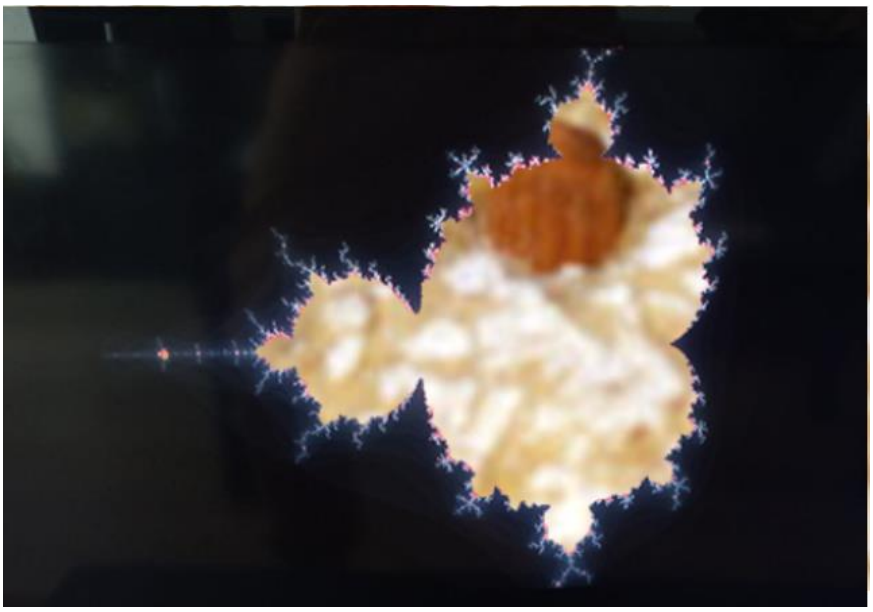
Hes·so Master

MSE | MASTER OF SCIENCE IN ENGINEERING

Projet Fractales (LPSC)



Auteur : Spinelli Isaia

Prof : Fabien Vannel et Joachim Schmidt

Date : 08.06.2021

Classe : MA_LPSC

1. Table des matières

2. Introduction

Ce rapport permet de décrire la réalisation du projet fractale du cours LPSC. Ce projet consiste à réaliser un système qui calcul la fractale de Mandelbrot puis l'affiche sur un écran via une interface HDMI. Une grande partie du projet est fourni qui comprend principalement la gestion du HDMI et la génération des différents domaines d'horloges.

Objectifs

Notre objectif est de remplacer un composant qui affiche constamment le drapeau du canton de Neuchâtel par une fractale de Mandelbrot. De plus, il est demandé de modifier les connexions afin de passer par une mémoire BRAM.

3. Architecture globale

Afin d'avoir une bonne visualisation sur l'architecture global de notre système, nous avons fait un schéma :

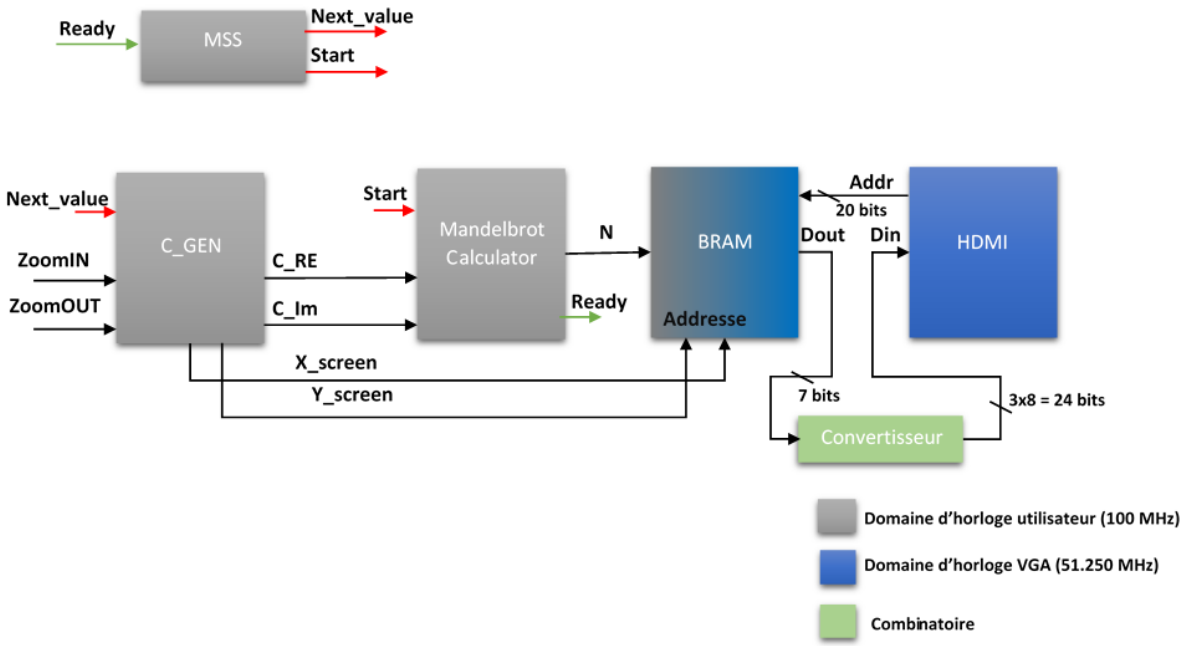


Figure 3-1 : Architecture globale

On peut voir les principaux modules composant notre système :

- **C_GEN** : Permet de générer les constantes réelles et imaginaires ainsi que l'index du pixel correspondant. Le signal d'entrée « Next_value » permet de contrôler l'incrément.
- **Mandelbrot Calculator** : Permet de calculer le résultat de l'équation de Mandelbrot pour un point donné. ($Z_{n+1} = Z_n^2 + C$). Le signal d'entrée « Start » permet de commencer un nouveau calcul. Le signal de sortie « Ready » permet d'indiquer que le module est prêt pour un nouveau calcul. Le signal de sortie « N » permet d'indiquer le nombre d'itération correspondant au dernier calcul.
- **MSS** : Cette machine d'état permet de faire le lien entre les modules « C_GEN » et « Mandelbrot Calculator ». Plus précisément, elle permet d'indiquer au module « C_GEN » quand générer les prochaines constantes. Ceci est effectué lorsque le module « Mandelbrot

Calculator » est prêt pour un nouveau calcul. Une fois que les constantes suivantes sont générées, cette machine d'état lance le prochain calcul de Mandelbrot.

- **BRAM** : Ce module permet de stocker tous les résultats des itérations pour chaque pixel de la résolution choisie. De l'autre côté, il permet au module « HDMI » de lire ces résultats. En intégrant deux ports, il permet d'écriture et de lire dans la même zone mémoire à des fréquences différentes.
- **Convertisseur** : Permet de convertir le nombre d'itérations (max = 100 => 7 bits) en valeur RGB sur 24 bits afin d'être compatible avec le module « HDMI ».
- **HDMI** : Ce module qui nous a été fourni permet d'afficher sur un écran les pixels lus et convertis de la mémoire BRAM.

On peut confirmer la bonne implémentation de cette architecture avec le rapport schématique de Vivado présenté dans la

Figure 3-2 :

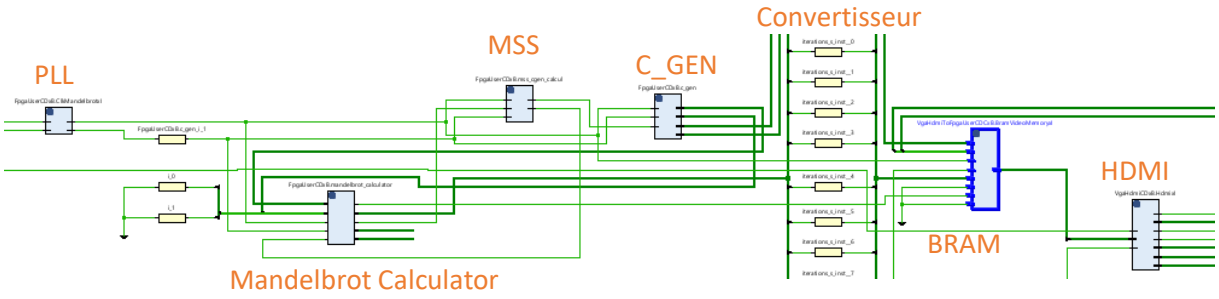


Figure 3-2 : Schéma de l'architecture de Vivado

Etant donné que la qualité ne permet pas de bien lire, nous avons ajouté des titres orange proches des différents modules sur le schéma. Grâce à cet outil, nous avons pu confirmer que la structure et les liens étaient correctes.

Type d'implémentation choisie

Dans le cadre de ce projet, nous avons décidé d'implémenter les différentes parties en VHDL sans utiliser IP core.

Réalisation du bloc CALCUL (MandelbrotCalculator)

Il existe différentes variantes à cette courbe de Mandelbrot. Dans ce projet, nous avons choisi d'implémenter celle-ci :

$$Z_{n+1} = Z_n^2 + C$$

Afin de simplifier l'implémentation et de bien saisir cette formule, nous avons commencé par la décomposer avec la partie réelle d'un côté et imaginaire de l'autre :

$$(Z_{n+1})\,Re = Z_{Re^2} - Z_{Im^2} + C_{RE}$$

$$(Z_{n+1})Im = 2 * Z_{Re} * Z_{Im} + C_{IM}$$

On peut voir dans la Figure 3-3 la décomposition du calcul effectué avec des composants logiques :

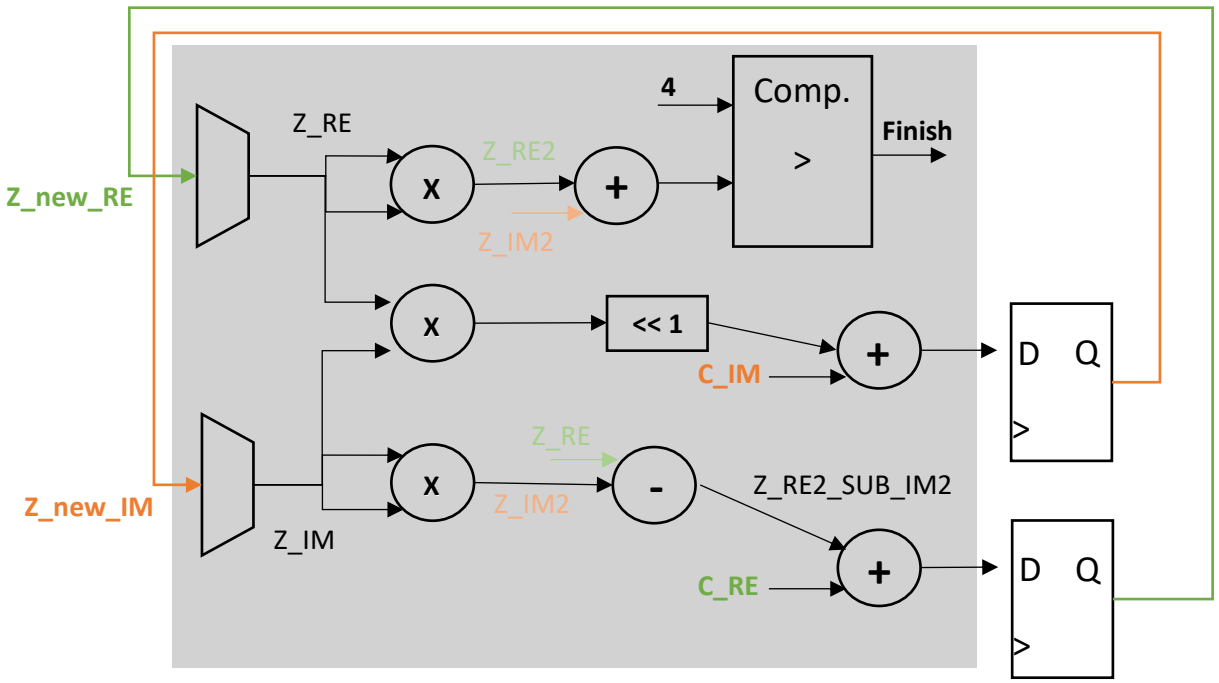


Figure 3-3 : Décomposition du calcul de Mandelbrot

Cette décomposition nous a permis de réaliser un bloc combinatoire en VHDL correspondant au bloc gris dans la Figure 3-3. En plus de ce bloc, nous avons besoin d'autres éléments, tels que : des bascules D, une machine d'état et un compteur.

Les bascules D permettent de mémoriser les états de plusieurs signaux : le résultat de la partie réelle du calcul, le résultat de la partie imaginaire et l'état de la machine d'état. Le compteur permet de compter le nombre d'itérations effectuées par calcul. Finalement, la machine d'état permet de gérer les états d'un calcul : attente et calcul. L'état d'attente permet d'initialiser correctement différents signaux, comme le compteur. L'état de calcul permet d'effectuer le calcul ainsi que d'incrémenter le compteur.

Afin de s'assurer du bon fonctionnement du calculateur, nous avons écrit un petit code python. Celui-ci permet de donner en entrée une constante réelles et imaginaire, puis affiche les résultats de chaque itérations. Ce code est aussi présent sur le *GitHub*.

Modification du bloc C_GEN

Le bloc « C_GEN » nous a été fourni. Il permet de générer les constantes réelles et imaginaires ainsi que l'index du pixel correspondant. Nous avons simplement dû ajouter la connexion du signal d'entrée « nextValue ».

Réalisation du bloc MSS

Cette machine d'état permet de faire le lien entre les modules « C_GEN » et « Mandelbrot Calculator ». Plus précisément, elle permet d'indiquer au module « C_GEN » quand générer les prochaines constantes. Ceci est effectué lorsque le module « Mandelbrot Calculator » est prêt pour un nouveau calcul. Une fois que les constantes suivantes sont générées, cette machine d'état lance le prochaine calcul de Mandelbrot. Voici une image d'une simulation afin de visualiser les différentes étapes :

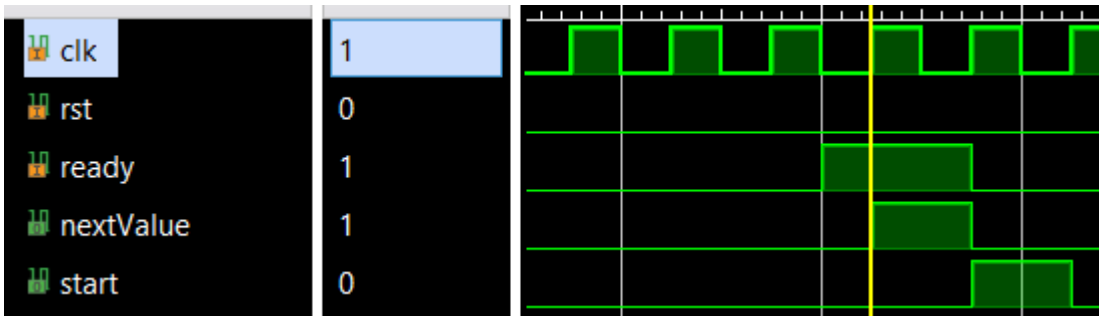


Figure 3-4 : Simulation MSS

Dès que le calculateur est prêt pour un nouveau calcul (ready à '1'), le signal « nextValue » est mis à '1', puis, on comment un nouveau calcul en mettant le signal « start » à '1'.

Réalisation du bloc Convertisseur

Ce petit bloc permet de convertir la valeur du nombre d'itérations stockée dans la BRAM en une valeur RGB sur 24 bits afin d'être compatible avec le module « HDMI ». Nous pouvons voir ci-dessous, le code qui permet cette conversion :

```
DataBramMV2HdmixAS : DataBramMV2HdmixD <= BramVideoMemoryReadDataxD(C_BRAM_VIDEO_MEMORY_DATA_SIZE-3 downto 0) & '0' &
                                          BramVideoMemoryReadDataxD(C_BRAM_VIDEO_MEMORY_DATA_SIZE-3 downto 0) & '0' &
                                          BramVideoMemoryReadDataxD(C_BRAM_VIDEO_MEMORY_DATA_SIZE-3 downto 0) & '0';
```

Nous récupérons les 7 bits LSB de la valeur du nombre d'itérations, puis la multiplions par deux pour accentuer les différences ainsi que pour avoir une représentation sur 8 bits.

4. Résultats et analyses de performances

Nous avons commencé par observer le résumé des timings à l'aide du logiciel *Vivado* :

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-5,379 ns	Worst Hold Slack (WHS):	0,073 ns	Worst Pulse Width Slack (WPWS):	0,922 ns

Figure 4-1 : Résumé des timings

On peut constater que le pire temps calculé pour divers chemins traversant les domaines d'horloges spécifiés est de -5,379 ns. Ceci signifie que la contrainte de la fréquence de 10 MHz n'est pas garantie. Afin d'en savoir un peu plus, nous avons regardé plus précisément d'où provenait ce problème. On peut le voir dans la Figure 4-2 :

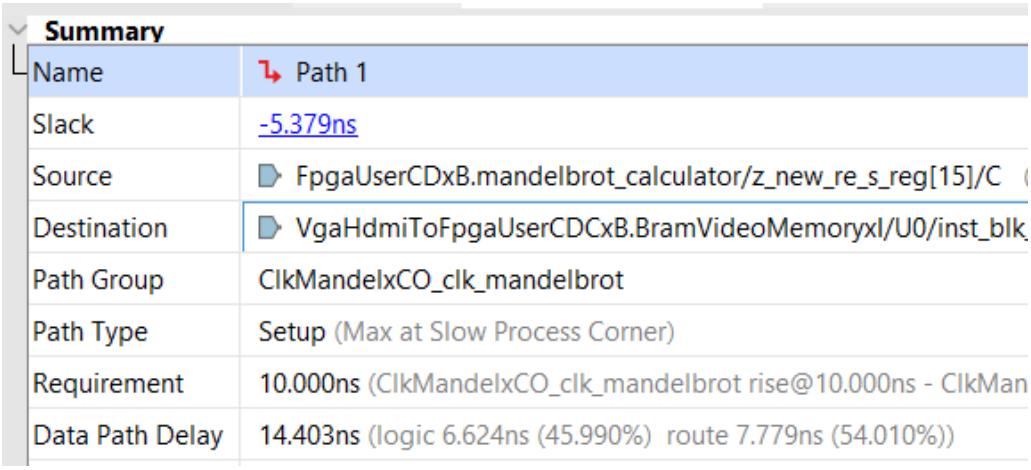


Figure 4-2 : Précision du chemin le plus long

On peut constater que la majorité du temps est passé sur le chemin et non dans la logique. Dans un premier temps, cette observation nous indique qu'il serait préférable de descendre la fréquence maximale à 65 MHz afin de respecter le chemin le plus long, introduit par la logique combinatoire implémentée. Cependant, il serait envisageable d'améliorer le placement afin de gagner du temps de routage. Toutefois, nous pensons que la solution la plus professionnelle serait de pipeliner notre logique combinatoire. Cette méthode ajouterait une latence pour les premiers résultats mais permettrait le fonctionnement du système avec une fréquence plus élevée. Cette solution est plus détaillée dans le chapitre « Amélioration ».

Ensuite, nous pouvons observer les informations sur l'utilisation des ressources du FPGA dans la Figure 4-3 :

Summary

Resource	Utilization	Available	Utilization %
LUT	1015	134600	0.75
FF	234	269200	0.09
BRAM	150	365	41.10
DSP	5	740	0.68
IO	19	285	6.67
MMCM	2	10	20.00

Figure 4-3 : Résumé des utilisations

On peut voir que nous utilisons 150 composants de BRAMs, ce qui correspond à plus de 40 % des BRAMs à dispositions. Si on souhaite comprendre pourquoi nous avons besoin de 150 blocs de

BRAMs, nous devons ouvrir la fenêtre de paramétrage du ip core BRAM. Nous pouvons voir ci-dessous la configuration actuelle de l'IP :

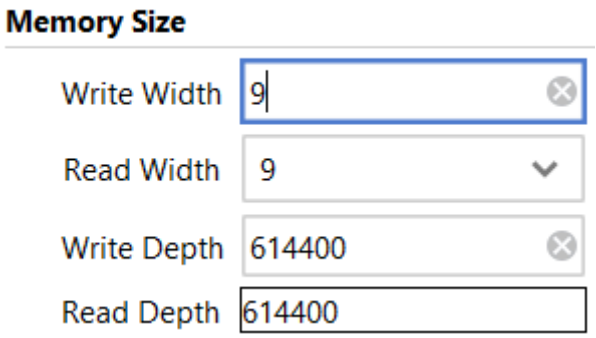


Figure 4-4 : Configuration IP BRAM

Voici comment nous pouvons calculer le nombre de bloc BRAM de 36K dont nous avons besoin :

$$\frac{depth * width}{BLOC\ size} = \frac{614400 * 9}{(36 * 1024)} = 150$$

Mise à part ceci, nous n'utilisons pas énormément de ressources dans les autres différents blocs. On peut constater que le système a réussi à comprendre que nous souhaitions utiliser des DSPs, 5 sont utilisés. On peut confirmer que ces DSPs sont bien utilisés pour notre calculateur :

Name	Used
N mandelbrot_pinout	5
FpgaUserCDxB.mandelbrot_calculator (mandelbrot_calculator)	5

Figure 4-5 : DSPs du calculateur

5. Résultat final

Nous pouvons voir ci-dessous deux résultats de notre implémentation :

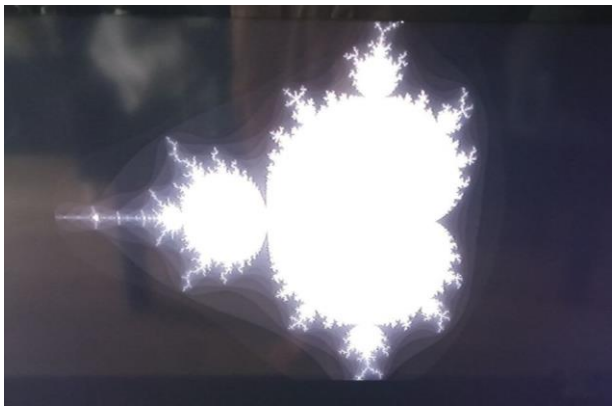


Figure 5-1 : Résultat final réel

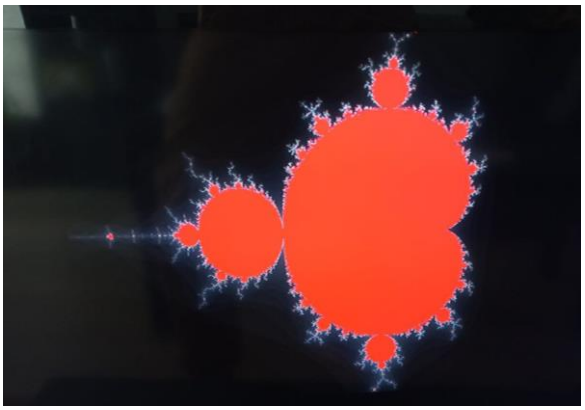


Figure 5-2 : Résultat modifié

L'image de gauche représente le résultat final réel. On peut apercevoir les différentes limitations de la forme Mandelbrot avec les dégradés de blanc. Sur l'image de droite nous avons forcé en rouge les pixels qui ont pu être calculés avec le nombre d'itérations maximum.

6. Amélioration

Il serait possible d'améliorer grandement la fréquence maximale du système en découpant notre logique en plusieurs étages. En augmentant la profondeur de notre pipeline, nous augmenterons la latence du premier résultat mais diminuerons le plus long chemin. Ce qui se traduit directement par une fréquence maximale plus élevée. On peut voir dans la Figure 6-1 un exemple de notre logique divisée en quatre étages :

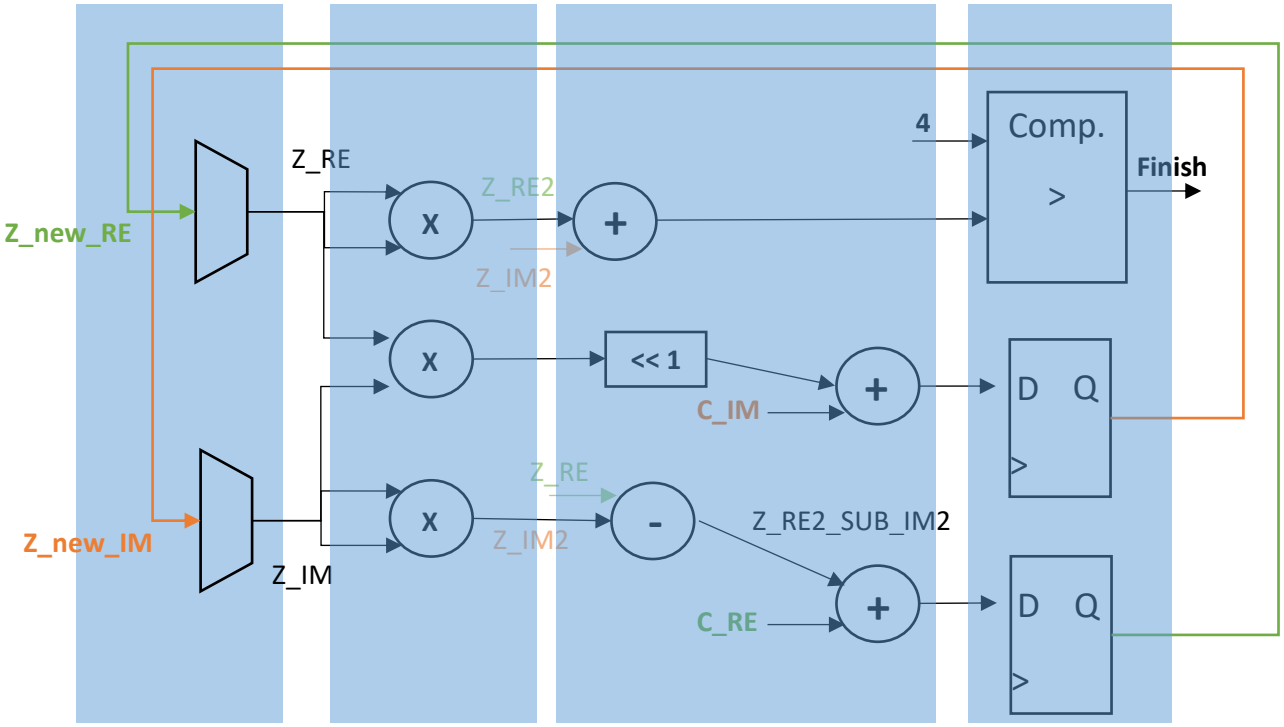


Figure 6-1 : Logique pipelinée

7. Conclusion

Problèmes survenus

Un processus de synthèse et d'implémentation prend approximativement 15 minutes. Ce temps n'est pas négligeable. De plus, j'ai eu un problème lorsque je souhaitais mettre en place l'analyseur logique. Il y avait un problème avec la longueur du chemin d'accès. Pour résoudre ce problème j'ai dû déplacer mon projet de répertoire. Voici ci-dessous le message d'erreur correspondant :

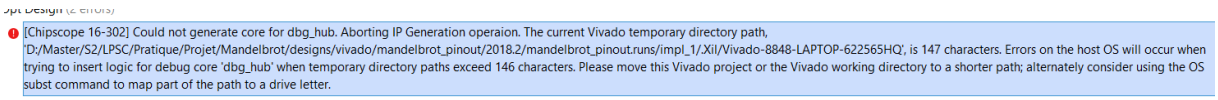


Figure 7-1 : Message d'erreur sur la longueur d'un chemin d'accès

Améliorations

Durant ce projet, nous avons concentré le traitement dans un processus combinatoire. Cependant, il aurait préférable d'implémenter une solution pipelinée. Cela permettrait d'utiliser une fréquence plus élevée et permettrait d'augmenter la vitesse de calcul.

Compétences acquises

- Analyse et compréhension d'un projet relativement grand
- Familiarisation des outils de développement de Xilinx
- Mise en place d'un analyseur logique
- Analyse de la synthèse de notre code afin de vérifier comment l'outil le traduit.

Date : 20.06.21

Nom de l'étudiant : Spinelli Isaia