TP d'architectures reconfigurables
Réalisation d'un filtre 2D

Firmin Launay

Semestre 7 – Objets connectés (IoT)

Octobre 2024

Au cours des vingt heures de TP d'architectures reconfigurables, l'objectif était de réaliser un filtre à images sur FPGA, capable d'appliquer différentes méthodes de filtrage.

Les quatre méthodes retenues sont le moyennage, la détection de contours verticale, horizontale et enfin complète à l'aide de matrices de voisinage (type filtre de Sobel).

Remarque Le code source du filtre réalisé est disponible publiquement sur ce dépôt Git, pour un aperçu plus détaillé :

`https://github.com/Firmin-ESIREM/fpga-2d-filter.git`

1 Structure

L'objectif était d'obtenir une entité « filtre ». Celui-ci a été réalisé de manière assez générique. En effet, bien que des tests aient uniquement été effectués avec une image de 256×256 pixels, celui-ci prend en entrée des paramètres de largeur et de hauteur de l'image.

La structure de l'entité est présentée en figure 1, telle que documentée dans le code.

```
entity filter is
    Port ( data_entry: IN STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Each pixel
     ↪ should be provided individually, line after line, on each clock
     ↪ rise, starting one clock rise after enable.
           data_output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  -- Each pixel of
     ↪  the provided image will be output individually, with a shift of {
     ↪ (width * 2) + 9 } clock cycles from the entry.
           valid_output: OUT STD_LOGIC;  -- This is set to 1 when the
     ↪ filter is currently outputing some valid pixels to the data_output
     ↪ bus.
           image_width: IN STD_LOGIC_VECTOR(9 DOWNTO 0);  -- This value is
     ↪ read when enable is activated, it is ignored passed that point.
           image_height: IN STD_LOGIC_VECTOR(9 DOWNTO 0);  -- This value is
     ↪  read when enable is activated, it is ignored passed that point.
           filter_type: IN STD_LOGIC_VECTOR(1 DOWNTO 0);  -- This should be
     ↪  set to "00" for an averaging filter, to "01" for a vertical
     ↪ contour-detecting filter, to "10" for a horizontal contour-
     ↪ detecting filter, or to "11" for a full contour-detecting filter.
     ↪ This value should not change while you have not at least provided
     ↪ every pixel.
           enable: IN STD_LOGIC;  -- This should be set to '1' when the
     ↪ image's width and height are provided, and should not be set back
     ↪ to '0' before you have retreived the whole output image. It should
     ↪ be set back to '0' before proceeding with another image.
           clock: IN STD_LOGIC;
           reset: IN STD_LOGIC
         );
end filter;
```

FIGURE 1 – Structure externe de l'entité « filter »

2 Ligne à retard

2.1 Conception

Avant de commencer à réellement filtrer les images, il est nécessaire d'analyser les besoins en mémoire du FPGA pour ce faire.

Les filtrations se font pixel par pixel, en se reposant uniquement sur les pixels immédiatement voisins. Il est donc nécessaire de stocker deux lignes complètes de l'image, et trois pixels supplémentaires. La solution la plus simple est d'utiliser neuf bascules D et deux fifos (d'une capacité de $L - 3$ mots de 8 bits, avec L la largeur de l'image en pixels) pour « retenir » ces données.

On crée donc ainsi une sorte de « ligne à retard », schématisée en figure 2.

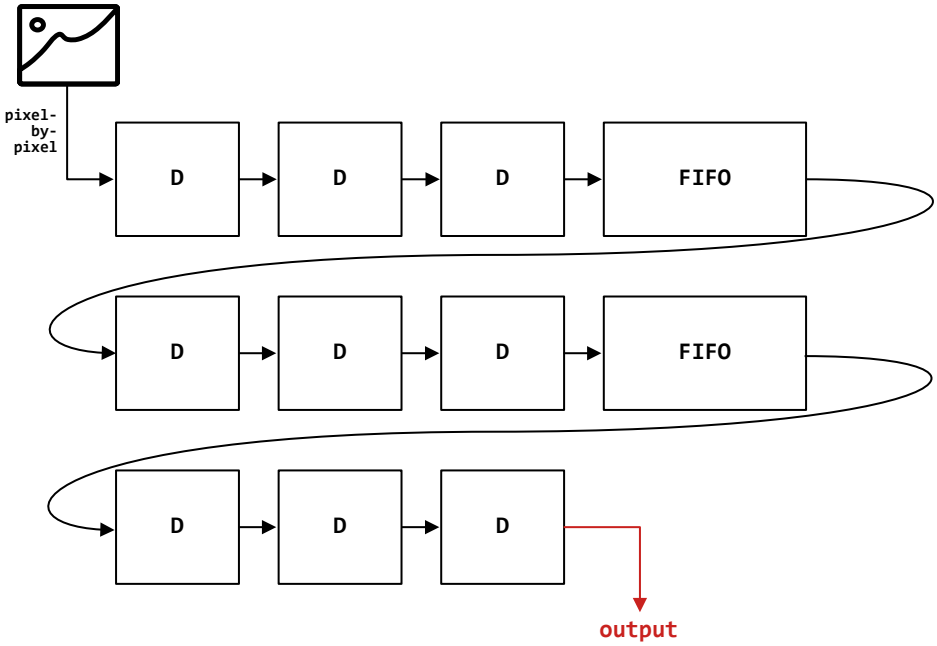


FIGURE 2 – Schéma de fonctionnement de la ligne à retard

2.2 Implémentation

Pour ce faire, on crée donc des bascules D et des FIFO, qui vont faire « circuler » les pixels d'une case à l'autre d'un *array* bidimensionnel, comme celui déclaré en figure 3.

```
1 type t_std_logic_vector8_3_3 is array (0 to 3, 0 to 2) of STD_LOGIC_VECTOR
      ↪ (7 DOWNTO 0);
2 signal d_s: t_std_logic_vector8_3_3;
```

FIGURE 3 – Tableau bidimensionnel où circulent les pixels de l'image

Il suffit ensuite de générer nos FIFO et nos bascules D, en réglant les entrées et sorties de sorte à ce que les pixels circulent bien, comme en figure 4.

```
fifos:
    for i in 0 to 1 generate
       fifo_x: fifo
            PORT MAP ( clk              => clock,
                       rst              => reset_fifos_and_latches,
                       din              => d_s(3, i),
                       wr_en            => enable_fifo(i),
                       rd_en            => prog_full_s(i),
                       prog_full_thresh => prog_full_thresh_s,
                       dout             => d_s(0, i+1),
                       full             => full_s(i),
                       empty            => empty_s(i),
                       prog_full        => prog_full_s(i)
                     );
end generate fifos;

latches_column:
    for i in 0 to 2 generate
        latches_line:
            for j in 0 to 2 generate
                latch_y: d_latch
                    PORT MAP ( D     => d_s(i, j),
                               Q     => d_s(i+1, j),
                               CLK   => clock,
                               EN    => enable,
                               RESET => reset_fifos_and_latches
                             );
        end generate latches_line;
end generate latches_column;
```

FIGURE 4 – Génération des bascules D et des FIFO génériques

Dès lors, on retrouve l'image d'entrée, ressortie dans son intégralité avec un simple retard.

3 Traitement de l'image

3.1 Principe

Une fois que les $(L \times 2) + 3$ pixels sont entrés en mémoire, on dispose toujours d'un carré de neuf pixels adjacents dans les bascules D. C'est le pixel de la bascule « centrale » (dans le référentiel de la figure 2) que l'on cherche à traiter.

Le choix effectué dans cette version est de ressortir « tels quels » les pixels ne pouvant pas être traités par cette méthode (en substance, la première et la dernière ligne).

Afin de ressortir les pixels traités en continuité des pixels non traités, sans avoir de « trou », il est nécessaire de créer une ligne à retard parallèle (en pratique, un FIFO) entre la bascule D « centrale » et l'output.

La méthode choisie pour mettre en place de cette solution nécessite de suivre le nombre de cycles d'horloge (et ainsi le « nombre de circulation effectuées ») depuis l'entrée du premier pixel. Lorsqu'on arrive au cycle d'horloge garantissant la présence d'un pixel traité au bout de notre ligne à retard parallèle, on bascule la sortie du composant depuis la sortie de la ligne à retard

principale vers la sortie de cette ligne parallèle.

3.2 Algorithme de traitement

L'idée était d'implémenter un filtre moyenneur, deux filtres de détection de contours Sobel (vertical et horizontal), et un filtre de détection de contours Laplacien.

Tous ces filtres peuvent aisément être décrits sous forme de matrices (respectivement, M_m, M_{Sv}, M_{Sh} et $M_{\mathcal{L}}$ en équation 1).

$$M_m = \begin{bmatrix} 1/8 & 1/8 & 1/8 \\ 1/8 & 0 & 1/8 \\ 1/8 & 1/8 & 1/8 \end{bmatrix}, \quad M_{Sv} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix},$$
$$M_{Sh} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad M_{\mathcal{L}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{1}$$

On peut donc implémenter l'algorithme de traitement matriciel de la figure 5, pour réutiliser un maximum de code. Celui-ci s'appuie sur les filtres matriciels définis en figure 6.

```
result := 0;
for i in 0 to 2 loop
    for j in 0 to 2 loop
        result := result + ( to_integer( unsigned(d_s(i, j)) ) *
    ↪ filter_matrices(to_integer(unsigned(filter_type)))(i, j) );
    end loop;
end loop;
result := abs(result) / filter_factors(to_integer(unsigned(filter_type)));
filtered_pixel <= std_logic_vector(to_unsigned(result, 8));
```

FIGURE 5 – Algorithme de traitement matriciel

```
type filter_matrix is array (0 to 2, 0 to 2) of integer;
type filter_matrices_container is array (0 to 3) of filter_matrix;
type filter_factors_container is array (0 to 3) of integer;

-- BE CAREFUL WHEN IMPLEMENTING A NEW FILTER MATRIX: Each element of the
    ↪ array represents a COLUMN, NOT A LINE!

variable averaging_matrix: filter_matrix := ((1, 1, 1), (1, 0, 1), (1, 1,
    ↪ 1));

variable vertical_contour_matrix: filter_matrix := ((-1, -2, -1), (0, 0, 0)
    ↪ , (1, 2, 1));

variable horizontal_contour_matrix: filter_matrix := ((-1, 0, 1), (-2, 0,
    ↪ 2), (-1, 0, 1));

variable full_contour_matrix: filter_matrix := ((0, -1, 0), (-1, 4, -1),
    ↪ (0, -1, 0));

variable filter_matrices: filter_matrices_container := (averaging_matrix,
                                                        vertical_contour_matrix,
                                                      horizontal_contour_matrix,
                                                           full_contour_matrix);

variable filter_factors: filter_factors_container := (8,
                                                      1,
                                                      1,
                                                      1);
```

FIGURE 6 – Déclaration des matrices de filtrage

3.3 Résultats

En sortie, on obtient bien une image traitée.

À l'aide d'un simple *test bench*, on peut tester nos filtres sur l'image de test standard *Lenna*[1], en nuances de gris, comme en figure 7.

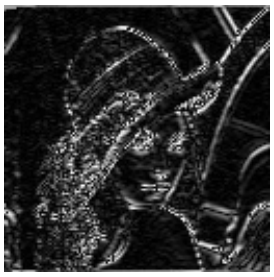
FIGURE 7 – Image de test standard Lenna

On obtient, pour chacun des filtres implémentés, les résultats de la figure 8.

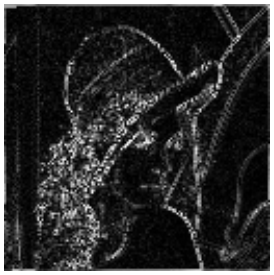
1. https://en.wikipedia.org/wiki/Lenna

(a) Lenna après passage dans le filtre moyenneur

(b) Lenna après passage dans le filtre Sobel de détection de contours verticaux



(c) Lenna après passage dans le filtre Sobel de détection de contours horizontaux



(d) Lenna après passage dans le filtre Laplacien de détection de contours

FIGURE 8 – Image de test Lenna après passage dans les différents filtres

4 Conclusion

En conclusion, ce projet a permis de créer un projet relativement complexe en VHDL, et de mettre à profit les compétences acquises au cours du semestre, en utilisant des structures imbriquées, des FIFO, ...

Celui-ci pourrait encore être amélioré en divisant la structure sous forme de plus petites

entités notamment. La lisibilité du code en serait grandement améliorée. L'implémentation d'une machine à états aurait également pu être une bonne option pour améliorer cette lisibilité.

Par manque de temps, ce projet n'a pas pu être implémenté sur une carte comportant un FPGA, comme la Digilent Nexys 4, précédemment utilisée en cours.