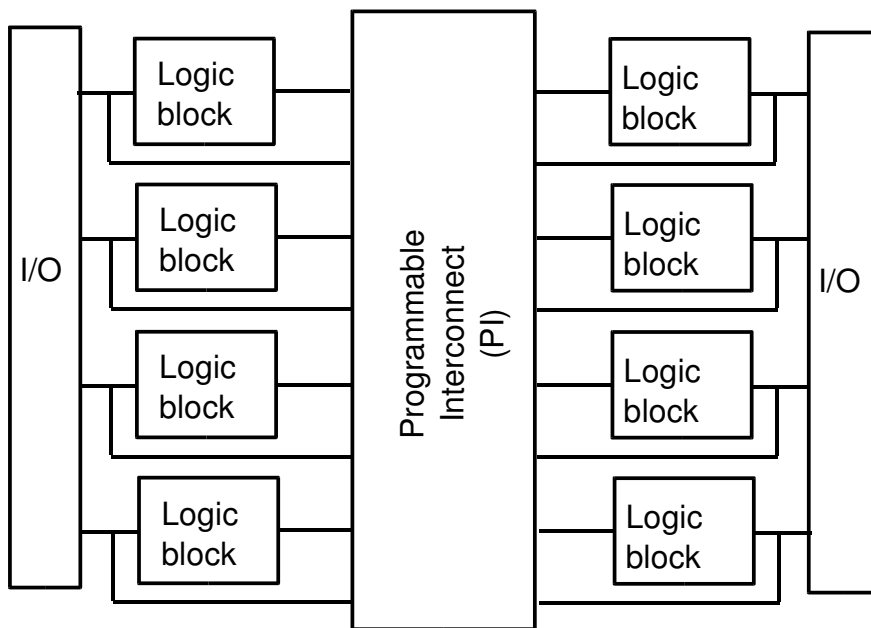
Réaliser un transcodeur BCD -> Excess 3.

Écrire le programme VHDL correspondant.

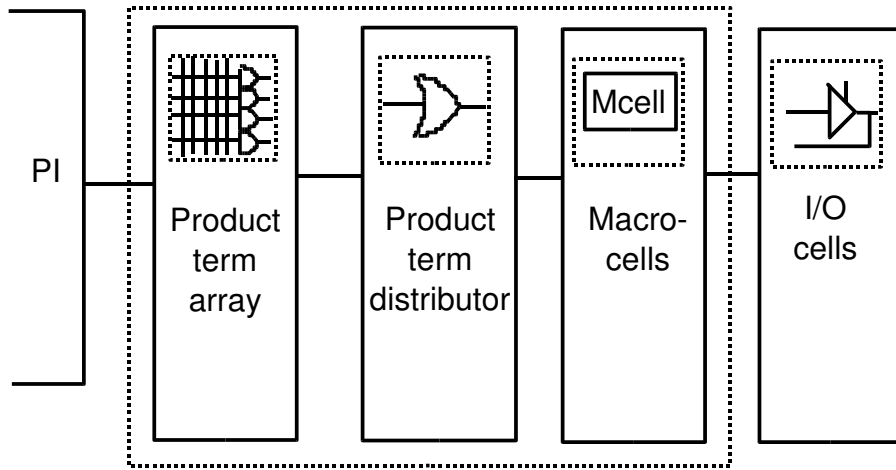
Compléter le schéma de la page précédente( petites croix).

**IV) CPLDs**

Un CPLD peut être défini comme un assemblage de PLDs de type 22V10. Nous en donnons la structure ci-dessous.



Architecture CPLD générique



Bloc logique générique (LAB=Logic Array Block)

## TD3 A2I13 : VHDL et logique séquentielle programmable

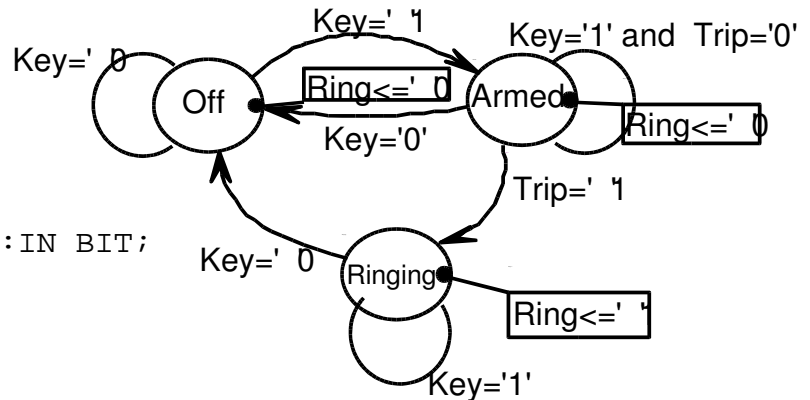
### I) VHDL et le séquentiel

Nous avons déjà eu l'occasion de présenter la programmation du séquentiel en VHDL a2i11-TD9 et a2i13-TD1. Nous allons présenter une technique dite « à un process ».

Exemple du réveil :

```
ENTITY Alarm IS
  PORT(
    clock,Key,Trip :IN BIT;
    Ring :OUT BIT
  );
END Alarm;
```

```
ARCHITECTURE ar OF Alarm IS
  TYPE typetat IS (Armed, Off, Ringing);
  SIGNAL etat : typetat;
BEGIN
  PROCESS (clock,etat) BEGIN -- partie séquentielle
    IF Clock ='1' AND Clock'EVENT THEN
      CASE etat IS
        WHEN Off => IF key ='1' THEN etat <= Armed;
                     ELSE etat <= Off;
                     END IF;
        WHEN Armed => IF Key = '0' THEN
                       etat <= Off;
                     ELSIF Trip ='1' THEN
                       etat <= Ringing;
                     ELSE etat <= Armed;
                     END IF;
        WHEN Ringing => IF Key ='0' THEN
                        etat <= Off;
                       ELSE etat <= Ringing;
                       END IF;
      END CASE;
    END IF;
    IF etat=Ringing THEN -- partie combinatoire
      Ring<='1';
    ELSE Ring <='0';
    ENDIF
  END PROCESS;
END ar;
```



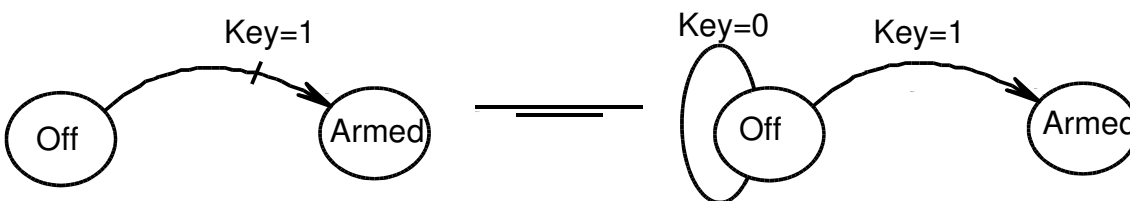
**Remarques :**

1°) Il existe un outil permettant de dessiner une machine d'état avec Warp : active-HDL FSM (Finite State Machine) dont nous avons pris la notation des actions.

2°) Warp est capable de générer le codage des états même si on ne lui déclare pas les INOUT nécessaires. Par exemple le programme ci-dessus sera correctement compilé et en cours de compilation il créera la broche nécessaire au codage des états sur deux bits (en utilisant la sortie Ring comme deuxième bit).

**II) Programmation de Graphes d'états**

Nous avons déjà eu l'occasion de parler de ce problème en a2i11 (voir Tds 15 et 16) avec des équations de récurrence. Présentons d'abord la technique avec et sans initialisation.



-- sans initialisation

```

BEGIN
  PROCESS (clock) BEGIN
    IF clock'EVENT AND clock='1' THEN
      CASE etat IS
        WHEN Off => IF key ='1' THEN etat <= Armed;
                     ELSE etat <= Off;
                     END IF;
        ....
      END CASE;
    END IF;
  END PROCESS;
  ....
  
```

-- avec initialisation synchrone

```

BEGIN
  PROCESS (clock) BEGIN
    IF clock'EVENT AND clock='1' THEN
      IF Init='1' THEN etat <=Off; --initialisation synchrone
      ELSE
        CASE etat IS
          WHEN Off => IF key ='1' THEN etat <= Armed;
                       ELSE etat <= Off;
                       END IF;
          ....
        END CASE;
      END IF
    END IF;
  END PROCESS;
  
```

....

### III) Le codage des états

La programmation des états nécessite une déclaration symbolique comme on l'a vu dans le cas du réveil :

```
TYPE typetat IS (Armed, Off, Ringing); -- dans architecture
SIGNAL etat : typetat;
```

Quand la synthèse sera demandée plusieurs solutions peuvent se présenter suivant le codage des états. Une telle déclaration débouchera sur un codage Armed=00, Off=01 et Ringing=10.

On peut modifier ce codage à l'aide de deux attributs différents : `enum_encoding` et `state_encoding`. `Enum_encoding` est normalisé par le standard IEEE 1076.6.

```
type state is (s0,s1,s2,s3);
attribute enum_encoding of state:type is "00 01 10 11";
```

La directive `state_encoding` spécifie la nature du code interne pour les valeurs d'un type énuméré.

```
attribute state_encoding of type-name:type is value;
```

Les valeurs légales de la directive `state_encoding` sont `sequential`, `one_hot_zero`, `one_hot_one`, and `gray`.

sequential : on code en binaire au fur et à mesure de l'énumération avec autant de bits que nécessaire.

one\_hot\_zero : on code la première valeur par zéro, puis le reste en utilisant à chaque fois un seul un : N états nécessiteront donc N-1 bits.

one\_hot\_one : idem à `one_hot_zero` sauf que l'on n'utilise pas le code zéro. N états nécessiteront donc N bits.

Gray : les états suivent un code GRAY.

Exemples :

```
type state is (s0,s1,s2,s3);
attribute state_encoding of
state:type
is one_hot_zero;
```

```
type s is (s0,s1,s2,s3);
attribute state_encoding of s:type is
gray;
```

### Exercice 1

Design a state machine that detects, starting with the left most bit, the sequence « 1111010 ».

a) Draw the state flow diagram

b) Find the corresponding sequential functions with state encoding and the VHDL program.

**Exercice 2**

Ecrire les équations de récurrence du réveil avec et sans codage des états. On utilisera les deux codages d'états des attribute state\_encoding ci-dessus.

**IV) Les forçages synchrones et asynchrones**

Il existe deux façons pour initialiser en utilisant les fronts d'horloge (synchrone) ou non (asynchrone).

```
-- gestion de l'asynchrone
process(clk,reset) begin
  if reset='1' then
    -- avant horloge donc asynchrone
    q<= "0000";
  elsif clk'event and clk='1' then
    -- ici le synchrone
  end if;
end process;
```

```
1°) façon
-- initialisation synchrone
process(clk) begin
  if clk'event and clk='1'then
    -- equations de récurrence +
    -- init ou
    -- equations de récurrence
    --./init
  end if;
end process;
```

```
2°) façon

if clk'event and clk='1'then
  if init ='1' then
    q<= "0000";
  else
    -- ici case ou equations de
    -- récurrences
  end if;
end if;
```

**Exercice 3 DS a2i13 (2003)**

Un étudiant a utilisé active-FSM et généré un programme VHDL correspondant au dessin ci-dessous. Puis il a réalisé un projet pour compiler et prétend que les extraits ci-dessous sont tirés du fichier de rapport correspondant. On vous demande de dire si l'étudiant a raison ou pas. Pour cela on vous propose de remplir l'habituel tableau état présent état futur ci-dessous et d'en déduire s'il correspond bien au dessin. Pour remplir le tableau, vous utiliserez naturellement les équations de récurrence du fichier rapport. Dans ce fichier, la notation

Etat présent s s0	Condition e0 e1 Init	Etat futur s <sup>+</sup> s0 <sup>+</sup>
0 0	0 X 0	
0 0	1 X 0	
0 1	0 X 0	
0 1	1 X 0	
1 0	X 0 0	
1 0	X 1 0	
1 1		

truc.D signifie truc<sup>+</sup> et truc.Q signifie truc. On notera sregSBV\_0 tout simplement

s0.

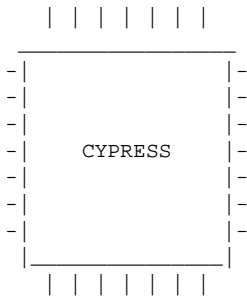
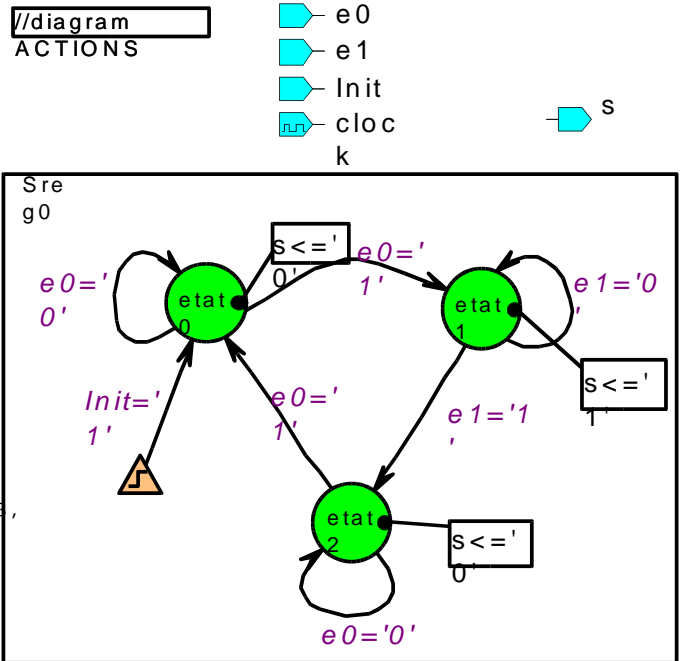
**Réponse :**

Une croix sur une entrée indique que l'on n'a pas besoin de connaître sa valeur pour en déduire l'état futur. Etudier l'état futur de 11. sreg0SBV\_0 est le poids faible, s est le poids fort.

(SBV=State Bit Vector)

Conclusion, est-ce bien le fichier correspondant ?

ds02  
03



....  
 state variable 'sreg0' is represented by a Bit\_vector(0 to 1).  
 State encoding (sequential) for 'sreg0' is:

```
etat0 := b"00";
etat1 := b"01";
etat2 := b"10";
```

PLD Compiler Software: PLA2JED.EXE 21/SEP/1998 [v4.02 ] 5.1 IR 14  
 (19:08:48)

```
DESIGN EQUATIONS
sreg0SBV_0.D = /e0 * /init * sreg0SBV_0.Q
              + e1 * /init * s.Q
s.D = e0 * /init * /s.Q * /sreg0SBV_0.Q
      + /e1 * /init * s.Q
```

C20V8C

clock =	1	24	* not used
init =	2	23	* not used
e1 =	3	22	* not used
e0 =	4	21	* not used
not used *	5	20	* not used
not used *	6	19	* not used
not used *	7	18	* not used
not used *	8	17	* not used
not used *	9	16	= (sreg0SBV_0)
not used *	10	15	= s
not used *	11	14	* not used
not used *	12	13	* Reserved

## TD4 A2I13 : VHDL et CAO

### I) VHDL et la librairie IEEE

Les seuls types utilisés jusqu'à maintenant sont les « bit » et « bit\_vector ». Un bit prend seulement deux valeurs et ne permet pas de gérer le trois états par exemple. IEEE propose en supplément une librairie appelée std\_logic. Son utilisation nécessite l'écriture de

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

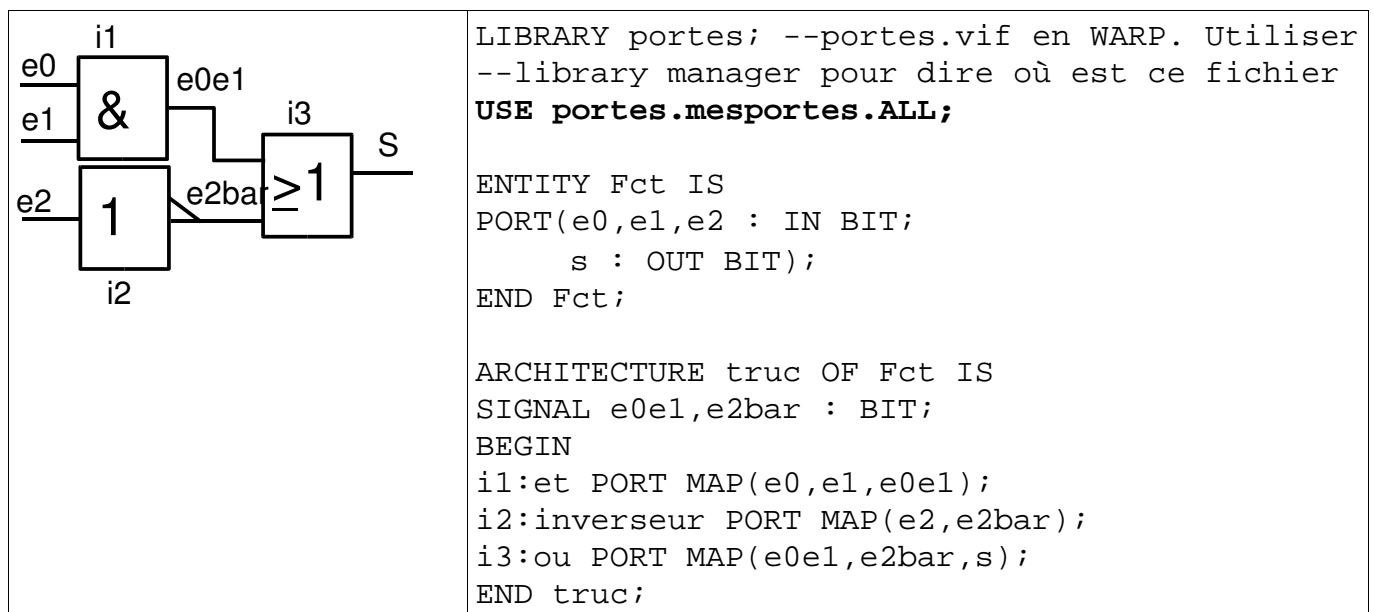
en début du programme qui l'utilise On a alors accès aux types «std\_logic» et «std\_logic\_vector». Les valeurs prises par ces types sont :

'U' Uninitialised	'Z' High Impedance
'X' Forcing Unknow	'W' Weak Unknow
'0' Forcing 0	'L' Weak 0
'1' Forcing 1	'H' Weak 1
'-' -- Don't Care	

On dispose de plus des fonctions rising\_edge et falling\_edge.

### II) Ecrire son propre package (schémas et netlist)

Lors du TD2 de a2i11 (assemblage de fonctions) nous avons passé sous silence le fait que pour faire la description structurelle présentée il fallait utiliser notre propre package définissant ce qu'est un et, un ou et un inverseur. Nous écrivons ci-dessous la version complète du programme.



Voici en condensé comment on réalise un package :

<b>PACKAGE mesportes IS</b>		
COMPONENT et PORT(e0,e1 : IN BIT; s : OUT BIT); END COMPONENT;	COMPONENT ou PORT(e0,e1 : IN BIT; s : OUT BIT); END COMPONENT;	COMPONENT inverseur PORT(e : IN BIT; s : OUT BIT); END COMPONENT;
<b>END mesportes;</b>		

```
ENTITY et IS
PORT(e0,e1 : IN BIT;
s : OUT BIT);
END et;

ARCHITECTURE aet OF et
IS
BEGIN
s<=e0 AND e1;
END aet;
```

```
ENTITY ou IS
PORT(e0,e1 : IN BIT;
s : OUT BIT);
END ou;

ARCHITECTURE aou OF ou
IS
BEGIN
s<=e0 OR e1;
END aou;
```

```
ENTITY inverseur IS
PORT(e : IN BIT;
s : OUT BIT);
END inverseur;

ARCHITECTURE ainv OF
inverseur IS
BEGIN
s<= NOT e;
END ainv;
```

Il n'y a aucun standard sur les packages du type ci-dessus. Chaque constructeur propose son propre package pour programmer ses composants ce qui pose un problème de portabilité. A noter quand même une initiative avec LPM (Library of Parameterized Modules) (voir plus loin).

### **III) Génération automatique de Netlist**

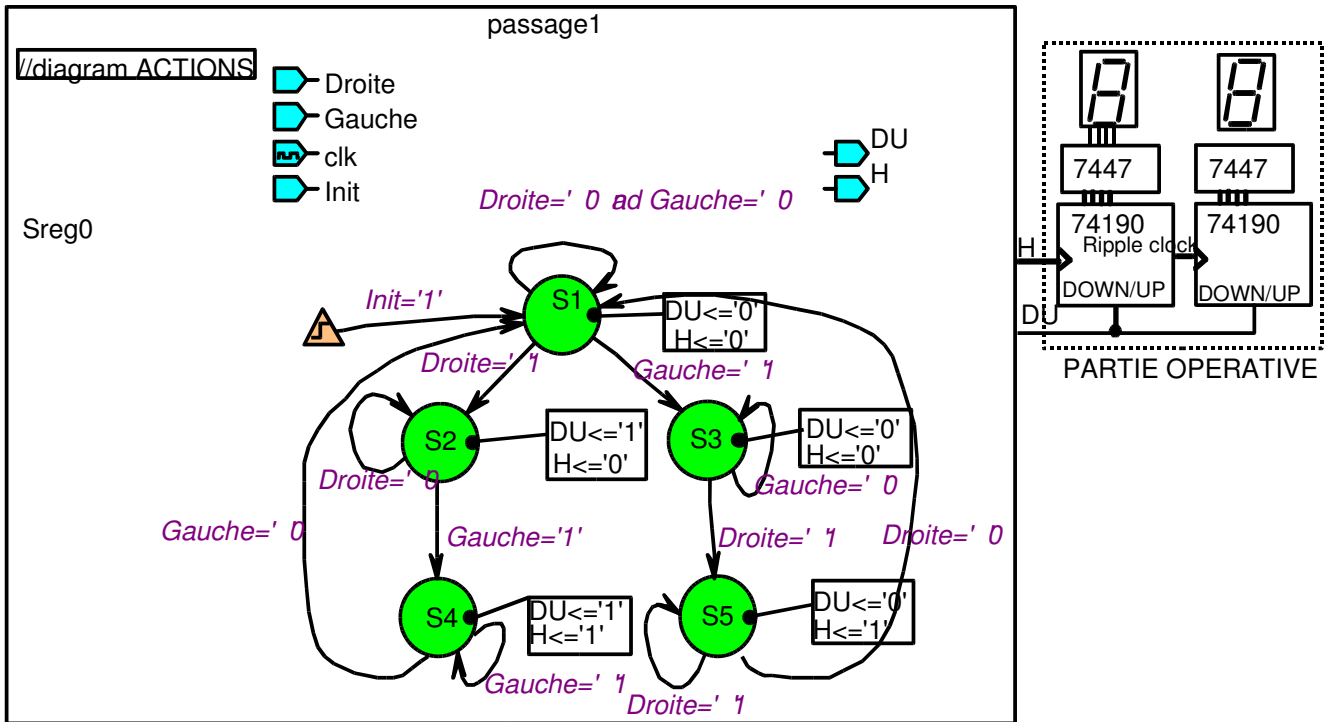
La majorité des logiciels de schéma modernes sont capables de transformer des schémas en netlist VHDL. Il est donc possible à partir d'un schéma, de générer un PCB ou de partir vers la logique programmable.

### **IV) Décomposer une synthèse complexe**

Lorsque l'on doit réaliser un circuit complexe, il convient en général de le décomposer en partie de commande et partie opérative (voir a2i12 et algorithmes). Cette décomposition demande une expérience d'autant plus grande que les parties utilisées se décomposent elles-mêmes en sous-parties... Disons pour simplifier que le niveau simple correspond à ce que l'on fait en schéma logique traditionnel : utiliser des composants simples existants. C'est ce niveau qui va nous intéresser maintenant.

### **V) Utilisations de compteurs**

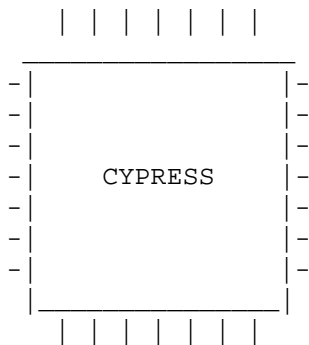
Le compteur de passages des Tps est un exemple d'utilisation de compteurs séquencés.



**Exercice 1**

On donne le fichier rapport correspondant au compteur de passage.

- 1°) Préciser si les états sont codés et si oui, comment ?
- 2°) Réécrire les équations de récurrence et vérifier les transitions à partir de l'état S1.
- 3°) Corriger le diagramme d'évolution qui ne peut fonctionner tel quel (oreilles de Mickey de S2 et S3).
- 4°) Ecrire un programme VHDL component + netlist... décrivant le compteur de passage complet dans un seul composant. (Cette synthèse conduit à un warning justifié concernant la fabrication de l'horloge du 74190 par du combinatoire, ce qui est déconseillé)
- 5°) Refaire l'ensemble de manière correcte : partie commande sur front montant d'une horloge et partie opérative sur front descendant d'une même horloge.



Warp VHDL Synthesis Compiler: Version 5.2 IR 17  
 Copyright (C) 1991, 1992, 1993,  
 1994, 1995, 1996, 1997, 1998, 1999 Cypress Semiconductor

.....  
 State variable 'sreg0' is represented by a Bit\_vector (0 to 2).  
 State encoding (sequential) for 'sreg0' is:

```

s1 :=      b"000";
s2 :=      b"001";
s3 :=      b"010";
s4 :=      b"011";
s5 :=      b"100";

```

.....

```

PLD Compiler Software:          PLA2JED.EXE      02/APR/1999  [v4.02 ] 5.2
IR 17

```

```

DESIGN EQUATIONS                (10:35:11)

```

```

h = du.Q * sreg0SBV_1.Q
    + sreg0SBV_0.Q

```

```

du.D = /sreg0SBV_0.Q * /sreg0SBV_1.Q * droite * /gauche * /init
      + du.Q * gauche * /init
      + du.Q * /sreg0SBV_1.Q * /init

```

```

du.C = clk

```

```

sreg0SBV_0.D = /du.Q * sreg0SBV_1.Q * droite * gauche * /init
              + sreg0SBV_0.Q * droite * /init

```

```

sreg0SBV_0.C =  clk

```

```

sreg0SBV_1.D = /du.Q * /sreg0SBV_0.Q * /droite * gauche * /init
              + /sreg0SBV_0.Q * /sreg0SBV_1.Q * droite * gauche * /init
              + /du.Q * sreg0SBV_1.Q * /gauche * /init
              + du.Q * sreg0SBV_1.Q * gauche * /init

```

```

sreg0SBV_1.C =  clk

```

## **Exercice 2**

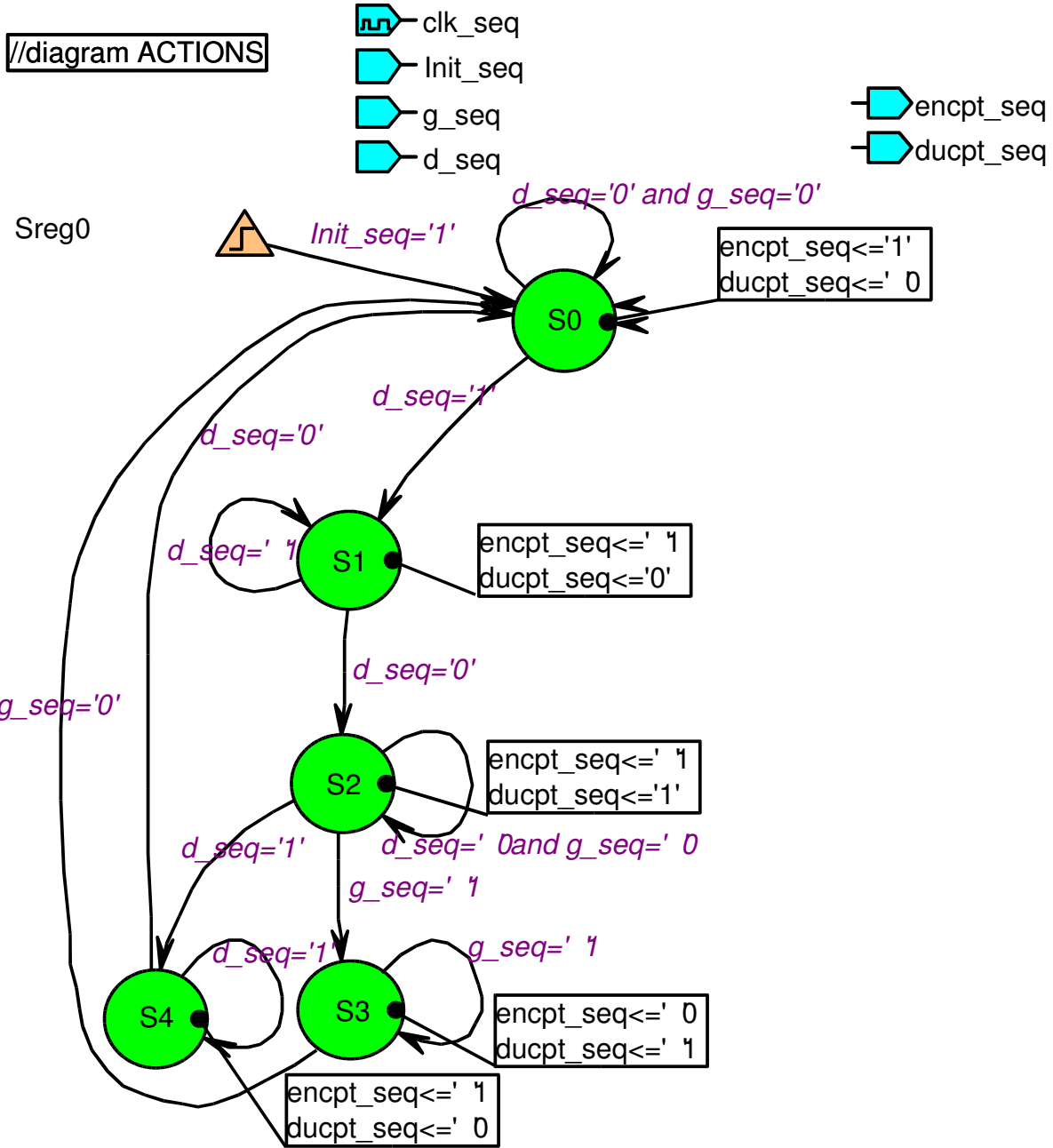
1°) On donne le diagramme d'évolution partiel du séquenceur du compteur de passages avec demi-tour. Compléter ce diagramme d'évolution sachant que la gestion des compteurs 74190 se fait maintenant avec l'entrée de validation EN et le comptage/décomptage DOWN/UP.

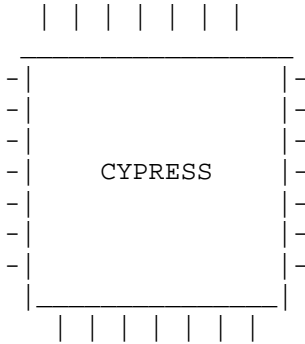
2°) On désire implanter le compteur de passage complet dans un FPGA suivant le schéma donné ci-après. Ecrire tous les composants correspondant à une éventuelle librairie qui contiendrait tous les éléments nécessaires à cette synthèse.

3°) Ecrire le programme structurel décrivant ce même schéma.

**Remarque :** la partie compteur n'est pas implantée comme dans l'exercice précédent. L'implantation de l'exercice 1 est possible mais elle conduira à un warning car la génération d'une horloge par du combinatoire est une opération fortement déconseillée.

Passage2





Warp VHDL Synthesis Compiler: Version 5.1 IR 14  
 Copyright (C) 1991, 1992, 1993,  
 1994, 1995, 1996, 1997, 1998 Cypress Semiconductor

```
=====
Compiling: Passage2.vhd
Options:   -yu -e10 -w100 -o2 -ygs -fP -v10 -dc20v8 -pPALCE20V8-25PC -u passage2.hie
Passage2.vhd
```

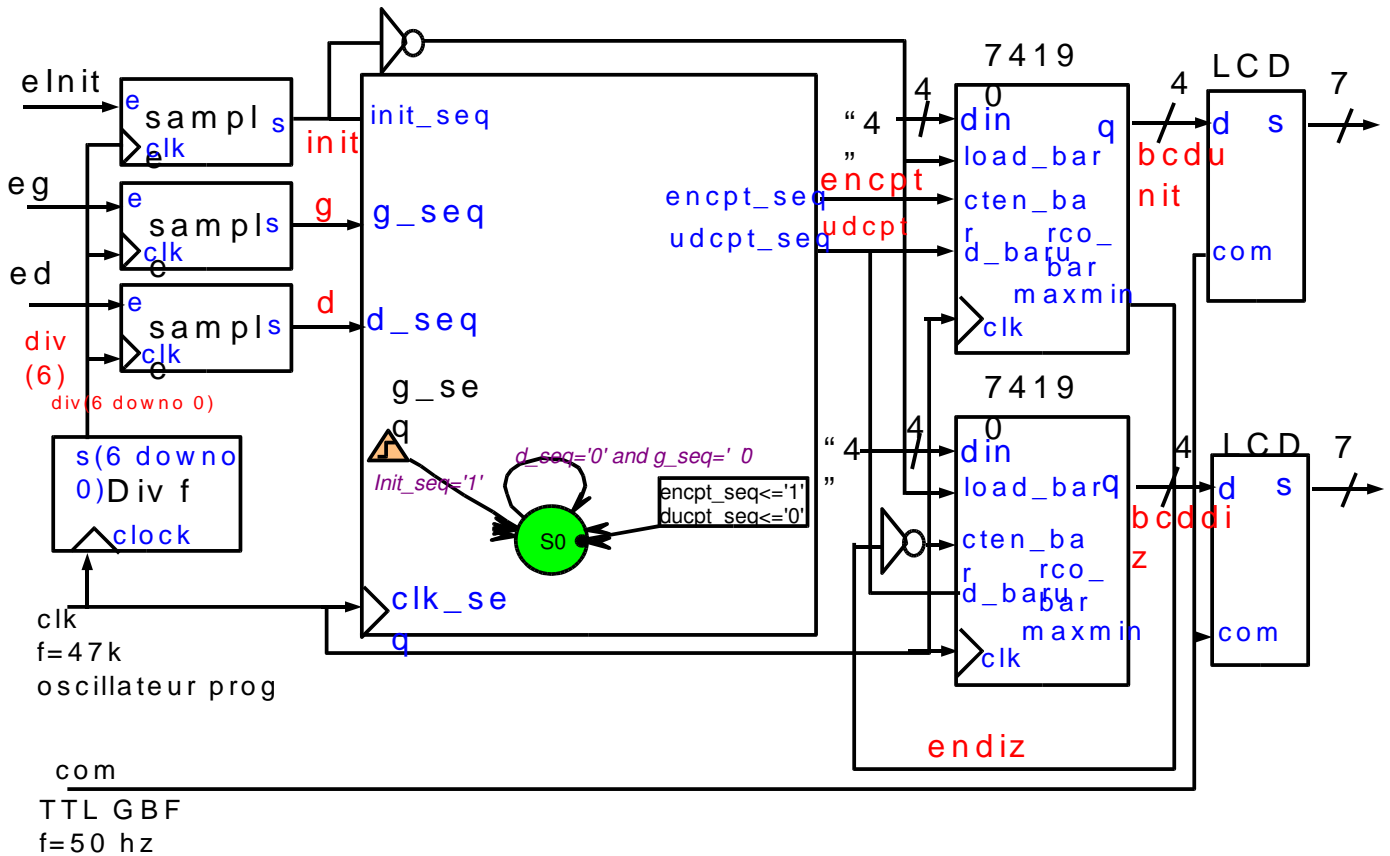
---

State encoding (sequential) for 'sreg0' is:

```
s0 := b"0000";
s1 := b"0001";
s2 := b"0010";
s3 := b"0011";
s4 := b"0100";
s5 := b"0101";
s6 := b"0110";
s7 := b"0111";
s8 := b"1000";
```

#### C20V8C

clk_seq =	1	24	* not used
d_seq =	2	23	* not used
init_seq =	3	22	* not used
gauche =	4	21	* not used
not used *	5	20	= encpt_seq
not used *	6	19	= udcpt_seq
not used *	7	18	= (sreg0SBV_0)
not used *	8	17	= (sreg0SBV_1)
not used *	9	16	= (sreg0SBV_2)
not used *	10	15	= (sreg0SBV_3)
not used *	11	14	* not used
not used *	12	13	* Reserved



## VI) Utilisation de registres à décalage

Nous nous trouvons dans la situation suivante : nous disposons de plusieurs registres qui forment ce que l'on appellera notre partie opérative, et nous désirons synthétiser un circuit qui les utilise. Ce circuit peut être combinatoire ou séquentiel. Nous allons appréhender cela avec un exercice.

### Exercice 3

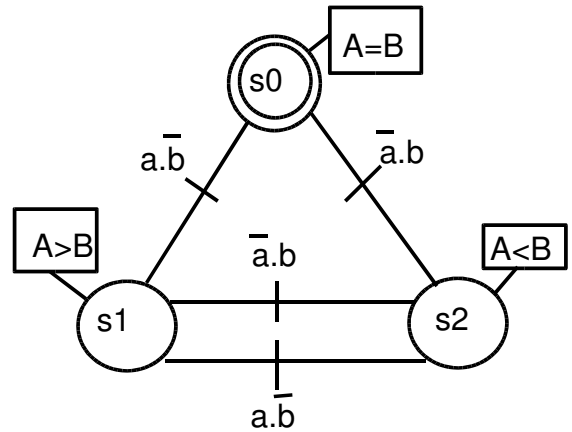
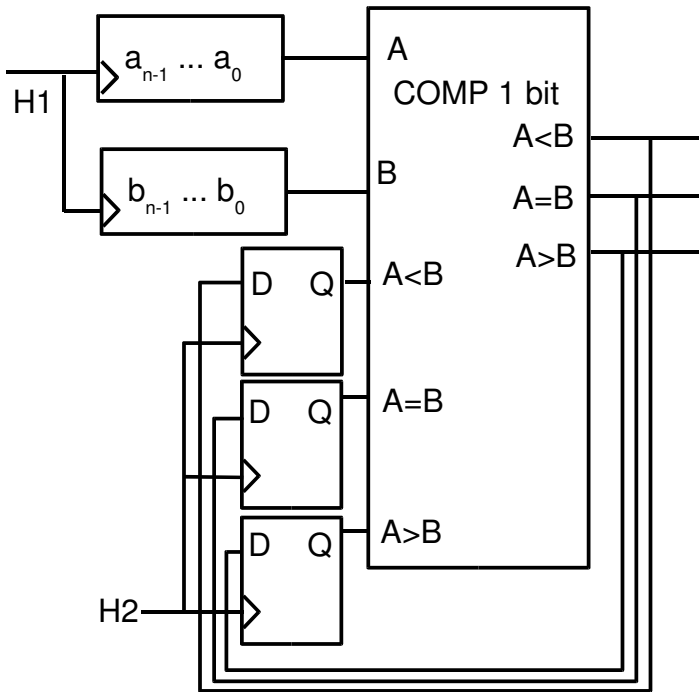
Draw the state diagram for the following state machines with inputs A and B :

(1) a serial comparator with outputs A>B, A=B and A<B.

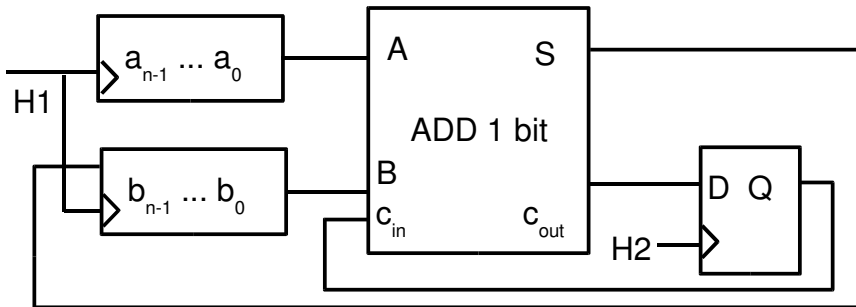
(2) a serial adder, and in each case the data enters least-significant bit first.

Show how a third synchronous input can be used to delimit words, and modify the flowcharts to include it. Write the corresponding VHDL program.

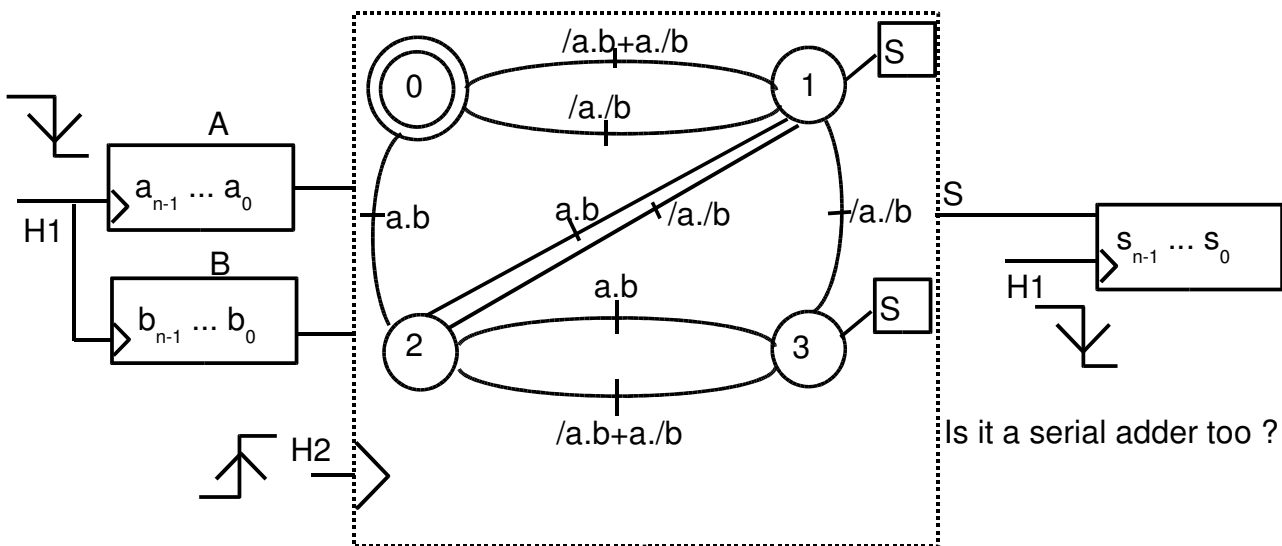
**Hint :**



Finite State Machine for comparator.



What is a serial adder ?



Is it a serial adder too ?

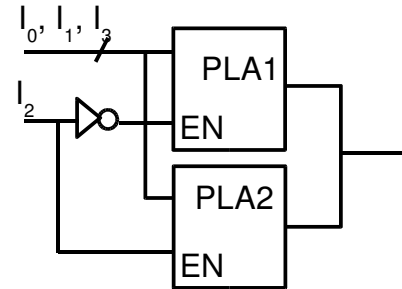
Serial adder with Finite State Machine

## VII) Utiliser plusieurs PALs

Parfois un ensemble de fonctions combinatoires ne peuvent pas tenir dans un seul composant. La solution est de décomposer le problème de telle manière que cet ensemble tienne dans un réseaux de deux ou plusieurs composants.

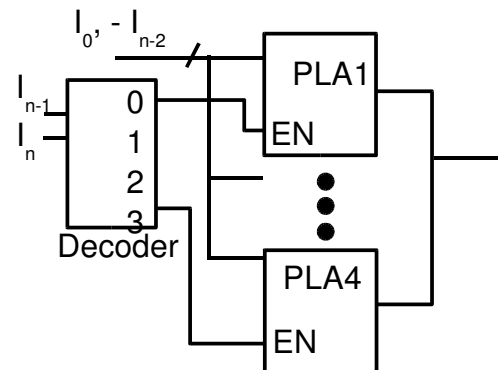
### 1°) Extension du nombre termes produits

La technique consiste à diviser la table des produits de telle manière qu'une variable soit à 0 dans un ensemble et à un dans l'autre. Si cette variable n'apparaît pas dans certains termes on double le terme en la mettant à 0 dans un ensemble et à un dans l'autre.



### 2°) Extension du nombre d'entrées

La technique de base est montrée ci-contre. Elle possède plusieurs variantes. Par exemple l'une d'entre elle consiste à remplacer le décodeur par une PAL.



### Exercice 3

The 20V8 PAL has a maximum of eight products per output. Show how this can be raised to 14 by using the output enable control terms. What condition must be satisfied by the products in such a solution ?

### Exercice 4

Design a 4-bit shifter. This combinational logic device has seven data inputs, D0 – D6, and four data outputs Y0 – Y3. The function control inputs I0 and I1, determine the number of places the input data is to be shifted. Table below defines the operation of the circuit. Select the most suitable type of PLD and specify its personality in the most compact way. Give the VHDL program.

I1	I2	Y3	Y2	Y1	Y0
0	0	D6	D5	D4	D3
0	1	D5	D4	D3	D2
1	0	D4	D3	D2	D1
1	1	D3	D2	D1	D0

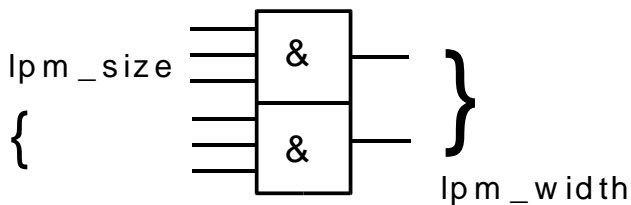
## VIII) La synthèse LPM (Library of Parameterized Modules)

(<http://www.edif.org/lpmweb/>)

Les méthodes de synthèse traditionnelles ne peuvent pas être balayées sous prétexte que l'on utilise un langage au lieu d'un schéma. Au début des compilateurs VHDL étaient proposés des bibliothèques TTL (voir Warp avant version 5). Une technique commune et plus générale est proposée par un ensemble de fournisseurs VHDL sous l'impulsion de Altera : Library of Parameterized Modules ou LPM. L'intérêt par rapport à une bibliothèque classique est l'emploi de la généricité et surtout de permettre au concepteur d'utiliser des techniques de synthèse qu'il connaissait déjà.

### **1°) Utilisation de Mand**

Composant lpm : Mand



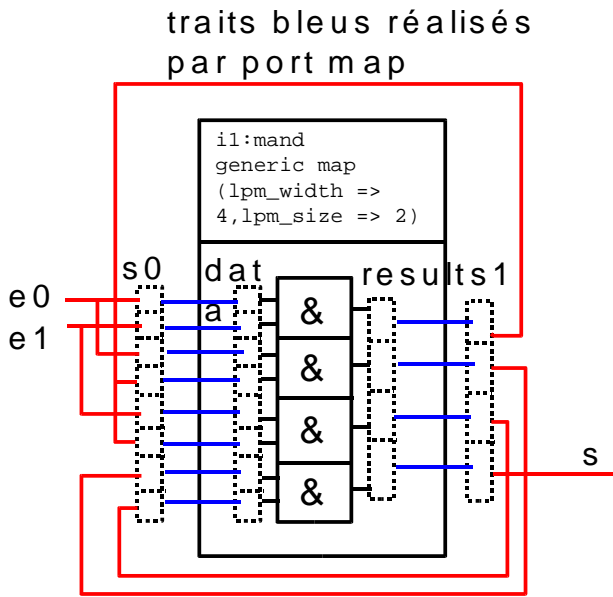
```
library ieee,lpmpkg,cypress;
use cypress.cypress.all;
use lpmpkg.lpmpkg.all;
use ieee.std_logic_1164.all;
```

```
ENTITY demoLPM IS
    PORT ( e0,e1:in std_logic;s:out std_logic );
END demoLPM;
```

```
ARCHITECTURE ademoLPM OF demoLPM IS
    signal s0 : std_logic_vector(7 downto 0);
    signal s1 : std_logic_vector(3 downto 0);
BEGIN
    s0(0)<=e0;
    s0(1)<=e1;
    s0(2)<=e0;
    s0(3)<=s1(0);
    s0(4)<=e1;
    s0(5)<=s1(0);
    s0(6)<=s1(1);
    s0(7)<=s1(2);
    s<=s1(3);
    i1:mand
        generic map (lpm_width => 4,lpm_size => 2)
            port map(data=>s0,result=>s1);
END ademoLPM;
```

**Cet exemple n'a aucun intérêt : fabriquer une et à 2 entrées à partir de 4**

**et à 2 entrées !**

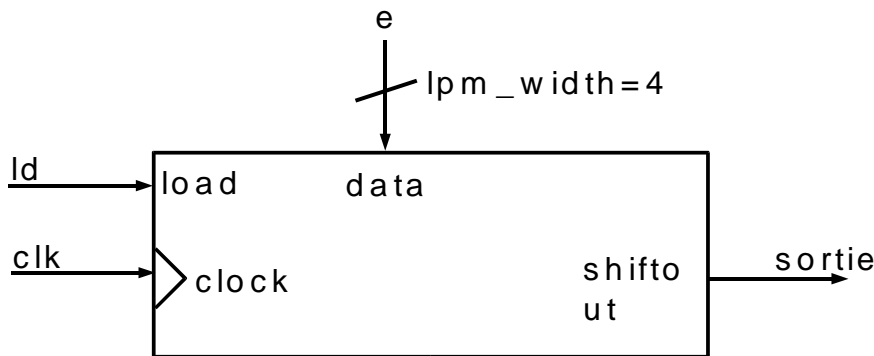


```

component mand
generic(lpm_width : positive;
lpm_size: positive,
lpm_hint: goal_type := SPEED;
lpm_data_pol: string := "");
lpm_result_pol:string := "");
port( data: in std_logic_vector
((lpm_width*lpm_size)-1 downto 0));
result: out std_logic_vector
(lpm_width-1 downto 0));
end component;
    
```

**2°) Essai d'un registre à décalage LPM**

On cherche à réaliser le schéma ci-dessous :



Cela donne le programme suivant :

```

library ieee,lpmpkg,cypress;
use ieee.std_logic_1164.all;
use lpmpkg.lpmpkg.all;
use cypress.all;
ENTITY essai_reg IS
    --GENERIC ( );
    PORT ( e:in std_logic_vector(3 downto 0);
    ld,clk:in std_logic;
    sortie:out std_logic );
END essai_reg;

ARCHITECTURE aessai_reg OF essai_reg IS
-- Constant signals
    signal zero : std_logic := '0' ;
    
```

```

signal one : std_logic := '1' ;
component mshiftreg
  generic(lpm_width : positive);
  -- lpm_direction : shdir_type:= LPM_LEFT;
  -- lpm_avalue : string := "";
  -- lpm_svalue : string := "";
  -- lpm_pvalue : string := "";
  -- lpm_hint : goal_type := SPEED);
  port( data : in std_logic_vector(lpm_width-1 downto 0) ;
  -- :=(others => zero);

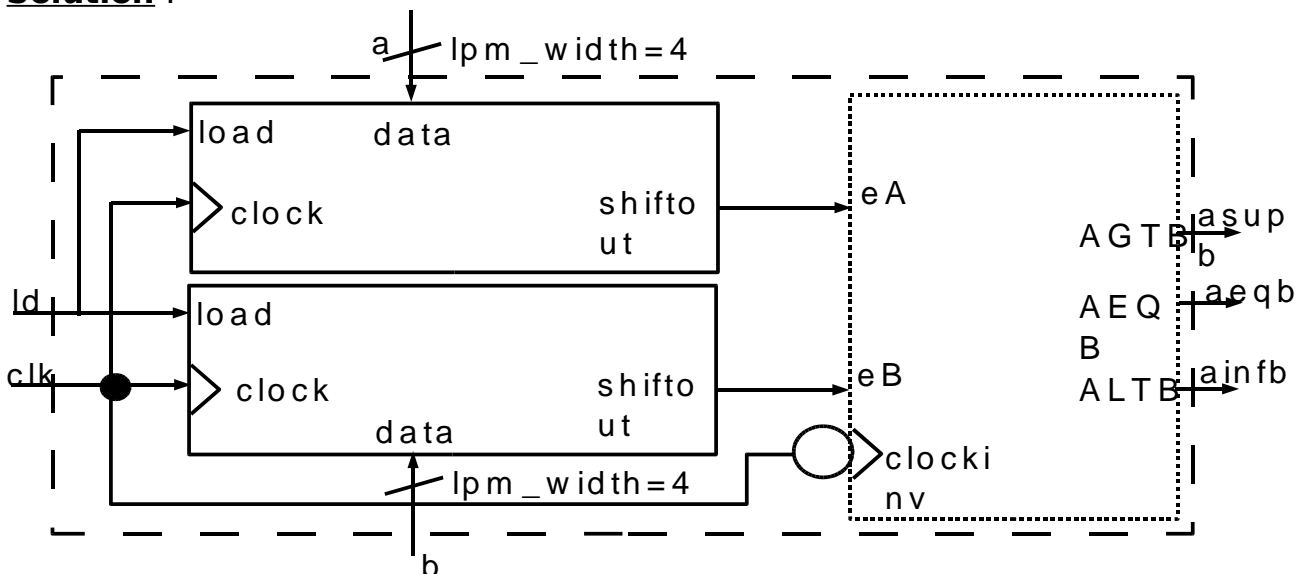
  clock : in std_logic;
  enable : in std_logic := one;
  shiftin : in std_logic := zero;
  load : in std_logic := zero;
  q : out std_logic_vector(lpm_width-1 downto 0);
  shiftout : out std_logic;
  aset : in std_logic := zero;
  aclr : in std_logic := zero;
  sset : in std_logic := zero;
  sclr : in std_logic := zero;
  testenab : in std_logic := zero;
  testin : in std_logic := zero;
  testout : out std_logic);
end component;
BEGIN
  net:mshiftreg
  GENERIC MAP (lpm_width=>4)
  PORT MAP (data =>e,load=>ld,clock=>clk,shiftout=>sortie);
END aessai_reg;

```

**Exercice 5**

Reprendre le comparateur série en utilisant les LPMs

**Solution :**



```

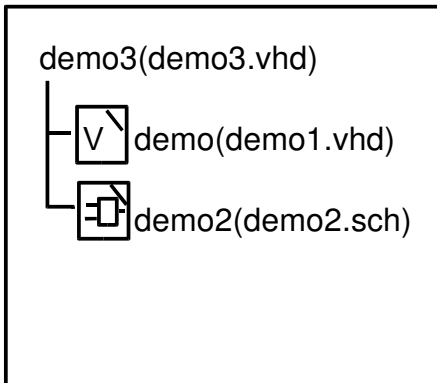
library ieee,lpmpkg,cypress;
use ieee.std_logic_1164.all;
use lpmpkg.lpmpkg.all;
use cypress.all;
ENTITY essai_reg IS
  --GENERIC ( );
  PORT ( a,b:in std_logic_vector(3 downto 0);
        ld,clk:in std_logic;
        asupb,ainfb,aeqb:out std_logic );
END essai_reg;
ARCHITECTURE aessai_reg OF essai_reg IS
  signal zero : std_logic := '0' ;
  signal one  : std_logic := '1' ;
  signal s0,s1 : std_logic;
  component mshiftreg
    generic(lpm_width      : positive);
    port(  data            : in std_logic_vector(lpm_width-1 downto 0) ;
          clock            : in std_logic;
          enable           : in std_logic := one;
          shiftin          : in std_logic := zero;
          load             : in std_logic := zero;
          q                 : out      std_logic_vector(lpm_width-1 downto 0);
          shiftout         : out      std_logic;
          aset             : in std_logic := zero;
          aclr             : in std_logic := zero;
          sset             : in std_logic := zero;
          sclr             : in std_logic := zero;
          testenab        : in std_logic := zero;
          testin           : in std_logic := zero;
          testout          : out      std_logic);
  end component;
  component serialcomp is
    port (clkinv: in STD_LOGIC;
          eA,eB,Init: in STD_LOGIC;
          AEQB,AGTB,ALTB: out STD_LOGIC);
  end component;
BEGIN
  net1:mshiftreg
  GENERIC MAP (lpm_width=>4)
  PORT MAP (data =>a,load=>ld,clock=>clk,shiftout=>s0);
  net2:mshiftreg
  GENERIC MAP (lpm_width=>4)
  PORT MAP (data =>b,load=>ld,clock=>clk,shiftout=>s1);
  net3:serialcomp
  PORT MAP (eA=>s0,eB=>s1,clkinv=>clk,Init=>ld,AEQB=>aeqb,
           ALTB=>ainfb,AGTB=>asupb);
END aessai_reg;

```

### 3°) Comment Xilinx mélange VHDL et schémas

A priori les LPM sont absents de XILINX mais remplacés par des bibliothèques très riches.

Projet :

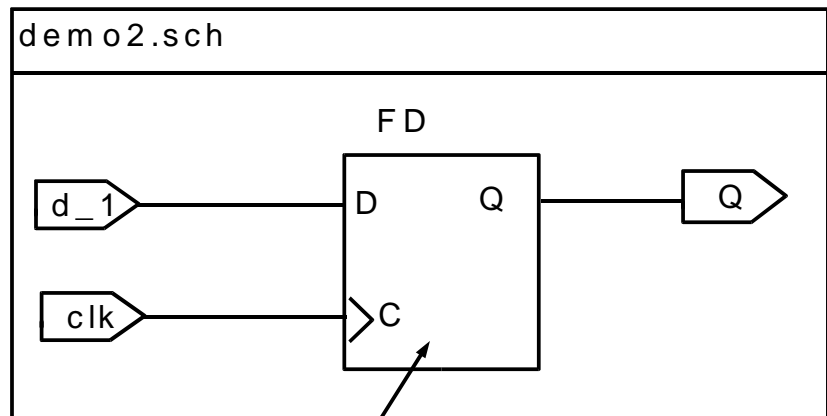


```

-- demo3.vhd
entity demo3 is
  Port ( globale : in bit_vector(3 downto 0);
        globals : out bit_vector(1 downto 0));
end demo3;
architecture Behavioral of demo3 is
  component demo
  port (e:in bit_vector(1 downto 0);
        s:out bit);
end component;
  component demo2
  port (D_1,clk:in bit;
        Q:out bit);
end component;
begin
  map1:demo
    port map(e=>globale(1 downto 0),s=>globals(0));
  map2:demo2
    port map(D_1=>globale(2),clk=>globale(3),
            Q=>globals(1));
end Behavioral;
  
```

```

-- demo1.vhd
entity demo is
  port (e:in bit_vector(1
  downto 0);
        s:out bit);
end demo;
architecture ademo of demo
is
begin
  s<=e(0) and not e(1);
end ademo;
  
```



flip\_flop  
fd

### Exercice 6

- 1°) Faire un schéma fonctionnel du projet présenté dans ces pages.
- 2°) Implanter sous xilinx webpack
- 3°) Essayer la génération de code VHDL par dessin de machine d'états.
- 4°) Réaliser l'additionneur série du TD4.

## TD5 A2I13 : VHDL et simulation

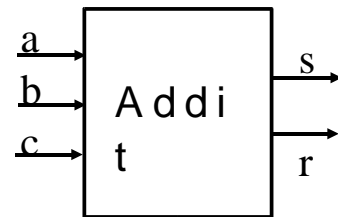
Ce Td est présent parcequ'il a été écrit mais n'est pas fait avec nos étudiants

VHDL est un langage de spécification, il est donc très lié à la simulation. Nous allons présenter un certain nombre de façons de simuler. La simulation dépend beaucoup des fournisseurs de VHDL.

### I) Test Bench = test en VHDL [voir FreeVHDL (<http://www.freehdl.seul.org>)]

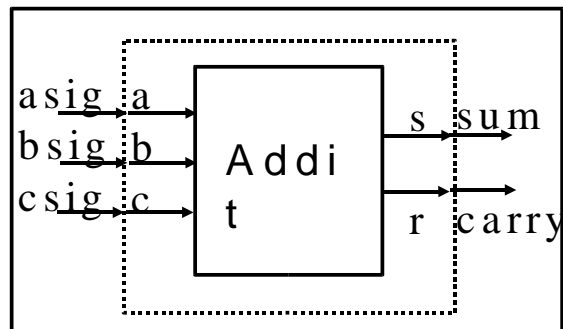
Nous allons commencer par deux exemples :

```
entity addit is port (
a,b,c: in bit;
s,r: out bit);
end addit;
architecture addit of addit is
begin
s <= a xor b xor c;
r <= ((a and b) or (c and (a xor b)));
end addit;
```



Si l'on veut tester cet additionneur il faut bâtir un fichier destiné à construire les signaux de tests : on appelle cela un Test Bench. On parle classiquement de test bench lorsque la simulation se réalise en VHDL (sauf pour visualiser les résultats). On prendra par exemple : fichier demo1.vhdl

```
entity demol is
end demol;
architecture demol of demol is
signal asig,bsig,csig,sum,carry : bit;
begin
asig <= not asig after 100 ns;
bsig <= not bsig after 200 ns;
csig <= not csig after 400 ns;
ad1:entity work.addit
port map (asig,bsig,csig,sum,carry);
end demol;
```



Autre exemple (séquentiel celui-là) :

```
entity cmpt is port (
clk: in bit;
q0,q1: inout bit);
end cmpt;
architecture cmpt of cmpt is
begin
cmpt1 : process (clk) begin
if (clk'event and clk='1') then
q0 <= not q0;
q1 <= q0 xor q1;
end if;
end process;
end cmpt;
```

```

end if;
end process;
end cmpt;

```

Si l'on veut tester ce compteur il faut bâtir un fichier destiné à construire les signaux de tests. On prendra par exemple : fichier demo2.vhdl

```

entity demo2 is
end demo2;
architecture demo2 of demo2 is
    signal clksig,q0sig,q1sig : bit;
begin
    clksig <= not clksig after 10 ns;
    cpt1:entity work.cmpt
    port map (clksig,q0sig,q1sig);
end demo2;

```

**Remarque** : la plupart des simulateurs y compris celui de Warp (Aldec) acceptent ce style de test.

## II) Façon Alliance (<http://www-asim.lip6.fr/recherche/alliance/>)

Nous allons commencer par deux exemples (combinatoire puis séquentiel) :

a3	a2	a1	a0	s1	s0
0	1	0	1	1	1
0	1	1	0	0	1
1	1	0	1	1	0

```

ENTITY demo IS
    PORT(
        a : in BIT_VECTOR(0 TO 3) ;-- 4 entrees
        s : out BIT_VECTOR(0 TO 1) ;-- 2 sorties
        vdd, vss : in BIT -- les alimentations
    );
END demo;

```

(ce qui n'est pas mentionné correspond à 00 en sortie )

C'est un fichier de patterns qu'il faudra créer soit manuellement soit avec un outil spécialisé (toute ligne commençant par un # est un commentaire) :

```

in vdd;
in vss;
in a(3 to 0);
out s(1 to 0);
begin
# a=0101, we expect value 3 on s
pat_1 : 1 0 0101 ?11 ;
# a=0110, we expect value 1 on s
pat_2 : 1 0 0110 ?01 ;
# a=13, we expect value 2 on s
pat_3 : 1 0 1101 ?10 ;
# a=7,we still expect bin 0 on s
pat_4 : 1 0 0111 ?00 ;
end;

```

Pour le séquentiel, l'exemple du réveil du TD3 pourrait être testé avec :

```

in clock;
in key;
in trip;
out ring;
begin
pat_1 : 000 ?u; #u car indéterminé
pat_2 : 100 ?0;
pat_3 : 010 ?0;
pat_4 : 110 ?0;
pat_5 : 011 ?0;
pat_6 : 111 ?1;
pat_7 : 001 ?1;
pat_8 : 101 ?0;
end;

```

Remarque : l'outil asimut d'alliance permet aussi de prendre en compte le temps et surtout permet d'écrire des tests en C.

### **III) Façon Warp**

C'est une méthode graphique intuitive qui ne demande pas d'explications. Il est possible de réaliser des Tests Bench aussi.

### **IV) Exercice**

#### **Exercice 1**

(Smith « Application Specific Integrated Circuits » Addison Wesley (1997)

Vocabulaire : beetle<->scarabé, gem<->pierre précieuse, maze<->labyrinthe  
overhead<-aérien

(\*\*\*Beetle problem) (Based on a problem by Seitz.)

A planet has many geological gem mazes: A maze covers a square km or so, on a 10 mm grid; a maze cell is 10 mm by 10 mm and gems lie at cell centers; there is a path from every maze cell to every other; on average one in 64 cells has an overhead opening; on average one in seven cells has a single gem; there are no gems under overhead openings.

You are to design a gem-mining beetle ASIC with the following inputs: a nominal 1 MHz single-phase clock, CLK; wall sensors: WL, WR, WF, WB (wall to left/right/forward/behind); light sensors: LL, LR, LF, LB (light left/right/forward/behind); low-battery indicator: BLOW; gem sensor: GEM (directly over a gem); opening sensor: OPEN (when under an opening).

All signals are active high and the light sensor outputs are mutually exclusive. The beetle ASIC must produce the following (mutually exclusive) signals: move forward, MF; move backward, MB; turn 90 degrees clockwise, TC; turn 90 degrees anticlockwise, TA; pick up a gem, PICKUP; throw gem up and out of overhead opening, THROWUP; jump up to surface and shut down, SHUTUP.

The beetle specifications and limitations are as follows: Beetles are dropped into the

maze to find the gems; beetles must find gems and carry them to an opening; beetles can eject gems through openings; beetles can carry only one gem at a time. A beetle move is one of the following: taking one step (moving to an adjacent cell), turning 90 degrees, picking up a gem, ejecting a gem, jumping out of opening-all take the same time and energy. A battery can provide energy for about 200 moves before the low-battery signal comes on. After the low-battery warning is signaled the battery has energy for 50 moves to find an overhead opening, and the beetle must then eject itself for recharging. The cost of the beetle determines that we would like the probability of losing a beetle be below 0.01.

The following describes a state machine to drive a beetle. Jim Rowson used a state machine language that he developed-along with the first CAD tool that could automatically create state machines:

# Jim Rowson's beetle

```
sm smbt1;
clock clk;
reset res                                --> resetState;
inputs WL WL WR WB GEM LF LL LR LB OPEN BLOW;
outputs MF=0 MB=0 TC=0 TA=0 PICKUP=0 THROWUP=0 SHUTUP=0;
outputs haveAgem SHUTUP;
let getout = (BLOW|haveAgem) & (LL|LF|LR|LB);

state resetState                          --> searchState haveAgem=0 SHUTUP=0;
state searchState
    BLOW & OPEN                            --> jumpState,
    haveAgem & OPEN                        --> ejectstate,
    getout & LL                             --> turnLstate,
    getout & LF                             --> goFwdState,
    getout & LR                             --> turnRstate,
    getout & LB                             --> turnAroundState,
    !haveAgem & GEM                         --> getGemState,
    !WL                                     --> turnLState,
    !WF                                     --> goFwdState,
    !WR                                     --> turnRState,
    !WB                                     --> turnAroundState;
state goFwdState                          --> MF searchState;
state turnLState                          --> TA goFwdState;
state turnRState                          --> TC goFwdState;
state turnAroundState                    --> TC turnAgainState;
state ejectState                          --> THROWUP !haveAgem searchState;
state jumpState                           --> SHUTUP shutdownState;
state getGemState                         --> PICKUP haveAgem searchState;
state shutDownState                      --> SHUTUP shutDownState;
state turnAgainState                     --> TC searchState;
end
```

(120 min.) Draw Jim's state machine diagram and translate it to an HDL.

(120 min.) Build a model for the maze that will work with Jim's design.

(120 min.) Simulate the operation of Jim's beetle using your maze model.

(Hours) Can you do better than Jim?

**Exercice 2 : Simulation textIO**

On donne le programme :

```
ENTITY Counter_1 IS end;
--library work;use work.textio.all;
library std;use std.textio.all; -- avec warp2
architecture behave of Counter_1 is
  signal clk : bit:='0';
  signal Count : integer:=0;
begin
  process begin
    wait for 10 ns;
    clk <= not clk;
    if( now > 340 ns) then wait; end if;
  end process;
  process begin
    wait until (Clk='0');
    if (Count=7) then Count <=0;
    else Count <=Count+1;
    end if;
  end process;
  process (Count) variable L:LINE; begin
    write(L,now);write(L,STRING(" Count= "));
    write(L,Count);writeline(Output,L);
  end process;
end;
```

Le faire fonctionner en écrivant le process de comptage dans un style plus familier pour nous : en utilisant une liste de sensibilité et un if clk'event.

Réaliser un module (librairie) qui permet de tester un afficheur 7 segments sur du texte. Si le temps manque, on vous donnera le module.

**Solution**

```
--library work;use work.textio.all;
library std;use std.textio.all;
PACKAGE test7seg IS
COMPONENT SeptSeg
  PORT (a : IN BIT_VECTOR(6 DOWNT0 0));
END COMPONENT;
END test7seg;
ENTITY SeptSeg IS
  PORT (a : IN BIT_VECTOR(6 DOWNT0 0));
END SeptSeg;
library std;use std.textio.all; --imperatif avant architecture
ARCHITECTURE aSeptSeg OF SeptSeg IS
BEGIN
  process (a) variable L:LINE;
  begin
    write(L,now);
    writeline(Output,L);
    if a(0)='1' then
      write(L,STRING(" **"));
      writeline(Output,L);
    end if;
  end process;
end;
```

```

else
  write(L,STRING'(" --"));
  writeline(Output,L);
end if;
if (a(5)='1' and a(1)='0') then
  write(L,STRING'("* |"));
  writeline(Output,L);
  write(L,STRING'("* |"));
  writeline(Output,L);
elseif (a(5)='1' and a(1)='1') then
  write(L,STRING'("* *"));
  writeline(Output,L);
  write(L,STRING'("* *"));
  writeline(Output,L);
elseif (a(5)='0' and a(1)='1') then
  write(L,STRING'("| *"));
  writeline(Output,L);
  write(L,STRING'("| *"));
  writeline(Output,L);
else
  write(L,STRING'("| |"));
  writeline(Output,L);
  write(L,STRING'("| |"));
  writeline(Output,L);
end if;
if a(6)='1' then
  write(L,STRING'(" **"));
  writeline(Output,L);
else
  write(L,STRING'(" --"));
  writeline(Output,L);
end if;
if (a(4)='1' and a(2)='0') then
  write(L,STRING'("* |"));
  writeline(Output,L);
  write(L,STRING'("* |"));
  writeline(Output,L);
elseif (a(4)='1' and a(2)='1') then
  write(L,STRING'("* *"));
  writeline(Output,L);
  write(L,STRING'("* *"));
  writeline(Output,L);
elseif (a(4)='0' and a(2)='1') then
  write(L,STRING'("| *"));
  writeline(Output,L);
  write(L,STRING'("| *"));
  writeline(Output,L);
else
  write(L,STRING'("| |"));
  writeline(Output,L);
  write(L,STRING'("| |"));
  writeline(Output,L);
end if;
if a(3)='1' then
  write(L,STRING'(" **"));
  writeline(Output,L);
else
  write(L,STRING'(" --"));
  writeline(Output,L);
end if;

```

```

end process;
END aSeptSeg;

```

## Un fichier test du style :

```

ENTITY transcod IS
  PORT(a:IN BIT_VECTOR(3 DOWNT0 0);s:OUT BIT_VECTOR(6 DOWNT0 0));
END transcod;
ARCHITECTURE atranscod OF transcod IS
BEGIN
  with a SELECT
  s <="0111111" WHEN "0000",
    "0000110" WHEN "0001",
    "1011011" WHEN "0010",
    "1001111" WHEN "0011",
    "1100110" WHEN "0100",
    "1101101" WHEN "0101",
    "1111101" WHEN "0110",
    "0000111" WHEN "0111",
    "1111111" WHEN "1000",
    "1101111" WHEN "1001",
    "0000000" WHEN OTHERS;
END atranscod;

```

```

ENTITY test7seg IS END;
library afficheur7seg;
use afficheur7seg.all;
ARCHITECTURE atest7seg OF test7seg IS
SIGNAL e4 :BIT_VECTOR(3 DOWNT0 0);
SIGNAL s7 : BIT_VECTOR(6 DOWNT0 0);
COMPONENT SeptSeg
  PORT (a : IN BIT_VECTOR(6 DOWNT0 0));
END COMPONENT;
COMPONENT transcod
  PORT(a:IN BIT_VECTOR(3 DOWNT0 0);s:OUT BIT_VECTOR(6 DOWNT0 0));
END COMPONENT;
BEGIN
  c1:transcod PORT MAP(e4,s7);
  c2:SeptSeg PORT MAP(s7);
  e4(0) <= not e4(0) after 100 ns;
  e4(1) <= not e4(1) after 200 ns;
  e4(2) <= not e4(2) after 400 ns;
  e4(3) <= not e4(3) after 800 ns;
END;

```

donnera un rapport : (\*\* : segment allumé, -- segment éteint) pas très facile à lire ...

```

0 ns
:  --
:  |  |
:  |  |
:  --
:  |  |
:  |  |
:  --
:  0 ns
:  **
:  *  *

```

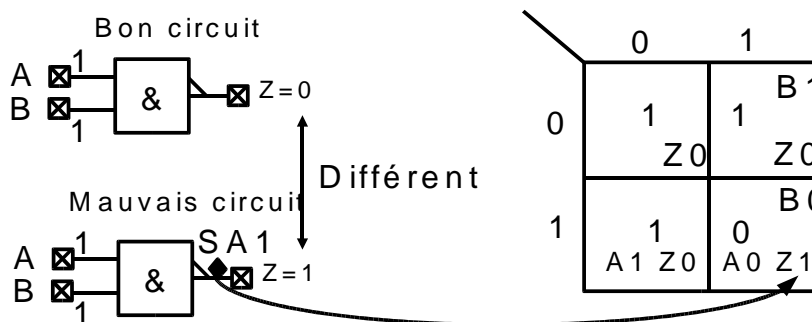
```
: * *
:  --
: * *
: * *
: **
: 100 ns
:  --
: | *
: | *
:  --
: | *
: | *
:  --
: **
: | *
: | *
: **
: * |
: * |
: **
: 300 ns
: **
: | *
: | *
: **
: | *
: | *
: **
: 400 ns
:  --
: * *
: * *
: **
: | *
: | *
:  --
: 500 ns
: **
: * |
: * |
: **
: | *
: | *
: **
: 600 ns
: **
: * |
: * |
: **
: * *
: * *
: **
: 700 ns
```

```
.....
KERNEL: stopped at time: 1600 ns
```

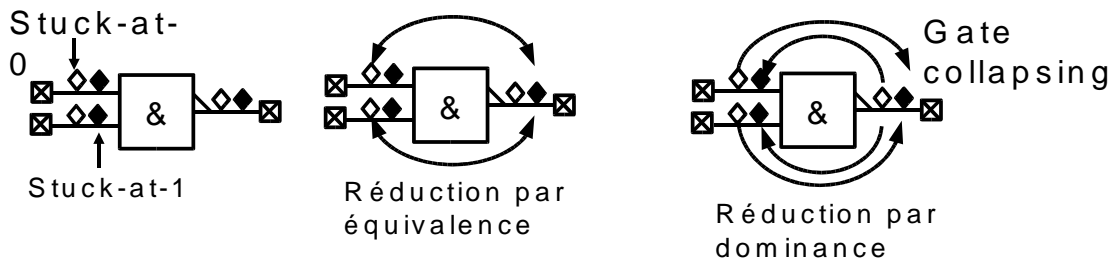
## TD6 A2I13 : Initiation aux tests

### I) Modélisation des fautes

Une faute est à l'origine d'un mauvais fonctionnement d'un circuit. Elle peut être due physiquement à plusieurs raisons : poussière qui fait court-circuit, piste coupée .... Il est indispensable de modéliser les fautes. Un modèle courant est la faute unique de collage (Single Stuck-at Fault : SSF) qui consiste essentiellement à modéliser toutes les fautes par un collage à 1 (stuck-at 1 - SA1 ou s@1) ou collage à 0 (stuck-at 0 - SA0 ou s@0). Ce modèle est très pauvre : il ne permet pas de modéliser tous les problèmes de circuiterie, mais il marche.



En poursuivant comme montré ci-dessus, on peut remplir le tableau de Karnaugh complètement où A0 désigne SA0, B1 désigne SB1... On dit que Z0 domine A1 et B1 car partout où est A1 il ya Z0 (mais pas le contraire). On dit aussi que les fautes A0, B0 et Z1 sont équivalentes.

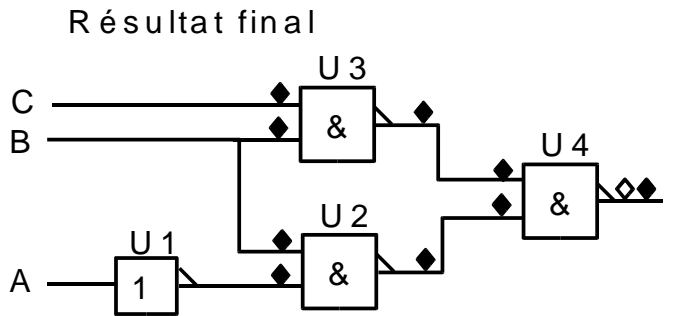
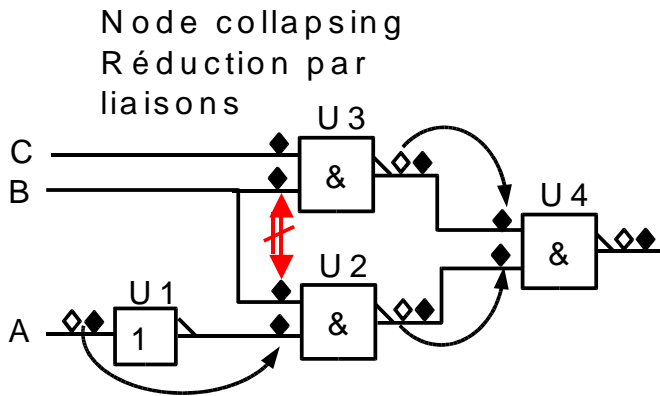
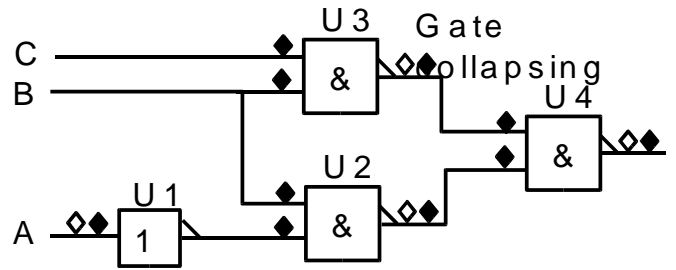
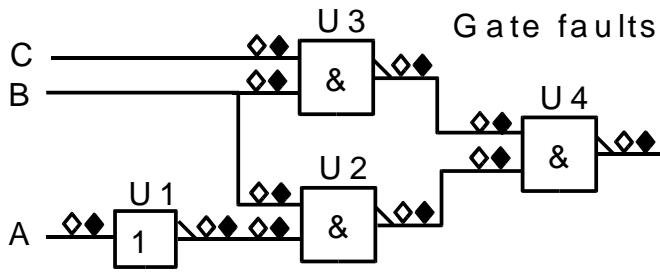


Une porte à deux entrées comporte à priori six fautes comme indiquées ci-dessus. En utilisant la réduction par équivalence on peut ramener ce nombre à 4 fautes (SA1, SB1, SZ0 et SZ1). En utilisant la réduction par dominance on peut réduire ce nombre à 3 fautes (SA0, SB0 et SZ0). L'utilisation de la réduction par dominance peut faire perdre l'information de localisation des fautes.

### Exercice 1

Draw up tables to show how input and output faults collapse using gate collapsing for the primitive logic gates : AND, OR, NAND, NOR and EXOR (assume two-input logic cells in each case with inputs A, B and output F); a two-input MUX (inputs S0,S1, and SEL0; output F).

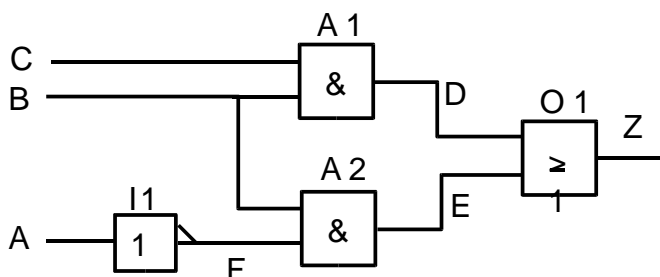
Application aux circuits complexes :



Une faute de liaison (net fault or node fault) force toutes les cellules logiques reliées à cette liaison à un 1 logique ou à un 0 logique. Si l'on regroupe toutes les fautes équivalentes on obtient une classe d'équivalence et pour gagner du temps on ne teste qu'une seule faute de cette classe que l'on appelle première faute (prime fault)

**Exercice 2**

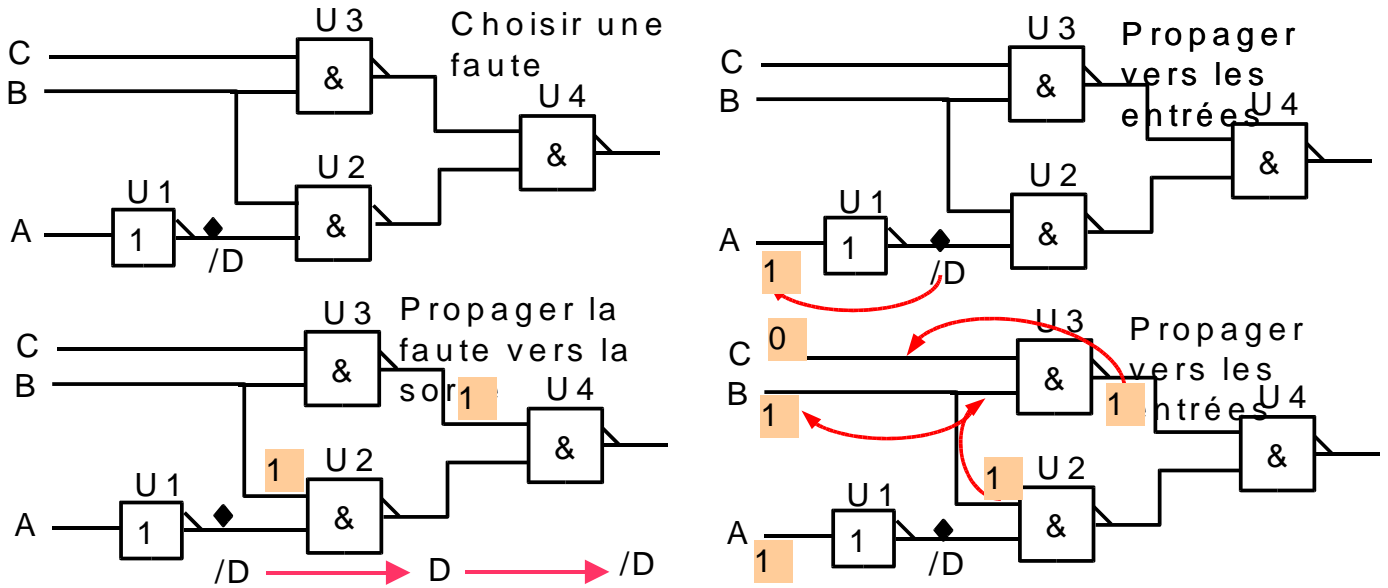
- a) List all the possible stuck-at faults for the circuit in figure below using node faults.
- b) Find all of the equivalent fault classes using node collapsing.
- c) List the prime faults.
- d) List all possible stuck-at faults using input and output faults (use A1.B and A2.B to distinguish between different inputs and outputs on the same net).
- e) List the fault-equivalence classes using gate collapsing.
- f) List the prime faults.



**II) Le D-calcul**

Un des défis des tests est la propagation des fautes : vous avez une faute enfouie dans une couche d'un circuit et vous voulez la faire gagner une des sorties du

circuit avec une combinaison à trouver en entrée. On introduit pour cela la notation  $D \Leftrightarrow 1$  logique sur le bon circuit et  $0$  logique sur le mauvais circuit et  $/D \Leftrightarrow 0$  logique sur le bon circuit et  $1$  logique sur le mauvais circuit.



**Exercice 3**

En reprenant le schéma de l'exercice 2 donner la combinaison des entrées qui permet de propager  $F=D$  et  $F=/D$ . Donner un exemple de circuit pour lequel la propagation d'une faute vers la sortie est impossible.

14.1 (Faults and nodes, 10 min.) Smith Application-Specific Integrated Circuits

How many faults are there in a circuit with n nodes?

Considering fanout how many collapsed faults are there?

Estimate how many test cycles a fault simulator needs to find these faults.

With a 10 MHz clock, how long is a 100 k-gate test (with your estimates)?

Using a 100 MHz computer, how long does this fault simulation take? (Assume simulation time is four orders of magnitude slower than real time.)

Answer:

Using the SSF model the number of faults that we need to consider in a circuit is given by the following equation,

$$\# \text{ faults} = \# \text{ nodes} \times 2.$$

Using fault collapsing we eliminate approximately half the number of faults we need consider so that

# collapsed faults = #nodes.

Then the number of test cycles that the fault simulator needs to run is as follows:

simulation test cycles = # test vectors  $\times$  # collapsed faults.

For a large test program with 100,000 test vectors and a 100 kgate ASIC with at least 100 knodes, we must simulate 10<sup>10</sup> test cycles. For a 100 MHz clock this is 10<sup>2</sup> seconds in real time. Simulators run at least four orders of magnitude slower than real time so that this fault simulation would require at least 10<sup>6</sup> seconds or over 10 days.

If we wish to record the faulty circuit responses during fault simulation we need

simulator file size = (PIs + POs)  $\times$  # test vectors  $\times$  # collapsed faults.

If our 100 kgate design has 200 package pins the fault simulator file size is 2  $\times$  10<sup>12</sup> bits or 0.25TB.

These calculations over-simplify fault simulation, but they do illustrate the very real problems that demand alternative approaches to conventional simulation.