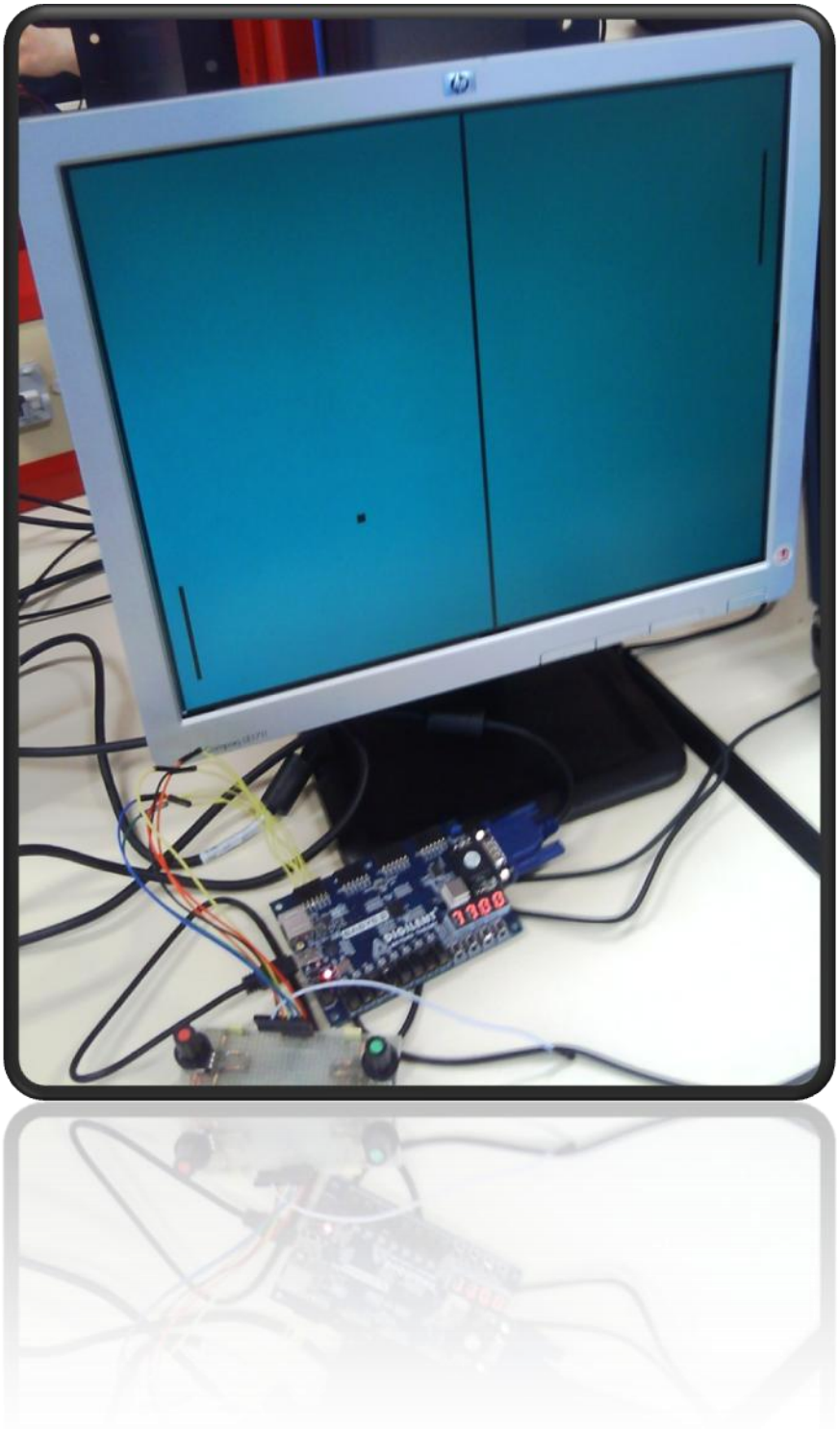
Comte-Gaz Quentin 1AG2TD2TP3

Pizel Jérôme Professeur encadrant : Tauvel Antoine

Le télécran :



Plan :

I) Objectif

a) L'énoncé

Ce projet consiste à réaliser le télécran présenté en examen à l'ENSEA en 2012.
Ce télécran sera piloté par deux encodeurs rotatifs permettant de jouer sur les pointeurs horizontaux et verticaux.
La résolution visée est de 160x120 pixels, monochrome (noir / couleur à définir, code sur 1 bit).
On utilisera pour cela une carte Spartan3E.

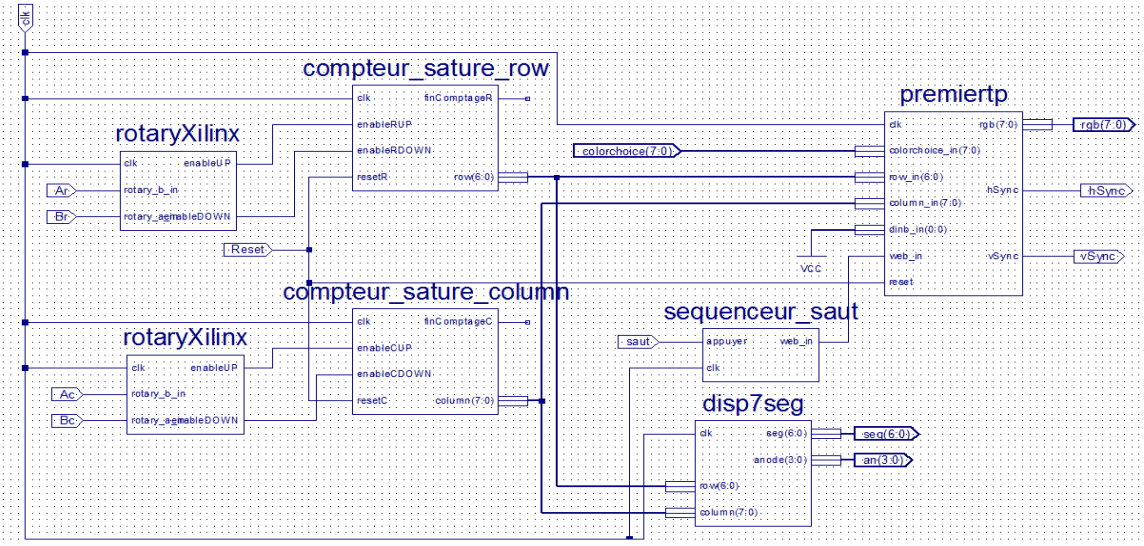
b) Les points importants de cet énoncé

- La **résolution de l'écran** (800x600) est différente de la **résolution du télécran** (160x120).
- Nous allons devoir enregistrer le dessin en cours dans une **RAM**.
- La création d'une plaque contenant les deux encodeurs rotatifs est nécessaire.
- L'énoncé nous propose de faire un télécran « basique » → Libre à nous de l'**améliorer**.
- Au vu du temps pour faire ce projet, nous utiliserons une carte Basys2 Spartan 3E-100 (le CS nous a fait démontrer que cette carte est adaptée à notre projet).

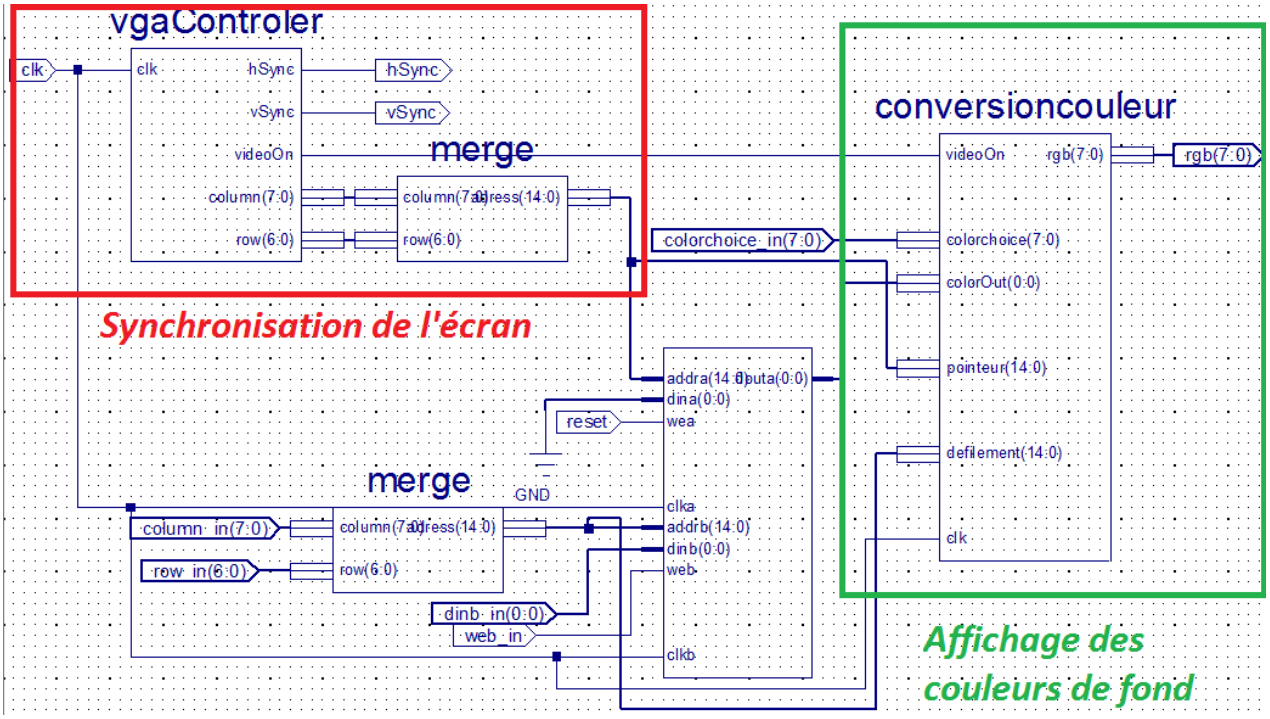
c) Les différentes parties du télécran

- Gestion de la synchronisation de l'écran et de la couleur de fond
- Affichage du pointeur/fond/dessin sur l'écran
- Enregistrement du dessin dans une RAM
- Améliorations : Le reset, le « levé de crayon », la couleur du pointeur

Voici un aperçu de notre projet au complet :



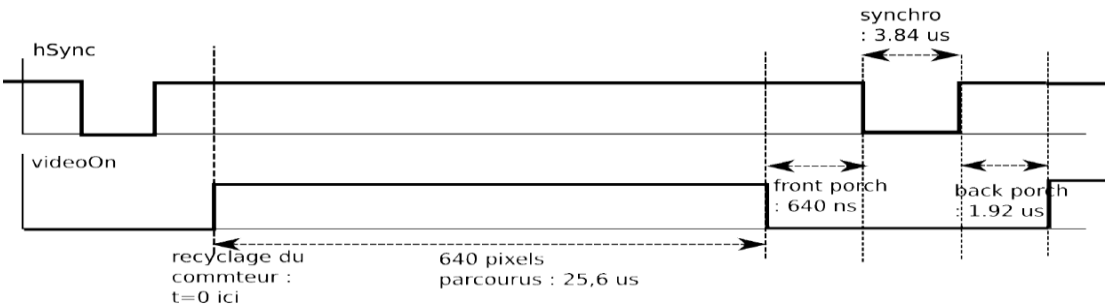
II) Gestion de l'écran



a) Synchronisation de l'écran

D'après la norme VGA, les signaux de synchronisations correspondent au temps nécessaire au faisceau pour revenir soit en haut, arrivé à la dernière ligne (synchronisation verticale), soit à gauche, après le dernier pixel (synchronisation horizontale). De plus, avant et après ces signaux se trouvent un « front porch » et un « back porch » correspondant à des temps pendant lesquels le faisceau est en dehors de l'écran.

Changement de la résolution :
On passe d'une résolution de 800*600 à une résolution de 160*120.
D'où l'utilité de ce bloc permettant de synchroniser l'écran verticalement (vSync) et horizontalement (hSync).
VideoOn permet d'afficher quelque chose si l'on est bien sur l'écran.



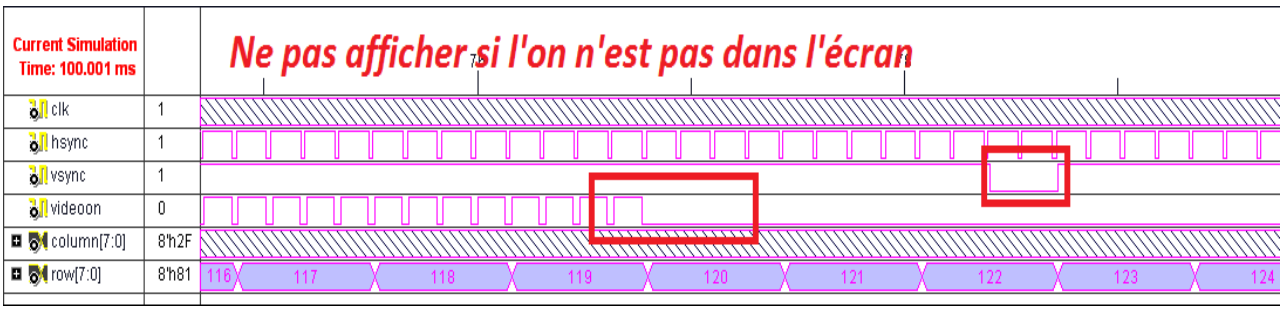
Timing de la synchronisation horizontale VGA 640 x 480 à 60 Hz

Remarquons que notre horloge est à 50Mz et non à 60Hz.
Ainsi, le temps de parcours de 640 pixels est beaucoup plus faible que celui écrit dans la figure ci-dessus.

Voici le code associé :

```
25  entity vgaControler is
26     port (clk : in std_logic;
27           hSync, vSync : out std_logic;
28           videoOn : out std_logic;
29           column : out std_logic_vector (7 downto 0);
30           row : out std_logic_vector (6 downto 0));
31  end vgaControler;
32
33  architecture Behavioral of vgaControler is
34  signal hCounter : std_logic_vector (9 downto 0) := (others=>'0');
35  signal vCounter : std_logic_vector (9 downto 0) := (others=>'0');
36
37     begin
38        process (clk)
39        begin
40           if (clk'event and clk='1') then
41              if hCounter = 799 then
42                 hCounter <= (others=>'0');
43              else hCounter <= hCounter + '1';
44              end if;
45           end if;
46        end process;
47
48     hSync <= '0' when (hCounter > 655) and (hCounter < 751) else '1';
49     vSync <= '0' when (vCounter > 489) and (vCounter < 492) else '1';
50
51     process (clk)
52        begin
53           if (clk'event and clk='1') then
54              if (hCounter = 799 and vCounter=520) then
55                 vCounter <= (others=>'0');
56              elsif (hCounter = 799) then
57                 vCounter <= vCounter + '1';
58              end if;
59           end if;
60     end process;
61
62     videoOn <= '1' when hCounter<640 and vCounter < 480 else '0';
63
64     column <= hCounter (9 downto 2);
65     row <= vCounter (8 downto 2);
66
67  end Behavioral;
```

Nous avons vérifier le fonctionnement de ce bloc sur Xilinx :



Nous avons bien les 2 lignes pour vSync comme prévu dans le fichier basysc.

On remarque que, comme prévu, le bloc compte les colonnes et les lignes et revient à la première colonne à chaque changement de ligne, cela jusqu'à la fin de l'écran.

b) Affichage des couleurs de fond

Le bloc « conversioncouleur » permet d'afficher :

- La couleur de fond : Grâce à 8 interrupteurs disponibles sur la carte Basys, nous pouvons choisir notre couleur de fond (colorchoice) parmi 2^8-1 = 255 possibilités.

- Le dessin (lorsque la RAM nous renvoie '1').

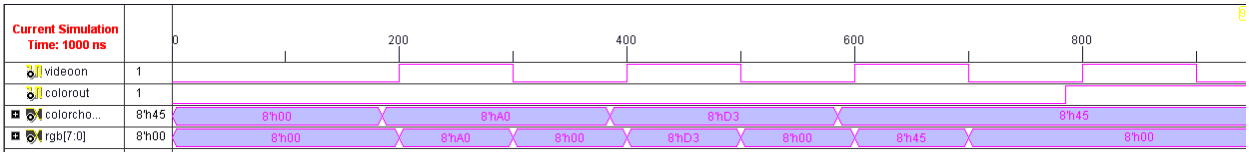
- Le pointeur (sa couleur est le complémentaire de colorchoice afin de savoir où le pointeur se trouve).

Nous envoyons sur RGB la couleur que l'on veut afficher (pixel par pixel).

Voici le tableau logique de notre bloc :

VideoOn (in)	ColorOut (in)	RGB (out)	
0	X	00000000	← Si l'on n'est pas sur l'écran
1	1	00000000	← Affichage du dessin
1	0	colorchoice	← Affichage du fond

Grace à une simulation sur Xilinx, on remarque ce cela est bien respecté :

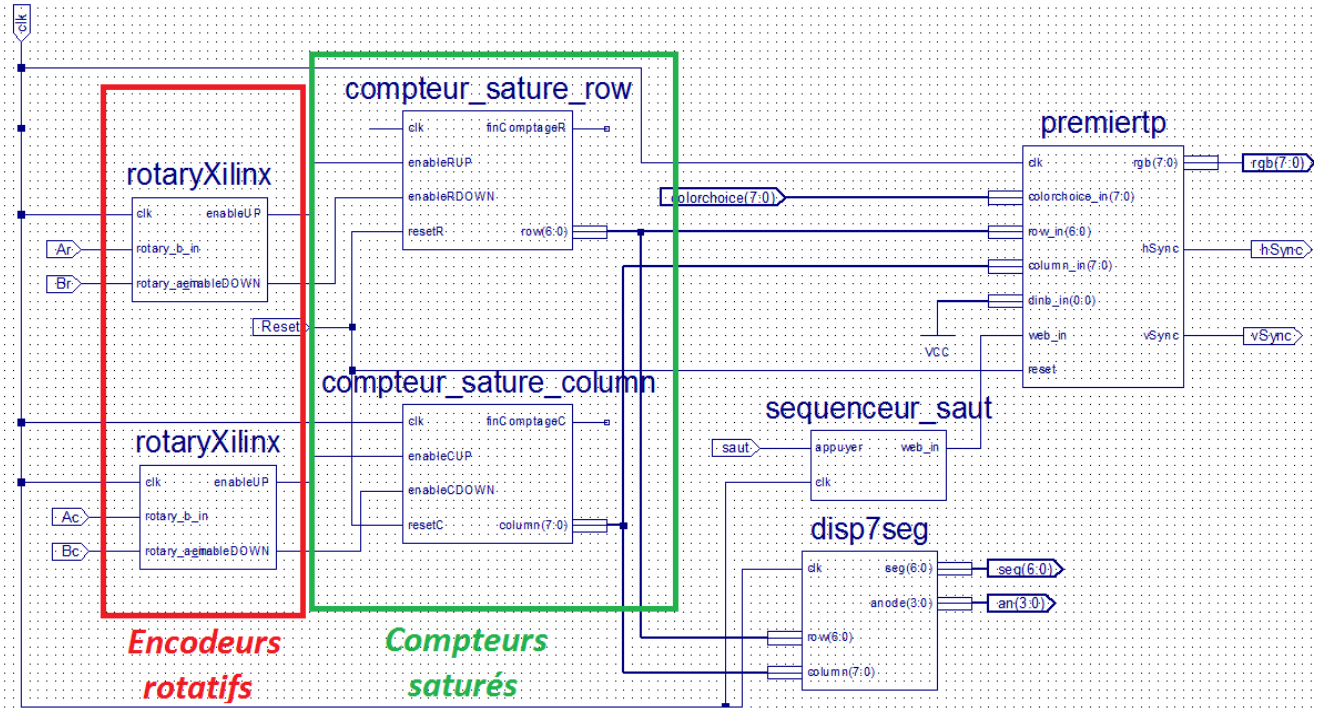


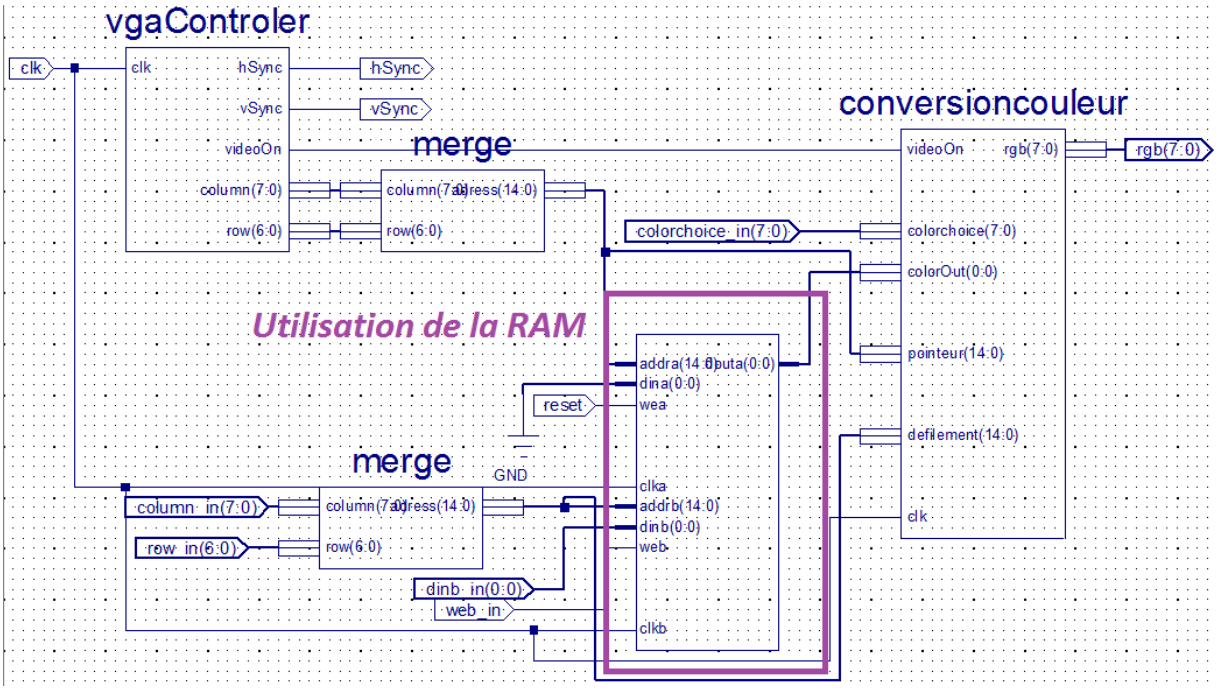
Voici le code associé à ce bloc :

```
30  entity conversioncouleur is
31      Port ( clk, videoOn : in  STD_LOGIC;
32             colorOut : in  STD_LOGIC_VECTOR (0 downto 0);
33             colorchoice : in  STD_LOGIC_VECTOR (7 downto 0);
34             rgb : out  STD_LOGIC_VECTOR (7 downto 0);
35             pointeur, defilement : in std_logic_vector (14 downto 0)
36             );
37  end conversioncouleur;
38
39  architecture Behavioral of conversioncouleur is
40  signal egal : std_logic:='0';
41     begin
42              process (clk)
43              begin
44                 if (clk'event and clk='1')
45                 then
46                    if pointeur=defilement
47                    then egal <= '1';
48                    else egal <= '0';
49                    end if;
50                 end if;
51              end process;
52              rgb <= (not colorchoice) when egal='1' else "00000000" when (videoOn ='0' or colorOut="1") else colorchoice;
53
54
55
56  end Behavioral;
```

Remarquons que l'on a not(colorchoice) comme couleur du pointeur (cas que l'on a omis dans le tableau logique).

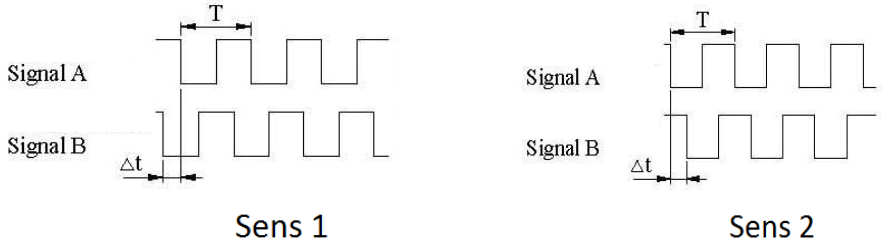
III) Gestion du pointeur et du dessin





a) Les encodeurs rotatifs

L'encodeur rotatif nous envoie les signaux suivants selon le sens de rotation :

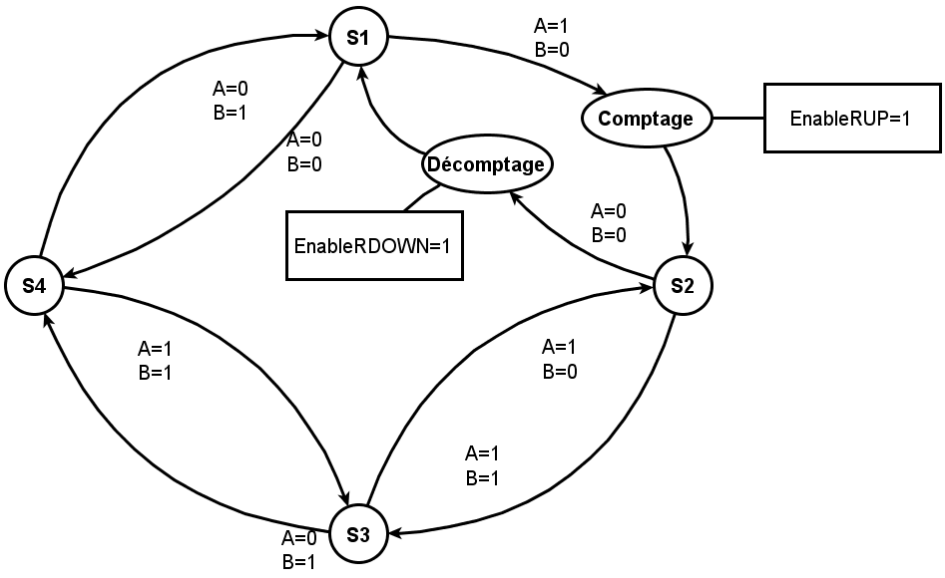


Sens 1 Sens 2

Il nous faut donc créer un séquenceur permettant de **savoir si nous tournons ce codeur rotatif** et quel est le **sens de rotation**.
Pour cela, nous allons vous présenter 2 encodeurs : un qui nous a été fourni par Xilinx et un que nous avons conçu mais qui ne marche qu'en partie.

- **Notre séquenceur ne fonctionnant pas complètement :**

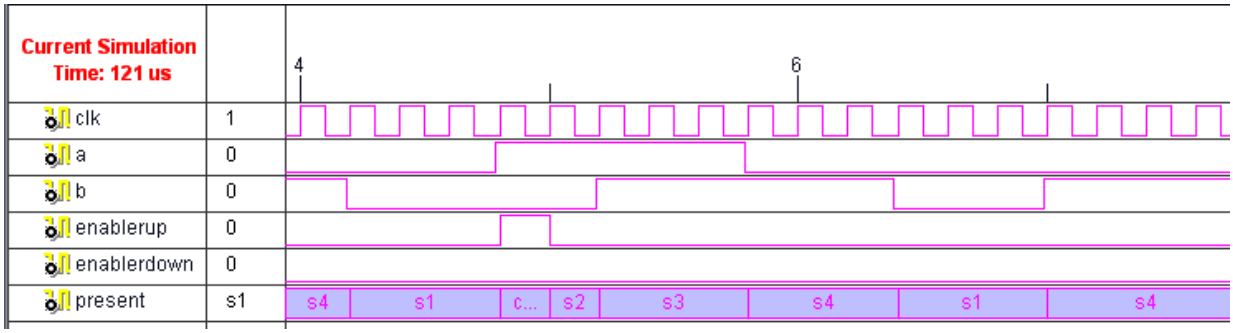
Nous avons conçu un séquenceur qui n'a pas eu l'effet voulu.
En effet, il ne marchait que si nous voulions aller en haut ou vers la droite (cela viens sans doute du fait que nous n'avions pas pris en compte le rebond et l'enregistrement du sens de rotation).



Voici le code VHDL associé :

```
entity Sequenceur is
    Port ( clk : in  STD_LOGIC;
           A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           EnableRUP : out  STD_LOGIC;
           EnableRDOWN : out  STD_LOGIC);
end Sequenceur;

architecture Behavioral of Sequenceur is
type etat is (S1,S2,S3,S4,Comptage,Decomptage);
signal present,futur : etat;
begin
      process(clk)
      begin
            if(clk'event and clk='1') then
               present<=futur;
            end if;
      end process;

      process(present,A,B)
      begin
      case present is
            when S1 =>
               if A='1' and B='0'
                  then futur <= S2;
                  elsif A&B="01"  then futur <= S4;
                  else futur <= S1;
               end if;

            when S2 =>
               if A='1' and B='1'
                  then futur <= S1;
                  elsif A&B="00"  then futur <= Comptage;
                  else futur <= S2;
               end if;


            when S3 =>
               if A='1' and B='0'
                  then futur <= S2;
                  elsif A&B="01"  then futur <= S4;
                  else futur <= S3;
               end if;


            when S4 =>
               if A='1' and B='1'
                  then futur <= S1;
                  elsif A&B="00"  then futur <= Decomptage;
                  else futur <= S4;
               end if;


            when Comptage =>
               futur <= S3;

            when Decomptage =>
               futur <= S3;


      end case;
      end process;

      EnableRUP <= '1' when (present = Comptage) else '0';
      EnableRDOWN <= '1' when (present = Decomptage) else '0';

end Behavioral;


```

La simulation sous Xilinx donne des résultats pouvant nous amener à croire que ce séquenceur fonctionne bel et bien (ce qui n'est pas le cas) :



- **Rotary By Xilinx :**

Ce code VHDL définit un séquenceur permettant de gérer le déplacement du pointeur à l'écran.
Les deux signaux essentiels sont Rotary_q1 et Rotary_q2 :
Rotary_q1 détermine si on tourne ou non l'encodeur et Rotary_q2 détermine le sens de rotation.

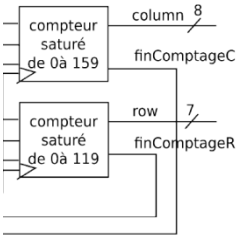
Le process « direction » permet de garder le dispositif synchrone et de conserver en mémoire le sens de rotation.

Voici le code VHDL
commenté :

```
entity rotaryXilinx is
    Port ( clk : in  STD_LOGIC;
           rotary_b_in : in  STD_LOGIC;
           rotary_a_in : in  STD_LOGIC;
           enableUP : out STD_LOGIC;
           enableDOWN : out  STD_LOGIC
           );
end rotaryXilinx;

architecture Behavioral of rotaryXilinx is
signal rotary_event, rotary_left, rotary_q1, rotary_q2, delay_rotary_q1, delay_rotary_q2 : std_logic:='0';
signal rotary_in : std_logic_vector (1 downto 0) := "00";

begin

   rotary_filter: process(clk)
      begin
      if clk'event and clk='1' then
         rotary_in <= not (rotary_b_in & rotary_a_in);
-- rotary_q1 est à 0 si l'on ne tourne pas l'encodeur et à 1 dans le cas contraire
-- rotary_q2 est à 0 si on tourne vers la gauche et à 1 si l'on tourne vers la droite

         case rotary_in is
               when "00" =>   rotary_q1 <= '0';
                              rotary_q2 <= rotary_q2;
                              -- Si on ne bouge pas l'encodeur alors on ne bouge pas le pointeur

               when "01" =>   rotary_q1 <= rotary_q1;
                              rotary_q2 <= '0';
                              -- Si on bouge dans sens 0 alors on pose rotary_q2=0

               when "10" =>   rotary_q1 <= rotary_q1;
                              rotary_q2 <= '1';
                              -- Si on bouge dans sens 1 alors on pose rotary_q2=1

               when "11" =>   rotary_q1 <= '1';
                              rotary_q2 <= rotary_q2;
                              -- Si on bouge l'encodeur et que l'on va dans le même sens que l'état précédent

               when others => rotary_q1 <= rotary_q1;
                              rotary_q2 <= rotary_q2;
                               -- Cas de bug
         end case;
      end if;
   end process rotary_filter;

direction: process(clk)
-- !! Cette partie permet de garder ce dispositif synchrone et de conserver en mémoire le sens de rotation !!
   begin
      if clk'event and clk='1' then
         delay_rotary_q1 <= rotary_q1;
         if rotary_q1='1' and delay_rotary_q1='0' then
         -- Si l'on a tourné d'un cran
         -- Alors il y a eu un "évènement de rotation" et rotary_left récupère le sens de rotation
            rotary_event <= '1';
            rotary_left <= rotary_q2;
         else
         -- Sinon, on n'a pas tourné d'un cran et rotary_left garde sa valeur.
            rotary_event <= '0';
            rotary_left <= rotary_left;
   end if;
   end if;
end process direction;

   enableUP<='1' when rotary_event='1' and rotary_left ='1' else '0';
   -- Si l'encodeur a tourné et qu'il tourne à gauche Alors on incrementera le compteur saturé.
   enableDOWN<='1' when rotary_event='1' and rotary_left ='0' else '0';
   -- Idem mais lorsque l'encodeur tourne à droite.

end Behavioral;
```

b) Les compteurs saturés

On a deux compteurs/décompteur saturés : un pour colonne, l'autre pour ligne.
Ils sont la suite logique des encodeurs rotatifs. L'un fait bouger le compteur/décompteur de haut en bas et l'autre de droite à gauche.
Ceci jusqu'à la fin de l'écran où le compteur/décompteur se bloquera.
Ils permettent donc de bloquer le pointeur aux extrémités de l'écran tout en « comptant » la position de la ligne et de la colonne.

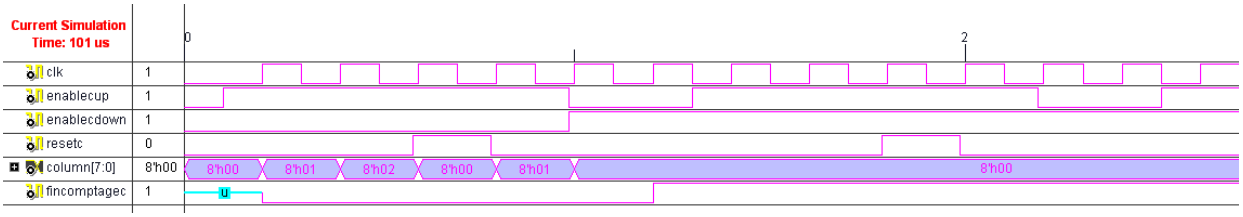


Voici le code VHDL pour le compteur saturé colonne (pour les lignes, le compteur fonctionne de la même manière) :

```
entity compteur_sature_column is
    Port ( clk : in  STD_LOGIC;
           enableCUP : in  STD_LOGIC;
           enableCDOWN : in  STD_LOGIC;
           resetC : in  STD_LOGIC;
           column : out  STD_LOGIC_VECTOR (7 downto 0);
           finComptageC : out  STD_LOGIC);
end compteur_sature_column;

architecture Behavioral of compteur_sature_column is
signal comptage : std_logic_vector(7 downto 0):=conv_std_logic_vector(80,8);

begin
process(clk)
begin

if(clk='1' and clk'event)
   then

      if(comptage<"10011111"  and enableCUP='1' and enableCDOWN='0' and resetC='0')
         then  comptage<=comptage+'1';
               finComptageC<='0';


      elsif(comptage="10011111"  and enableCUP='1' and enableCDOWN='0')
         then  finComptageC<='1';


      elsif(comptage>"00000000"  and enableCUP='0' and enableCDOWN='1' and resetC='0')
         then     comptage<=comptage-'1';
            finComptageC<='0';


      elsif(comptage="00000000"  and enableCUP='0' and enableCDOWN='1')
         then  finComptageC<='1' ;


      elsif(resetC='1')
         then comptage<="01010000";
      end if;

end if;
end process;

column<=comptage;


end Behavioral;
```

Remarquons le cas ResetC='1' que nous étudierons dans la partie IV.

Voici une simulation sur Xilinx montrant le fonctionnement de ce bloc :



c) Utilisation de la RAM

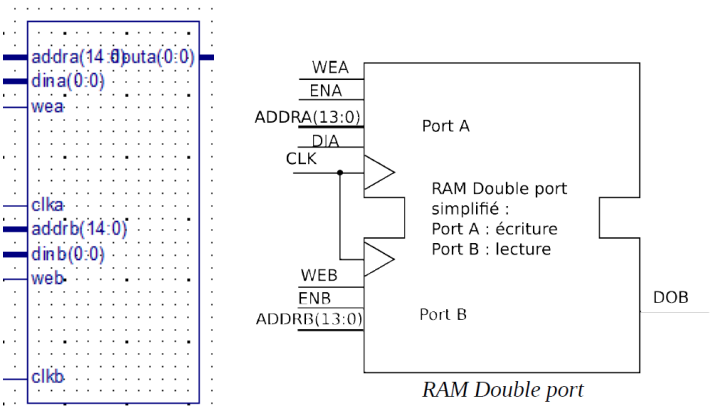
La RAM va nous permettre de **stocker l'image que l'on trace sur le télécran**.

Nous utilisons un « Memories and storage element : Dual port block memory » sous Xilinx.
En effet, la carte Basys comporte une RAM intégrée.
Il n'est donc pas nécessaire d'utiliser les bascules pour créer de la RAM (de plus, 2000 bascules est insuffisant pour créer la quantité de RAM que l'on souhaite).

Voici comment cette RAM double port est utilisée sur le port B :
A partir de la position du pointeur, définie par le couple ((column_in ; row_in), on écrit dans le port B de la RAM.

- Si dans la RAM un pixel est à « 0 », il s'affichera de la couleur de l'écran.
- Si dans la RAM un pixel est à « 1 », il s'affichera noir.

Pour tracer notre image, on va donc avoir deux pointeurs, *row* et *column* permettant de nous repérer dans l'image. Ils se codent respectivement sur 7 et 8 bits.



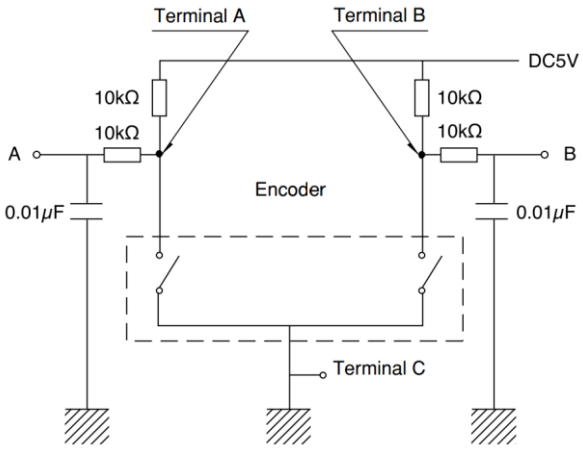
– ENx permet d'activer le port X.
– WEx : S'il vaut '1' : écriture. S'il vaut '0' : lecture.
– DIx : Donnée entrée sur le port X.
– DOx : Donnée lue sur le port X.
– ADDRx : Adresse de travail du port X.

Figure : Ram double port

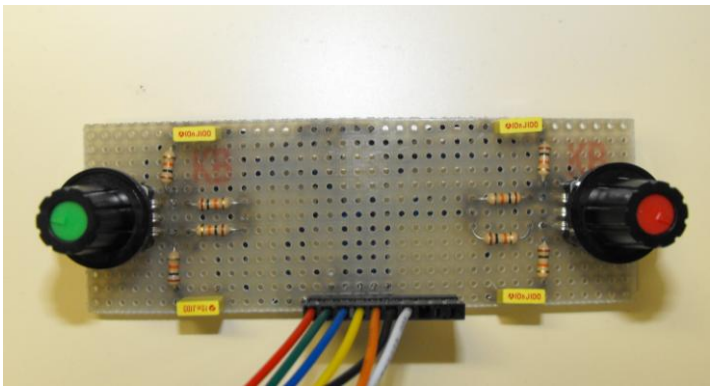
IV) Circuit du montage et les améliorations

a) Le circuit

Les 2 encodeurs rotatifs sont soudés sur la plaque de la manière suivante :

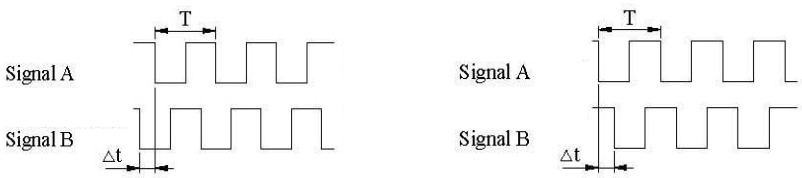


Voici donc une photo du montage :



Nous utilisons 6 broches pour relier la carte Basys à ce montage :

- Vcc
- Gnd
- 2 ports (A et B) pour chaque encodeur rotatif permettant de connaitre le sens de rotation grâce aux caractéristiques suivantes :



b) Cerise sur le gâteau

Ne pouvant plus nous arrêter, nous avons décidé d'améliorer notre projet en rajoutant quelques fonctionnalités :

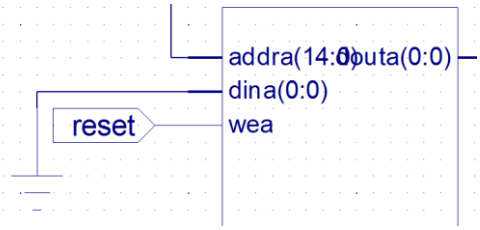
i. Le reset

Le Reset, qui permet d'effacer l'image, se décompose en deux parties :

- La remise du pointeur dans son état initial, c'est-à-dire au milieu de l'écran :
Dans les deux compteurs saturés, nous avons donc rajouté trois lignes de code :

```
elsif(resetC='1')
   then comptage<="01010000";
end if;
```

- La réinitialisation de la RAM par le biais du port A (on écrit des '0'dans la RAM pendant l'appui sur un bouton « reset ») :



ii. Couleur du pointeur

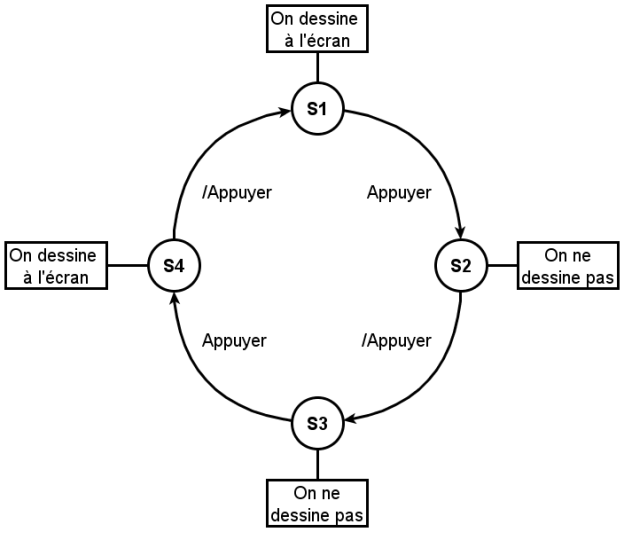
Cette amélioration permet d'avoir une couleur du pointeur différente de celle du fond.
Pour cela, nous avons modifié la forme de notre bloc « conversioncouleur » pour qu'il renvoie not(colorchoice) quand le défilement de l'écran arrive sur le pixel pointeur.
Ainsi, la couleur du pointeur sera l'opposé de celle du fond.

```
process (clk)
begin
   if (clk'event and clk='1')
   then
      if pointeur=defilement
      then egal <= '1';
      else egal <= '0';
      end if;
   end if;
end process;
rgb <= (not colorchoice) when egal='1'
```

iii. Le « levé de crayon »

Le séquenceur « levé de crayon » permet de se balader sur l'écran avec le pointeur sans rien tracer. Cela est équivalent à choisir d'envoyer un '1' ou un '0' dans la RAM.

Si l'on appuie une fois sur le bouton « saut », alors on peut se balader sans rien tracer.
Quand on ré-appuie sur le bouton, alors on peut tracer à nouveau.



Voici le code VHDL associé :

```
entity sequenceur_saut is
    Port ( appuyer : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           web_in : out  STD_LOGIC);
end sequenceur_saut;

architecture Behavioral of sequenceur_saut is
type etat is (S1,S2,S3,S4);
signal present,futur : etat;
begin
      process(clk)
      begin
            if(clk'event and clk='1') then
               present<=futur;
            end if;
      end process;

      process(present,appuyer)
      begin
      case present is
         when S1 =>
            if appuyer='1'
               then futur <= S2;
            else futur <= S1;
            end if;

         when S2 =>
            if appuyer='0'
               then futur <= S3;
            else futur <= S2;
            end if;

         when S3 =>
            if appuyer='1'
               then futur <= S4;
            else futur <= S3;
            end if;

         when S4 =>
            if appuyer='0'
               then futur <= S1;
            else futur <= S4;
            end if;

end case;
end process;

web_in <= '1' when (present = S1 or present = S4) else '0';

end Behavioral;
```

iv. Le débuggeur

Ce bloc n'est pas vraiment ce que l'on peut appeler une amélioration.
En effet, il permet seulement au programmeur de visualiser sur les afficheurs 7 segments de la carte basys où se situe le pointeur (2 afficheurs pour les lignes et 2 pour les colonnes).
Ce bloc est très pratique pour mieux comprendre ce que fait notre programme et permet ainsi d'identifier plus facilement les erreurs éventuelles.

Ce bloc récupère la valeur de column et row pour les envoyer sur les 4 afficheurs 7 segments sous forme hexadécimale :

```
30  entity disp7seg is
31      Port (  clk : in std_logic;
32              row : in  STD_LOGIC_VECTOR (6 downto 0);
33              column : in std_logic_vector (7 downto 0);
34              seg : out  STD_LOGIC_VECTOR (6 downto 0);
35              anode : out  STD_LOGIC_VECTOR (3 downto 0));
36  end disp7seg;
37
38  architecture Behavioral of disp7seg is
39  signal data_in : std_logic_vector (15 downto 0);
40  signal comptage : std_logic_vector (18 downto 0) :=(others=>'0');
41  alias value is comptage (18 downto 17);
42  signal multiplex : std_logic_vector (3 downto 0) :=x"0";
43  type tv is array (0 to 15) of std_logic_vector (6 downto 0);
44  constant table : tv := (   "1000000", "1111001",
45                             "0100100", "0110000",
46                             "0011001", "0010010",
47                             "0000010", "1111000",
48                             "0000000", "0010000",
49                             "0001000", "0000011",
50                             "1000110", "0100001",
51                             "0000110", "0001110");
52  begin
53
54     data_in <= '0' & row & column;
55
56     process (clk)
57     begin
58        if (clk'event and clk='1')
59        then
60           comptage <= comptage + '1';
61        end if;
62     end process;
63
64     anode <= "1110" when value="00" else "1101" when value="01" else "1011" when value="10" else "0111";
65
66     multiplex <= data_in (conv_integer(value)*4 + 3 downto conv_integer(value)*4);
67
68     seg <= table (conv_integer(multiplex));
69
70  end Behavioral;
```

V) Annexes

a) Ressources utilisées

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	69	1,920	3%
Number of 4 input LUTs	139	1,920	7%
Logic Distribution			
Number of occupied Slices	104	960	10%
Number of Slices containing only related logic	104	104	100%
Number of Slices containing unrelated logic	0	104	0%
Total Number of 4 input LUTs	175	1,920	9%
Number used as logic	139		
Number used as a route-thru	36		
Number of bonded IOBs	35	83	42%
Number of Block RAMs	2	4	50%
Number of GCLKs	1	24	4%
Total equivalent gate count for design	132,698		
Additional JTAG gate count for IOBs	1,680		

Remarquons que nous utilisons peu de ressources type bascules et entrées/sorties mais que nous consommons 50% de la RAM de la carte.
Cela vient du fait que nous enregistrons chaque pixel de notre écran 160x120 dans cette RAM.
Comme nous utilisons très peu de bascules, nous avons encore la possibilité de rajouter de nombreuses fonctionnalités dans ce projet.

b) Améliorations possibles

Plusieurs améliorations étaient envisageables, mais par faute de temps nous n'avons pas pu les implémenter dans notre projet.
Nous aurions ainsi pu **créer une gomme** afin d'effacer seulement quelques pixels de notre écran.
Mais aussi **afficher une image préenregistrée** à l'aide d'un bouton (nous avons pile la place dans la RAM).

c) Télécharger notre projet

Vous pouvez télécharger le dossier, le code VHDL ainsi que les fichiers utiles à la création de ce projet à cette adresse :
http://quentincg.free.fr/ensea/projet_telecran_ensea.zip (veuillez lire le fichier « A LIRE.txt » avant d'utiliser notre projet).

d) Bibliographie

- http://moodle.ensea.fr/ (la plupart des documents importants pour le projet se trouvent ici)
- -http://www.google.fr (pour avoir une idée plus précise de ce qu'est un télécran)
- http://www.xilinx.com/ (code VHDL du séquenceur « Rotary by Xilinx »)