

**Compte-Rendu Projet filtre 2D**

**Table des matières**

[INTRODUCTION GÉNÉRALE 4](#_Toc182219504)

[1 : Duplication de l’image Lena 5](#_Toc182219505)

[2 : Génération FIFO + test unitaire 8](#_Toc182219506)

[3 : Implémentation ligne à retard + test unitaire 10](#_Toc182219507)

[4 : Implémentation filtre 2D + test complet 15](#_Toc182219508)

[5 : Mise en place filtre 2D sur la carte NEXYS 4 24](#_Toc182219509)

[CONCLUSION GÉNÉRALE 25](#_Toc182219510)

**Table des figures**

[Figure 1 : Image de Léna 5](#_Toc182219478)

[Figure 2 : Process read 6](#_Toc182219479)

[Figure 3 : Process write 6](#_Toc182219480)

[Figure 4 : Simulation de la copie de Léna 7](#_Toc182219481)

[Figure 5 : Architecture de la FIFO 8](#_Toc182219482)

[Figure 6 : Test de la FIFO 8](#_Toc182219483)

[Figure 7 : fifo\_tb.vhd 9](#_Toc182219484)

[Figure 8 : Architecture ligne à retard 10](#_Toc182219485)

[Figure 9 : delayed\_line.vhd 1 11](#_Toc182219486)

[Figure 10 : delayed\_line.vhd 2 12](#_Toc182219487)

[Figure 11 : delayed\_line\_tb.vhd 13](#_Toc182219488)

[Figure 12 : Vérification simulation delayed\_line\_tb 14](#_Toc182219489)

[Figure 13 : top\_filter2D\_Average.vhd 1 15](#_Toc182219490)

[Figure 14 : Machine à états du filtre 2D 16](#_Toc182219491)

[Figure 15 : top\_filter2D\_Average.vhd 2 17](#_Toc182219492)

[Figure 16 : top\_filter2D\_Sobe.vhd 18](#_Toc182219493)

[Figure 17 : tb\_lena\_processing.vhd 1 19](#_Toc182219494)

[Figure 18 : tb\_lena\_processing.vhd 2 19](#_Toc182219495)

[Figure 19 : Valeur limite du compteur d’arrêt du filtre 20](#_Toc182219496)

[Figure 20 : Vérification simulation tb\_lena\_processing.vhd 1 20](#_Toc182219497)

[Figure 21 : Vérification simulation tb\_lena\_processing.vhd 2 20](#_Toc182219498)

[Figure 22 : Programme Python 21](#_Toc182219499)

[Figure 23 : Programme MATLAB 22](#_Toc182219500)

[Figure 24 : Filtre moyenneur 23](#_Toc182219501)

[Figure 25 : Filtre Sobel 23](#_Toc182219502)

[Figure 26 : Filtre Laplacien 23](#_Toc182219503)

# 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**

A screenshot of a computer program

Description automatically generated

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.

A screen shot of a computer code

Description automatically generated

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**

A screen shot of a computer

Description automatically generated

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**

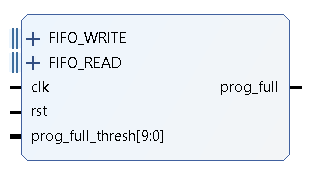
****

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 :

A green and black grid

Description automatically generated

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 :

A screenshot of a computer program

Description automatically generated

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**

A diagram of a computer

Description automatically generated

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 :

A screenshot of a computer program

Description automatically generated

Figure 9 : delayed\_line.vhd 1

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

A screenshot of a computer code

Description automatically generated

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 :

A screenshot of a computer code

Description automatically generated

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**

A screenshot of a computer

Description automatically generated

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 :

A computer screen shot of a computer code

Description automatically generated

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.

**A diagram of a process

Description automatically generated**

Figure 14 : Machine à états du filtre 2D

A screenshot of a computer program

Description automatically generated

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 » :

A screenshot of a computer

Description automatically generated

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:

A screenshot of a computer program

Description automatically generated

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.

A screenshot of a computer code

Description automatically generated

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.

A white background with black text

Description automatically generated

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

**Vérification :**

A screenshot of a computer

Description automatically generated

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.

A green line on a black background

Description automatically generated

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 :**

A computer screen with text and images

Description automatically generated

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.

1. **MATLAB :**

A screenshot of a computer program

Description automatically generated

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 :

A collage of a person wearing a hat

Description automatically generated

Figure 24 : Filtre moyenneur

Filtre Sobel :

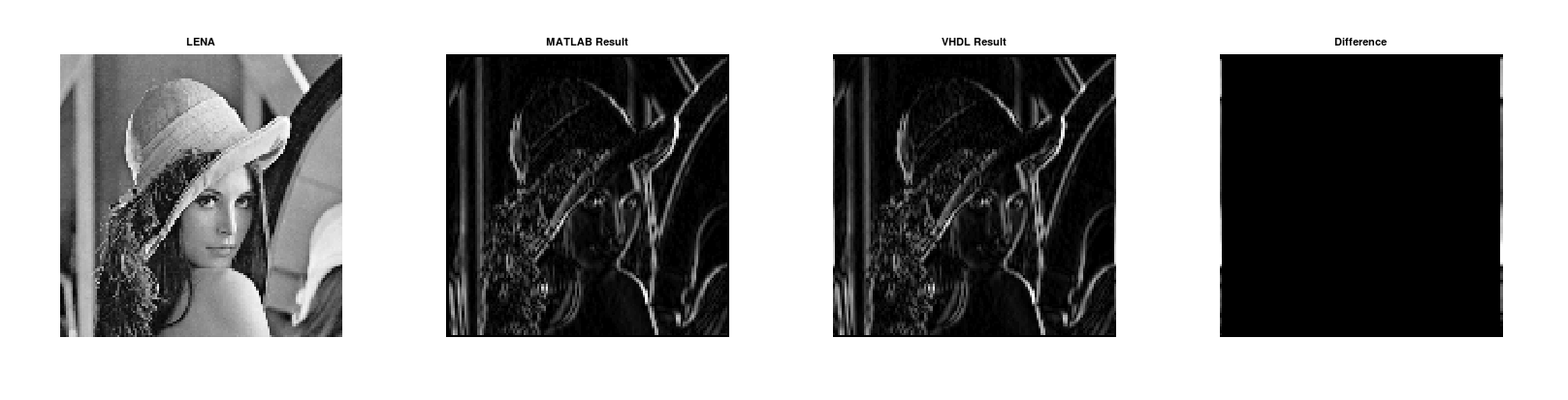


Figure 25 : Filtre Sobel

Filtre Laplacien :

**A close-up of a black and white image

Description automatically generated**

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.