01/06/2023

Atmani Hicham & Kamal Kherchouch

FOrmation Safran chez AJC

Interface VGA

Rapport de validation (projet finale)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| 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](#_Toc139032463)

[1.1. Code VHDL du module synchronisation 4](#_Toc139032464)

[1.2. Code VHDL du test Bench pour le module de synchronisation 4](#_Toc139032465)

[1.3. Résultat de simulation du module synchronisation 5](#_Toc139032466)

[1.3.1. Validation du fonctionnement du reset 5](#_Toc139032467)

[1.3.2. Validation de la création de l’horloge 5](#_Toc139032468)

[1.3.3. Validation du signal H\_SYNCH 6](#_Toc139032469)

[1.3.4. Validation du signal V\_SYNCH 7](#_Toc139032470)

[2. Ajout et simulation du module PLL au projet 8](#_Toc139032471)

[2.1. Code VHDL du module PLL 8](#_Toc139032472)

[2.2. Code VHDL du test Bench pour le module PLL 8](#_Toc139032473)

[2.3. Résultat de simulation du module PLL au fichier top du projet 9](#_Toc139032474)

[2.3.1. Validation de la création de l’horloge 9](#_Toc139032475)

[2.3.2. Validation de l’horloge VGA 9](#_Toc139032476)

[3. Ajout et simulation du module Pattern\_VGA au projet 10](#_Toc139032477)

[3.1. Code VHDL du module Pattern\_VGA 10](#_Toc139032478)

[3.2. Code VHDL du test Bench pour la validation du module Pattern\_VGA 11](#_Toc139032479)

[3.3. Résultat de simulation du module Pattern\_VGA au fichier top du projet 11](#_Toc139032480)

[4. Test de la solution intermédiaire 13](#_Toc139032481)

[4.1. Validation de la synthèse 13](#_Toc139032482)

[4.2. Validation de l’implémentation 14](#_Toc139032483)

[4.3. Génération du bitstream et test à l’oscilloscope 15](#_Toc139032484)

[4.3.1 Validation du signal H\_SYNCH 15](#_Toc139032485)

[4.3.2. Validation du signal V\_SYNCH 16](#_Toc139032486)

[4.3.3. Représentation de ligne à deux et trois parties blanche 17](#_Toc139032487)

[4.4. Démonstration de la partie intermédiaire 18](#_Toc139032488)

[5. Ajout et simulation du module Filtre\_Gauss au projet (finale) 19](#_Toc139032489)

[5.1. Code VHDL du module Filtre\_Gauss 19](#_Toc139032490)

[5.2. Code VHDL du test Bench pour la validation du module Filtre\_Gauss 20](#_Toc139032491)

[5.3. Résultat de simulation du module Filtre\_Gauss au fichier top du projet 21](#_Toc139032492)

[5.3.1. Validation du remplissage des registres 21](#_Toc139032493)

[5.3.2. Validation de l’écriture et de lecture des registres (FIFOs) 22](#_Toc139032494)

[5.3.3. Validation de la convolution Gaussienne 23](#_Toc139032495)

[5.3.4. Validation de l’activation de l’image 23](#_Toc139032496)

[6. Test de la solution finale 24](#_Toc139032497)

[6.1. Validation de la Synthèse 24](#_Toc139032498)

[6.2. Validation de l’implémentation 24](#_Toc139032499)

[6.3. Génération du Bitstream et test à l’oscilloscope 25](#_Toc139032500)

[6.3.1. Validation du signal H\_SYNCH\_filtre 25](#_Toc139032501)

[6.3.2. Validation du signal V\_SYNCH\_filtre 26](#_Toc139032502)

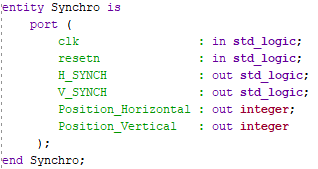
[6.4. Démonstration de la partie finale 27](#_Toc139032503)

# Création et simulation du module de synchronisation

## Code VHDL du module synchronisation

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

On obtient l’entité suivante :



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\_SYNCH, V\_SYNCH, 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\_SYNCH et V\_SYNCH, ces derniers devront respectivement intervenir entre [655 : 751] et entre [489 : 491].

## 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\_SYNCH.

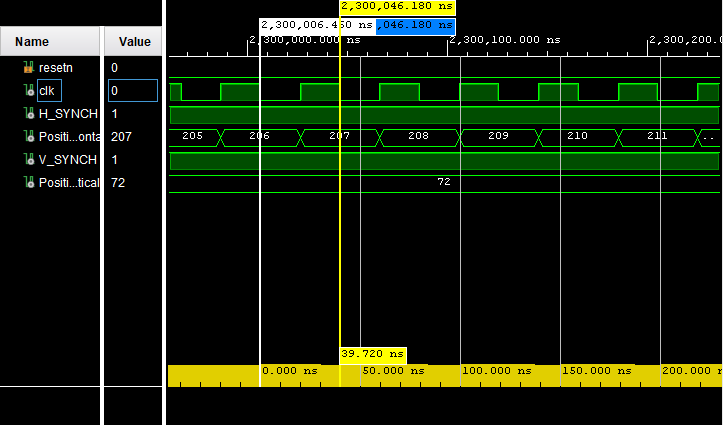
## Résultat de simulation du module synchronisation

### Validation du fonctionnement du reset



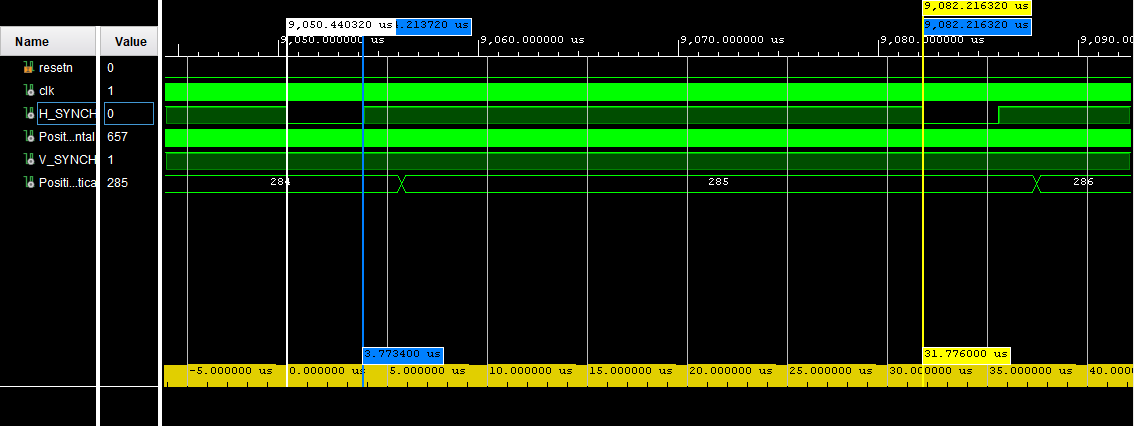
Le fonctionnement du reset est correct, car lorsque le reset passe à 1, les signaux H\_SYNCH et V\_SYNCH passe directement à 0 et les compteurs Position\_Horizontal et Position\_Vertical passe également à 0.

### Validation de la création de l’horloge



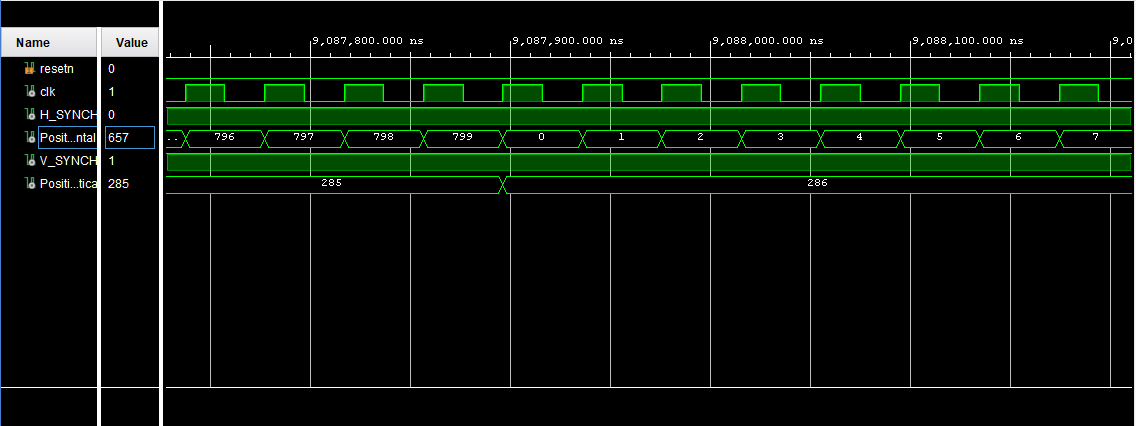
Notre période est de 39.72ns avec un rapport cyclique à 50%, ce qui correspond bien à une horloge cadencée à 25.175MHz comme souhaité.

### Validation du signal H\_SYNCH



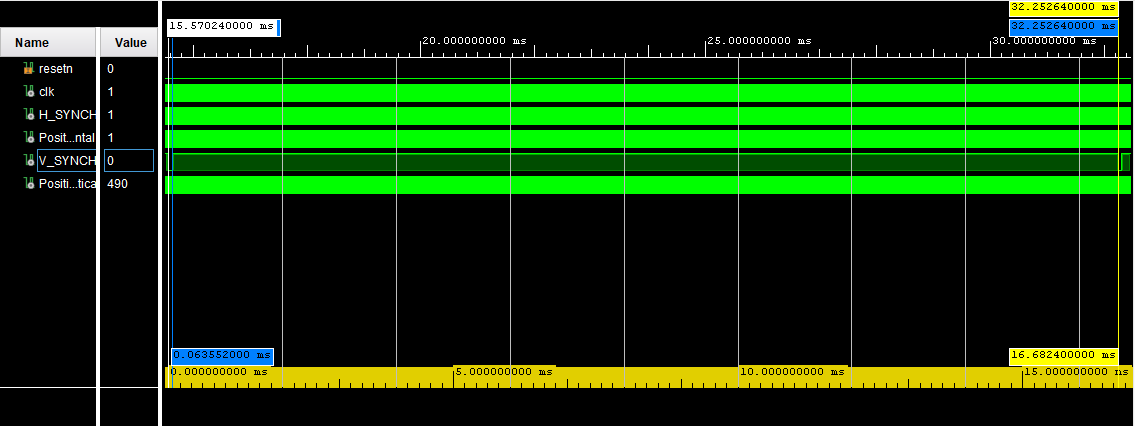
Ici on voit bien que notre « Pulse » sur le signal H\_SYNCH dure 3.7734µs ce qui corresponds à notre 3.8µs attendu comme décrit dans la partie **Erreur ! Source du renvoi introuvable.**.

On voit également que le signal complet de H\_SYNCH dure 31.7351µs ce qui correspond également à notre 31.778µs attendu comme décrit dans la partie **Erreur ! Source du renvoi introuvable.**.



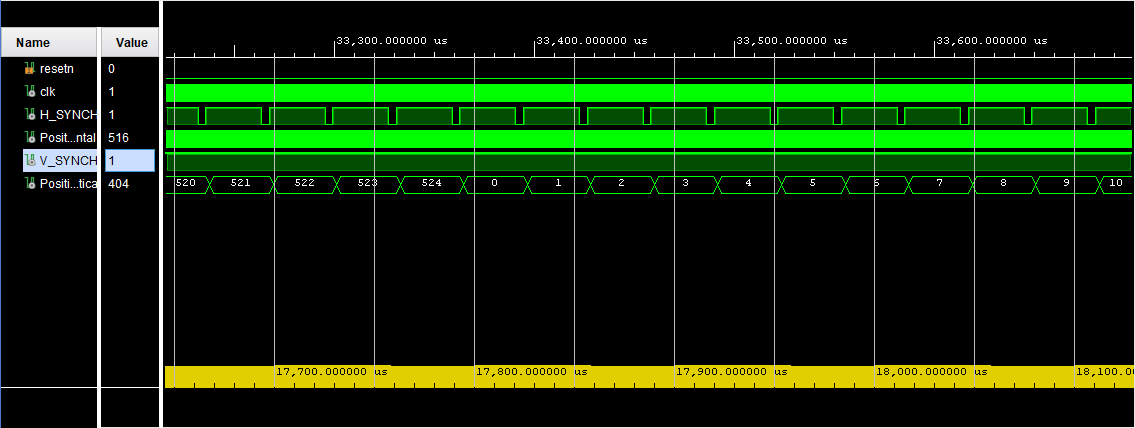
Le compteur compte bien de 0 à 799 comme demandé.

### Validation du signal V\_SYNCH



Ici on voit bien que notre « Pulse » sur le signal V\_SYNCH dure 0.063552ms ce qui corresponds à notre 64ms attendu comme décrit dans la partie **Erreur ! Source du renvoi introuvable.**.

On voit également que le signal complet de V\_SYNCH 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.

# Ajout et simulation du module PLL au projet

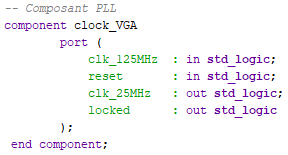
## Code VHDL du module PLL

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

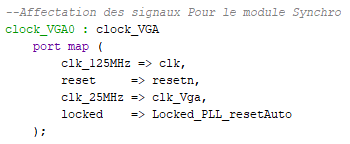
Comme expliqué dans le plan de validation nous prenons une horloge d’entrée de **clk = 125 MHz** (horloge fournit par la carte Cora Z7).

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 :



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



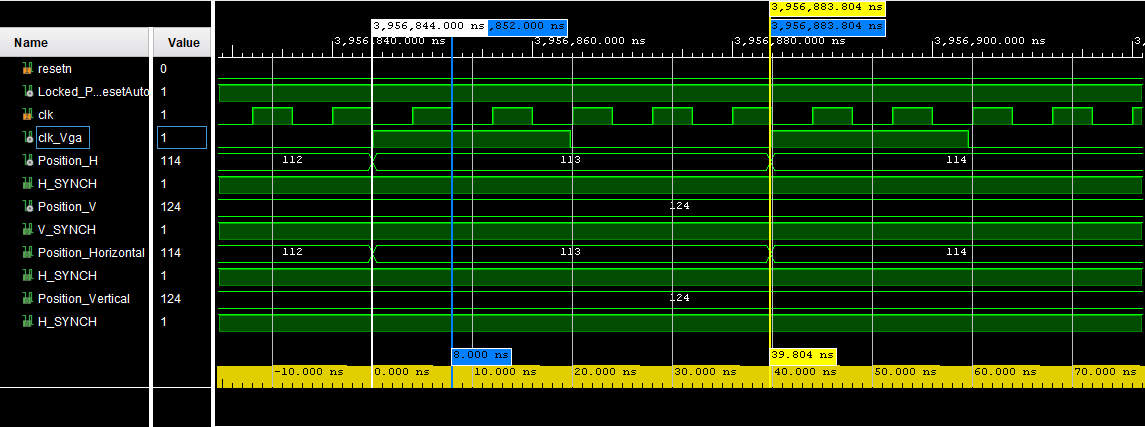
## Code VHDL du test Bench pour le module PLL

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.

## Résultat de simulation du module PLL au fichier top du projet

### Validation de la création de l’horloge

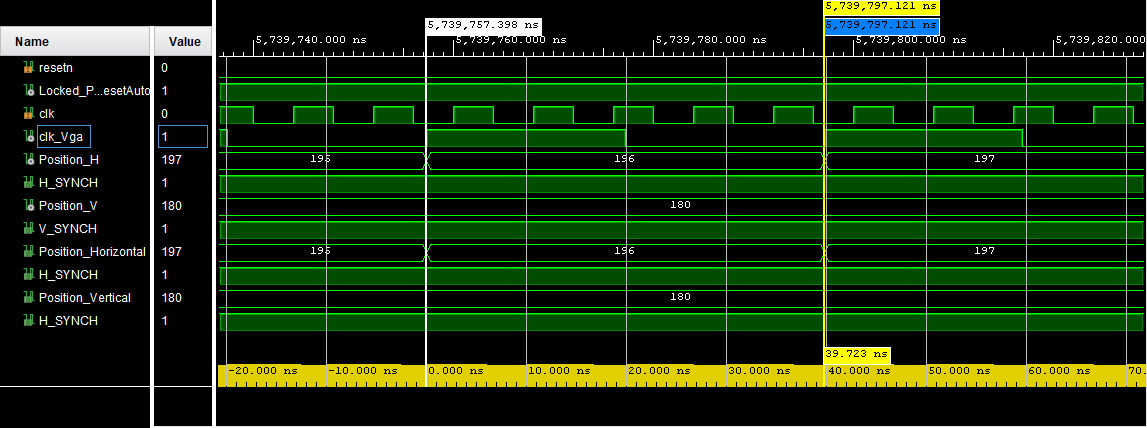
Dans un premier temps, nous allons valider que nous créons une horloge clk cadencé à 125 MHz.



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

### 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.

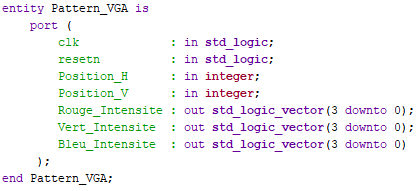
# Ajout et simulation du module Pattern\_VGA au projet

## Code VHDL du module Pattern\_VGA

Pour réaliser le module Pattern\_VGA, on créer déjà la boite noire en y retrouvant :

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

Le code est donc le suivant :



Ensuite on passe à la rédaction de l’architecture de notre module Pattern\_VGA.

Dans cette architecture, nous créons deux signaux internes :

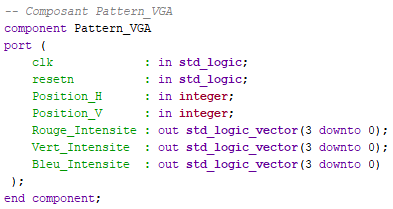
* Intensite\_faible : qui est un std\_logic\_vector de 4 bits
* Intensite\_haute : qui est un std\_logic\_vector de 4 bits

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 »).

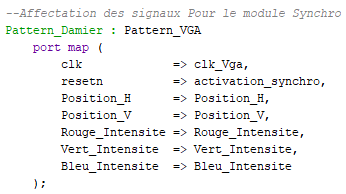
On créer un processus qui sera sensible aux signaux reset et horloge. Lorsque le reset sera activé, on remet les signaux de couleur rouge, vert et bleu à une intensité faible.

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 :



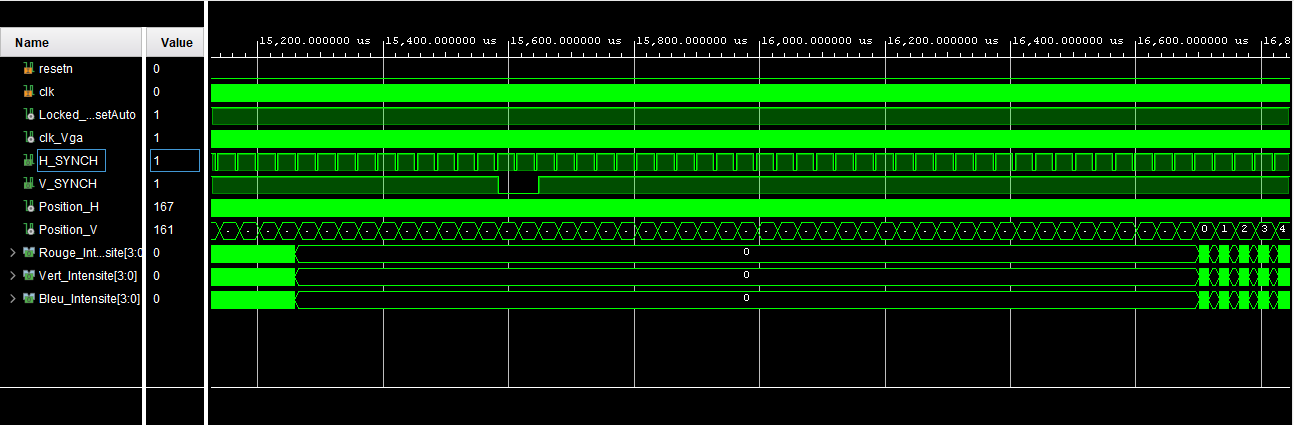
Puis on affecte nos signaux internes du module aux différents signaux de notre projet :



## 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.

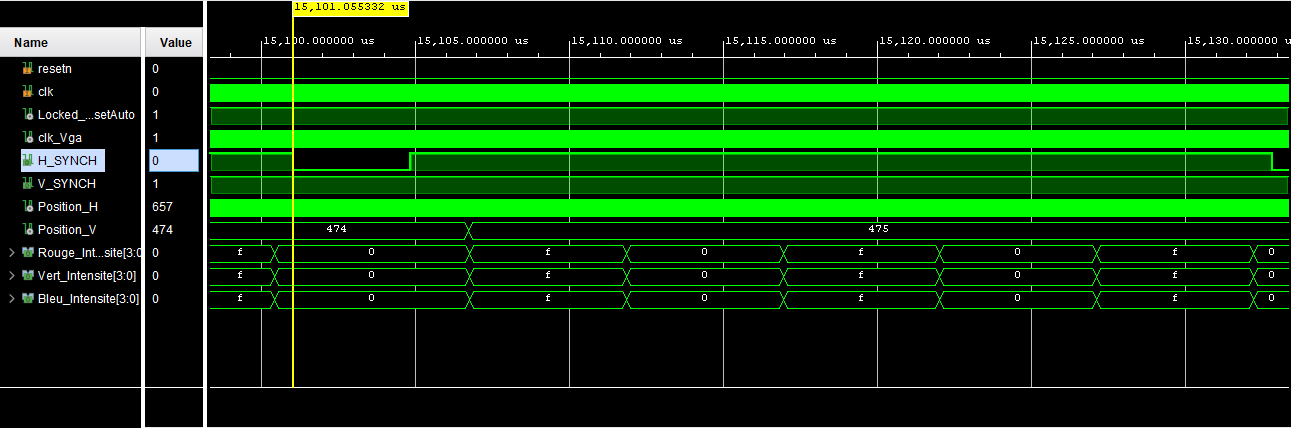
## Résultat de simulation du module Pattern\_VGA au fichier top du projet



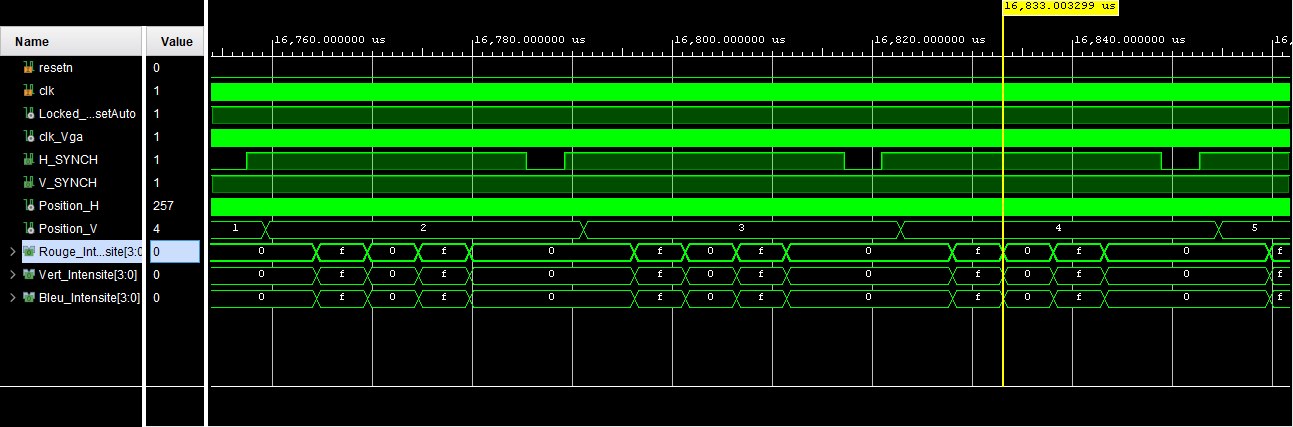
Sur cette simulation, on voit bien que la norme VGA est bien respectée. En effet les couleurs rouges, vert et bleu passe sont forcé à 0 lorsque l’on se trouve dans les plages non visible de notre image, soit :

* 10 pixels verticaux pour le front porch (10 cycles sur H\_SYNCH)
* 2 pixels pour la largeur de syncronisation (moment ou V\_SYNCH est à 0) (2 Cycle de H\_SYNCH)
* 33 pixels verticaux pour le Back porch (33 cycles sur H\_SYNCH)

Représentation d’une ligne avec 3 parties blanches :



Représentation d’une ligne avec 2 parties blanches :



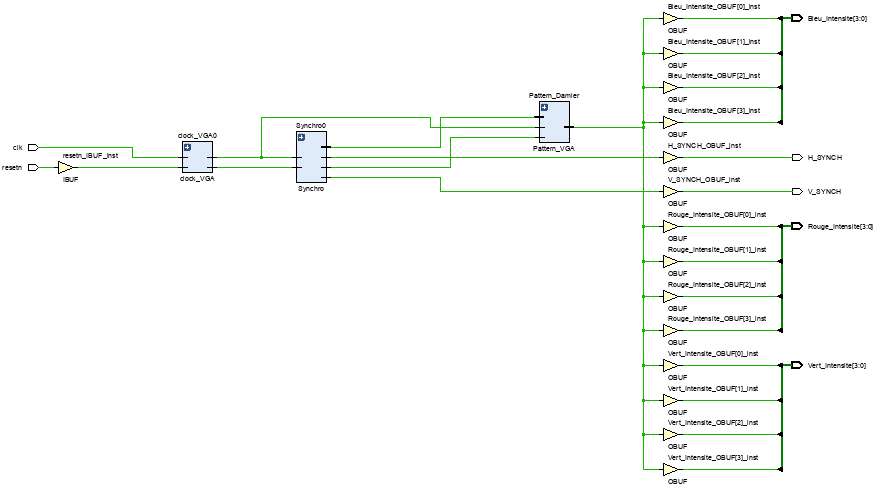
# Test de la solution intermédiaire

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.

## 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.



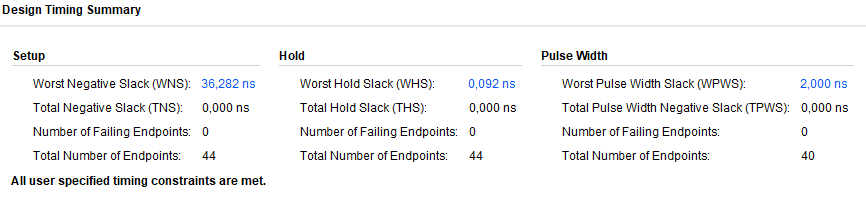
Ici, on voit bien que l’on a :

* Deux signaux d’entrée
  + Clk (1 bit)
  + Resetn (1 bit)
* Cinq signaux de sortie
  + H\_Synch (1 bit)
  + V\_SYNCH (1 bit)
  + Rouge\_intensite (4 bits)
  + Vert\_intensite (4 bits)
  + Bleu\_intensite (4 bits)
* Trois modules
  + Module PLL
  + Module Synchro
  + Module Pattern\_VGA

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

## Validation de l’implémentation

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

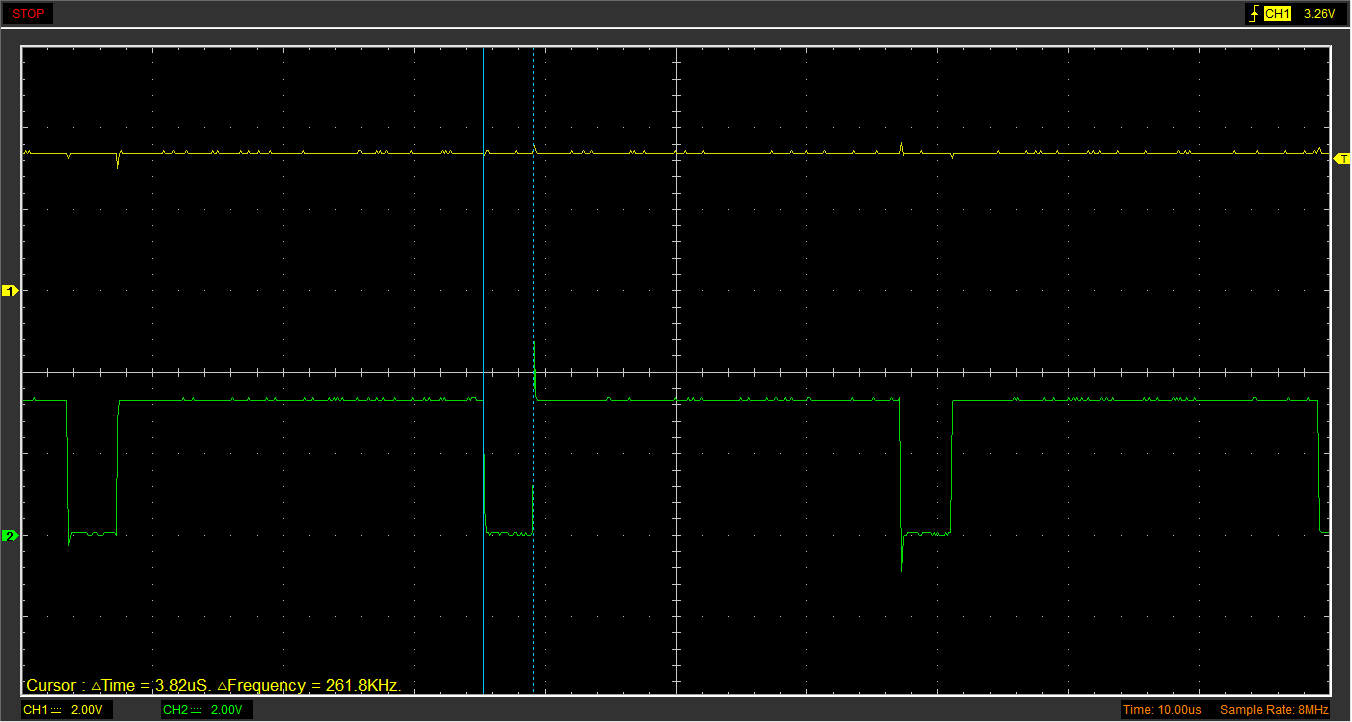


Dans notre cas, tout est correct puisque nous n’avons pas de stack.

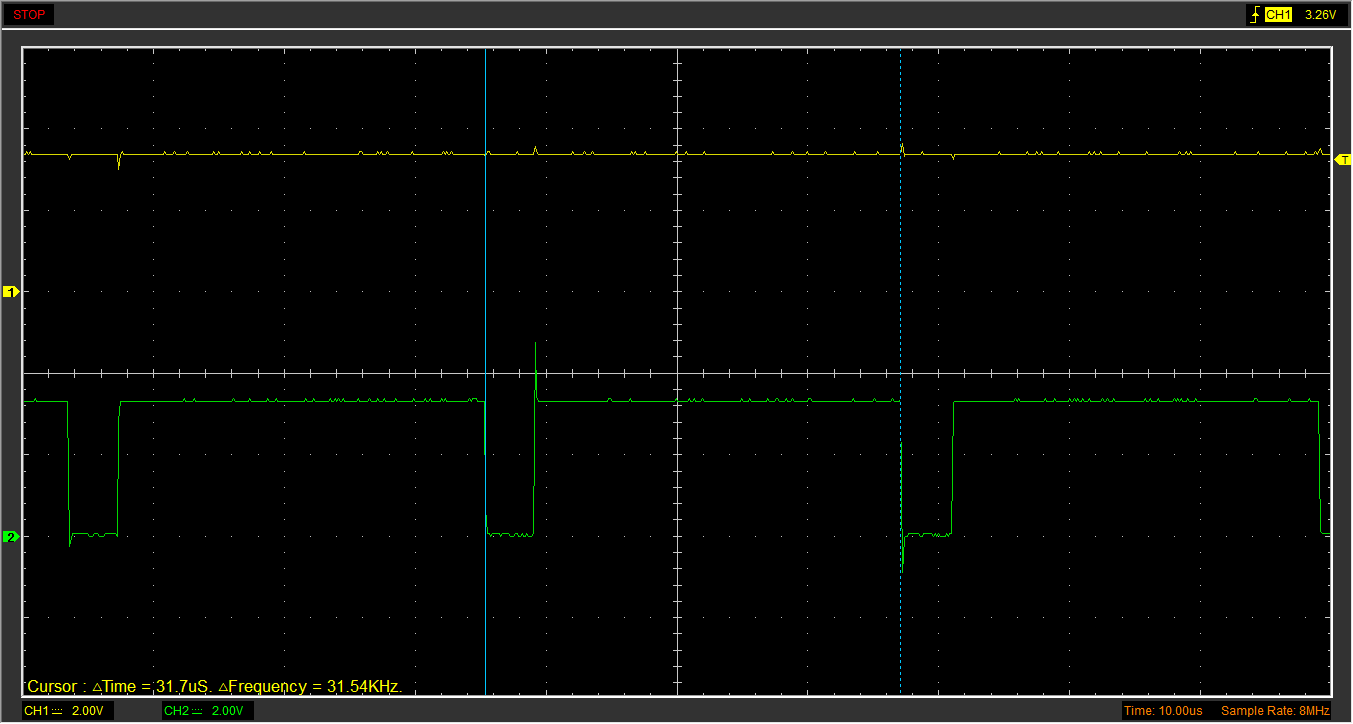
## Génération du bitstream et test à l’oscilloscope

Le bitstream généré (Generateur\_Pattern.bit) est présent dans le dossier document de sortie.

### 4.3.1 Validation du signal H\_SYNCH



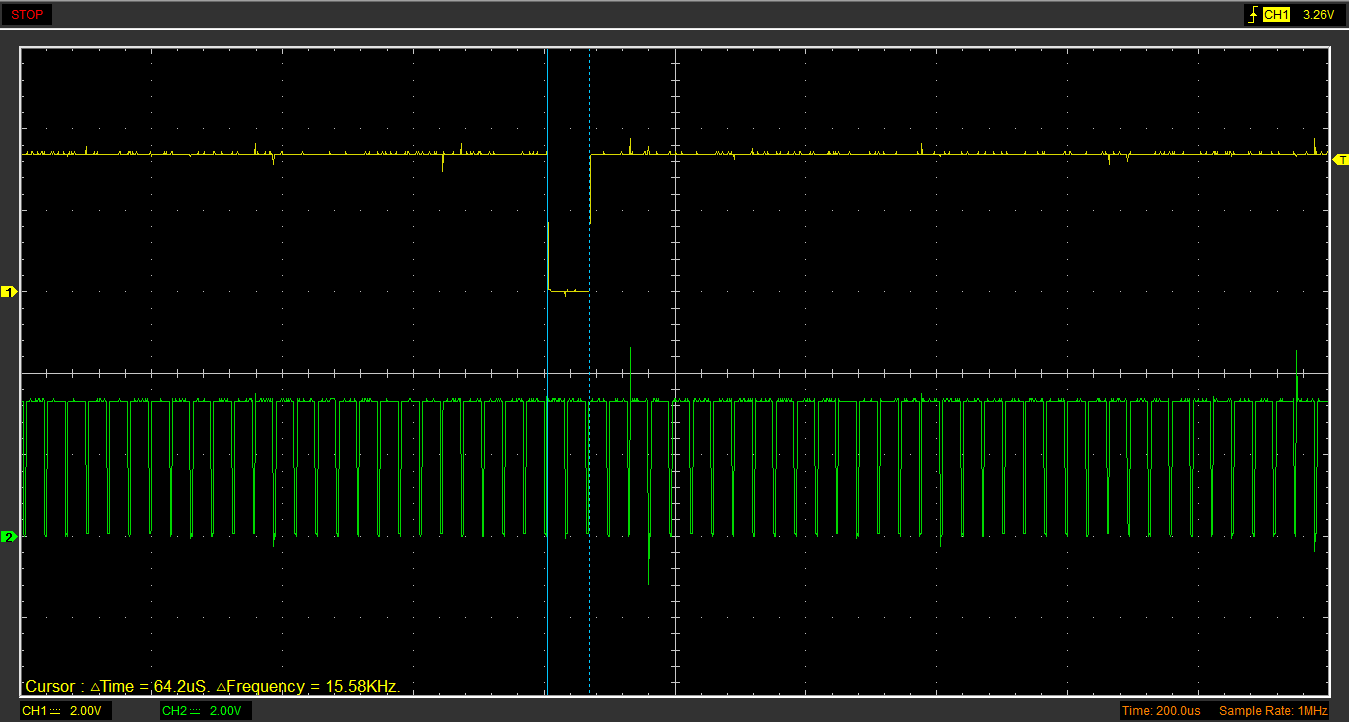
Ici, on voit bien que l’impulsion du signal H\_SYNCH est de 3.82µs ce qui correspond à notre attente.



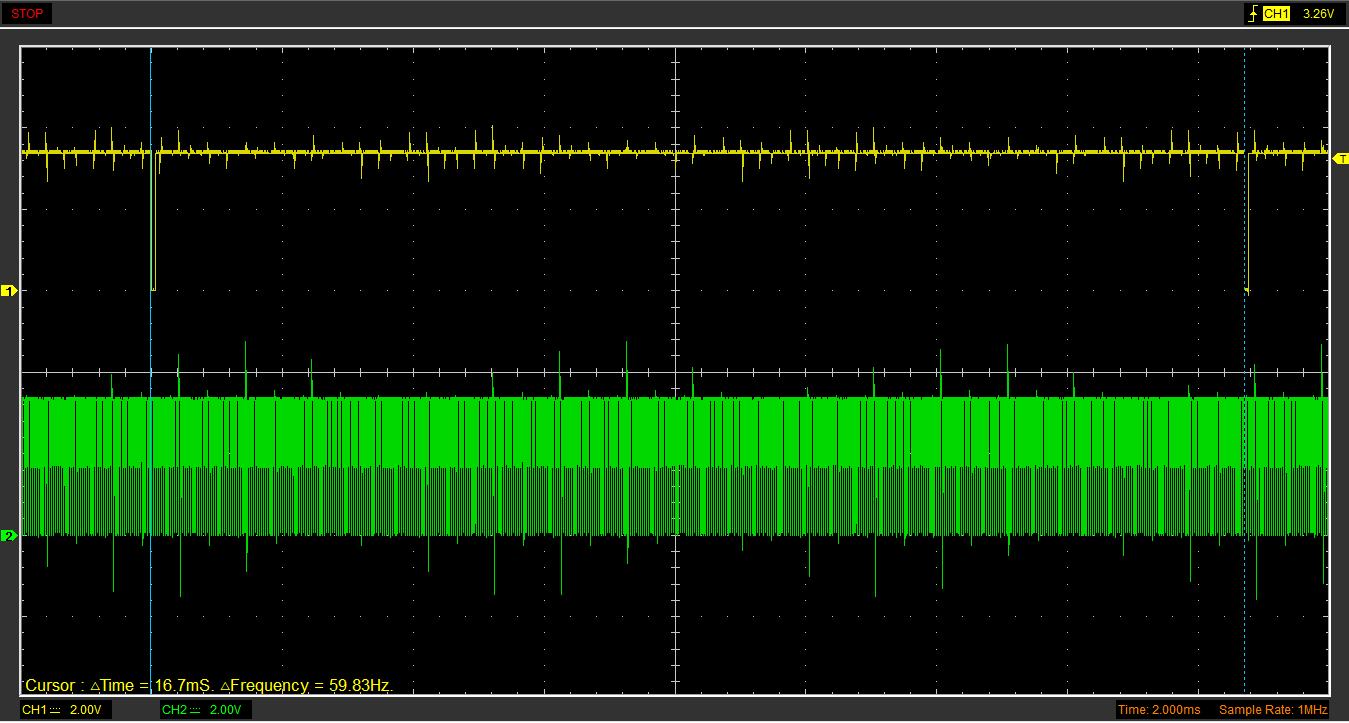
Et que la période du signal H\_SYNCH est de 31.7µs ce qui est conforme également.

Le signal H\_SYNCH est donc correct à notre attente.

### 4.3.2. Validation du signal V\_SYNCH



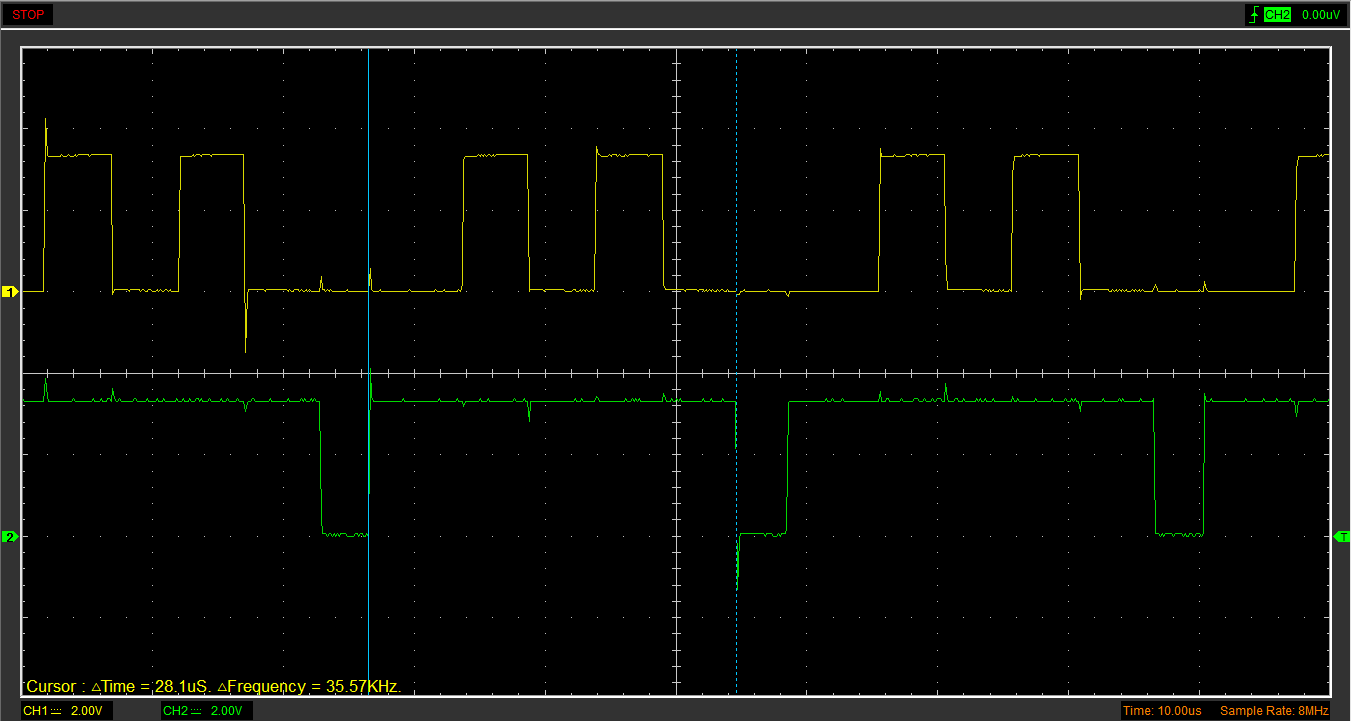
Ici, on voit bien que l’impulsion du signal V\_SYNCH est de 64.2µs ce qui correspond à notre attente.



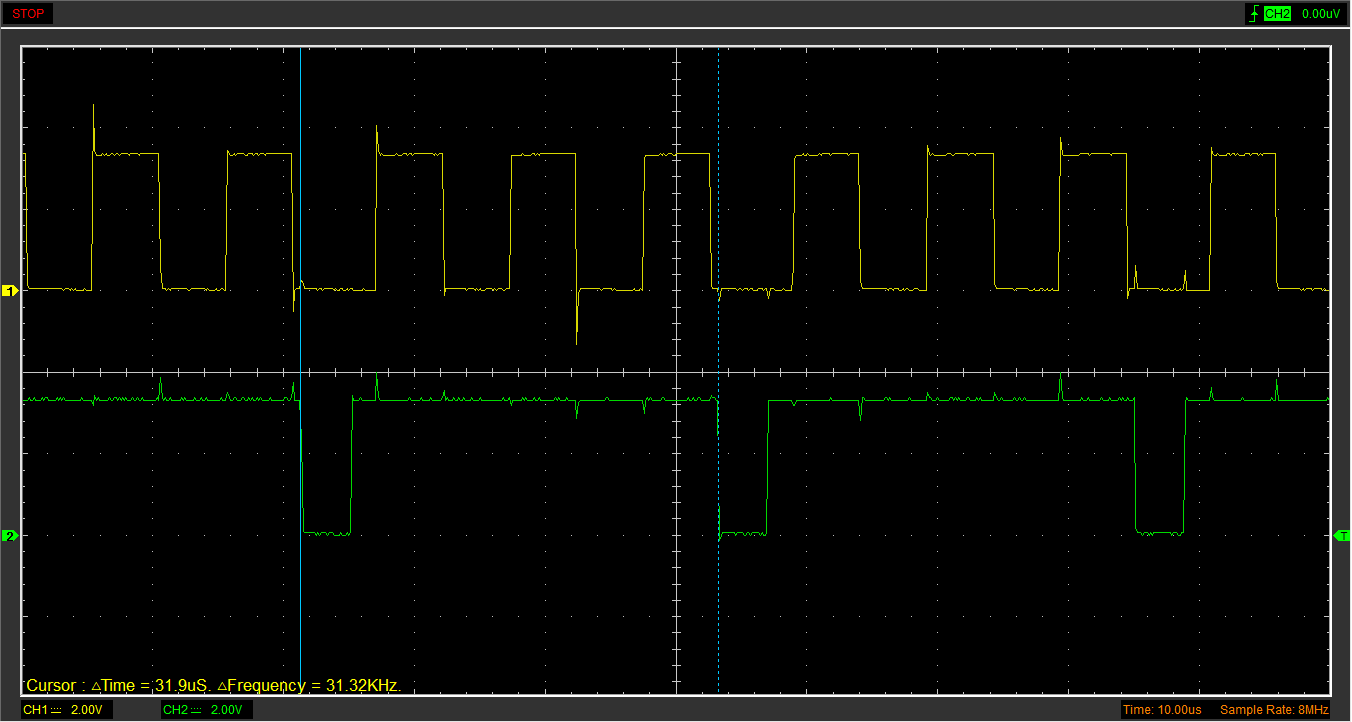
Et que la période du signal V\_SYNCH est de 16.7ms ce qui est conforme également.

Le signal V\_SYNCH est donc correct à notre attente.

### 4.3.3. Représentation de ligne à deux et trois parties blanche

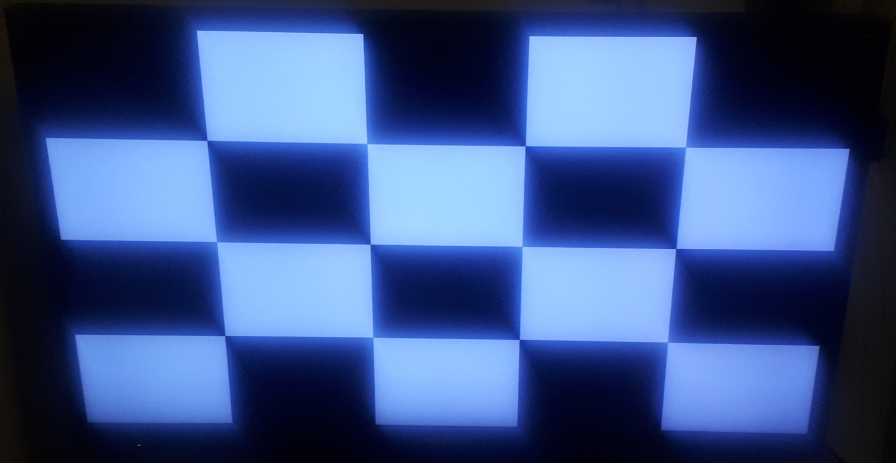


Ici, on voit bien la représentation d’une ligne à deux parties blanches.



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

## 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.

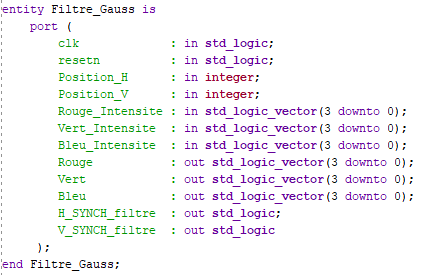
# Ajout et simulation du module Filtre\_Gauss au projet (finale)

## Code VHDL du module Filtre\_Gauss

Pour réaliser le module Filtre\_Gauss, on créer déjà la boite noire en y retrouvant :

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

Le code est donc le suivant :



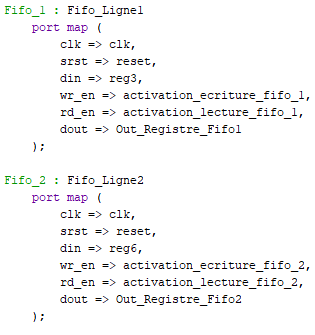
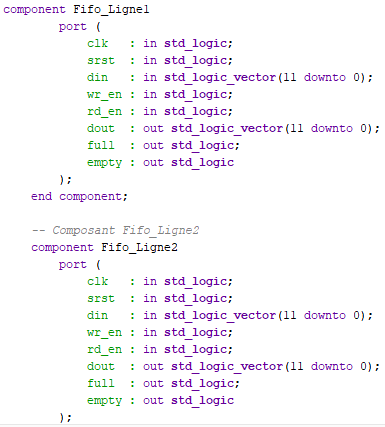
Ensuite on passe à la rédaction de l’architecture de notre module Filtre\_Gauss.

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’entrés (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 :



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.

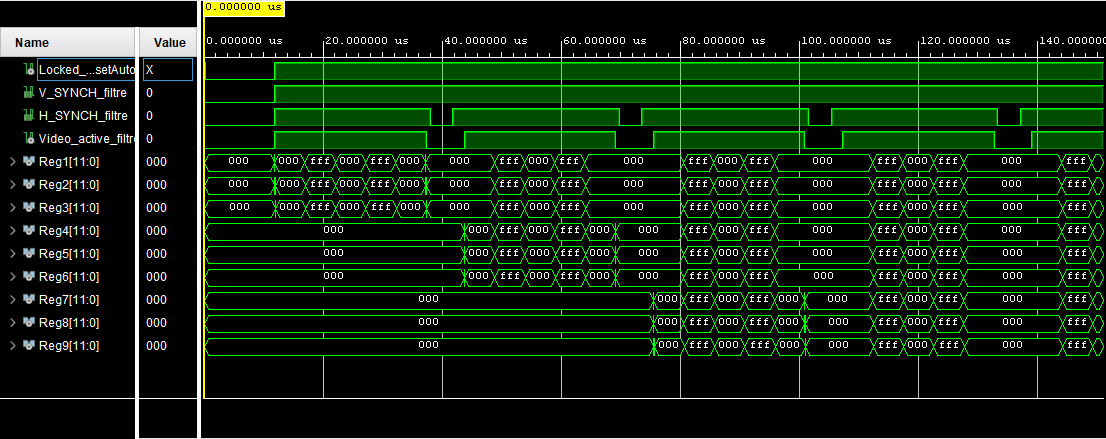
Le dernier processus permet de réaliser les calculs de convolution et d’affecter les résultats de ces calculs aux signaux de couleurs de sortie.

## 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.

## Résultat de simulation du module Filtre\_Gauss au fichier top du projet

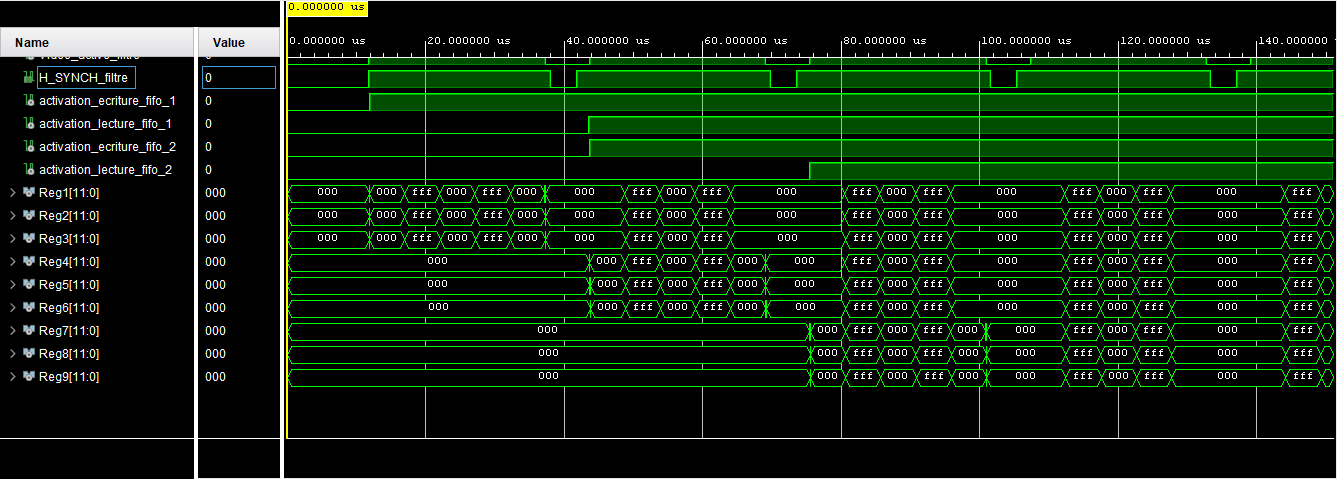
### Validation du remplissage des registres



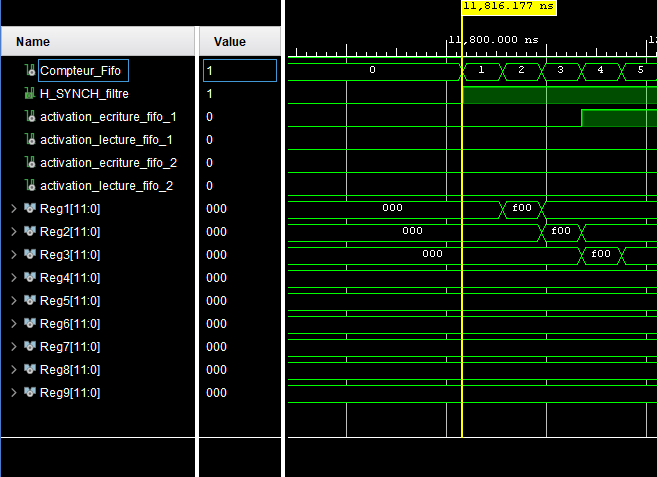
Ici, on voit bien que lorsque l’on est sur la première ligne de l’image on remplit les registres 1 à 3.

Lorsque l’on se trouve sur la second ligne de l’image, les registres 1 à 3 sont remplit 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.

### Validation de l’écriture et de lecture des registres (FIFOs)

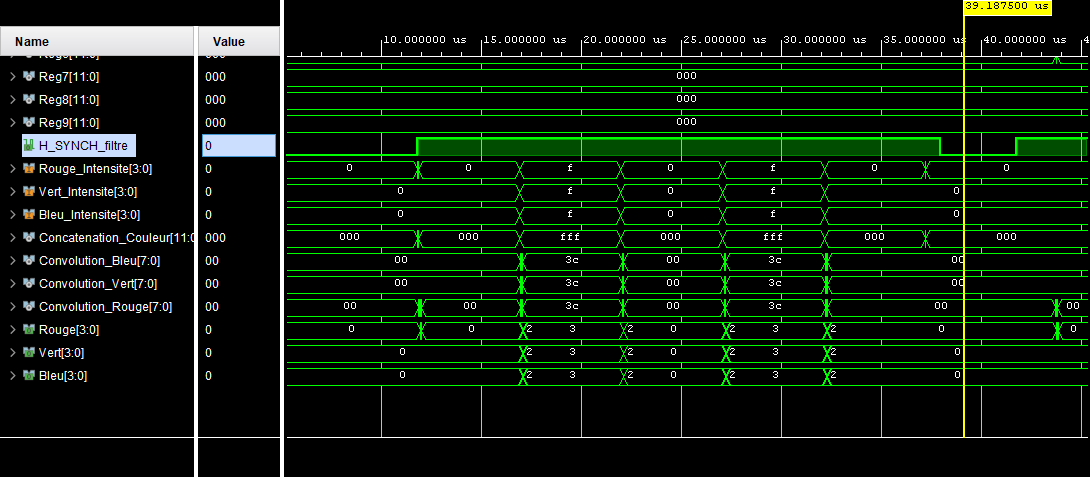


Ici, on voit bien que l’écriture de la Fifo 1 est activée lors de la première ligne de l’image avec un décalage de trois pixels afin de laisser le temps au registre 1 à 3 de se remplir, comme le montre l’image suivante :



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.

### Validation de la convolution Gaussienne



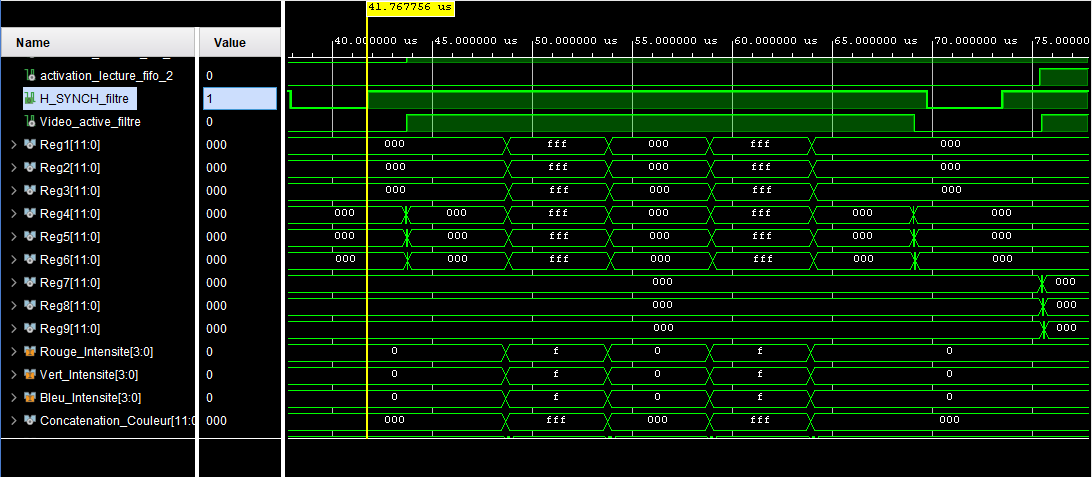
Ici, on voit bien que lors de la première ligne de l’image :

* 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.

### 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\_SYNCH\_filtre est passé, et lorsque l’on remplit les différents registres.

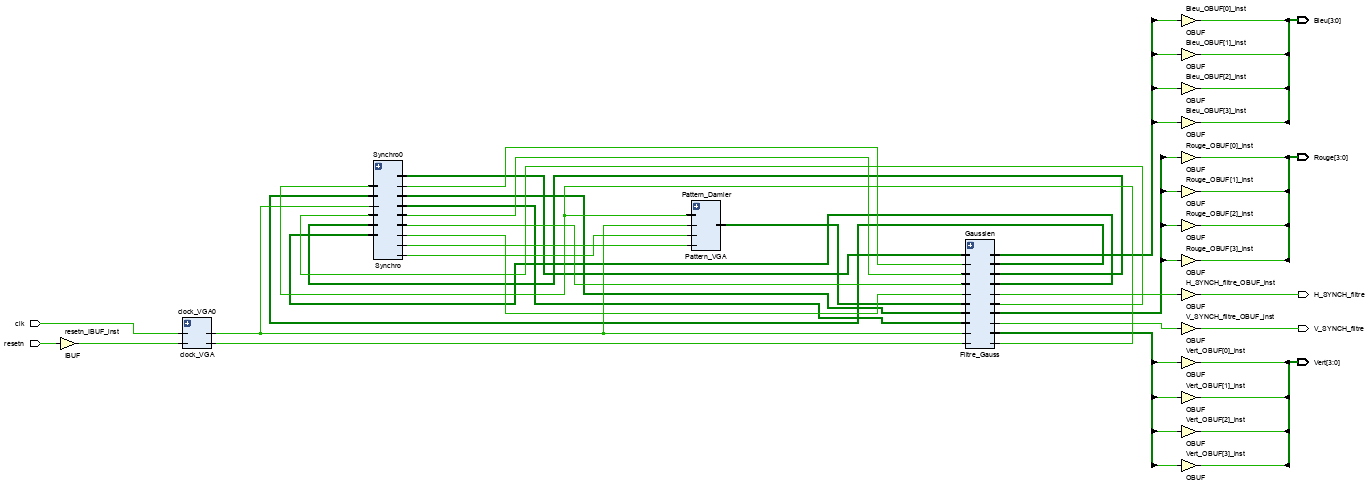
# Test de la solution finale

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.

## 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.



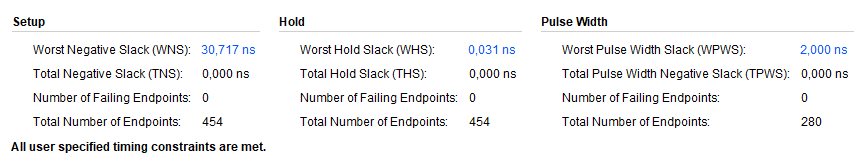
Ici, on voit bien que l’on a :

* Deux signaux d’entrée
  + Clk (1 bit), Resetn (1 bit)
* Cinq signaux de sortie
  + H\_Synch\_filtre (1 bit), V\_SYNCH\_filtre (1 bit)
  + Rouge (4 bits), Vert (4 bits) et Bleu (4 bits)
* quatre modules
  + Module PLL, Module Synchro, Module Pattern\_VGA, Module Filtre\_Gauss

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

## Validation de l’implémentation

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

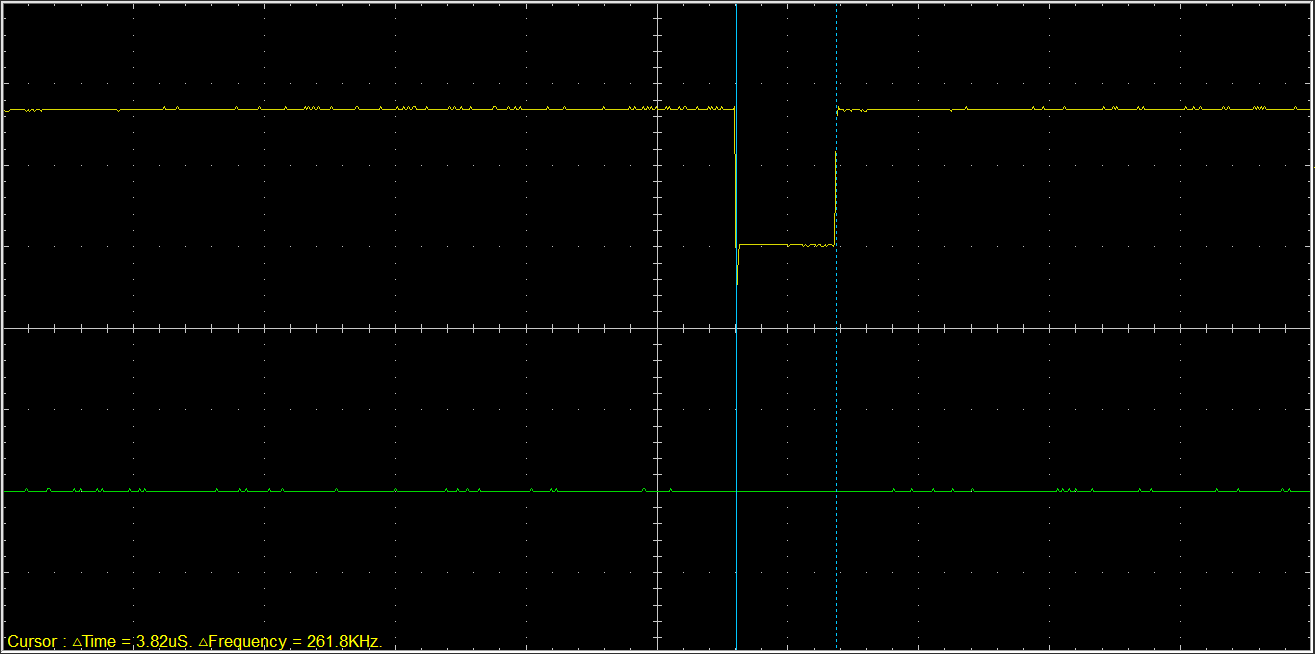


Dans notre cas, tout est correct puisque nous n’avons pas de stack.

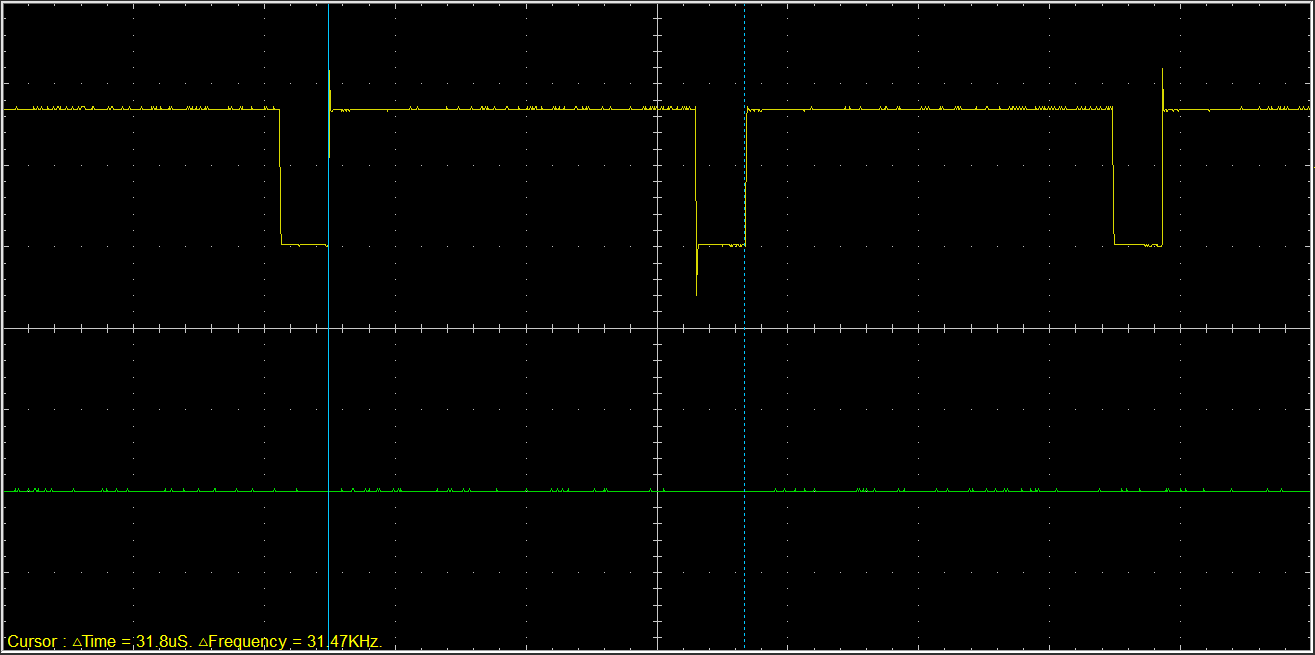
## Génération du Bitstream et test à l’oscilloscope

Le bitstream généré (Porjet\_VGA\_Gaussien.bit) est présent dans le dossier Bitstream.

### Validation du signal H\_SYNCH\_filtre



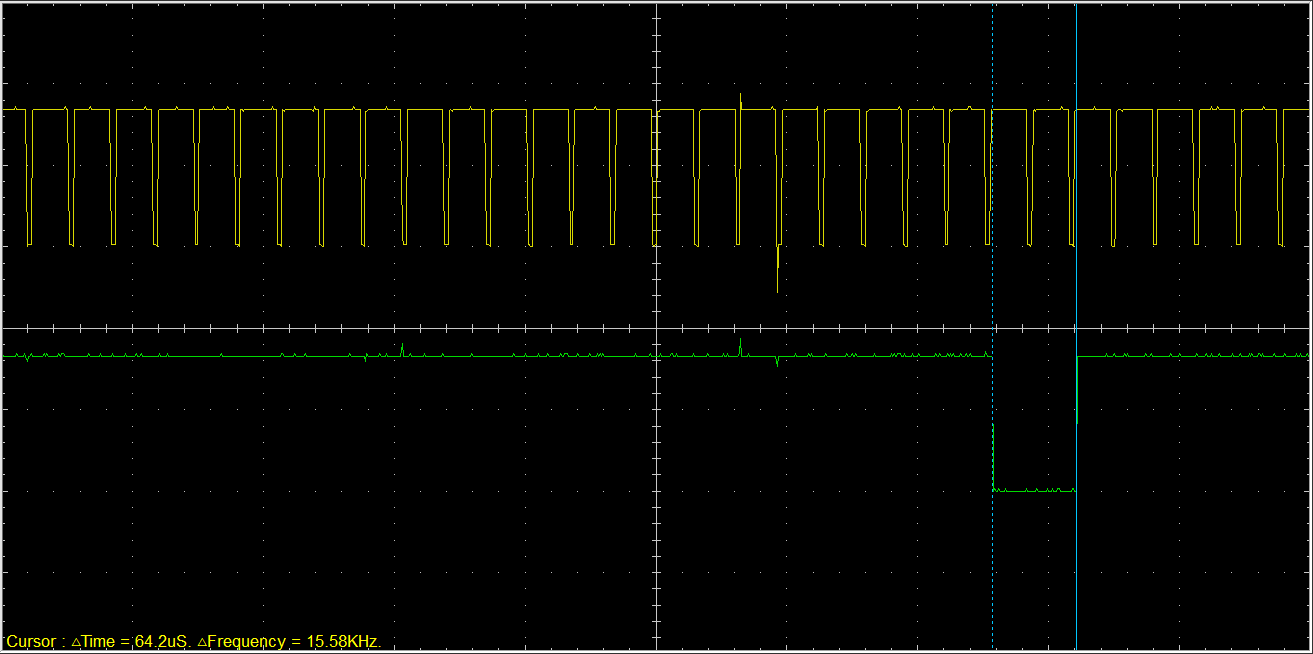
Ici, on voit bien que l’impulsion du signal H\_SYNCH est de 3.82µs ce qui correspond à notre attente.



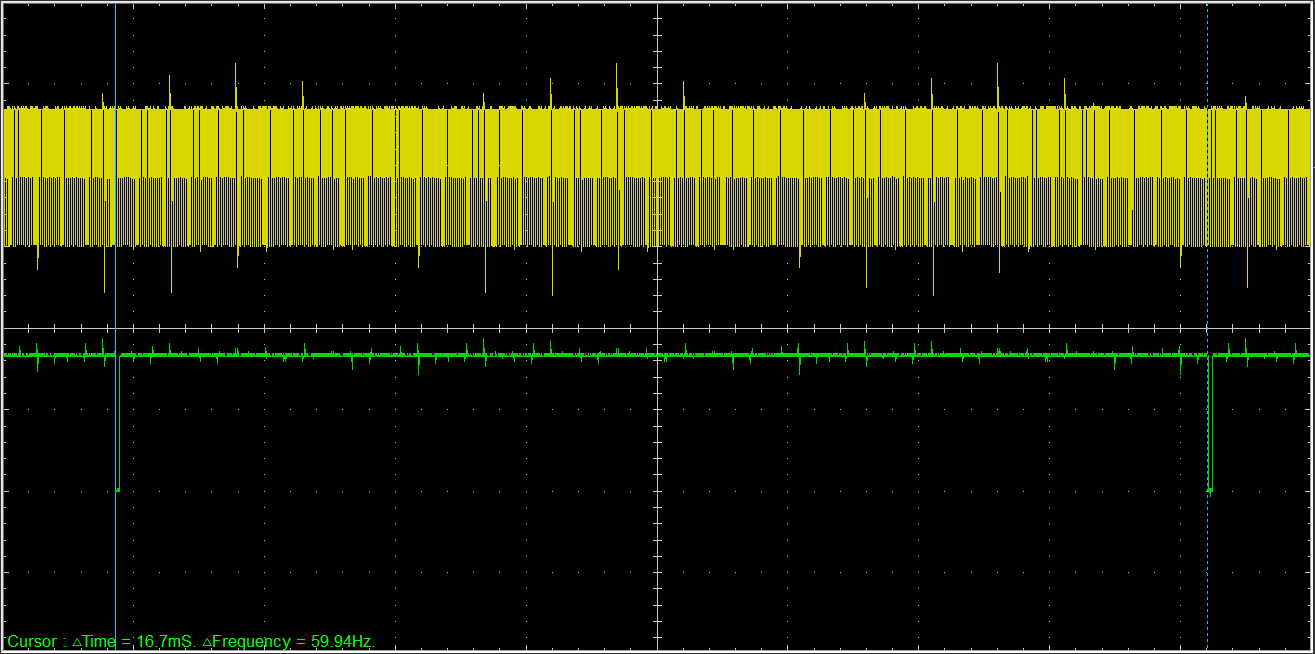
Et que la période du signal H\_SYNCH est de 31.7µs ce qui est conforme également.

Le signal H\_SYNCH est donc correct à notre attente.

### Validation du signal V\_SYNCH\_filtre



Ici, on voit bien que l’impulsion du signal V\_SYNCH est de 64.2µs ce qui correspond à notre attente.



Et que la période du signal V\_SYNCH est de 16.7ms ce qui est conforme également.

Le signal V\_SYNCH est donc correct à notre attente.

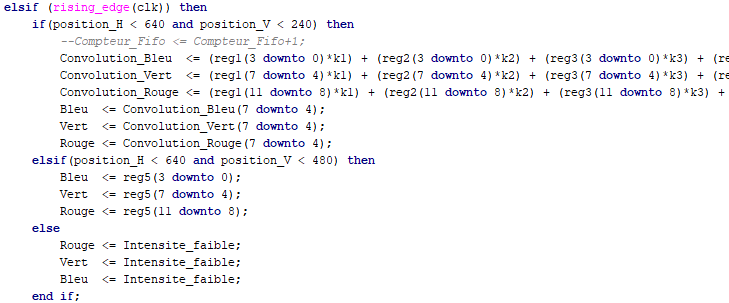
## Démonstration de la partie finale

Etant compliqué de prendre des photos ou 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 :



La projection de notre pattern sur l’afficheur nous permet de valider notre solution finale 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.
* 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 flouté et la partie basse est nette.

