



## EQUIPE DART - INRIA

### RAPPORT DE STAGE

---

#### Générateur des Nombres Aléatoire Parallelle et Dynamique sur FPGA

---

*Stagiaire:*  
Abdurrahman YAŞAR

*Superviseur:*  
Paméla WATTEBLEED

*Responsable:*  
Jean-Luc DEKEYSER

15 Septembre 2011



## **Remerciements**

J'ai eu la chance de pouvoir effectuer mon stage à INRIA sous la direction de Monsieur Jean-Luc Dekeyser professeur à l'Université des Sciences et Technologie de Lille. Je suis vraiment reconnaissant de m'avoir accueilli au de l'équipe DART, de me donner un sujet de stage aussi intéressant, je tiens également à lui dire merci pour ses aides. Je voudrais également remercier à Mme. Paméla Wattebled doctorante à l'université Bretagne Sud-Lorient pour m'avoir donné cette opportunité. Pendant mon stage j'ai pu profité de sa compréhension, de ses idées et de son intuition remarquable.

Les critiques de M. Samy Meftali ont attiré mon attention sur mon sujet. J'aimerais également le remercier pour ses aides.



## Résumé

Pendant mon stage à INRIA j'ai travaillé sur les générateur des nombres aléatoire sur FPGA avec les algorithmes du Prof. Jean-luc Dekeyser siter sa thèse. Ce stage m'a permis d'apprendre les concepts du langage VHDL, les outils de simulation/syntèse. J'ai pu appliquer ces nouvelles connaissances dans le cadre de mon sujet. Dans ce document vous trouverez les algorithmes pour générer des nombre aléatoire sur FPGA mis en place avec leurs principes et leurs implémentations. Mais également leurs temps d'exécution, des tests avec la Méthode Monte Carlo. Je compare également les différents algorithmes.



# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Les Algorithmes . . . . .	4
1.1.1	Algorithme 1 : Accès Iteratif . . . . .	4
1.1.2	Algorithme 2 : Accès Direct . . . . .	4
1.2	Les Outils . . . . .	6
1.2.1	Carte FPGA . . . . .	6
1.2.2	Environnement Utilisée . . . . .	6
<b>2</b>	<b>Implémentation des Algorithmes</b>	<b>8</b>
2.1	Implémentation des Algorithmes . . . . .	8
2.1.1	Implémentation de l'Algorithme Iteratif . . . . .	8
2.1.2	Implémentation de l'Algorithme Direct . . . . .	9
2.2	Sur la Carte FPGA . . . . .	11
2.2.1	Algorithme Itérative . . . . .	11
2.2.2	Algorithme Direct . . . . .	12
2.2.3	Les Deux en Même Temp . . . . .	14
<b>3</b>	<b>Analyses et Temps d'Exécutions</b>	<b>15</b>
3.1	Algorithme Iteratif . . . . .	15
3.1.1	Un peu d'Analyse . . . . .	16
3.1.2	Graphes . . . . .	17
3.2	Algorithme Direct . . . . .	19
3.2.1	Un Peu d'Analyse et Graphes . . . . .	22
3.3	Comparaison . . . . .	23
3.3.1	Resources Utilisées . . . . .	23
3.3.2	Comparaison - $X_1$ à $X_N$ . . . . .	24
<b>4</b>	<b>Problèmes et Solutions Proposées</b>	<b>26</b>
4.1	Problemes . . . . .	26
4.1.1	Parité . . . . .	26
4.1.2	Répétition . . . . .	26
4.2	Idées pour les problèmes . . . . .	27
4.2.1	Idée 1 : Modulo avec le nombre premier plus proche . . . . .	27
4.2.2	Idée 2 : Décalage circulaire . . . . .	28
4.2.3	Idée 3 : Permutation circulaire . . . . .	28
4.2.4	Répétition . . . . .	28

<b>5 Méthode de Monte Carlo</b>	<b>30</b>
5.1 Détermination de la valeur de $\pi$ . . . . .	30
5.2 Implémentation . . . . .	32
5.2.1 Exécution . . . . .	33
5.2.2 Points sur un Graphe . . . . .	34
<b>6 Conclusions et Perspectives</b>	<b>35</b>
6.1 Conclusions . . . . .	35
6.1.1 Ressources Utilisés . . . . .	35
6.1.2 Temps d'Exécution . . . . .	36
6.1.3 Problèmes et Solutions . . . . .	36
6.1.4 Méthode Monte Carlo . . . . .	36
6.2 Perspectives . . . . .	36
<b>A Codes</b>	<b>39</b>
A.1 Algorithme Iteratif . . . . .	39
A.2 Algorithme Direct . . . . .	40
<b>B Temps D'Exécutions</b>	<b>41</b>
B.1 Algorithme Itérative . . . . .	41
B.2 Algorithme Direct . . . . .	42
<b>C Algorithme Itératif avec permutation</b>	<b>43</b>

# 1

## Introduction

Le sujet de mon stage était de faire une générateur des nombre aléatoire sur un FPGA, donc un générateur de nombres aléatoire matériel.

Tout d'abord pour générer des nombres Prof. Jean-luc Dekeyser a proposé deux algorithmes different qui produisent les même nombres. Mon premier objectif était d'écrire les codes vhdl pour ces algorithmes et après tester leurs performances.

### 1.1 Les Algorithmes

Les 2 algorithmes proposés produisent les même nombres. La difference entre ces deux algorithmes est le méthode utilisé: Premier algorithme calcule les nombres itérativement, donc pour accéder N'ième nombre on calcule N-1 nombre précédent. La deuxième peut accéder N'ième nombre directement.

#### 1.1.1 Algorithme 1 : Accés Iteratif

Pour une constante A on calcule le N'ième nombre aléatoire en multipliant A avec (N-1)'ième nombre aléatoire.

$$X_n = (X_{n-1} * A) \bmod 2^p \quad (1.1)$$

Sur cet équation :

$X_n$  indique le n'ième nombre aléatoire,  
 $X_{n-1}$  indique le (n-1)'ième nombre aléatoire,  
p indique la taille en nombre de bit de notre nombre(mot).

#### 1.1.2 Algorithme 2 : Accés Direct

On obtient cet algorithme par récurrence de l'algorithme Iteratif:

$$X_n = (X_{n-1} * A) \bmod 2^p$$

Pour n = 1 :

$$X_1 = (X_0 * A) \bmod 2^p \quad (1.2)$$

Pour n = 2 :

$$X_2 = (X_1 * A) \bmod 2^p = ((X_0 * A) * A) \bmod 2^p = (X_0 * A^2) \bmod 2^p \quad (1.3)$$

...

Pour n = N

$$X_N = (X_0 * A^N) \bmod 2^p \quad (1.4)$$

Pour n=0 et n=1 on a montré que l'équation 1.4 est vrai donc maintenant on suppose que 1.4 est vrai pour N.

Pour n=N+1

$$X_{N+1} = (X_N * A) \bmod 2^p$$

on a supposé que

$$X_N = (X_0 * A^N) \bmod 2^p$$

donc

$$X_{N+1} = ((X_0 * A^N) * A) \bmod 2^p = (X_0 * A^{N+1}) \bmod 2^p \quad (1.5)$$

à la fin pour n = N+1 :

$$X_{N+1} = (X_0 * A^{N+1}) \bmod 2^p \quad (1.6)$$

Donc on a démontré que l'équation 1.4 est vrai:

$$X_N = (X_0 * A^N) \bmod 2^p$$

Grace à cet algorithme on peut accéder au N'i'eme nombre aléatoire directement mais notre travail n'a pas encore fini. On va optimiser cet algorithme pour le système binaire. Dans les équations suivant '.\*' indique le symbole du concatenation et  $n_i$  indique une bit pour  $p>i\geq 0$ .

D'abord on écrit N dans le base binaire. Alors;

$$N = n_{p-1}.n_{p-2}....n_1.n_0$$

Après si on transforme N :

$$N = \sum_{0 \leq i < p} n_i * 2^i$$

Maintenant on remplace la formule qu'on a trouvé avec N dans l'équation 1.4 et après l'équation 1.4 devient :

$$X_N = (X_0 * A^{\sum_{0 \leq i < p} n_i * 2^i}) \bmod 2^p$$

Enfin on peut faire la dernier transformation:

$$A^{\sum_{0 \leq i < p} n_i * 2^i} = \prod_{0 \leq i < p} A^{n_i * 2^i}$$

C'est à dire si  $n_i$ -ième bit est 1 on calcule  $A^{2^i}$  sinon on ne calcule pas. Par exemple pour  $N=00001001$  on va calculer seulement ( $A^{2^0} * A^{2^3}$ ) =  $A^9$ . Donc on va calculer N-ième puissance de A plus facile qu'avant.

Par conséquent l'algorithme devient :

$$X_N = \left( X_0 * \left( \prod_{0 \leq i < p} A^{n_i * 2^i} \right) \right) \bmod 2^p \quad (1.7)$$

## 1.2 Les Outils

J'ai utilisé différents outils pour réaliser mon projet afin de simuler, tester mon code.

### 1.2.1 Carte FPGA

Le code que j'ai produit a été testé sur la carte FPGA Digilent NEXYS2 qui a les propriétés suivantes:

- Xilinx Spartan3E-500 ou 1200
- 16Mo Micron Cellular RAM
- 16Mo Numonyx StrataFlash
- USB2 Interface
- Buttons, Switches et LEDs sur la carte
- 100Mhz+ Expansion Connector
- 8-bit VGA, RS-232 serial et PS/2 Ports

### 1.2.2 Environnement Utilisée

J'ai développé mon projet avec la plateforme Xilinx ISE 12.4. J'ai écrit mes codes en VHDL et généré le fichier exécutable (.bit) avec ISE.

#### Implémentation

J'ai implémenté le fichier .bit à la carte FPGA en utilisant Adept qui est une outil de digilent pour programmer la carte.

## **Simulation**

Pour mes simulations et tests j'ai utilisé le simulateur ISIM qui vient avec ISE.

Pour mes tests et simulations j'ai suivi la procédure suivant:

- D'abord la simulation comportementale "Behavioral Simulation"
- Après "Post and Map Simulation"
- Enfin "Post and Route Simulation"

## 2

# Implémentation des Algorithmes

Dans la suite du document , je vais expliquer comment j'ai écrit les composants et comment ils fonctionnent.

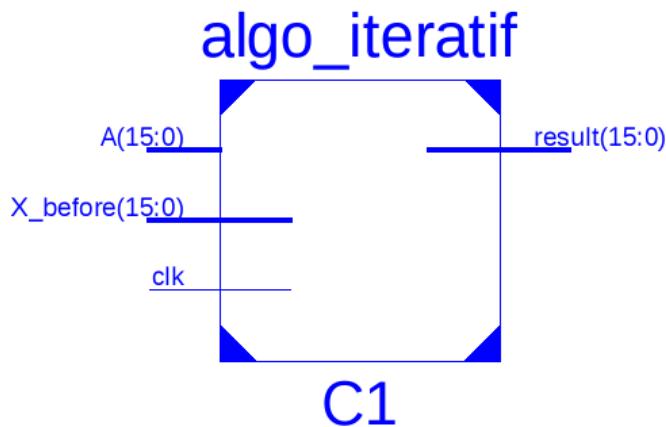
## 2.1 Implémentation des Algorithmes

### 2.1.1 Implémentation de l'Algorithme Iteratif

Pour calculer des nombre aléatoire on utlise le formule:

$$X_n = (X_{n-1} * A) \bmod 2^p$$

Donc pour calculer n'ième nombre on a besoin de savoir (n-1)'ième nombre et une constante A. Donc notre composante pour cet algorithme va prendre A et  $X_{n-1}$  comme entrées. D'autre part on a bessoin d'une horlage('clk') alors notre troisième entrée est la 'clock'. Nous avons une unique sortie qui est notre nombre généré sur 16 bits.



Comme la figure ci-dessus :

- A est sur 16 bit et il indique la constante A à fournir.
- X\_before est sur 16 bits et il indique  $X_{n-1}$  qui est le nombre généré précédent.
- clk indique l'horlage.
- result est aussi sur 16 bit et il indique notre nombre aléatoire.

J'ai choisi d'utiliser 16 bits pour tous les entrées et sorties en raison de la facilité d'affichage des nombres générées sur la carte FPGA mis à ma disposition.

### Pseudo-Code

Cet algorithme est plus facile que le deuxième car ce composante ne fait qu'une multiplication.

```
{Créer un process controle avec clock : Process(clk)}
Process(clk)
{Créer une variable temporelle tempresult}
Begin
if clk = 1 and clk'Event then
    tempresult ← Xbefore * A
end if
EndProcess
result ← tempresult
```

Pour voir le code écrit en VHDL de l'algorithme Iteratif voir l'Appendix A.1.

#### 2.1.2 Implémentation de l'Algorithme Direct

Comme on avait montré dans le chapitre précédent pour calculer des nombres aléatoires avec l'algorithme directe on utilise la formule suivant :

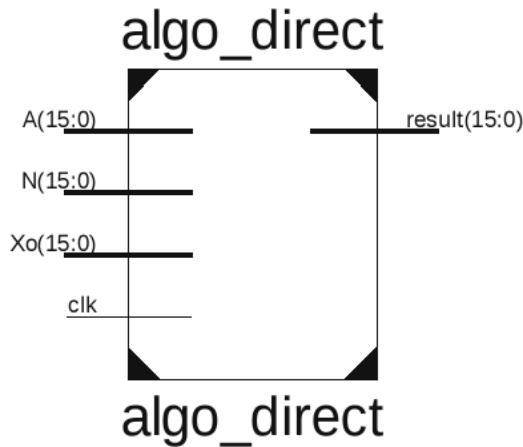
$$X_N = \left( X_0 * \left( \prod_{0 \leq i < p} A^{n_i * 2^i} \right) \right) \bmod 2^p$$

Dans cette équation  $n_i$  indique le n'ième bit de N qui est l'ordinalité du nombre à générée. Donc comme vous voyez facilement si  $n_i$  est égale à 0 on fait rien et grâce à cette situation on calcule des nombres plus vite.

Pour le composant de cette algorithme on aura besoin de 4 entrée une pour A qui est une constante une pour  $X_0$  qui est le premier élément de la suite une pour N qui est l'ordinalité du nombre à générée et une pour l'horlage. À la sortie nous obtiendrons le nombre aléatoire générée. Tous ces signaux seront sur 16 bits sauf l'horlage parce qu'il y a seulement 4 afficheur 7 segments sur la carte fpga donc faire des opérations sur 16 bits est plus facile à afficher. Un afficheur 7 segments permet d'afficher 1 chiffre coder sur 4 bits.

Même si l'accès direct à la N'ième nombre aléatoire cette composante a besoin de beaucoup plus de ressource. Je ferais une critique dessus dans le prochain chapitre. L'algorithme direct a besoin d'un registre pour garder les puissance de A de 0 a  $2^{p-1}$ . Car grâce à ces registres on n'aura plus besoin de calculer des puissances de A en chaque itération.

Voilà une petite figure qui montre les entrées sorties pour cette composante:



### Pseudo-Code

Pour cet algorithme avant de créer un process on doit initialisé les puissances( $2^0, 2^1, 2^2, \dots, 2^{p-1}$ ) de A dans un tableau pour pouvoir utiliser après. Par conséquent pour le premier nombre on va perdre plus de temps mais pour les suivants on gagnera du temps. Dans le pseudo-code tableau est nomée 'd'.

```

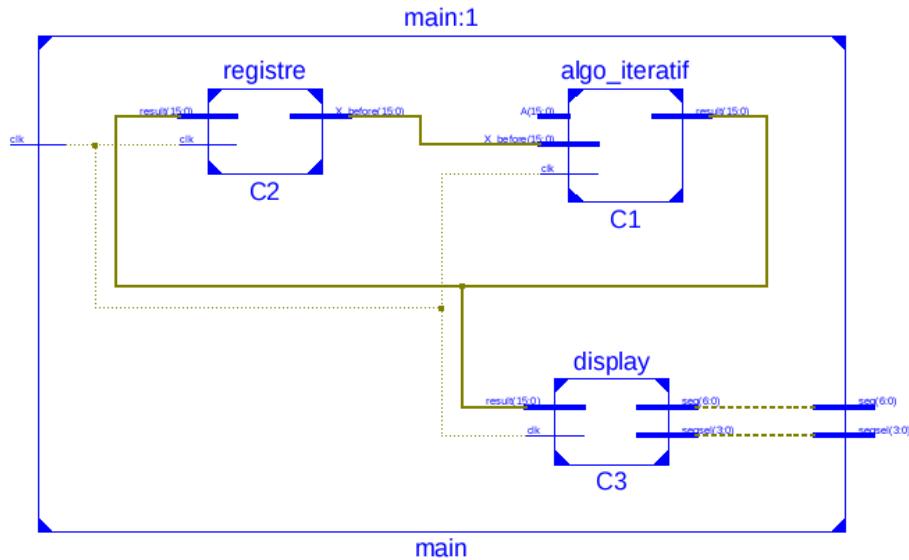
 $d(0) \leftarrow X_0$ 
 $i \leftarrow 1$ 
for  $i < p$  do
     $d(i) \leftarrow d(i - 1) * d(i - 1)$ 
end for
{Créer un process contrôle avec clock : Process(clk)}
Process(clk)
{Créer une variable temporelle  $temp_{result}$ }
Begin
if  $clk = 1$  and  $clk'Event$  then
    for  $i < p$  do
        if  $N(i) = '1'$  then
             $temp_{result} \leftarrow temp_{result} * d(i)$ 
        end if
    end for
end if
 $result \leftarrow temp_{result}$ 
EndProcess
    
```

Pour voir le code écrit en VHDL de l'algorithme Direct voir l'Appendix A.2.

## 2.2 Sur la Carte FPGA

### 2.2.1 Algorithme Itérative

Pour voir les nombres générées, j'ai utilisé le système suivant pour le test: qui calcule d'abord des nombres et après les affiche.



- count est le composante pour ralentir la génération des nombres. il attend quelques temps et après il modifie  $X_{before}$  pour le nombre prochain.
- algo\_iteratif est le composant où on calcule le nombre.
- display est le composant pour afficher les nombres sur la carte.

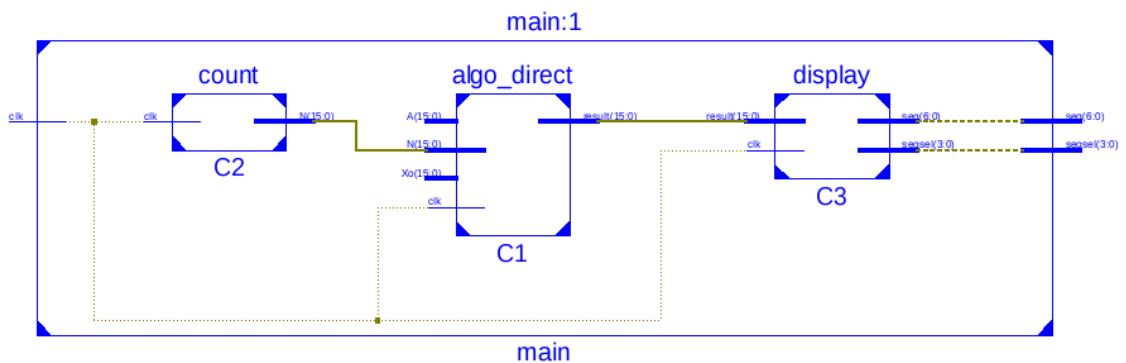
Par conséquent le système a généré des nombres sans problème.



Voila un petit figure de la carte FPGA avec affichage d'un nombre généré.

### 2.2.2 Algorithme Direct

Pour voir les nombres générées, j'ai utilisé le système suivant pour le test: qui calcule d'abord des nombres et après les affiche.



- count est le composant pour augmenter le N et un peu d'atteindre.
- algo\_direct est le composant où on calcule le nombre.
- display est le composant pour afficher des nombres sur la carte.

Par conséquent le système a généré des nombres sans probleme.

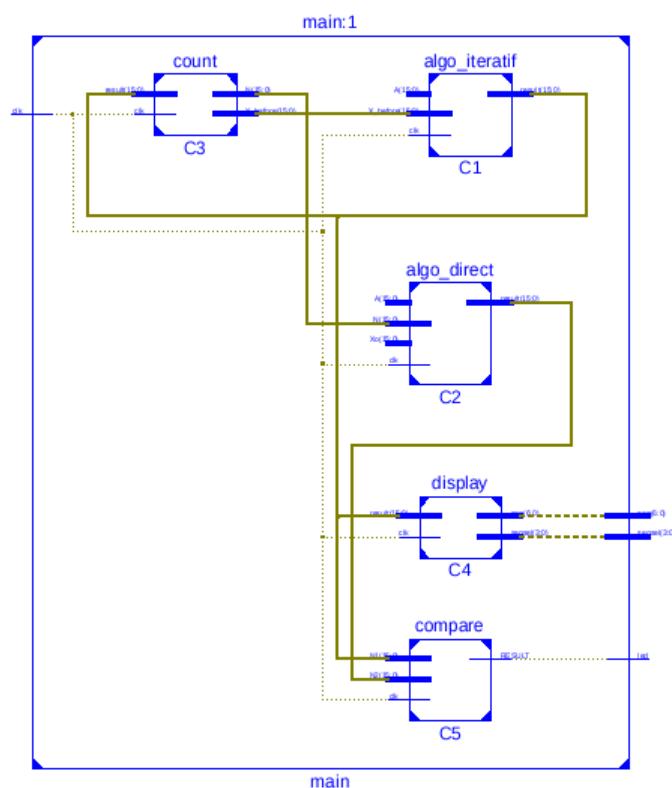


Voila un petit figure de la carte FPGA avec affichage d'un nombre générér.

### 2.2.3 Les Deux en Même Temp

Pour voir si les deux composantes générèrent les mêmes nombres j'ai fait un petit système. Dans ce système les deux composantes vont produire des nombres 0 à N parallèlement et un troisième composant va comparer ces nombres et s'ils ne sont pas égaux il va afficher une led (Pour mes tests c'est le LD1 de la carte Nexys2). D'autre part un composant nommé display va afficher les nombres générée par le composant algo\_iteratif ou par le composant algo\_direct.

Pour voir les nombres générées, j'ai utilisé le système suivant pour le tester: qui calcule d'abord des nombres et après les affiche.



# 3

## Analyses et Temps d'Exécutions

Dans le chapitre précédent on a montré que l'implémentation des 2 algorithmes fonctionne bien sur la carte FPGA et les deux produisent les même nombres avec les mêmes constantes.

Dans ce chapitre notre premier objectif est de d'abord trouver les temps d'exécutions de chaque algorithmes. Enfin pour la vérification de ces temps d'exécution une démonstration sur la carte FPGA est faite comme le dernier chapitre mais cette fois on va travailler avec les temps réels de la production des nombres.

### 3.1 Algorithme Iteratif

Notre algorithme itératif :

$$X_n = (X_{n-1} * A) \bmod 2^p$$

X <sub>n-1</sub>	A	Temps D'exécution(ps)
0	0	5487
0	1	5487
1	0	5487
0	1	5487
1	1	17727
1	3	19411
1	15	19411
1	255	19411
1	2 <sup>12</sup> - 1	19411
2	2 <sup>16</sup> - 1	19411
4	2 <sup>16</sup> - 1	18758
2 <sup>6</sup>	2 <sup>16</sup> - 1	18758
2 <sup>16</sup> - 1	2 <sup>16</sup> - 1	17727

On ne fait qu'une opération de multiplication pour cette composante mais encore le temps d'exécution peut varier selon  $X_{n-1}$  et la constante A.

Après les tests on a noté que si l'un de  $X_{n-1}$  ou A est égal à zéro on obtient le résultat dans 5487 ps qui est le plus vite. Mais pour autres cas le temps d'exécution varie parce que les pins n'obtiennent pas les résultats en même temps. Par exemple pour  $X_{n-1} = 2^{16} - 1$  et A = 1 temp d'exécution est 19411 ps mais pour  $X_{n-1} = 2^{16} - 1$  et A =  $2^{16} - 1$  temps d'exécution devient 17,727 ps.

Ensuite après les tests on a obtenu les temps d'exécution entre 17727 ps et 19411 ps. Donc avec un horlage de 50Mhz(20 ns) on peut calculer les nombres dans un T-cycle avec ce composant. Vous pouvez voir les temps d'exécutions pour des autres constantes dans l'appendice B.1.

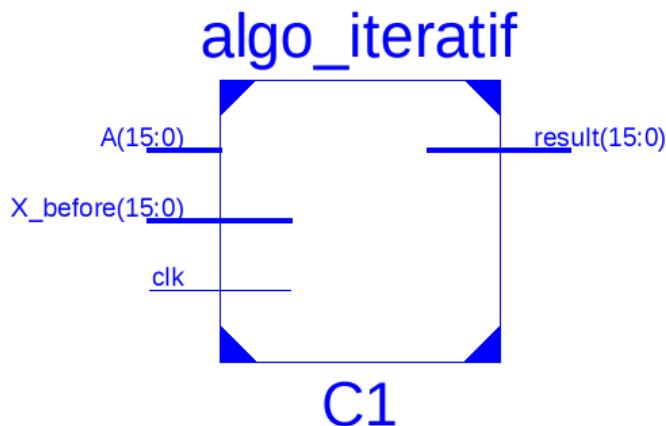
Pour ce composant on n'a pas de temps d'initialisation donc pour chaque période d'horlage de 50Mhz on peut calculer un nouveau nombre.

### 3.1.1 Un peu d'Analyse

On a vu qu'avec ce composant on peut calculer un nombre au moins dans 19411ps qui tend vers 20ns. Donc comme on a dit avant avec un horlage de 50Mhz on peut calculer un nombre.

#### Quel est le temps passé pour accéder N'ième nombre

Comme vous pouvez voir dans la figure suivant le composant de l'algorithme itératif ne prend que  $X_{n-1}$  et A. Pour accéder N'ième nombre.



Sachant que pour chaque nombre on a besoin au moins 20ns alors pour accéder à N on aura besoin  $20*N$  ns. Si on fait une formule pour le temps passé pour accéder à N:

$$f(N) = T * N + T_{init}$$

Ci-dessus f est la fonction qui donne le temps pour accéder à  $X_N$ , T indique le période donc  $T = 20\text{ns}$  et  $T_{init}$  indique le temps d'initialisation donc il est 0. Alors notre équation devient :

$$f(N) = 20 * N$$

La fonction f est donc linéaire.

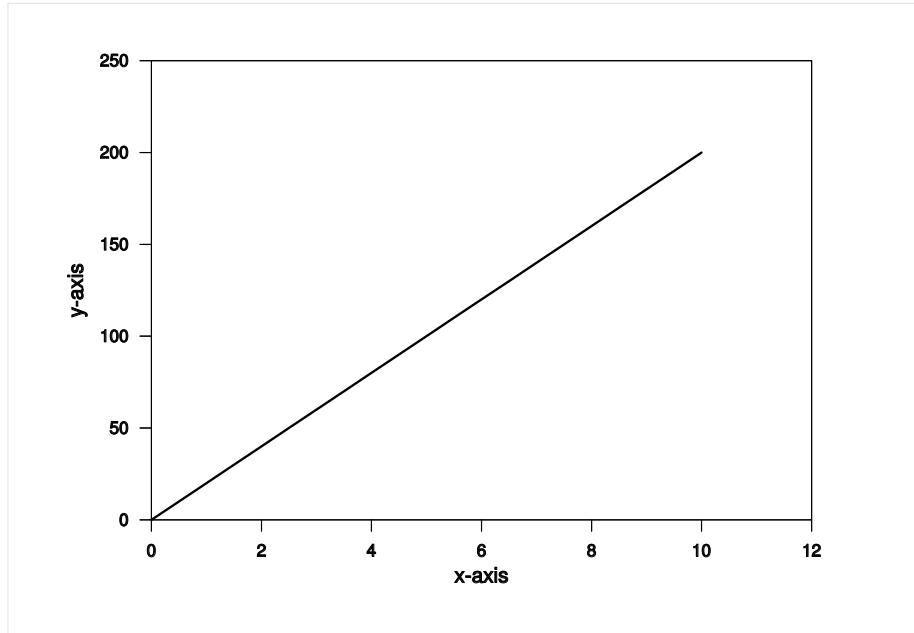
### 3.1.2 Graphes

Dans cette partie on va vous montrer les graphes de temps-N pour voir la meilleure performance.

#### Accès $X_N$

Dans la figure suivant vous voyez le graphe de la fonction précédent. Il n'y a pas une besoin de temps pour initialiser et pour chaque 20ns il calcule un nouveau nombre. x-axis représente N qui est l'ordinalité du nombre et y-axis représente le temps d'exécution.

$$f(N) = 20 * N$$



On a vu que pour accéder  $X_N$  on a besoin de calculer les  $N-1$  nombres précédents. Maintenant si on veut calculer une suite des nombres par exemple de 0 à N le temps passer va changer comment? Pour trouver des fonctions et dessiner les graphes on a 2 scénarios.

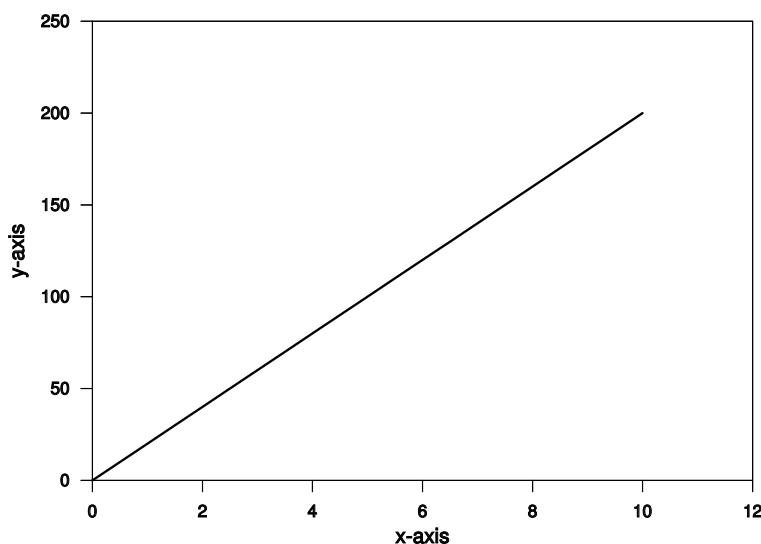
#### Scénario 1 : Avec un registre

Dans le premier scénario après le calcul de chaque nombre on va garder la valeur du résultat dans un registre et pour calculer le nombre suivant on va

utiliser la valeur dans ce registre à la place de  $X_{n-1}$  donc à chaque fois on va calculer une nouveau nombre. On voit que la fonction du temps pour produire des nombre de  $X_0$  à  $X_N$  est la même que la fonction de temps pour accéder à  $X_N$ . Donc la fonction est :

$$g(N) = 20 * N$$

Alors le graphe est le même que le précédent.



### Scénario 2 : Sans Registre

Dans le deuxième scénario on doit calculer à chaque fois tout les nombres jusq'au  $X_N$ . C'est à dire pour le i<sup>e</sup>ème nombre on doit calculer  $X_1$  après  $X_2$  jusq'à  $X_{N-1}$  et enfin  $X_N$ . En d'autres termes chaque fois on doit accéder à  $X_i$  de 1 à N. Alors :

$$h(N) = \sum_{i=1}^N (20 * i)$$

On peut écrire cette équation sous la forme :

$$h(N) = 20 * \sum_{i=1}^N i$$

et on sait que :

$$\sum_{i=1}^N i = \frac{N * (N + 1)}{2}$$

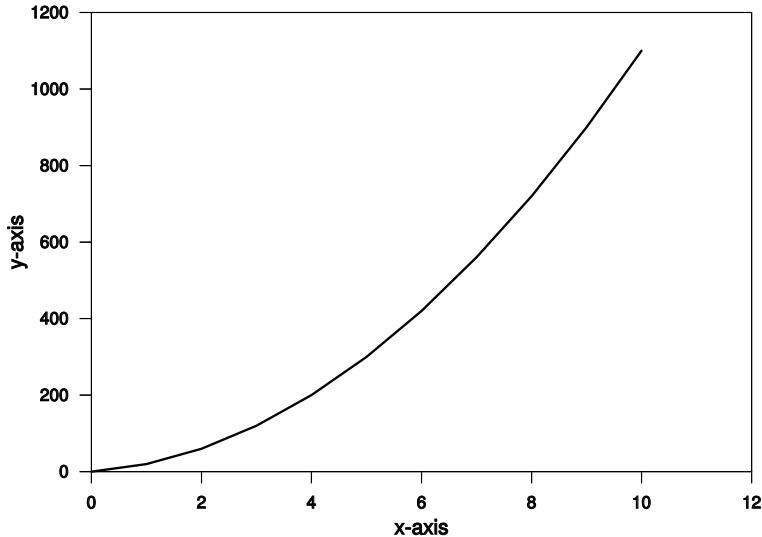
Donc:

$$h(N) = 20 * \frac{N * (N + 1)}{2}$$

qui est égale à:

$$h(N) = 10N^2 + 10N$$

Enfin voilà la graphe pour le deuxième scénario:



Dans ce graphe x-axis montre le N qui est cardinalité des nombre va générer et y-axis montre le temps passé

### 3.2 Algorithme Direct

L'algorithme direct :

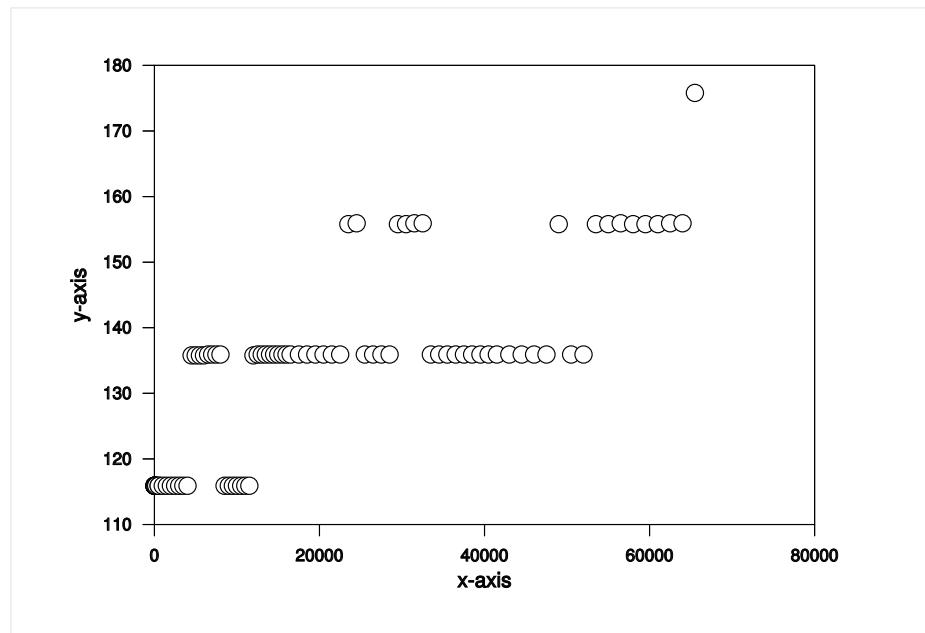
$$X_N = \left( X_0 * \left( \prod_{0 \leq i < p} A^{n_i * 2^i} \right) \right) \bmod 2^p$$

L'implémentation de cet algorithme sur la carte FPGA utilise beaucoup plus de ressources que le premier. Parce que tout d'abord avant de commencer à calculer les nombres on a besoin d'un temps d'initialisation pour calculer les puissances de A qui sont  $A^{2^0}, A^{2^1}, A^{2^2}, \dots, A^{2^{p-1}}$  et on va les garder dans une tableau pour les utiliser pendant les calculs.

Tout d'abord on a changé le code un peu pour faire des tests pour voir les temps d'initialisation. Après ces tests on a mesuré le temps d'initialisation 99269ps(environ 100ns). C'est à dire qu'on peut commencer à calculer le premier nombre aléatoire qu'après cette phase d'initialisation qui dure 100ns.

Ensuite, après avoir le temps d'initialisation on a continué à faire des tests pour voir les temps d'exécutions. D'abord on a mesuré les temps d'exécution pour des constantes  $A = 3$  et  $X_0 = 1$  et on a changé seulement N. Dans le graphe suivant vous voyez les points qui indique les temps d'exécution pour different

$N$  de  $A = 3$  et  $X_0 = 1$ . Dans ce graphe x-axis indique  $N$  et y-axis indique temps d'exécutions.



J'ai choisi les  $N$  en augmentant  $N$  périodiquement. Vous pouvez voir les valeurs de  $N$  et leur temps d'exécutions dans le tableau ci-dessus:

N	Temps D'exécution(ns)	N	Temps D'exécution(ns)
1	115.927	17501	135.927
2	115.897	18501	135.927
3	115.927	19501	135.927
4	115.922	20501	135.927
5	115.927	21501	135.927
10	115.922	22501	135.927
100	115.922	23501	155.789
200	115.922	24501	155.927
500	115.912	25501	135.927
501	115.927	26501	135.927
1001	115.927	27501	135.927
1501	115.927	28501	135.927
2001	115.927	29501	155.789
2501	115.927	30501	155.789
3001	115.927	31501	155.927
3501	115.927	32501	155.927
4001	115.927	33501	135.927
4501	135.789	34501	135.927
5001	135.789	35501	135.927
5501	135.789	36501	135.927
6001	135.789	37501	135.927
6501	135.927	38501	135.927
9501	115.927	46001	135.927
10001	115.927	47501	135.927
10501	115.927	49001	155.789
11001	115.927	50501	135.927
11501	115.927	52001	135.927
12001	135.789	53501	155.791
12501	135.927	55001	155.791
13001	135.927	56501	155.927
13501	135.927	58001	155.789
14001	135.927	59501	155.789
14501	135.927	61001	155.789
15001	135.927	62501	155.927
15501	135.927	64001	155.927
16001	135.927	65501	175.789
16501	135.927		

Après les tests on a vu que temps d'exécution du nombre varie selon N et il est entre 115.927 ns et 175.789 ns. Ces valeurs contiennent les temps d'initialisation. Après l'initialisation on peut calculer des nombre entre 16.667 ns et 76.529 ns pour des constantes  $A = 3$  et  $X_0 = 1$ .

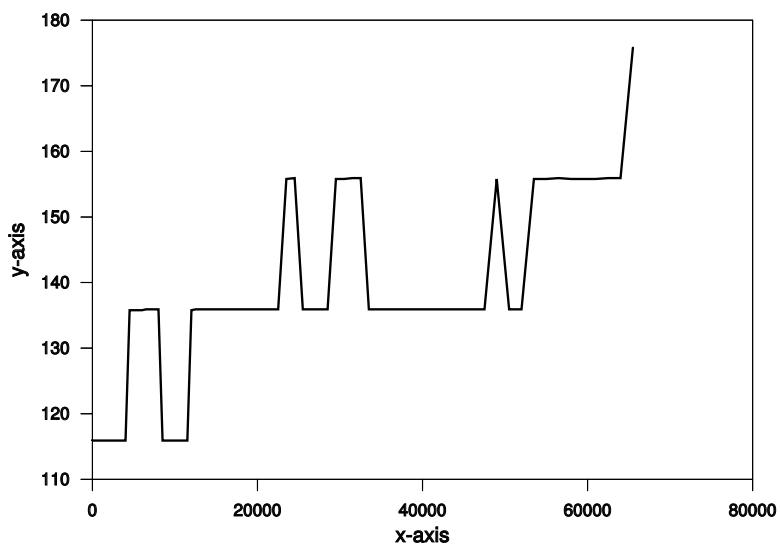
À la suite pour voir si le temps d'exécutions dépend aussi des constantes A et  $X_0$ , on a fait ces tests avec différents constantes mais j'ai pu constater que le temps d'exécution ne dépend que de N.

### 3.2.1 Un Peu d'Analyse et Graphes

Ici on va travailler un peu sur le temps des calculs.

#### Accés à N

Le premier but de cet algorithme est d'accéder  $X_N$  plus vite possible. On a vu que ce composant peut générer des nombres au pire de cas en 176 ns avec le temps d'initialisation. Afin de mieux voir j'ai connecter les points qu'on a trouvé pendant les tests. Voilà le graphe(x-axis:N et y-axis:Temp):



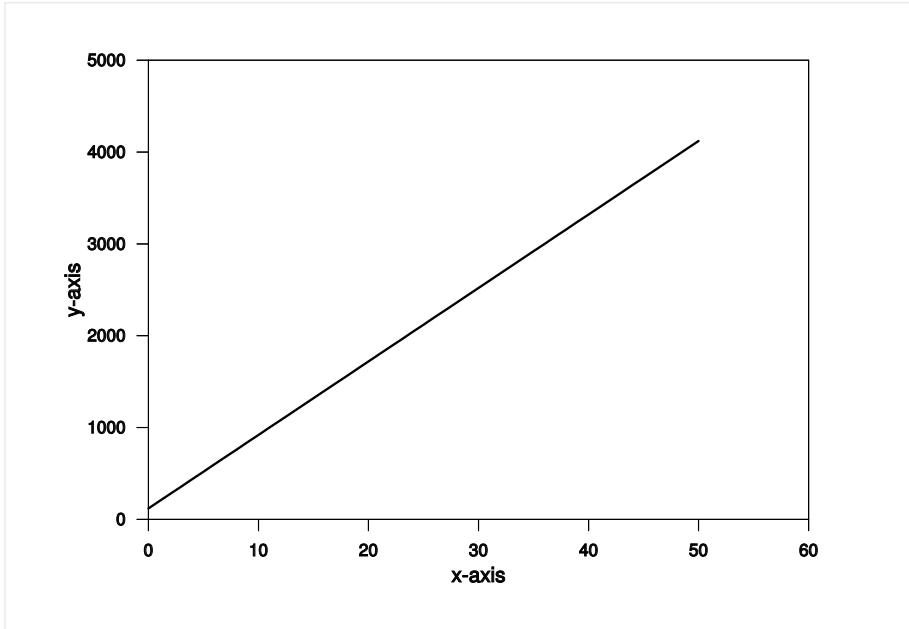
Alors si on suppose qu'au début on a fait l'initialisation(environ 100ns), pour garantir le calcul du nombre, on aura besoin de 76 ns. Donc avec un clok 50Mhz(20ns) on a besoin de 4 T-cycle(80ns) pour générer un nombre.

#### De $X_1$ à $X_N$

Comme on avait fait avant avec le composant itératif on veut produire des nombres de  $X_1$  à  $X_N$ . On peut calculer un nombre dans 80ns(4T) et on a un temps d'initialisation de 100ns. C'est à dire Pour chaque nombre on auras besoin de 80ns et pour le premier on a besoin de 100ns. Alors:

$$f(N) = 80 * N + 100$$

Dans la fonction N indique le cardinalité du nombre qu'on veut générer et cette fonction. Voilà le graphe de la fonction x-axis indique N et y-axis indique le temp:



### 3.3 Comparaison

Après avoir regretté les temps d'exécutions pour chaque composant, nous allons maintenant faire une comparaison des deux algorithmes. Dans cette partie nous commençons par discuter des ressources matérielles utilisées par les deux algorithmes puis nous comparons les temps d'exécutions de ces différents algorithmes.

#### 3.3.1 Resources Utilisées

Comme nous pouvons le voir dans le tableau ci-dessous l'algorithme direct consomme beaucoup plus de ressources que l'algorithme itératif. En effet comme nous l'avons vu l'algorithme itératif ne fait qu'une multiplication et donc n'utilise aucun luts, juste un multiplieur disponible sur la carte FPGA. L'algorithme direct a besoin de plus de ressources puisqu'il ne fait pas un simple multiplication mais il utilise une boucle for et nous avons toute la phase d'initialisation avec la sauvegarde des puissances de A (comme vu précédemment) qui consomme des ressources matérielles.

Resource	Disponible	Composante Itérative	Composante Direct
Slices	8672	0	320
LUTs	17344	0	619
IOBs	250	49	65
BUFGMUX	24	1	1
MULTT18	28	1	28

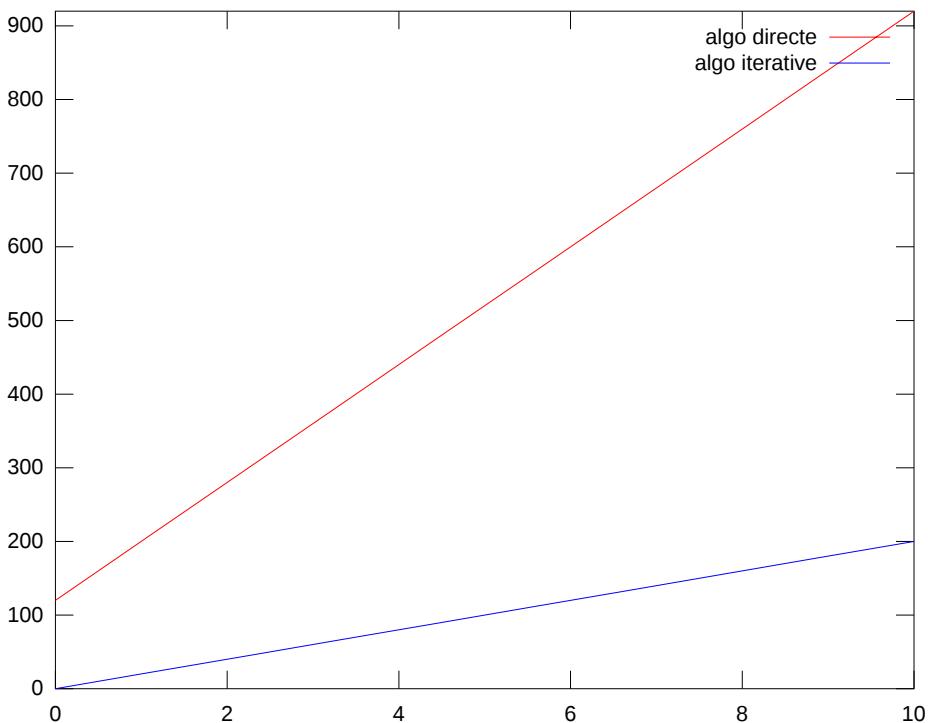
### 3.3.2 Comparaison - $X_1$ à $X_N$

Dans cette section, nous allons comparer les deux algorithmes en terme de temps d'exécution. J'ai voulu tester deux scénarios différents en ce qui concerne l'algorithme itératif. Le premier scénario est le fait de conserver le résultat précédent pour calculer  $X_n$ , donc nous utilisons un registre. Le deuxième scénario possible est de ne pas avoir de registre, et donc de recalculer les  $X_0 \dots X_{n-1}$  à chaque fois que nous voulons calculer  $X_n$ . Afin de pouvoir comparer les deux algorithmes, nous produisons les mêmes nombres aléatoires avec les différents algorithmes.

#### Scénario 1: Utilisation d'un Registre

Dans la figure suivante est représenté le temps d'exécution des différents algorithmes selon le nombre de nombres aléatoires que nous voulons obtenir.

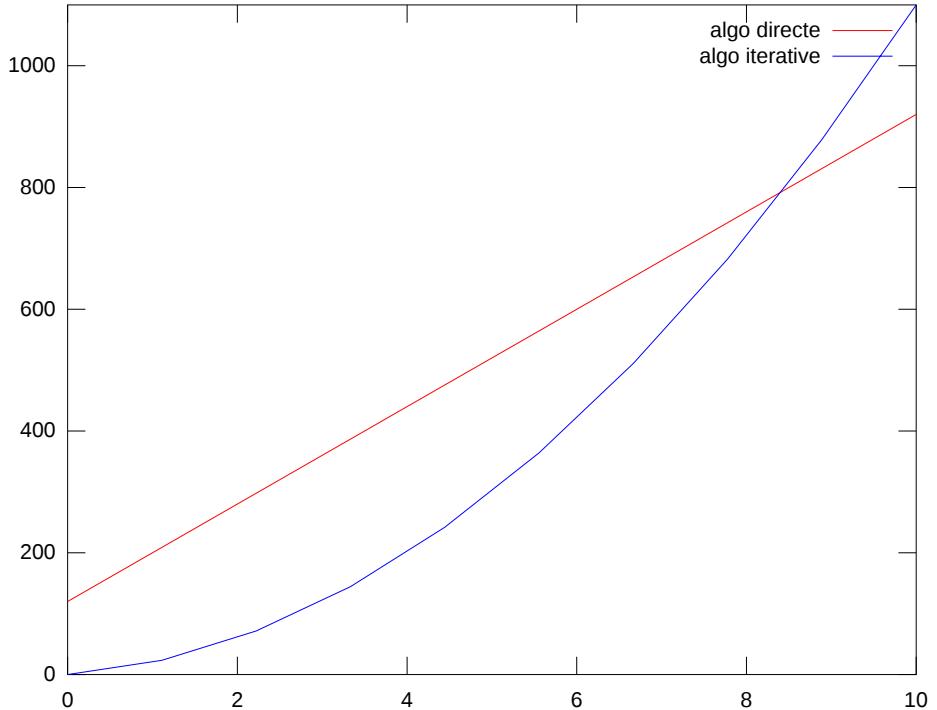
Nous pouvons constater que l'algorithme itératif est vraiment beaucoup plus rapide. En effet étant ne faisant qu'une simple multiplication il est rapide.



### Scénario 2: On ne garde pas $X_{n-1}$

Dans la figure suivante nous comparons, l'algorithme itératif sans registre avec l'algorithme direct, en terme de temps d'exécution.

Nous constatons qu'après avoir obtenu 8 nombres aléatoires l'algorithme direct est plus rapide que l'algorithme itératif. (voir aussi conclusions)



# 4

## Problèmes et Solutions Proposées

Après avoir eu un apperçu des ressources utilisées par les différents algorithmes, nous analysons les résultats données par ceux-ci. J'ai pu constater deux problèmes:

- Parité
- Répétition

### 4.1 Problèmes

#### 4.1.1 Parité

Notre premier problème est la parité. Pour calculer les nombres après chaque calcul on effectue une opération modulo avec  $2^p$  et  $p$  indique le nombre de bits qu'on utilise. Voilà notre formule:

$$X_n = (X_{n-1} * A) \bmod 2^p$$

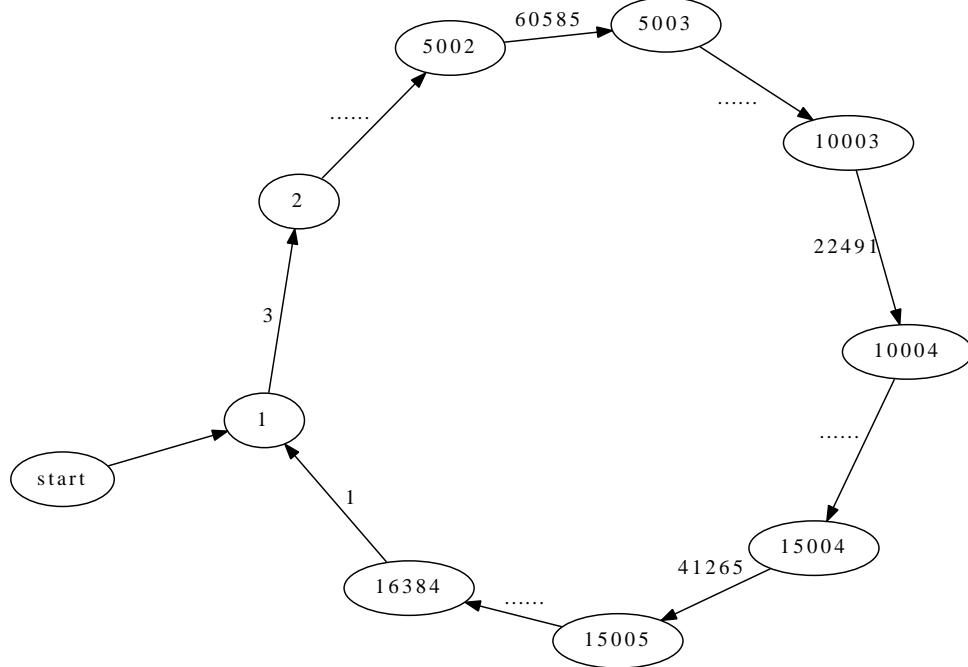
Si  $X_1$  est pair tous les nombres qu'on va générer après seront pairs aussi, car  $(\text{pair} * (\text{pair ou impair}) = \text{pair})$  et  $(\text{pair mod pair} = \text{pair})$ . Réciproquement si  $X_1$  est impair tous les nombres générés seront impair.

#### 4.1.2 Répétition

Tout générateur qui fonctionne avec un algorithme, la répétition est indispensable. Avec notre algorithme au plus on peut générer  $2^p$  nombres différents et après le générateur commence à répéter les résultats. (Facile à démontrer avec démonstrations par absurd)

Pour voir quand le générateur commence à répéter les nombres générés j'ai l'implémenté avec langage C et j'ai regardé les résultats. Par conséquent d'après le A et  $X_0$  le générateur commence à répéter après un Nième nombre pour  $0 < N < 2^p$ . Dans le graphe suivant vous voyez les quelques résultats pour A

$= 3$  et  $X_0 = 1$ . Après le 16384<sup>ème</sup> nombre il commence à répéter.



## 4.2 Idées pour les problèmes

A la suite vous trouverez 3 idées que j'ai trouvé comme solution possible au problème de parité:

### 4.2.1 Idée 1 : Modulo avec le nombre premier plus proche

D'abord j'ai pensé qu'au lieu de effectuer un modulo avec  $2^p$  je peux utiliser un nombre impair. Pour voir les nombres et les analyser j'ai utilisé C et au lieu de mod  $2^p$  j'ai utilisé mod  $2^p-1$ . Mais même si le générateur a commencé de générer les nombres pairs et impairs le résultat devient vite 0 et si une fois le résultat est 0 tous les nombres suivants deviennent 0. Donc j'ai décidé d'utiliser le nombre premier qui est plus proche à  $2^p$ . Parce que si on utilise un nombre premier le résultat ne devient jamais 0 car on n'utilise que la multiplication. Après pour voir les nombres comme précédemment, j'ai utilisé C et vérifier que le générateur génère les nombres pairs et impairs et ça ne devient jamais 0.

Enfin j'ai commencé à regarder les opérateurs VHDL pour l'opération modulo; et j'ai réalisé que l'opérateur mod en VHDL n'est effectué qu'avec des nombres qui sont puissances de 2. Donc cette solution n'est pas utilisable sur un FPGA. On peut créer un autre composant pour effectuer ce modulo mais ce ne sera pas efficace; ça va ralentir le générateur.

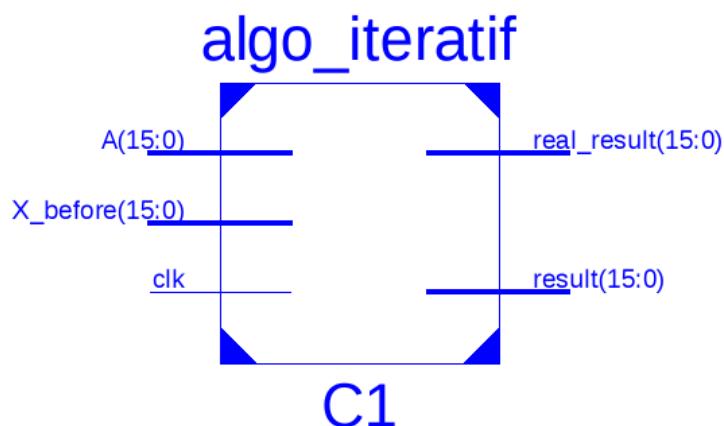
#### 4.2.2 Idée 2 : Décalage circulaire

Pour résoudre le problème de parité j'ai pensé à utiliser un décalage circulaire à droite. Grâce à cette méthode si le bit plus forte égale 0 le bit le moins fort devient 0 donc notre nombre est pair sinon impair. Mais cette méthode a une handicap si on génère des nombres impairs le bit plus fort est toujours 1 et si on génère des nombres pairs le bit plus fort est toujours 0. Donc on perd de nombreux de nombres avec cette méthode.

#### 4.2.3 Idée 3 : Permutation circulaire

Au lieu de faire un décalage circulaire, on peut changer seulement la valeur du premier bit avec une autre bit. Normalement les composants calculent des nombres sur 32 bit et prendre leur premier 16 bits à la sortie. À ce point on change le premier bit avec un autre bit et pour ça j'ai utilisé le 17ième bit car il n'est pas inclus dans notre résultat. Donc si 17ième bit est 0 notre nombre est pair sinon impair. Pour ce changement bien sur on peut utiliser un autre bit.

Ensuite si on utilise cette méthode on doit faire des petits changements dans le composant. Avec cette solution à la sortie du composant on aura 2 vecteurs. L'un est le résultat qu'on a obtenu en utilisant cette méthode et la deuxième est le vrai résultat qu'on va utiliser pour notre  $X_{n-1}$ .



Result est le résultat modifié qu'on a obtenu en utilisant cette méthode et  $\text{real\_result}$  est le nombre généré sans modification.

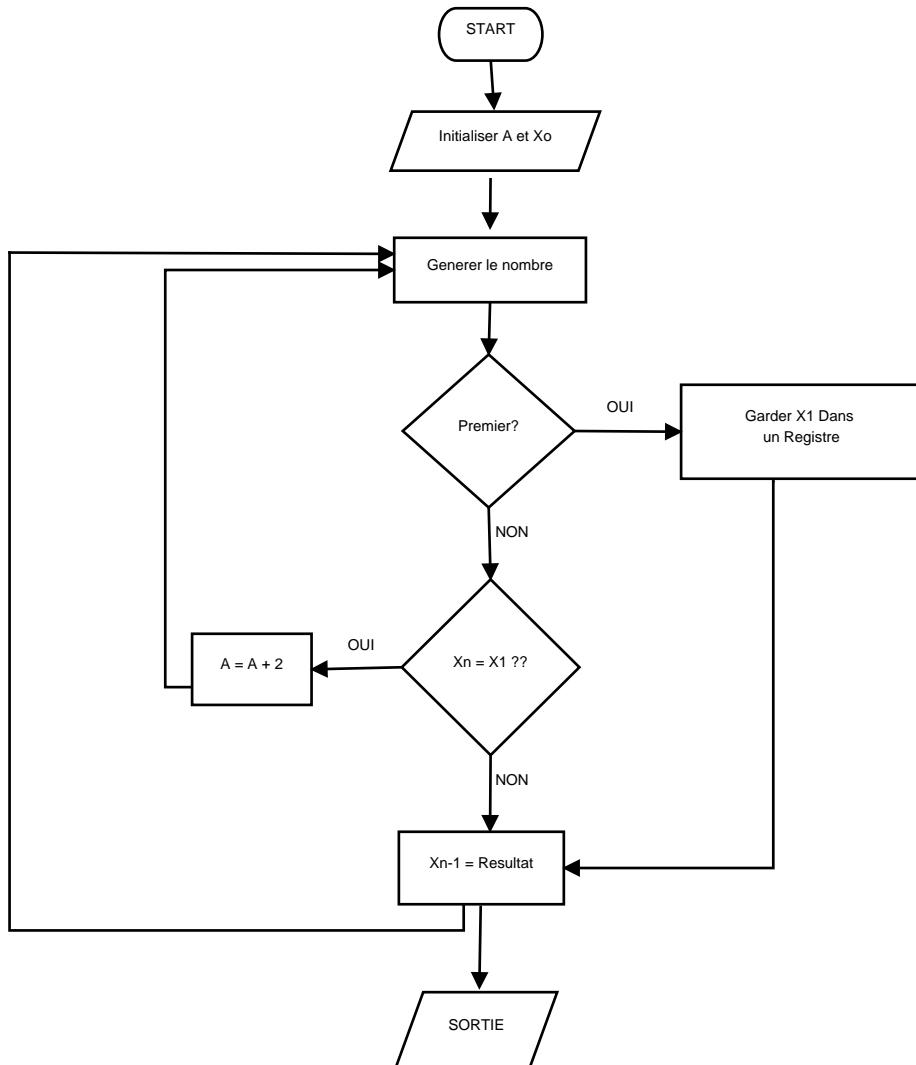
Dans la suite de document j'utilise cette méthode.

#### 4.2.4 Répétition

Comme on avait dit au début, la répétition est indispensable. On peut générer au maximum  $2^p$  nombres différents après les nombres se répètent périodiquement.

Pour éviter la répétition j'ai décidé d'implémenter un autre composant qui garde le premier nombre généré dans un registre et après chaque itération ce composant contrôle si le nombre généré est égal au premier nombre ou pas et si c'est égal le composant augmente la valeur de la constante A. Le premier nombre devient  $X_0$  et enfin le registre est modifié avec le nouveau nombre généré. Grâce à ça j'ai pu générer tous les nombres et j'ai évité les répétitions.

Le principe du fonctionnement de ce composant est exprimé dans la figure suivante :



J'ai choisi d'utiliser des nombres impairs pour mes tests; car grâce à des nombres impairs le nombre généré ne peut pas être 0 et c'est pourquoi j'augmente A avec +2. Parce que si le résultat est égal à 0 alors on n'aura plus que des 0.

# 5

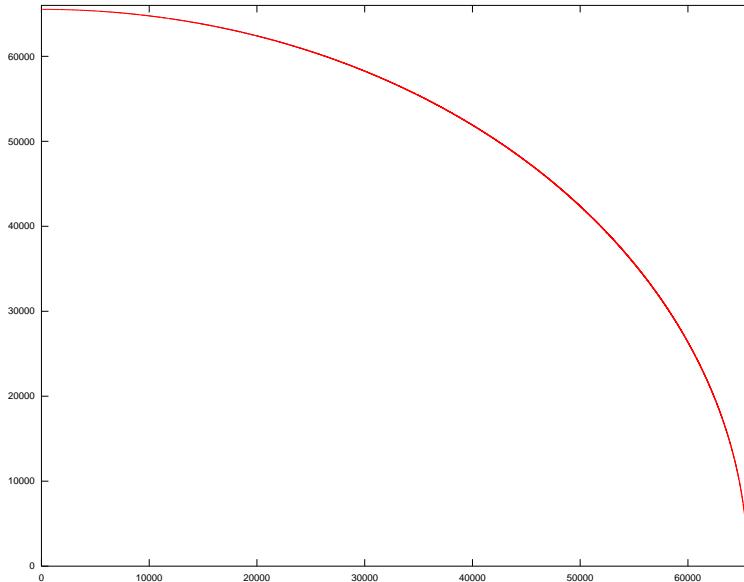
## Méthode de Monte Carlo

Cette Méthode a été inventé en 1947 par Nicholas Metropolis. Méthode Monte Carlo, désigne toute méthode visant à calculer une valeur numérique en utilisant des procédés aléatoires et ces méthodes sont particulièrement utilisées pour calculer des intégrales en dimensions plus grandes que 1.

Pour vérifier que notre générateur de nombre aléatoire on va déterminer la valeur de  $\pi$  avec la méthode Monte Carlo.

### 5.1 Détermination de la valeur de $\pi$

On veut faire un rapprochement à la valeur du  $\pi$  grâce à la Méthode Monte Carlo sur un cercle, et ici on va le determiner sur un quart de cercle.



On ne génère que des nombres positifs donc on va déterminer la valeur de  $\pi$  sur un quart de cercle.

Notre générateur fait ces opérations sur  $p$  bit donc au maximum on peut générer  $2^p - 1$ . Pour les tests on utilise 16 bit alors au maximum on peut générer 65535; c'est à dire le rayon du cercle sera 65535 ( $R=65535$ ). C'est à dire la surface générale sera :

$$Surface = R * R = 65535 * 65535 = 65535^2$$

La surface de quart cercle :

$$SurfaceCercle = \frac{\pi * R^2}{4}$$

Et si on prend la proportion de ces 2 surfaces :

$$\frac{SurfaceCercle}{Surface} = \frac{\frac{\pi * R^2}{4}}{R^2} = \frac{\pi}{4}$$

Donc la proportion entre la surface du cercle et surface normale est  $\frac{\pi}{4}$ . Alors pour déterminer  $\pi$  d'abord on va générer des points aléatoires et après chaque fois on va augmenter la variable  $N$  qui compte le nombre total des points générés et à la suite on va tester si le point est dans le cercle ou pas s'il est dedans on va augmenter la variable  $DEDANS$  qui compte le nombre des points qui sont dans le cercle. Enfin après  $K$  itérations la proportion doit être égale à  $\frac{DEDANS}{N} = \frac{\pi}{4}$ . Alors :

$$\frac{DEDANS}{N} = \frac{\pi}{4}$$

donc :

$$\pi = 4 * \frac{DEDANS}{N}$$

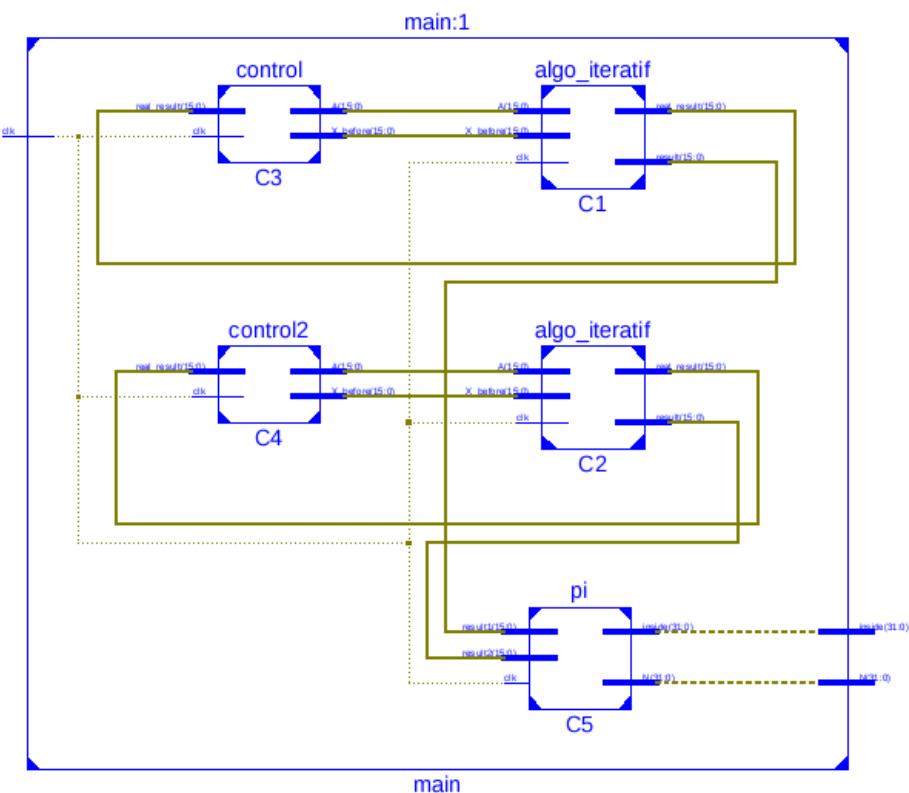
C'est à dire pour déterminer la valeur de  $\pi$  :

- $x$  = générer un nombre
- $y$  = générer un nombre
- chaque fois augmente  $N$
- si  $x^2 + y^2 \leq R^2$  augmente  $DEDANS$
- repeter  $K$  fois
- enfin calculer  $4 * \frac{DEDANS}{N}$

Si la valeur est proche à 3.14 notre générateur est passé le test avec succès.

## 5.2 Implémentation

Pour générer des nombres j'ai utilisé la méthode de permutation de bit présenté dans le chapitre précédent avec un composant de contrôle également présenté dans le chapitre précédent pour éviter les répétitions.

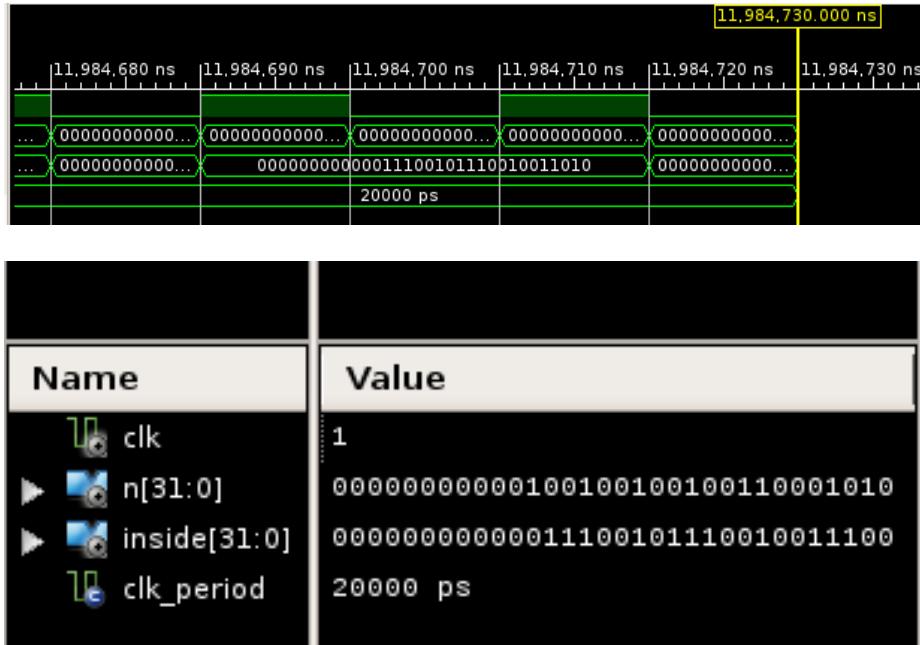


- C1 est le composant qui génère x
  - C2 est le composant qui génère y
  - C3 est le composant pour controller repetition du composant C1
  - C4 est le composant pour controller repetition du composant C2
  - C5 est le composant qui prende x et y en entrée compte des points générés et si  $x^2 + y^2 \leq R^2$  il augmente la variable inside
  - C5 a 2 sorties inside qui est DEDANS et N qui est le nombre total des points générés.

Avec VHDL sans utiliser un microprocesseur on ne peut pas calculer des nombres flottant donc pour pouvoir faire le test Monte Carlo les composants C5 nous donne 2 valeurs; N et DEDANS à la sorties et avec ces valeurs on peut calculer  $\pi$  et voir si notre générateur passe le test ou pas.

### 5.2.1 Exécution

Voilà ci dessous les résultats d'une des simulations.



Voila notre N vaut 100100100100110001001 et DEDANS vaut 11100101110010011011 en base binaire. Si on les transforme à base decimale :

$$N = 100100100100110001001 = 1198473$$

$$DEDANS = 11100101110010011011 = 941211$$

C'est à dire on a généré 1198473 points et 941211 de ces points sont dans le cercle. Alors

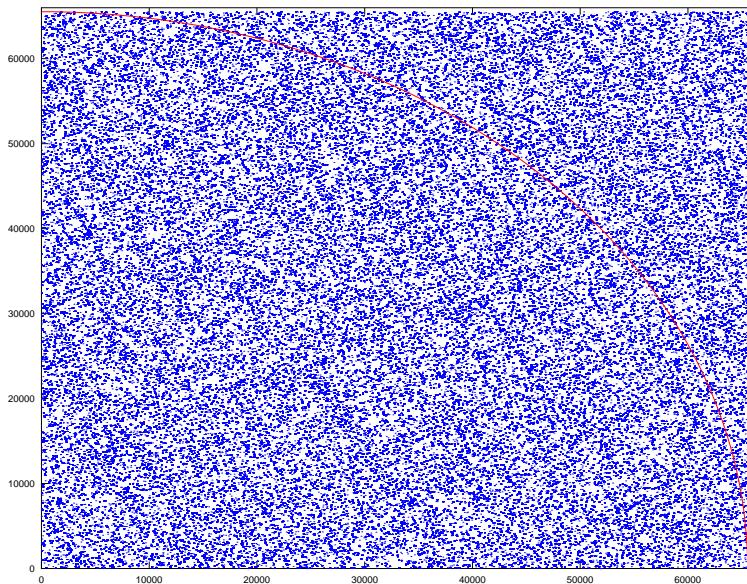
$$\pi = 4 * \frac{DEDANS}{N}$$

$$\pi = 4 * \frac{1198473}{941211} = 3.1413673900037797$$

Par conséquent on a déterminé que  $\pi$  est environ égale à 3.14. C'est à dire notre générateur a passé le test de la génération des nombres aléatoire avec succès.

### 5.2.2 Points sur un Graphe

On a vu que notre système bien produit des nombres aléatoires car il a passé le test Monte Carlo. Mais un générateur qui répète périodiquement peut aussi passer ce test. Alors pour voir si le système repete ou pas et vérifier que tous les nombres sont généré j'ai implementé l'algorithme avec les modifications et le composant contrôl avec langage C avec les même constantes A et  $X_0$ . Après j'ai sauvegard les points générés dans un fichier. Ensuite j'ai crée le graphe de ces points :



Nous pouvons constater que tout les points générés sont équitablement répartis sur le graphe; ce qui nous redémontre l'efficacité de notre générateur de nombres aléatoire.

# 6

# Conclusions et Perspectives

## 6.1 Conclusions

Dans ce document deux algorithmes pour la génération de nombres aléatoires a été présenté, puis implémenté en VHDL pour enfin être testé sur la carte FPGA(Digilent Nexys2).

- Le premier algorithme est l'algorithme "itératif"

$$X_n = (X_{n-1} * A) \bmod 2^p$$

- Le second algorithme est l'algorithme "direct"

$$X_N = \left( X_0 * \left( \prod_{0 \leq i < p} A^{n_i * 2^i} \right) \right) \bmod 2^p$$

Ces deux algorithmes produisent les même nombres aléatoires dès lors qu'on utilise la même constante A ainsi que la même X<sub>0</sub>

La seule différence est l'accès au X<sub>n<sup>ième</sup>-1</sub> alors qu'avec le second algorithme nous pouvons y accéder directement. Mais cette accès direct n'est pas gratuit, il couté cher ressources matérielles ainsi qu'en temps d'exécution.

### 6.1.1 Ressources Utilisés

L'algorithme direct est plus compliqué que l'algorithme itératif donc il a bessoin plus de ressources que l'algorithme itératif.

Resource	Disponible	Composante Itérative	Composante Direct
Slices	8672	0	320
LUTs	17344	0	619
IOBs	250	49	65
BUFGMUX	24	1	1
MULTT18	28	1	28

### 6.1.2 Temps d'Exécution

Dans le tableau ci-dessous, les temps d'exécutions et d'initialisations des deux algorithmes pour générer un nombre avec l'horlage de 20 ns.

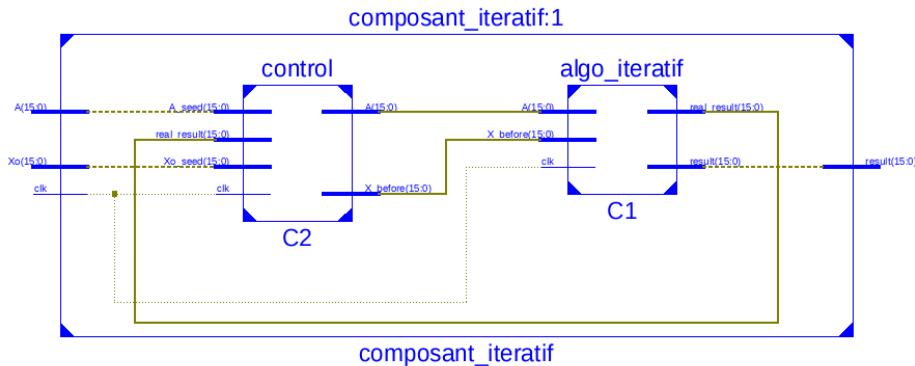
Algorithme	Initialisation	Execution	
Itératif	0	T	T = 20 ns
Direct	5T	4T	

Comme nous pouvons le constater l'algorithme direct est beaucoup plus pour générer un nombre aléatoire.

### 6.1.3 Problèmes et Solutions

En analysant les résultats de notre générateur de nombres aléatoires nous avons pu constater 2 handicaps dans nos algorithmes; le premier étant un problème de parité. En effet si notre premier nombre généré est pair tout les nombres générés seront pairs et inversement avec les nombres impairs. Le second problème est un problème de répétition. Après avoir générer un certains nombre de nombres aléatoires notre générateur regénère les même nombres. Je propose donc des solutions pour résoudre ces deux problèmes.

Pour le problème de parité j'ai utilisé la solution de permutation de bit, et pour le problème de répétition j'ai crée un composant de contrôle. Ce qui donne pour le composant itératif.



### 6.1.4 Méthode Monte Carlo

En utilisant les solutions, on a tester notre générateur avec l'une des méthodes de Monte Carlo qui est la détermination de la valeur de  $\pi$  et notre générateur de nombres aléatoires passe ce test avec succès.

## 6.2 Perspectives

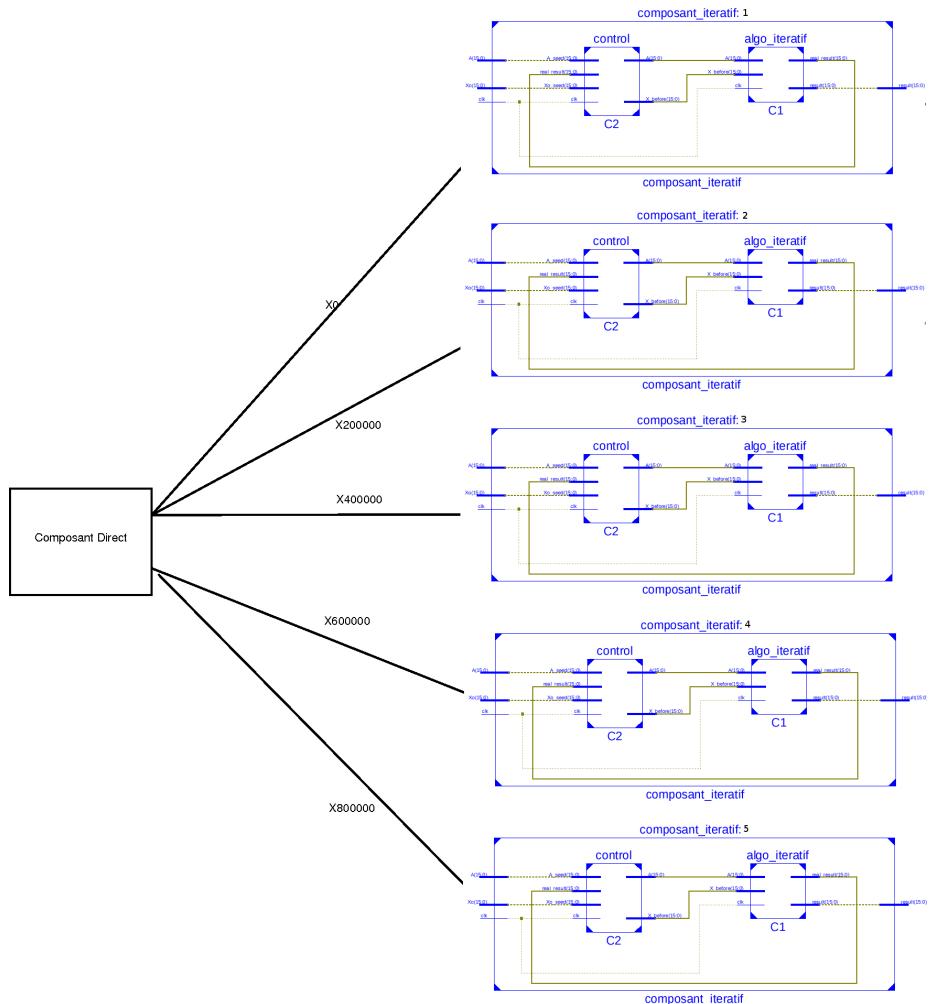
Même si le composant direct est plus lent et consomme plus de ressources matérielles pourquoi veut on quan même utiliser?

Par exemple l'échantillonnage a vraiment besoin des nombres aléatoires. C'est une notion importante en météologie : lorsqu'on ne peut pas saisir un événement dans son ensemble, il faut effectuer des mesures en nombre fini, afin de représenter l'événement. Afin de trouver cet ensemble on a besoin de nombreux nombres aléatoires.

Par exemple pour un travail de statistiques si nous avons besoin de 1 000 000 nombre aléatoires. Si on génère ces nombres itérativement avec le composant itératif ça va nous coûter:

$$TempsExecution = 20 * 1000000 = 20000000 = 20ms$$

Donc on a besoin de 20 milisecondes. Mais si on crée un système mixte comme le suivant nous pouvons gagner du temps:



Le principe est simple d'abord le composant direct crée  $X_0$  pour le composant itératif - 1,  $X_{200000}$  pour le composant itératif - 2, ,  $X_{400000}$  pour le composant itératif - 3, ,  $X_{600000}$  pour le composant itératif - 4 et ,  $X_{800000}$  pour le composant

itératif - 5 après les composants itératifs commence à générer la série des nombres. Chaque composant itératif génère 200000 nombres aléatoires en même temps. C'est à dire:

$$\begin{aligned} \text{Temps d'Exécution} &= \text{Temps d'Exécution Direct} + \text{Temps d'Exécution Itératif} \\ \text{TE} &= \text{Temps d'Exécution} \end{aligned}$$

$$TE = 5T + 5 * 4T + 200000T$$

$$T=20\text{ns}$$

$$TE = 5 * 20 + 5 * 4 * 20 + 200000 * 20 = 4000500\text{ns} = 4.0005\text{ms}$$

Donc temps d'exécution est environ 4 milisecondes donc notre système est devenu 5 fois plus rapide que le précédent.

Grâce au composant direct on a gagner du temps et un système plus performant en utilisant peu de ressources. En effet l'algorithme itératif qui est répété ne consomme que très peu de ressources alors que le composant direct en consomme beaucoup plus, mais celui là n'est présent qu'une seule fois dans le système.

En utilisant le système mixte, l'algorithme direct devient la phase d'initialisation de l'algorithme itératif.

# Appendix A

## Codes

### A.1 Algorithme Iteratif

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity algo_iterative is
    port (clk:in STD_LOGIC;
          A: in STD_LOGIC_VECTOR(15 downto 0);
          X_before: in STD_LOGIC_VECTOR(15 downto 0);
          result: out STD_LOGIC_VECTOR(15 downto 0));
end algo_iterative;

architecture Behavioral of algo_iterative is
signal rst: STD_LOGIC_VECTOR(31 downto 0);
begin
    process(clk)
    begin
        if clk = '1' and clk'Event then
            rst<=X_before*A;
        end if;
    end process;
    result<=rst(15 downto 0);
end Behavioral;
```

## A.2 Algorithme Direct

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity algo_direct is
    Port( clk: in STD_LOGIC;
           result:out STD_LOGIC_VECTOR(15 downto 0);
           A: in STD_LOGIC_VECTOR(15 downto 0);
           Xo:in STD_LOGIC_VECTOR(15 downto 0);
           N:in STD_LOGIC_VECTOR(15 downto 0));
end algo_direct;

architecture Behavioral of algo_direct is
TYPE dim IS ARRAY (15 downto 0) OF STD_LOGIC_VECTOR(31 downto 0);
signal d: dim;
begin
    --i am calculating the square powers of A
    d(0)(15 downto 0)<=A;
    p:for i in 1 to 15 generate
        d(i)<=d(i-1)(15 downto 0)*d(i-1)(15 downto 0);
    end generate p;

    process(clk)
    variable tmp : std_logic_vector(31 downto 0);
    begin
        if clk = '1' and clk'Event then
            tmp(15 downto 0):=Xo;
            for i in 0 to 15 loop
                if N(i) = '1' then
                    tmp:=d(i)(15 downto 0) * tmp(15 downto 0);
                end if;
            end loop;
            result<=tmp(15 downto 0);
        end if;
    end process;
end Behavioral;

```

# Appendix B

## Temps D'Exécutions

### B.1 Algorithme Itérative

X <sub>n-1</sub>	A	Temps D'exécution(ps)
0	0	5487
0	1	5487
1	0	5487
0	1	5487
1	1	17727
1	3	19411
1	15	19411
1	255	19411
1	2 <sup>12</sup> - 1	19411
2	2 <sup>16</sup> - 1	19411
4	2 <sup>16</sup> - 1	18758
2 <sup>6</sup>	2 <sup>16</sup> - 1	18758
2 <sup>16</sup> - 1	2 <sup>16</sup> - 1	17727

## B.2 Algorithme Direct

N	Temps D'exécution(ns)	N	Temps D'exécution(ns)
1	115.927	17501	135.927
2	115.897	18501	135.927
3	115.927	19501	135.927
4	115.922	20501	135.927
5	115.927	21501	135.927
10	115.922	22501	135.927
100	115.922	23501	155.789
200	115.922	24501	155.927
500	115.912	25501	135.927
501	115.927	26501	135.927
1001	115.927	27501	135.927
1501	115.927	28501	135.927
2001	115.927	29501	155.789
2501	115.927	30501	155.789
3001	115.927	31501	155.927
3501	115.927	32501	155.927
4001	115.927	33501	135.927
4501	135.789	34501	135.927
5001	135.789	35501	135.927
5501	135.789	36501	135.927
6001	135.789	37501	135.927
6501	135.927	38501	135.927
9501	115.927	46001	135.927
10001	115.927	47501	135.927
10501	115.927	49001	155.789
11001	115.927	50501	135.927
11501	115.927	52001	135.927
12001	135.789	53501	155.791
12501	135.927	55001	155.791
13001	135.927	56501	155.927
13501	135.927	58001	155.789
14001	135.927	59501	155.789
14501	135.927	61001	155.789
15001	135.927	62501	155.927
15501	135.927	64001	155.927
16001	135.927	65501	175.789
16501	135.927		

## Appendix C

# Algorithme Itératif avec permutation

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity algo_iteratif is
    port (clk:in STD_LOGIC;
          A: in STD_LOGIC_VECTOR(15 downto 0);
          X_before: in STD_LOGIC_VECTOR(15 downto 0);
          real_result: out STD_LOGIC_VECTOR(15 downto 0);
          result: out STD_LOGIC_VECTOR(15 downto 0));
end algo_iteratif;

architecture Behavioral of algo_iteratif is
signal rst: STD_LOGIC_VECTOR(31 downto 0);
begin
    process(clk)
    begin
        if clk = '1' and clk'Event then
            rst<=X_before*A;
        end if;
    end process;
    result(15 downto 1)<=rst(15 downto 1);
    result(0)<=rst(6);
    real_result<=rst(15 downto 0);
end Behavioral;
```