MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so

# Laboratoire Mandelbrot

| | |
|---|---|
| Département : | EIE |
| Unité d'enseignement : | Logique programmable pour systèmes complexes et performants (LPSC) |


| | |
|---|---|
| Auteur | Sébastien Deriaz |
| Professeur | Fabien Vannel |
| Assistant | Joachim Schmidt |
| Date | 24 mai 2022 |

# Table des matières

# 1 Introduction

Le but de se laboratoire est d'implémenter un affichage de la fractale de Mandelbrot sur FPGA.

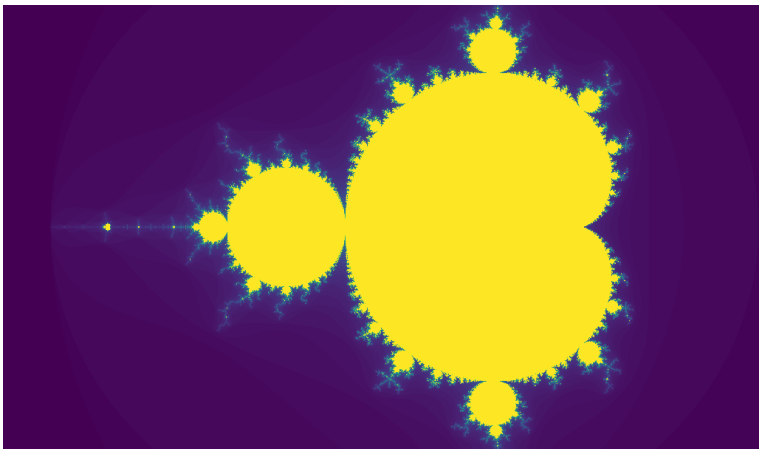


FIGURE 1 – Fractale de Mandelbrot

Pour obtenir la fractale, il faut évaluer l'équation 1 avec comme point de départ $C$ (dans le plan complexe).

$$z_{k+1} = z_k^2 + C \tag{1}$$

Après chaque itération, on évalue si la norme euclidienne du nombre complexe $z_{k+1}$ dépasse un rayon donné $R = 2$. Le nombre d'itérations réalisées jusqu'ici constitue la valeur associée à chaque pixel.

## 1.1 Couleurs

Pour passer d'un nombre d'itérations (de 0 à 100), on utilise une lookup table pour affecter une couleur à chaque valeur d'itération (et rendre la fractale plus intéressante à regarder)

# 2 Analyse

## 2.1 Virgule fixe

Comme les calculs sont réalisés sur FPGA, il est plus adapté d'utiliser des nombres en virgule fixe. Étant donné que les DSP sont sur 18 bits, les nombres seront eux aussi sur 18 bits avec :

- Un bit de signe
- Deux bits avant la virgule (pour écrire des nombres entre 3 et -4)
- Quinze bits après la virgule ce qui permet d'avoir une résolution de $3.05 \times 10^{-5}$

## 2.2 Organisation

La base du calcul (l'itérateur) sera réalisé en premier. Il s'agit d'un bloc VHDL capable d'effectuer le calcul

$$z_{k+1} = z_k^2 + C$$

Tout en indiquant si le rayon limite est atteint et le nombre d'itérations en sortie. Si l'entrée `done_in` est activée, alors le nombre d'itérations en sortie est le même que le nombre d'itérations en entrée.

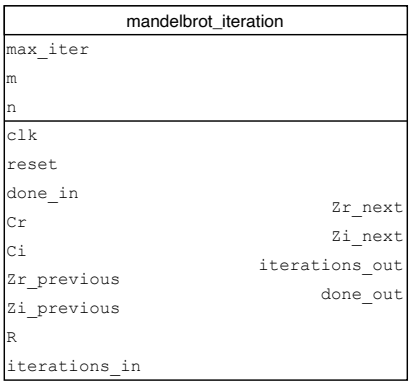



FIGURE 2 – Bloc `mandelbrot_iteration`

Ce bloc sera ensuite réutilisé dans deux versions du programme

1. Bouclage sur lui-même (loop) pour calculer un pixel à la fois
2. Pipeline, 100 itérateurs (le nombre maximal d'itérations) sont reliés les uns après les autres ce qui permet de calculer un pixel par coup d'horloge

### 2.2.1 Bouclage

Un seul itérateur est utilisé et prend initialement les entrées du système puis est bouclé sur lui-même jusqu'au signal `done`

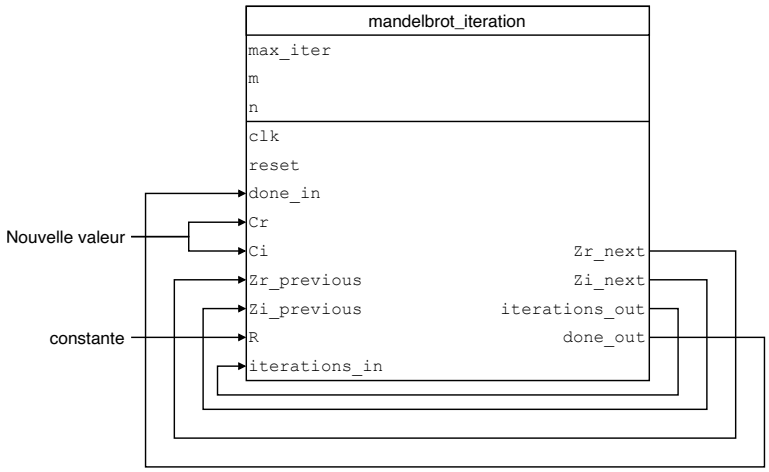



FIGURE 3 – Bloc `mandelbrot_loop`

### 2.2.2 Pipeline

Les itérateurs sont reliés à la suite les uns des autres, les entrées `Cr` et `Ci` sont propagées aux même rythme que le pipeline (registres à la suite)

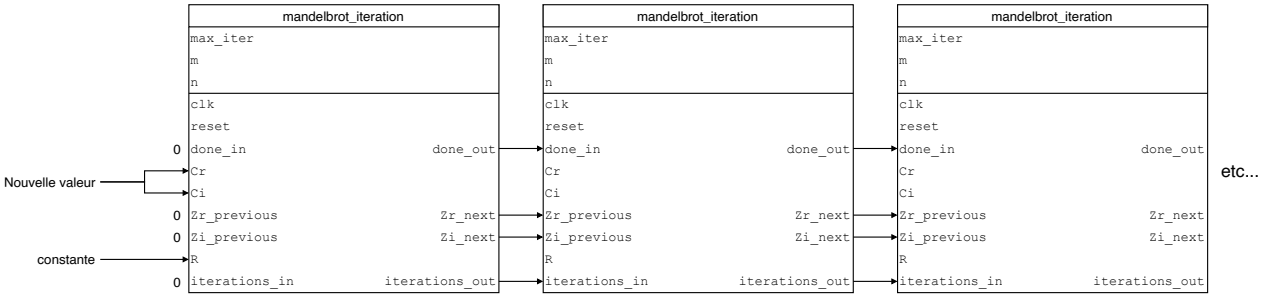



FIGURE 4 – Bloc `mandelbrot_pipeline`

## 2.3 Calculs sur Python

Avant d'implémenter la fractale sur FPGA, il est souhaitable d'avoir un modèle complet pour comparer et débugger le système. Le modèle sera réalisé dans Python en utilisant le package `fixedpoint` [1]. De cette façon il est possible d'effectuer les calculs en virgule flottante (pour avoir la "vraie" fractale) puis les mêmes calculs en virgule fixe afin de comparer de pouvoir créer un banc de test VHDL.

1. `https://pypi.org/project/fixedpoint/`

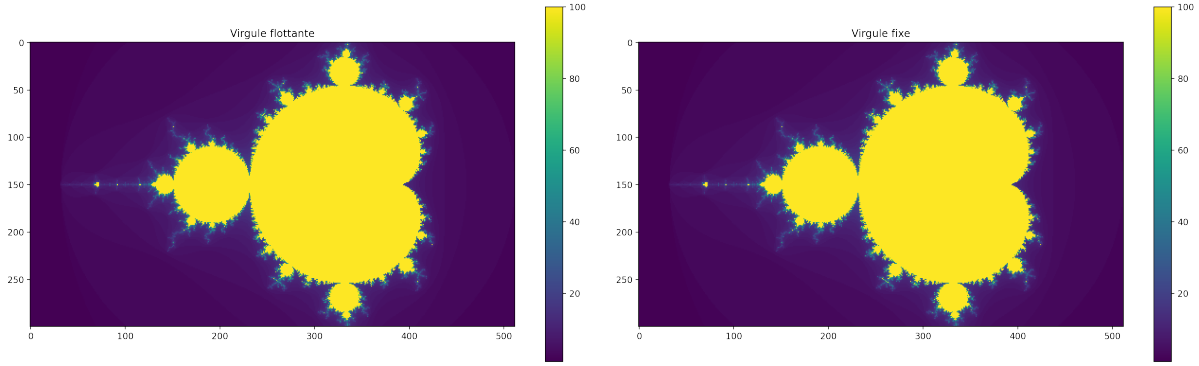



FIGURE 5 – Fractales avec virgule fixe et virgule flottante

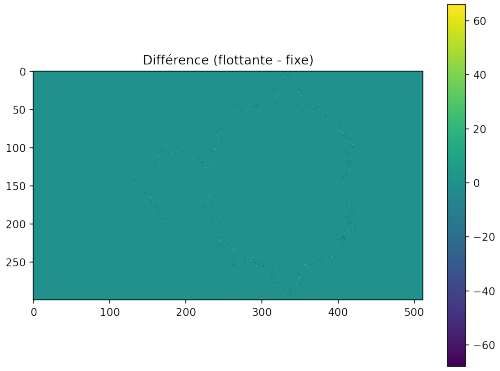



FIGURE 6 – Différence entre les deux résultats

On remarque qu'il existe une différence entre les deux approches (à cause des arrondis), notamment sur les bords de la figure centrale, mais elle est négligeable dans notre application car elle n'impacte pas le rendu visuel.

## 2.4 Couleurs

Afin de déterminer les couleurs de chaque pixel en fonction du nombre d'itérations, on utilise une interpolation sur des points donnés pour chaque couleur (rouge, vert, bleu). Il existe plusieurs choix, mais le celui donné par cette réponse [2] donne un bon résultat

2. `https://stackoverflow.com/a/25816111/8418125`

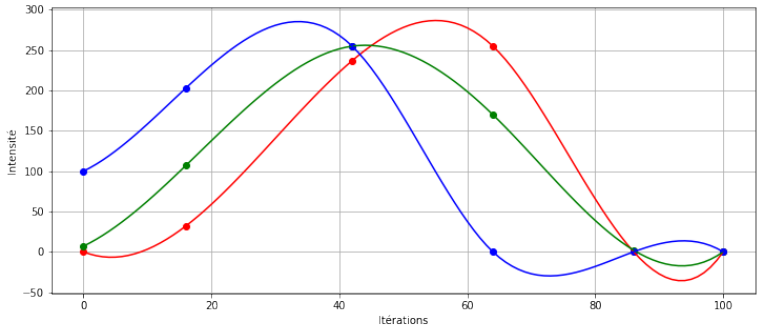



FIGURE 7 – Interpolation cubique sur les couleurs

Après ajustage (pour borner les intensités entre 0 et 255), on obtient la répartition de couleurs suivante en fonction du nombre d'itérations :

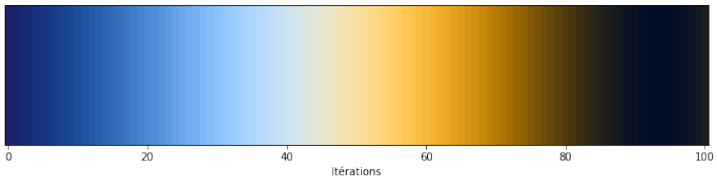



FIGURE 8 – LUT pour la couleur de chaque pixel en fonction des itérations

La LUT est auto-générée dans un package VHDL.

# 3 Réalisation

Tous les codes sont disponibles sur la page GitHub du projet [3]

## 3.1 Organisation

Des fichiers "wrapper" sont utilisés pour gérer la connexion entre le bloc de calcul (loop ou pipeline) et l'interface de la BRAM. La structure du projet est décrite dans la figure 9

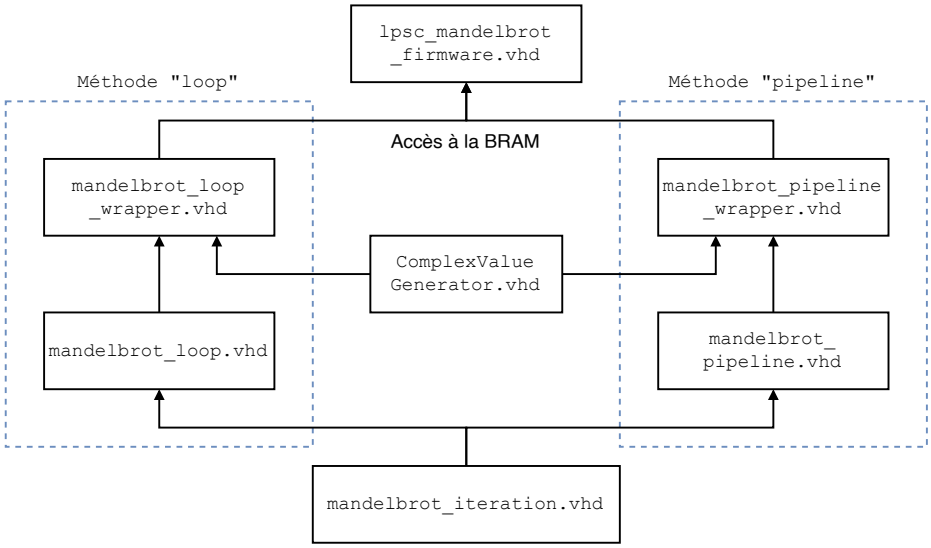



FIGURE 9 – Organisation des fichiers

Les fichiers `mandelbrot_loop.vhd` et `mandelbrot_pipeline.vhd` se chargent de calculer une itération complète pour un pixel. Dans le cas du loop un pixel est réalisé à la fois, dans le cas du pipeline, les pixels sont envoyés à chaque coups d'horloge et les résultats sont présent après un délai de 100 coups d'horloge (le nombre maximal d'itérations).

## 3.2 Itération

L'itérateur (`mandebrot_iteration.vhd`) est basé sur un process synchrone qui réalise tous les calculs en un coup d'horloge. Il est primordial d'avoir des registres sur toutes les sorties afin d'utiliser les itérateurs dans le pipeline ainsi que dans le loop.

## 3.3 Loop

Pour boucler, il suffit simplement de relier les sorties de l'itérateur sur les entrées (car elles sont équipées de registres). Lorsqu'un signal `start` est actif, les entrées sont remplacées par des valeurs initiales.
Il existe aussi un process synchrone pour gérer les sorties du loop (`done` et `iterations`). Le process synchrone désactive la sortie `done` lorsque le calcul est lancé et la ré-active lorsque le calcul est terminé, il va aussi mettre à jour la valeur de `iterations`.

3. `https://github.com/SebastienDeriaz/LPSC_TP3`

## 3.4 Loop wrapper

Le loop wrapper va utiliser le fichier `ComplexValueGenerator.vhd` pour générer les nombres complexes et les transmettre au loop. Lorsqu'un pixel est calculé, le prochain est envoyé. Le loop wrapper est basé sur une machine d'état doit voici le graph

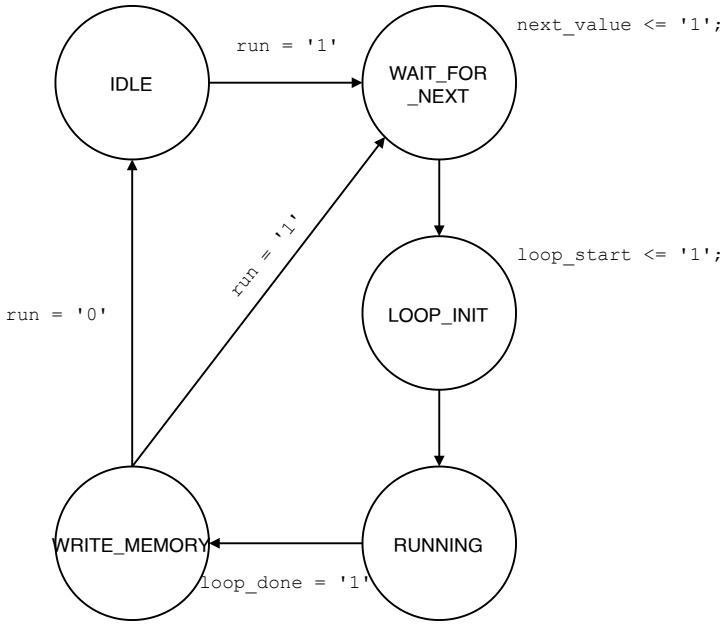



FIGURE 10 – Machine d'état du loop wrapper

Le signal `next_value` demande un nouveau nombre au générateur de nombre complexe (0 dans les autres états). Le signal `loop_start` démarre le calcul du point (0 dans les autres cas).
La sortie du loop wrapper est une interface mémoire pour écrire dans la BRAM du `lpsc_mandelbrot_firmware.vhd`

## 3.5 Pipeline

La base du pipeline est une boucle `for generate` qui instancie 100 itérateurs. Le premier itérateur prend comme entrée les entrées du pipeline (point à calculer), le dernier itérateur fourni les sorties du pipeline (nombre d'itérations) et les itérateurs "centraux" sont reliés ensemble à l'aide de signaux. Afin de déterminer si la sortie du pipeline est valide (lorsque les données fournies à l'entrée sont arrivés à la sortie), le nombre d'itérations à la sortie doit être plus grand que 0 (car au moins une itération est toujours réalisée).
Les valeurs de `Cr` et `Ci` sont également propagées dans des registres pour fournir à chaque itération le bon point de départ. Ceci aurait pu être réalisé dans les itérateurs pour simplifier un peu le code du pipeline.

## 3.6 Pipeline wrapper

Le pipeline wrapper est plus simple que le loop wrapper, il doit simplement transmettre les points fournis par le générateur de nombres complexes au pipeline à chaque coup d'horloge. Les valeurs à la sortie du pipeline sont directement écrites dans la BRAM du `lpsc_mandelbrot_firmware.vhd`.

### 3.6.1 Zoom

Pour vérifier la vitesse de rafraichissement du pipeline, un zoom a été implémenté en faisant varier le point de départ et l'incrémentation du générateur de nombre complexe. Cette approche permet d'utiliser uniquement des compteurs et une incrémentation au lieu de stocker tous les zoom dans une mémoire (comme avec `c_gen.vhd`).
Le zoom représente une grosse partie du process synchrone du pipeline wrapper mais consiste simplement en un compteur qui fourni une horloge à 10 Hz pour effectuer l'itération de zoom et un autre compteur pour revenir à l'état initial (pas de zoom) après une centaine. Il y a aussi un temps d'attente de ≈1 min dans l'état de zoom minimal et maximal afin de laisser le temps d'observer la fractale.

## 3.7 Résolution des couleurs

Afin d'obtenir une meilleure résolution de couleurs, le nombre d'itérations a été stocké dans la mémoire (au lieu de la couleur de chaque pixel) et la LUT est appliquée à la sortie de la mémoire (côté HDMI), ceci permet de diminuer le nombre de BRAM utilisés (7 bits stockés au lieu de 9) tout en améliorant la fidélité des couleurs.

## 3.8 Fréquence

Avec l'implémentation du pipeline, il existait un "worst negative slack" de −8 ns avec une fréquence de 100 MHz. Comme le pipeline est capable de générer l'image à raison de 1 pixel par coup d'horloge, ceci donne un framerate de environ 163 images par seconde ce qui est inutilement important. La fréquence a donc été diminuée à 50 MHz pour respecter les contraintes de timing tout en ayant une fréquence de rafraichissement largement confortable de 82 images par seconde.

# 4 Simulations

Les simulations ont été réalisées sur l'itérateur et le loop. Une fois que ces deux blocs ont été vérifiées, la réalisation en pipeline n'as pas été difficile et n'as pas nécessité de testbench.

## 4.1 Testbench de l'itérateur

Une centaine de stimulis sont calculés dans une simulation Python en choisissant des points de départ au hasard dans le plan complexe considéré. Ils sont ensuite stockés dans un fichier texte puis lu par le testbench VHDL, cette approche est très flexible et permet de régénérer rapidement un testbench. Le testbench a permis de trouver beaucoup d'erreurs liées à l'ordre des opérations (nombre d'itération faux de 1 par exemple). Après correction des erreurs de VHDL il a également fallu corriger les arrondis de nombres à virgule fixe dans Python (voir fonction `resize` dans `mandelbrot_functions.py` qui corrige des arrondis effectués par le package). Le testbench final ne montre aucune erreur

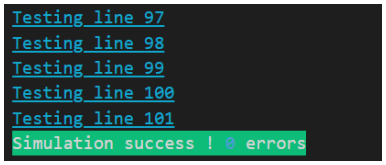



FIGURE 11 – Résultat du testbench de l'itérateur

En plus du système de lecture de fichiers texte par le banc de test, un système de log avec couleur a été implémenté ce qui permet de générer un fichier compatible avec le standard de couleurs ANSI. Une fois le fichier ouvert dans un terminal Linux ou avec une extension appropriée sur VS Code, il est possible d'observer le log en couleur (voir figure 11)

## 4.2 Testbench du loop

Comme pour le testbench de l'itérateur, un fichier contient les stimulis générés avec Python. Le testbench a également montré des erreurs de timing (évaluation d'un signal avant ou après sa vraie valeur) mais une fois les erreurs corrigées, le testbench ne montre plus aucune erreur.

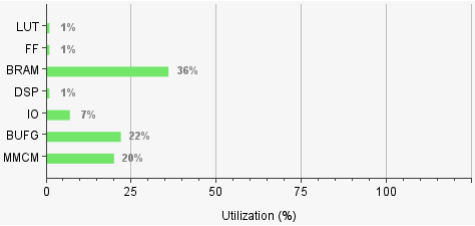

FIGURE 12 – Résultat du testbench du loop

# 5 Conclusion

Le but principal du laboratoire a été atteint (loop avec affichage de la fractale) et un objectif secondaire a été réalisé : l'implémentation sous forme de pipeline. Un zoom "dynamique" a également été réalisé afin de démontrer la vitesse de calcul du pipeline. Il faut noter toutefois que la vitesse du zoom (10 rapprochements par seconde) est toujours bien en dessous de la fréquence de rafraichissement du pipeline (82 images par seconde) avec une fréquence de fonctionnement abitraire de 50 MHz. Cette fréquence pourrait d'ailleurs être augmentée en recherchant précisément le maximum sans produire d'erreurs de timing.

## 5.1 Utilisation du FPGA

### 5.1.1 Méthode loop

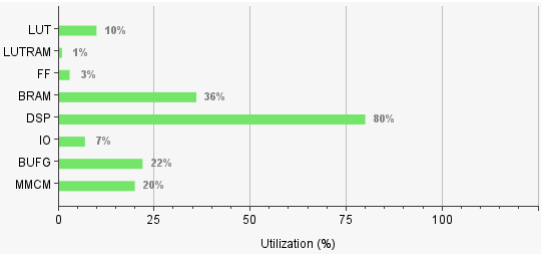



| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 563 | 133800 | 0.42 |
| FF | 211 | 269200 | 0.08 |
| BRAM | 131.50 | 365 | 36.03 |
| DSP | 5 | 740 | 0.68 |
| IO | 19 | 285 | 6.67 |
| BUFG | 7 | 32 | 21.88 |
| MMCM | 2 | 10 | 20.00 |

FIGURE 13 – Graph et table d'utilisation du FPGA fourni par Vivado

L'utilisation des BRAM est importante et l'implémentation de l'itérateur a nécessité 5 DSP

### 5.1.2 Méthode pipeline




| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 12738 | 133800 | 9.52 |
| LUTRAM | 80 | 46200 | 0.17 |
| FF | 8105 | 269200 | 3.01 |
| BRAM | 131.50 | 365 | 36.03 |
| DSP | 591 | 740 | 79.86 |
| IO | 19 | 285 | 6.67 |
| BUFG | 7 | 32 | 21.88 |
| MMCM | 2 | 10 | 20.00 |

FIGURE 14 – Graph et table d'utilisation du FPGA fourni par Vivado

L'utilisation des DSP est très importante, ce qui est attendu car on en attend environ 100x plus qu'avec la version loop (+ les DSP pour le zoom). L'utilisation des BRAM est identique car la mémoire vidéo est la même.

Lausanne, le 24 mai 2022
Sébastien Deriaz