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

Nous allons présenter dans ce chapitre les rudiments de programmation VHDL. Les descriptions abordées sont souvent qualifiée de RT-level (Niveau Transfert de registres) car elles concernent en général des circuits utilisés autour des registres (addition,....)

Programme VHDL simple

Un programme VHDL est composé d'au moins un couple entité/architecture. Une autre façon de présenter les choses est de dire que le programme minimum contient une entité et une architecture. Le rôle de l'entité est de déclarer quelles sont les entrées et comment elles s'appellent et la même chose pour les sorties. Par exemple si l'on désire décrire une fonction ET :

```
1    ENTITY ET IS
2    PORT(e0,e1 : IN BIT; --2 entrees appelees e0 et e1
3      s : OUT BIT);  -- 1 sortie appelée s
4    END ET;
5
6    ARCHITECTURE aET OF ET IS
7    BEGIN
8      s <= e0 and e1; -- equation de sortie
9    END aET;
```

Programme 1: Un programme simple pour un et logique

Les lignes 1 à 4 déclarent l'entité et les lignes 6 à 9 décrivent l'architecture. Tout programme VHDL contient au moins un couple entité architecture. On verra par la suite qu'il peut en contenir éventuellement plusieurs.

Les styles de programmation VHDL pour le combinatoire

Nous présentons la technique des BIT_VECTOR ainsi que l'ensemble des styles de programmation. 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
0	1	1	0	0	1
1	1	0	1	1	0

```
ENTITY demo IS PORT(
  a : in BIT_VECTOR(3 DOWNTO 0);-- 4 entrées
  s : out BIT_VECTOR(1 DOWNTO 0));
  -- 2 sorties
END demo;
```

(ce qui n'est pas mentionné correspond à 00 en sortie pour les 13 lignes manquantes)

Une table de vérité comporte deux parties : partie gauche appelée partie SI qui doit donner l'ensemble des conditions possibles sur les entrées et une partie droite appelée partie ALORS donnant les valeurs des sorties.

Le plus simple quand on a un cahier des charges sous forme d'une table de vérité est d'utiliser le constructeur VHDL "with select when". Il vous faut à tout prix apprendre à passer de l'un à l'autre sans trop vous poser de questions, c'est à dire comprendre le mécanisme de transformation. On remarquera, par exemple, que la partie SI se trouve à gauche dans la table de vérité, mais à droite dans le programme VHDL. Le programme VHDL en style "with select when" s'écrit :

```
1         ARCHITECTURE mydemo OF demo IS
2         BEGIN
3           WITH a SELECT            --style with select when
4             s <= "11" WHEN "0101", -- premiere ligne
5                  "01" WHEN "0110", -- deuxieme ligne
6                  "10" WHEN "1101", -- troisieme ligne
7                  "00" WHEN OTHERS;
8         END mydemo;
```

Programme 2: Le style with select when

Ce style est adapté aux tables de vérité. En effet tous deux nécessitent des conditions mutuellement exclusives dans la partie SI (pour la table de vérité) et dans la partie WHEN du « WITH SELECT WHEN ». Tous les deux nécessitent une description exclusive de toutes les possibilités, visible avec le « WHEN OTHERS » en ligne 7 du programme 2.

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

```
1         ARCHITECTURE mydemo OF demo IS
2         BEGIN
3             -- style when else
4             s <= "11" WHEN a="0101" ELSE -- premiere ligne
5                  "01" WHEN a="0110" ELSE -- deuxieme ligne
6                  "10" WHEN a="1101" ELSE -- troisieme ligne
7                  "00";
8         END mydemo;
```

Programme 3: Combinatoire en style when else

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

```
1         j<= w when a='1' else
2             x when b='1' else
3             0;
```

les conditions ne sont pas mutuellement exclusives. Qu'adviendra-t-il en effet si a='1' et b='1' arrivent simultanément ? (Réponse : j <= w : c'est le premier qui gagne). On ne pourrait pas utiliser dans ce cas directement une structure "with select when" qui nécessite des conditions absolument exclusives.

Le style case when peut être aussi utilisé (en combinatoire comme en séquentiel, il nécessite un process dans les deux cas) :

```
1        ARCHITECTURE mydemo OF demo IS
2        BEGIN
3          PROCESS(a) BEGIN -- absolument necessaire
4            CASE a is  --style case when
5              WHEN "0101" => s <="11"; -- premiere ligne
6              WHEN "0110" => s <="01"; -- deuxieme ligne
7              WHEN "1101" => s <="10"; -- troisieme ligne
8              WHEN OTHERS => s <="00";
9             END CASE;
10         END PROCESS;
11       END mydemo;
```

Programme 4: Combinatoire avec style case when

Un autre style nécessite un process : if then else :

```
1        ARCHITECTURE mydemo OF demo IS
2        BEGIN
3          PROCESS(a) BEGIN
4              IF a="0101" THEN s <="11"; -- premiere ligne
5                ELSIF a="0110" THEN s <="01"; -- deuxieme ligne
6                ELSIF a= "1101" THEN s <="10"; -- troisieme ligne
7              ELSE s <="00";
8             END IF;
9          END PROCESS;
10       END mydemo;
```

Programme 5: Combinatoire avec style if then else

Conseil : utiliser le style with select when pour programmer du combinatoire si vous ne voulez pas établir d'équations. Si vous disposez des équations utiliser-les en vous rappelant que VHDL n'a pas de priorités du ET sur le OU. Il vous faudra donc des parenthèses.

Exercice 1

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

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 des deux lignes suivantes

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

Programme 6: déclaration et utilisation d'une librairie ieee

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 :

```
4        'U'  Uninitialised        'Z'  High Impedance
5        'X'  Forcing Unknow       'W'  Weak Unknow
6        '0'  Forcing 0                  'L'  Weak 0
7        '1'  Forcing 1                  'H'  Weak 1
8        '-' -- Don't Care
```

On dispose de plus des fonctions rising_edge et falling_edge pour la détection de fronts montants et descendants (au lieu des if clk'event...).

Indication : il vous faudra écrire les deux lignes d'utilisation de la librairie devant chaque entités si vous en avez plusieurs. Si vous avez un programme avec quatre entités, il vous faudra écrire quatre fois ces lignes.

Exercice 2
Écrire le programme de l'exercice 1 en utilisant la librairie IEEE.

Programme comportant plusieurs composants

Il existe plusieurs façons d'écrire un programme comportant plusieurs composants. Quelque soit la méthode, vous commencez par compter les composants différents et vous en obtenez N. Si vous avez deux composants ET, vous ne le comptez qu'une seule fois. Il vous faudra un couple entité architecture par composant. Par exemple, le schéma ci-dessous comporte N=3 composants (ET, OU, NON). Vous aurez à écrire autant de couples entité - architecture qu'il y a de composants plus un couple entité - architecture pour la description globale. Vous aurez donc N+1 couples.

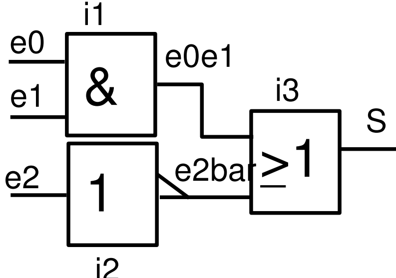


Figure 1: Ensemble de trois composants

Cette programmation s'appelle structurelle et elle revient à décrire un schéma ; dans la terminologie électronique cela s'appelle aussi une netlist. Une méthode consiste à utiliser un seul fichier.

Utiliser un seul fichier pour mettre plusieurs composants

L'utilisation d'un seul fichier se fait en déclarant des signaux et des composants avant le begin de l'architecture globale. Voici l'exemple de la figure 1

```
1         ENTITY Fct IS – entité globale
2         PORT(e0,e1,e2 : IN BIT;
3           s : OUT BIT);
4         END Fct;
5
6         ARCHITECTURE truc OF Fct IS
7         -- signaux et composants avant le begin de l'architecture
8         SIGNAL e0e1,e2bar : BIT;
9         COMPONENT et
10        PORT(e0,e1 : IN BIT;
11          s : OUT BIT);
12        END COMPONENT;
13        COMPONENT ou
14        PORT(e0,e1 : IN BIT;
15          s : OUT BIT);
16        END COMPONENT;
17        COMPONENT inverseur
18        PORT(e : IN BIT;
```

```
19        s : OUT BIT);
20      END COMPONENT;
21      BEGIN
22        i1:et PORT MAP(e0=>e0,e1=>e1,s=>e0e1);
23        i2:inverseur PORT MAP(e=>e2,s=>e2bar);
24        i3:ou PORT MAP(e0=>e0e1,e1=>e2bar,s=>s);
25      END truc;
26      -- fin de l'architecture globale
27      ENTITY et IS
28      PORT(e0,e1 : IN BIT;
29        s : OUT BIT);
30      END et;
31      ENTITY ou IS
32      PORT(e0,e1 : IN BIT;
33        s : OUT BIT);
34      END ou;
35      ENTITY inverseur IS
36      PORT(e : IN BIT;
37        s : OUT BIT);
38      END inverseur;
39      ARCHITECTURE aet OF et IS
40      BEGIN
41        s<=e0 AND e1;
42      END aet;
43      ARCHITECTURE aou OF ou IS
44      BEGIN
45        s<=e0 OR e1;
46      END aou;
47      ARCHITECTURE ainv OF inverseur IS
48      BEGIN
49        s<= NOT e;
50      END ainv;
```

Utiliser deux fichiers dont un package

Lors du TD2 de ENSL1 (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, on commence par le fichier principal :

```
1       USE work.mesportes.ALL;
2       ENTITY Fct IS
3       PORT(e0,e1,e2 : IN BIT;
4         s : OUT BIT);
5       END Fct;
6
7       ARCHITECTURE truc OF Fct IS
8       SIGNAL e0e1,e2bar : BIT;
9       BEGIN
10        i1:et PORT MAP(e0=>e0,e1=>e1,s=>e0e1);
11        i2:inverseur PORT MAP(e=>e2,s=>e2bar);
12        i3:ou PORT MAP(e0=>e0e1,e1=>e2bar,s=>s);
13      END truc;
```

Programme 7: Fichier principal pour l'entité globale

La gestion des librairies dépend beaucoup de l'environnement dont vous disposez. Chaque fois que vous en utilisez une vous devez vous poser la question de savoir comment votre compilateur sait dans quel fichier il va trouver votre package. Ci-dessus (programme 7), est présenté la façon simple de l'environnement Xilinx : c'est le gestionnaire de projet dans lequel est rentré tous les fichiers sources qui

est capable de trouver où est le package mesportes. Avec l'environnement Warp (Cypress), le gestionnaire de librairie doit connaitre où est le fichier librairie (qui est compilé). L'entête est alors :

```
1         LIBRARY portes; --portes.vif en WARP. Utiliser
2         --library manager pour dire où est ce fichier
3         USE portes.mesportes.ALL;
```

Et voici en condensé comment on réalise un package. La partie package déclare les composants (component) et le corps du fichier va comporter les entités et architectures des composants. Le fichier définitif aura donc la forme suivante :

```
4         PACKAGE mesportes IS
5         COMPONENT et -- tout cela est visible dans un .ALL
6         PORT(e0,e1 : IN BIT;
7           s : OUT BIT);
8         END COMPONENT;
9         COMPONENT ou
10        PORT(e0,e1 : IN BIT;
11          s : OUT BIT);
12        END COMPONENT;
13        COMPONENT inverseur
14        PORT(e : IN BIT;
15          s : OUT BIT);
16        END COMPONENT;
17        END mesportes; --********** fin du package ***********
18        ENTITY et IS  --********** debut implantation ***********
19        PORT(e0,e1 : IN BIT;
20          s : OUT BIT);
21        END et;
22        ENTITY ou IS
23        PORT(e0,e1 : IN BIT;
24          s : OUT BIT);
25        END ou;
26        ENTITY inverseur IS
27        PORT(e : IN BIT;
28          s : OUT BIT);
29        END inverseur;
30        ARCHITECTURE aet OF et IS
31        BEGIN
32          s<=e0 AND e1;
33        END aet;
34        ARCHITECTURE aou OF ou IS
35        BEGIN
36          s<=e0 OR e1;
37        END aou;
38        ARCHITECTURE ainv OF inverseur IS
39        BEGIN
40          s<= NOT e;
41        END ainv;
```

Faire de l'électronique à l'ancienne, c'est assembler des fonctions toutes faites pour en composer de nouvelles et assembler ces nouvelles pour en réaliser des encore plus complexes et ainsi de suite. On voudrait retrouver cette façon de faire avec VHDL. Ce qui fait la difficulté d'utiliser VHDL, c'est qu'il n'y a aucun composant prédéfini : même les ET, OU ... ne sont pas prédéfinis (contrairement à verilog, le concurrent de VHDL, qui définit AND, NAND, OR, et NOR). A noter quand même une initiative VHDL avec LPM (Library of Parameterized Modules) que malheureusement Xilinx n'utilise pas.

Le séquentiel

Le séquentiel dispose lui-aussi de ses propres styles. L'équivalent de la table de vérité (spécification sans équation) est le diagramme d'évolution. Il est possible ici encore d'utiliser soit les équations (de récurrence alors) soit le diagramme d'évolution. Nous présentons les deux styles maintenant. Ils sont caractérisés par la détection d'un front montant sur une horloge.

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

Dans cette section, on ne s'intéresse qu'aux diagrammes d'évolution simples c'est à dire avec des transitions inconditionnelles. Les diagrammes d'évolutions un peu plus complexes sont traités plus loin (chapitre 3).

On établit d'abord un tableau état présent état futur duquel on déduit des équations de récurrences. La méthode est présentée sur un exemple. Nous commençons par le graphe d'évolution (avec ses 4 états et 4 flèches transitions) :

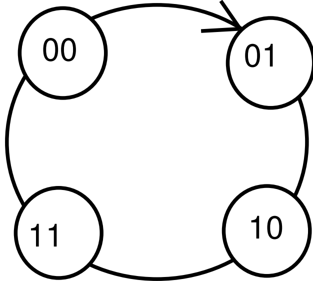


Figure 2: Diagramme d'évolution sur quatre états

Puis le tableau état présent / états futurs ainsi que le tableau de Karnaugh associés sont présentés :

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

Q1+ :

Q1 \ Q0	0	1
0	0	1
1	1	0

Q0+ :

Q1 \ Q0	0	1
0	1	0
1	1	0

Équations de récurrence :

$Q_1^+ = Q_1$ XOR Q_0

$Q_0^+ = /Q_0$

Figure 3: Tableau état présent / état futur et ses équations de récurrences

Cela donne le programme :

```
1          -- compteur premiere version
2          ENTITY cmpt IS PORT (
3          clk: IN BIT;
4          q0,q1: BUFFER BIT); -- BUFFER déconseillé dans un PORT par Xilinx
5          END cmpt;
6          ARCHITECTURE acmpt OF cmpt IS
7          BEGIN
8            PROCESS (clk) BEGIN
9            IF (clk'EVENT AND clk='1') THEN
10               q0 <= NOT q0;
11               q1 <= q0 XOR q1;
12           END IF;
13           END PROCESS;
14         END acmpt;
```

Programme 8: Le séquentiel simple avec équations de récurrences : un compteur

où les deux équations de récurrences se trouvent en ligne 10 et 11 encadrées par une détection de front d'horloge.

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

La programmation d'un diagramme d'évolution se fait directement à partir du tableau état présent / état futur comme le montre le code VHDL ci-dessous (programme 9) :

```
1          -- compteur deuxieme version voir programme 8
2          ENTITY cmpt IS PORT (
3          clock: IN BIT;
4          q : BUFFER BIT_VECTOR(1 DOWNTO 0));
5          END cmpt;
6          ARCHITECTURE mydemo OF cmpt IS
7            BEGIN
8                 PROCESS(clock) BEGIN
9                   IF clock'EVENT AND clock='1' THEN
10                 CASE q IS  --style case when
11                   WHEN "00" => q <="01";
12                   WHEN "01" => q <="10";
13                   WHEN "10" => q <="11";
14                   WHEN OTHERS => q <="00" ;
15                 END CASE;
16                  END IF;
17                END PROCESS;
18         END mydemo;
```

Programme 9: Le séquentiel simple sans équations de récurrences : toujours le même compteur

Xilinx : Do not use buffers when a signal is used internally and as an output port.

Pour éviter les BUFFER dans les PORTS, utiliser un signal interne.

Exercice 3

Réaliser un compteur GRAY sur 3 bits en utilisant ces deux méthodes et sans BUFFER.

TD2 : Séquentiel régulier : compteurs et registres

Le séquentiel consiste toujours à calculer un état futur à partir d'un état présent. Le terme calcul est employé ici de manière très générale, au sens booléen ou au sens arithmétique ou autre. Lorsque ce calcul s'écrit avec un opérateur simple on parlera de **séquentiel régulier**. Parmi les opérateurs simples on trouve l'incrémentation qui donne lieu à des compteurs. Ils permettent de gérer tout ce qui est temporisation et évidemment comptage. Un autre opérateur simple est la concaténation (mise en bout à bout) réalisée en VHDL avec l'opérateur et commercial « & ». Il nous permettra de réaliser les registres à décalage.

Le compteur simple

Il est possible d'utiliser un style case pour programmer un compteur mais cela devient vite fastidieux lorsque le nombre de bits augmente. On multiplie par deux le nombre d'états chaque fois que l'on ajoute un bit.

Exercice 1
Combien d'états comporte un compteur de n bits et donc combien de lignes pour chacun des « case » ?
Application numérique : prendre n=16.

L'idéal serait donc de pouvoir écrire quelque chose du style

```
1        compteur <= compteur + 1;
```

Cela peut se faire en respectant les conditions suivantes :

- utilisation de la librairie IEEE 1164
- utilisation de la librairie IEEE ARITH
- utilisation de la librairie IEEE UNSIGNED
- déclaration de compteur comme std_logic_vector

Avec XILINX cela se fait avec les lignes (devant chacune des entités concernées) :

```
1        library ieee;
2        use ieee.std_logic_1164.all;
3        use ieee.std_logic_arith.all;
4                -- WARP : use work.std_arith.all;
5        use ieee.std_logic_unsigned.all;
```

Programme 10: Utilisations des packages Xilinx (non portables)

Mais cette façon de faire n'est pas portable comme le montre le commentaire. En fait les packages "std_logic_arith" et "std_logic_unsigned" semblent ne pas appartenir à la norme IEEE contrairement à ce que semblerait indiquer leur nom dans le programme 10.

```
1        library ieee;
2        use ieee.std_logic_1164.all;
3        use ieee.numeric_std.all;
```

Programme 11: Utilisations des packages IEEE (portables)

Les trois lignes du programme 11 semblent plus portables mais nécessitent de changer systématiquement la façon de compter. Il faut alors transformer

```
1        compteur <= compteur + 1;
```

en

```
1        compteur <= std_logic_vector(unsigned(compteur) + 1);
```

Exercice 2

Vous disposez d'une horloge rapide et vous voulez réaliser en réaliser une plus lente dont la fréquence est divisée par 32768. Proposez un compteur avec comme entrée h_rapide et comme sortie h_lente (toutes deux sur un bit). Le compteur intermédiaire sera réalisé par un signal.

Possibilité d'utiliser un signal de type integer

Dans ce cas le compilateur peut avoir des problèmes pour trouver le nombre de bits nécessaires pour le compteur. Il faudra donc utiliser une comparaison supplémentaire :

```
1        architecture a_cmpt of cmpt is
2        signal Count : integer :=0;
3          process(Clk) begin
4            if (Clk'event and Clk='0');
5          if (Count=7) then Count <=0; -- sur 3 bits
6          else Count <=Count+1;
7          end if;
8          end process;
9        end a_cmpt;
```

Programme 12: utilisations du type integer dans un compteur

Compteur avec Remise à zéro (raz)

L'entrée raz sur un compteur est une entrée qui permet de mettre la valeur du compteur à 0. Elle peut être synchrone (prise ne compte seulement sur front d'horloge) ou asynchrone. Commençons par l'entité qui nous servira pour nos deux styles de forçage.

```
1        library ieee;
2        use ieee.std_logic_1164.all;
3        use ieee.numeric_std.all;
4        ENTITY Compteur IS
5        PORT (
6          clk,raz :IN std_logic;
7          q : BUFFER std_logic_vector(3 downto 0));
8        END Compteur;
9
```

Programme 13: Entité générale d'un compteur (avec les packages IEEE)

Nous allons présenter tour à tour la méthode synchrone d'abord puis la méthode asynchrone.

Méthode synchrone

La méthode synchrone consiste à réaliser le raz (remise à zéro) dans le front d'horloge. Nous en présentons les éléments essentiels dans le programme ci-dessous.

```
1         PROCESS(clk) BEGIN
2           IF clk'event and clk='1' THEN
3             IF raz='1' THEN
4               q<=(OTHERS=>'0');
5             ELSE
6               q<=std_logic_vector(unsigned(q)+1);
7             END IF;
8           END IF;
9         END PROCESS;
```

Programme 14: Raz synchrone pour un compteur

Remarquez le (OTHERS=>'0') en ligne 4 qui permet de remettre le compteur à zéro quelque soit son nombre de bits. L'écriture q<="0000" nécessite de savoir qu'il y a 4 zéros dans la chaine de caractères. Quant au « IF raz » de la ligne 3 il se trouve bien à l'intérieur du « IF clk'event » de la ligne précédente. Bien sûr ce process est à mettre dans une architecture.

Méthode asynchrone

La méthode asynchrone consiste à réaliser le raz (remise à zéro) en dehors du front d'horloge. Ce signal devient alors prioritaire.

```
1         PROCESS(clk,raz) BEGIN
2           IF raz='1' THEN
3               q<=(OTHERS=>'0');
4           ELSIF clk'event and clk='1' THEN
5             q<=std_logic_vector(unsigned(q)+1);
6           END IF;
7         END PROCESS;
```

Programme 15: Raz asynchrone pour un compteur

Remarquez en ligne 1 que la liste de sensibilité du process comporte maintenant l'entrée raz en plus de l'horloge.

Exercice 3

Réaliser un compteur avec SET et RESET synchrones et asynchrones.

Modifier ce compteur pour qu'il compte jusqu'à 24.

Compteur avec chargement parallèle

Le chargement parallèle est en général asynchrone. L'entité devra posséder les entrées de prépositionnement du compteur. Cela peut se réaliser comme dans le listing suivant pour l'architecture de ce compteur :

```
1        ARCHITECTURE acmpt OF Compteur IS BEGIN
2          PROCESS(clk,load) BEGIN
3            IF load='1' THEN
4              q<=qe;
5              -- ou q<=31; valeur predefinie
6            ELSIF clk'event and clk='1' THEN
7              q<=std_logic_vector(unsigned(q)+1);
8            END IF;
9          END PROCESS;
10       END acmpt;
```

Programme 17: Architecture d'un compteur avec chargement parallèle

Temporisation

L'application la plus courante des compteurs est la temporisation.

Exercice 4

On désire réaliser les deux signaux hsynch et vsynch nécessaire au bon fonctionnement d'un écran VGA. Ils sont caractérisés par les durées suivantes :

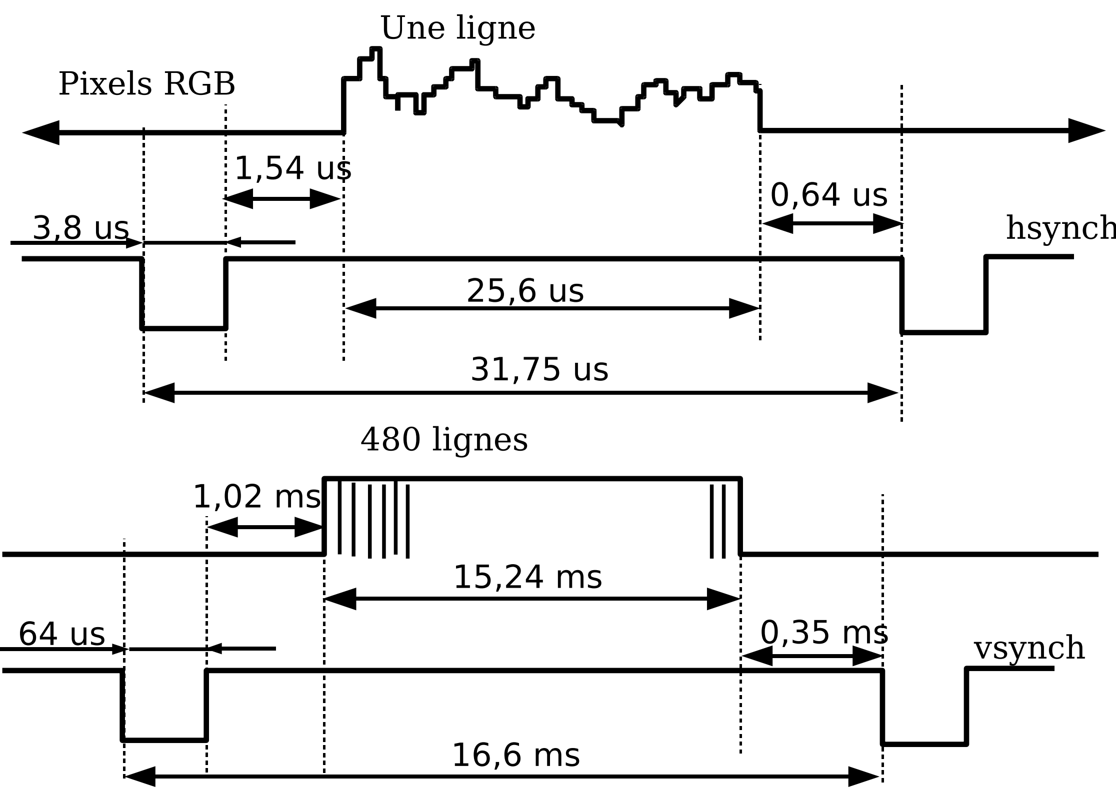


Figure 4: Chronogramme VGA

Techniquement la réalisation est faite de la manière suivante :

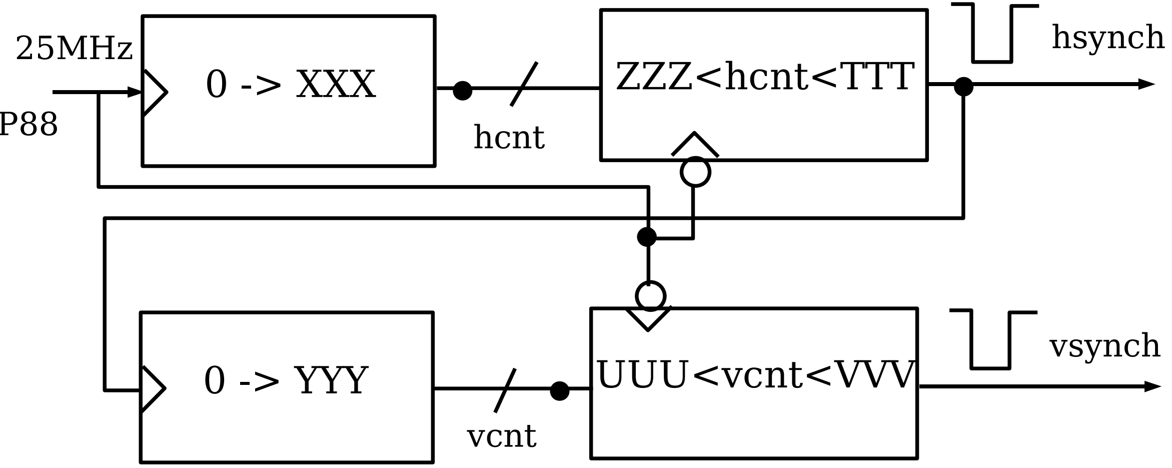


Figure 5: Réalisation matérielle de la synchronisation VGA

On voit apparaître deux compteurs et une partie combinatoire de comparaison qui doit être synchronisée.

1°) Calculer la période de P88.

2°) Le compteur 0 -> XXX commence à compter au début des 25,6 µs. Jusqu'à combien doit-il compter pour réaliser ces 25,6 µs ?

3°) Il lui faut réaliser ensuite 0,64 µs, jusqu'à combien doit-il compter ? Il lui faut réaliser ensuite 3,8 µs, jusqu'à combien doit-il compter ? Il lui faut réaliser ensuite la période complète 31,75 µs, jusqu'à combien doit-il compter ? (C'est la valeur de XXX à un près) On arrondit en général XXX à 799.

Déduire de tout cela la valeur de ZZZ et TTT.

4°) Ce sont les hsynch qui incrémentent le compteur 0->YYY. Quelle est la période correspondante (si l'on prend XXX=799) ?

5°) Combien de temps dure alors la période des 480 lignes avec le résultat de la question 4° (à comparer à 15,24 ms de la spécification VGA).

6°) A l'aide du résultat de 4°) trouver de combien doit compter le compteur pour réaliser le temps de 0,35 ms.

7°) A l'aide du résultat de 4°) trouver de combien doit compter le compteur pour réaliser le temps de 64 µs.

8°) A l'aide du résultat de 4°) trouver de combien doit compter le compteur pour réaliser la période complète de 16,6 ms. Est-il normal d'arrondir à 520 ?

9°) Déduire les valeurs de UUU et VVV

Registre à décalage

L'opérateur de concaténation & est utile pour ce genre de registre :

Le programme 18 décrit un registre à décalage à droite.

```
1         -- registre à décalage
2         library IEEE;
3         use IEEE.STD_LOGIC_1164.ALL;
4         entity ShiftReg is
5           port(clk,entree : in std_logic;
6                q : out std_logic_vector(7 downto 0));
7         end ShiftReg;
8
9         architecture aShiftReg of ShiftReg is
10          signal dataq : std_logic_vector
11                                  (7 downto 0);
12        begin
13          process(clk) begin
14             if clk'event and clk='0' then
15               dataq <= entree& dataq(7 downto 1);
16            end if;
17          end process;
18          process(dataq)begin
19             q<=dataq;
20          end process;
21        end aShiftReg;
22
```

Programme 18: Un registre à décalage complet

Exercice 5 (filtrage de rebonds)

Cet exercice décrit une partie d'un TP que l'on peut trouver en

http://fr.wikiversity.org/wiki/Very_High_Speed_Integrated_Circuit_Hardware_Description_Language

particulièrement au TP2.

L'architecture peut être décrite comme suit : un registre à décalage 4 bits sensible aux fronts descendants de T9, une bascule D qui mémorise l'état de notre sortie et une partie combinatoire qui génère un 1 à l'entrée de la bascule D dès que le registre est rempli par 4 bits à 1 et que sortie vaut 0. Compléter les chronogrammes.

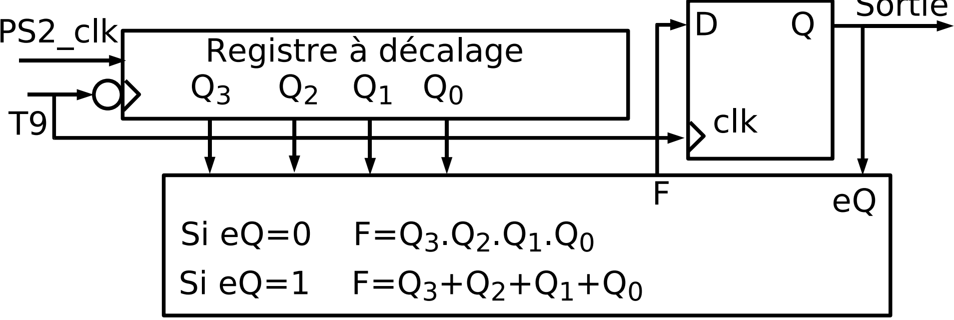


Figure 6: Filtage anti-rebonds

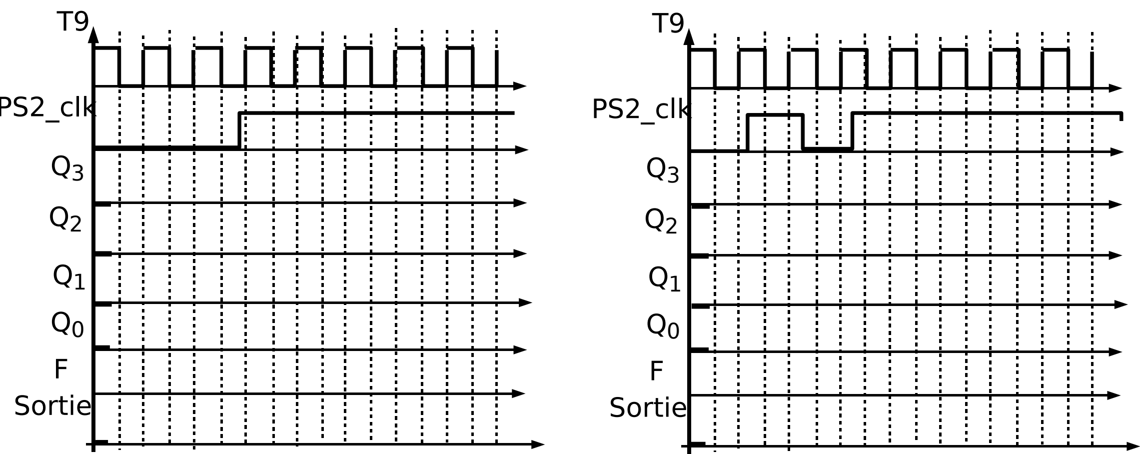


Figure 7: Chronogrammes à compléter

Le principe est le suivant : le registre à décalage échantillonne les entrées PS2_clk. Si la sortie de la bascule D est à zéro, F passera à un quand les 4 valeurs du registres seront à un (autrement dit pas de présence de zéro donc pas de rebond). Le OU pour le F quand eQ est à un a pour objectif de supprimer les rebonds à un car il faut que toutes les entrées Q_i soient à zéro pour que le F passe à zéro.

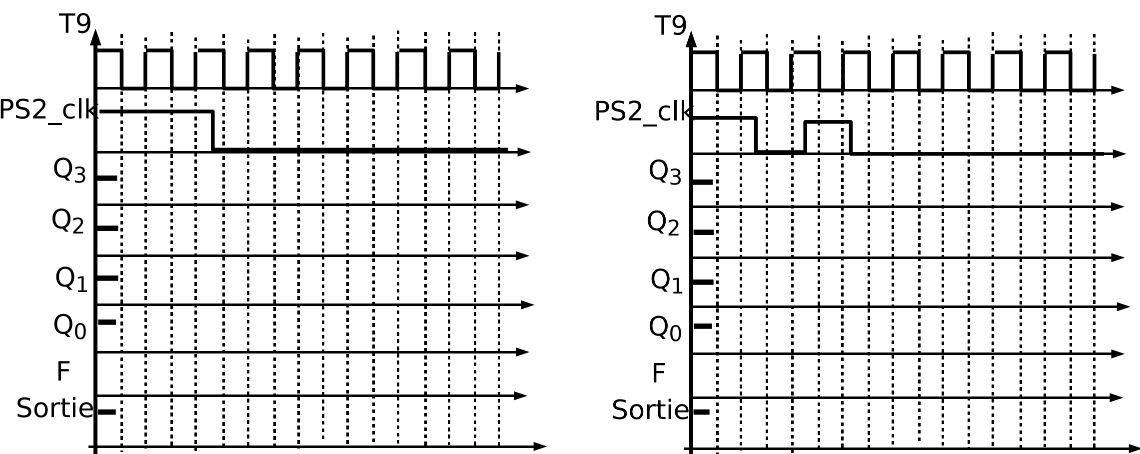


Figure 8: Chronogrammes à compléter (suite)

TD3 : VHDL et logique séquentielle non régulière

Nous allons nous intéresser dans ce chapitre aux machines séquentielles simples et à leur programmation en VHDL. Le style de programmation est assez proche de celui des compteurs au moins en ce qui concerne les initialisations synchrones ou asynchrones.

Un **automate fini** (on dit parfois **machine à états finis** ou encore **machine séquentielle**), en anglais *finite state automaton* ou *finite state machine* (FSA, FSM), est une machine abstraite utilisée pour spécifier la logique séquentielle.

Programmation de graphes d'états et graphes d'évolutions

Dans toute la suite de ce document, on désignera par graphe d'évolution un ensemble d'états notés par des ronds et un ensemble de flèches reliant ces états. Ces flèches seront maintenant indicées par des conditions sur les entrées. C'est ce qui différencie les exemples du premier chapitre (Figure 2 et Programme 9) avec ce que l'on se propose d'expliquer dans ce chapitre. Un graphe d'états sera un peu similaire sauf que les transitions seront réceptives : cela se note avec un trait qui barre la transition et la signification de ce trait est donnée dans le figure 9.

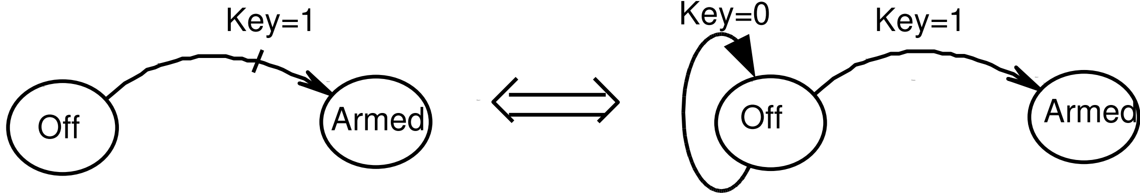
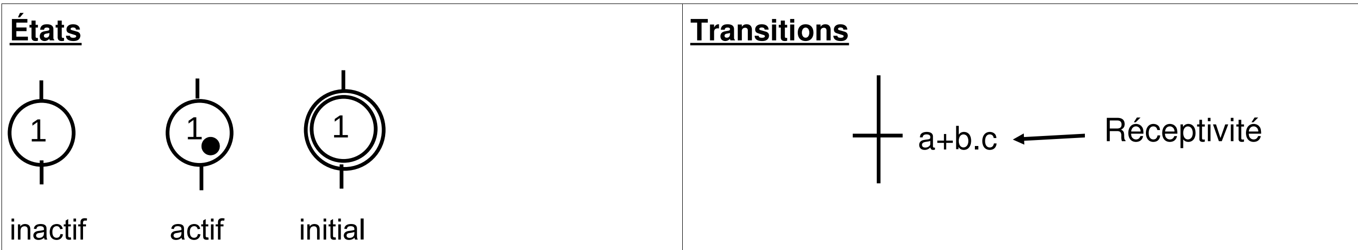


Figure 9: Graphe d'état et son équivalent en graphe d'évolution

Un graphe d'état est une suite d'états et de transitions réceptives.



Pour bien comprendre la différence entre les deux types de graphe nous allons donner un exemple. Commençons par le graphe d'évolution : nous avons choisi un graphe de trois états avec la représentation utilisée par certains logiciels qui permettent en plus la déclaration des entrées et des sorties. A noter aussi comment sont spécifiées les sorties dans ce graphe d'état (ici la sortie unique s'appelle « s »). Le graphe d'évolution est donc plus sophistiqué que celui de la figure 2 et le programme qui en résultera aussi.

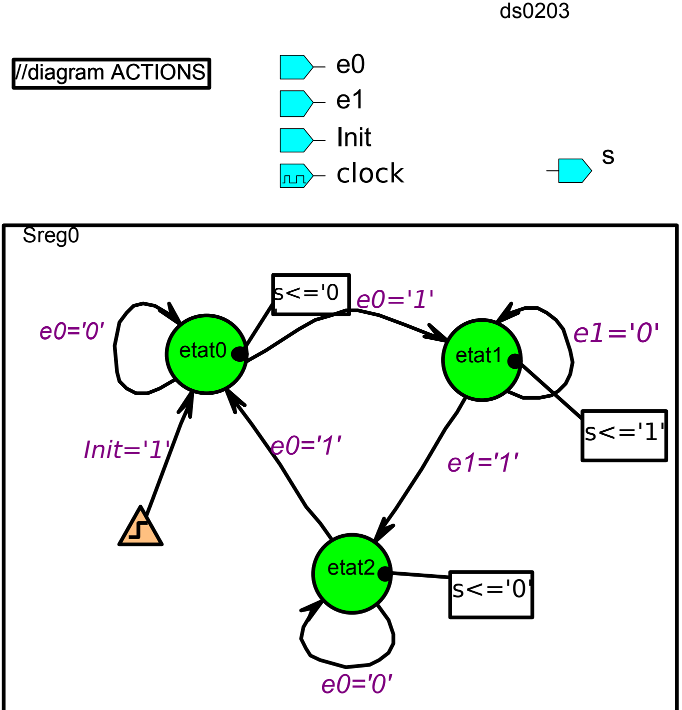


Figure 10: Exemple de diagramme d'évolution

Comme graphe d'état, nous choisissons un exemple qui sera détaillé plus loin :

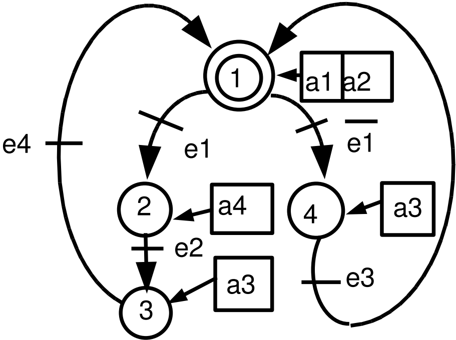


Figure 11: Exemple de graphe d'état

Dans cet exemple, nous avons quatre états, trois entrées e1, e2 et e3 et quatre sorties a1, a2, a3 et a4.

Les graphes d'évolutions et le style « case when »

Présentons d'abord la technique de programmation avec et sans initialisation. Commençons par la programmation sans initialisation.
Le principe consiste à déclarer d'abord un type énuméré avec une définition symbolique de chacun des états (ici Armed, Off, Ringing) :

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

Programme 19: Définition symbolique des états et du signal correspondant

Ensuite dans un « case when » on détaillera toutes les transitions possibles comme montré ci-dessous dans le cas où l'on ne s'intéresse pas à une initialisation :

```
1         -- sans initialisation
2         BEGIN
3           PROCESS (clock) BEGIN
4             IF clock'EVENT AND clock='1' THEN
5               CASE etat IS
6               WHEN Off => IF key ='1' THEN etat <= Armed;
7                           ELSE etat <= Off;
8                           END IF;
9                           ....
10              END CASE;
11            END IF;
12          END PROCESS;
13          ....
```

Programme 20: Utilisation du style case dans une programmation de transitions

L'idée générale est donc d'utiliser un « case » sur les états avec des « if » pour gérer l'ensemble des transitions.
L'ajout d'une initialisation synchrone se fait dans le "if clk'event" comme indiqué ci-dessous :

```
1         -- avec initialisation synchrone
2         BEGIN
3           PROCESS (clock) BEGIN
4             IF clock'EVENT AND clock='1' THEN
5               IF Init='1' THEN etat  <=Off; --initialisation synchrone
6               ELSE
7                 CASE etat IS
8                 WHEN Off => IF key ='1' THEN etat <= Armed;
9                             ELSE etat <= Off;
10                            END IF;
11                            ....
12                END CASE;
13              END IF;
14            END IF;
15          END PROCESS;
16        ....
```

Programme 21: Utilisation du style case et d'une initialisation synchrone

Comme pour les compteurs, l'initialisation asynchrone se fait elle, avant le "if clk'event" sans oublier d'ajouter l'entrée d'initialisation (ici reset) dans la liste des sensibilités du « process » correspondant.

```
1         -- avec initialisation asynchrone
2         BEGIN
3           PROCESS (clock,reset) BEGIN
4             IF Init='1' THEN etat  <=Off; --initialisation asynchrone
5             ELSIF clock'EVENT AND clock='1' THEN
6                 CASE etat IS
7                 WHEN Off => IF key ='1' THEN etat <= Armed;
8                             ELSE etat <= Off;
9                             END IF;
10                            ....
11                END CASE;
12            END IF;
13          END PROCESS;
14        ....
15
```

Programme 22: Utilisation du style case et d'une initialisation asynchrone

Notez en ligne 5 l'utilisation d'un "elsif" en lieu et place d'un "else if" dont l'intérêt est d'économiser un "end if".

Le codage des états

La programmation des états nécessite une déclaration symbolique comme on peut voir ci-dessous :

```
1         TYPE typetat IS (Armed, Off, Ringing); -- dans architecture
2         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.

```
1         type state is (s0,s1,s2,s3);
2         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.

```
1         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 :

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

et

```
1         type s is (s0,s1,s2,s3);
2         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.

La connaissance du codage des états permet de trouver les équations de récurrences.

Programmation des graphes d'états par équations de récurrences

Après avoir examiné la programmation des graphes d'évolution avec un style « case when » nous allons nous intéresser dans cette section à la programmation des graphes d'états par les équations de récurrences. Cette façon de présenter les choses pourrait laisser penser que le style de programmation VHDL est lié au type de graphe que l'on utilise. Ceci est naturellement faux, puisqu'il est facile de passer d'un graphe d évolution à un graphe d'états et inversement, comme on l'a montré en tout début de ce chapitre avec la figure 9. D'autre part, dans le chapitre I, la section Le séquentiel nous a montré comment on pouvait utiliser les deux styles pour un graphe d'évolution simple.

Nous allons donc commencer par rappeler comment on passe d'un graphe d'états à des équations de récurrences.

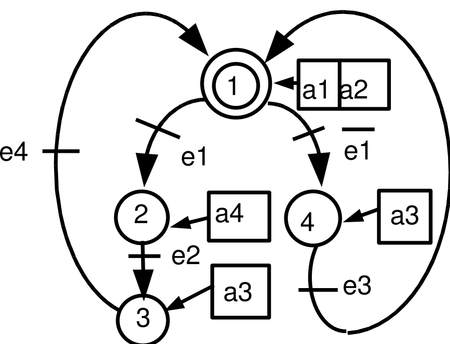


Figure 12: Notre graphe d'état d'exemple

On cherche pour chacun des états i les conditions d'activations AC_i et les déactivations D_i puis on écrit :

$$x_i^+ = AC_i + \overline{D_i} \cdot x_i$$

pour chacun des états. Pour notre exemple cela donne :

AC1 = x3.e4+x4.e3	D1 = e1+/e1=1
AC2 = x1.e1	D2 = e2
AC3 = x2.e2	D3 = e4
AC4 = x1./e1	D4 = e3

et donc comme équations :

Équations de récurrences	Équations de sorties
$x1^+$ = x3.e4+x4.e3 $x2^+$ = (x1.e1+x2./e2) $x3^+$ = (x2.e2+x3./e4) $x4^+$ = (x1./e1+x4./e3)	a1 = x1 a2 = x1 a3 = x3 + x4 a4 = x2

On a ajouté pour être complet les équations de sorties.

Les forçages synchrones et asynchrones avec les équations de récurrences

Tout graphe d'état ou graphe d'évolution peut donc être transformé en un ensemble d'équations de récurrences. En général cette transformation prend en compte le codage des états que l'on vient d'évoquer.

On retrouve les deux façons pour initialiser, celle qui utilise les fronts d'horloge (synchrone) et celle qui ne les utilise pas (asynchrone).

Initialisation asynchrone à partir des équations de récurrences

L'asynchrone ressemble énormément à ce que l'on a déjà présenté et ne nécessite pas d'explications supplémentaires.

```
1        -- gestion de l'asynchrone
2        process(clk,reset) begin
3          if reset='1' then
4          -- avant horloge donc asynchrone
5          q<="0000"; -- ou q <=(OTHERS =>'0');
6          elsif clk'event and clk='1' then
7          -- ici le synchrone : équations de récurrences
8          end if;
9        end process;
```

Programme 23: Initialisation asynchrone et équations de récurrences

Remarquons que le signal q est un std_logic_vector initialisé à "0000" (ce qui n'est pas ordinaire lorsqu'on utilise des équations de récurrences). Il comporte donc quatre bits et il y aura donc quatre équations de récurrences pour q(0), q(1), q(2) et q(3).

Initialisation synchrone à partir des équations de récurrences

La gestion d'une initialisation synchrone peut se faire en suivant deux styles différents présentés en programmes 24 et 26. Commençons par le changement des équations de récurrences. Cela consiste à écrire

$$x_i^+ = AC_i + \overline{D_i} \cdot x_i + Init$$

pour l'équation concernant l'état initial et par écrire

$$x_i^+ = (AC_i + \overline{D_i} \cdot x_i) \cdot \overline{Init}$$

pour les états non initiaux. On prend en compte une entrée appelée « Init ». Pour l'exemple qui nous intéresse (Figure 12):

Équations de récurrences	Équations de sorties
$x1^+ = x3.e4+x4.e3 + Init$ $x2^+ = (x1.e1+x2./e2)./Init$ $x3^+ = (x2.e2+x3./e4)./Init$ $x4^+ = (x1./e1+x4./e3)./Init$	a1 = x1 a2 = x1 a3 = x3 + x4 a4 = x2

```
1        -- initialisation synchrone
2        process(clk) begin
3          if clk'event and clk='1'then
4          -- equations de récurrence +
5          -- init ou
6          -- equations de récurrence
7          --./init
8          end if;
9        end process;
```

Programme 24: Initialisation synchrone et équations de récurrences

Pour être complet nous allons donner le programme complet du graphe d'états de la Figure 12.

```
1        -- programme VHDL correspondant au graphe d'états précédent
2        ENTITY graf1 IS
3        PORT (I,e1,e2,e3,e4,clk : IN BIT;
4              a1,a2,a3,a4 : OUT BIT);
5        END graf1;
6        ARCHITECTURE agraf1 OF graf1 IS
7        SIGNAL x1,x2,x3,x4,x5 : BIT;
8        BEGIN
9          PROCESS(clk) BEGIN
10           IF (clk'event AND clk='1') THEN
11             x1 <= (x3 AND e4) OR (x4 AND e3) OR I;
12            x2 <= (x1 AND e1 AND NOT I) OR (x2 AND NOT e2 AND NOT I);
13             x3 <= (x2 AND e2 AND NOT I) OR (x3 AND NOT e4 AND NOT I);
14             x4 <= (x1 AND NOT e1 AND NOT I) OR
15                                (x4 AND NOT e3 AND NOT I);
16           END IF;
17         END PROCESS;
18         a1 <= x1;
19         a2 <= x1;
20         a3 <= x3 OR x4;
21         a4 <= x2;
22       END agraf1;
```

Programme 25: Initialisation synchrone et équations de récurrences de la Figure 12

Remarquez comment les équations de sorties sont programmées.

Si l'on fait un codage d'état autre que « one_hot_one », il convient d'utiliser les deux techniques précédentes suivant notre désir d'initialiser le bit correspondant à un ou à zéro. Toute initialisation à un se fera par l'ajout de + init (OU init) à l'équation de récurrence correspondante, tandis que toute initialisation à zéro se fait en multipliant par le complément logique de init.

Le deuxième style utilise une technique déjà utilisée précédemment (Programme 21):

```
1         -- initialisation synchrone
2         process(clk) begin
3         if clk'event and clk='1'then
4           if init ='1' then
5             q<= "0001";
6           else
7             -- ici case ou equations de
8             -- récurrences
9           end if;
10        end if;
11        end process;
```

Programme 26: Initialisation synchrone et équations de récurrences

L'initialisation étant faite à "0001", c'est le bit de poids faible q(0) qui devra gérer l'équation de récurrence de l'état initial.

Exercice 2

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 obtenu le fichier de rapport ci-dessous.

1°) Remplir le tableau état présent état futur ci-contre en choisissant un état futur à $(11)_2$.

2°) Trouver les équations de récurrences correspondantes.

3°) Comparer au fichier rapport. Pourquoi n'y a-t-il pas d'équation combinatoire de sortie ? Où est la sortie ?

État présent s1 s0	Condition e0 e1 Init	État futur $s1^+$ $s0^+$
0 0	0 X 0	
0 0	1 X 0	
0 1	X 0 0	
0 1	X 1 0	
1 0	0 X 0	
1 0	1 X 0	
1 1	X X X	

Une croix sur une entrée indique que l'on n'a pas besoin de connaître sa valeur pour en déduire l'état futur. (SBV=State Bit Vector)

Conclusion, est-ce bien le fichier correspondant ?

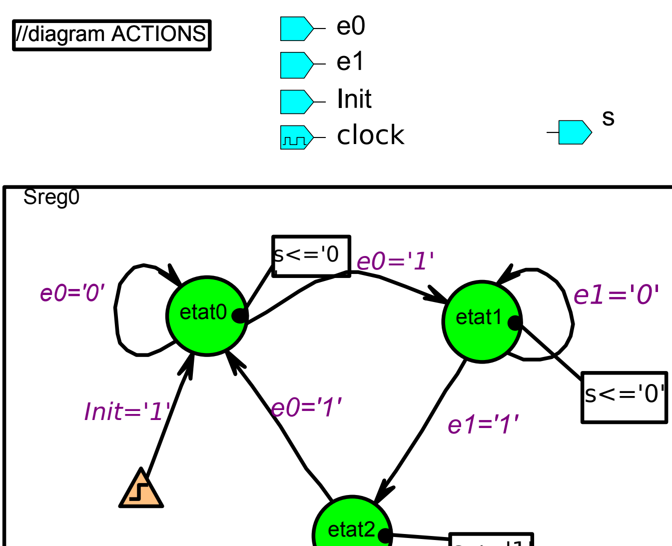


Figure 13: Diagramme d'évolution

```
       | | | | | | |
     _________________
    -|               |-
    -|               |-
    -|               |-
    -|    CYPRESS    |-
    -|               |-
    -|               |-   Warp VHDL Synthesis Compiler:
    -|               |-   Copyright (C) 1991, 1992, 1993,
     |_______________|    Cypress Semiconductor
       | | | | | | |

....
State variable 'sreg0' is represented by a Bit_vector(1 downto 0).
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
DESIGN EQUATIONS            (19:08:48)
    sreg0SBV_0.D =  e0 * /init * /s.Q * /sreg0SBV_0.Q
                  + /e1 * /init * /s.Q * sreg0SBV_0.Q
    s.D = /e0 * /init * s.Q  * /sreg0SBV_0.Q
          + e1 * /init * /s.Q * sreg0SBV_0.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
                 __________________________________________
```

Programmation de GRAFCET

La technique de programmation des GRAFCETs par équations de récurrences et équations de sorties peut être complètement calquée sur celle des graphes d'états. Pour simplifier, un GRAFCET est un graphe d'états dans lequel on remplace les états par des étapes et donc les cercle par des carrés. Une autre différence est qu'il peut y avoir plusieurs jetons dans un GRAFCET donc plusieurs étapes initiales, mais ces détails (très importants) n'ont aucunes conséquences sur notre façon de faire parce qu'on choisit une technique « une équation de récurrence par étape ».

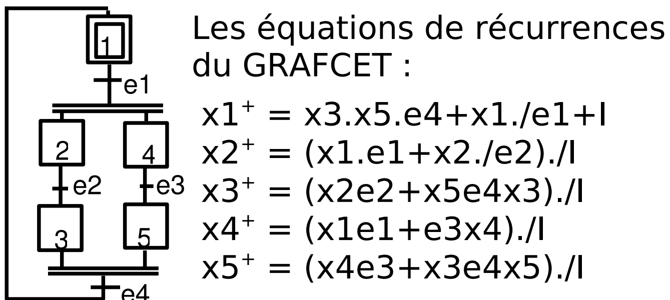


Figure 14: Un GRAFCET et ses équations de récurrences

TD4 : Architectures programmables CPLD et FPGA

Nous allons nous intéresser, dans ce chapitre, à des notions matérielles, c'est à dire aux architectures des composants programmables. Contrairement aux années précédentes, nous abandonnons les architectures simples GALs (appelées parfois SPLD Simple Programmable Logic Device) pour nous intéresser aux architectures complexes CPLD et FPGA.

Architectures complexes : un petit passage vers les CPLDs

Un CPLD peut être défini comme un assemblage de SPLDs. Nous en donnons la structure ci-dessous où chaque bloc logique peut avoir la complexité d'une 22V10 (voir TPs première année).

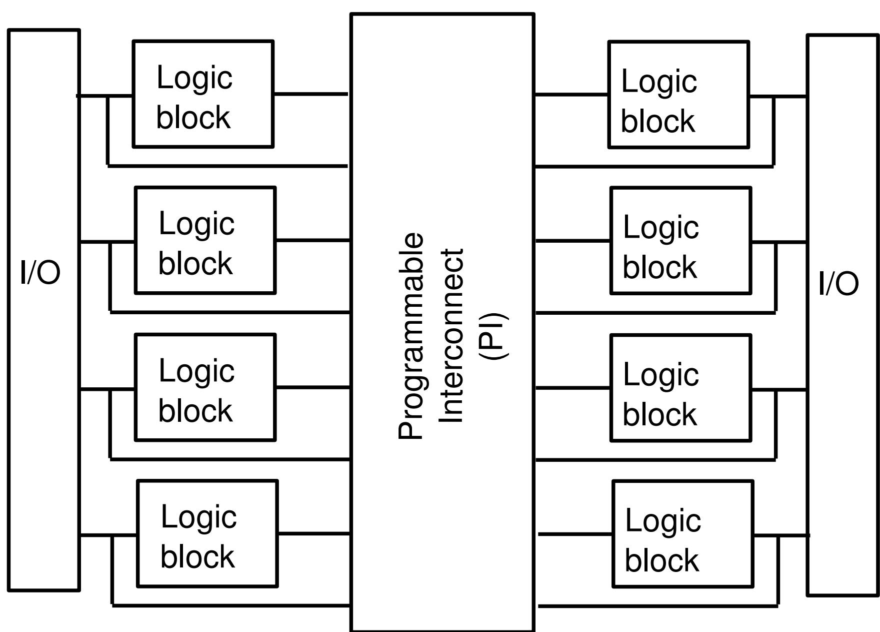
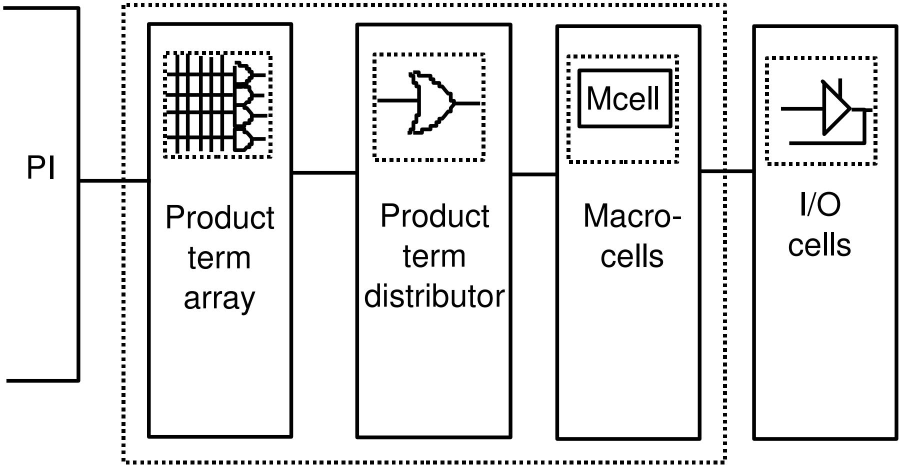


Figure 15: Schéma de principe des CPLD

La partie d'interconnexion (PI) est un ensemble de fils que l'on peut encore représenter par des fils croisés pouvant ou non s'interconnecter par fusibles programmables.

On vous montre sans le détailler l'architecture d'un bloc logique en figure 16. On reconnaît bien les éléments constitutifs d'une PAL.



Bloc logique générique (LAB=Logic Array Block)

Figure 16: CPLD (suite)

Architectures complexes les FPGA

On examinera dans cette partie certains aspects des FPGA mais pas l'architecture détaillée, ce composant étant bien trop complexe pour être complètement détaillé.

1°) Rappel sur les LUTs

Une table de vérité de trois entrées peut être représentée par un nombre 8 bits que l'on convertit en hexadécimal. Soit donc la table de vérité suivante (trois entrées notées e0, e1 et e2, une sortie notée s) :

e2	e1	e0	s	
0	0	0	0	b0
0	0	1	1	b1
0	1	0	1	b2
0	1	1	0	b3
1	0	0	1	b4
1	0	1	0	b5
1	1	0	1	b6
1	1	1	0	b7

Vous pouvez synthétiser la table de vérité à l'aide d'un seul nombre sur 8 bit (poids faible en haut) :

- en binaire ce nombre vaut : 0b01010110
- en hexadécimal ce nombre vaut : 0x56 (noté X"56" en VHDL)

Aucune simplification à faire ici

La valeur hexadécimale 56 (**notée X"56" en VHDL**) est la valeur avec laquelle il faudra initialiser votre LUT avec un composant **lut3**.

Pour 4 entrées (ce qui est notre cas), on utilise une **lut4** avec 4 chiffres hexadécimaux.

2°) Utiliser des LUTs en VHDL

Un exemple est donné maintenant :

```
1	library IEEE; -- transodeur binaire -> 7 segments
2	use IEEE.STD_LOGIC_1164.ALL;
3	library unisim;
4	use unisim.vcomponents.all;
5	ENTITY transcodeur IS PORT(
6	  e : in STD_LOGIC_VECTOR(3 DOWNTO 0);    -- 4 entrées
```

```
7          s : out STD_LOGIC_VECTOR(6 DOWNTO 0)); -- 7 sorties
8        END transcodeur;
9        ARCHITECTURE atranscodeur OF transcodeur IS BEGIN
10       i1 : LUT4
11         generic map (INIT => X"EAAA")
12         port map( I0 => e(0),
13                   I1 => e(1),
14                   I2 => e(2),
15                   I3 => e(3),
16                    O => s(0) );
17       .....
```

Cet exemple vous montre comment on câble une **LUT4** en VHDL (port map) et comment on l'initialise (generic map). Le câblage de ce composant est correct mais pas son initialisation puisqu'on vous demande de la calculer plus loin.

Les deux lignes library ... et use ... sont à ajouter avant toute entité qui utilise une LUT en plus bien sûr de "library ieee;".

3°) Exercice 1

Réaliser le schéma correspondant au "port map" de l'exemple ci-dessus dans le composant "**transcodeur**" (deux rectangles, un appelé "**transcodeur**" et un appelé "**LUT4**"). Compléter ce schéma avec d'éventuels pointillés en montrant que sept LUTs seront nécessaires pour réaliser le transcodeur complet.

Réaliser une table de vérité complète du décodage demandé.

En déduire les 7 valeurs hexadécimales d'initialisation des LUTs.

Réaliser l'architecture complète de "**transcodeur**" en VHDL..

4°) Mémoire Block RAM (BRAM) dans un FPGA

Pour éviter tout changement des options de compilation pour qu'ISE infère correctement une mémoire BRAM (Block RAM) nous allons utiliser un composant faisant partie encore une fois (comme les LUTs) des librairies Xilinx. Ce composant s'appelle RAM16X8S. Lisez le tout petit extrait de sa documentation :

Unless they already exist, copy the following two statements and paste them before the entity declaration.

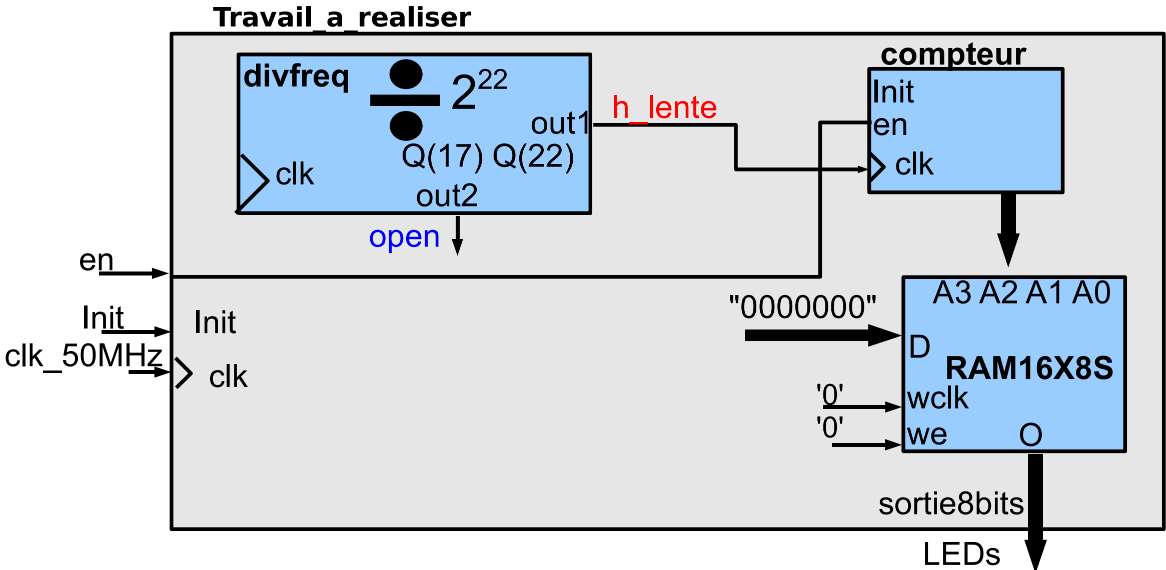
```
1        Library UNISIM;
2        use UNISIM.vcomponents.all;
3        -- RAM16X8S: 16 x 8 posedge write distributed  => LUT RAM
4        --             Virtex-II/II-Pro
5        -- Xilinx HDL Libraries Guide, version 10.1.2
6        RAM16X8S_inst : RAM16X8S
7        generic map (
8        INIT_00 => X"0000", --  INIT for bit 0 of RAM
9        INIT_01 => X"0000", --  INIT for bit 1 of RAM
10       INIT_02 => X"0000", --  INIT for bit 2 of RAM
11       INIT_03 => X"0000", --  INIT for bit 3 of RAM
12       INIT_04 => X"0000", --  INIT for bit 4 of RAM
13       INIT_05 => X"0000", --  INIT for bit 5 of RAM
14       INIT_06 => X"0000", --  INIT for bit 6 of RAM
15       INIT_07 => X"0000") --  INIT for bit 7 of RAM
16       port map (
17       O => O,       -- 8-bit RAM data output
18       A0 => A0,     -- RAM address[0] input
19       A1 => A1,     -- RAM address[1] input
20       A2 => A2,     -- RAM address[2] input
21       A3 => A3,     -- RAM address[3] input
22       D => D,       -- 8-bit RAM data input
23       WCLK => WCLK, -- Write clock input
```

```
24        WE => WE      -- Write enable input
25        );
26        -- End of RAM16X8S_inst instantiation
27
```

Ce que l'on vous demande de faire à ce stade est de **comprendre comment cette RAM est initialisée**.

5°) Exercice 2

1°) Calculer les valeurs prédéfinies à mettre dans la RAM pour réaliser un chenillard aller et retour d'une LED si la sortie de la mémoire est sur 8 leds et si cette mémoire est utilisée à l'aide d'un compteur lent comme indiqué en figure ci-dessous :



2°) Écrire le generic map et le port map complets de le RAM 16X8S

TD5 Embarquer un processeur PicoBlaze.

Description matérielle

Un schéma simple vous fera comprendre ce que l'on cherche à réaliser.

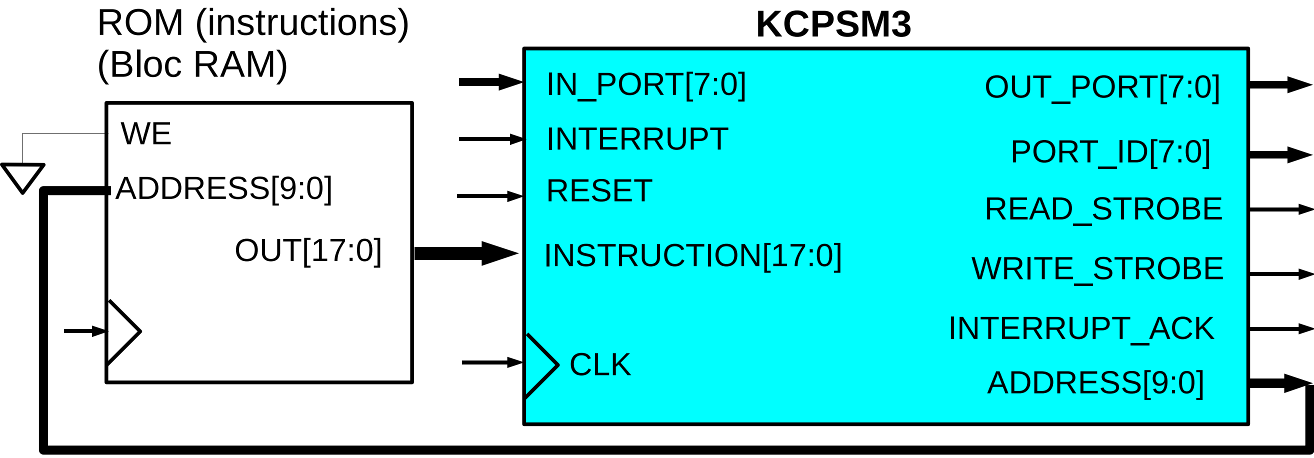


Figure 17: Description matérielle du PicoBlaze

Vous y distinguez deux parties essentielles : une mémoire qui contient le programme à exécuter et un processeur appelé **picoBlaze** avec lequel on va travailler. Le point essentiel est de remarquer qu'il n'y a que deux entrées : une horloge et un reset et aucune sortie. Autrement dit, si vous réalisez ce schéma en VHDL, vous ne verrez pas grand chose se passer sur votre carte FPGA !

Face à ce schéma vous êtes susceptible de vous posez deux questions essentielles :

- comment fais-je pour mettre un programme dans la mémoire ?

- comment fais-je pour entrer et sortir des informations comme allumer des LEDs, lire des interrupteurs enfin toutes les choses classiques que l'on a vu jusqu'ici ?

Le modèle de programmation du PicoBlaze

Il peut être présenté de la manière suivante :

7	s0	0	7	s1	0
7	s2	0	7	s3	0
7	s4	0	7	s5	0
7	s6	0	7	s7	0
7	s8	0	7	s9	0
7	sA	0	7	sB	0
7	sC	0	7	sD	0
7	sE	0	7	sF	0

9	Pointeur de pile SP	0

9	Compteur ordinal PC	0

Z	I	C

C : retenue (Carry) Z : zéro

I : masque d'interruption

La mémoire programme est organisée en 1024 instructions sur 18 bits. L'adressage se fait donc sur 10 bits (2^{10}=1024).

La mémoire donnée est organisée en 64 octets. L'adressage se fait donc sur 6 bits (2^6=64)

Par comparaison, les micro-processeurs 8 bits avaient un espace d'adressage sur 16 bits. Cet espace mélangeait données et programme. Ici on a une séparation, une telle architecture est appelée Harvard.

Quelques instructions

Les modes d'adressage

Le PicoBlaze connaît quatre modes d'adressage. Nous n'en aborderons que deux dans cette section.

- **Adressage registre** : c'est l'adressage le plus simple, il ne peut concerner que des registres.

Op-code	*instruction*	*opérande(s)*	*commentaire*
`0 10 10`	`load`	`s0,s1`	`;charger s1 dans s0`

L'opération qui permet de passer de l'ensemble des instructions et opérandes à l'op-code s'appelle l'assemblage. Le programme qui réalise cette opération est un assembleur. L'opération inverse s'appelle désassemblage.

Remarquez la taille de l'op-code : 18 bits. Ces 18 bits sont composés de 5 caractères hexadécimaux de 4 bits. Ils représentent donc 20 bits. Pour se contenter de 18 bits, le caractère le plus à gauche ne peut dépasser la valeur 3.

- **Adressage immédiat (Immediate)** : l'opérande correspondant se trouve directement dans le programme derrière le code de l'instruction et un premier opérande. Il se fait sur 8 bits.

```
0 00 7F    load s0,7F       ;charger 127 dans s0
1 80 10    add s0,10        ;additionner 16
```

On peut remarquer que les nombres (deuxième opérande) sont systématiquement en hexadécimal.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
LOAD	sX	aaaaaaaa	0	0	0	0	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

xxxx est le code de X sur 4 bits.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ADD	sX	aaaaaaaa	0	1	0	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ADDCY	sX	aaaaaaaa	0	1	1	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

sX <- sX+aaaaaaaa + C (addition avec retenue)

Nous avons déjà abordé deux modes d'adressage en section I.3.1. Nous en présentons deux nouveaux maintenant.

- **Immédiat 10 bits** : l'adresse est donnée sur 10 bits. L'instruction JUMP utilise ce mode d'adressage parce que l'adresse fournie est une adresse absolue de programme.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
JUMP	aaaaaaaaaa		1	1	0	1	0	0	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- **Direct 6 bits** : l'adresse de la mémoire se trouve dans le dernier octet du code de l'instruction (sur les 6 bits de poids faible). Les instructions STORE et FETCH fonctionnent avec ce mode d'adressage et permettent d'adresser 64 octets (seulement) de la mémoire.

```
0601F           FETCH s0,1F     ;charge le contenu de 1F dans s0
18056           ADD s0,56       ;additionne 56 en hexadécimal
34188           JUMP 188        ;saute à l'adresse 188
                                ;(sur 10 bits)
```

A noter les opcodes sur 18 bits (5 caractères hexadécimaux).

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
FETCH	sX	aaaaaa	0	0	0	1	1	0	x	x	x	x	0	0	a	a	a	a	a	a
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
STORE	sX	aaaaaa	1	0	1	1	1	0	x	x	x	x	0	0	a	a	a	a	a	a
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- **Indexé** : l'adresse est calculée par le contenu du registre sX utilisé dans l'adressage. Ce registre est mis entre parenthèses dans ce cas. L'espace mémoire étant limité à 64 octets, seuls les 6 bits de poids faibles sont utilisés dans ce mode d'adressage.

```
     00205      LOAD s2,05       ;boucle 5 fois (s2 compteur de boucle)
0188 07010      FETCH s0,(s1)    ;charge le contenu donné par s1 dans s0
     18300      ADD s3,s0        ;additionne 5F en hexadécimal
     18101      ADD s1,01        ;permet d'aller à l'adresse suivante
                SUB S2,01        ;s2 <- s2-1
     34188      JUMP NZ,188      ;saute à l'adresse 188 (sur 10 bits)
```

Les tableaux

Les tableaux sont les structures de données correspondant à l'adressage indexé. Contrairement à ce qui se passe en langage C, il faut gérer la mémoire des tableaux vous-même.

Mon premier programme en assembleur

Nous avons déjà eu l'occasion de présenter quelques champs à la section 3-1. Nous allons présenter d'autres possibilités maintenant. Regardez le programme ci-dessous :

```
1         CONSTANT m,10 ;commentaire
2         ADDRESS 300
3               LOAD s3,m
4               JUMP suite
5         suite:      LOAD ...
```

On distingue dans ce programme une étiquette , une définition symbolique, un commentaire et la définition de l'origine du programme. L'étiquette est un nom suivi de deux points, ici "suite:". La définition symbolique est m pour la valeur 10 (en hexadécimal). L'origine du programme est en adresse 300 (hexadécimal) ; on utilise la directive ADDRESS.

Comparaison des instructions

Si vous développez autour du PicoBlaze vous serez amené à utiliser plusieurs assembleurs très proches mais pas complètement similaires, celui du KCPSM3 et celui du simulateur PBlazeIDE. A noter que si vous travaillez sous Linux, vous n'aurez pas ce problème : la syntaxe est celle de KCPSM3.

KCPSM3 et kpicosim	***PBlazeIDE***
addcy	addc
subcy	subc
compare	comp
store sX, (sY)	store sX, sY
fetch sX, (sY)	fetch sX, sY
input sX, (sY)	in sX, sY
input Sx,KK	in sX,$KK
ouput sX, (sY)	out sX, sY
return	ret
returni	reti
enable interrupt	eint
disable interrupt	dint

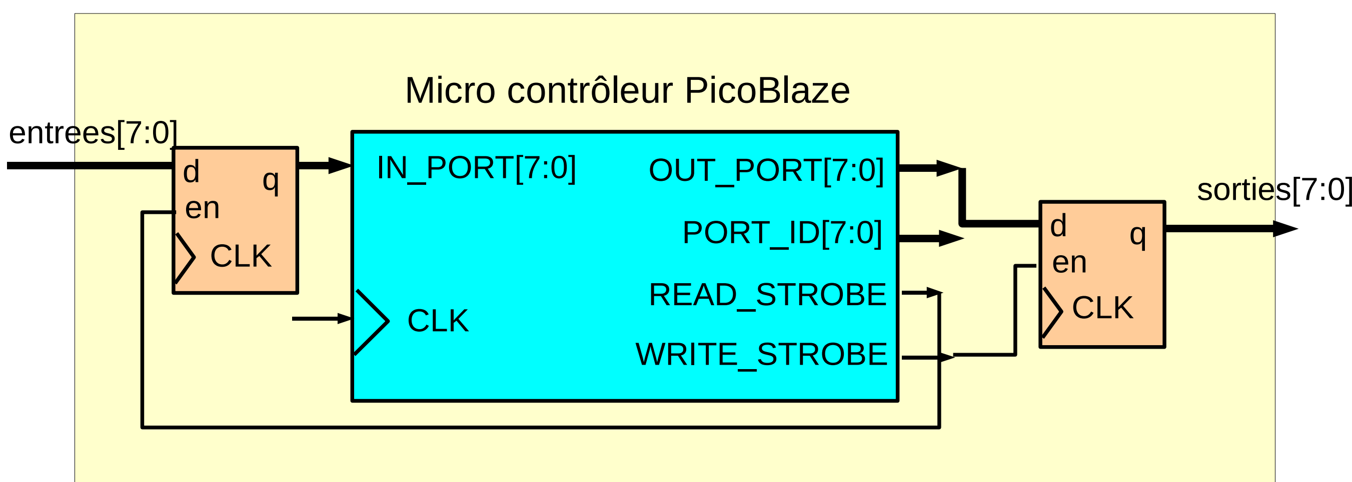
Il existe d'autres différences avec les directives que nous présentons maintenant.

Fonction	***KCPSM3 (et kpicosim)***	***PBlazeIDE***
positionnement de code	address 3FF	org $3FF
constante	constant MAX, 3F	MAX equ $3F
alias de registre	namereg addr, s2	addr equ s2
alias de ports	constant in_port, 00	in_port dsin $00

constant out_port 10	out_port dsout $10
constant bi_port, 0F	bi_port dsio $0F

Exercice 1

On dispose d'un montage qui possède un port en entrée et un port en sortie comme montré en figure ci-dessous. Les entrées sont reliées à des interrupteurs et les sorties à un afficheur sept segments.



1°) Soit le programme ci-dessous

```
1        NAMEREG s0, interrupteurs
2        debut :
3          INPUT interrupteurs,0
4          OUTPUT interrupteurs,0
5          JUMP debut
```

Que nous donne ce programme si segment "a" en PORT0.B7 jusqu'à "g" en PORT0.B1 et le point en PORT0.B0 ? Écrire un programme qui permet d'afficher correctement les valeurs entrées sur 4 bits en affichage hexadécimal. On utilisera l'adressage indexé pour initialiser les mémoires avec des valeurs précalculées (qui ne sont pas les bonnes ici) :

```
1        LOAD s0,01 ; met 1 dans registre s0
2        STORE s0,00  ; met s0 dans la mémoire RAM à l'adresse 00
3        LOAD s0,4F ; met 4F dans registre s0
4        STORE s0,01 ; met s0 dans la mémoire RAM à l'adresse 01
```

puis une boucle infinie :

```
1        boucle:
2          ; ajouter aquisition de nb ici
3          FETCH sortie,(nb) ; conversion
4          OUTPUT sortie,0
5          JUMP boucle
```

2°) Écrire le programme VHDL correspondant à la partie matérielle.

Exercice 2

Le PORT de sortie est maintenant relié à des LEDs. Écrire un programme qui fait un chenillard aller et retour sur une LED en utilisant la technique de l'exercice 1 : initialisation de mémoire puis boucle de lecture. On prendra soin de faire une triple boucle d'attente.

Annexe : les instructions du PicoBlaze

Instructions de contrôle

Les instructions de contrôle peuvent toutes être conditionnelles, dépendant du positionnement ou pas des deux drapeaux du registre d'état, à savoir C et Z.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
JUMP	aaaaaaaaaa		1	1	0	1	0	0	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP Z,	aaaaaaaaaa		1	1	0	1	0	1	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP NZ,	aaaaaaaaaa		1	1	0	1	0	1	0	1	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP C,	aaaaaaaaaa		1	1	0	1	0	1	1	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JUMP NC,	aaaaaaaaaa		1	1	0	1	0	1	1	1	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
CALL	aaaaaaaaaa		1	1	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CALL Z,	aaaaaaaaaa		1	1	0	0	0	1	0	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CALL NZ,	aaaaaaaaaa		1	1	0	0	0	1	0	1	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CALL C,	aaaaaaaaaa		1	1	0	0	0	1	1	0	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CALL NC,	aaaaaaaaaa		1	1	0	0	0	1	1	1	a	a	a	a	a	a	a	a	a	a
	(10 bits)		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
RETURN			1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURN Z			1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURN NZ			1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURN C			1	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

RETURN NC			1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instructions de mémorisations

64 octets de RAM sont accessibles à ces instructions soit en adressage direct, soit en adressage indexé. FETCH va chercher les données tandis-que STORE les stocke.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
FETCH	sX	aaaaaa	0	0	0	1	1	0	x	x	x	x	0	0	a	a	a	a	a	a
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

FETCH	sX	(sY)	0	0	0	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
STORE	sX	aaaaaa	1	0	1	1	1	0	x	x	x	x	0	0	a	a	a	a	a	a
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

STORE	sX	(sY)	1	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
		(6 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instructions arithmétiques

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ADD	sX	aaaaaaaa	0	1	0	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ADD	sX	sY	0	1	0	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ADDCY additionne les deux opérandes avec le drapeau de retenue C.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ADDCY	sX	aaaaaaaa	0	1	1	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ADDCY	sX	sY	0	1	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
SUB	sX	aaaaaaaa	0	1	1	1	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SUB	sX	sY	0	1	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
SUBCY	sX	aaaaaaaa	0	1	1	1	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SUBCY	sX	sY	0	1	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

L'instruction COMPARE réalise une soustraction mais, contrairement à l'instruction SUB, le résultat n'est pas mis à jour. Seuls les bits du regitre d'état sont positionnés.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
COMPARE	sX	aaaaaaaa	0	1	0	1	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

COMPARE	sX	sY	0	1	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instructions logiques

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
LOAD	sX	aaaaaaaa	0	0	0	0	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LOAD	sX	sY	0	0	0	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
AND	sX	aaaaaaaa	0	0	1	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AND	sX	sY	0	0	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
OR	sX	aaaaaaaa	0	0	1	1	0	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR	sX	sY	0	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
XOR	sX	aaaaaaaa	0	0	1	1	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XOR	sX	sY	0	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

L'instruction de test fait un ET logique entre deux opérandes mais, contrairement à l'instruction AND, le résultat final n'est pas mis à jour. Seuls les bits d'état sont mis à jour.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
TEST	sX	aaaaaaaa	0	1	0	0	1	0	x	x	x	x	a	a	a	a	a	a	a	a
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TEST	sX	sY	0	1	0	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instructions d'entrées sorties

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
INPUT	sX	pppppppp	0	0	0	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INPUT	sX	(sY)	0	0	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
OUTPUT	sX	pppppppp	1	0	1	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
		(8 bits)	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUTPUT	sX	(sY)	1	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Instructions d'interruptions

Les instructions qui permettent d'autoriser ou non les interruptions sont :

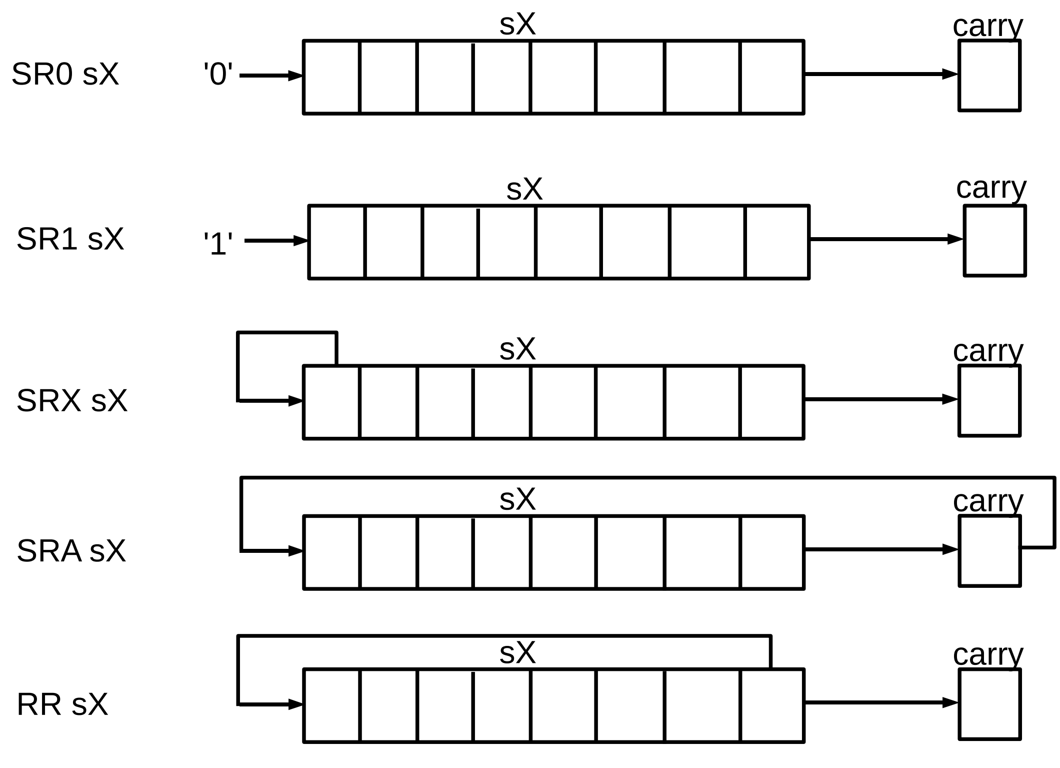
Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
ENABLE	INTERRUPT		1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISABLE	INTERRUPT		1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

L'instruction RETURNI doit être utilisée à chaque fin d'interruption. Elle fonctionne comme RETURN sauf qu'elle restore les bits de retenue et de détection de zéro du registre d'état.

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
RETURNI	ENABLE		1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RETURNI	DISABLE		1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	

Intructions de décalages

Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
SR0	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	1	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SR1	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	1	1
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRX	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	0	1	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRA	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RR	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0



Mnémonique	*Operande1*	*Opérande2*	*Opcode*																	
SL0	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SL1	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	1
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SLX	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	1	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SLA	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RL	sX		1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	0	0
			17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Ancien TD4 VHDL et machines à états algorithmiques

Tout ce qui a été traité jusqu'à maintenant fait partie du niveau de spécification RTL (Register Transfer Level). Le niveau d'abstraction abordé maintenant est appelé ESL (Electronic System level)

Quand la fonction à synthétiser devient complexe, la synthèse doit être décomposée en deux parties : un chemin de données et une machine séquentielle destinée à ordonnancer ce chemin de données. En automatisme le chemin de données est plutôt appelé partie opérative. On utilisera cette terminologie parfois. La synthèse correspondante nécessite alors des raisonnements spécifiques : cette décomposition demande une expérience d'autant plus grande que les parties utilisées se décomposent elles-même 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. Dans ce chapitre nous n'aborderons que superficiellement le sujet, avec des exemples.

Terminologie :

- organigramme de programmation : http://fr.wikipedia.org/wiki/Organigramme_de_programmation

- algorigrammes : représentation d'un algorithme http://troumad.developpez.com/C/algorigrammes/

Utilisations de compteurs (comme chemin de données)

Le compteur de passages est un exemple utilisant des compteurs séquencés. Il s'agit de compter des événements si on a d'abord le capteur gauche puis le capteur droit activés et de décompter si le capteur droit est activé avant le capteur gauche.

En logique traditionnelle, la partie opérative est composé par deux compteurs 74190 et le séquenceur par une machine à états. Le séquenceur peut être réalisé avec des composants programmables simples (SPLD). Donnons-en un schéma de principe :

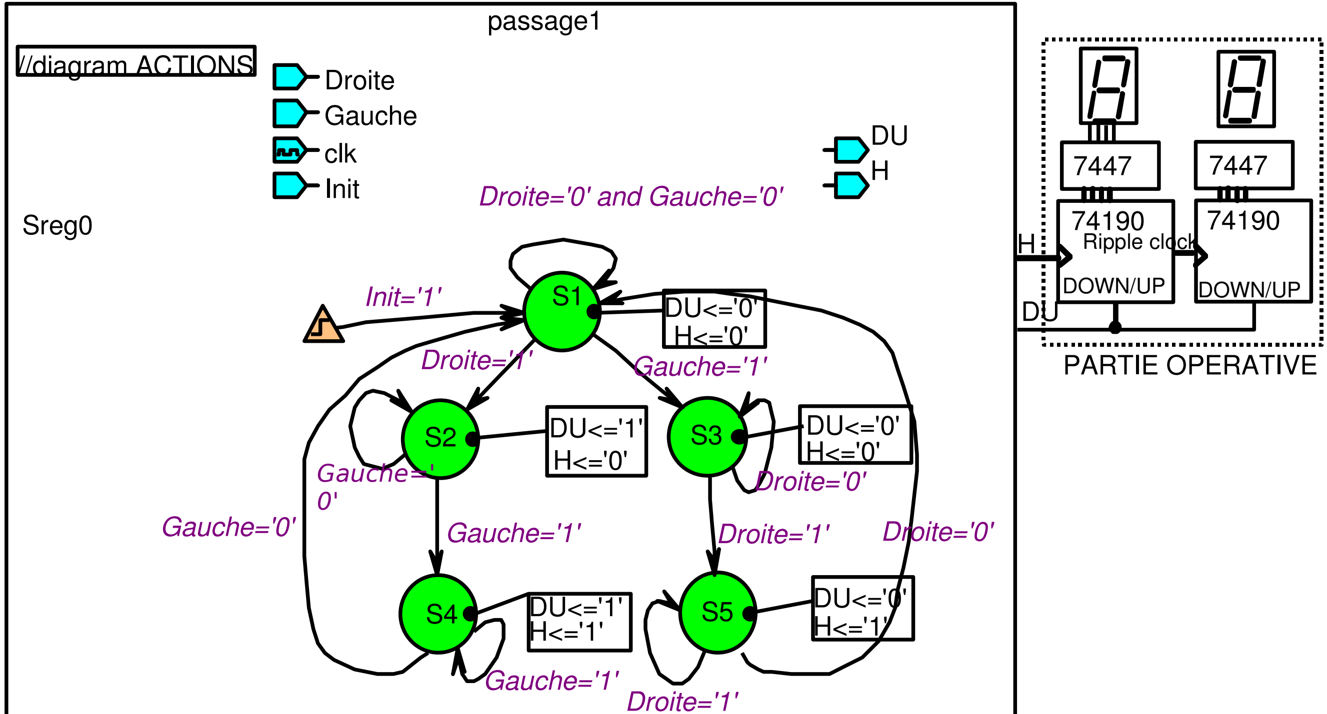


Figure 18: Compteur de passages

Cet exemple n'est pas conforme aux règles d'assemblage du séquentiel. En effet l'horloge H de la partie opérative est réalisée par les actions du séquenceur et donc par du combinatoire ce qui est interdit

lorsqu'on veut faire une synthèse fiable. Nous avons eu l'occasion de réaliser cet ensemble soit avec un PAL et deux 74190, soit dans un seul composant de type FPGA en utilisant un modèle VHDL du 74190 et pour ces deux montages on obtient un aléa de fonctionnement. Nous le présentons donc ici à seul titre d'exemple, qui a l'avantage d'être simple à comprendre.

Nous allons donc nous intéresser au respect des règles d'assemblage du séquentiel et modifier en conséquence l'exemple ci-dessus du compteur de passages.

Mise en conformité du compteur de passages

Le principe de base est de changer la partie chemin de données (les compteurs). On lui ajoute une entrée « en_tick » de validation synchrone. Le fonctionnement des compteurs est alors le même qu'un 74190 tant que en_tick=1 tandis que le compteur reste figé si en_tick=0. La partie séquenceur est alors chargée de réaliser les deux signaux « UD » et « en_tick » mais pas l'horloge. L'horloge sera commune au séquenceur et au chemin de données, mais pour un bon fonctionnement on s'arrangera pour que ces deux parties soient sensibles à un front différent : par exemple front montant pour le séquenceur et front descendant pour le chemin de données.

C'est une bonne chose lorsqu'il s'agit de faire la séparation entre le séquenceur et le chemin de donnée de considérer que le premier fonctionne sur un type de front d'horloge (par exemple montant) et le deuxième sur le front complément (par exemple descendant).

Application

Exercice 1

Modifier le chemin de données ci-dessus pour qu'il utilise la même horloge que la partie séquenceur. Donner le schéma global correspondant en montrant qu'il faudra ajouter deux états à la machine d'états du séquenceur (que se passe-t-il si l'on reste dans S4 ?).

Exercice 2

La gestion de l'écran VGA déjà présenté en figure 5 comporte une horloge asynchrone pour le compteur vertical de 0 à 519. Cette horloge provient d'une comparaison c'est à dire d'un circuit combinatoire ce qui ne respecte pas la règle de base de l'assemblage du séquentiel (pas d'horloge provenant de combinatoire). Comment modifier le montage pour avoir un montage complètement synchrone (même horloge pour les deux compteurs ?

Remarque : la façon de procéder de l'exercice original fonctionne correctement en fait car le combinatoire n'est pas pur mais synchronisé sur front descendant. Cela reste cependant une technique non recommandée.

Utilisation de registres (comme chemin de données)

Nous nous trouvons dans la situation suivante : nous disposons de plusieurs registres qui forment ce que l'on appellera notre chemin des données, et nous désirons synthétiser un circuit qui les utilise. Ce circuit peut être combinatoire ou séquentiel. Nous allons appréhender cela avec un exemple de calcul de PGCD. Son calcul est réalisé à l'aide de l'algorithme d'Euclide. Cet algorithme peut être décrit de manière récursive de la façon suivante :

a et b sont les deux valeurs dont on cherche le PGCD (avec a > b)

si b = 0 alors le PGCD est a autrement on recherche le PGCD de b et du reste de la division de a par b.

Les FSMD (Finite State Machine with Data path)

Nous avons eu l'occasion de présenter les **automates finis** (ou **machines à états finis** ou *finite state machine* FSM). Nous allons généraliser avec les FSMD (Finite State Machine with Data path. La terminologie française associée est assez variée :

- Automates Finis avec chemin de Données (AFD)
- machines à registres

– machines algorithmiques ou automates algorithmiques

Au delà de la terminologie, ce qui nous intéresse est un moyen de décrire le fonctionnement de ces machines. Nous avons appris à spécifier les automates finis à l'aide des graphes d'états (ou parfois des graphes d'évolutions) nous allons présenter maintenant notre manière de spécifier les machines à registres. Cela peut se faire avec des organigrammes dans lesquels on a la possibilité de décrire des transferts ou des opérations entre registres (appelés parfois **organigramme fonctionnel**).

Revenons sur notre exemple de calcul du PGCD. Il peut être décrit par l'organigramme :

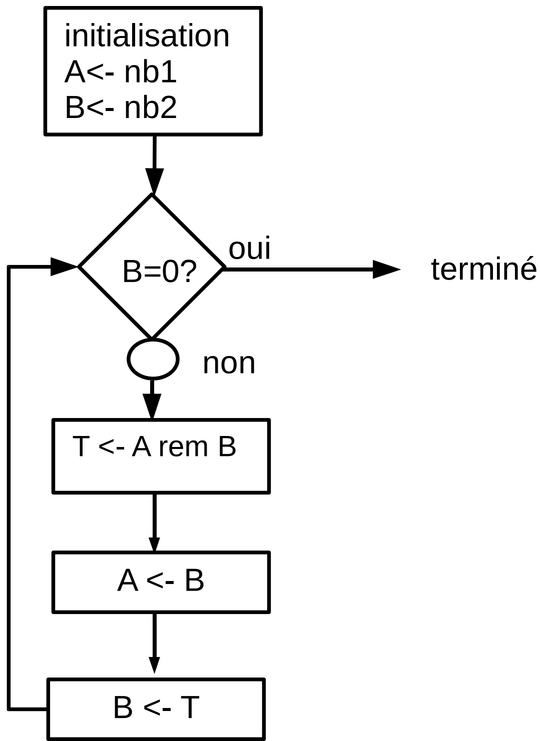


Figure 19: Organigramme fonctionnel du calcul du PGCD

On suppose dans cet organigramme que l'on sait réaliser l'opération T reçoit A rem B, c'est à dire le reste de la division de A par B. Prenons par exemple A=nb1=8 et B=nb2=6.

B étant différent de 0, T <- 2 puis A<-6 et B<-2

B étant différent de 0 ,T <- 0 puis A <-2 et B <- 0

c'est fini et le résultat est dans A soit 2.

Tout cela est bien gentil, mais cela nécessite un composant capable de calculer le reste de la division. Si, comme on va le supposer par la suite, ce n'est pas le cas, il nous faut réaliser en plus cette opération. Cela se fait simplement avec une suite de soustractions jusqu'à obtenir un nombre plus petit que le diviseur. Il nous faut donc modifier l'organigramme en conséquence (voir figure 20)

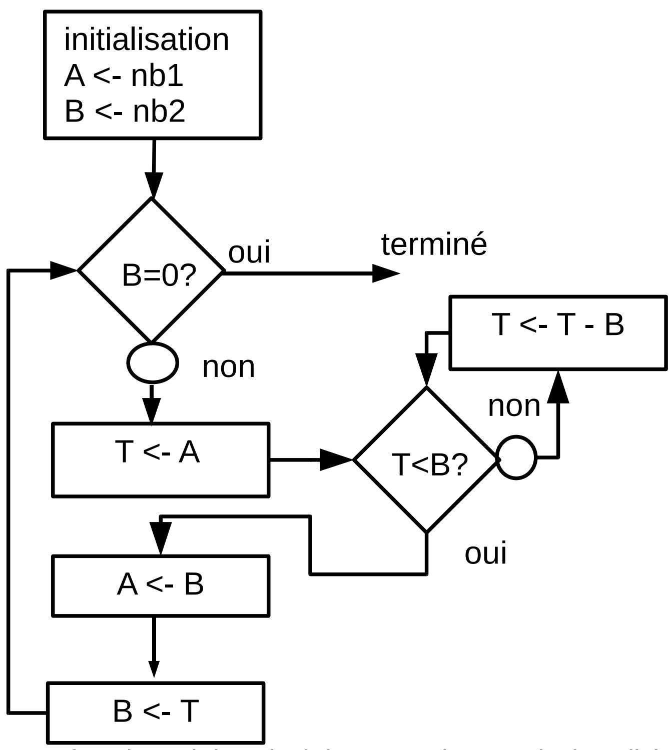
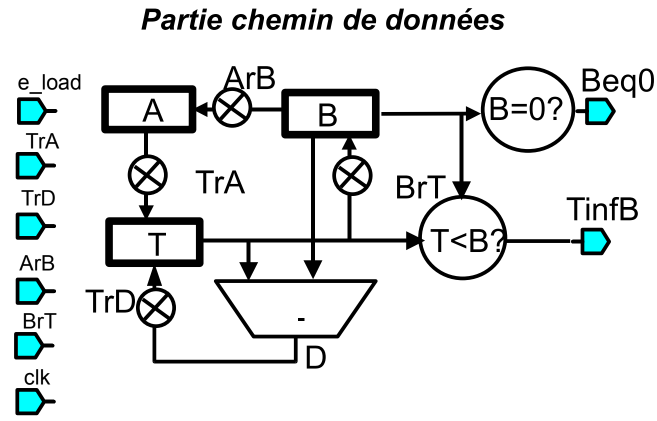


Figure 20: Organigramme fonctionnel du calcul du PGCD (avec calcul explicite du reste)

Une fois vérifié le fonctionnement de notre algorithme, il nous reste à le matérialiser. La partie chemin de données sera responsable de tous les transferts de registres et les calculs combinatoires (en général). Pour notre exemple, nous avons trois registres (A, B et T), quatre transferts de registres (T <- A, T <- A-B, A <- B et B <- T), un calcul d'une différence (A-B) et deux tests B=0? et T<B?. Voilà donc tout ce que doit être capable de faire la partie chemin de données.

La partie chemin de données

La partie chemin de données est présentée maintenant :

Partie chemin de données	***Explications***
 *Figure 21: Chemin de données pour calcul PGCD*	Tous les rectangles représentent des registres 8 bits, il y en a donc trois : A,B et T. Le trapèze représente un soustracteur 8 bits réalisant la différence D=T-B (et non D=B-T !). C'est donc une partie combinatoire. Les deux grands cercles représentent aussi des parties combinatoires destinées à réaliser des comparaisons. Les actions possibles sur cette partie opérative sont représentées par des "boutons" : cercles avec croix. Elles ne représentent que des transferts de registre.

Appuyez sur le bouton ArB et le registre A reçoit le contenu du registre B. On peut donc lire ArB : A reçoit B. Il en est de même pour les trois autres boutons.

Nous allons présenter maintenant le programme VHDL réalisant cette partie. Commençons par l'entité :

```
1         library IEEE;
2         use IEEE.std_logic_1164.all;
```

```
3         use IEEE.std_logic_arith.all;
4         use IEEE.STD_LOGIC_UNSIGNED.all;
5         entity chemin_donnees is
6           port( ArB: in STD_LOGIC;
7                 BrT: in STD_LOGIC;
8                 TrA: in STD_LOGIC;
9                 TrD: in STD_LOGIC;
10          e_load: in STD_LOGIC;
11          e_A: in STD_LOGIC_VECTOR(7 DOWNTO 0);
12          e_B: in STD_LOGIC_VECTOR(7 DOWNTO 0);
13          s_A: out STD_LOGIC_VECTOR(7 DOWNTO 0);
14          s_B: out STD_LOGIC_VECTOR(7 DOWNTO 0);
15          s_T: out STD_LOGIC_VECTOR(7 DOWNTO 0);
16          Beq0: out STD_LOGIC;
17          TinfB: out STD_LOGIC;
18          clk: in STD_LOGIC);
19        end chemin_donnees;
```

Programme 27: Entité du chemin de données pour calcul du PGCD

Le lecteur perspicace aura remarqué la présence de e_A et e_B qui ne sont pas dessinés dans le schéma figure 21. Ces deux entrées sont destinées à mettre initialement des valeurs dans les deux registres A et B.

Intéressons-nous maintenant à l'architecture où l'on retrouve un process par registre et un process par partie combinatoire :

```
1         architecture chemin_donnees_arch of chemin_donnees is
2         -- declaration des registres
3         signal regA,regB,regT : STD_LOGIC_VECTOR(7 DOWNTO 0);
4         begin
5         -- gestion des registres
6         reg_A:process(clk)begin
7           if clk'event and clk='0' then
8             if ArB='1' then
9             regA <= regB;
10            elsif e_load='1' then regA <= e_A;
11          else
12            regA <= regA;
13          end if;
14          end if;
15        end process;
16        reg_B:process(clk)begin
17          if clk'event and clk='0' then
18            if BrT='1' then
19            regB <= regT;
20            elsif e_load='1' then regB <= e_B;
21          else
22            regB <= regB;
23          end if;
24          end if;
25        end process;
26        reg_T:process(clk)begin
27          if clk'event and clk='0' then
28            if TrA='1' then
29            regT <= regA;
30          elsif TrD='1' then
31            regT <= regT - regB;
32          else
33            regT <= regT;
34          end if;
35          end if;
```

```
36        end process;
37        -- partie combinatoire
38        T_inf_B:process(regT,regB)begin
39          if regT < regB then
40            TinfB <='1';
41          else
42            TinfB <='0';
43          end if;
44        end process;
45        B_eq_0:process(regB)begin
46          if regB = "00000000" then
47            Beq0 <='1';
48          else
49            Beq0 <='0';
50          end if;
51        end process;
52          s_A <= regA;
53          s_B <= regB;
54          s_T <= regT;
55        end chemin_donnees_arch;
```

Programme 28: Architecture du chemin de données pour calcul du PGCD

La partie séquenceur

La machine d'états doit réaliser le séquencement des opérations pour que le calcul se déroule correctement. La spécification de celui-ci se fait facilement à partir de l'organigramme figure 20. Donnons-la à l'aide d'un diagramme d'états :

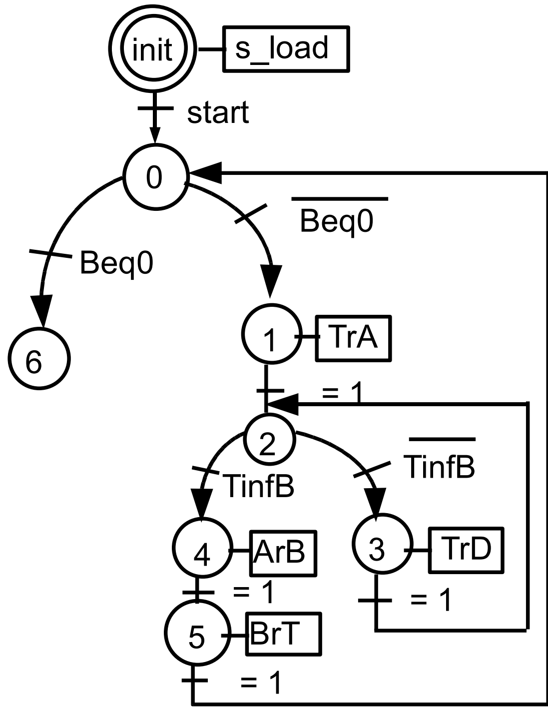


Figure 22: Diagramme d'états du séquenceur pour le calcul du PGCD (avec calcul explicite du reste)

Le calcul est terminé lorsqu'on arrive dans l'état 6. Pour simplifier on a omis le retour de l'état 6 vers l'état « init »

<u>Exercice 3</u> (DS parcours AUE Nov. 2008 - extraits)

On initialise le registre A à 8 et B à 6. Un « start » nous fait passer dans l'état 0 puis dans l'état 1.

Compléter le tableau ci-dessous à l'aide du diagramme d'états du séquenceur (en figure 22) et à l'aide du schéma du chemin de données (en figure 21)

Etats du graphe	***Init***	***0***	***1***	***2***																
A	8	8	8																	
B	6	6	6																	
T	X	X	8																	
D=T-B	X	X	2																	
TinfB	X	X	0																	
Beq0	0	0	0																	

Vers les systèmes monopuces (SoC)

Si l'on désire aller plus loin dans l'intégration de systèmes dans un FPGA, il faut commencer à penser aux microprocesseurs et aux micro contrôleurs. Nous allons commencer par une architecture qui comporte un jeu d'instructions réduit (quatre instructions) mais très bien pensé.

Étude du miniprocesseur embarqué "MCPU"

Tim Boescke, cpldcpu@opencores.org a publié en 2001-2004 un miniprocesseur de quatre instructions (http://opencores.org/project,mcpu) destiné à tenir dans un CPLD. Les instructions sont les suivantes :

Mnémonique	***Opcode***	***Description***
NOR	00AAAAAA	Accu = Accu NOR mem[AAAAAA]
ADD	01AAAAAA	Accu = Accu + mem[AAAAAA] + maj retenue
STA	10AAAAAA	mam[AAAAAA] = Accumulateur
JCC	11DDDDDD	positionne PC à DDDDDD quand retenue =0 + clear carry

L'astuce de cette architecture est qu'il est possible de définir par dessus ces instructions des macros instructions de manière assez subtile :

Macro	***Assembleur***	***Description***
CLR	NOR allone	Accu = 0 (allone doit contenir0xFF)
LDA mem	NOR allone,ADD mem	Charge la memoire dans l'accumulateur
NOT	NOR zero	inverse le contenu de l'accumulateur (zero contient 0x00
JMP dst	JCC dst, JCC dst	Saut inconditionnel à dest
JCS dst	JCC *+2, JCC dst	Saut si retenue
SUB mem	NOR zero,ADD mem, ADD one	Soustrait mem de l'accu (one contient 0x01)
MOV src,dest	NOR allone , ADD src , STA dest	déplacement de la mémoire src vers dest

Vous disposez à ce point de 11 instructions (4 instructions d'origine + 7 macro), ce qui est suffisant pour un calcul de PGCD.

Transformer notre FSMD en programme

Il est toujours possible de transformer un FSMD ou un diagramme d'évolution tel que celui de la figure 22 en un ensemble d'instructions. Il faudra ensuite imaginer une architecture capable de réaliser un tel programme. Puisque notre architecture existe il nous faut être imaginatif et adapter les instructions. Pour notre exemple de PGCD cela peut se faire avec un programme du style :

```
1         USE "cpu3.inc"
2
3         start:
4           LDA   allone
5           ADD   b
6           JCC   NotBeq0
7           JMP   end
8         NotBeq0:      MOV   a,t   ; t <- A
9         rem:
10          NOR   b
11          ADD   one   ;Akku = - b
12          ADD   a     ;Akku = a - b
13                      ;Carry set when akku >= 0
14          STA   t
15          NOT         ;Acc = -t
16          ADD   b     ;Acc = b-t
17          JCS   rem
18          MOV   b,a   ;a <- b
19          MOV   t,b   ;b <- t
20          JMP   start
21        end:
22          JMP   end
23        a:
24          DCB   (126)
25        b:
26          DCB   (124)
27        t:
28          DCB   (0)
```

Programme 29: Calcul du PGCD sur MCPU

Ce programme est facile à comprendre pour qui a un peu d'expérience car il reprend la technique utilisée dans le séquenceur (j'ai modifié le programme original de Tim Boescke pour cela).

Transformer notre programme en ...

Quand on a l'habitude de faire fonctionner des processeurs on sait qu'il suffit de compiler un programme en fichier hex qu'un programmateur permettra d'envoyer à une EPROM. Mais ici, comment et où va finir notre programme ? En général on dispose d'un convertisseur qui permet de transformer notre fichier hex en VHDL. Ce VHDL finira dans des blocs logiques ou dans de la RAM (il y en a dans tout FPGA moderne) ; c'est le style du programme VHDL qui définira si l'on utilise des blocs logiques ou de la RAM.

Exercice 5

Le programme VHDL de description original de Tim Boescke pour décrire son architecture est tellement simple que nous le donnons maintenant complètement :

```
29        library ieee;
30        use ieee.std_logic_1164.all;
31        use ieee.std_logic_unsigned.all;
32
33        entity CPU8BIT2 is
34          port ( data:  inout  std_logic_vector(7 downto 0);
35                 adress:       out    std_logic_vector(5 downto 0);
36                 oe:    out    std_logic;
37                 we:    out    std_logic;   -- Asynchronous memory interface
38                 rst:   in     std_logic;
```

```
39              clk:   in     std_logic);
40      end;
41
42      architecture CPU_ARCH of CPU8BIT2 is
43        signal akku:  std_logic_vector(8 downto 0);    -- akku(8) is carry !
44        signal adreg: std_logic_vector(5 downto 0);
45        signal        pc:    std_logic_vector(5 downto 0);
46        signal states:      std_logic_vector(2 downto 0);
47      begin
48        process(clk,rst)
49        begin
50          if (rst = '0') then
51              adreg  <= (others => '0');-- start execution at memory location 0
52              states <= "000";
53              akku <= (others => '0');
54              pc   <= (others => '0');
55          elsif rising_edge(clk) then
56              -- PC / Adress path
57              if (states = "000") then
58                    pc     <= adreg + 1;
59                    adreg  <= data(5 downto 0);
60              else
61                    adreg  <= pc;
62              end if;
63              -- ALU / Data Path
64              case states is
65                when "010" => akku <= ("0" & akku(7 downto 0)) + ("0" & data);  -- add
66                when "011" => akku(7 downto 0) <= akku(7 downto 0) nor data;    -- nor
67                when "101" => akku(8) <= '0';-- branch not taken, clear carry
68                when others => null;    -- instr. fetch, jcc taken (000), sta (001)
69              end case;
70              -- State machine
71              if (states /= "000") then states <= "000";    -- fetch next opcode
72              elsif (data(7 downto 6) = "11" and akku(8)='1') then states <= "101";   --
        branch not taken
73              else  states <= "0" & not data(7 downto 6);-- execute instruction
74              end if;
75          end if;
76        end process;
77        -- output
78        adress <= adreg;
79        data   <= "ZZZZZZZZ" when states /= "001" else akku(7 downto 0);
80        oe <= '1' when (clk='1' or states  = "001" or rst='0' or states = "101") else
        '0';     -- no memory access during reset and
81        we <= '1' when (clk='1' or states /= "001" or rst='0') else '0';              --
        state "101" (branch not taken)
82      end CPU_ARCH;
```

Décomposer cette architecture en chemin de données et séquenceur, d'abord de manière schématique, puis en programme VHDL.