PROJECTVHDLFILTRE2D

Rapport

JANUARY 14, 2025

MAMBO MOTSOU JUNIOR DEOGRACIAS

I. Introduction

1. Objectif du projet

L'objectif de ce projet est de concevoir, valider et implémenter un filtre 2D destiné au traitement d'images, avec une implémentation sur une carte d'évaluation Nexys4 DDR. Ce filtre doit permettre d'appliquer différents types de traitements sur des images, comme le flou, la détection de contours ou encore le lissage pour réduire le bruit.

Les deux grandes étapes de ce projet sont les suivantes :

- **Conception et validation d'un filtre 2D personnalisable** sous forme d'un composant IP (Intellectual Property). Cela inclut la création des modules nécessaires, leur simulation, ainsi que leur optimisation pour un fonctionnement efficace.
- **Implémentation et validation sur une carte FPGA Nexys4 DDR**. Cette étape implique l'intégration du filtre dans une chaîne fonctionnelle et sa mise à l'épreuve avec des images réelles.

2. Contexte

Les filtres 2D jouent un rôle crucial dans le domaine du traitement d'images, notamment dans les systèmes embarqués où l'optimisation des ressources et des performances est essentielle. Ces filtres sont utilisés dans de nombreuses applications, telles que :

- **Le flou (blur)** : pour créer des effets visuels ou réduire les détails inutiles.
- **La détection de contours** : pour identifier les éléments clés d'une image, comme dans les systèmes de vision par ordinateur.
- **Le lissage** : pour éliminer le bruit dans les images, ce qui est essentiel dans les capteurs embarqués soumis à des conditions de mesure bruitées.

Dans le cadre de ce projet, la carte FPGA Nexys4 DDR a été choisie en raison de ses capacités avancées, notamment son FPGA Artix-7, ses nombreuses entrées/sorties configurables, et ses outils de développement compatibles comme Vivado de Xilinx. Ces caractéristiques en font une plateforme idéale pour le développement de solutions embarquées innovantes.

L'objectif final est d'obtenir un filtre 2D fonctionnel, optimisé et adaptable, capable de traiter des images en temps réel avec une efficacité maximale, tout en respectant les contraintes de ressources liées à l'architecture FPGA.

II. Pré-conception et Tests Unitaires

1. Cache Mémoire

- **Objectif :** Tester les modules de base nécessaires pour la gestion des données d'entrée.

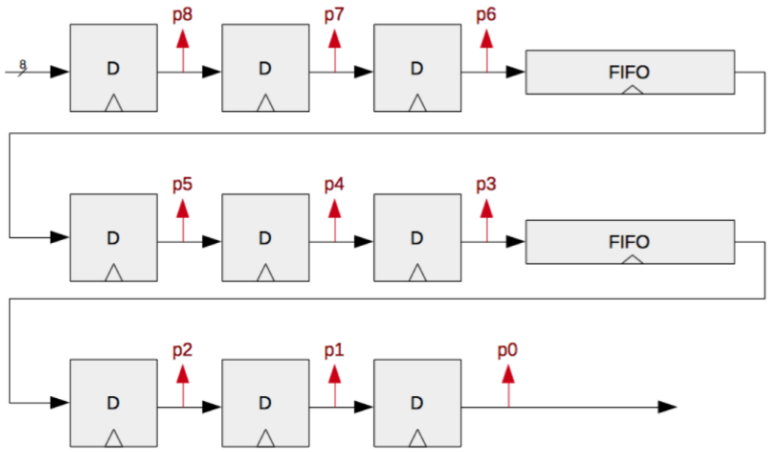

Figure 1:Cache Memory

Le cache mémoire est conçu pour permettre l'accès simultané aux pixels, rendant ainsi possible l'accès à un voisinage de 3x3 pixels en un cycle d'horloge. La structure repose sur des registres à bascule et mémoire FIFO.

1.1 Flip-Flop

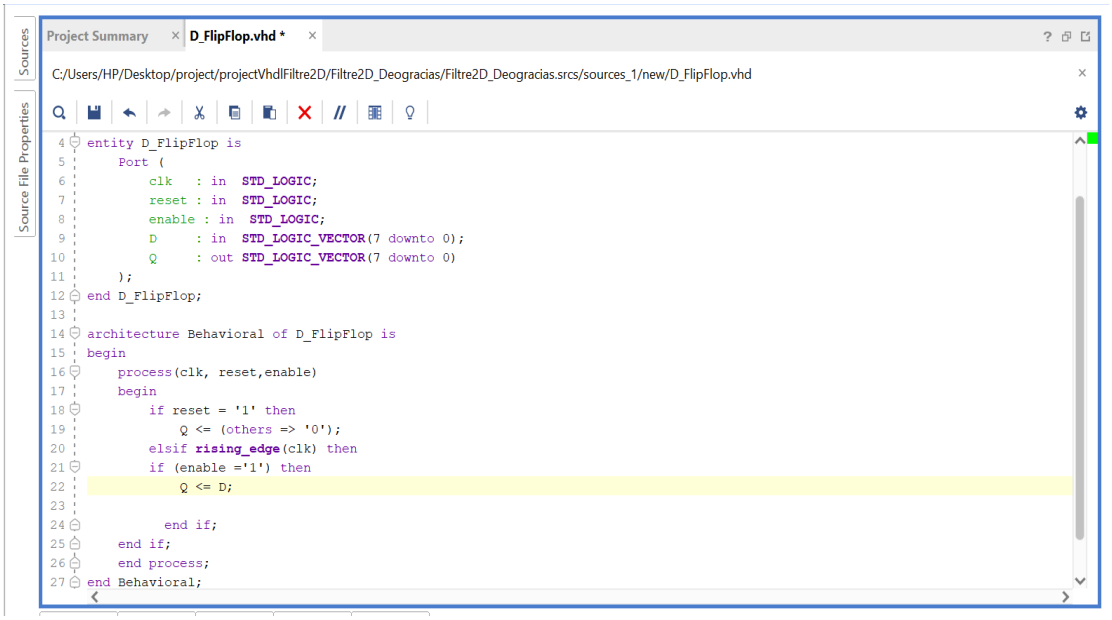

Figure 2: FlipFlop module

Aucun autre test n'a été réalisé sur le flip-flop, car il s'agit du même composant utilisé lors des TP et TD en classe. Ces validations précédentes ont confirmé son fonctionnement correct,

notamment en termes de stockage des bits individuels et de synchronisation avec les signaux de contrôle comme le reset et l'enable.

1.2 FIFO (First-In-First-Out)

Le professeur nous a montré comment générer une FIFO dans Vivado en utilisant les bibliothèques IP, ce qui a considérablement simplifié la conception et l'intégration de ce composant dans notre projet.

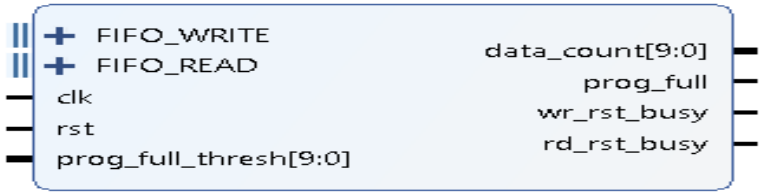

Figure 3:FIFO physical representation

a) Simulation pour vérifier l'écriture et la lecture des données en respectant l'ordre.

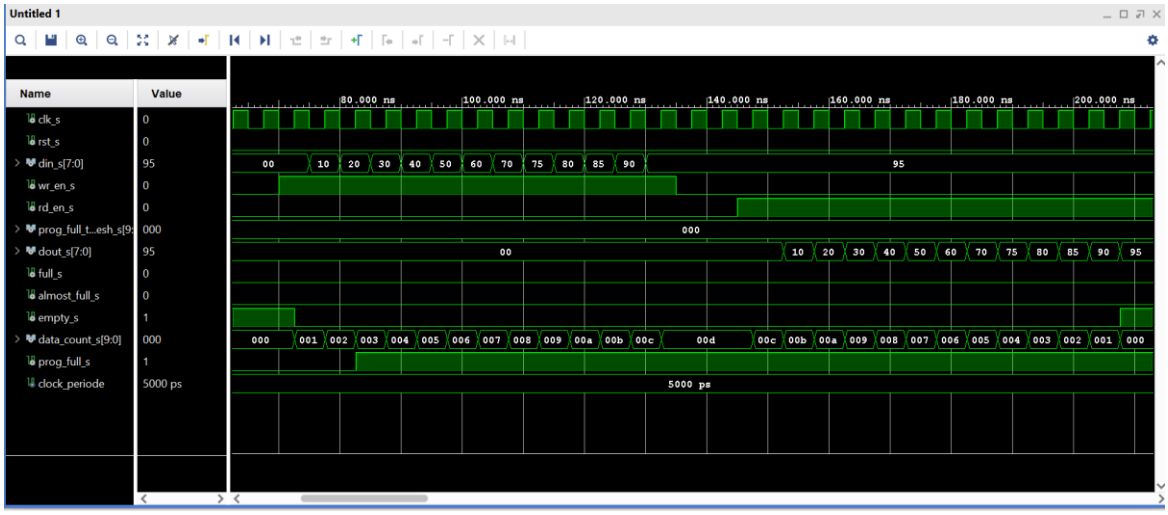

Figure 4:Control des signaux lecture/écriture(Fifo)

*b) Test de la gestion des états de saturation (**full / empty**)*

En lisant la documentation de la FIFO et avec l'aide du professeur, j'ai compris qu'après un reset, la FIFO nécessite 10 cycles d'horloge pour devenir fonctionnelle. Pendant cette période, les signaux **full** et **empty** de la FIFO sont activés simultanément à 1, indiquant son état d'initialisation.

Pour gérer cette phase, j'ai développé deux méthodes d'initialisation :

1. **Compteur de cycles :** Créer un compteur qui décompte les 10 cycles nécessaires après le reset avant de considérer la FIFO comme opérationnelle.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity compt_decompt is
    Port (
        clk        : in  std_logic;
        enable     : in  std_logic;
        reset      : in  std_logic;
        ten_cycles : out std_logic
    );

end compt_decompt;
```

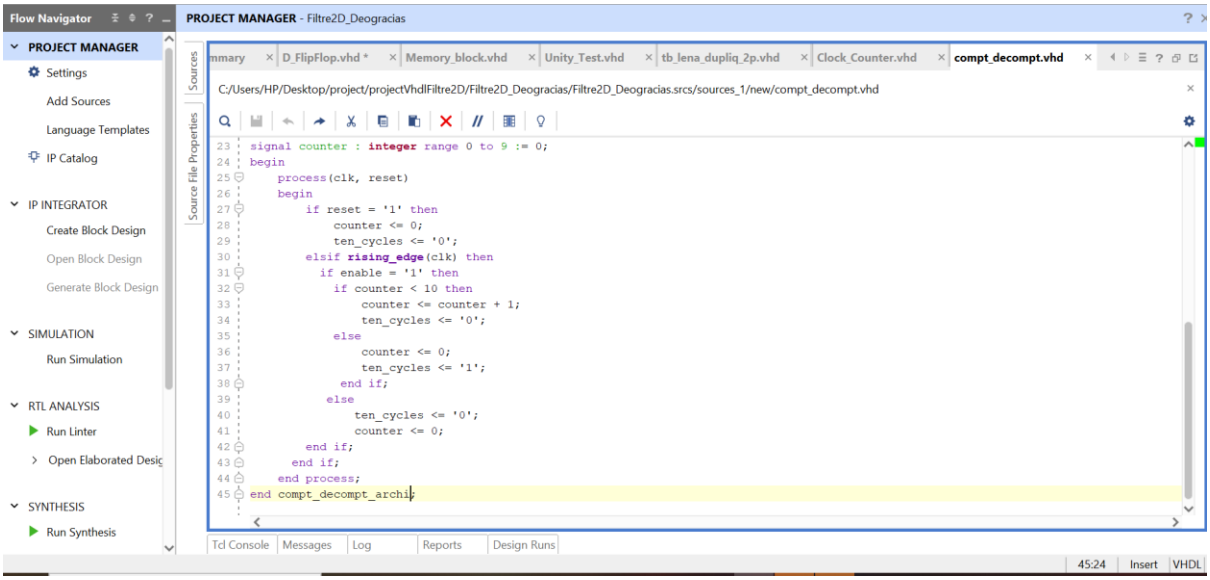

Figure 5:Compteur de cycles

Ce code implémente un **compteur sur 10 cycles d'horloge** qui active le signal ten_cycles après avoir compté 10 cycles. Le compteur s'incrémente lorsque le signal enable est actif et se réinitialise si reset est déclenché ou si enable est désactivé. Lorsque le compteur atteint 10, il se remet à 0, et ten_cycles passe à '1' pendant un cycle d'horloge. Ce module est utile pour créer des délais ou des phases d'initialisation dans des systèmes numériques.

2. **Utilisation des signaux de contrôle de la FIFO :** Vérifier si les signaux **full** et **empty** sont activés à 1 en même temps, ce qui indique que la FIFO est encore en phase d'initialisation. Une fois ces signaux désactivés, la FIFO est prête à être utilisée.

```
if((full1 and empty1)='1')
```

Figure 6:condition de test sur les signaux full/empty

La condition if ((full1 and empty1) = '1') vérifie si les deux signaux **full1** et **empty1** sont activés simultanément à '1'.

Dans ce cas, (phase d'initialisation) ou juste après un reset, la FIFO peut temporairement activer ces deux signaux pour indiquer qu'elle n'est pas encore opérationnelle. Cette condition est donc utilisée pour détecter cet état particulier et différencier la phase d'initialisation des phases normales de fonctionnement.

Ces deux approches garantissent une synchronisation correcte du système avec la FIFO après chaque réinitialisation.

J'ai choisi d'utiliser la **deuxième solution** pour gérer l'état d'initialisation de la FIFO, car elle est plus simple à implémenter et nécessite moins de ressources matérielles. En utilisant directement les signaux de contrôle **full** et **empty** pour détecter l'état initial (lorsqu'ils sont simultanément à '1'), je peux éviter d'ajouter un compteur ou une logique supplémentaire.

De plus, cette méthode est plus fiable car elle s'appuie sur le comportement natif de la FIFO, tel que défini par sa documentation et son architecture interne. Cela garantit une meilleure synchronisation avec les autres composants du système.

c) Ligne de retard

NB : Test Bench disponible dans le Projet (Unity-test)

Pour transformer une FIFO en ligne de retard, j'ai configuré deux FIFO en cascade des FliFlop. Cette architecture permet de retarder les données de plusieurs lignes d'une image afin d'obtenir, à chaque cycle, une fenêtre 3x3 de pixels nécessaires pour appliquer les filtres 2D. La première FIFO retarde les données d'une ligne complète, tandis que la seconde FIFO crée un décalage supplémentaire pour aligner les pixels voisins.

J'ai choisi de fixer le seuil **prog_full_thresh** à `0001111100` (décimal : 124) pour chacune des deux FIFO. Ce choix garantit que chaque FIFO contient toujours 124 pixels prêts à être lus, permettant ainsi de générer la fenêtre 3x3. Cette valeur correspond à la largeur de l'image moins 3, soit :

- $128-3=124$;

Cette configuration est nécessaire car, pour appliquer un filtre 3x3, il faut que chaque pixel de la ligne courante soit aligné avec ses deux voisins des lignes précédentes et suivantes.

Ce seuil a été calculé en fonction des dimensions de l'image (128x128). Une ligne complète contient 128 pixels, mais seuls 124 pixels peuvent être utilisés après le décalage nécessaire pour former une fenêtre 3x3 en continu. En choisissant ce seuil, j'ai garanti un fonctionnement stable et continu de la ligne de retard.

2. Partie Traitement

La partie traitement est responsable du filtrage. Les neuf pixels extraits à chaque cycle dans le cache mémoire sont utilisés dans une architecture pipelinée.

Objectif : Valider les modules de calcul avant intégration dans le pipeline.

- **Tests réalisés :**

2.1 Module de multiplication :

- Vérification des produits entre les coefficients des filtres et les pixels entrants.

2.2 Additionneur multi-entrées :

- Test de l'addition des 9 pixels multipliés.
- Vérification des dépassements (overflow) et arrondis.

2.3 Pipeline :

- Simulation du traitement étape par étape pour chaque pixel.

3. Génération des Stimuli

- Utilisation d'un fichier d'image transformé en matrice de pixels pour alimenter les simulations (ex. image de 128x128 pixels).
- Comparaison des résultats simulés avec des attentes théoriques calculées sous MATLAB.

III. Architecture Générale du Filtre 2D

1. Cache Mémoire

- **Description:**
    - Le cache mémoire stocke temporairement les données d'entrée pour permettre l'accès simultané à une fenêtre 3x3 pixels en un cycle d'horloge.
    - Structure : *"La mémoire cache est composée de registres flip-flop et de mémoire FIFO, transformée en ligne de retard pour s'adapter à la résolution spatiale de l'image (128x128)."*

Partie 1 : Hardware

Dans cette section, nous présentons en détail l'architecture matérielle du système, en expliquant comment les composants (Flip-Flops, FIFO) sont interconnectés pour faciliter le traitement des pixels. Nous détaillons également les signaux utilisés et justifions le choix d'une connexion manuelle pour accéder et manipuler efficacement les 9 pixels nécessaires au filtrage.

Plusieurs signaux ont été déclarés pour gérer les données entre les Flip-Flops, les FIFO et les autres composants. Voici une brève description des principaux signaux utilisés :

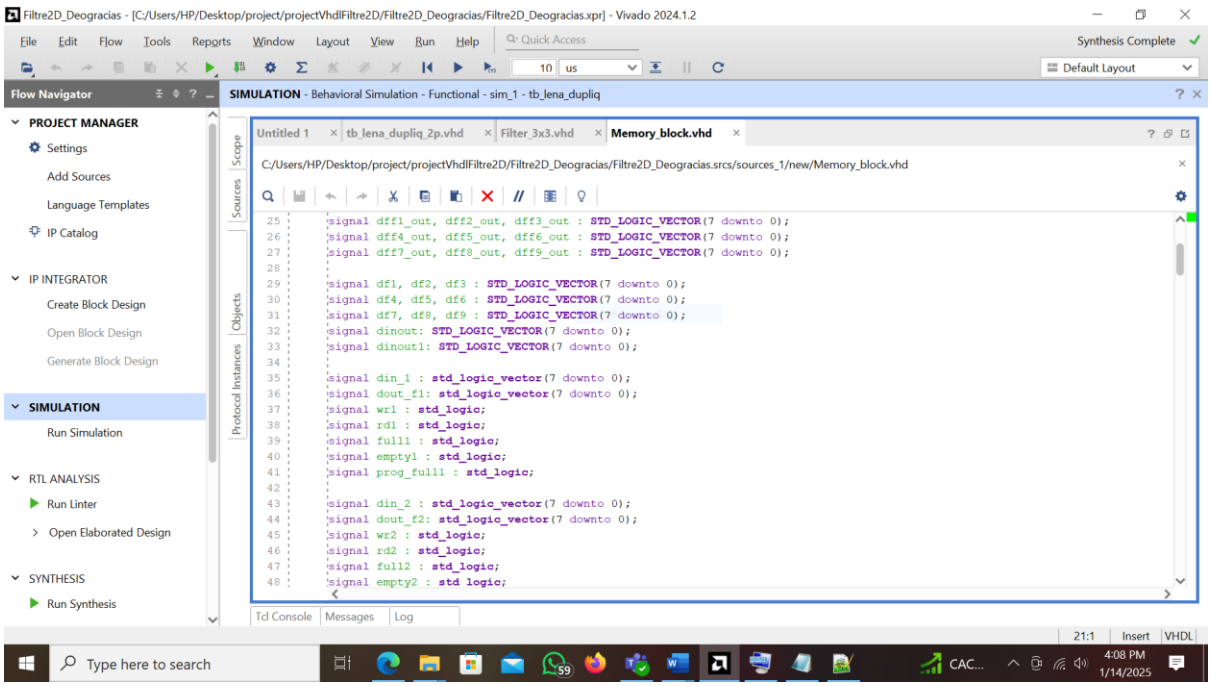

Figure 7:signaux de connection

- **dff1_out à dff9_out** : Ces signaux transportent les données stockées dans chaque Flip-Flop, correspondant aux pixels nécessaires pour le filtrage 3x3.
- **dout_f1, dout_f2** : Signaux de sortie des FIFO pour récupérer les données après le stockage temporaire.
- **enable_fd1, enable_fd2, enable_fd3** : Signaux d'activation des Flip-Flops dans chaque étape.
- **wr1, rd1, wr2, rd2** : Signaux de contrôle d'écriture et de lecture des FIFO.

→ Justification des Connexions Manuelles

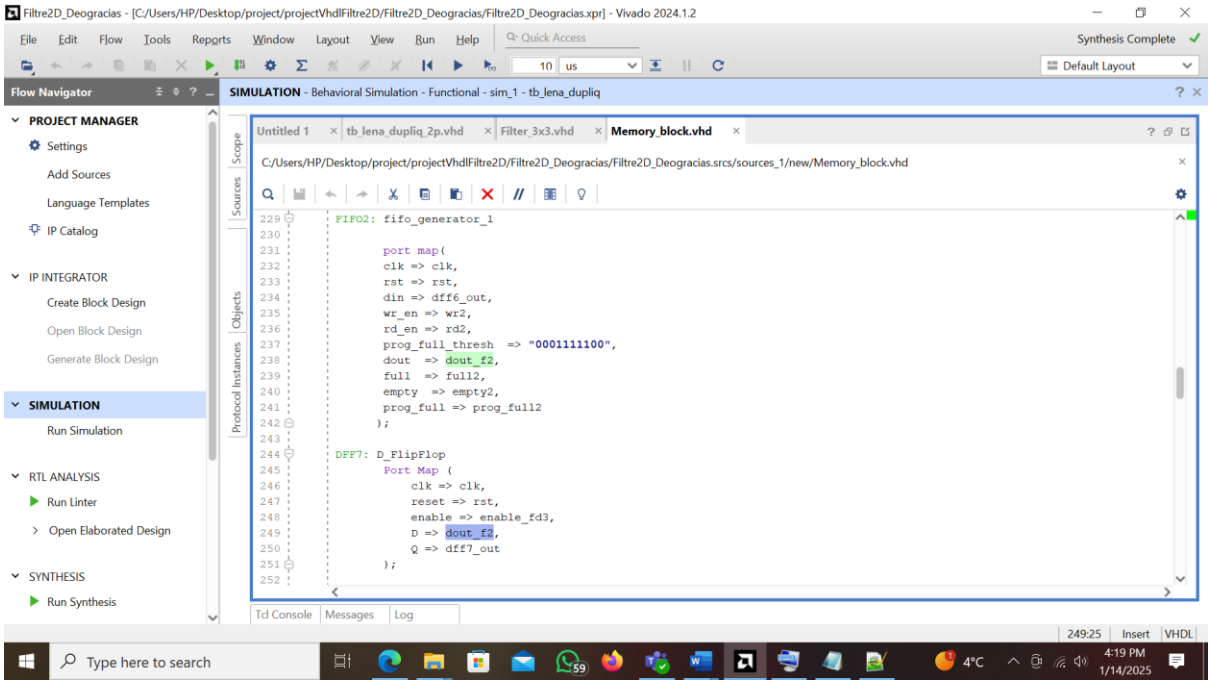

Figure 8:Connexions Manuelles

Nous avons choisi de connecter manuellement les Flip-Flops et les FIFO plutôt que d'utiliser une boucle de génération (if generate) pour les raisons suivantes :

- **Accès Direct aux Pixels** : Cette méthode permet de cibler directement les signaux de sortie des Flip-Flops (dff1_out à dff9_out), facilitant l'extraction des 9 pixels nécessaires au filtrage 3x3.
- **Simplicité de Maintenance** : La connexion manuelle offre une visibilité complète sur le flux de données. En cas de défaillance, il est plus facile de localiser et de corriger les erreurs.
- **Flexibilité dans les Modifications** : Cette méthode offre la possibilité d'ajouter ou de modifier des connexions sans affecter la structure globale du circuit.

- **Réduction de la Complexité du Débogage** : Avec des connexions explicites, il est plus simple de suivre le cheminement des données et de détecter d'éventuelles incohérences.

Dans cette partie, nous avons démontré comment les composants matériels, tels que les Flip-Flops et les FIFO, sont connectés pour traiter efficacement les données des pixels. La méthode de connexion manuelle a été préférée pour son accessibilité, sa clarté et sa facilité de maintenance, ce qui en fait une solution adaptée à la gestion du filtrage 3x3.

Partie 2 : Machine à États (State Machine)

La machine à états est un composant essentiel du système. Elle pilote le matériel en définissant les différentes étapes de traitement des données. Chaque état de la machine joue un rôle crucial dans le flux de données et garantit le fonctionnement synchronisé des éléments matériels. Voici une description détaillée des différents états :

État 1: STAGE1

- **Fonctionnement** : Cet état initialise le système. Si les indicateurs full1 (FIFO1 pleine) et empty1 (FIFO1 vide) sont activés en même temps, le système reste dans cet état. Sinon, il passe à l'état suivant.

  NB : le choix de cette méthode a été expliqué plus haut dans la préconception du dispositive

- **Importance** : Cet état s'assure que le système est correctement synchronisé avant de commencer le traitement.

État 2: STAGE2

- **Fonctionnement** : Cet état initialise la fonction de la FIFO en ligne de retard et vérifie si les pixels de l'image sont disponibles à l'entrée de la mémoire (wr_en au premier flip-flop). Si les pixels sont disponibles, il active les trois premiers flip-flops (enable_fd1) et déclenche le compteur pour compter 3 périodes, il passe à l'état suivant.
- **Importance** : Cet état joue un rôle clé en préparant les données pour leur traitement et en s'assurant que les conditions sont remplies avant de progresser.

État 3: STAGE3

- **Fonctionnement** : L'écriture dans la FIFO1 (wr1) est activée. Si la FIFO1 devient pleine (prog_full1), le système passe à l'état suivant.
- **Importance** : Cet état permet de préparer les données pour les étapes suivantes tout en surveillant l'état des FIFOs.

État 4: STAGE4

- **Fonctionnement** : Cet état poursuit l'écriture dans la FIFO1 tout en activant les signaux nécessaires pour transférer les données dans la FIFO2(il active les trois deuxième flip-flops (enable_fd2) et déclenche le compteur pour compter 3 périodes). Le compteur est également activé pour mesurer la progression. Lorsque le compteur indique la fin (done_s), le système passe à l'état suivant.
- **Importance** : Cet état garantit que les données sont transférées correctement de la FIFO1 à la FIFO2.

État 5: STAGE5

- **Fonctionnement** : L'écriture dans la FIFO2 (wr2) commence, et les registres intermédiaires sont activés (enable_fd2). Si la FIFO2 devient pleine (prog_full2), le système passe à l'état suivant.
- **Importance** : Cet état est critique pour alimenter correctement la deuxième FIFO tout en assurant une synchronisation continue.

État 6: STAGE6

- **Fonctionnement** : Cet état active tous les 3 dernier flipflop (enable_fd3) pour garantir que les données sont correctement préparées pour le filtre.
- **Importance** : Cet état prépare les données pour leur traitement final par les filtres.

Dans ce projet, nous avons porté une attention particulière à l'optimisation énergétique et au bon fonctionnement global de la mémoire cache. Pour cela, les signaux d'activations **enable_fd1, enable_fd2** et **enable_fd3** ont été configurés de manière à activer les flip-flops uniquement lorsque les données sont effectivement disponibles sur leurs ports respectifs. Cette approche garantit une utilisation efficace de l'énergie en évitant le gaspillage, contribuant ainsi à réduire la pollution électronique.

De plus, les signaux d'écriture du FIFO ont été programmés pour s'activer uniquement lorsque les données sont présentes à leurs ports d'entrée. Cela empêche le stockage de données indésirables qui pourraient perturber le fonctionnement global et compromettre les résultats de la mémoire cache.

Un compteur de période de trois cycles a également été intégré afin de déterminer précisément le moment où les données sont disponibles sur les ports du FIFO. Cette synchronisation précise améliore la fiabilité et l'efficacité du traitement, assurant un flux de données optimal pour les opérations de filtrage et d'autres fonctions critiques du système.

Ces choix techniques renforcent l'efficacité énergétique et l'intégrité des données, tout en minimisant l'impact environnemental, ce qui reflète une approche moderne et responsable dans la conception des systèmes numériques.

2. Traitement

2.1 Description des Filtres Moyenneur et Gaussien

a) *Filtre Moyenneur*

Coefficients utilisés:

- Première configuration:
  [1 1 1],[1 1 1],[1 1 1]

  [1 11], [1 0 1],[1 1 1]

  [1 1 1],[1 1 1],[1 1 1]
- Deuxième configuration:
  [1 1 1],[1 1 1],[1 1 1]

  [1 11], [1 1 1],[1 1 1]

  [1 1 1],[1 1 1],[1 1 1]

Description : Le filtre moyenneur permet de réduire le bruit dans une image en remplaçant chaque pixel par la moyenne pondérée des pixels environnants selon les coefficients donnés.

Système en pipeline : Les neuf pixels issus de la ligne de retard (formée par des FIFO et des registres) sont transmis simultanément à un module de traitement.

- Le pipeline décompose le calcul en plusieurs étapes : Addition progressive des produits obtenus.
- Division par un facteur de normalisation.

Ce système garantit un traitement rapide et efficace des pixels, permettant d'appliquer le filtre en temps réel.

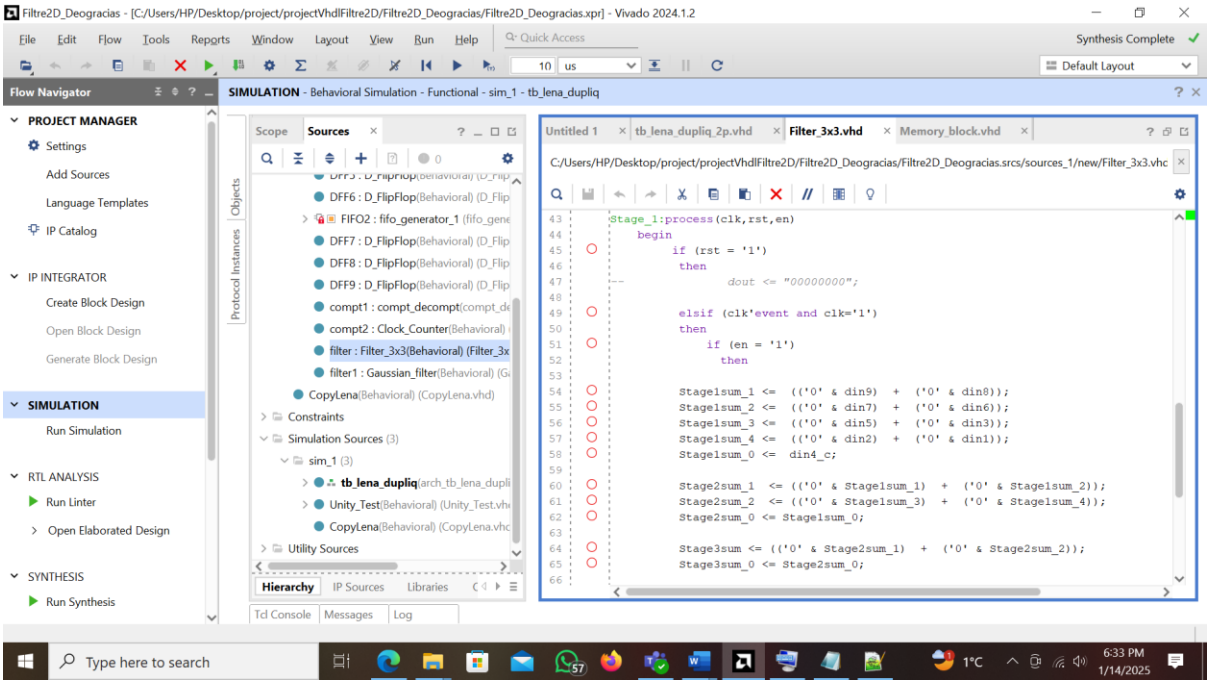

Figure 9:Filtre Moyenneur

b) *Filtre Gaussien*

Description :
Le filtre gaussien est utilisé pour effectuer un lissage plus précis tout en préservant les contours importants de l'image. Ses coefficients suivent une distribution gaussienne, favorisant les pixels proches du centre.

Système en pipeline :

- Comme pour le filtre moyenneur, les 9 pixels sont extraits de la mémoire cache via la ligne de retard.
- Les coefficients du filtre gaussien sont appliqués dans l'étape de multiplication.
- L'addition des produits se fait en cascade pour minimiser les retards.
- La sortie est normalisée en divisant par la somme des coefficients pour obtenir une image correctement filtrée.

Avantages du Système en Pipeline :

- Traitement en parallèle : Plusieurs étapes du calcul sont exécutées simultanément, augmentant la vitesse d'exécution.

- Réduction du délai global : Le pipeline réduit le temps nécessaire pour traiter chaque pixel.

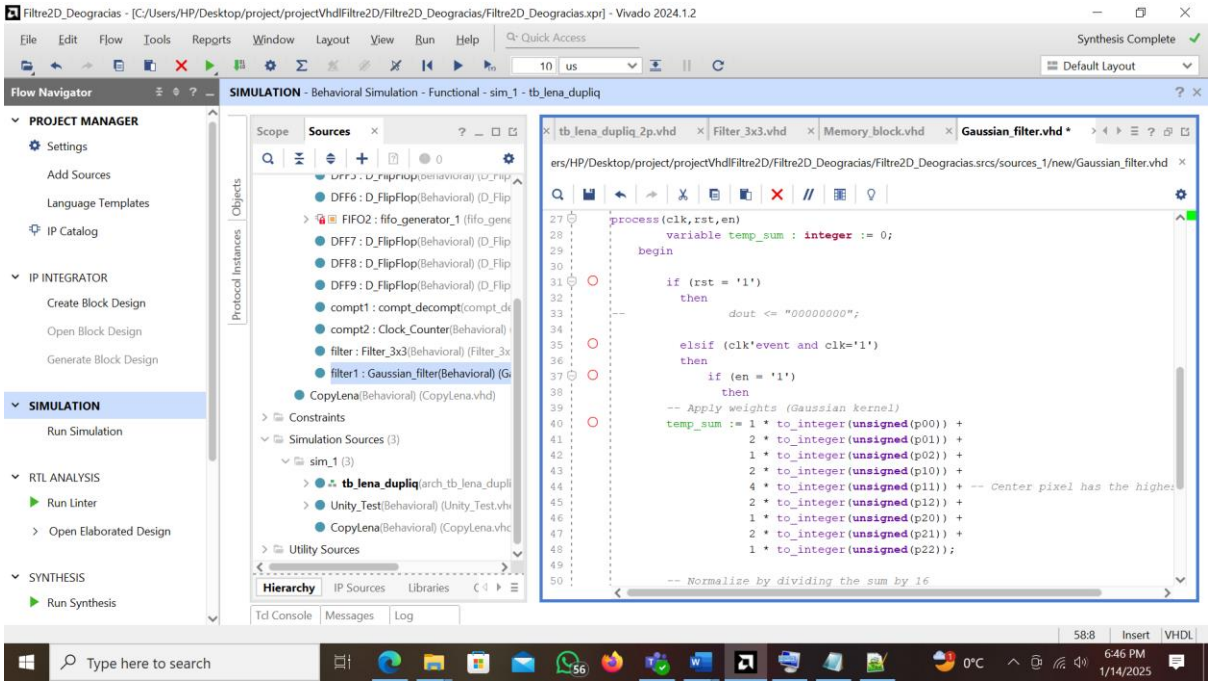

Figure 10:Filtre Gaussien

IV. Implémentation et Test du Filtrage

1. Implémentation du Filtrage

Pour réaliser l'implémentation des filtres Moyenneur et Gaussien, une approche basée sur l'utilisation de blocs de mémoire a été adoptée. Un mécanisme de contrôle précis a été mis en place grâce à un drapeau (**flag**) pour déclencher le filtrage de l'image au bon moment.

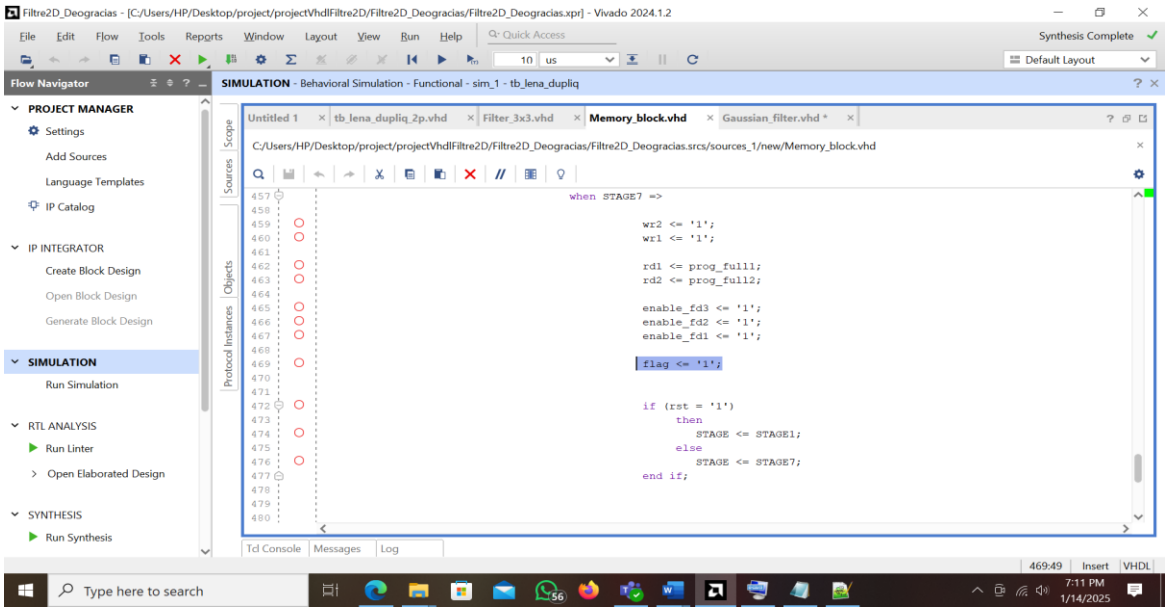

Figure 11:déclencher le filtrage (stage 7)

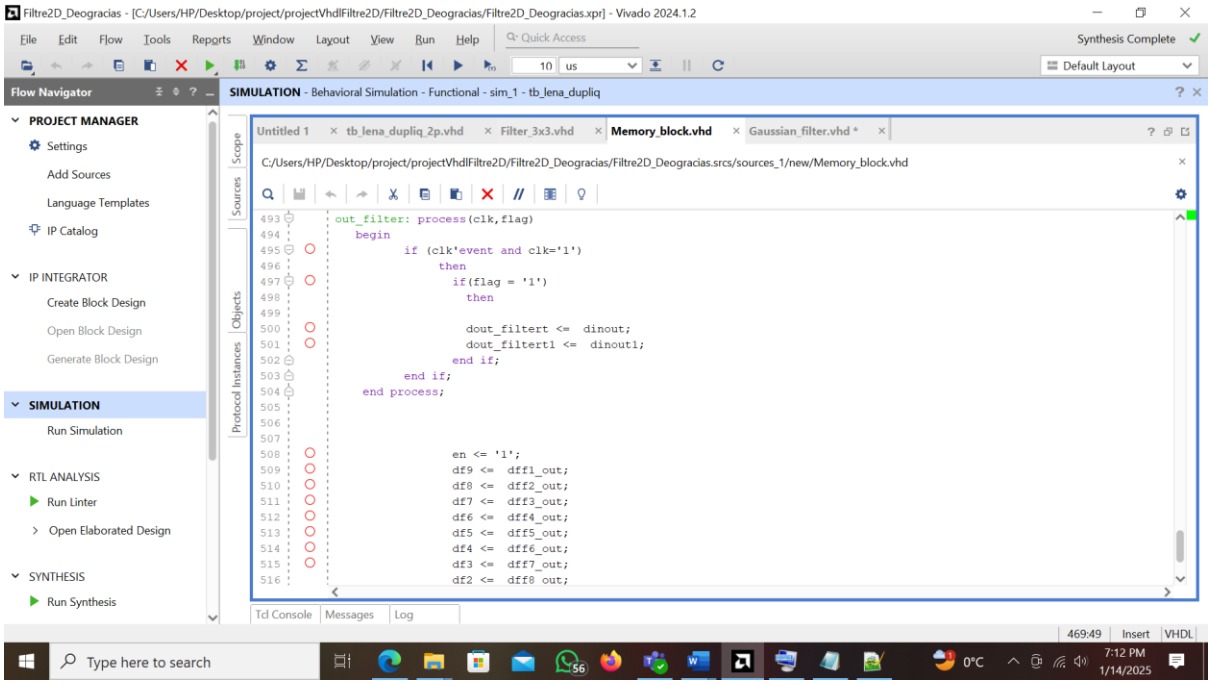

Figure 12:Utilisation du Flag pour déclencher le filtrage

2. Test du Filtrage

Pour tester le filtrage, j'ai créé un banc de test appelé tb_lena_dupliq. J'ai initialisé mon bloc mémoire, puis utilisé l'image de Lena qui avait été convertie en un fichier .dat contenant ses pixels. Ce fichier a servi de source d'entrée pour le test.

J'ai ensuite appliqué le filtrage sur l'image à l'aide de mon circuit et généré deux fichiers dans lesquels les sorties des filtres ont été stockées. Ces fichiers contiennent les pixels filtrés, qui ont ensuite été transformés en une image pour vérifier les résultats du filtrage. Ce processus permet de valider le bon fonctionnement de l'algorithme de filtrage et de s'assurer que l'image est correctement traitée.

2.1 tb_lena_dupliq

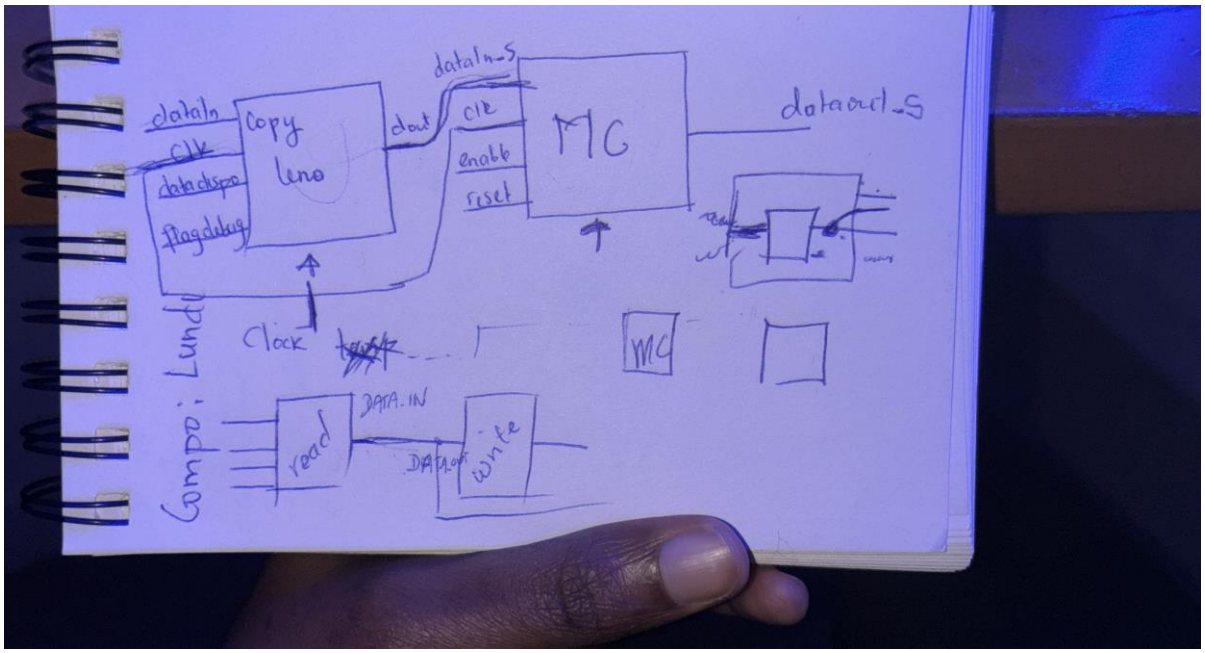

Figure 13:pre-conception de tb_lena_dupliq

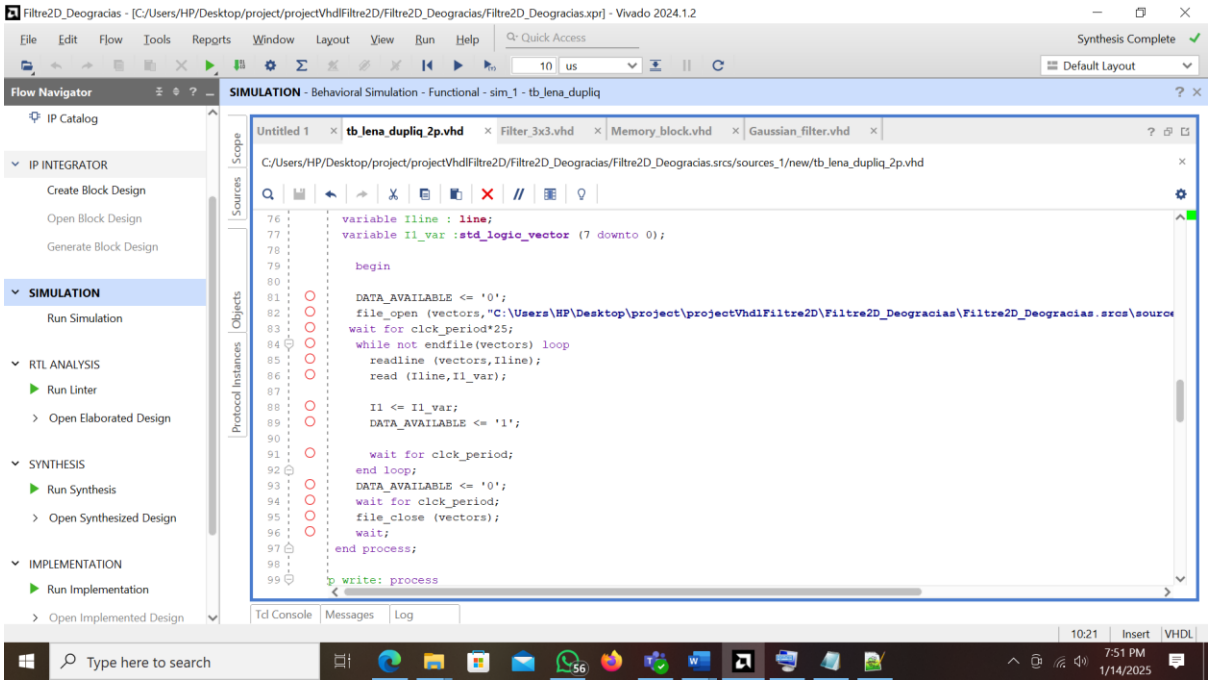

Figure 14:conception de tb_lena_dupliq

Le banc de test consiste principalement en trois processus : la lecture des données d'entrée, l'écriture des résultats filtrés et la simulation du signal de contrôle.

1. **Processus de lecture des données (p_read)**
   Un fichier contenant les pixels de l'image de Lena, formaté sous forme de données binaires (fichier .dat), est ouvert pour être utilisé comme entrée du filtre. Le processus lit chaque pixel du fichier et le transmet à l'entrée du bloc mémoire (I1). La disponibilité des données est signalée par le signal DATA_AVAILABLE, qui est mis à '1' lorsque de nouvelles données sont lues. Après avoir envoyé toutes les données, le fichier est fermé et le processus se termine.

2. **Processus d'écriture des résultats filtrés (p_write et p_write1)**
   Les résultats des filtres sont ensuite écrits dans deux fichiers différents. Un fichier reçoit les résultats du premier filtre (O1) et l'autre du second filtre (O2). Les processus attendent que les données soient disponibles (signal DATA_AVAILABLE = '1') avant d'écrire chaque pixel filtré dans les fichiers. Chaque pixel est écrit dans le fichier sous forme binaire, puis les résultats sont sauvegardés et les fichiers sont fermés à la fin de l'écriture.

3. **Processus de simulation (sim)**
   Ce processus gère les signaux de contrôle, notamment le signal de réinitialisation (rest_TB) et le signal d'écriture (wr_TB). Il réinitialise le système au début, active l'écriture après un certain délai, puis désactive l'écriture une fois que toutes les données ont été envoyées pour traitement. Ce processus permet de simuler le fonctionnement du système et de contrôler l'ordre des événements dans le test.

4. **Bloc mémoire et filtrage**
   Le bloc mémoire (Memory_block) est responsable de stocker et de traiter les pixels. Les signaux de contrôle, comme wr_en, permettent de contrôler l'écriture des données dans la mémoire. La sortie du filtre est récupérée via les signaux **dout_filtert_TB** et **dout_filtert1_TB,** qui sont ensuite affectés aux sorties (**O1** et **O2**) pour être écrites dans les fichiers.

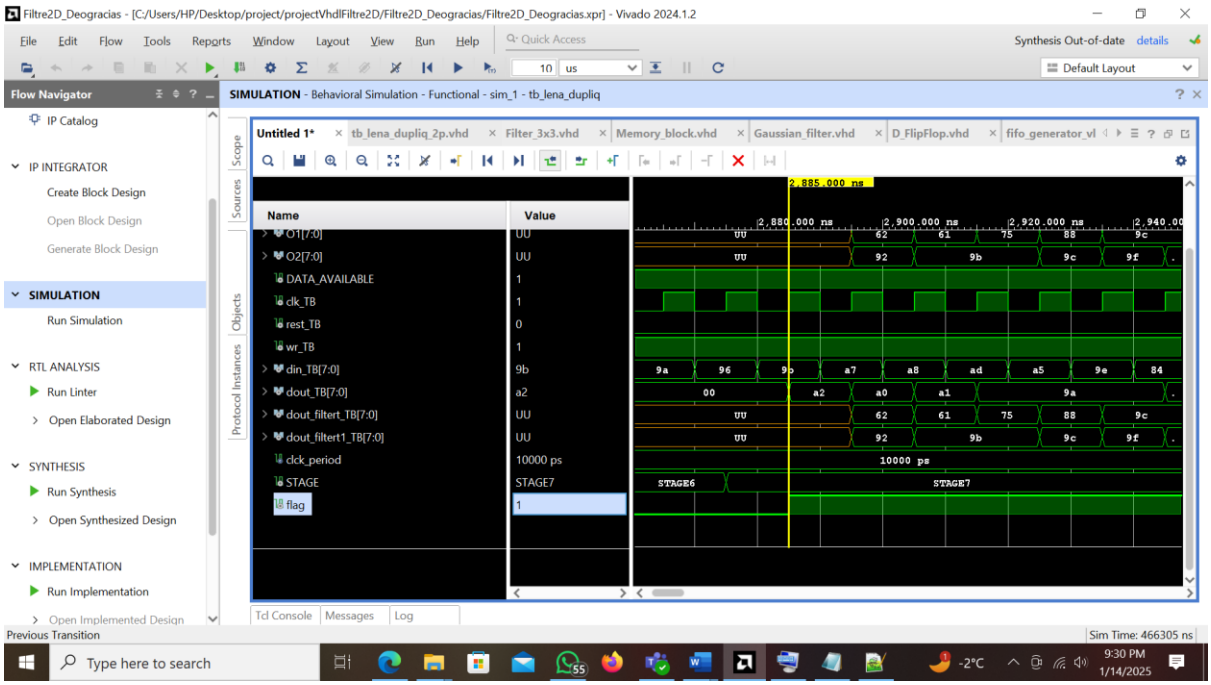

Figure 15:Test global du system

Cette structure me permet de tester le filtrage d'images de manière automatisée, tout en sauvegardant les résultats dans des fichiers qui peuvent être convertis en images pour vérifier visuellement les effets du filtre.

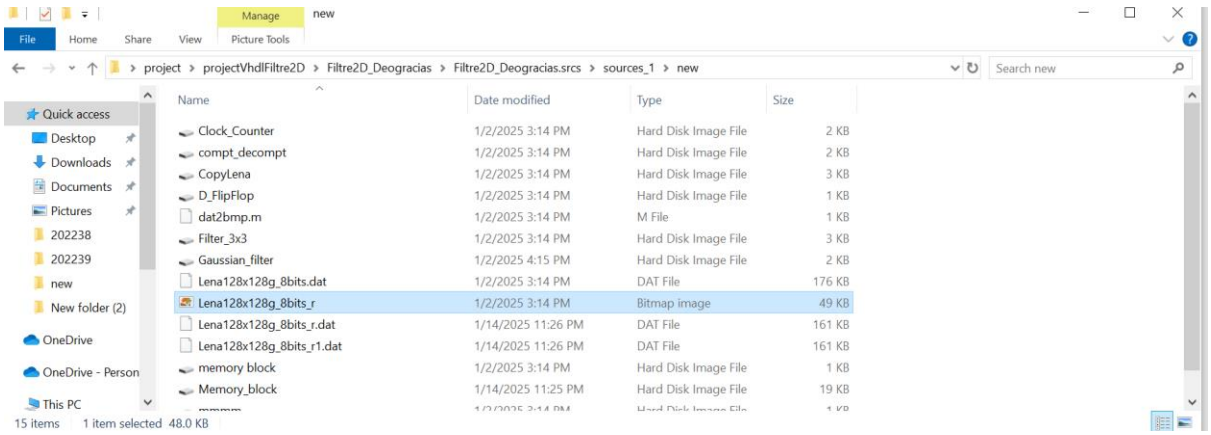

Figure 16:Image résultant

L’objectif ici étant de tester les différents filtres conçus. En utilisant le code ci-dessous on obtient les deux résultats qui suivent pour le filtre flou et Gaussien.

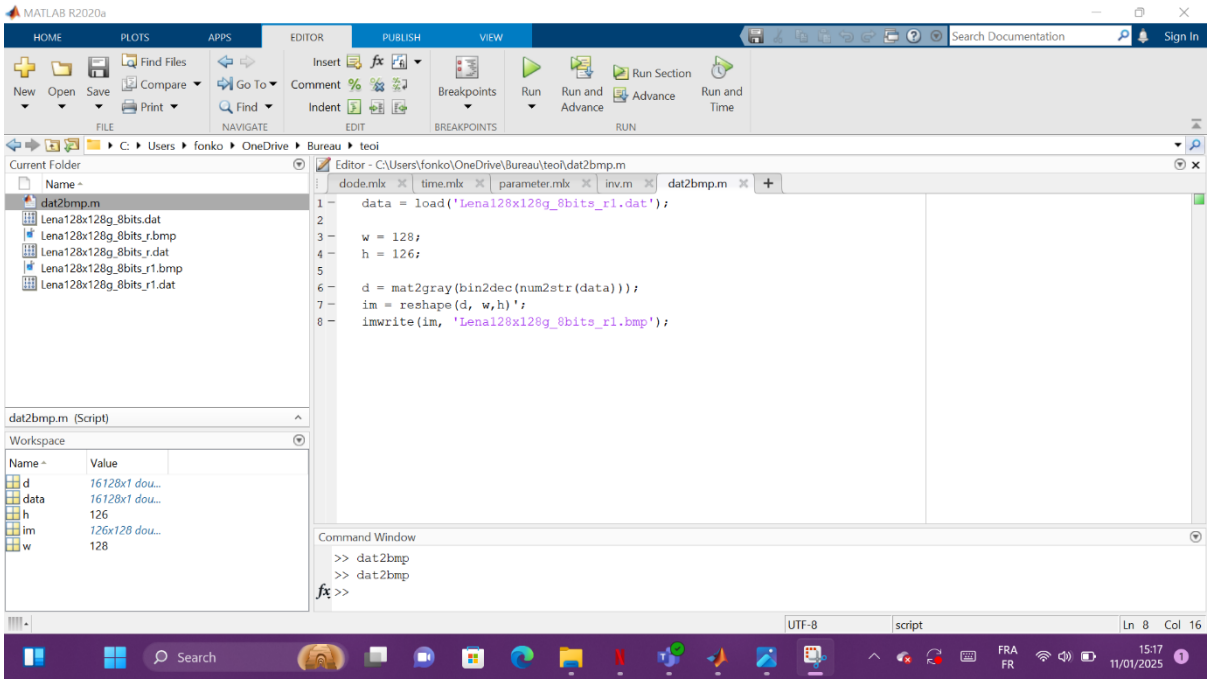

Figure 17:Re-generation de l'image après filtrage

Figure 18:Lena Flou

Figure 19:Lena Gaussien

V. Difficultés rencontres

Lors de la réalisation de ce projet, plusieurs défis ont été rencontrés :

1. **Gestion des signaux dans le VHDL** : L'activation des signaux tels que senable_fd1, enable_fd2, et enable_fd3 a nécessité une synchronisation précise pour éviter des états indéterminés.
2. **Conversion des images en données** : Transformer l'image Lena en un fichier de données (format .dat) utilisable dans le test bench a demandé un traitement spécifique pour assurer la correspondance des pixels.
3. **Tests et validation des filtres** : La vérification des sorties des filtres en simulant le comportement du FIFO et des données a pris plus de temps que prévu en raison de la complexité de la logique impliquée.
4. **Gestion des fichiers de sortie** : Générer des fichiers contenant les résultats des filtres et les transformer en images exploitables a nécessité plusieurs itérations pour garantir leur exactitude.

VI. Améliorations

Pour améliorer ce projet à l'avenir, plusieurs aspects peuvent être optimisés :

1. **Optimisation énergétique** : L'utilisation de signaux d'activation uniquement lorsque les données sont disponibles permet déjà d'économiser de l'énergie. Cependant, des optimisations supplémentaires, comme la réduction de la fréquence d'horloge lorsque l'activité est faible, pourraient être explorées.
2. **Interface utilisateur** : Ajouter une interface graphique pour afficher les résultats directement sans passer par des conversions manuelles améliorerait l’expérience utilisateur.
3. **Extensions des filtres** : Développer des variantes de filtres (par exemple, passe-bas, passe-haut, etc.) pour couvrir une gamme plus large de traitements d'images.
4. **Documentation automatisée** : Intégrer un système de génération automatique de rapports ou de documentation pour gagner du temps.

VII. Conclusion

Ce projet a permis de mettre en pratique des compétences variées en programmation VHDL, en gestion des signaux numériques, et en simulation de composants électroniques. La conception des signaux d’activation, tels que senable_fd1, enable_fd2, et enable_fd3, a été particulièrement importante pour garantir une utilisation efficace de l'énergie et limiter les impacts environnementaux.
De plus, l'utilisation de compteurs de périodes a permis une gestion précise des flux de données, assurant ainsi la fiabilité des résultats. Bien que des défis aient été rencontrés, ils ont permis de mieux comprendre les contraintes et opportunités liées à la conception de systèmes numériques complexes.

Ce projet constitue une base solide pour de futures améliorations et pour des applications dans des contextes industriels réels.