Adrien BONNAND

POLYTECH
DIJON

Compte-Rendu
Projet filtre 2D

Table des matières

Table des figures

INTRODUCTION GÉNÉRALE

L'objectif de ce TP est de réaliser un filtre 2D sur une image qui a une hauteur et une largeur de 128 pixels. Afin de réaliser cette tâche, plusieurs étapes seront nécessaires :

1) Ecrire un programme qui lit une image dans un fichier et la réécrit dans un autre fichier.

2) Générer une FIFO et s'assurer qu'elle fonctionne correctement

3) Créer une ligne à retard en ajoutant des bascules et deux FIFOs citées précédemment.

4) Ecrire un programme qui permet de contrôler cette ligne à retard afin de récupérer les pixels de l'image et de les filtrer (premièrement avec une moyenne et deuxièmement avec un filtre Sobel/Laplacien).

5) Implémenter le filtre 2D sur carte (Nexys 4 DDR)

Cela permettra, en réalisant l'ensemble de ces tâches, de montée en compétences dans la programmation VHDL sur architecture FPGA.

1 : Duplication de l'image Lena

Figure 1 : Image de Léna

Introduction

L'objectif est de reprendre le testbench fournit par le professeur, pour pouvoir lire l'intégralité d'un fichier et réécrire ses données dans un autre fichier. Quand le filtre sera prêt, il sera possible de lire les pixels de Léna, les filtrer et les écrire dans un nouveau fichier grâce à ce testbench.

Analyse du code modifié

```
p_read : process
 FILE vectors : text;
 variable Iline : line;
 variable I1_var :std_logic_vector (7 downto 0);

   begin
   DATA_AVAILABLE <= '0';
   file_open (vectors,"Lena128x128g_8bits.dat", read_mode);
   wait for 20 ns;

   while not endfile(vectors) loop
     readline (vectors,Iline);
     read (Iline,I1_var);

     I1 <= I1_var;
     DATA_AVAILABLE <= '1';
     wait for 10 ns;
   end loop;
   DATA_AVAILABLE <= '0';
   wait for 10 ns;
   file_close (vectors);
   wait;
end process;
```

Figure 2 : Process read

Ce process permet d'ouvrir le fichier « Lena128x128g_8bits.dat » en mode lecture et ensuite de lire le fichier, ligne après ligne, toutes les 10 ns, en stockant chaque valeur dans « I1_var ». Lors de la lecture, le signal « DATA_AVAILABLE » est mis à 1 et il repasse à zéro quand l'intégralité du fichier est lu.

```
p_write: process
  file results : text;
  variable OLine : line;
  variable O1_var :std_logic_vector (7 downto 0);

    begin
    file_open (results,"Lena128x128g_8bits_r.dat", write_mode);
    wait for 10 ns;
    wait until DATA_AVAILABLE = '1';
    wait for 10 ns;

    while DATA_AVAILABLE ='1' loop
      write (Oline, O1, right, 2);
      writeline (results, Oline);
      wait for 10 ns;
    end loop;
    file_close (results);
    wait;
 end process;
```

Figure 3 : Process write

Ce process ouvre le fichier « Lena128x128g8bits_r.dat » en mode écriture et attend que « DATA_AVAILABLE » soit à 1 pour commencer son écriture. Dès que c'est le cas, toutes les 10 ns, il va écrire dans le fichier « Lena128x128g8bits_r.dat » les valeurs qu'il reçoit du signal « O1 » et lorsque que « DATA_AVAILABLE » est égal à zéro, il s'arrête d'écrire.

Il suffit de brancher I1 à O1 pour pouvoir faire une copie de Léna.

Simulation

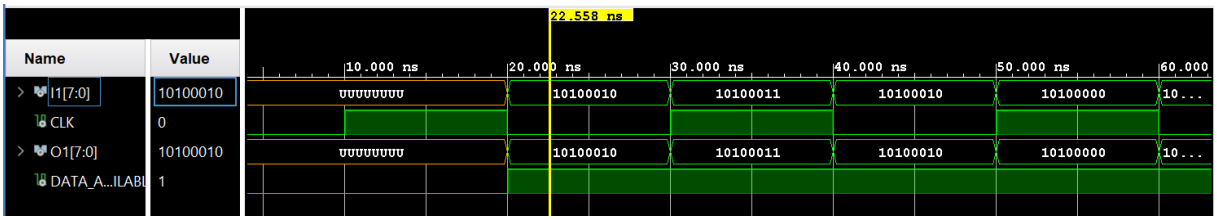

Figure 4 : Simulation de la copie de Léna

Maintenant, en lançant ce testbench, il est possible de lire les pixels présents dans « Lena128x128g8bits.dat » et de les écrire dans le fichier « Lena128x128g8bits_r.dat »

A l'aide du programme « dat2bmp » fourni, l'image peut être convertie en « Lena128x128g8bits_r.bmp », ce format permet de visualiser l'image résultante.

Lien des fichiers :

- *tools/tb_lena_dupliq_2p.vhd*
- *tools/dat2bmp*

2 : Génération FIFO + test unitaire

Introduction

L'objectif est de générer une FIFO avec des paramètres spécifiques et de vérifier son bon fonctionnement. Elle permettra, par la suite, de stocker les pixels lus avant de les traiter.

Génération de la FIFO

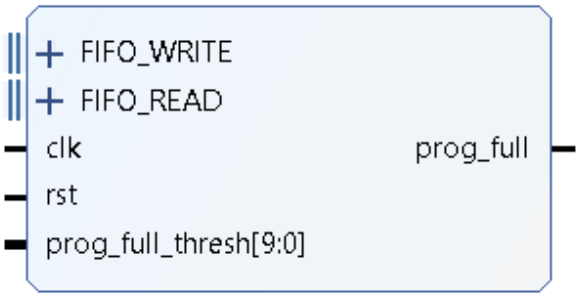

Figure 5 : Architecture de la FIFO

Lors de la configuration, deux ports sont ajoutés : « prog_full_thresh » et « prog_full ».

Cela permettra de mettre « prog_full » à 1 lorsque la FIFO stockera un nombre de pixels supérieur au seuil « prog_full_thresh ».

Test de la FIFO

Afin de tester la FIFO, 100 valeurs ont été rentrées (de 1 à 100) puis ont été dépilées :

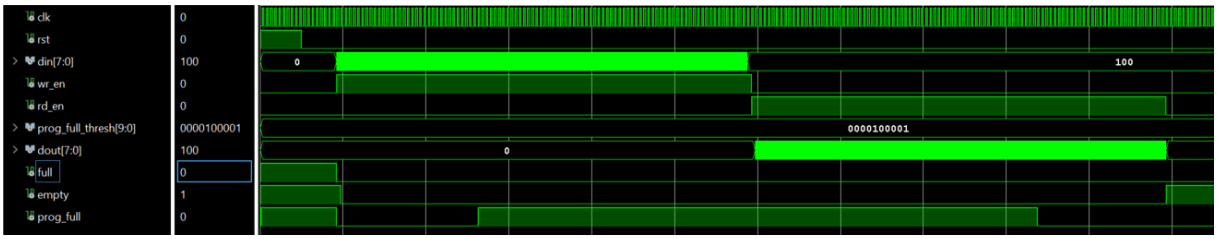

Figure 6 : Test de la FIFO

Le signal « prog_full » est mis à 1 au bout de 1/3 de l'écriture. Cela est cohérent, car « prog_full_thresh » est égale à 33 dans le testbench.

Analyse du code

fifo_tb.vhd :

```
fifo_storage: process
  variable compteur : integer := 0;
begin
  rst<='1';
  din<="00000000";
  wr_en<='0';
  prog_full_thresh<="0000100001";
  wait for 100 ns;
  rst<='0';
  wait until full = '0';
  wr_en<='1';

  while (compteur /= 100) loop
    compteur:=compteur+1;
      din<=std_logic_vector(to_unsigned(compteur,8));
    wait for 10ns;
  end loop;

  wr_en<='0';
  wait;
end process;

fifo_unstack : process
  variable compteur : integer := 0;
begin
  rd_en<='0';
  wait until falling_edge(wr_en);
  rd_en<='1';
  wait until empty = '1';
  rd_en<='0';

  wait;
end process;
```

Figure 7 : fifo_tb.vhd

Le premier process initialise « prog_full_thresh » à 33 et dès que « full » passe à 0, il vient stocker dans la FIFO, toutes les 10 ns, des valeurs provenant d'un compteur. Le deuxième process quant à lui, attend que l'écriture des 100 valeurs soit finie pour mettre « rd_en » à 1 et donc dépiler la FIFO.

Ce test permet de vérifier le bon fonctionnement de la FIFO et de comprendre que ce composant est en phase d'initialisation lorsque ses signaux *full* et *empty* sont tous deux à 1 (il est indisponible pendant cette durée).

Lien des fichiers :

- *Projet_TP.srcs/sources_1/new/fifo.vhd*
- *Projet_TP.srcs/sim_1/new/fifo_tb.vhd*

3 : Implémentation ligne à retard + test unitaire

Introduction

L'objectif est de créer une ligne à retard à l'aide de deux FIFOs et 9 bascules et de vérifier que celle-ci fonctionne correctement grâce à un test unitaire. Cette ligne à retard permettra de récupérer uniquement les pixels de l'image à filtrer.

Architecture de la ligne à retard

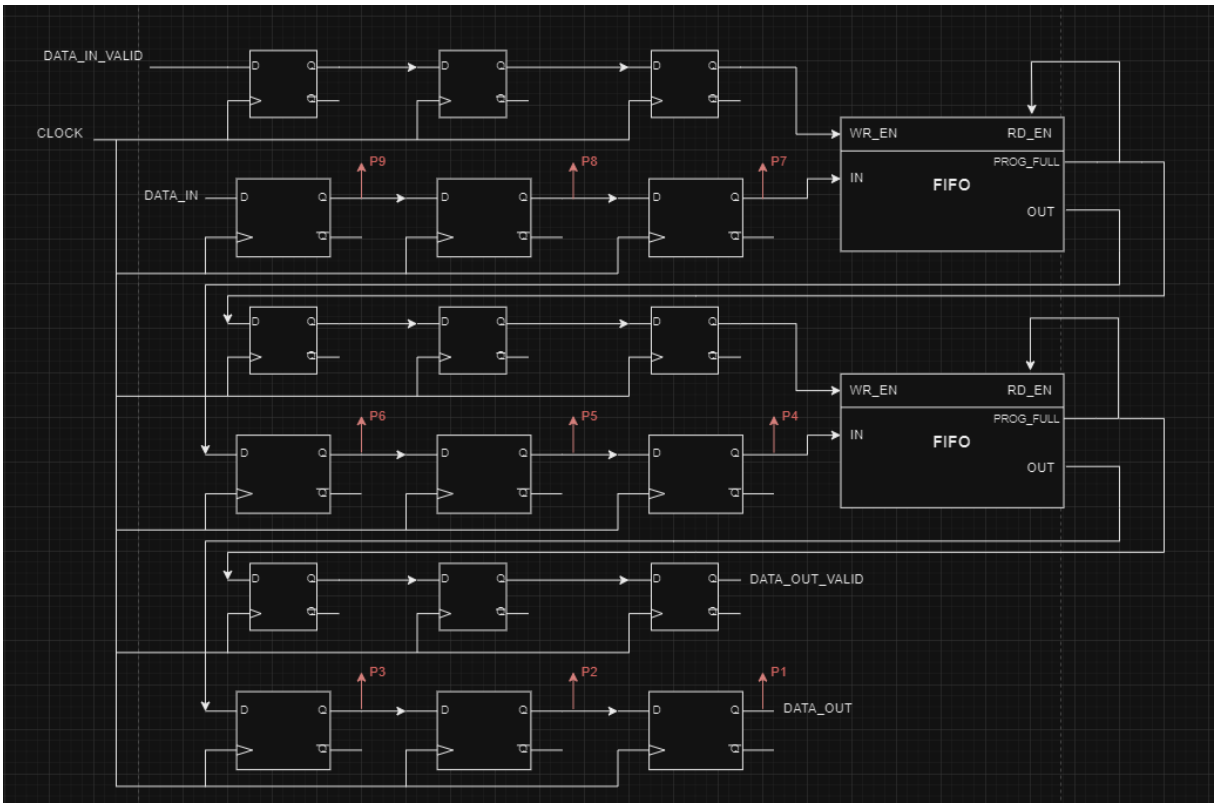

Figure 8 : Architecture ligne à retard

Dans cette architecture, les pixels peuvent être traités au moment voulu grâce à des temporisations, qui sont gérées par des FIFO et des bascules.

Le signal « PROG_FULL » est directement connecté à « RD_EN », ce qui permet de déclencher la lecture de la première FIFO lorsque les 128 premiers pixels (correspondant à la première ligne) sont arrivés. De la même manière, la lecture de la deuxième FIFO est activée lorsque les 256 premiers pixels (deuxième ligne) sont disponibles.

Analyse du code

delayed_line.vhd :

```
-------------------------------------------------
-- COMPONENTS INSTANTIATION
-------------------------------------------------

-- Wiring signals as expected in 2D_FILTER architecture
fifo1 : fifo_generator_0
            port map(clk => CLK,
                     rst => RST,
                     din => s_line_data_1(23 downto 16),
                     wr_en => line_valid_1(2),
                     rd_en => s_prog_full_1,
                     prog_full_thresh => PROG_FULL_THRESH,
                     dout=>data_out_fifo_1,
                     full=> s_full_1,
                     empty=> s_empty_1,
                     prog_full=> s_prog_full_1
                     );

fifo2 : fifo_generator_0
            port map(clk => CLK,
                     rst => RST,
                     din => s_line_data_2(23 downto 16),
                     wr_en => line_valid_2(2),
                     rd_en => s_prog_full_2,
                     prog_full_thresh => PROG_FULL_THRESH,
                     dout=>data_out_fifo_2,
                     full=> s_full_2,
                     empty=> s_empty_2,
                     prog_full=> s_prog_full_2
                     );
--
```

Figure 9 : delayed_line.vhd 1

Les FIFOs sont branchées exactement comment dans l'architecture présentée ci-dessus.

```
-------------------------------------------------
-- PROCESSING
-------------------------------------------------
    process(CLK,RST)
    begin
         if (RST='1') then -- Asynchronous reset
            -- Pixels value
            s_line_data_1<=(others => '0');
            s_line_data_2<=(others => '0');
            s_line_data_3<=(others => '0');
            -- Signal data valid delayed
            line_valid_1<=(others => '0');
            line_valid_2<=(others => '0');
            line_valid_3<=(others => '0');
            --
         else
            if(rising_edge(CLK))then
               if((s_full_1='1' and s_empty_1='1') or (s_full_2='1' and s_empty_2='1')) then
                    -- Pixels value
                    s_line_data_1<=(others => '0');
                    s_line_data_2<=(others => '0');
                    s_line_data_3<=(others => '0');
                    -- Signal data valid delayed
                    line_valid_1<=(others => '0');
                    line_valid_2<=(others => '0');
                    line_valid_3<=(others => '0');
                    --
               else -- Implementation of shift registers that replace flip-flops
                    -- Pixels value
                    s_line_data_1<=s_line_data_1(15 downto 0) & DATA_IN;
                    s_line_data_2<=s_line_data_2(15 downto 0) & data_out_fifo_1;
                    s_line_data_3<=s_line_data_3(15 downto 0) & data_out_fifo_2;
                    -- Signal data valid delayed
                    line_valid_1<=line_valid_1(1 downto 0) & DATA_IN_VALID;
                    line_valid_2<=line_valid_2(1 downto 0) & s_prog_full_1;
                    line_valid_3<=line_valid_3(1 downto 0) & s_prog_full_2;
                    --
                end if;
            end if;
         end if;
    end process;
end Behavioral;
```

Figure 10 : delayed_line.vhd 2

Ces registres à décalage (par exemple : « line_data_1 ») reproduisent exactement le comportement de plusieurs bascules en cascade. De plus, ils sont routés de la même manière que si des bascules avaient été générées par Vivado puis instanciées dans le code.

Au démarrage du programme, les FIFOs ne sont pas encore disponibles. Il faut attendre que leurs signaux « FULL » et « PROG_FULL » passent à zéro pour commencer à les utiliser, une condition est donc présente afin de forcer les données à zéro tant que celles-ci ne sont pas disponibles.

delayed_line_tb.vhd :

```
stimulus: process
 variable compteur : integer := 0;
begin
  RST<='1';
  DATA_IN<="00000000";
  DATA_IN_VALID<='0';
  PROG_FULL_THRESH<="0000000101";
  wait for 100 ns;
  RST<='0';
  wait for 200 ns;
  DATA_IN_VALID<='1';

  while (compteur /= 100) loop
    compteur:=compteur+1;
    DATA_IN<=std_logic_vector(to_unsigned(compteur,8));
    wait for 10ns;
  end loop;

  DATA_IN_VALID<='0';
  wait;

  stop_the_clock <= true;
  wait;
end process;
```

Figure 11 : delayed_line_tb.vhd

Ce testbench rentre, toutes les 10 ns, des valeurs provenant d'un compteur dans la ligne à retard.

Le signal « prog_full_thresh » est initialisé à 5 pour tester simplement qu'au bout de 10 pixels, le première FIFO se dépile et qu'au bout de 20 pixels, la deuxième se dépile aussi.

Simulation du code

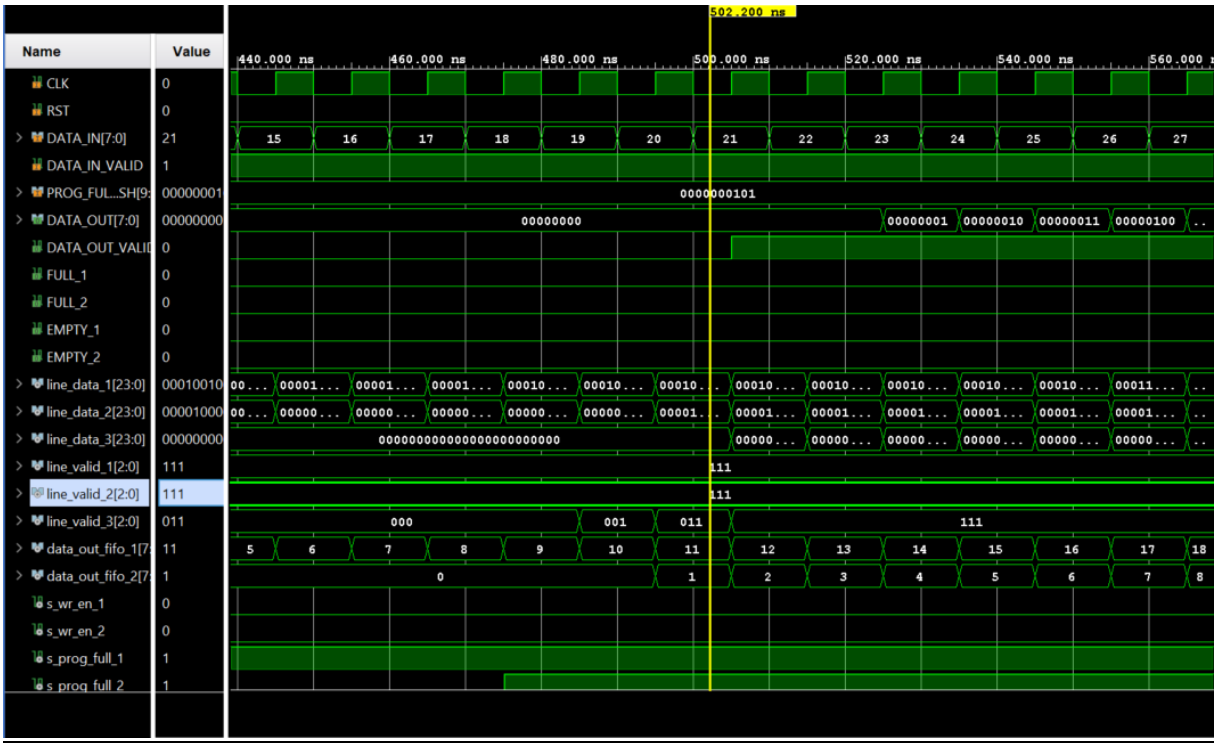

Figure 12 : Vérification simulation delayed_line_tb

- "prog_full_thresh" = 5 (ligne de 10 pixels)

Ce testbench montre que les pixels sortant des FIFOs sont alignés, il est maintenant possible de récupérer 9 pixels à chaque fois. Etant donné qu'aucun décalage n'est présent dans notre ligne à retard, elle va donc pouvoir être implémenté dans le filtre 2D.

Lien des fichiers :

- *Projet_TP.srcs/sources_1/new/delayed_line.vhd*
- *Projet_TP.srcs/sim_1/new/ delayed_line_tb.vhd*

4 : Implémentation filtre 2D + test complet

Introduction

L'objectif de cette partie est de piloter la ligne à retard avec une machine à états, afin de sortir les pixels des bascules et de les filtrer.

Différents outils seront mis en place, un premier pour combler les pixels manquants sur l'image résultante. Un deuxième qui vérifiera la différence entre l'image filtrée par le programme en VHDL et une image filtrée en MATLAB.

Analyse du code

top_filter2D_Average.vhd :

```
-- Implementation of a clock period counter
Counter_process : process(CLK,RST)
    begin
         if (RST='1') then
            counter<="000";
         else
            if(rising_edge(CLK))then
                if(reset_counter='1') then
                    counter<="000"; -- Counter reset
                else
                    counter<=STD_LOGIC_VECTOR(unsigned(counter) + to_unsigned(1,3)); -- Counter incrementation
                end if;
            end if;

         end if;
    end process;
```

Figure 13 : top_filter2D_Average.vhd 1

Implémentation d'un compteur sur 3 bits qui se réinitialise lorsque le signal « reset_counter » est égale à 1. Il permet de temporiser le signal « DATA_OUT_VALID » afin que le programme ait le temps de faire le calcul des pixels filtrés avant de les envoyer sur la sortie.

Afin de simplifier la compréhension de la machine à états du programme, le graphe ci-joint décrit les conditions de changement d'état. De plus, des commentaires sont présents dans le code afin de décrire toutes les subtilités de celle-ci.

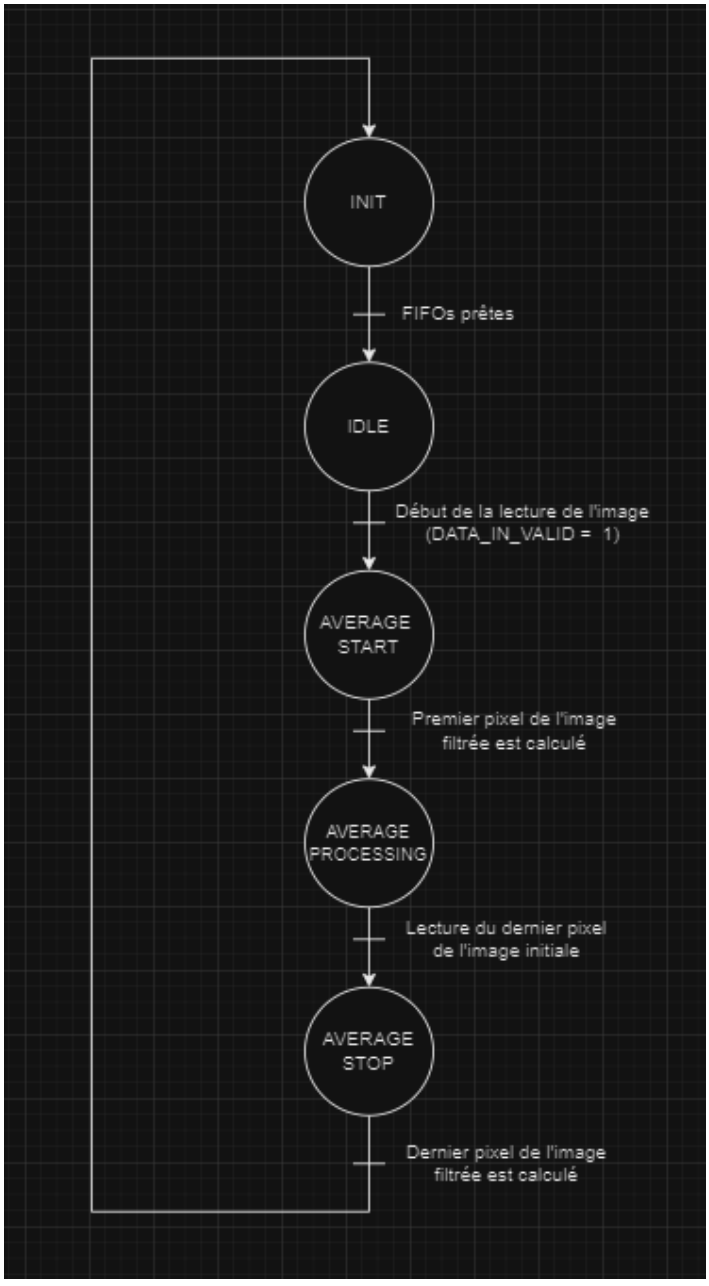

Figure 14 : Machine à états du filtre 2D

```
image_processing : process (CLK)
begin
    if(rising_edge(CLK))then
        if ( (FSM=AVERAGE_PROCESSING) or (FSM=AVERAGE_STOP) )then

            -- The pixels are added each others
            result_add_1_1<= unsigned('0' & s_line_data_1(23 downto 16)) + unsigned('0' & s_line_data_1(15 downto 8));
            result_add_1_2<= unsigned('0' & s_line_data_1(7 downto 0)) + unsigned('0' & s_line_data_2(23 downto 16));
            result_add_1_3<= unsigned('0' & s_line_data_2(7 downto 0)) + unsigned('0' & s_line_data_3(23 downto 16));
            result_add_1_4<= unsigned('0' & s_line_data_3(15 downto 8)) + unsigned('0' & s_line_data_3(7 downto 0));

            result_add_2_1<=('0' & result_add_1_1) + ('0' & result_add_1_2);
            result_add_2_2<= ('0' & result_add_1_3) + ('0' & result_add_1_4);

            result_add_3<= ('0' & result_add_2_1) + ('0' & result_add_2_2);
            --
        end if;
    end if;
end process;

output_processing : process (result_add_3,valid_output)
begin
    if(valid_output='1') then
        DATA_OUT<=STD_LOGIC_VECTOR(result_add_3(10 downto 3)); -- Pixels division by the normalisation factor
     else
        DATA_OUT<=(others => '0');
     end if;
 end process;
end Behavioral;
```

Figure 15 : top_filter2D_Average.vhd 2

Le premier process fait l'addition de tous les pixels, tout en respectant les bits d'overflow avec le résultat de l'addition sur un bit de plus.

Le deuxième process, quant à lui, divise le résultat par 8 pour renormaliser l'image avec des valeurs de pixels allant de 0 à 255. La condition sur « valid_output » permet de mettre la sortie à zéro quand aucun filtrage n'est en cours.

top_filter2D_Sobel.vhd et top_filter2D_Laplacian.vhd :

Pour les programmes « top_filter2D_Sobel » et « top_filter2D_Laplacian » quelques différences apparaissent. Etant donné, que ces deux filtres fonctionnent avec des coefficients, une étape de multiplication est présente dans le programme, exemple de « top_filter2D_Sobel » :

```
image_processing : process (CLK)
begin
    if(rising_edge(CLK))then
        if ( (FSM=AVERAGE_PROCESSING) or (FSM=AVERAGE_STOP) )then

            -- The pixels are multiplied by the coefficients of the filter matrix
            result_mult_1<=signed('0' & s_line_data_1(7 downto 0))* coef_9;
            result_mult_2<=signed('0' & s_line_data_1(15 downto 8))* coef_8;
            result_mult_3<=signed('0' & s_line_data_1(23 downto 16))* coef_7;
            result_mult_4<=signed('0' & s_line_data_2(7 downto 0))* coef_6;
            result_mult_5<=signed('0' & s_line_data_2(15 downto 8))* coef_5;
            result_mult_6<=signed('0' & s_line_data_2(23 downto 16))* coef_4;
            result_mult_7<=signed('0' & s_line_data_3(7 downto 0))* coef_3;
            result_mult_8<=signed('0' & s_line_data_3(15 downto 8))* coef_2;
            result_mult_9<=signed('0' & s_line_data_3(23 downto 16))* coef_1;

            -- The pixels are added each others
            result_add_1_1<= (result_mult_1(10) & result_mult_1) + (result_mult_2(10) & result_mult_2);
            result_add_1_2<= (result_mult_3(10) & result_mult_3) + (result_mult_4(10) & result_mult_4);
            result_add_1_3<= (result_mult_5(10) & result_mult_5) + (result_mult_6(10) & result_mult_6);
            result_add_1_4<= (result_mult_7(10) & result_mult_7) + (result_mult_8(10) & result_mult_8)
                             + (result_mult_9(10) & result_mult_9);

            result_add_2_1<=(result_add_1_1(11) & result_add_1_1) + (result_add_1_2(11) & result_add_1_2);
            result_add_2_2<= (result_add_1_3(11) & result_add_1_3) + (result_add_1_4(11) & result_add_1_4);

            result_add_3<= (result_add_2_1(12) & result_add_2_1) + (result_add_2_2(12) & result_add_2_2);
            --
        end if;
    end if;
end process;

output_processing : process (result_add_3,valid_output)
begin
    if(valid_output='1') then
        DATA_OUT<=STD_LOGIC_VECTOR(abs(result_add_3(10 downto 3))); -- Absolute value of pixels and division by the normalisation factor
     else
        DATA_OUT<=(others => '0');
     end if;
 end process;
end Behavioral;
```

Figure 16 : top_filter2D_Sobe.vhd

Dans les multiplications, il faut ajouter un neuvième bit égal à zéro à la valeur des pixels pour éviter que celle-ci soit considérée comme négative. Mais aussi, dans les additions, une extension de bit est requise afin de garder le signe de la valeur des pixels.

Avant de transmettre les pixels filtrés à la sortie, il faut faire la valeur absolue de ceux-ci et les multiplier par le facteur de normalisation (Sobel=1/8, Laplacian=1/4).

Ces calculs nécessitent un cycle d'horloge supplémentaire, il faut donc que le compteur contrôlant le signal « valid_output » se réinitialise une période plus tard (incrémentation de la limite du compteur) par rapport à avant.

tb_lena_processing:

```
p_read : process
 FILE vectors : text;
 variable Iline : line;
 variable I1_var :std_logic_vector (7 downto 0);

   begin
   DATA_AVAILABLE <= '0';
   RST<='1';
   s_prog_full_thresh<="0001111011";
   file_open (vectors,"Lena128x128g_8bits.dat", read_mode);
   wait for 100 ns;
   RST<='0';
   wait for 300 ns;
   while not endfile(vectors) loop
     readline (vectors,Iline);
     read (Iline,I1_var);

     I1 <= I1_var;
     pixel_in<=pixel_in+1;
     DATA_AVAILABLE <= '1';
     wait for 10 ns;
   end loop;
   DATA_AVAILABLE <= '0';
   wait for 1 us;
   file_close (vectors);
   wait;
end process;
```

Figure 17 : tb_lena_processing.vhd 1

Ce process vient lire l'image de Lena originale, et toutes les 10 ns, rentre la valeur des pixels dans le signal « I1 ». Le signal « pixel_in » permet de compter le nombre de pixels qui sont rentrés. « DATA_AVAILABLE » (DATA_IN_VALID) est mis à 1 pendant toute la lecture.

```
p_write: process
  file results : text;
  variable OLine : line;
  variable O1_var :std_logic_vector (7 downto 0);

    begin
    file_open (results,"Lena128x128g_8bits_filtered.dat", write_mode);
    wait for 10 ns;
    wait until s_data_out_valid = '1';
    wait for 10 ns;

    while s_data_out_valid ='1' loop
      write (Oline, O1, right, 2);
      writeline (results, Oline);
      pixel_out<=pixel_out+1;
      wait for 10 ns;
    end loop;
    wait for 1 us;
    file_close (results);
    wait;
 end process;
--Instancier composant filtre à place de la simple recopie entre I1 vers sortie O1 (top)
s_data_in<= I1;
O1<=s_data_out;
s_data_in_valid<=DATA_AVAILABLE;
```

Figure 18 : tb_lena_processing.vhd 2

Le deuxième process vient ouvrir le fichier où l'image filtrée sera stockée et attend que le premier pixel filtré sorte du filtre (« s_data_out_valid » à 1). Ensuite, toutes les 10 ns, il écrit les pixels filtrés dans le fichier résultant. Un signal de comptage des pixels sortant du filtre est aussi présent («pixel_out»).

Au niveau du câblage, les pixels lus par le process read sont envoyés dans l'entrée du filtre et les pixels sortant du filtre sont transmis au process write.

Simulation du code

Avant d'avoir un filtre opérationnel, certaines modifications ont dû être faites.

Au début, il y avait un léger décalage sur l'image résultante, il a donc fallu jouer sur la valeur de « PROG_FULL_THRESH » pour sortir les pixels au bon moment.

Ensuite, le programme générait un pixel en excès, ce qui a nécessité une réduction de la valeur limite du compteur d'arrêt du filtre.

```
when AVERAGE_STOP =>
    -- Waiting last addition finishes before set valid_output at low level
    if(counter="011")then
        reset_counter<='1'; -- Counter stoped
        valid_output<='0';
        FSM <= INIT;
    end if;
```

Figure 19 : Valeur limite du compteur d'arrêt du filtre

Vérification :

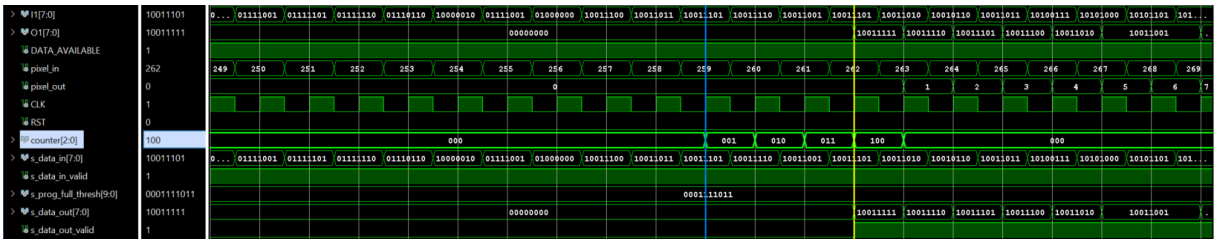

Figure 20 : Vérification simulation tb_lena_processing.vhd 1

Le système est bien synchronisé, car dès l'arrivée du 259e pixel (3e pixel de la 3e ligne), le processus de filtrage commence. En effet, le compteur débute son incrémentation et trois cycles d'horloge après (une fois les calculs du premier nouveau pixel terminés), le signal « s_data_out_valid » passe à 1 pour indiquer le commencement de la disponibilité des nouveaux pixels.

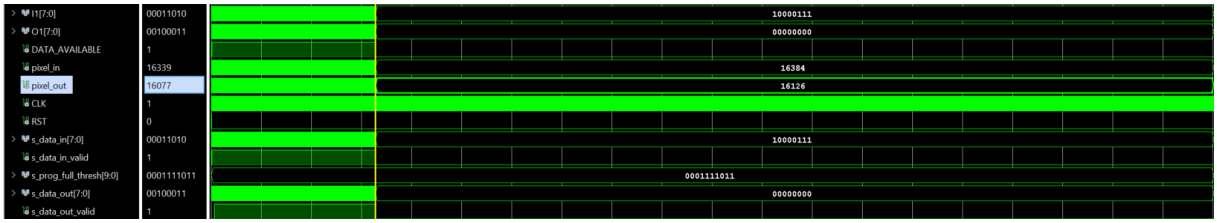

Figure 21 : Vérification simulation tb_lena_processing.vhd 2

Le compteur « pixel_in » est égale à 16384 (=128*128), cela implique que tous les pixels de l'image originale rentrent correctement dans le filtre.

Le compteur « pixel_out » est égale à 16126 (=(128*126) - 2), cela implique que le bon nombre de pixels filtrés sortent du filtre.

Outils

1 Python :

```python
import shutil

file_path = '../Projet_TP.sim/sim_1/behav/xsim/Lena128x128g_8bits_filtered.dat'

''' Reading content of the file '''
with open(file_path, 'r') as file:
    lines = file.readlines()

''' Adding 258 pixels, all set to 0, 129 at the bottom and 129 at the top of the image '''
zero_lines = ['00000000\n'] * 129
lines = zero_lines + lines + zero_lines

''' Overwrites the old image with the new padded image '''
with open(file_path, 'w') as file:
    file.writelines(lines)

''' Copy the new padded image into "tools" directory '''
shutil.copy(file_path, './')
```

Figure 22 : Programme Python

Ce programme permet d'ajouter des pixels ayant la valeur de zéro en haut et en bas de l'image filtrée. Cela est nécessaire, car le filtre 2D ne peut pas faire la moyenne des pixels de la première et de la dernière ligne, la solution est donc de combler cette image avec des pixels qui sont égaux à zéro.

2 MATLAB :

```
clearvars;
close all;

% Define convolution kernel (several options provided)
% kernel = [0 0 0 ; 0 1 0 ; 0 0 0]; % Identity filter (delta)
% kernel = [1 1 1 ; 1 0 1 ; 1 1 1] * 1/8; % Blur filter
 kernel = [-1 0 1 ; -2 0 2 ; -1 0 1] * 1/8; % Sobel-Y
% kernel = [-1 -2 -1 ; 0 0 0 ; 1 2 1] * 1/8; % Sobel-X
% kernel = [0 -1 0 ; -1 4 -1 ; 0 -1 0] * 1/4; % Laplacian

% Load grayscale image data from binary file
[binary_data] = textread('Lena128x128g_8bits.dat', '%s');
decimal_values = bin2dec(binary_data);
original_image = uint8(reshape(decimal_values, [128, 128]))';

% Initialize output image to apply filter
filtered_image = zeros(size(original_image));

% Perform manual convolution
for i = 2:size(original_image, 1) - 1
    for j = 2:size(original_image, 2) - 1
        local_patch = double(original_image(i - 1:i + 1, j - 1:j + 1));
        conv_result = local_patch .* kernel;
        filtered_image(i, j) = sum(conv_result(:));
    end
end

% Load filtered image generated by VHDL
[vhdl_data] = textread('Lena128x128g_8bits_filtered.dat', '%s');
vhdl_decimal = bin2dec(vhdl_data);
vhdl_output_image = uint8(reshape(vhdl_decimal, [128, 128]))';

% Convert to uint8 for display purposes
filtered_image = uint8(abs(filtered_image));

% Calculate difference between MATLAB and VHDL results
difference_image = vhdl_output_image - filtered_image;

% Display images for comparison
figure;
% Increased figure size for larger display
set(gcf, 'Position', [200, 200, 4800, 1600]);
subplot(1, 4, 1), imshow(original_image, []); title('LENA');
subplot(1, 4, 2), imshow(filtered_image, []); title('MATLAB Result');
subplot(1, 4, 3), imshow(vhdl_output_image, []); title('VHDL Result');
subplot(1, 4, 4), imshow(difference_image, []); title('Difference');
```

Figure 23 : Programme MATLAB

Ce programme permet, à l'aide des librairies MATLAB, de transformer une image avec le filtre souhaité (Moyenne, Sobel ou Laplacian) et de comparer cette image filtrée à l'image sortant de notre filtre 2D.

Par ailleurs, il calcule aussi la différence entre « l'image MATLAB » et « l'image VHDL » et affiche ces différences sous la forme d'une matrice de 128 par 128 pixels. Cela permet de vérifier si le programme VHDL fonctionne correctement, qu'il n'y a pas de décalage ou d'erreur avec les additions et multiplications de pixels.

Remarque : ce code est basé sur un programme existant trouvé sur un dépôt GitHub en ligne, il a cependant été modifié pour être adapté à mon projet.

Résultats

Filtre moyenneur :

Figure 24 : Filtre moyenneur

Filtre Sobel :

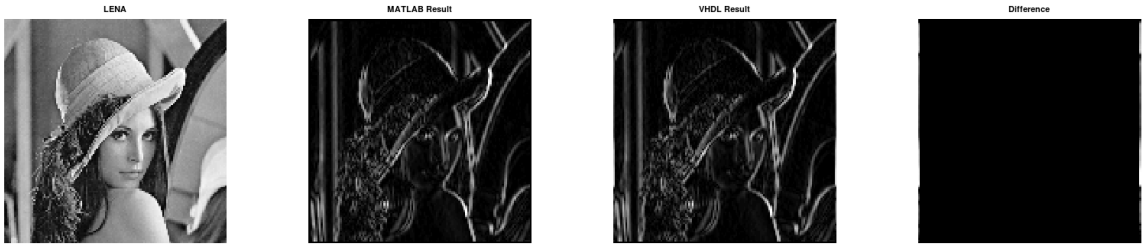

Figure 25 : Filtre Sobel

Filtre Laplacien :

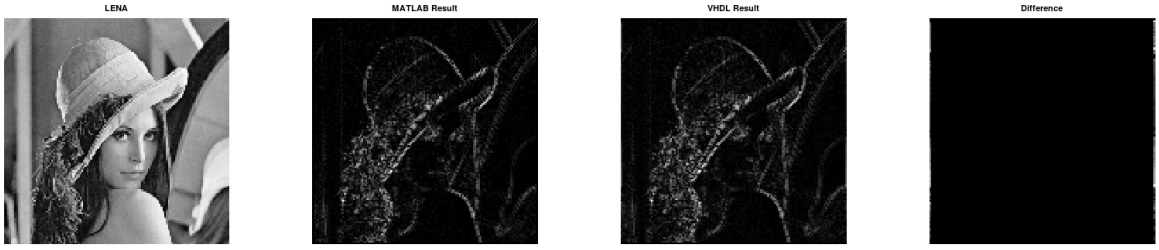

Figure 26 : Filtre Laplacien

Pour chaque filtre, l'image de « différence » montre que « l'image MATLAB » et « l'image VHDL » sont pareilles, excepté les bords à droite et à gauche de l'image.

Lien des fichiers :

- *Projet_TP.srcs/sources_1/new/top_filter2D_Average.vhd*
- *Projet_TP.srcs/sources_1/new/top_filter2D_Sobel.vhd*
- *Projet_TP.srcs/sources_1/new/top_filter2D_Laplacian.vhd*
- *Projet_TP.srcs/sim_1/new/ tb_lena_processing.vhd*
- *tools/padded_lena.py*
- *tools/dat2img*

5 : Mise en place filtre 2D sur la carte NEXYS 4

Contexte et limitations

A cause de problèmes de compilation sur Vivado (routage infini) et d'un manque de temps, cette partie n'a malheureusement pas pu être réalisée.

CONCLUSION GÉNÉRALE

Ce TP nous a permis de renforcer nos compétences dans le développement VHDL sur cible FPGA. Il nous a appris à générer et à brancher des composants entre eux (ligne à retard) ainsi qu'à contrôler celle-ci grâce à une machine à états.

Mais également, à tester nos développements de manière précise en utilisant des simulations sur Vivado, et à vérifier les résultats à l'aide d'outils de programmation tels que Python et MATLAB dans mon cas.

D'autres améliorations peuvent être envisagées, comme l'ajout de signaux génériques permettant de configurer la ligne de retard en fonction de la taille de l'image et de la matrice du filtre (nombre de bascules, nombre et taille des FIFOs). Ou encore, un processus pourrait être intégré pour définir les pixels des bords de l'image à zéro.