

A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

01/06/2023

Interface VGA

Rapport de validation (projet finale)

Several thin, curved lines in dark blue and light grey originate from the bottom left and curve upwards and to the right.

Atmani Hicham & Kamal Kherchouch
FORMATION SAFRAN CHEZ AJC

Date	Version	Remarque	Auteur
15/06/2023	A0	Rapport de validation en accord avec le plan de validation version A2 du 15/06/2023	AHI & KKH
30/06/2023	A1	Mise à jour du rapport de validation avec ajout des résultats de la solution finale	AHI & KKH

Table des matières

1.	Création et simulation du module de synchronisation	4
1.1.	Code VHDL du module synchronisation	4
1.2.	Code VHDL du test Bench pour le module de synchronisation.....	4
1.3.	Résultat de simulation du module synchronisation	5
1.3.1.	Validation du fonctionnement du reset	5
1.3.2.	Validation de la création de l'horloge	5
1.3.3.	Validation du signal H_SYNC	6
1.3.4.	Validation du signal V_SYNC	7
2.	Ajout et simulation du module PLL au projet.....	8
2.1.	Code VHDL du module PLL	8
2.2.	Code VHDL du test Bench pour le module PLL.....	8
2.3.	Résultat de simulation du module PLL au fichier top du projet.....	9
2.3.1.	Validation de la création de l'horloge	9
2.3.2.	Validation de l'horloge VGA	9
3.	Ajout et simulation du module Pattern_VGA au projet	10
3.1.	Code VHDL du module Pattern_VGA	10
3.2.	Code VHDL du test Bench pour la validation du module Pattern_VGA	11
3.3.	Résultat de simulation du module Pattern_VGA au fichier top du projet	11
4.	Test de la solution intermédiaire	13
4.1.	Validation de la synthèse	13
4.2.	Validation de l'implémentation.....	14
4.3.	Génération du bitstream et test à l'oscilloscope	15
4.3.1	Validation du signal H_SYNC	15
4.3.2.	Validation du signal V_SYNC	16
4.3.3.	Représentation de ligne à deux et trois parties blanche.....	17
4.4.	Démonstration de la partie intermédiaire	18
5.	Ajout et simulation du module Filtre_Gauss au projet (finale).....	19
5.1.	Code VHDL du module Filtre_Gauss.....	19
5.2.	Code VHDL du test Bench pour la validation du module Filtre_Gauss	20
5.3.	Résultat de simulation du module Filtre_Gauss au fichier top du projet	21
5.3.1.	Validation du remplissage des registres.....	21
5.3.2.	Validation de l'écriture et de lecture des registres (FIFOs).....	22
5.3.3.	Validation de la convolution Gaussienne	23
5.3.4.	Validation de l'activation de l'image	23
6.	Test de la solution finale	24

6.1.	Validation de la Synthèse	24
6.2.	Validation de l'implémentation.....	24
6.3.	Génération du Bitstream et test à l'oscilloscope	25
6.3.1.	Validation du signal H_SYNC_filtre	25
6.3.2.	Validation du signal V_SYNC_filtre	26
6.4.	Démonstration de la partie finale	27

1. Création et simulation du module de synchronisation

1.1. Code VHDL du module synchronisation

La première étape lors que la création de notre module de synchronisation est de rédiger l'entité de notre boîte noire en accord avec la proposition présenté dans le plan de validation.

On obtient l'entité suivante :

```
entity Synchro is
  port (
    clk           : in std_logic;
    resetn        : in std_logic;
    H_SYNC        : out std_logic;
    V_SYNC        : out std_logic;
    Position_Horizontal : out integer;
    Position_Vertical   : out integer
  );
end Synchro;
```

Ensuite on passe à la rédaction de l'architecture de notre module de synchronisation.

Dans cette architecture, nous créons deux signaux internes :

- S_Position_Horizontal : entier allant de 0 à 799 (soit 800)
- S_Position_Vertical : entier allant de 0 à 524 (soit 525)

Ces deux variables correspondront à la dimension de notre image complète (visible et invisible) et sont affectées aux sorties respectives Position_Horizontal et Position_Vertical.

On créer un processus qui sera sensible aux signaux reset et horloge. Lorsque le reset sera activé, on remet les signaux H_SYNC, V_SYNC, S_Position_Horizontal et S_Position_Vertical à 0. Sinon lorsque le reset ne sera pas activé, on réalise les incrémentations de nos variables S_Position_Horizontal et S_Position_Vertical à chaque front montant de notre horloge. Des que nos variables atteignent nos valeurs max, ils reviennent à 0. Ainsi on définit la taille de notre image complète.

Pour continuer, on génère les mises à 0 de nos signaux H_SYNC et V_SYNC, ces derniers devront respectivement intervenir entre [655 : 751] et entre [489 : 491].

1.2. Code VHDL du test Bench pour le module de synchronisation

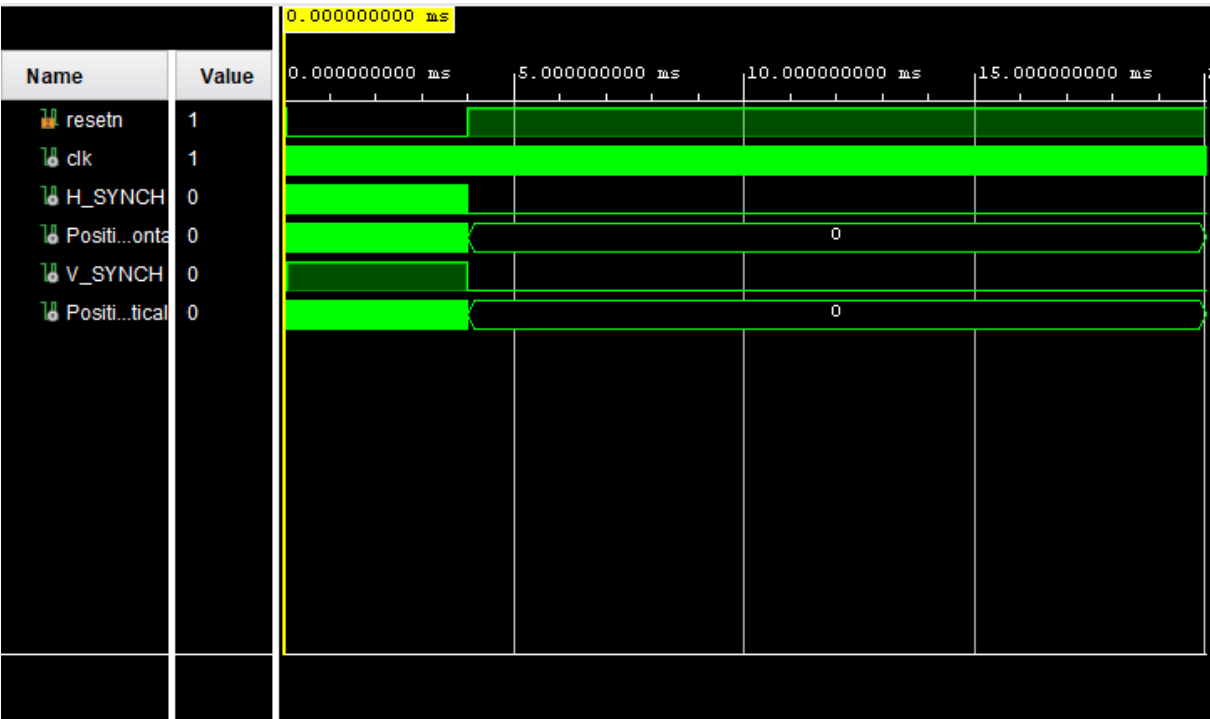
On réalise le test Bench pour valider notre module synchronisation en simulation. Pour ce faire, on réalise une horloge cadencée à 25.175 MHz. Ce qui donne une période de 39.72ns et un temps à l'état haut hp de 19.86ns.

Dans notre test bench on retrouvera deux processus, un pour l'horloge et un pour la gestion de notre reset.

Le reste sera désactivé pour permettre de visualiser deux séquences de V_SYNC.

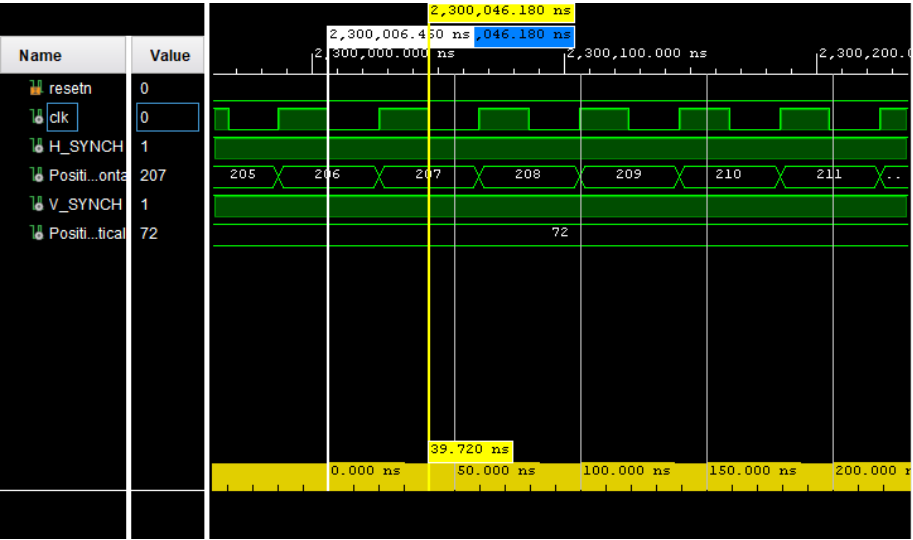
1.3. Résultat de simulation du module synchronisation

1.3.1. Validation du fonctionnement du reset



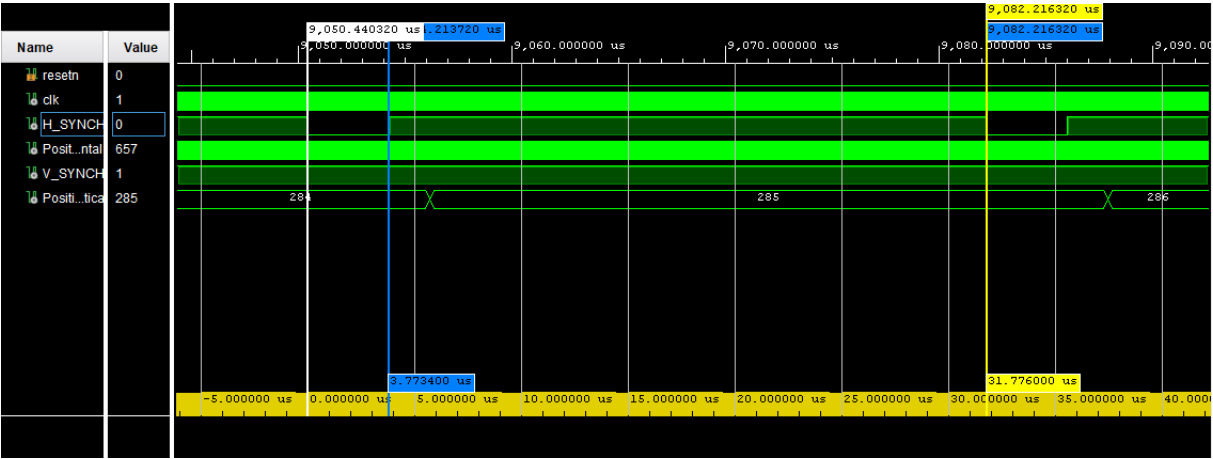
Le fonctionnement du reset est correct, car lorsque le reset passe à 1, les signaux H_SYNCH et V_SYNCH passe directement à 0 et les compteurs Position_Horizontal et Position_Vertical passe également à 0.

1.3.2. Validation de la création de l'horloge

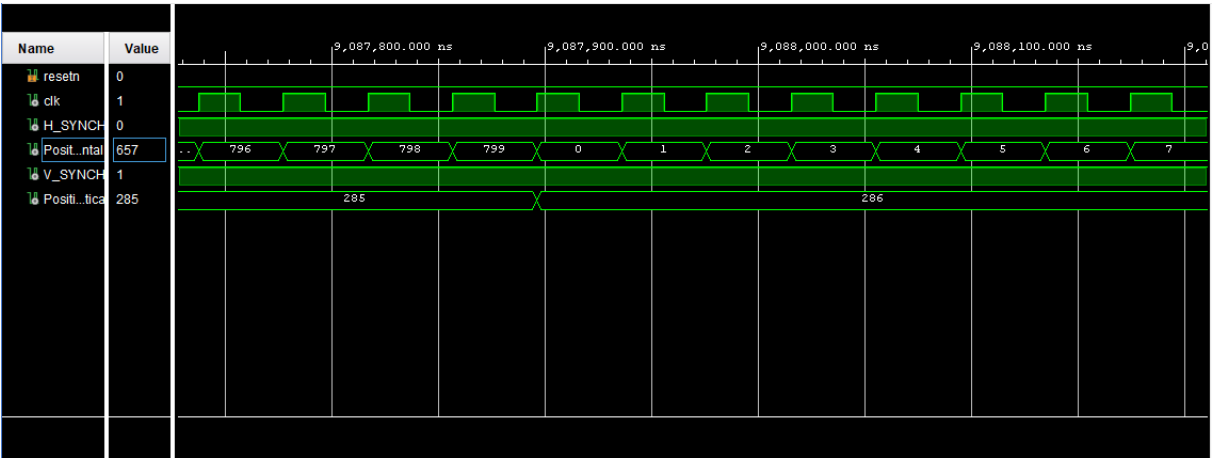


Notre période est de 39.72ns avec un rapport cyclique à 50%, ce qui correspond bien à une horloge cadencée à 25.175MHz comme souhaité.

1.3.3. Validation du signal H_SYNC

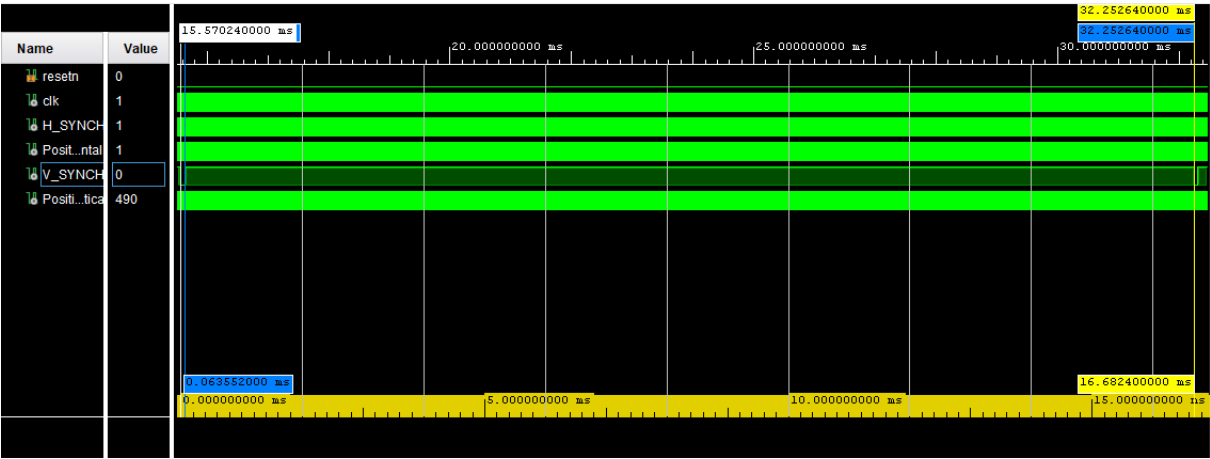


Ici on voit bien que notre « Pulse » sur le signal H_SYNC dure 3.7734µs ce qui correspond à notre 3.8µs attendu comme décrit dans la partie **Erreur ! Source du renvoi introuvable.**
On voit également que le signal complet de H_SYNC dure 31.7351µs ce qui correspond également à notre 31.778µs attendu comme décrit dans la partie **Erreur ! Source du renvoi introuvable.**

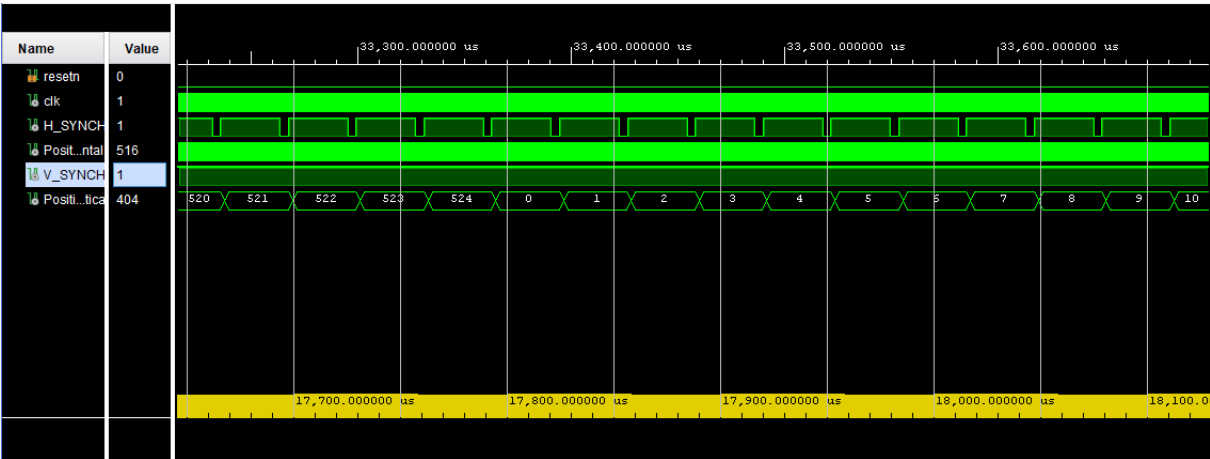


Le compteur compte bien de 0 à 799 comme demandé.

1.3.4. Validation du signal V_SYNC



Ici on voit bien que notre « Pulse » sur le signal V_SYNC dure 0.063552ms ce qui correspond à notre 64ms attendu comme décrit dans la partie **Erreur ! Source du renvoi introuvable.**
On voit également que le signal complet de V_SYNC dure 16.6824ms ce qui correspond également à notre 16.68ms attendu comme décrit dans la partie **Erreur ! Source du renvoi introuvable.**



Le compteur compte bien de 0 à 524 comme demandé.

2. Ajout et simulation du module PLL au projet

2.1. Code VHDL du module PLL

Pour la création du module PLL nous utilisons le composant « Clocking Wizard » présent dans le catalogue IP proposé par Vivado.

Comme expliqué dans le plan de validation nous prenons une horloge d'entrée de **clk = 125 MHz** (horloge fournie par la carte Cora Z7).

Une fois notre module PLL créé, il nous faut l'implémenter dans notre code.

Pour ce faire on ajoute le composant dans l'architecture de notre système, comme suit :

```
-- Composant PLL
component clock_VGA
    port (
        clk_125MHz : in std_logic;
        reset       : in std_logic;
        clk_25MHz   : out std_logic;
        locked      : out std_logic
    );
end component;
```

Ensuite, il nous faut affecter les signaux de notre module PLL aux signaux de notre projet :

```
--Affectation des signaux Pour le module Synchro
clock_VGA0 : clock_VGA
    port map (
        clk_125MHz => clk,
        reset      => resetn,
        clk_25MHz  => clk_Vga,
        locked     => Locked_PLL_resetAuto
    );
```

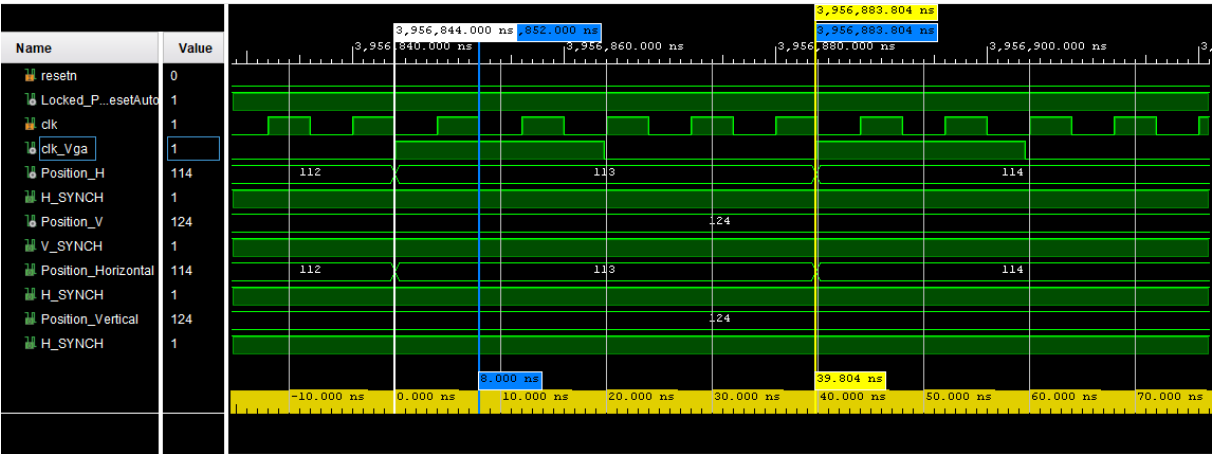
2.2. Code VHDL du test Bench pour le module PLL

On réalise le test Bench pour valider notre module PLL en simulation. Pour ce faire, on réalise une horloge cadencée à 125 MHz. Ce qui donne une période de 8 ns et un temps à l'état haut hp de 4 ns.

2.3. Résultat de simulation du module PLL au fichier top du projet

2.3.1. Validation de la création de l'horloge

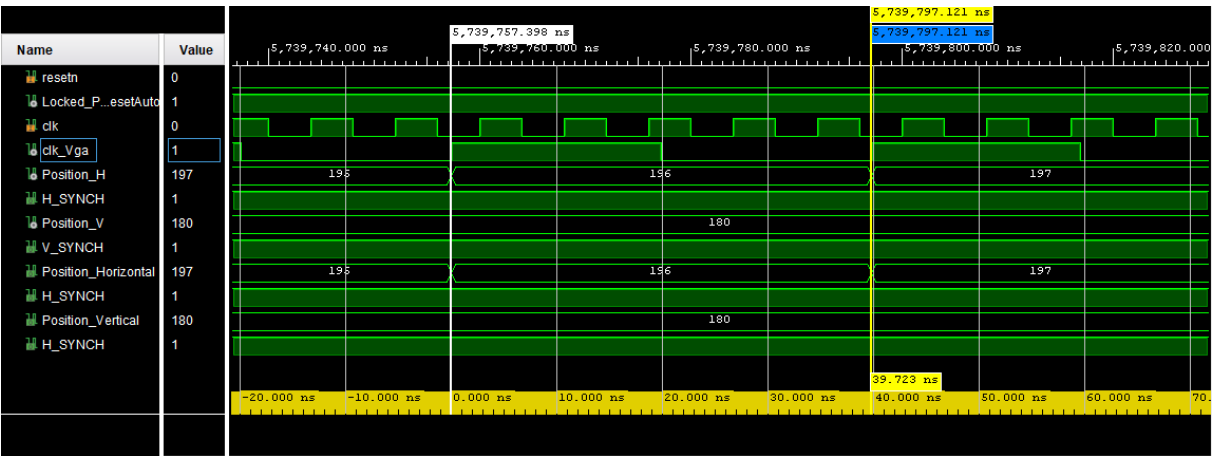
Dans un premier temps, nous allons valider que nous créons une horloge clk cadencé à 125 MHz.



Ici, on voit bien que notre période est de 8 ns soit une fréquence de 125 MHz. La création de notre horloge est correcte.

2.3.2. Validation de l'horloge VGA

Dans un second temps, nous allons valider que la PLL nous délivre bien un signal d'horloge cadencé à 25.175MHz.



Ici, on voit bien que notre période est de 39.723 ns soit une fréquence de 25.175 MHz. La création de notre horloge VGA est correcte.

3. Ajout et simulation du module Pattern_VGA au projet

3.1. Code VHDL du module Pattern_VGA

Pour réaliser le module Pattern_VGA, on crée déjà la boîte noire en y retrouvant :

- En entrée :
 - o L'Horloge : cadencé à l'horloge de fonctionnement du VGA
 - o Le reset : ce qui permettra de réinitialiser le système
 - o La localisation de pixel en horizontale et verticale
- En sortie :
 - o Nos intensités de couleurs pour le rouge, le vert et le bleu

Le code est donc le suivant :

```
entity Pattern_VGA is
  port (
    clk          : in std_logic;
    resetn       : in std_logic;
    Position_H   : in integer;
    Position_V   : in integer;
    Rouge_Intensite : out std_logic_vector(3 downto 0);
    Vert_Intensite  : out std_logic_vector(3 downto 0);
    Bleu_Intensite  : out std_logic_vector(3 downto 0);
  );
end Pattern_VGA;
```

Ensuite on passe à la rédaction de l'architecture de notre module Pattern_VGA.

Dans cette architecture, nous créons deux signaux internes :

- Intensite_faible : qui est un std_logic_vector de 4 bits
- Intensite_haute : qui est un std_logic_vector de 4 bits

Ces deux variables correspondent à la représentation d'une intensité minimum (avec le signal égale à « 0000 ») et d'une intensité maximum (avec le signal égale à « 1111 »).

On crée un processus qui sera sensible aux signaux reset et horloge. Lorsque le reset sera activé, on remet les signaux de couleur rouge, vert et bleu à une intensité faible.

Sinon lorsque le reset ne sera pas activé, on réalise la création de notre damier sur la partie visible de l'image (640 x 480). Le reste des pixels (partie non visible de l'image) sont fixées à une intensité faible.

On ajoute le module à notre fichier top du projet pour ce faire, on ajoute le nouveau composant :

```
-- Composant Pattern_VGA
component Pattern_VGA
  port (
    clk          : in std_logic;
    resetn       : in std_logic;
    Position_H   : in integer;
    Position_V   : in integer;
    Rouge_Intensite : out std_logic_vector(3 downto 0);
    Vert_Intensite  : out std_logic_vector(3 downto 0);
    Bleu_Intensite  : out std_logic_vector(3 downto 0);
  );
end component;
```

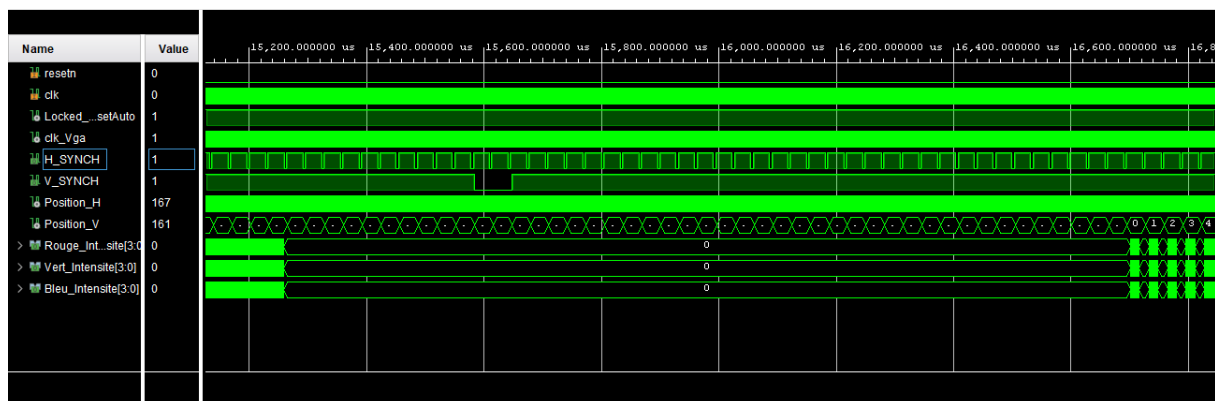
Puis on affecte nos signaux internes du module aux différents signaux de notre projet :

```
--Affectation des signaux Pour le module Synchro
Pattern_Damier : Pattern_VGA
    port map (
        clk            => clk_Vga,
        resetn         => activation_synchro,
        Position_H      => Position_H,
        Position_V      => Position_V,
        Rouge_Intensite => Rouge_Intensite,
        Vert_Intensite  => Vert_Intensite,
        Bleu_Intensite  => Bleu_Intensite
    );
```

3.2. Code VHDL du test Bench pour la validation du module Pattern_VGA

On réalise le test Bench pour valider notre module Pattern_VGA en simulation. Pour ce faire, on réalise une horloge cadencée à 125 MHz. Ce qui donne une période de 8 ns et un temps à l'état haut hp de 4 ns.

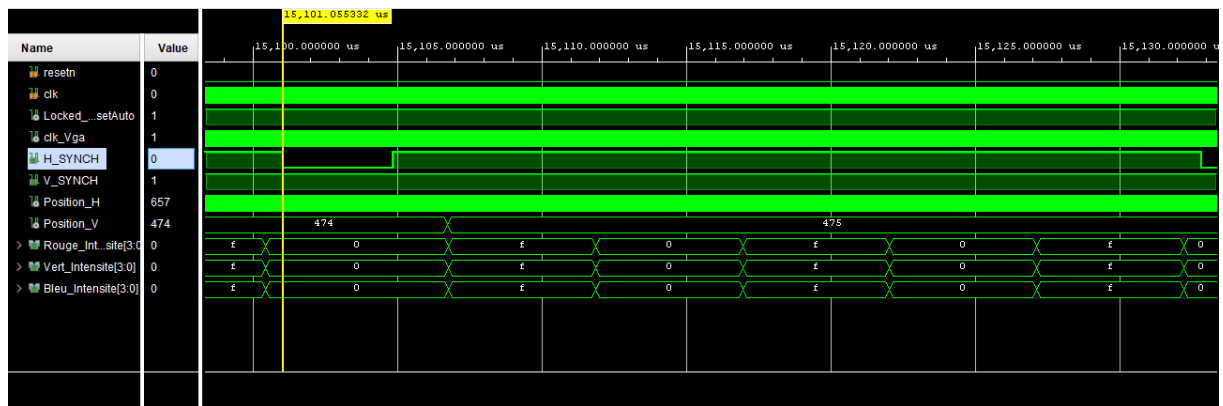
3.3. Résultat de simulation du module Pattern_VGA au fichier top du projet



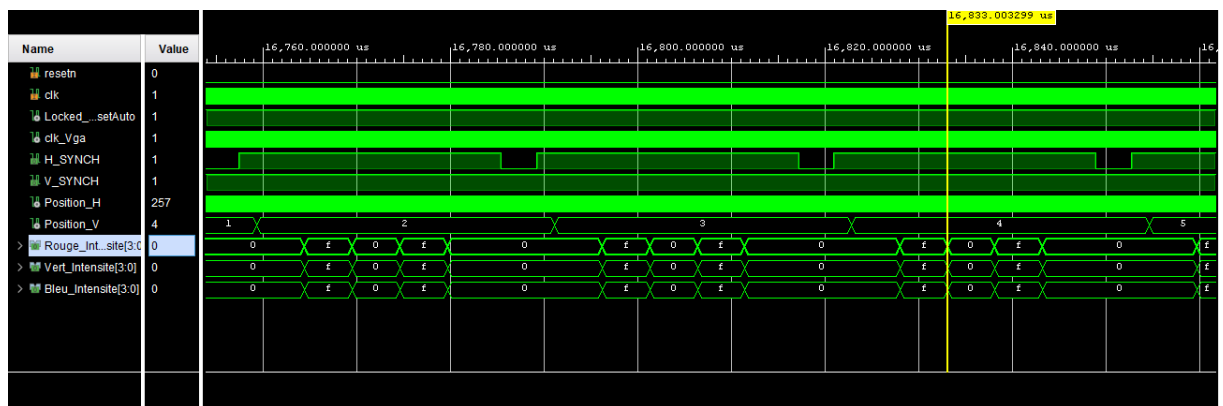
Sur cette simulation, on voit bien que la norme VGA est bien respectée. En effet les couleurs rouges, vert et bleu passe sont forcé à 0 lorsque l'on se trouve dans les plages non visible de notre image, soit :

- 10 pixels verticaux pour le front porch (10 cycles sur H_SYNC)
- 2 pixels pour la largeur de synchronisation (moment ou V_SYNC est à 0) (2 Cycle de H_SYNC)
- 33 pixels verticaux pour le Back porch (33 cycles sur H_SYNC)

Représentation d'une ligne avec 3 parties blanches :



Représentation d'une ligne avec 2 parties blanches :

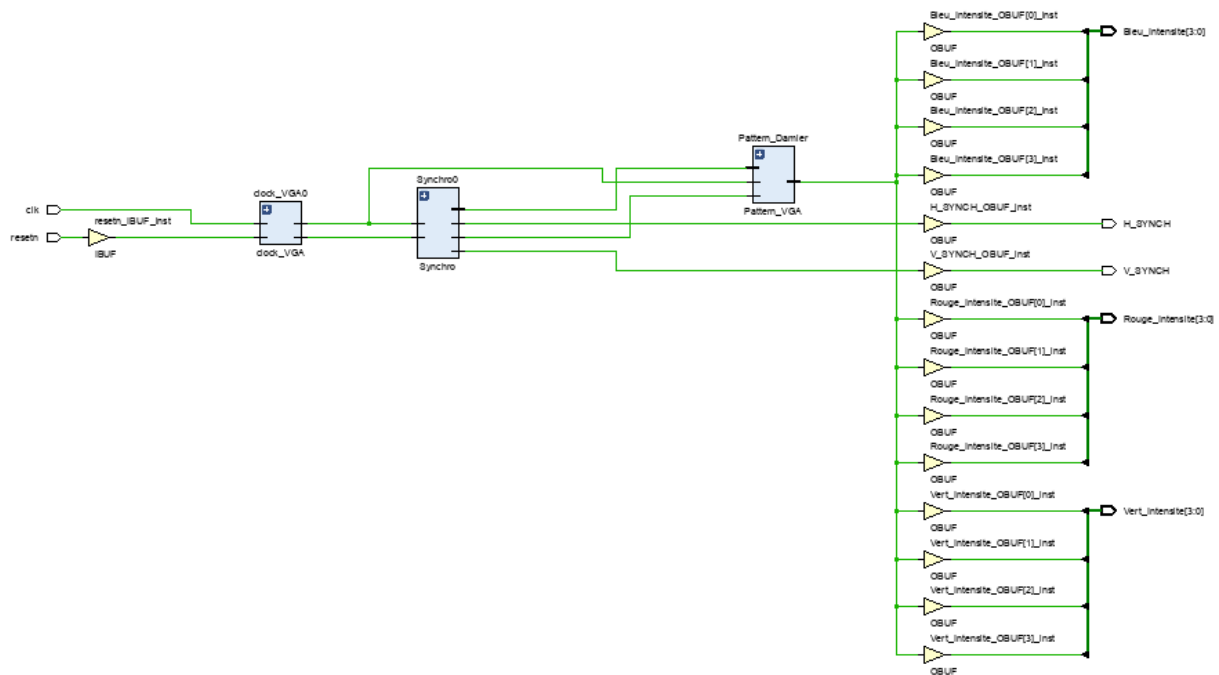


4. Test de la solution intermédiaire

La validation de notre solution étant terminée, nous pouvons passer au test de notre solution. Pour ce faire nous allons réaliser la programmation de notre carte Cora et valider les différents signaux via un oscilloscope.

4.1. Validation de la synthèse

La première étape, pour générer un Bitstream, consiste à lancer une implémentation. L'implémentation nous permet également de voir le schéma de notre projet.



Ici, on voit bien que l'on a :

- Deux signaux d'entrée
 - o Clk (1 bit)
 - o Resetrn (1 bit)
- Cinq signaux de sortie
 - o H_Synch (1 bit)
 - o V_SYNC_H (1 bit)
 - o Rouge_intensite (4 bits)
 - o Vert_intensite (4 bits)
 - o Bleu_intensite (4 bits)
- Trois modules
 - o Module PLL
 - o Module Synchro
 - o Module Pattern_VGA

On peut donc en déduire que notre solution respecte les attentes demandées.

4.2. Validation de l'implémentation

L'implémentation nous permet de valider que nous n'avons pas de violation de timing.

Design Timing Summary

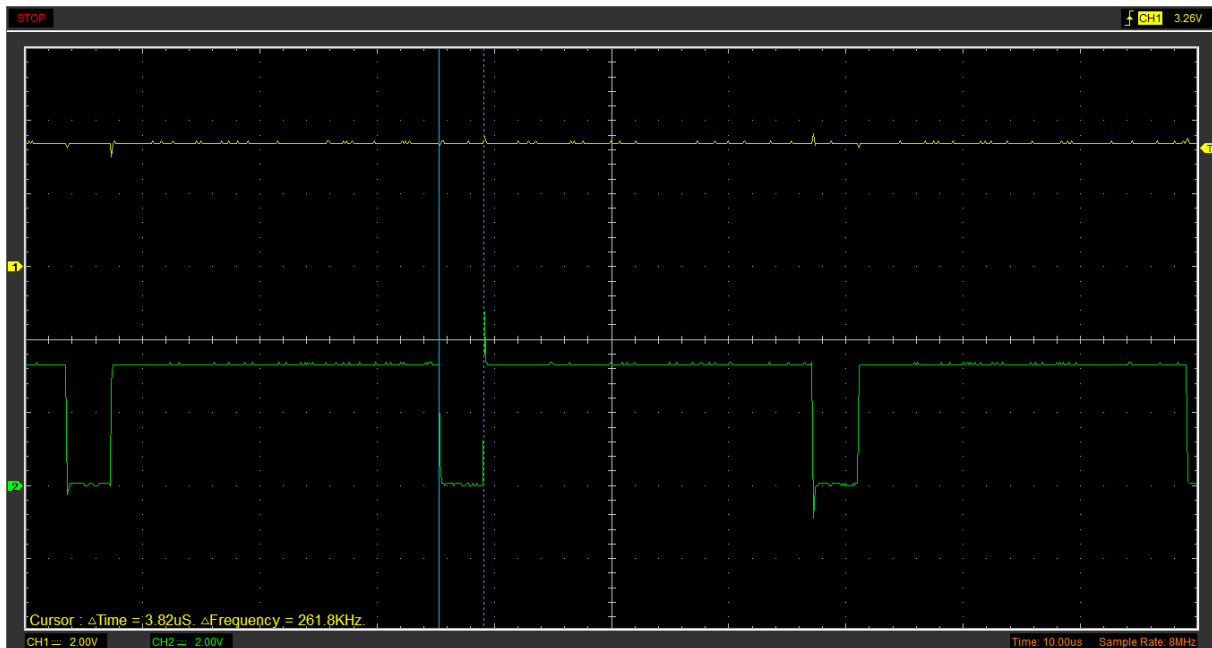
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 36,282 ns	Worst Hold Slack (WHS): 0,092 ns	Worst Pulse Width Slack (WPWS): 2,000 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 44	Total Number of Endpoints: 44	Total Number of Endpoints: 40
All user specified timing constraints are met.		

Dans notre cas, tout est correct puisque nous n'avons pas de stack.

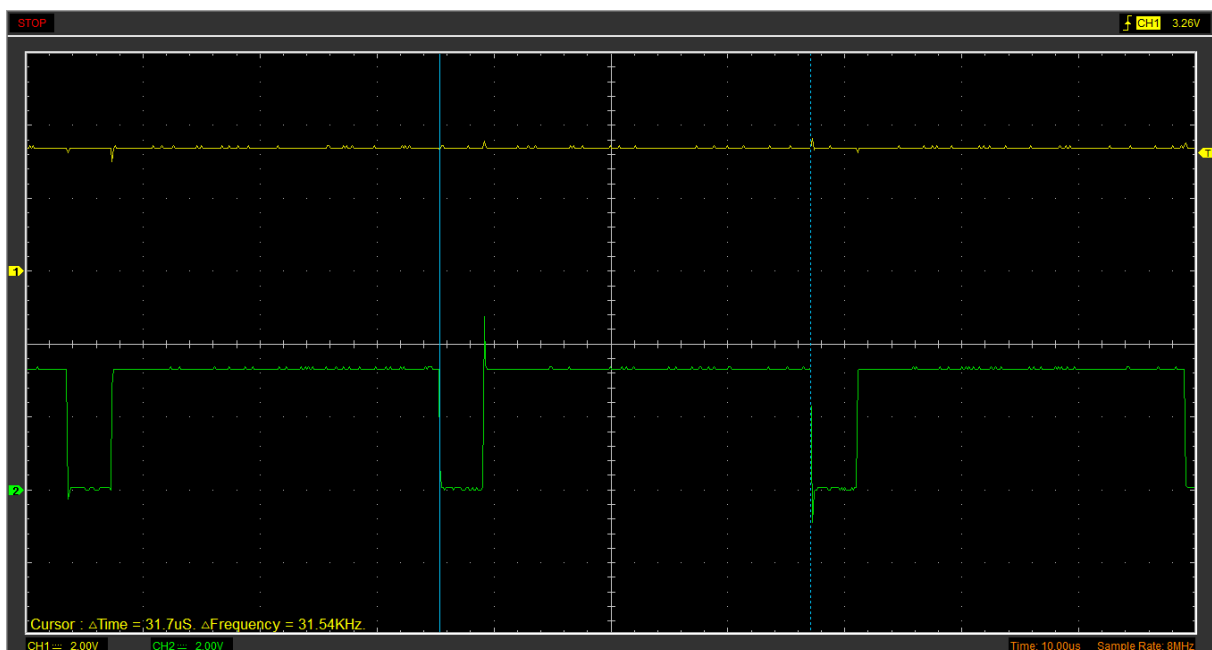
4.3. Génération du bitstream et test à l'oscilloscope

Le bitstream généré (Générateur_Pattern.bit) est présent dans le dossier document de sortie.

4.3.1 Validation du signal H_SYNC



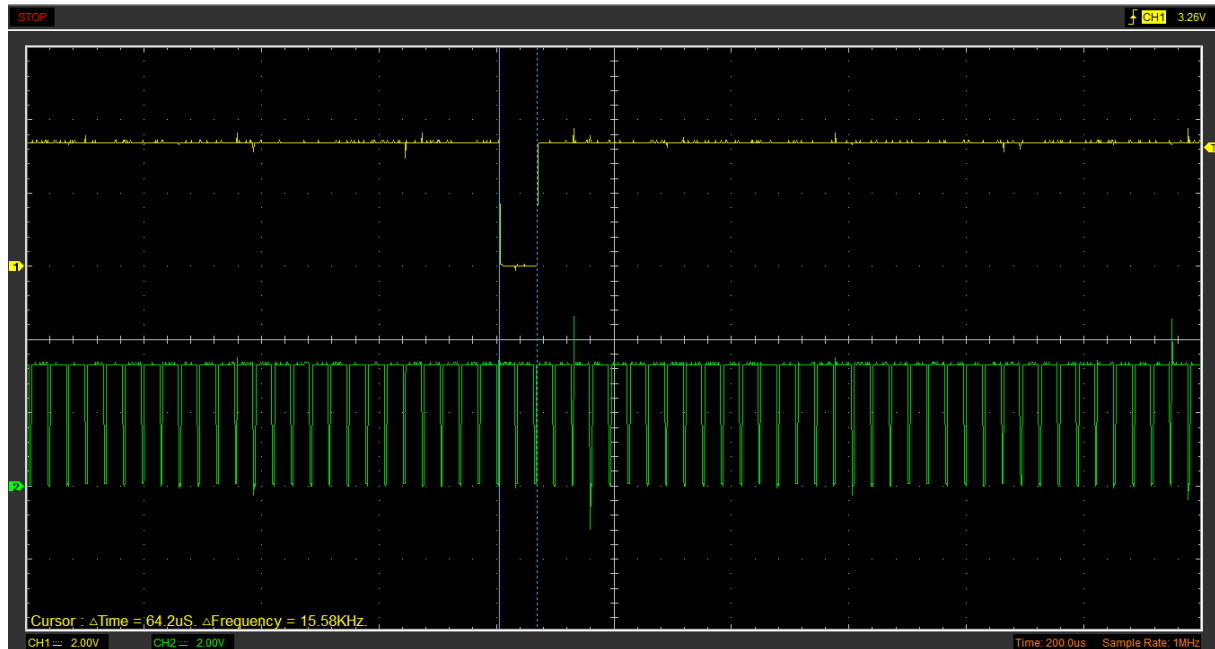
Ici, on voit bien que l'impulsion du signal H_SYNC est de 3.82µs ce qui correspond à notre attente.



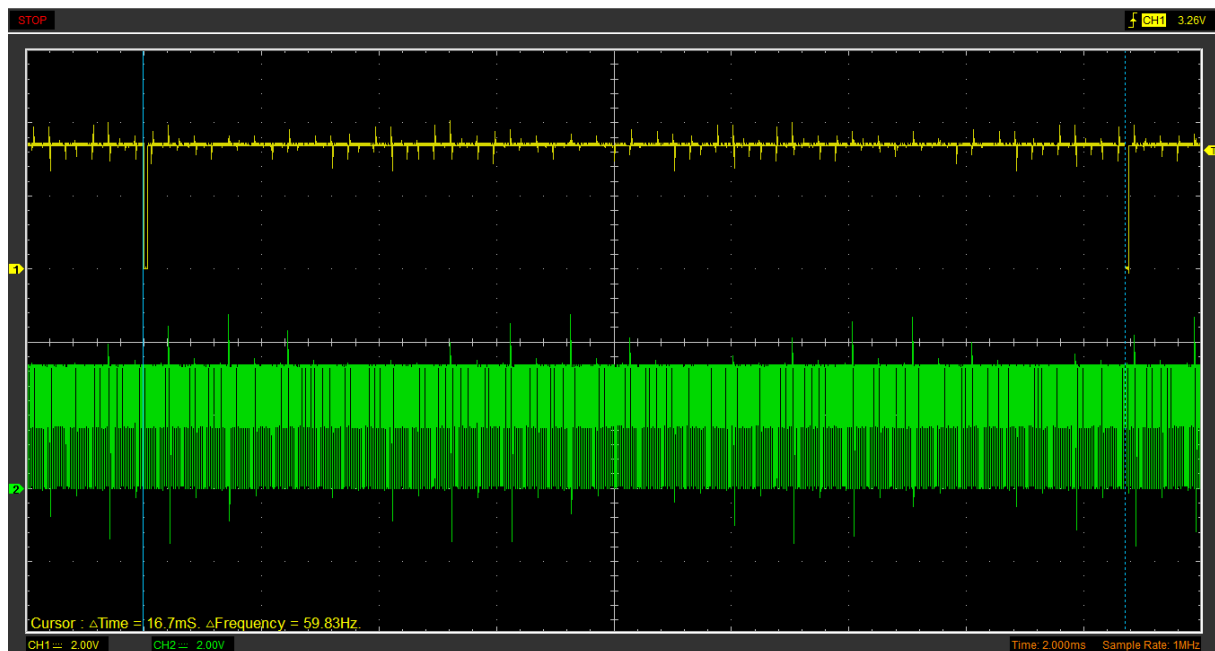
Et que la période du signal H_SYNC est de 31.7µs ce qui est conforme également.

Le signal H_SYNC est donc correct à notre attente.

4.3.2. Validation du signal V_SYNC



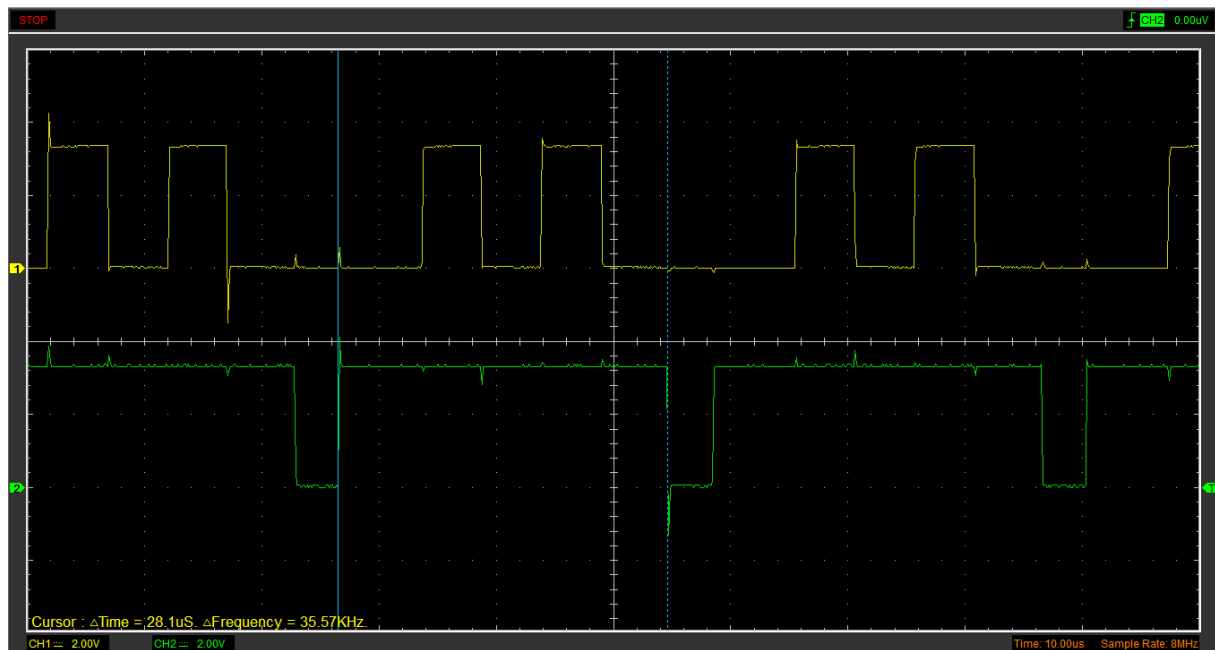
Ici, on voit bien que l'impulsion du signal V_SYNC est de $64.2\mu\text{s}$ ce qui correspond à notre attente.



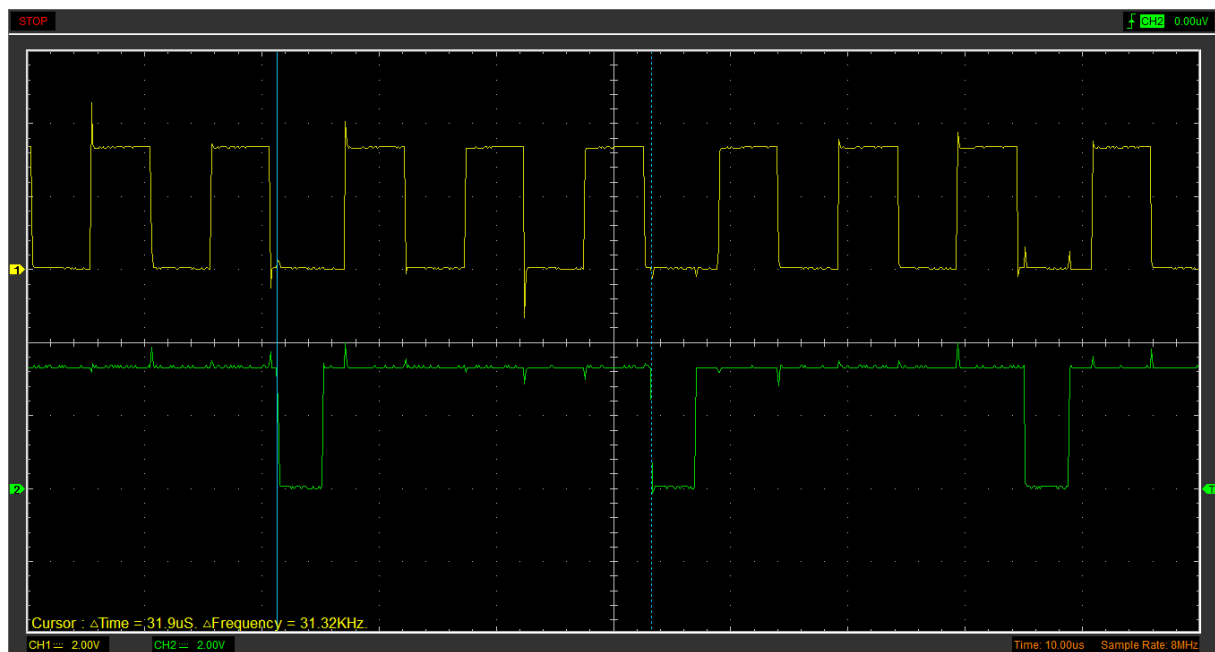
Et que la période du signal V_SYNC est de 16.7ms ce qui est conforme également.

Le signal V_SYNC est donc correct à notre attente.

4.3.3. Représentation de ligne à deux et trois parties blanche

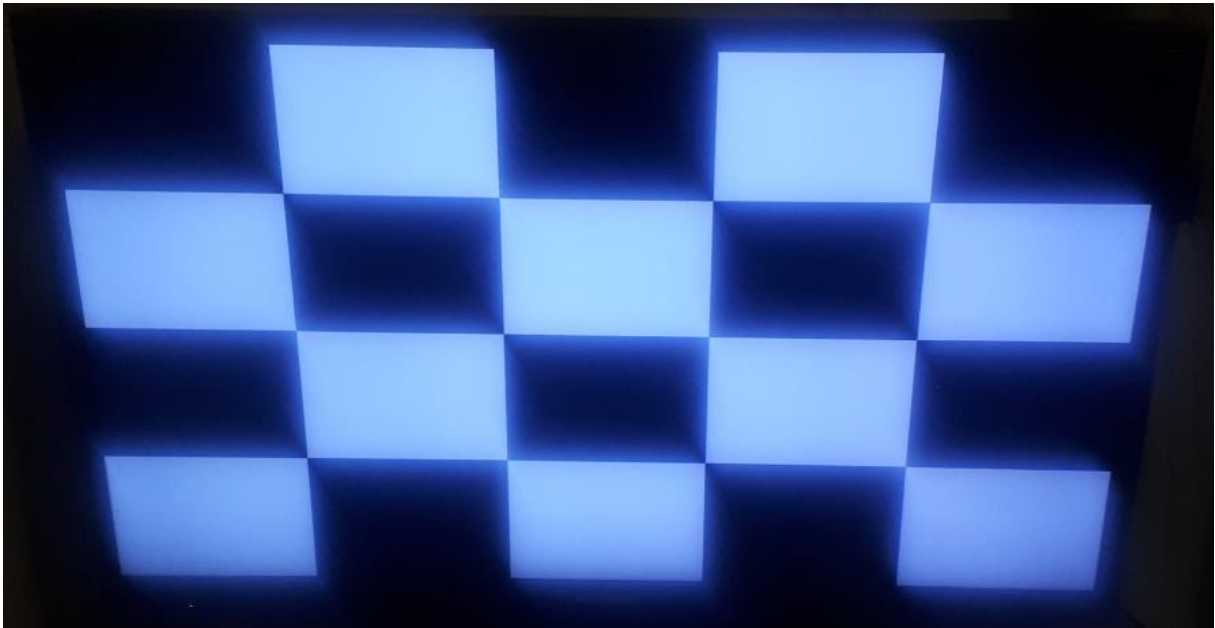


Ici, on voit bien la représentation d'une ligne à deux parties blanches.



Ici, on voit bien la représentation d'une ligne à trois parties blanches.

4.4. Démonstration de la partie intermédiaire



La projection de notre pattern sur l'afficheur nous permet de valider notre solution intermédiaire car on voit bien que :

- Le damier est correctement configurer avec cinq colonnes et quatre lignes avec des lignes à 2 parties blanches et des lignes à 3 parties blanches
- Qu'aucune ligne n'est décalée car les carrés blanc sont bien placé au bonne endroit
- Notre damier reste bien dans la partie visible de notre afficheur.

De plus, lorsque l'on appuie sur le bouton 0 de notre carte cora, le reset s'active, et l'afficheur s'éteint.

5. Ajout et simulation du module Filtre_Gauss au projet (finale)

5.1. Code VHDL du module Filtre_Gauss

Pour réaliser le module Filtre_Gauss, on crée déjà la boîte noire en y retrouvant :

- En entrée :
 - o L'Horloge : cadencée à l'horloge de fonctionnement du VGA
 - o Le reset : ce qui permettra de réinitialiser le système
 - o La localisation de pixel en horizontale et verticale
 - o L'intensité des couleurs provenant du module Pattern_VGA
- En sortie :
 - o Nos intensités de couleurs pour le signal rouge, le vert et le bleu
 - o Nos signaux de synchronisation

Le code est donc le suivant :

```
entity Filtre_Gauss is
    port (
        clk           : in std_logic;
        resetn        : in std_logic;
        Position_H     : in integer;
        Position_V     : in integer;
        Rouge_Intensite : in std_logic_vector(3 downto 0);
        Vert_Intensite  : in std_logic_vector(3 downto 0);
        Bleu_Intensite  : in std_logic_vector(3 downto 0);
        Rouge          : out std_logic_vector(3 downto 0);
        Vert           : out std_logic_vector(3 downto 0);
        Bleu           : out std_logic_vector(3 downto 0);
        H_SYNCH_filtre : out std_logic;
        V_SYNCH_filtre : out std_logic
    );
end Filtre_Gauss;
```

Ensuite on passe à la rédaction de l'architecture de notre module Filtre_Gauss.

Dans cette architecture, nous créons deux signaux internes :

- activation_ecriture_fifo_1 : activation de l'écriture de la Fifo 1 (1 bit)
- activation_ecriture_fifo_2 : activation de l'écriture de la Fifo 2 (1 bit)
- activation_lecture_fifo_1 : activation de la lecture de la Fifo 1 (1 bit)
- activation_lecture_fifo_2 : activation de la lecture de la Fifo 2 (1 bit)
- Convolution_Bleu : permet le calcul de convolution de la couleur bleu (8 bits)
- Convolution_Vert : permet le calcul de convolution de la couleur vert (8 bits)
- Convolution_Rouge : permet le calcul de convolution de la couleur rouge (8 bits)
- Concatenation_Couleur : concaténation des couleurs rouge, vert et bleu (12 bits)
- Compteur_Fifo : compteur permettant la gestion de l'activation et désactivation de la lecture et l'écriture des fifos (1bit)
- Out_Registre_Fifo1 : signal de sortie pour la fifo 1 (12 bits);
- Out_Registre_Fifo2 : signal de sortie pour la fifo 2 (12 bits);
- K1 à K9 : constante du cœur de convolution du filtre de Gausse (4 bits)
- reg1 à reg9 : registre permettant de stocker les pixels d'entrées (12bits)

- Intensite_faible : intensité de couleur faible (4 bits)
- Intensite_haute : intensité de couleur forte (4 bits)

Ces deux variables correspondent à la représentation d'une intensité minimum (avec le signal égale à « 0000 ») et d'une intensité maximum (avec le signal égale à « 1111 »).

On ajoute à notre code les deux composants Fifos et on les affecte aux signaux de notre projet :

```

component Fifo_Ligne1
port (
    clk : in std_logic;
    srst : in std_logic;
    din : in std_logic_vector(11 downto 0);
    wr_en : in std_logic;
    rd_en : in std_logic;
    dout : out std_logic_vector(11 downto 0);
    full : out std_logic;
    empty : out std_logic
);
end component;

-- Composant Fifo_Ligne2
component Fifo_Ligne2
port (
    clk : in std_logic;
    srst : in std_logic;
    din : in std_logic_vector(11 downto 0);
    wr_en : in std_logic;
    rd_en : in std_logic;
    dout : out std_logic_vector(11 downto 0);
    full : out std_logic;
    empty : out std_logic
);
end component;

Fifo_1 : Fifo_Ligne1
port map (
    clk => clk,
    srst => reset,
    din => reg3,
    wr_en => activation_ecriture_fifo_1,
    rd_en => activation_lecture_fifo_1,
    dout => Out_Registre_Fifo1
);

Fifo_2 : Fifo_Ligne2
port map (
    clk => clk,
    srst => reset,
    din => reg6,
    wr_en => activation_ecriture_fifo_2,
    rd_en => activation_lecture_fifo_2,
    dout => Out_Registre_Fifo2
);

```

On crée trois processus qui seront sensible aux signaux reset et horloge. Lorsque le reset sera activé, on remet tous les signaux à 0.

Le premier processus permet de mettre à jour les signaux de synchronisation horizontale et verticale. Le but de ce processus est de pouvoir synchroniser notre image autour de reg5 qui est le centre de nos calculs de convolution.

Le second processus permet d'activer et ou désactiver la lecture des fifos ainsi que le remplissage des registres reg1 à reg9.

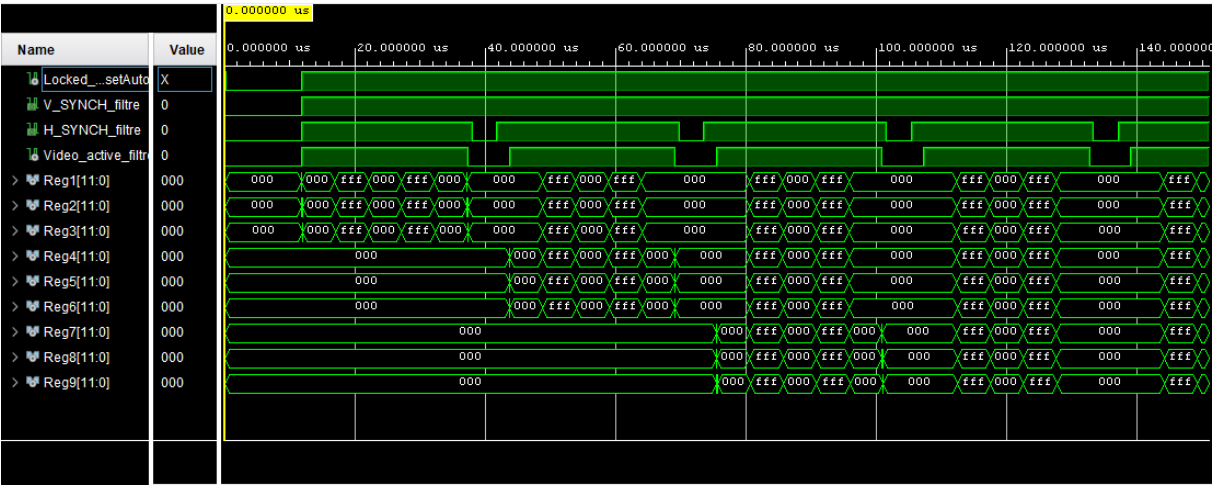
Le dernier processus permet de réaliser les calculs de convolution et d'affecter les résultats de ces calculs aux signaux de couleurs de sortie.

5.2. Code VHDL du test Bench pour la validation du module Filtre_Gauss

On réalise le test Bench pour valider notre module Filtre_Gauss en simulation. Pour ce faire, on réalise une horloge cadencée à 125 MHz. Ce qui donne une période de 8 ns et un temps à l'état haut hp de 4 ns.

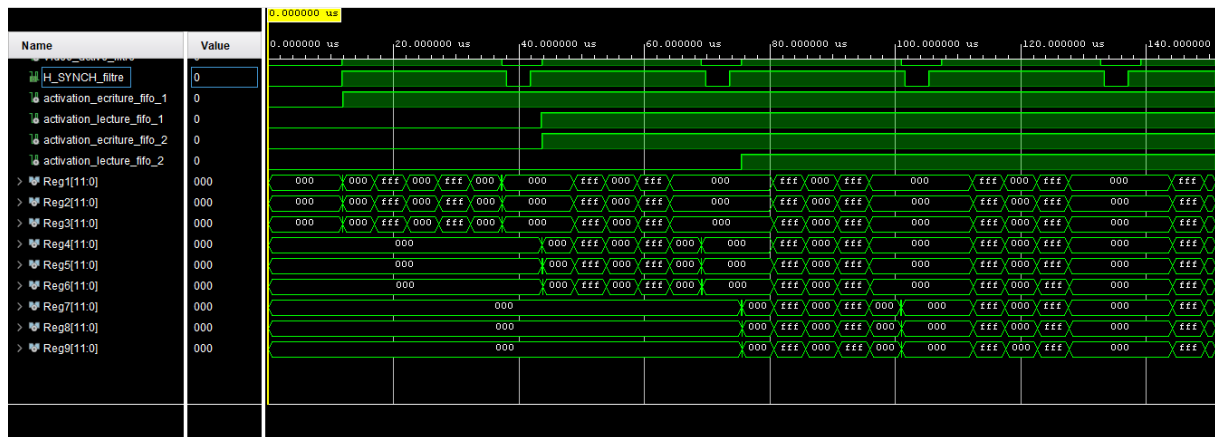
5.3. Résultat de simulation du module Filtre_Gauss au fichier top du projet

5.3.1. Validation du remplissage des registres

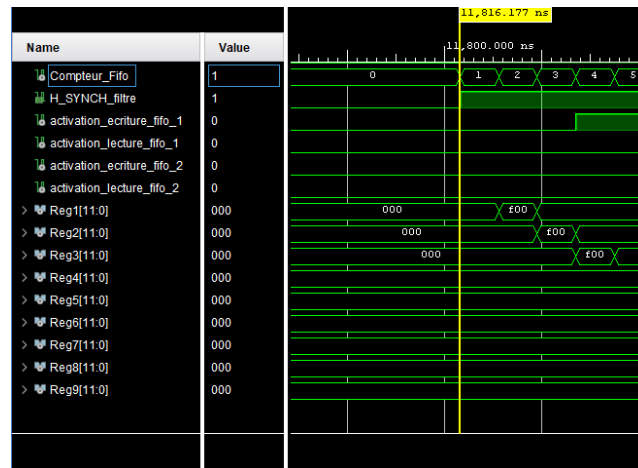


Ici, on voit bien que lorsque l'on est sur la première ligne de l'image on remplit les registres 1 à 3. Lorsque l'on se trouve sur la second ligne de l'image, les registres 1 à 3 sont rempli par les nouvelles valeurs et les anciennes valeurs sont transmis au registre 4 à 6. Et sur la troisième ligne on retrouve le même principe avec les registres 7 à 9, et ainsi de suite.

5.3.2. Validation de l'écriture et de lecture des registres (FIFOs)

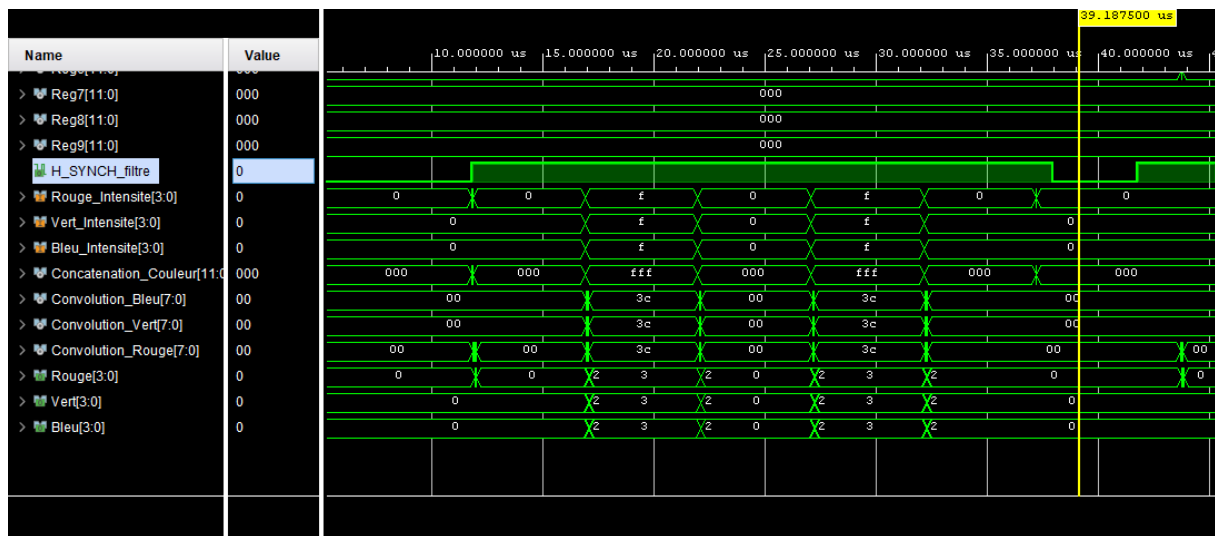


Ici, on voit bien que l'écriture de la Fifo 1 est activée lors de la première ligne de l'image avec un décalage de trois pixels afin de laisser le temps au registre 1 à 3 de se remplir, comme le montre l'image suivante :



On voit également que la lecture de la Fifo 1 et l'écriture de la Fifo 2 sont activés lors de la seconde ligne (avec le même décalage de 3 pixels). Puis lors de la troisième image (avec le décalage de 3 pixels) on active la lecture de la Fifo 2.

5.3.3. Validation de la convolution Gaussienne



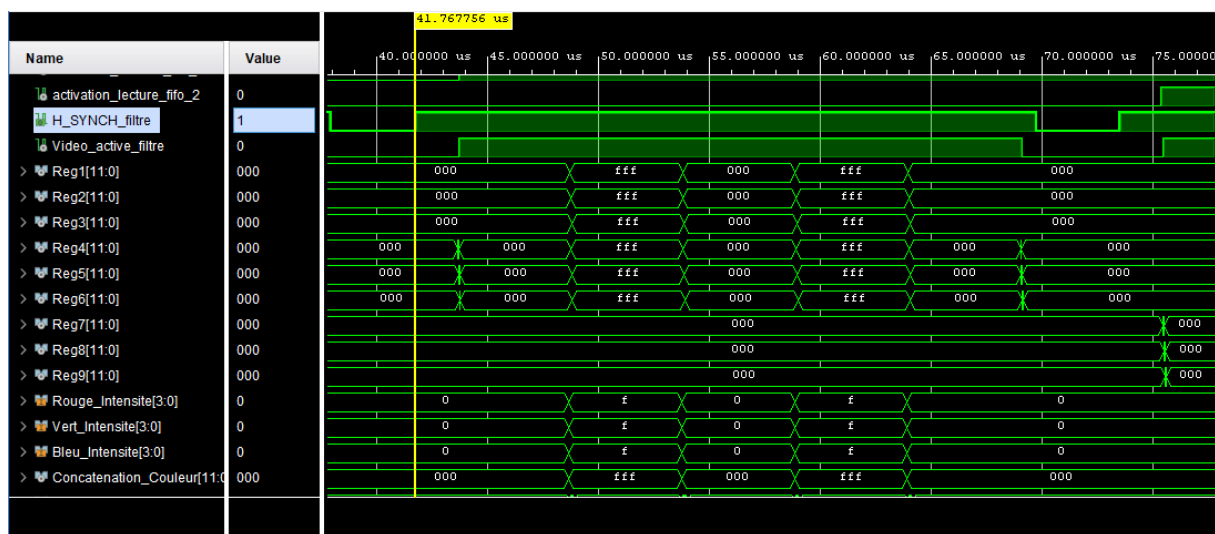
Ici, on voit bien que lors de la première ligne de l'image :

- Les signaux Rouge_intensite, Vert_intensite et Bleu_intensite sont conforme à la sortie de notre module Pattern_VGA (avec une ligne comprenant 3 parties noires et deux parties blanches)

Remarque : sur le signal Rouge_intensite un pixel a été ajouté en début et fin de ligne afin de pouvoir valider la taille de l'image.

- Le signal Concatenation_Couleur prend bien la concaténation des signaux provenant du module Pattern_VGA.
- Les calculs de convolution sont bien sur 8 bits.
- Les signaux de sortie Rouge, Vert et Bleu sont bien impactés par les calculs de convolution et la dé-concaténation de ces derniers.

5.3.4. Validation de l'activation de l'image



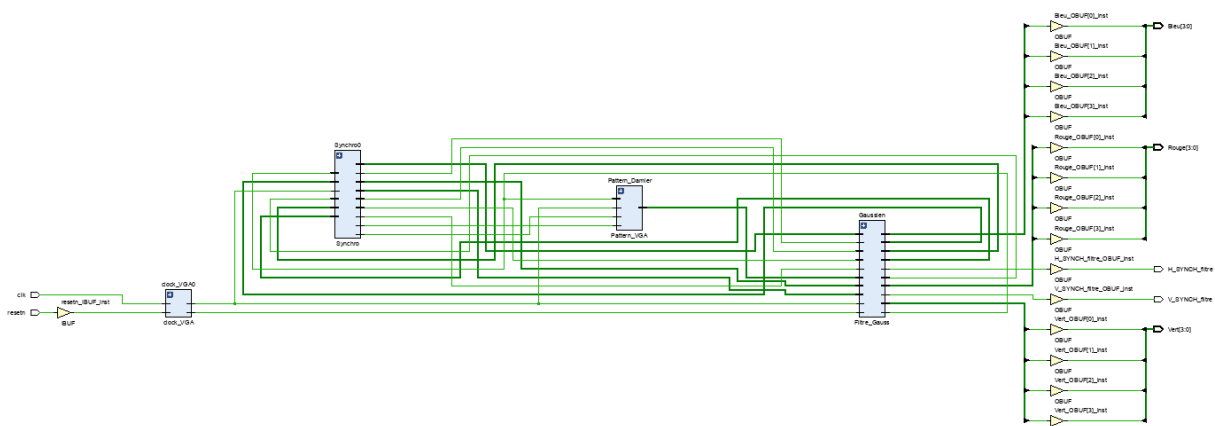
Ici on voit bien que le signal Video_active_filtre passe à 1 lorsque le Back Porch du signal H_SYNC_filtre est passé, et lorsque l'on remplit les différents registres.

6. Test de la solution finale

La validation de notre solution étant terminée, nous pouvons passer au test de notre solution. Pour ce faire nous allons réaliser la programmation de notre carte Cora et valider les différents signaux via un oscilloscope. Une mise à jour du fichier de contrainte a été réalisée afin d'affecter les nouveaux signaux à la carte Cora Z7.

6.1. Validation de la Synthèse

La première étape, pour générer un Bitstream, consiste à lancer une implémentation. L'implémentation nous permet également de voir le schéma de notre projet.



Ici, on voit bien que l'on a :

- Deux signaux d'entrée
 - o Clk (1 bit), Resetn (1 bit)
- Cinq signaux de sortie
 - o H_Synch_filtre (1 bit), V_Synch_filtre (1 bit)
 - o Rouge (4 bits), Vert (4 bits) et Bleu (4 bits)
- quatre modules
 - o Module PLL, Module Synchro, Module Pattern_VGA, Module Filtre_Gauss

On peut donc en déduire que notre solution respect les attentes demandées.

6.2. Validation de l'implémentation

L'implémentation nous permet de valider que nous n'avons pas de violation de timing.

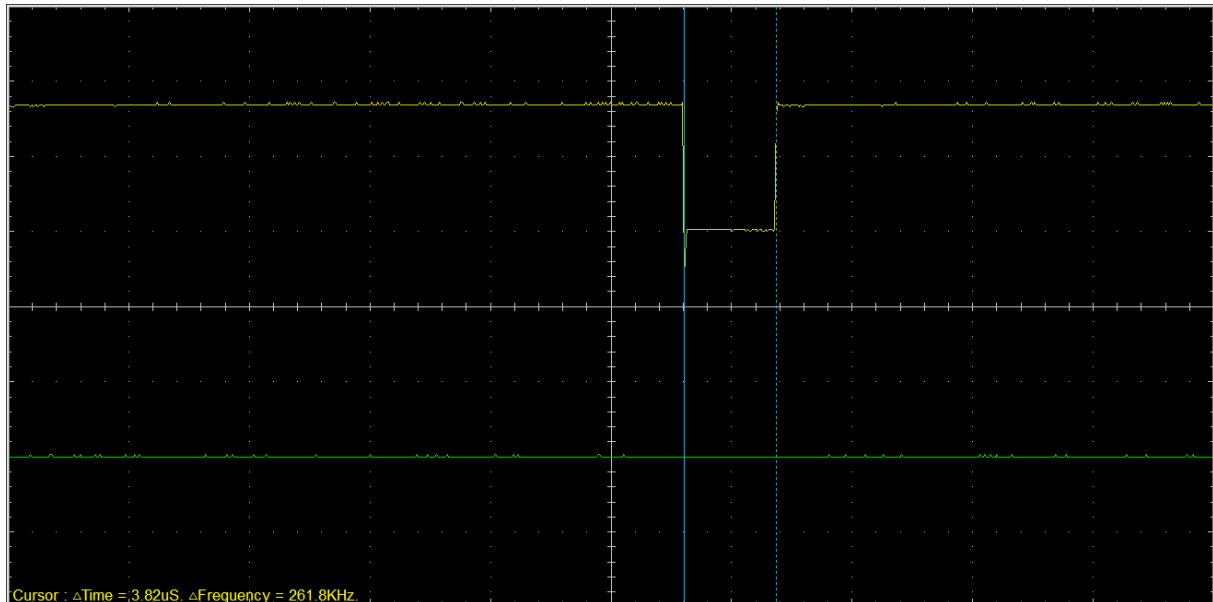
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 30,717 ns	Worst Hold Slack (WHS): 0,031 ns	Worst Pulse Width Slack (WPWS): 2,000 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 454	Total Number of Endpoints: 454	Total Number of Endpoints: 280
All user specified timing constraints are met.		

Dans notre cas, tout est correct puisque nous n'avons pas de stack.

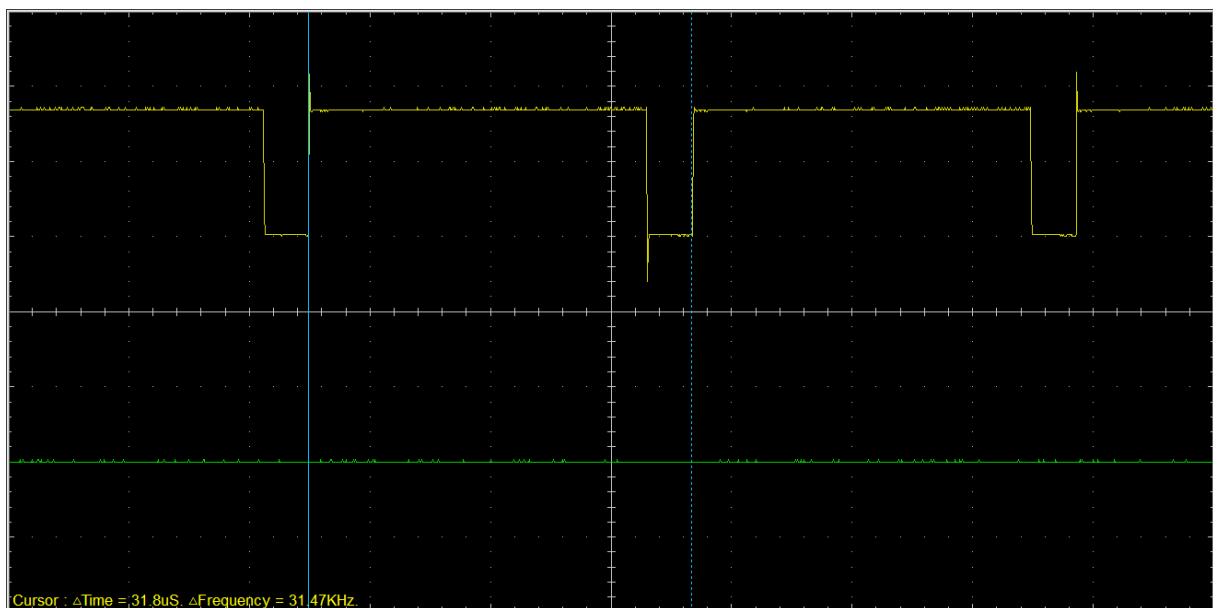
6.3. Génération du Bitstream et test à l'oscilloscope

Le bitstream généré (Porjet_VGA_Gaussien.bit) est présent dans le dossier Bitstream.

6.3.1. Validation du signal H_SYNC_filtre



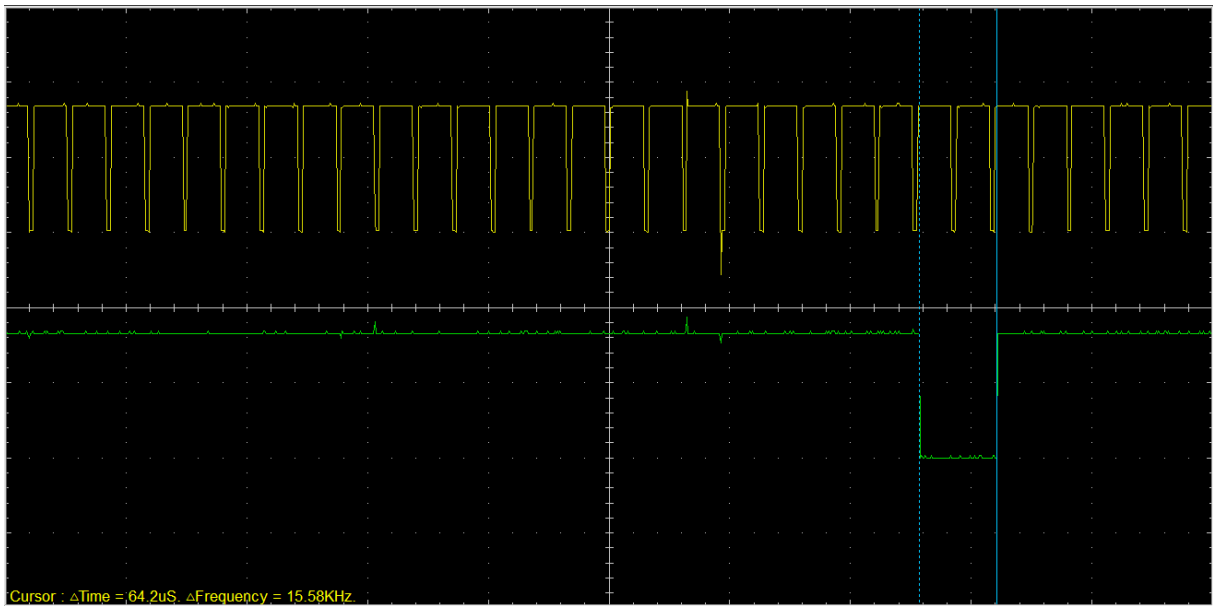
Ici, on voit bien que l'impulsion du signal H_SYNC est de $3.82\mu\text{s}$ ce qui correspond à notre attente.



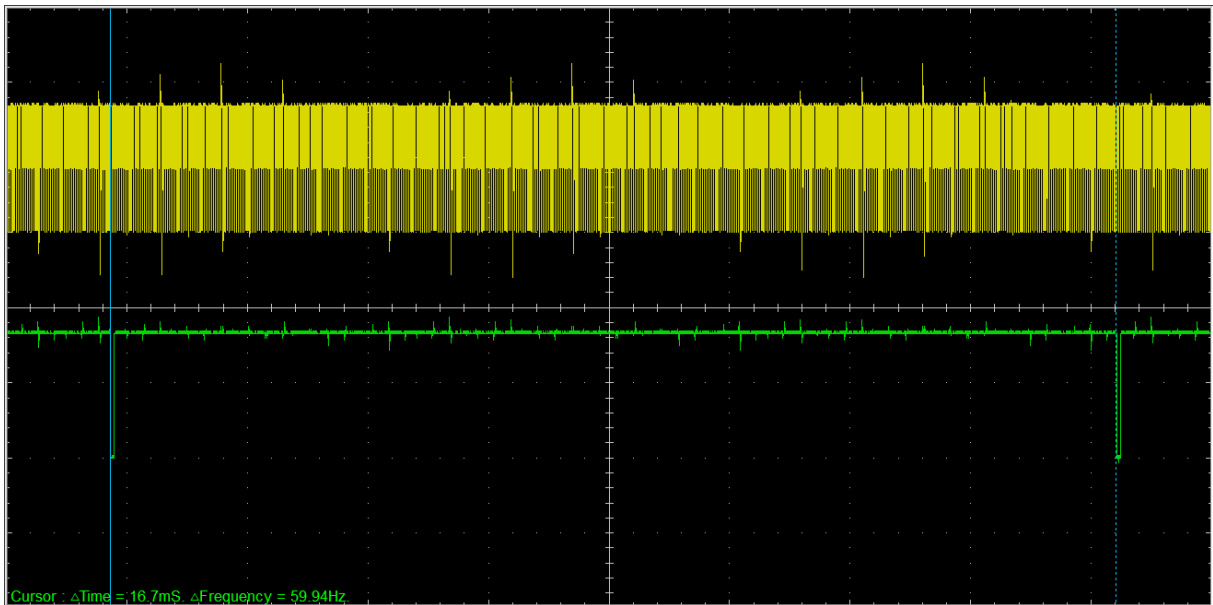
Et que la période du signal H_SYNC est de $31.7\mu\text{s}$ ce qui est conforme également.

Le signal H_SYNC est donc correct à notre attente.

6.3.2. Validation du signal V_SYNC_filtre



Ici, on voit bien que l'impulsion du signal V_SYNC est de $64.2\mu\text{s}$ ce qui correspond à notre attente.



Et que la période du signal V_SYNC est de 16.7ms ce qui est conforme également.

Le signal V_SYNC est donc correct à notre attente.

6.4. Démonstration de la partie finale

Etant compliqué de prendre des photos où le filtre est réellement visible, et pour faire la démonstration de notre projet, nous avons généré un nouveau bitstream avec une légère modification dans le code VHDL.

Dans le dossier Bitstream, il y aura donc :

- Generateur_Pattern.bit : Bitstream de la solution intermédiaire (c'est-à-dire le damier net)
- Porjet_VGA_Gaussien.bit : Bitstream de la solution finale (c'est-à-dire le damier flouté)
- Projet_VGA_filtre_Gaussien_Partie_Haute.bit : bitstream pour valider le projet (c'est-à-dire que le filtre est appliqué que sur les deux premières lignes de rectangle du damier. Cela nous permet de voir la différence entre la partie avec et sans filtre, ce qui n'est pas forcément visible sur la photo prise).

La modification du code consiste à ajouter une condition sur le troisième processus du module Filtre_Gauss, comme voici :

```
elsif (rising_edge(clk)) then
  if(position_H < 640 and position_V < 240) then
    --Compteur_Fifo <= Compteur_Fifo+1;
    Convolution_Bleu <= (reg1(3 downto 0)*k1) + (reg2(3 downto 0)*k2) + (reg3(3 downto 0)*k3) + (reg4(3 downto 0)*k4);
    Convolution_Vert <= (reg1(7 downto 4)*k1) + (reg2(7 downto 4)*k2) + (reg3(7 downto 4)*k3) + (reg4(7 downto 4)*k4);
    Convolution_Rouge <= (reg1(11 downto 8)*k1) + (reg2(11 downto 8)*k2) + (reg3(11 downto 8)*k3) + (reg4(11 downto 8)*k4);
    Bleu <= Convolution_Bleu(7 downto 4);
    Vert <= Convolution_Vert(7 downto 4);
    Rouge <= Convolution_Rouge(7 downto 4);
  elsif(position_H < 640 and position_V < 480) then
    Bleu <= reg5(3 downto 0);
    Vert <= reg5(7 downto 4);
    Rouge <= reg5(11 downto 8);
  else
    Rouge <= Intensite_faible;
    Vert <= Intensite_faible;
    Bleu <= Intensite_faible;
  end if;
end if;
```

La projection de notre pattern sur l'afficheur nous permet de valider notre solution finale car on voit bien que :

- Le damier est correctement configuré avec cinq colonnes et quatre lignes avec des lignes à 2 parties blanches et des lignes à 3 parties blanches
- Qu'aucune ligne n'est décalée car les carrés blancs sont bien placés au bon endroit
- Notre damier reste bien dans la partie visible de notre afficheur.
- Avec le bitstream Porjet_VGA_Gaussien.bit tout semble flouté.
- Avec le bitstream Projet_VGA_filtre_Gaussien_Partie_Haute.bit la moitié haute est bien floutée et la partie basse est nette.

