Polytech Dijon

FISA IOT 4A

Architecture Reconfigurable

Projet Filtre 2D

Auteur :
HELLE Evan

Tuteur :
DUBOIS Julien

2024-2025

Sommaire

Table des Figures

1 Compte Rendu

1.1 Objectif du projet :

L'objectif de ce projet est de réaliser un filtre 2D afin de filtrer des images allant jusqu'à 1024 pixels (2^{10} bits).

Ce projet nous permettra de traiter des images avec différents filtres comme un Sobel ou un Roberts.

Si nous avançons bien, une implémentation sera réalisée sur carte `Nexys4 xc7a100tcsg324`

1.2 Duplication de Lena

1.2.1 contexte

afin d'appliquer des filtres à nos images, nous allons dans un premier temps dupliquer une image de Léna.

Pour ce faire, nous allons utiliser le fichier fournit `tb_lena_dupliq_2p`. Ce dernier n'est pas complet, mais est déjà une bonne base de travail afin de dupliquer une image.

```
--process ajoutée j'ai changé les wait avec des cycle d'horloge plutôt que des timings
clocking: process
  begin
     CLK <= '0';
     wait for clock_period/2;
     CLK <= '1';
     wait for clock_period/2;
end process;
```

FIGURE 1.1 – Clock_Process

La première modification effectuée a été d'ajouter un processus d'horloge afin de ne pas avoir de variable indéfinie. Cela à permis de synchroniser les `wait` avec les cycles d'horloge.

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

    begin

    file_open (results,"C:\Users\eh648454\Documents\Archi-reconfigurable\Projet\PROJET_TP\Lena128x128g_8bits_r.dat", write_mode);
    --ligne ajoutée
    wait for 1*clock_period;
    wait until DATA_AVAILABLE = '1';
    wait for 1*clock_period;
    -- changer la valeur '1' en '0' pour écrire quand nous somme à data_available à 1
    while not DATA_AVAILABLE ='0' loop
      write (Oline, O1, right, 2);
      writeline (results, Oline);
      wait for 1*clock_period;
    end loop;
    file_close (results);
    wait;
 end process;
```

FIGURE 1.2 – Writting_Process

Ensuite. Il fallait modifier la valeur attendue dans le `While`. En effet, sa valeur en simulation ne varie pas comme on peut se voir ci-dessous :

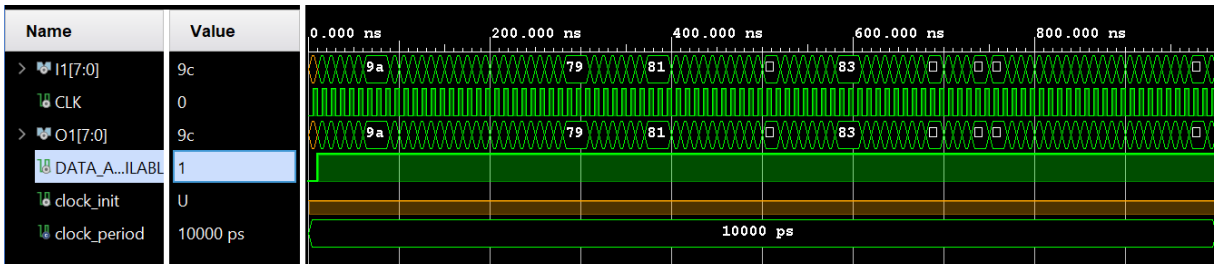


FIGURE 1.3 – Data_Available_simulation

Afin d'écrire dans notre fichier, il fallait donc changer cette condition.
Il a également fallu changer le `Path` des fichiers pour que Vivado puisse y accéder.

Finalement, nous avons obtenu ce fichier :

tb_lena_dupliq_2p.vhd | Lena128x128g_8bits_r.dat

```
1   10100010
2   10100011
3   10100010
4   10100000
5   10100001
6   10011010
7   10011010
8   10011010
9   10011011
10  10100000
11  10100110
12  10101101
13  10100111
14  10100010
15  10011001
16  01110110
17  01011110
18  01100111
19  01101001
20  01101000
21  01101011
22  01101101
23  01101010
24  01101011
```

FIGURE 1.4 – Fichier Léna dupliqué

Ce fichier est très similaire à l'original. Cependant, nous allons utiliser la fonction `compare` de NotePad++ pour vérifier que ce sont bien les mêmes et c'est bien le cas.

Enfin, à l'aide du programme `dat2bmp` nous avons pu reconstituer l'image sur Octave et nous avons pu nous assurer que nous retombions bien sur Léna.

FIGURE 1.5 – originale vs Duplication

1.3 Génération d'une FIFO

Vivado nous permet de générer des composants. Celui qui nous intéresse est la FIFO. Pour la générer, il faut se rendre dans l'onglet `IP catalog -> Menories & Storage Elements -> FIFO Generator`.

Une interface s'ouvrira alors et nous allons pouvoir commencer la configuration de notre composant.
Nous garderons l'interface native de la FIFO car les autres options AXI ne servent uniquement que si nous souhaitons nous connecter à un CPU.
Nous n'utiliserons qu'une seule horloge, car nous voulons que l'entrée soit synchronisée sur la sortie.
Nous ajoutons également une entrée `Single Programmable Full Threshold Input Port` afin de pouvoir modifier la taille des images que nous traiterons pour une meilleure flexibilitée. Il est utile de préciser que nous n'utiliserons pas de `Programmable Empty Type` car le système est synchrone.

Voici le composant que nous obtenons avec cette configuration :

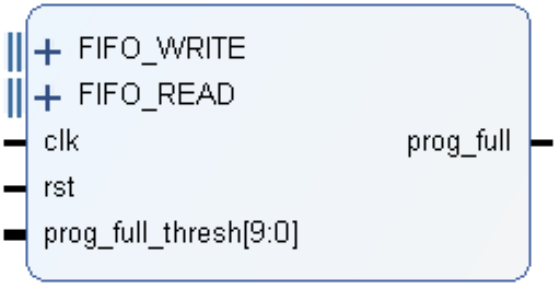


FIGURE 1.6 – Composant FIFO

1.4 Test unitaire de la FIFO et de la ligne à retardement

Voici un apperçu des tests que nous allons effectuer :

- Definir `prog_full_thresh` à 24 pour faire un retard de 3 octets. Envoyer des octets à `din` et regarder l'état des sorties `full` et `prog_full`.

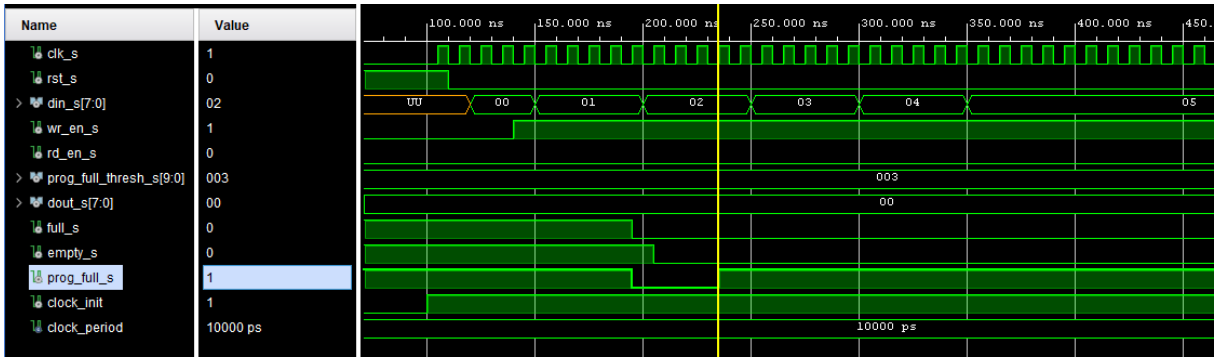


FIGURE 1.7 – Test Bench FIFO

Sur ce chronographe, nous voyons bien que la valeur de `threshold` est définie à 3 octets. Nous envoyons des mots de 8 bits tous les 5 cycles d'horloge. La barre jaune correspond au moment où `prog_full` passe à 1. Cela signifie que nous avons atteint le seuil que `threshold`.

- Afin de transformer la FIFO en ligne à rettardement, il faut connecter le `prog_full` à `rd_en` pour tester la ligne à retard et regarder la sortie `dout` quand `rd_en` et à 1.

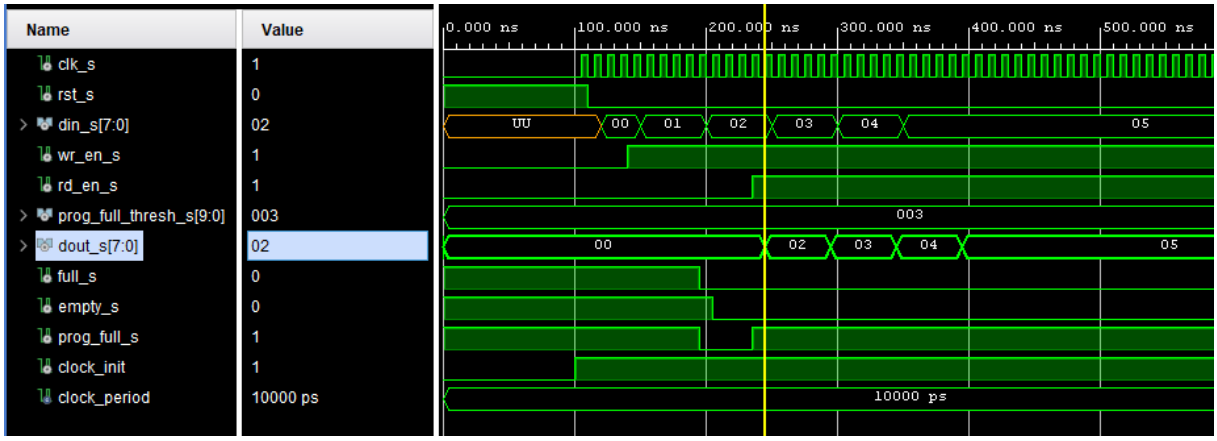


FIGURE 1.8 – Test Bench LR

Nous voyons sur ce chronographe que lorsque `prog_full` passe à 1, `rd_en` passe aussi à 1. cela active la lecture des octets contenue dans la FIFO sur la sortie `dout`.

1.5 Design de la Mémoire cache

1.5.1 Bascule D

pour les bascules D, j'ai réutilisé la bascule du TD1 `ff_jd`. Quelques modifications ont toutefois été faits par mes soins. En effet, sur le poste de travail Esirem, aucun problème avec les bascules, mais sur mon poste personnel, une erreur de `multi driven` apparaissait sur la sortie. Pour regler cela, j'ai changer la condition `RESET` en mettant tous le bit de `temp` à 0 plutôt que la sortie `Q` comme initialement dans la bascule que nous avions dans la correction du TD1.

```
1  begin
2
3  p1: process(CLK,RESET)
4  begin
5      if (RESET ='1') then Q <= (others => '0'); -- On passe désormais temp à 0.
6      elsif (CLK'event and CLK='1') then
7          if (EN ='1') then
8              temp <= D;
9          end if;
10     end if;
11 end process;
12
13 Q <= temp;
14
```

Dans mon fichier `Mem_cache.vhd` j'instancie et génère mes composants bascule et fifo

Nous utilisons un signal de 87 bits `transit` pour gérer les échanges de données entre les composants. Le shéma suivant montre que ce bus prend 95 bits. Ce serait le cas s'il y avait une troisième FIFO.

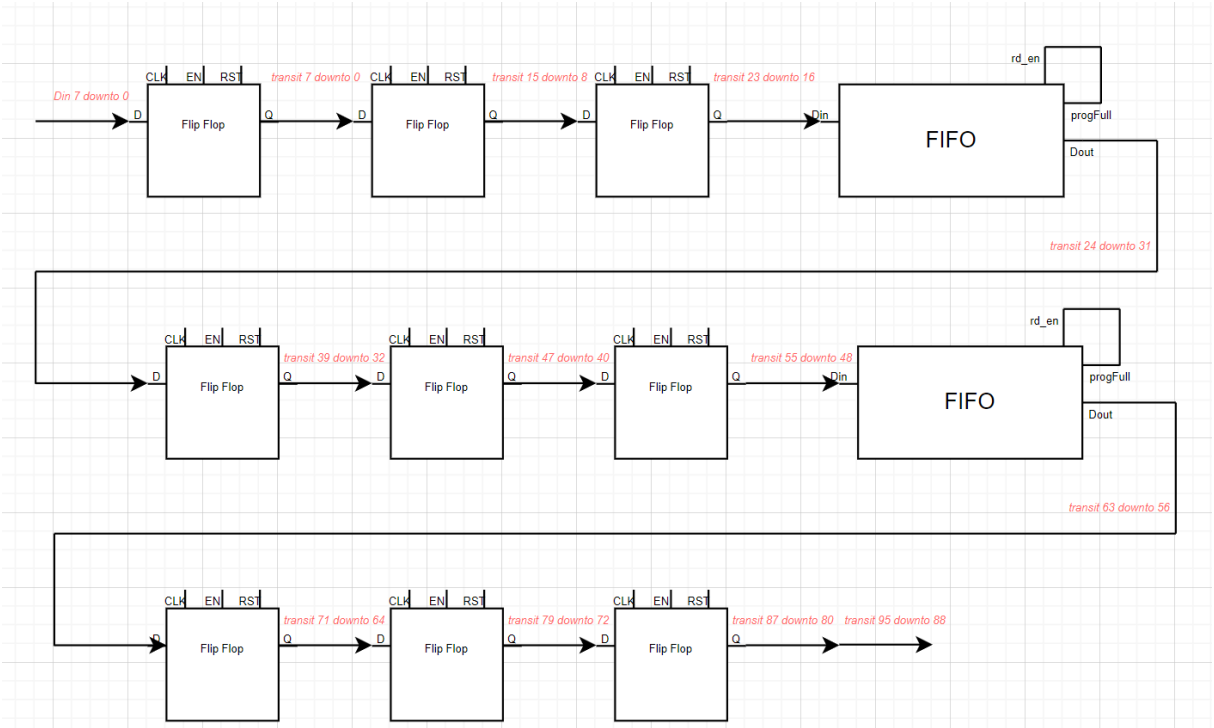


FIGURE 1.9 – design mémoire cache

Toutes les bascules ont le même `enable , clk et reset`. Nous aurions pu mettre un `enable` différent pour chaque composant afin de les activer seulement au moment où nous en avons besoin afin d'économiser des ressources, mais pour des raisons de temps, garder le même était plus simple.

tests unitaires

- Mettre le `treshold` à 7 pour voir le comportement de la mémoire.
- On souhaite commencer à lire écrire les données quand on commence à les recevoir donc `flag_synchro` passe à 1 quand on reçoit une donnée
- Changement du treshold pour avoir des images de 128 octets
- Envoie d'une image pour tester la reconstitution

Dans ce compte rendu, je ne vais pas m'attarder à détailler tous les tests unitaires. Je vais cependant détailler le dernier de la liste, soit, l'envoie d'une image à la mémoire pour tester que sa reconstitution se passe bien :

Pour l'instant, tout le processus de fonctionnement se passe dans le Test Bench. Ceci n'est pas une solution durable si on envisage une implémentation sur carte.
Voici le résultat d'une simulation qui permet de valider les tests :

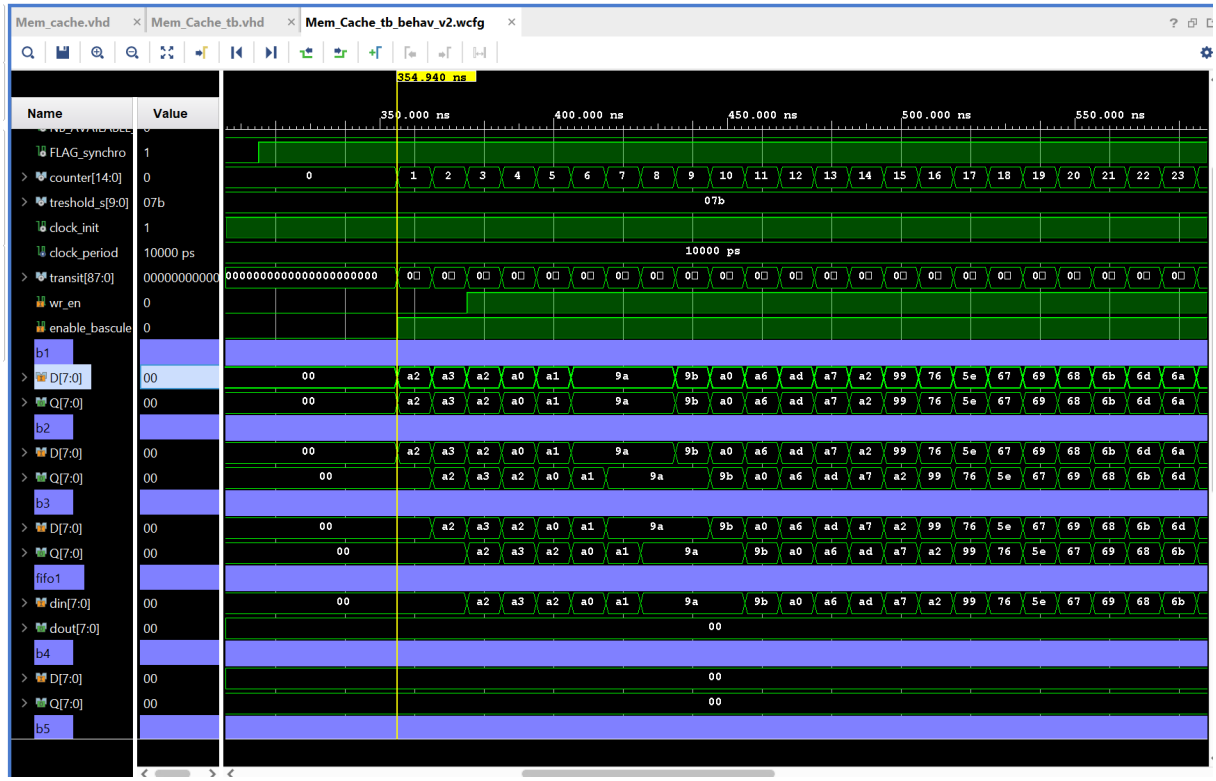
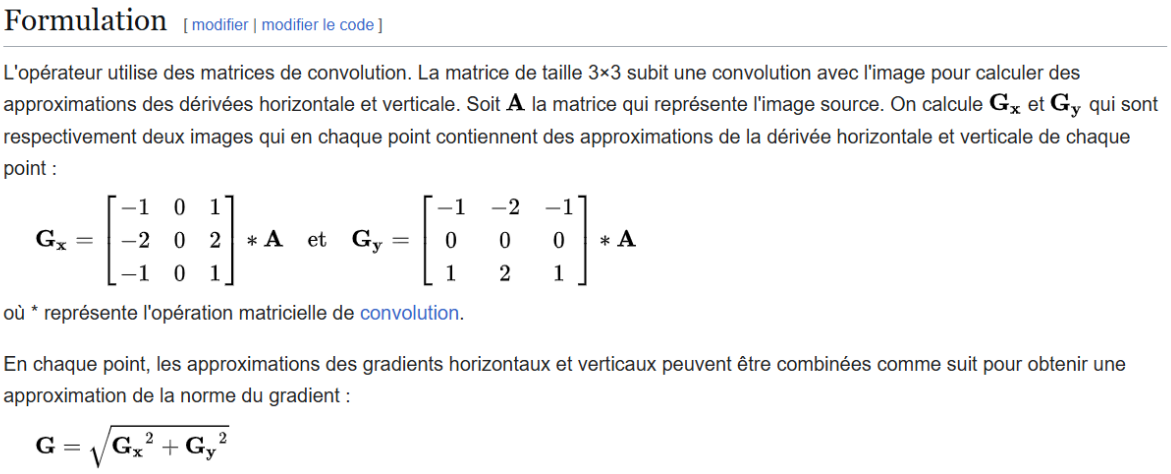


FIGURE 1.10 – Simulation tests unitaires

Les données sont bien transférées d'une bascule à une autre et retardées dans les fifos. Au final, ce test Bench permet de récupérer les données commme elles sont arrivées. Désormais, il faut les filtrer.

1.6 Filtrage Sobel

D'après Wikipédia, voici comment fonctionne le filtre de Sobel : `https://fr.wikipedia.org/wiki/Filtre_de_Sobel`



Formulation [modifier | modifier le code]

L'opérateur utilise des matrices de convolution. La matrice de taille 3×3 subit une convolution avec l'image pour calculer des approximations des dérivées horizontale et verticale. Soit $\mathbf{A}$ la matrice qui représente l'image source. On calcule $\mathbf{G_x}$ et $\mathbf{G_y}$ qui sont respectivement deux images qui en chaque point contiennent des approximations de la dérivée horizontale et verticale de chaque point :

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \mathbf{A} \quad \text{et} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \mathbf{A}$$

où * représente l'opération matricielle de convolution.

En chaque point, les approximations des gradients horizontaux et verticaux peuvent être combinées comme suit pour obtenir une approximation de la norme du gradient :

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2}$$

FIGURE 1.11 – Wikipédia Sobel

Voici mon implémentation dans le composant pour réaliser le filtre Sobel :

Il a été nécessaire d'utiliser beaucoup de signaux internes pour eviter les multi-driven. De plus pour eviter d'assigner en même temps deux valeurs sur le même bus, j'ai fait le choix d'un multiplexeur qui vient choisir l'entrée à prendre en fonction de la situation. Le multiplexeur fait passer la valeur du filtrage quand DATA_VALID = 1.

```
1  sobel: process(CLK, RESET)
2  begin
3      if RESET = '1' then
4          -- Réinitialisation des signaux
5          enable_temp <= '0';
6          Gx <= 0;
7          Gy <= 0;
8          temp_pixels <= (others => '0');
9          temp_transit_magnitude <= (others => '0');
10     elsif (CLK'event and CLK='1') then
11         if DATA_VALID = '1' then
12             -- Mise à jour des pixels
13             temp_pixels(7 downto 0) <= transit(7 downto 0);
14             temp_pixels(15 downto 8) <= transit(15 downto 8);
15             temp_pixels(23 downto 16) <= transit(23 downto 16);
16             temp_pixels(31 downto 24) <= transit(31 downto 24);
17             temp_pixels(39 downto 32) <= transit(39 downto 32);
18             temp_pixels(47 downto 40) <= transit(47 downto 40);
19             temp_pixels(55 downto 48) <= transit(55 downto 48);
20             temp_pixels(63 downto 56) <= transit(63 downto 56);
21             temp_pixels(71 downto 64) <= transit(71 downto 64);
22
23             -- Calcul des gradients
```

```
            Gx <= (-1 * to_integer(unsigned(temp_pixels(71 downto 64))) +
                  ( 1 * to_integer(unsigned(temp_pixels(55 downto 48))) +
                  (-2 * to_integer(unsigned(temp_pixels(47 downto 40))) +
                  ( 2 * to_integer(unsigned(temp_pixels(31 downto 24))) +
                  (-1 * to_integer(unsigned(temp_pixels(23 downto 16))) +
                  ( 1 * to_integer(unsigned(temp_pixels(7 downto 0))));

            Gy <= ( 1 * to_integer(unsigned(temp_pixels(71 downto 64))) +
                  ( 2 * to_integer(unsigned(temp_pixels(63 downto 56))) +
                  ( 1 * to_integer(unsigned(temp_pixels(55 downto 48))) +
                  (-1 * to_integer(unsigned(temp_pixels(23 downto 16))) +
                  (-2 * to_integer(unsigned(temp_pixels(15 downto 8))) +
                  (-1 * to_integer(unsigned(temp_pixels(7 downto 0))));

            -- Calcul de la magnitude
            temp_transit_magnitude <= std_logic_vector(to_unsigned(abs(Gx) + abs(Gy), 8));

            -- Activation du multiplexeur
            enable_temp <= '1';
        else
            -- Désactiver le multiplexeur si pas de données valides
            enable_temp <= '0';
        end if;
    end if;
end process;
```

En lançant une simulation avec ce filtrage, On obtient ce résultat :

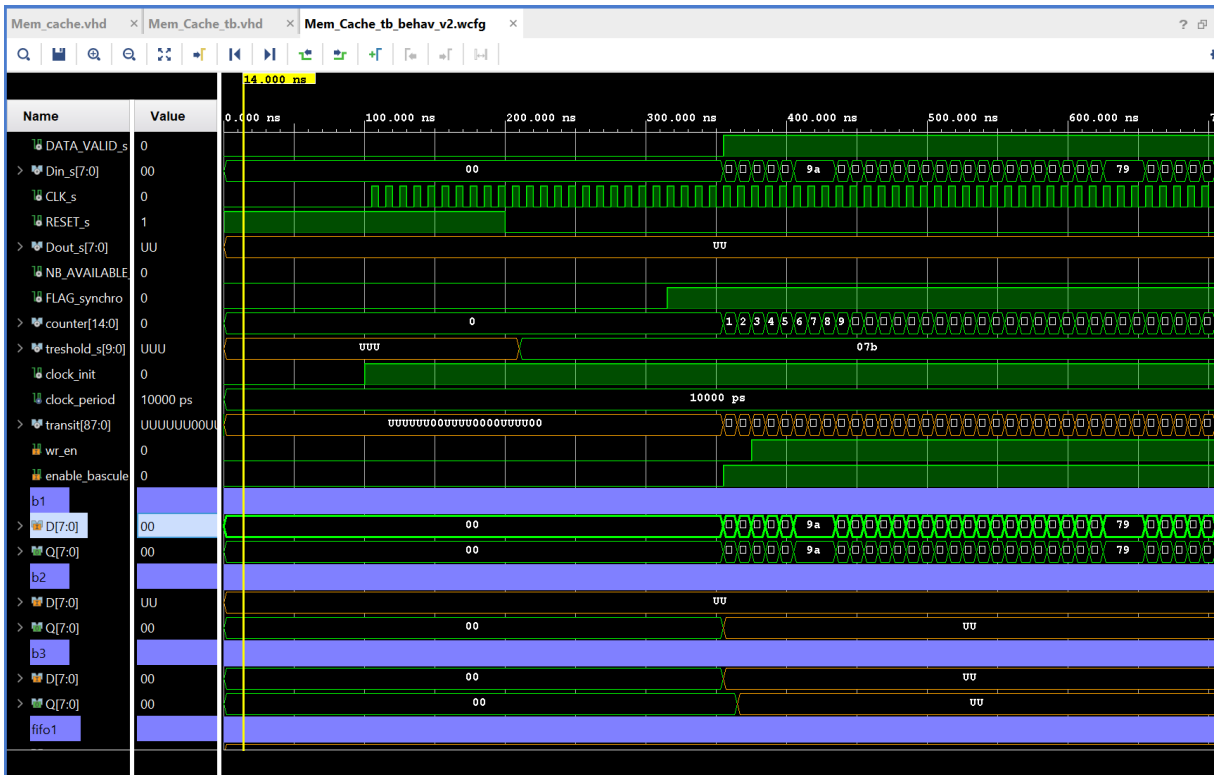


FIGURE 1.12 – Synthèse simulation Sobel

Les données ne passent plus dans les bascules avec le filtrage. Le problème provient

certainement de l'utilisation des signaux internes.

Il m'a manqué du temps pour debugger ce dernier problème et m'empêche de finaliser ce filtrage.
J'aurais aimé terminer ce projet et je suis certains que j'aurais fini par trouver le problème dans le code.
La prochaine fois, il me faudra mieux gérer le temps à allouer au projet.

2 Annexe

2.1 Code-source / Document...

Le dépôt public GitHub contenant le code source du projet se trouve sur ce lien : `https://github.com/HelleEvan/-ESIREM---Projet_VHDL_Helle_Evan`