h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Génération de vrais nombres aléatoires sur système embarqué

Thèse de Bachelor présentée par

Loïc Piccot

pour l'obtention du titre Bachelor of Science HES-SO en

Microtechniques
Spécialité : Conception électronique

Juin 2019

Professeur-e HES responsable

Andres Upegui

Source de l'illustration de couverture : *https ://www.networkworld.com/article/3336269/build-security-into-your-iot-plan-or-risk-attack.html*

TABLE DES MATIÈRES

A mes parents qui m'ont toujours soutenu

REMERCIEMENTS

Je remercie le Dr. Andres Upegui pour le soutient et l'encadrement qu'il m'a apporté au cours de ce projet. Je remercie également, Laurent Gantel pour m'avoir fourni les codes logiciels et matériels de l'algorithme SPRG.

ÉNONCÉ DU SUJET

h e p i a
Haute école du paysage, d'ingénierie et d'architecture de Genève

Printemps 2019
Session de bachelor

MICRO-TECNIQUES – SYSTEMES NUMERIQUES – SUJET N° 60

SYSTEME EMBARQUE GENERATEUR DE NOMBRES ALEATOIRES QUANTIQUES

Descriptif :

La société ID Quantique propose la puce QRNG (Quantum Random Number Generator), la plus petite du monde, développée par SK Telecom. La puce QRNG exploite le véritable caractère aléatoire du coup bruit d'une source de lumière capturée par un capteur d'image CMOS.

Le but de ce projet de diplôme est de développer une solution basée sur une puce QRNG, une FPGA miniature de Lattice iCE40 et une interface USB afin de fournir de nombres aléatoires certifiés et de-biaisés sous la forme d'un système portable et compacte sous la forme d'une clé USB.

Travail demandé :

Le travail de l'étudiant consiste à :

- Prendre en main les outils et la méthodologie pour programmer une FPGA Lattice iCE40.
- Comprendre et mettre en place les éléments matériel et logiciel nécessaires pour établir une communication entre une FPGA et un port USB.
- Implémenter une architecture streaming capable de lire les nombres aléatoires depuis le QRNG, les traiter, et l'interface USB.
- Implémenter la vérification et le dé-biassage de nombres sur la FPGA.

Des travaux d'usinage sont-ils prévus ? ☐ OUI x NON

Si oui, quel est le temps estimé : ☐ 1 jour ☐ 3 jours ☐ >3 jours

Documents à rendre : voir le fichier TIN_Memo_directives_travail_de_bachelor.doc qui vous est envoyé par email.

Candidat :
PICCOT LOÏC
Filière : MT3
Option :

Professeur(s) responsable(s) :
UPEGUI Andres

En collaboration avec : **ID-Quantique**
Travail de bachelor soumis à une convention de stage en entreprise : non

Travail de bachelor soumis à un contrat de confidentialité : non

RÉSUMÉ

Depuis quelques années, l'internet des objets a effectué une entrée fracassante dans notre vie quotidienne. Selon l'EPFZ, 150 milliards d'objets seront connectés d'ici 2025. Il est donc plus que jamais, indispensable de se focaliser sur les problématiques de sécurité apportées par ce type de système.

L'essentiel de la sécurité informatique repose, de nos jours, sur la capacité du système à générer des clés de chiffrement complexes et imprévisibles et, par extension, sur la génération de nombres aléatoires. Il est donc primordial de réussir à créer des systèmes qui seront capables de générer rapidement des nombres aléatoires de qualité tout en s'adaptant aux différentes contraintes imposées par les systèmes embarqués. Ils devront notamment répondre à la problématique d'espace qui est omniprésente dans le domaine des objets connectés.

Ce rapport présente l'utilisation de la puce génératrice quantique de nombres aléatoires la plus petite du marché actuelle. Il tâchera d'expliciter les méthodes de fabrication d'une architecture FPGA destinée à la génération de vrais nombres aléatoires. Pour cela, il exposera le fonctionnement d'un générateur de nombres pseudo-aléatoires basé sur la construction en éponge de la fonction de hachage cryptographique Keccak.

Enfin, il évaluera la convenance de ce système en se basant sur des critères de débit, de qualité, de consommation énergétique et d'embarquabilité

Mots clés : Génération de vrais nombres aléatoires, Générateur quantique de nombres aléatoires, Bruit de grenaille optique, Fonction de hachage, Keccak, SHA-3, SPRG, PRNG, TRNG, QRNG.

TABLE DES ILLUSTRATIONS

INDEX DES TABLEAUX

LISTE DE ACRONYMES

CAN Convertisseur Analogique-Numérique. 30, 31, 33

FIFO First In First Out (Mémoire en file d'attente des données). 36, 46, 49

I2C Inter-Integrated Circuit. 30, 31

IoT Internet Des Objets. 19

LED Diodes Electroluminescentes. 30, 31

LUT Look Up Table(Table de correspondance). 44, 52, 53

PLL Phase-locked loop. 20

PRNG Générateur de Nombres Pseudo-Aléatoires. 23, 24, 41, 42

QRNG Générateur Quantique de Nombres Aléatoires. 19, 20, 25, 29, 35, 36, 49

RAM Random Access Memory (mémoire vive). 44, 52, 53

RDO Random number generation Data Output. 32, 33, 36, 38, 49, 51, 57, 59

RNG Générateur de Nombres Aléatoires. 23, 24

SDO Sample Data Output. 32, 33, 36, 38, 49, 51, 54, 56, 59

SPI Serial Peripheral Interface. 19, 30, 31, 35, 51

TRNG Générateurs de Véritables Nombres Aléatoires. 23, 24, 41

INTRODUCTION

1. MOTIVATION À LA RÉALISATION DU PROJET

Depuis quelques années, l'Internet Des Objets (IoT) prend une place de plus en plus importante dans notre société [4]. Il se définit par l'interconnexion massive d'équipements électroniques tels que des smartphones, des capteurs ou des actionneurs qui ont la particularité d'être reliés à internet. Cependant, cet essor amène son lot de failles de sécurité ainsi qu'une aggravation de leurs conséquences. En effet, l'existence d'une brèche dans n'importe lequel de ces équipements peut compromettre l'ensemble du réseau.

Il est important ici de souligner que la nature même de certains objets connectés peut impliquer des contraintes auxquelles les systèmes de sécurité existants ne peuvent pas répondre dans leur état actuel. Par exemple, les problématiques liées à la consommation électrique, à l'espace disponible et à la faible disponibilité de ressources informatiques.Il devient donc crucial d'améliorer les technologies permettant la sécurisation informatique de l'ensemble de ces équipements.

En cryptographie, les clés de chiffrement permettent de modifier un message de sorte qu'il devienne incompréhensible par une autre personne que celui qui l'a créé [5]. L'utilisation de cet outil permet de sécuriser les échanges de données et d'informations.

En 1883, Auguste Kerckhoffs déclare, dans un article dédié à la cryptographie militaire, que l'ensemble du secret d'un cryptosystème doit être contenu dans la clé. Les algorithmes et autres méthodes utilisées pour le cryptage doivent pouvoir être divulgués [6]. Encore aujourd'hui, toute la sécurité informatique repose sur ce principe. Il est évident que dans la plupart des cas, la clé est basé sur une séquence de nombres aléatoires. En conséquence, il est primordial de réussir à maintenir les technologies concernant la génération de nombres aléatoires au plus haut niveau dans le but de résister aux attaques visant à décrypter la valeur des clés de chiffrement. Il va de soi que la qualité [1] ainsi que le débit de la génération des nombres aléatoires jouent un rôle crucial dans la performance de la sécurisation qui sera apportée au système.

C'est ainsi que naît la motivation à la réalisation du projet de "génération de vrais nombres aléatoires sur système embarqué".

1. La notion de qualité d'une suite aléatoire est définie dans la section 1 du chapitre 1

2. CONTEXTE ET OBJECTIFS

Ce travail se présente comme la suite logique du projet de même intitulé [7] dans lequel avait été développé un circuit imprimé permettant d'interfacer un Générateur Quantique de Nombres Aléatoires (QRNG) avec un circuit logique ou un micro-contrôleur par l'intermédiaire d'une communication Serial Peripheral Interface (SPI). L'utilisation d'un microcontrôleur avait permis de confirmer le bon fonctionnement du circuit imprimé ainsi que celui du générateur.

Dès lors, ce travail a pour objectifs, la réalisation d'une architecture FPGA. Permettant, dans un premier temps, l'extraction des nombres aléatoires générés par le QRNG. Puis, dans un second temps, l'ajout de l'algorithme de génération de nombres pseudo-aléatoires SPRG basé sur la fonction Keccak [2]. Cet algorithme sera utilisé dans un objectif de débiaisage de la sortie du QRNG. De plus, des modifications seront apportées à l'architecture de l'algorithme SPRG dans l'optique de le rendre modulable pour répondre aux contraintes de ressources imposées par l'utilisation de certains FPGA.

3. MATÉRIEL UTILISÉ

Pour la réalisation de ce projet, nous avons choisi de mettre en avant la notion d'embarquabilité et d'intégration du système dans un espace restreint en portant notamment notre choix sur le plus petit QRNG existant à ce jour appelé QT-Xr [3].

Cette volonté s'applique également au choix du FPGA. Pour celui-ci, nous avons opté pour l'utilisation de la famille iCE40 fabriqué par la société Lattice Semiconductor [8]. En particulier, nous utiliserons le iCE40UP5K [9] qui semble être un choix particulièrement adapté pour notre application, compte tenu de ses dimensions (2.15×2.55 mm dans sa version la plus compact). Il contient également un nombre extraordinairement élevé, pour un circuit de ces dimensions, d'éléments logiques (5280). De plus, il est peu consommateur de courant, puisque pour la plupart des applications, il arrive à maintenir une consommation inférieure à 10 mA.

Enfin, ce projet utilisera une carte de développement proposée par Lattice appelé "iCE40 UltraPlus Breakout Board" qui utilise le paquetage légèrement moins compact du UP5K (7×7 mm). La carte contient un port USB qui permet à la fois la programmation du FPGA mais également la mise en place d'une communication sériel "USB 2.0 High-Speed" (480 Mb/s) avec un ordinateur par l'intermédiaire d'un circuit intégré FTDI, le FT2232HL [10]. En outre, la carte

2. l'algorithme SPRG ainsi que la fonction Keccak sont présentés dans la section 3.2.3 du chapitre 2
3. le QT-Xr est présenté dans la section 2.1 du chapitre 2

propose de nombreuses pins d'entrée/sortie ainsi qu'un oscillateur (12 MHz). L'utilisation d'une Phase-locked loop (PLL), elle aussi disponible sur la carte, pourrait permettre d'augmenter cette fréquence dans le but d'accélérer les opérations. Toutefois, afin de ne pas augmenter la consommation du circuit, cette fonctionnalité ne sera pas exploitée au cours de ce projet.

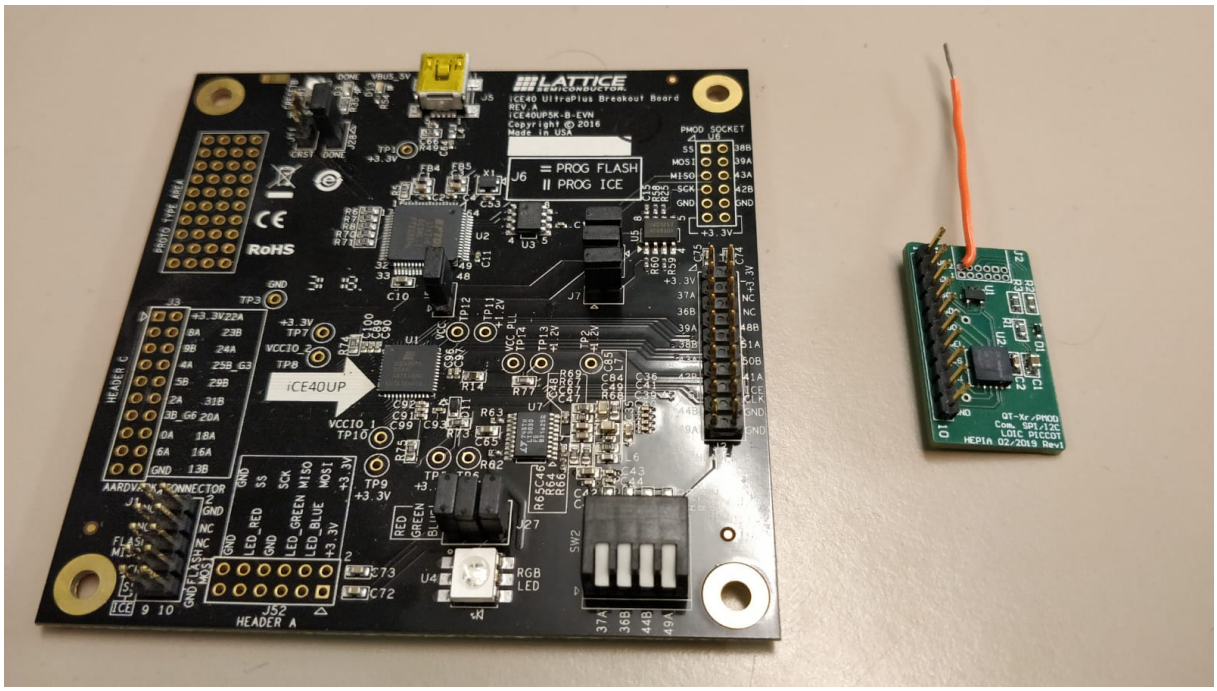


FIGURE 1 – Carte "iCE40 UltraPlus Breakout Board" (à gauche) et PCB du QT-Xr (à droite)

ETAT DE L'ART

Cette section comporte l'état de l'art et les notions théoriques qui seront nécessaires à la réalisation du projet.

1. SUITES ALÉATOIRES

Une suite aléatoire est une notion mathématique décrivant une suite de symboles dans laquelle il est impossible de déceler une régularité ou de trouver la moindre règle de prédiction des caractères suivants.[11][12]

La qualité d'une source d'aléa est quantifiée par la distribution statistique des séquences qu'elle produit, ainsi que par l'incapacité à déterminer le résultat du prochain tirage. Cette imprédictibilité est définie par la notion d'entropie.

La notions de biais, définit la tendance d'une suite à contenir une régularité ou une périodicité. Parmi les règles concernant la distribution statistique qu'une suite aléatoire doit respecter, on peut citer la plus triviale qui est l'hypothétique égalité de la proportion de chaque symbole. Par exemple, dans une suite contenant une grande quantité de nombres aléatoires binaires, la proportion de 0 et de 1 devrait théoriquement s'approcher de 50%.

2. LA GÉNÉRATION DE NOMBRES ALÉATOIRES

Comme énoncé en introduction, la génération de suites aléatoires est particulièrement décisive dans la création de clés de chiffrement et par extension dans la sécurisation d'un système d'information. Néanmoins, la génération de nombres aléatoires peut également être utile dans de nombreux autres domaines, tels que les simulations pour des travaux de recherches scientifiques et statistiques, le "machine learning", la génération de graines pour la blockchain ou encore les jeux de hasard [1]. Pour produire ces séquences, on utilise un dispositif appelé Générateur de Nombres Aléatoires (RNG)[13]. Ils sont séparés en deux catégories : les Générateur de Nombres Pseudo-Aléatoires (PRNG) et les Générateurs de Véritables Nombres Aléatoires (TRNG).

Le premier cas, est le plus courant car il est le plus facile à implémenter dans un processeur, en effet, le PRNG est basé sur l'utilisation d'un algorithme. Son implantation dans un système numérique qui a, par définition, des ressources limitées, le rend déterministe. Cela implique que ses séquences de sorties contiennent inévitablement des biais principalement représentés par

celui de la périodicité. Pour qu'un attaquant rencontre des difficultés à retrouver cette périodicité, et par conséquent les états futurs et passés de la séquence, il faut que celui-ci soit le plus complexe possible. Toutefois, l'expérience a démontré que pour être efficace, cette complexité doit être couplée à l'utilisation d'un état initial appelé graine (seed), qui doit être renouvelé régulièrement.

Pour la génération des graines, il est préférable d'utiliser des phénomènes liés au hasard. Parmi les PRNG existants, de nombreux sont basés sur des imperfections. On peut citer par exemple l'utilisation de la position de la souris de l'utilisateur d'un ordinateur, l'instant exact auquel a été fait la requête de génération ou encore le bruit d'une mesure (bits de poids faible d'un capteur très précis). L'ensemble de ces phénomènes se base sur un manque de performance (souvent apporté par l'humain) quant à leur perception de sorte qu'ils soient considérés comme liés au hasard. Pour illustrer cela, prenons l'exemple d'un tirage "pile ou face" d'une pièce de monnaie. Le caractère aléatoire de cette situation réside en réalité dans l'incapacité d'un homme à lancer deux fois la pièce de manière strictement identique. Or la physique affirme que si l'on connaît l'ensemble des conditions initiales du lancer de la pièce, il est possible de prédire avec certitude le résultat du tirage, ce qui le rend parfaitement déterministe. Ainsi, pour des raisons similaires, les phénomènes énoncés précédemment sont en réalité eux aussi déterministes.

Enfin ces méthodes de génération sont souvent plus susceptibles aux attaques extérieures. Par exemple, la simple augmentation de la température environnementale d'un générateur basé sur du bruit électronique pourrait insérer un biais dans la suite aléatoire.

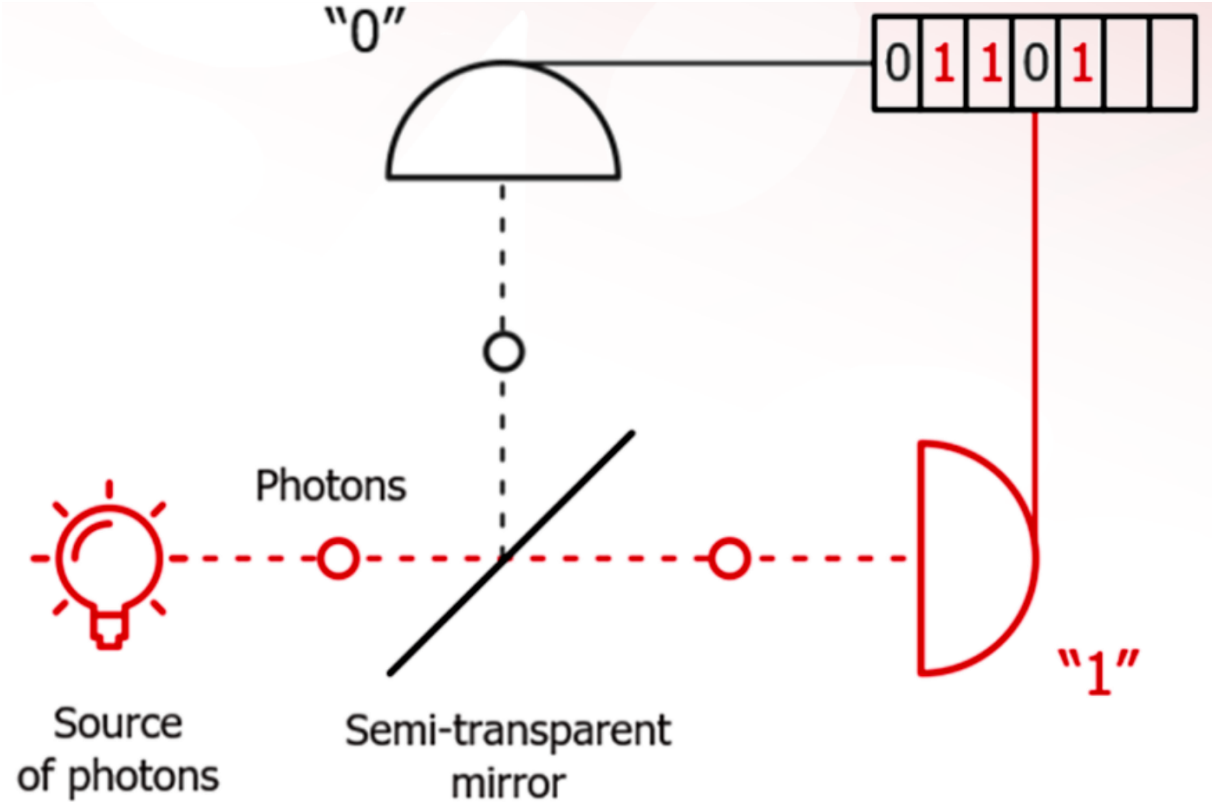
3. Les TRNG

Connaissant la vulnérabilité théorique des PRNG, il est évident qu'il devient important de trouver le moyen de réaliser des générateurs d'un autre type qui seraient alors basés sur des phénomènes physiques indéterministes. C'est en partant de ce postulat qu'on été développés les premiers générateurs de vrais nombres aléatoires.

La frontière entre les deux types de RNG est en réalité peu marquée puisque certains phénomènes physiques déterministes peuvent être estimés comme bien trop complexes pour entrer dans la première catégorie. Cette classe regroupe par exemple les générateurs à base d'anneaux asynchrones tels que celui de A. Cherkaoui [14] et le générateur basé sur l'agitation thermique autour de l'état métastable d'une bascule logique de J.-L. Danger *et al.* [15]. Certains autres TRNG tel que celui de M. Drutarovsky et P. Galajda[16] sont basés sur des phénomènes dit chaotiques, c'est à dire qu'une infime modification des conditions initiales entraîne une réac-

tion en chaîne débouchant sur un état finale totalement différent.

Une nouvelle fois, ces méthodes de génération peuvent être considérées comme insuffisantes pour le cas où l'attaquant possèderait une puissance de calcul exceptionnellement grande. Pour résoudre ce problème, des sociétés telles que ID Quantique[17], se sont attelées à augmenter la fiabilité de la génération en s'appuyant sur des phénomènes de la physique quantique qui sont fondamentalements aléatoires. En effet, de nombreuses études ont démontré que le comportement des particules au niveau subatomique était naturellement hasardeux. Ainsi, en utilisant le comportement aléatoire d'une particule quantique, il est possible de garantir un système véritablement impartial et imprévisible qu'on appelle QRNG. Parmi les phénomènes quantiques qui sont utilisés, il existe l'enregistrement de la présence ou non de la réflexion d'un photon sur un miroir semi-transparent (voir figure 1.1), la mesure de la position des photons après division d'un faisceau de lumière ou encore la mesure de la fluctuation de l'intensité lumineuse d'un faisceau. C'est cette dernière méthode qui va être utilisée dans le QRNG employé pour ce projet.



;

FIGURE 1.1 – Schéma représentant la génération quantique basé sur la réflexion d'un photon sur un miroir | Source : ID Quantique [1]

4. DÉBIAISAGE

L'histoire a montré que les suites aléatoires générées par une source d'entropie sont souvent entachées d'un biais. C'est à dire que leur distribution statistique n'est pas uniforme. C'est pour cela qu'ont été développé les premiers algorithmes de débiaisage destinés à améliorer la distribution statistique tout en gardant l'entropie de la source.

Voici quelques méthodes couramment utilisées pour débiaiser une source :

- Les portes OU-Exclusif : En effet, les combinaisons de portes XOR sont fréquemment utilisées pour effectuer des débiaisages car elles permettent de combiner deux bits d'entrée avec la même probabilité d'obtenir chacun des deux états en sortie. Exemple : le débiaisage LFSR (Registre à décalage à rétroaction linéaire) dont le fonctionnement est présenté dans la figure 1.2

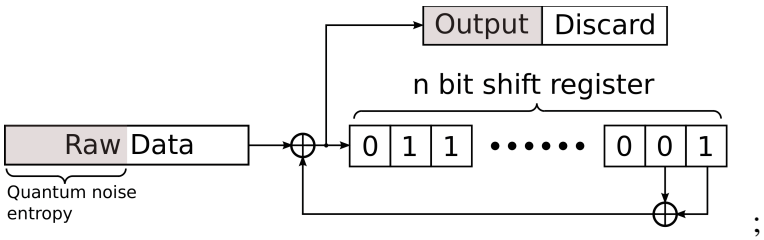


FIGURE 1.2 – Fonctionnement du débiaisage basé sur un LFSR | Source : Ultra-fast Quantum Random Number Generator [2]

- L'algorithme de Von Neuman : C'est l'un des premiers algorithmes de débiaisage, il est basé sur le schéma de Bernoulli. Son principe de fonctionnement est simple : les bits de la séquence d'entrée sont évalués deux à deux sans chevauchement, si les deux bits sont égaux, aucune valeur n'est transmise, dans le cas contraire, on fournit le premier bit (voir figure 1.3). Le principal défaut de cette algorithme est la diminution drastique de la taille de la séquence.

- Fonctions de hachage cryptographique : les fonctions de hachage sont présentées dans la section suivante. Leur grande utilisation pour des applications de débiaisage est due à leur capacité à ne pas diminuer le débit de la source et leur nature par définition extrêmement complexes à inverser et donc très sécuritaires.

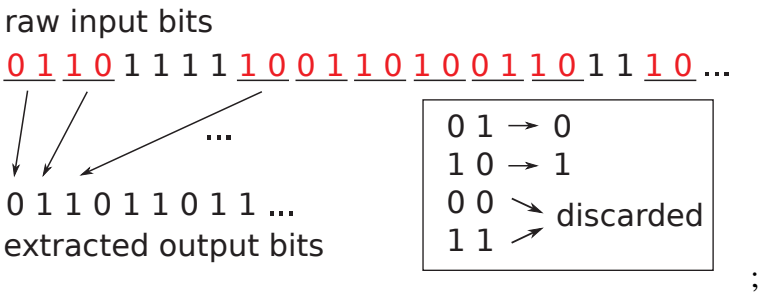


FIGURE 1.3 – Fonctionnement du débiaisage de Von Neuman | Source : Ultra-fast Quantum Random Number Generator [2]

5. FONCTIONS DE HACHAGE CRYPTOGRAPHIQUE

Pour la réalisation de ce projet, il est important de présenter les notions théoriques essentielles concernant les fonctions de hachage cryptographique. Cette famille désigne des fonctions qui, à une taille de message d'entrée variable, associe une sortie de taille fixe. Le résultat de la fonction est appelé valeur de hachage [18].

Un intérêt principal à l'utilisation de ce type de fonction est sa nature extrêmement complexe à inverser. C'est à dire qu'il est assez trivial de calculer la valeur de hachage d'un message mais qu'il est extrêmement compliqué de retrouver le message initial associé à une valeur de hachage. Pour qu'une fonction de hachage cryptographique soit idéale, il doit également être impossible de trouver deux messages différents ayant la même valeur de hachage. Les méthodes conçues pour prouver la faiblesse d'une fonction de hachage quant à ces règles, s'appellent les attaques de préimage :

1. L'attaquant tente de trouver le message d'entrée qui produira une valeur en sortie donnée (pour un y donné, il tente de trouver un x tel que $h(x) = y$).

2. L'attaquant tente de trouver deux messages d'entrée qui ont la même valeur de hachage (pour un x donné, il tente de trouver une deuxième préimage $x' \neq x$ tel que $h(x) = h(x')$.

On dit qu'une fonction de hachage cryptographique (H) est résistante aux collisions si elle est robuste à cette seconde attaque de préimage. On définit la limite supérieure de la résistance aux collisions grâce au paradoxe des anniversaires : pour une valeur de hachage de taille N, si il faut moins de $\sqrt{2^N}$ opérations de hachage à un attaquant pour trouver deux messages ayant la même valeur de hachage, la fonction de hachage ne peut pas être considérée comme fonction de hachage cryptographique. En conclusion, pour des applications en cryptographie, il faut que

la fonction de hachage soit suffisamment complexe et que la taille de la valeur de hachage soit suffisamment grande pour résister à une attaque des anniversaires.

ARCHITECTURE FPGA

1. PRÉSENTATION GÉNÉRALE

L'objectif principal de ce projet est l'implémentation d'une architecture FPGA en VHDL qui va permettre de récupérer et de traiter les nombres aléatoires générés par un QRNG. Cette section détaille le fonctionnement de l'architecture qui a été implémentée.

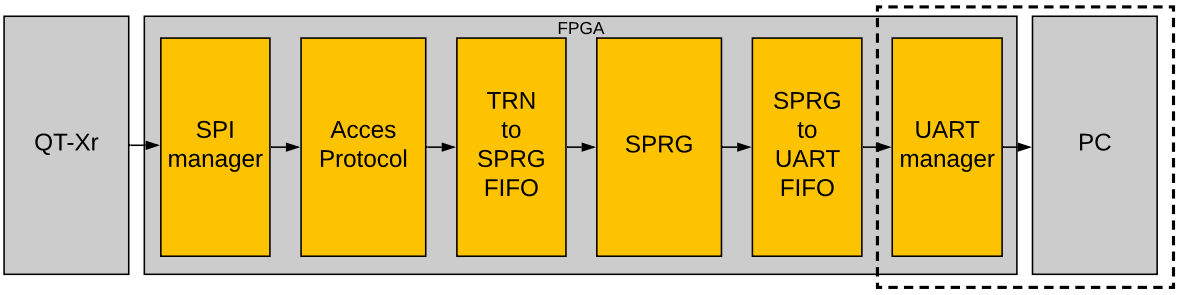


FIGURE 2.1 – Schéma général de l'architecture

2. LECTURE DES NOMBRES ALÉATOIRES

2.1. QT-Xr

Le QT-Xr est un QRNG développé par la société ID Quantique [1] en collaboration avec le groupe SK telecom [19]. Il est encore à l'heure actuelle en phase de développement, en conséquence, certaines informations de cette section sont susceptibles de changer à l'avenir.

2.1.1. Caractéristiques techniques

Avec des dimensions de l'ordre de 5 mm de côté (voir figure 2.2), le QT-Xr est actuellement le générateur quantique le plus compact du marché. De plus, il propose d'excellentes performances en terme de débit (1.5 Mb/s) et de consommation (89.3 mW max. lors de la génération des nombres aléatoires).

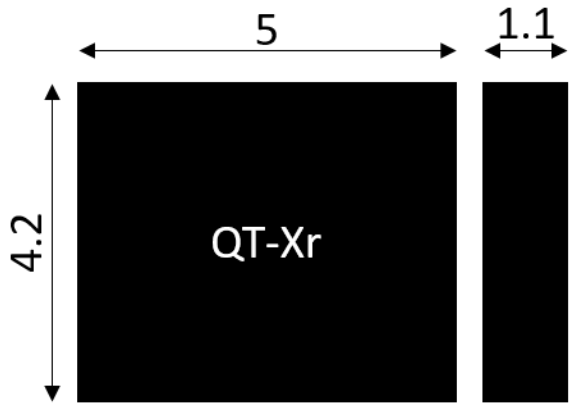


FIGURE 2.2 – Dimensions du QT-Xr

La configuration des registres internes ainsi que la lecture des données de la puce sont assurées par l'intermédiaire de deux protocoles de communication : le SPI et le Inter-Integrated Circuit (I2C) qui proposent respectivement une fréquence maximale de 24MHz et de 100 kHz. Enfin, il a été conçu pour avoir une bonne résistance aux perturbations environnementales et peut fonctionner dans une plage de température garantie de -30 à 85 °C.

Le prix de ce générateur n'a pas été divulgué à ce jour mais sera assurément abordable puisque sa vente est destinée à un très large public. Toutes ces conditions font de ce générateur un candidat parfait pour la génération de nombres aléatoires dans un système embarqué.

2.1.2. Génération des nombres aléatoires dans le QT-Xr

Pour des raisons évidentes de conservation de la propriété intellectuelle, certaines informations concernant la construction et le fonctionnement de la puce ne sont pas fournies dans la documentation [3]. Toutefois, ce rapport tâchera d'expliciter l'ensemble des informations nécessaires à son utilisation.

La génération des nombres aléatoires est basée sur l'utilisation du phénomène optique du bruit de grenaille. Lorsque l'on fixe le temps d'acquisition d'un capteur optique exposé à une source lumineuse, la somme des probabilités d'obtenir un photon durant un temps infiniment petit suit une distribution de Poisson. En d'autres termes, c'est la fluctuation de l'intensité lumineuse de la source qui va être capturée et utilisée pour la génération des nombres aléatoires. Plus l'intensité lumineuse est élevée, plus l'entropie sera bonne.

Dans le QT-Xr l'émission lumineuse est réalisée par des Diodes Electroluminescentes

(LED) tandis que l'acquisition est réalisée par une grille de (128×100) capteurs CMOS. Chaque capteur va transmettre son signal électrique de sortie au Convertisseur Analogique-Numérique (CAN) 10-bits qui transmettra à son tour les données au coeur logique qui contrôle l'ensemble des blocs dédiés à la génération.

Le QT-Xr dispose d'un contrôleur analogique qui joue un rôle très important dans le bon fonctionnement de la génération. Celui-ci régule l'intensité lumineuse détectée par les capteurs CMOS dans le but de la maintenir aussi haute que possible tout en évitant la saturation du CAN. Cette action se traduit par la modification du courant dans les LED ou du temps d'exposition des capteurs. La figure 2.3 représente la disposition des blocs participants à la génération.

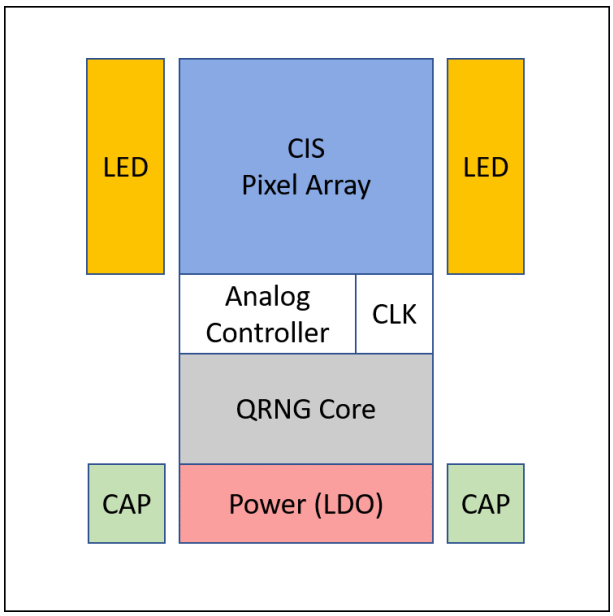


FIGURE 2.3 – Disposition de la génération des nombres aléatoires dans le QT-Xr | Source : SK Telecom [3]

2.1.3. Registres et modes de fonctionnement

Le QT-Xr est paramétrable par l'intermédiaire de quelques registres qui sont dévoués exclusivement à l'un des deux protocoles. Le protocole I2C est immédiatement écarté en raison de ses faibles performances en terme de débit, de plus, à la différence du SPI, il ne propose pas l'accès à différents modes de fonctionnement.

Les registres concernant la communication SPI sont résumés dans le tableau 2.1. Les adresses et les valeurs des registres sont, à l'image de l'ensemble des communications avec le QT-Xr, d'une taille de 16 bits [4].

4. Pour le reste du rapport l'appellation "doublet" sera utilisée pour désigner un mot de 16 bits

Register Name	Description
SYS_SLEEP_CTL	Wake-Up
	Sleep
TG_ENABLE	On/Off TG Operation
QRNG_MODE_CTL	RDO
	SDO
	Raw
QRNG_FIFO_CTL	SRAM Delay On/Off
	Clock Inversion
	Clear
SYS_SW_RESET	Register
	QRNG
	ISP
	TG
	MCU

TABLE 2.1 – Configuration des registres du QT-Xr | Source : ID Quantique

Le registre "QRNG_MODE_CTL" est particulièrement intéressant puisqu'il offre la possibilité de choisir entre trois différents modes de fonctionnement :

- Random number generation Data Output (RDO) : C'est le mode "classique" du QT-Xr, il permet de fournir des nombres aléatoires issus d'un post-traitement et donc de meilleure qualité.
- Sample Data Output (SDO) : C'est un mode alternatif qui permet de récupérer des échantillons des nombres générés par les capteurs
- Row : Aucune information n'est fournie dans la documentation quant à l'utilisation ou la manière de lire les données dans ce dernier mode. Celui-ci ne sera donc pas utilisé au cours de ce projet.

A première vue c'est l'utilisation du mode "RDO" qui semble la plus pertinente puisqu'il permet de récupérer des nombres de meilleure qualité (sur lesquels a déjà été effectué un post-traitement). Cependant, le mode "SDO" propose un débit maximum de 6 Mb/s qui est donc quatre fois plus important que celui du mode RDO.

Là encore, aucune explication concernant cette perte de débit n'est disponible mais il est cohérent d'émettre l'hypothèse qu'elle a lieu lors du post traitement et plus particulièrement durant une étape de débiaisage à l'image de la méthode de Von Neuman décrite dans la section 4 de l'état de l'art.

Pour les deux modes de fonctionnement, les données sont envoyées par cycle de 9 doublets consécutifs. Le tableau 2.2 résume l'ordre d'arrivée des doublets durant un cycle [5].

Index	1		2	3	4	5	6	7	8	9	
RDO	0xA2	Health	D1	D2	D3	D4	D5	D6	D7	D8	
SDO	D1		D2	D3	D4	D5	D6	D7	D8	0xA2	LN

TABLE 2.2 – Ordre d'arrivée des données lors d'un cycle de lecture du QT-Xr

Dans le mode RDO, la lecture des données sera donc précédée de deux octets, le premier est le code d'en-tête 0xA2 tandis que le second fournit des informations confirmant le bon fonctionnement de la génération des nombres aléatoires. Comme le montre le tableau 2.3, chacun des 7 derniers bits permet de signaler une erreur particulière dans la génération, tandis que le bit de poids fort confirme que la vérification de santé a bien été effectuée.

Health Bit	Description
b[7]	Health Check Ready
b[6]	RNG Proportion Error
b[5]	RNG Repetition Error
b[4]	Image Max Error
b[3]	Image Min Error
b[2]	Sensor Operation Error
b[1]	LED Operation Error
b[0]	Dark Sum Error

TABLE 2.3 – Informations sur la santé de la génération du QT-Xr

Le mode SDO permet quant à lui d'avoir accès à des échantillons bruts en provenance directe du CAN. Chaque pixel de la grille va fournir deux bits. Les données arrivent par l'intermédiaire de 8 premiers doublets (128 bits) suivie de deux octets, le premier pour l'en-tête 0xA2 tandis que le second fournit le numéro de la ligne correspondant aux 8 doublets. Il faudra donc deux cycles pour recevoir des échantillons d'une ligne complète et par conséquent 200 cycles pour recevoir des échantillons de l'ensemble de la grille.

5. Dans ce rapport, le mot cycle sera utilisé pour désigner la lecture de 9 doublets successifs

2.2. Gestionnaire SPI

La figure 2.4 présente la partie de l'architecture destinée à lire les données aléatoires générées par le QT-Xr.

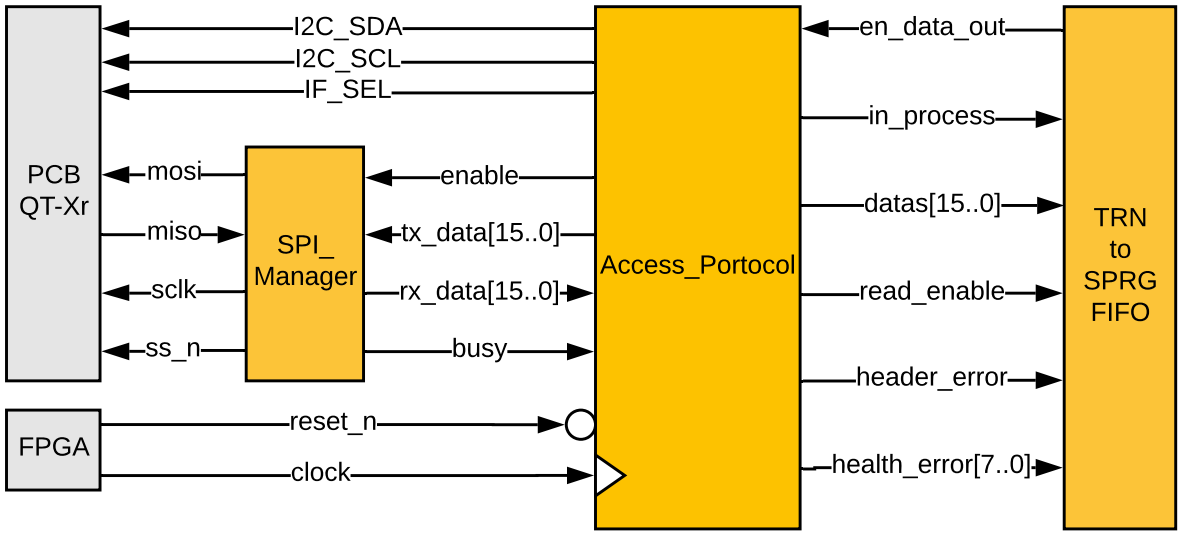


FIGURE 2.4 – Schéma détaillé de la cellule permettant l'extraction des données

L'entité "SPI_Manager" est un bloc de contrôle permettant essentiellement de traduire les envois de "access_protocol" vers le QT-Xr et inversement.

- *MOSI, MISO, SCLK* et *SS_n* : représentent les lignes de la communication SPI
- *CLK_DIV* : permet de définir la fréquence du signal d'horloge *SCLK* de sorte que : $F_{SCLK} = \frac{F_{CLK}}{2CLK\_DIV}$
- *enable* : permet d'initier une communication SPI
- *tx_data/rx_data* : permettent la transmission/réception des données en écriture/lecture
- *busy* : permet de signaler qu'une transmission est en cours

Ce rapport ne détaillera pas d'avantage ce bloc qui est assez trivial. Toutefois, une explication un peu plus détaillée de son fonctionnement est donnée dans la section 1.1 du rapport précédent [7].

2.3. Protocole de lecture

Pour la lecture des données du QT-Xr ainsi que la modification de ses registres, la note d'application de la puce [20] recommande l'usage d'un protocole précis. L'entité "Access_-protocol" est une machine d'état qui va permettre de réaliser ce protocole. Son diagramme est donné par la figure 2.5.

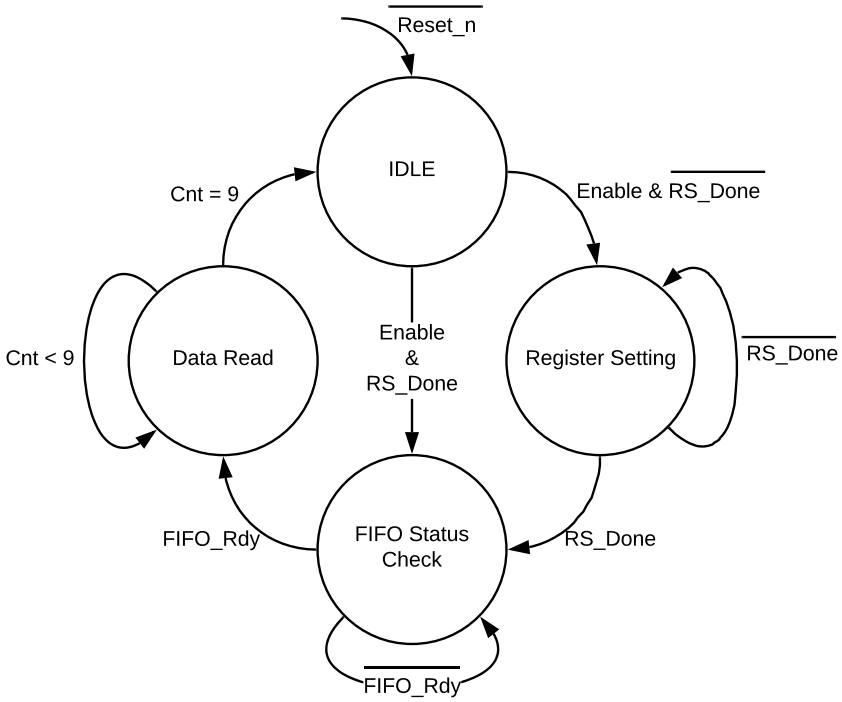


FIGURE 2.5 – Schéma de la machine d'état "Access_protocol"

2.3.1. Idle

Idle est une phase d'attente et d'initialisation de la communication avec le QT-Xr. Le protocole se lance sur le front montant du signal *Enable*. A ce moment là, les pins sont configurés pour signaler au QT-Xr que nous souhaitons communiquer via l'interface SPI. Pour cela, le signal *IF_SEL* est forcé à l'état bas tandis que les signaux *I2C_SCL* et *I2C_SDA* sont forcés à l'état haut.

2.3.2. Register Setting

Register Setting est un ensemble de 4 états et d'un délai durant lesquels sont configurés les valeurs des différents registres. La configuration débute par l'état *Reg_start* durant lequel est envoyé le code de lancement 0x1415 qui va signaler au QRNG de se préparer à la modification des registres. Après un délais de 30 μs, la machine envoie successivement l'adresse et la valeur des registres qui doivent être configurées. La fin de la modification des registres est signalée pendant l'état *Reg_Stop* par le code de terminaison 0xE75A.

Il est important de souligner que le QT-Xr contient deux First In First Out (Mémoire en file d'attente des données) (FIFO) de 3200 bits qui permettent de stocker les données en attente de lecture. Les tests réalisés lors de ce projet ont démontrés que lorsqu'il est mis sous tension ou lorsqu'une ré-initialisation complète a lieu, le QT-Xr commence à remplir immédiatement ces FIFO de données aléatoires en mode RDO (mode de fonctionnement par défaut). Ainsi, l'utilisation du mode RDO n'implique que la configuration du registre de contrôle tandis que celle du mode SDO implique de vider les FIFOs, en outre, cette action doit être effectué 100 ms après désactivation temporaire de la génération. Le schéma de la figure 2.6 résume les actions à effectuer pour l'utilisation du mode SDO.

2.3.3. Status Check

Status Check est un ensemble de deux états et d'un délai qui permettent de vérifier l'état actuel des FIFO du QRNG. La vérification des FIFO se fait par la l'envoie du code 0x4100, le QT-Xr répond instantanément par un doublet, cette réponse a trois états possibles :

Valeur	Etat
0x0001	Prêt
0x0006	FIFO en cours de remplissage
0x0000	La génération est en cours de lancement

TABLE 2.4 – État actuel des FIFO

Ainsi, seul la réception du code 0x0001 permet de lancer la lecture des données. Dans le cas contraire, il faut attendre 30 μs pour vérifier de nouveau l'état des FIFOs.

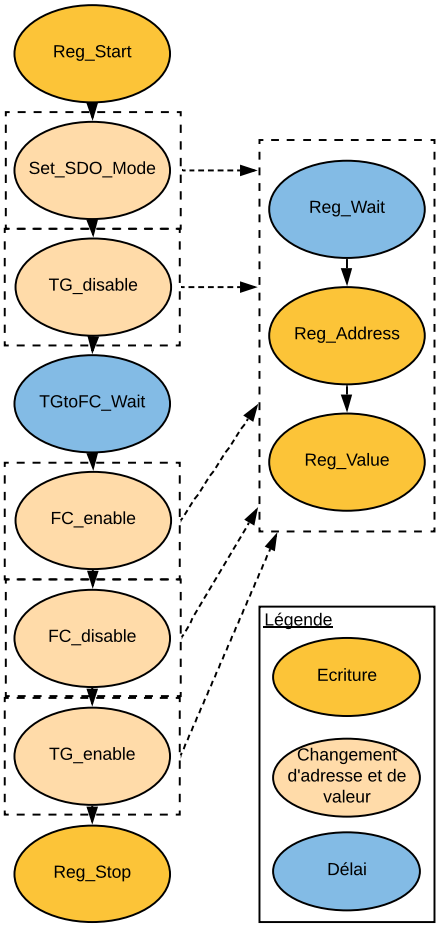


FIGURE 2.6 – Configuration des registres en mode SDO

2.3.4. Data Read

Data Read est lui aussi composé de deux états et un délai durant lesquels les nombres aléatoires vont être récupérés. Le QT-Xr envoie ses valeurs lorsqu'il reçoit 0x0000. Le doublet lu par la machine d'état est copié sur le signal *data* pour être transmis vers l'extérieur, sa disponibilité est signalée par *read_enable*. Un compteur permet de connaître le nombre de doublet qui ont été lu durant le cycle. Le délai doit permettre d'espacer les requêtes de données de 250 ns. A la fin du cycle, la machine d'état est ramené dans l'état IDLE et attend à nouveau un front montant du signal enable. La configuration des registres ayant déjà été effectuée, l'état suivant est cette fois *FC_Adress*. Le délai *DR2SC_Wait* permet d'espacer la lecture de la dernière donnée du cycle de la vérification de FIFO

La figure 2.7 présente le diagramme détaillé de fonctionnement de la machine d'état. La notation "not Busy" signifie l'attente de la transmission SPI. Pour clarifier la machine d'état, les flèches de maintien en état n'apparaissent pas sur cette représentation, il faut donc considérer

que si rien n'est spécifié, la machine garde son état actuel.

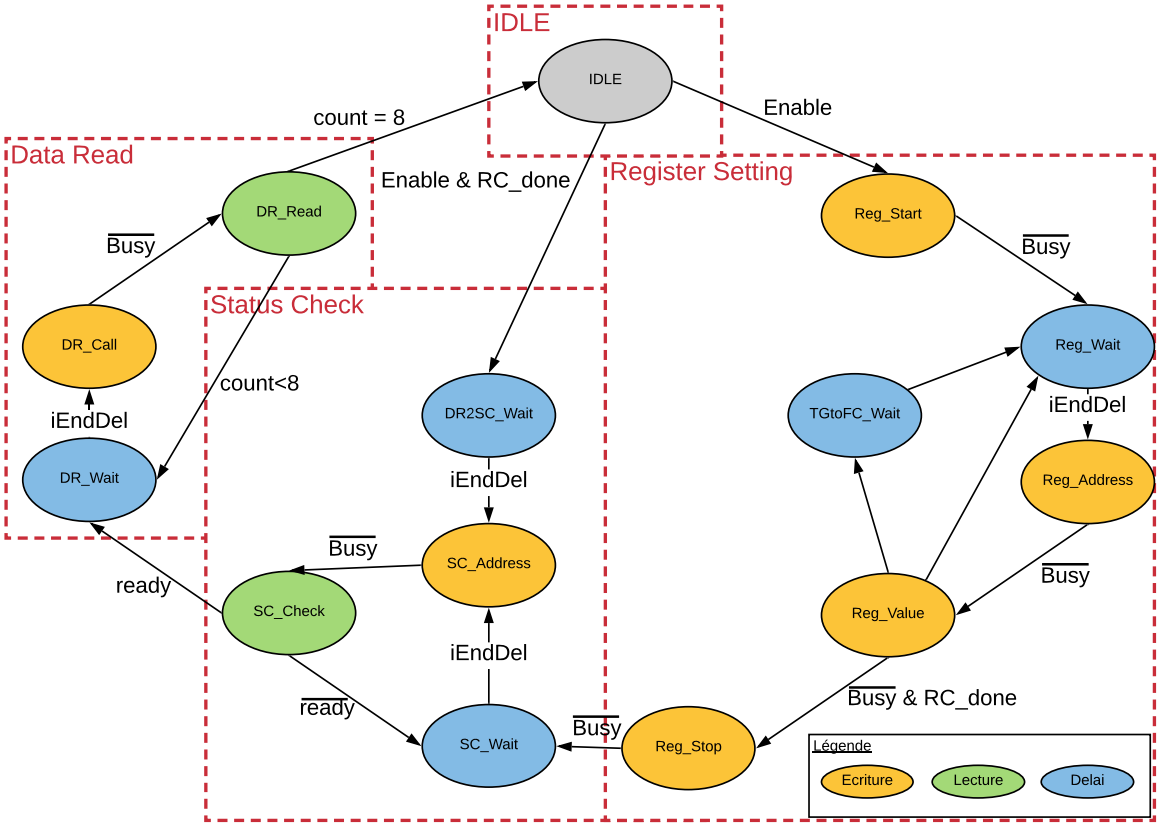


FIGURE 2.7 – Schéma détaillé de la machine d'état "Access_protocol"

3. DÉBIAISAGE

L'un des objectifs principaux de ce projet est de réussir à obtenir le débit le plus élevé possible. Or, l'utilisation du mode SDO permet d'obtenir un débit de 6 Mb/s. Cependant l'utilisation de ce mode n'est pas sans conséquence sur la qualité des suites que nous pouvons obtenir par rapport au mode RDO. Effectivement, le travail réalisé au cours de ce projet a pu démontrer que les séquences de nombres en provenance direct du mode d'échantillonnage comportaient quelques biais. Des résultats détaillés concernant les tests statistiques qui ont été effectués seront exposés dans la section 3 du chapitre 4.

En conséquence, il serait intéressant d'utiliser une méthode permettant de débiaiser les séquences générées tout en maintenant un débit optimal. La section 4 de l'état de l'art a présenté différentes méthodes parmi lesquels se trouvait l'utilisation d'une fonction éponge. Cette fonction, permet de maintenir un débit constant tout en améliorant drastiquement la distribution des suites.

3.1. Keccak

Keccak est une fonction de hachage cryptographique développé dans le cadre de la "NIST hash function competition" lancé en 2007 par le NIST [6]. Cette compétition avait pour objectif de définir SHA-3, une alternative à la fonction de hachage SHA-2 dans le cas ou celle-ci viendrait à être compromise. En 2012, Keccak a officiellement été annoncé vainqueur de la compétition [21, 22].

La fonction Keccak stocke des données dans un vecteur appelé état (state) qui est la concaténation de deux vecteurs appelés taux (rate) et capacité (capacity). Pendant l'exécution de l'algorithme, l'état va être brassé par l'intermédiaire d'une fonction appelée routine permutation qui est elle même composée d'un nombre Nr de round.

Elle repose sur les deux paramètres principaux que sont l et c.

Le premier va permettre à la fois de définir la taille b de l'état et le nombre N_r de round de la routine de permutation. De sorte que : $b = 25 \cdot 2^l$ et $N_r = 12 + 2 \cdot l$. Le paramètre c correspond à la taille du vecteur de capacité, et définit par extension la taille r du vecteur taux. Tel que $b = r + c$.

Les spécifications de Keccak autorisent 7 valeurs de l (de 0 à 6) cependant, seul la dernière a été retenue pour la fonction SHA-3. Il est donc beaucoup plus courant de rencontrer la version de l'algorithme où $b = 1600$ et $N_r = 24$.

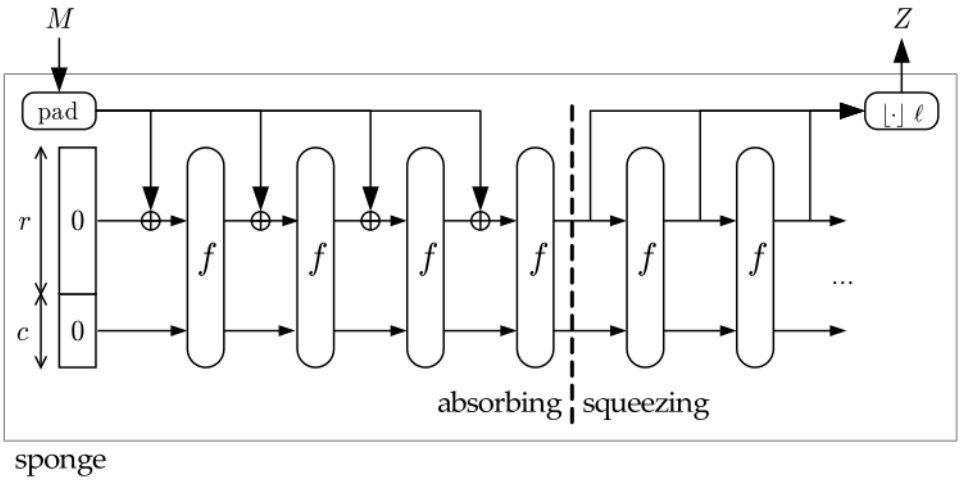


FIGURE 2.8 – Schéma de fonctionnement d'une fonction éponge | Source : keccak.team

La construction de Keccak est représenté sur le schéma de la figure 2.8. Keccak combine les avantages du chiffrement de flux avec ceux d'une fonction de hachage en utilisant une construc-

6. National Institute of Standards and Technology

tion appelée "fonction éponge"[23]. A l'image d'une fonction de hachage traditionnelle, dans Keccak, le message peut avoir n'importe qu'elle taille d'entrée mais ressortira toujours avec une taille fixe. Pour cela, dans une première partie de pré-traitement, le message d'entrée est divisé en P blocs de taille r qui seront récupérés et hachés par l'algorithme dans le but qu'il fournisse une valeur de hachage elle aussi répartit en plusieurs blocs de taille r. Si le message d'entrée n'est pas un multiple de r, celui-ci va être "rembourré" (padding). De plus, pendant cette étape de de pré-traitement, l'état du keccak est initialisé à '0'.

L'algorithme interne est construit suivant deux phases :

- Absorption : Dans une première phase, le message préalablement découpé en blocs de taille r va être XORé [7] avec le taux de l'état actuel. L'état ainsi obtenu est fourni à la routine de permutation f, dans laquelle les bits vont être brassés pour ressortir sous la forme d'un nouvel état.

- Restitution : La seconde phase de l'algorithme va permettre de récupérer (sur demande) la valeur de hachage par bloc de taille r à partir du taux actuel. Après chaque requête, une nouvelle routine de permutation est appliquée sur l'état.

La routine de permutation (noté f sur la figure 2.8) est décrite de la manière suivante. Le vecteur d'entrée est vu comme une matrice de taille $5 \times 5 \times w$ (où $w = 2^l$) (voir figure 2.9) .

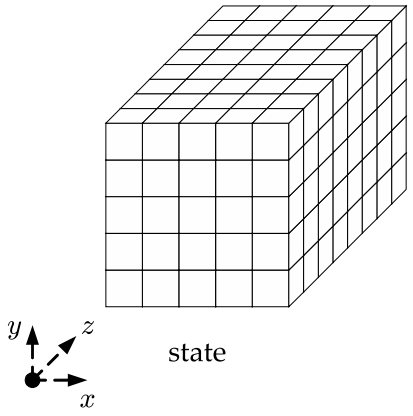


FIGURE 2.9 – Représentation matricielle de l'état | Source : keccak.team

Lors d'un round, cette matrice subit un brassage par l'intermédiaire de 5 sous-fonctions (theta, rho, pi, Chi et iota) [8].

7. Effectuer une porte logique OU-Exculsif entre deux entrés

8. afin de ne pas surcharger les explications, les sous-fonctions internes aux round ne seront pas détaillées dans ce rapport. Les spécifications de keccak les expliques précisément [24]

Enfin, la capacité joue le rôle d'élément de sécurité dans l'algorithme. En effet, alors que l'ensemble de l'état est brassé lors d'une permutation seul le taux est accessible depuis l'extérieur de la fonction, le manque d'information concernant la capacité constitue ainsi une difficulté majeur pour un attaquant. Cette difficulté augmentera avec la valeur de c.

Initialement pour la création de la fonction de hachage cryptographique SHA-3, le NIST avait demandé 4 tailles différentes pour la valeur de hachage : $\{224, 256, 384, 512\}$, celle-ci assuraient donc des sécurités [9] respectives de $\{2^{112}, 2^{128}, 2^{192}, 2^{256}\}$. Pour répondre à cette demande Keccak préconise simplement de prendre les premiers bits du taux qui sont nécessaires à l'extraction de la taille demandée. En revanche, les tailles de capacité les plus fréquemment utilisées pour Keccak1600 ont été choisies en fonction de ces paramètres suivant le tableau 2.6.

Output Size	b	r	c
224	1600	1152	448
256	1600	1088	512
384	1600	832	768
512	1600	576	1024

TABLE 2.5 – Paramètres de Keccak pour SHA-3

3.2. PRNG basé sur keccak

3.2.1. KeccakPRNG

L'un des faits remarquables d'une fonction éponge tel que Keccak est la très bonne qualité de la distribution statistique des bits de sorties. Bertoni *et al.* (créateurs de Keccak) ont donc naturellement décidé de publier un papier concernant la pseudo-génération de nombres aléatoires basé sur leur construction en éponge [25].

Dans celui-ci, ils décrivent un PRNG dans lequel est utilisé une source d'aléa de bonne entropie (typiquement un TRNG) pour alimenter l'entrée d'une fonction éponge. Cette méthode fournit des nombres de très bonne qualité puisqu'ils bénéficient de l'entropie initiale du TRNG couplé à la très bonne distribution des séquences de sortie de Keccak.

L'un des autres avantage de cette construction, est la possibilité de rafraîchir l'état initial ("seeding meterial") régulièrement pour résister au mieux aux différentes attaques. Cette méthode était bien entendu déjà employé auparavant mais était extrêmement demandeuse en ressources pour pouvoir stocker les données en attendant leur lecture. Dans le cas d'une fonction

9. résistance aux collisions, voir section 5 de l'état de l'art.

éponge en revanche, le nombre d'absorption peut être infini et n'est jamais limité par la place mémoire.

De plus, dans les fonctions éponges, les deux phases peuvent quasiment fonctionner indépendamment. Cela implique qu'à la différence de beaucoup d'autres PRNG, le débit d'entrée n'est pas nécessairement inférieur au débit de la sortie qui pourrait même être plus élevé.

Il faut toutefois être conscient des conséquences d'une telle action. En effet, si le débit de la sortie dépasse le débit de l'entrée, les séquences générées ne seront plus constamment bénéficiaires de l'entropie de la source d'aléa. En d'autres termes, l'état d'entrée ne sera plus rafraîchi assez régulièrement. C'est donc la sécurité et la qualité des séquences qui serait immédiatement impacté.

3.2.2. Limitations et améliorations

Deux principales limitations résident encore dans ce type de PRNG [26] :

- Le PRNG manque de confidentialité persistante [10] (forward secrecy).
- Dans certains cas, le générateur ne parvient pas à extraire des séquences avec une distribution suffisamment bonne pour qu'elles soient encore considérées comme aléatoire.

Pour pallier ces problèmes, Gaži & Tessaro ont proposé deux modifications à la construction initiale.

Ajouter une graine ("seed") en amont de l'absorption du message pour régler les problèmes de distribution et ajouter une nouvelle boucle lors de la phase de restitution pour empêcher l'attaquant de retrouver les états précédents. Cette boucle est composée d'une permutation suivie d'un effacement de la capacité.

Deux nouveaux paramètres de sécurité s'ajoutent alors à c. Le premier est le paramètre s qui définit le nombre de graines qui seront combinées alternativement avec les blocs d'entrée. Le second est le paramètre t qui correspond au nombre d'itération de la boucle après chaque lecture d'un bloc de sortie.

3.2.3. SPRG

Le nouveau PRNG ainsi modifié est nommé SPRG. Cet algorithme fut tout d'abord implémenté en programmation logiciel (language C) par L. Steiner dans le projet intitulé HERVA

10. Si un attaquant parvient à retrouver l'état du générateur, il pourra aisément retrouver les états précédents

[27] puis en matériel (language VHDL) par L. Gantel [28].

La figure 2.10 représente le schéma bloc de l'architecture du SPRG de Laurent Gantel. Seul les transmissions des données ainsi que la taille des signaux correspondant sont présentés dans cette figure.

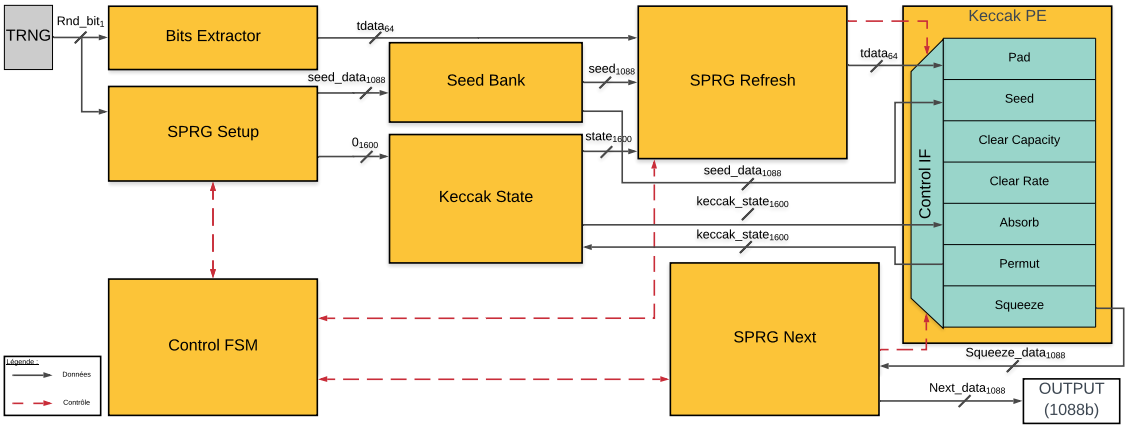


FIGURE 2.10 – Schéma de l'architecture du SPRG

- La fonction Keccak se retrouve dans l'entité *Keccak-PE*. Tandis que les autres blocs, en particulier *SPRG Setup, SPRG Refresh* et *SPRG Next* ont été ajouté pour implanter les modifications liées à l'algorithme SPRG.

- L'entité Keccak State permet de garder en mémoire l'état actuel. Il peut être écrit par *SPRG Setup* qui va réaliser l'initialisation de l'état à 0, ou par Keccak-PE qui va fournir le résultat de chaque permutation.

- Le bloc *SPRG Setup* va permettre la création de s graines de r nombres aléatoires en provenance de la source. Les graines sont stockées par le bloc *Seed Bank*.

- *SPRG Refresh* dirige la phase d'absorption de keccak. Il utilise un vecteur de contrôle de 7 bits permettant de choisir quelles fonctions de *Keccak-PE* doivent être utilisées ce contrôle est effectué par le signal d'activation des blocs (Exemple : en_pad_i). La figure 2.11 montre les différentes actions réalisées par le bloc *Keccak-PE* sous le contrôle de *SPRG Refresh*.

- SPRG Next dirige la phase de restitution de Keccak également par le biais d'un vecteur de contrôle. La figure 2.12 montre les actions réalisées par le bloc *Keccak-PE* sous le contrôle de *SPRG Next*.

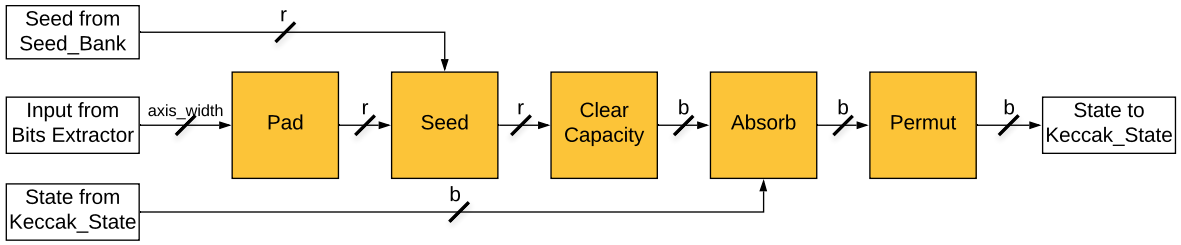


FIGURE 2.11 – Routine de fonctionnement de SPRG Refresh

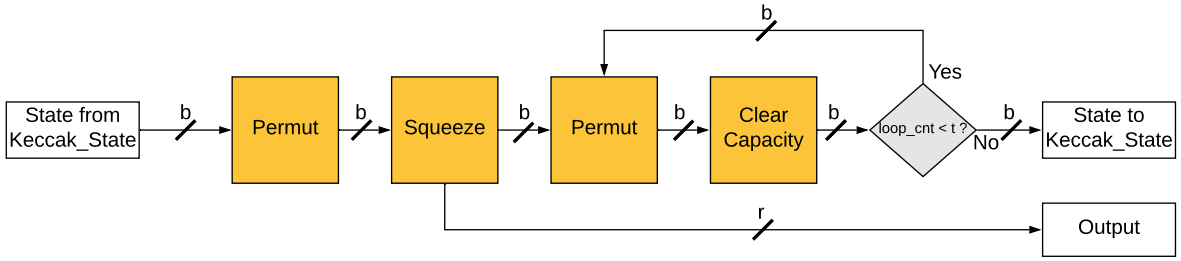


FIGURE 2.12 – Routine de fonctionnement de SPRG Next

3.3. Modularité

Le tableau 2.6 résume les valeurs des paramètres de Keccak qui ont étés retenues pour l'algorithme SPRG.

Paramètre	Notation	Valeur pour SHA-3
Taille de "state"	b	1600
Taille de "rate"	r	1088
Taille de "capacity"	c	512
Nombre de rounds	Nr	24

TABLE 2.6 – Paramètres de Keccak pour le SPRG

L'utilisation de ces paramètres, et en particulier le choix d'un état de 1'600 bits rend l'implémentation VHDL du SPRG extrêmement gourmande en ressource. Comme le montre la section 2 du chapitre 4, la synthèse de cette architecture sur la carte iCE40UP impose l'utilisation d'environ 30'000 Look Up Table(Table de correspondance) (LUT) et 68 blocs de mémoire Random Access Memory (mémoire vive) (RAM) alors que celui-ci ne contient que 5'280 LUT et 30 bloc de RAM .

Cette forte consommation est peu souhaitable dans notre projet puisque l'un des objectifs

principaux est de créer un système le plus compact possible. Or il va de soi que la taille d'un FPGA est souvent hautement corrélées avec son nombre d'éléments logiques disponibles ainsi que son nombre d'emplacement mémoire. Une solution intéressante à ce problème est la possibilité de rendre le code SPRG modulaire pour qu'il s'adapte au mieux au FPGA dans lequel il est embarqué.

Pour cela, nous allons utiliser les spécifications de l'algorithme Keccak pour permettre de réduire les ressources utilisées par l'algorithme et de les adapter au mieux au FPGA ciblé.

La méthode la plus efficace pour réduire le nombre de ressources utilisées par le SPRG est la diminution du paramètre l. Bien entendu, l'utilisation d'un état plus petit de Keccak diminue inévitablement les possibilités de choix de capacité ce qui a une incidence sur la sécurité de la génération. Néanmoins, dans leur papier présentant la construction du KeccakPRG [25] Bertoni *et al.* avaient réalisé, pour répondre à la même problématique d'embarquabilité, des essaies de construction avec un état de 200 bits et une capacité de 104 puis 136 bits. Dans les deux cas, leur algorithme a permis les générations de séquences qui ont passées les tests statistiques du NIST.

Ce projet n'étant pas ciblé initialement sur la cryptographie, il ne met pas en oeuvre plus d'étude concernant la sécurité du procédé. En revanche, il est important de donner également la possibilité à l'utilisateur de modifier les différents paramètres de sécurité.

L'architecture implémenté permet donc d'accéder à la modification des paramètres génériques : l, r (et par extension c), s et t.

4. INTERFACES

La figure 2.13 présente les modifications apportées à l'entité SPRG pour optimiser la récupération des données aléatoires dans le cadre de ce projet.

4.1. TRN to SPRG FIFO

L'objectif de ce module est de remplacer le bloc *Bits Extractor* de l'architecture SPRG. Ainsi, il permet de relier les données de sorties de *Access_protocol* (16 bits) avec l'entrée aléatoire tdata (8 bits) de l'algorithme SPRG. Basé sur le fonctionnement d'une FIFO, le bloc permet d'interfacer les deux entités en contrôlant d'un côté la lectures des données grâce au signal *en_data_out* et de l'autre l'écriture des données vers les deux différentes entités que sont *SPRG Refresh* et *SPRG Setup*. La mémoire qui lui est alloué est définit de manière générique par l'utilisateur. Deux variables permettent de connaître l'état actuel de la FIFO :

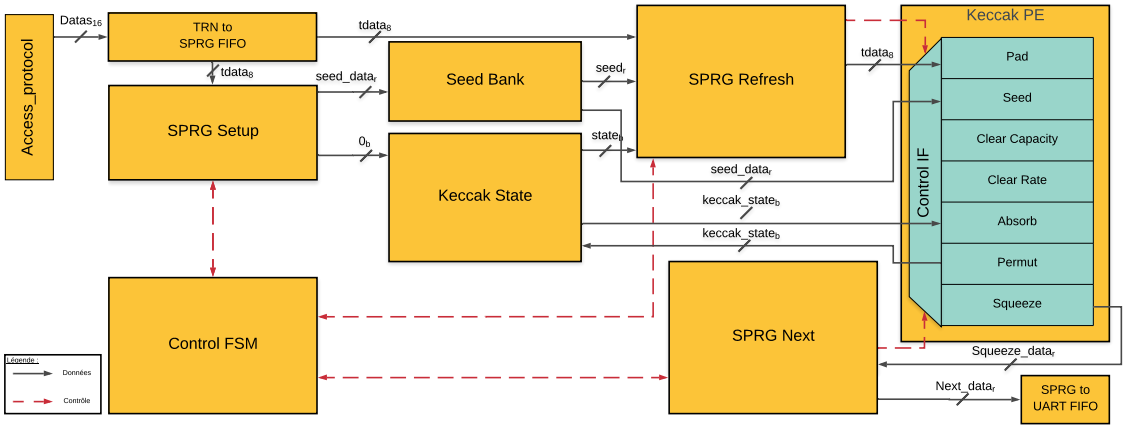


FIGURE 2.13 – Schéma de l'architecture SPRG modifié

- Tête (Head) : correspond à la position à laquelle les prochaines données doivent être lues pour être fournit au code SPRG. Sa valeur est incrémenté à chaque lecture.
- Queue (Tail) : correspond à la position à laquelle les prochaines données en provenance du QT-Xr doivent être écrites. Sa valeur est incrémenté à chaque écriture.

La figure 2.14 représente le fonctionnement de la FIFO.

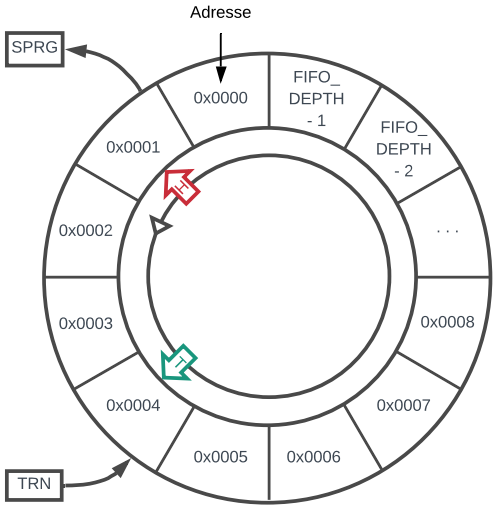


FIGURE 2.14 – Schéma de fonctionnement d'une FIFO circulaire

Si les deux variables prennent la même valeur, il existe deux possibilités :

- La queue rattrape la tête : Une valeur vient d'être ajouté, la FIFO est donc pleine et aucune nouvelle écriture ne doit avoir lieux tant que ces valeurs n'ont pas été lues.
- La tête rattrape la queue : Une valeur vient d'être lue, la FIFO est donc vide et aucune nouvelle lecture ne doit avoir lieux tant qu'il n'y a pas de nouvelle écriture.

4.2. SPRG to UART FIFO

De l'autre côté de l'entité SPRG, on retrouve une seconde FIFO construit de manière tout à fais similaire à la première. Celle-ci assurera l'envoie des bits de sortie du SPRG vers un gestionnaire UART (8bits) : *UART_manager*.

4.3. UART manager

Dans le cas de ce projet, *UART manager* permet la transmission des données vers une interface USB par l'intermédiaire du FTDI. C'est grâce à cette méthode que les nombre aléatoires peuvent être renvoyé vers un ordinateur. Bien entendu cette dernière partie de l'architecture est destiné à être remplacé pour convenir au mieux aux besoins de l'utilisateur concernant l'utilisation des séquences aléatoires.

MÉTHODES DE TEST

Ce chapitre a pour objectif de montrer les différentes méthodes utilisées pour vérifier le fonctionnement de l'architecture et pour évaluer la qualité des différentes séquences de nombres aléatoires générées.

1. LECTURE QT-XR

Pour récupérer les nombres aléatoires générés dans les modes de fonctionnement RDO et SDO, une première version de l'architecture dépourvue de l'entité SPRG a été mise en place. Pour la communication des données vers l'ordinateur, un nouveau bloc FIFO baptisé *RNG_read_controller* a été implémenté. La figure 3.1 présente le schéma de cette architecture.

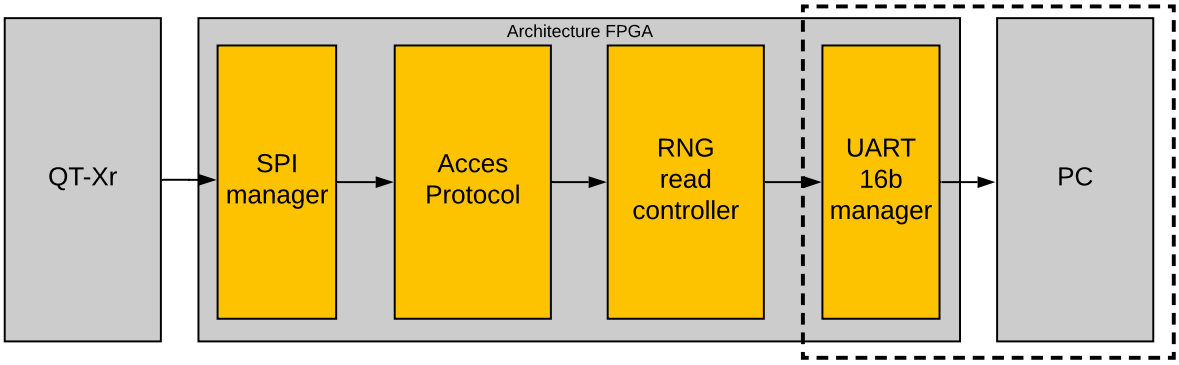


FIGURE 3.1 – Schéma de l'architecture de lecture du QT-Xr

La lecture du port sériel de l'ordinateur est effectué par l'intermédiaire d'un code Python qui écrit les données hexadécimales en chaînes de caractères dans un document texte, nous avons ainsi pu récupérer des fichiers de 60 Mb de données aléatoires sur lesquels nous pourront effectuer des tests dans le but de contrôler leur propriétés statistiques. Ce test permettra également de vérifier les débits atteignables avec les deux modes du QRNG.

2. TEST SOFTWARE DU SPRG

Afin d'avoir une base pour la vérification des modifications apportées à l'architecture matériel du SPRG, nous nous sommes procurés la version logiciel implémentée par L. Steiner [27]. Une légère modification du fichier SPRG.c regroupant les fonctions de la librairie Keccak, utilisées par le code SPRG, a permis le test de l'algorithme SPRG avec des versions allégés de Keccak. Nous ferons passer le fichier de 60 Mb généré par le mode SDO du générateur au

travers de l'algorithme SPRG. L'objectif est de récupérer un nouveau fichier de sortie de même taille qui permettra d'évaluer les performances de cette méthode de débiaisage. Nous pourrons également comparer les différentes versions de Keccak avec les données du mode RDO.

Attention toutefois, étant donné qu'il est impossible de récupérer simultanément les données des deux modes du QT-Xr, la comparaison ne concernera pas en réalité les mêmes données aléatoires. Pour palier à ce problème, il peut être utile de réitérer plusieurs fois les tests avec des données provenant d'une nouvelle mesure.

3. VÉRIFICATION D'IMPLÉMENTATION

L'ensemble des codes qui ont été réalisés ou modifiés au cours de ce projet, on été vérifiés en simulation. Pour l'entité *access_protocole* ainsi que les interfaces FIFO, SPI et UART, les codes ont été simulés à l'aide de fichiers DO tandis que les différentes entités de l'algorithme SPRG ont été simulés à l'aide de TestBenchs et comparés avec les résultats obtenus par le code software.

Le logicel de conception utilisé pour la famille de FPGA iCE40 est "iCECube2", pour la synthétisation du code VHDL, il utilise l'outil "Synplify Pro Synthésis". Pour les tests concernant les ressources nécessaire au bon fonctionnement de l'architecture, on règle au maximum le niveau d'effort de synthèse. Les résultats de synthèse seront présenté pour le FPGA "iCE40UP5K".

4. TESTS NIST

La publication 800-22 du NIST connu sous le nom "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications" [29] est, comme son nom l'indique, une suite de tests statistiques utilisée pour la qualification d'une séquence aléatoire destiné à une application cryptographique. Cette suite est composé de 15 tests [11]. Tous retournent une valeur-P correspondant à qualité de la séquence du point de vue du test en question. La séquence réussit le test si et seulement si sa valeur P est supérieur au seuil de 0.01. Si une séquence parvient à réussir l'intégralité des tests de la suite, elle peut alors être considéré comme aléatoire.

Pour la réalisation de ces tests statistiques, nous utiliserons l'implémentation Python de Steven Kho Ang disponnible sur GitHub [30].

11. les tests 11,13,14 et 15 peuvent posséder plusieurs variantes/paramètres

RÉSULTATS

.

1. LECTURE

L'architecture utilisée pour la lecture des données nous a permis de récupérer un fichier de 64 Mb de données aléatoires fourni par le mode RDO. La figure 4.2 est une représentation visuelle d'une partie du fichier. La grille a des dimensions de 500×500 pixels et chaque pixel représente 8 bits en niveau de gris, ainsi, cette image représente 2 Mb de nombres aléatoires. Même si la détection de motifs répétitifs dans l'image pourrait permettre de rapidement desceller un biais dans la séquence. Le cas contraire ne fonctionne pas, en effet, certains biais sont impossibles à observer à l'oeil nue et la vision d'une telle image ne peut pas suffire à déterminer la qualité de la séquence aléatoire. Pour illustrer ce fait, on peut utiliser à titre de comparaison l'image d'une séquence biaisé de nombres aléatoires généré par le mode SDO (voir figure 4.3). On remarque qu'aucune différence significative ne peut être observé.

La figure 4.1 montre les mesures effectuées sur les lignes de communication SPI à l'aide d'un analyseur logique pendant un cycle de lecture du mode RDO.

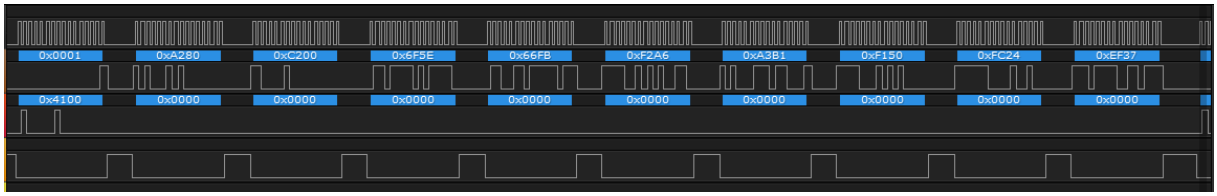


FIGURE 4.1 – Cycle de lecture du mode RDO, capturé avec un analyseur logique.

Les débits maximum observé lors de ces lectures ont pu confirmer les données exposé par la documentation :

- Le mode RDO offre bien un débit avoisinant les 1.5 Mb/s.

- Le mode SDO offre bien un débit avoisinant les 6 Mb/s

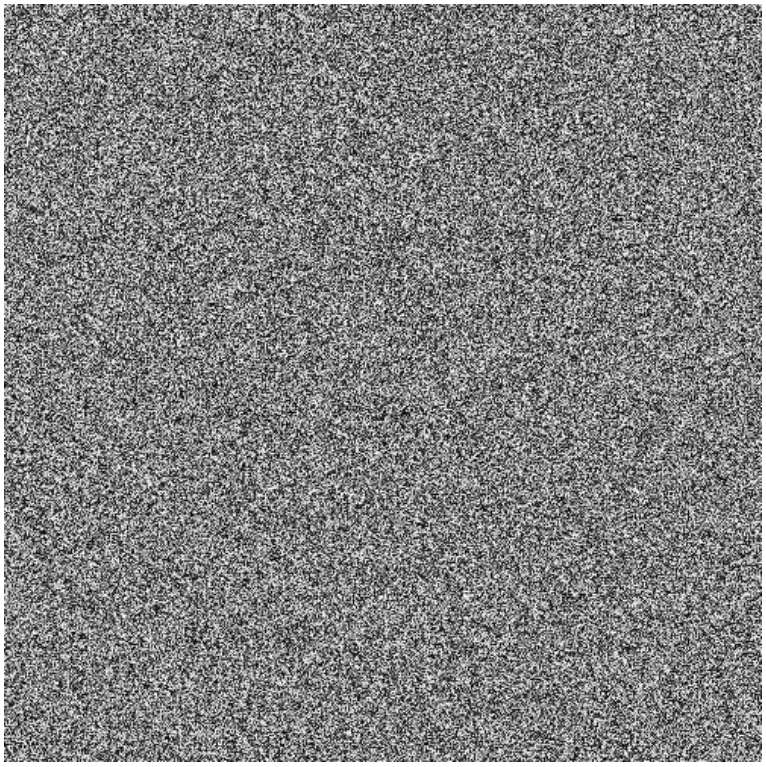


FIGURE 4.2 – Nombres aléatoires générés par le mode RDO du QT-Xr représentés sous forme d'image (Dimensions : 500x500x8)

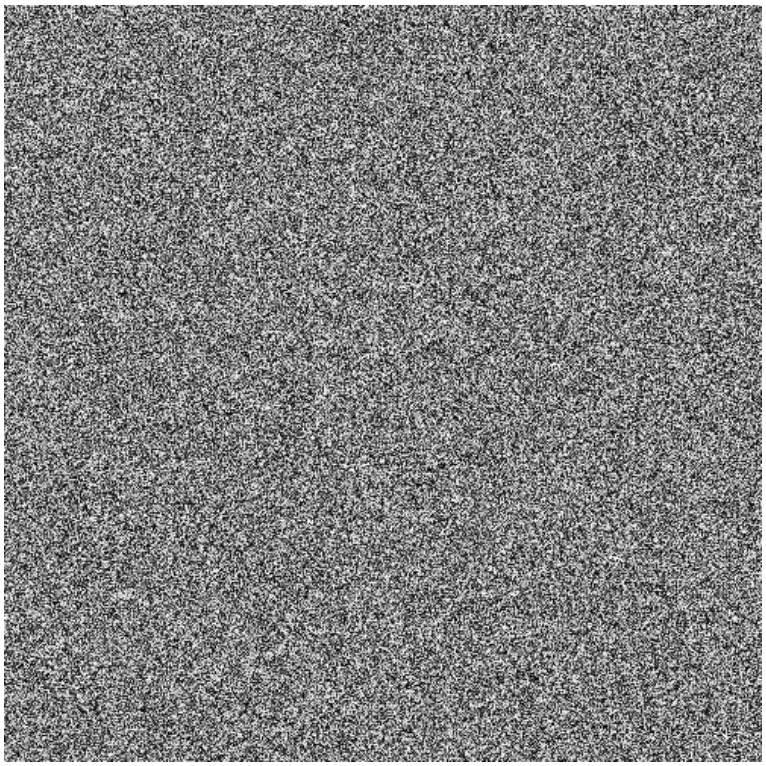


FIGURE 4.3 – Nombres aléatoires générés par le mode SDO du QT-Xr représentés sous forme d'image (Dimensions : 500x500x8)

2. RESSOURCES UTILISÉES

Les résultats présentés dans le tableau 4.1 concernent uniquement la synthèse de l'algorithme SPRG. On présente ici le nombre de LUT ainsi que le nombre de bloc RAM qui sont utilisés pour l'application d'une combinaison de paramètres génériques.

Les proportions d'utilisation sont donné pour le "iCE40UP5K" qui contient 5'280 LUT ainsi que 30 bloc RAM de 4kb [8].

ID	l	b	r	s	t	LUT's	Proportion LUT's	Blocks RAM's	Proportion RAM's
1	6	1600	1088	2	2	29781	564%	68	226%
2	5	800	288	2	2	12058	228%	18	30%
3	5	800	544	2	2	13949	264%	34	113%
4	4	400	144	2	2	6054	114%	9	30%
5	4	400	272	8	2	7207	136%	17	56%
6	3	200	136	2	2	4014	76%	9	30%
7	3	200	136	16	2	4028	76%	9	30%
8	3	200	136	2	20	4014	76%	9	30%

TABLE 4.1 – Résumé des ressources nécessaires pour contenir l'algorithme SPRG

La méthodologie choisi pour la réalisation de ces tests est la suivante.

1. Test représentant les paramètres initiaux utilisés pour le code SPRG. Cette mesure permet de démontrer l'intérêt de rendre l'algorithme modulaire puisqu'il utiliserait théoriquement 564% des LUT du FPGA.

2. La valeur de l est décrémenté de 1 mais le reste des paramètres restes inchangé. ($c = 512$) : cela permet d'observer la division drastique du nombre de LUT lorsque l'on utilise une autre version de Keccak.

3. Seul la valeur de r est modifié pour observer l'effet de la modification de la capacité. On peut en conclure que la partie taux de l'état utilise d'avantage de LUT que la capacité ceci n'est pas du à la routine de permutation qui va modifier l'ensemble des bits indépendamment de leur appartenance à l'un ou l'autre des vecteurs. En revanche, l'augmentation de la taille du taux impose l'augmentation du nombre de LUT nécessaire à la réalisation des fonctions telle que *pad,absorb ou seed*. On observe également un impact important sur la mémoire nécessaire à stocker les graines.

4. Test avec Keccak 400.

5. Modification du nombre de seed via la modification du paramètre s. Ce test permet de desceller une bonne méthode pour améliorer la sécurité du système progressivement, en augmentant légèrement les ressources nécessaires.

6. Test avec Keccak 200. C'est le premier test à pouvoir réellement être implémenté dans le "iCE40UP".

7. Grande augmentation de s. L'impact sur les ressources est peu significatif en raison de la diminution inévitable de la taille du taux.

8. Modification du paramètre t. Ce test n'a aucun impact sur les ressources nécessaires mais car la boucle de *SPRG Next* est successive et réutilise par conséquent les mêmes portes logiques. En revanche, cette modification aura une insidence sur la vitesse d'opération d'une étape de restitution des données.

3. TESTS NIST

Les tests statistique réalisés à l'aide de la suite de tests NIST de python on permis de vérifier la qualité des nombres aléatoires de cinq fichiers différents :

- RDO : Fichier de 64 Mb contenant des nombres aléatoires générés par le QRNG en mode RDO.

- SDO : Fichier de 60 Mb (taille légèrement différentes suite à la suppression des doublets permettant de lire le numéro de la ligne)

- SDO ligne 0x0b : Fichier de 596 kb contenant les nombres aléatoires générés par une seule ligne pour tenter de détecter la présence de biais supplémentaires.

- Keccak 1600 : Débiaisage SPRG des données du fichier SDO avec $l = 6$ pour keccak 1600.

- Keccak 400 : Débiaisage SPRG des données du fichier SDO avec $l = 4$ pour keccak 400.

La figure 4.4 expose les conclusions de chaque test pour les 5 fichiers différents.

File	Size (bits)	Result	Conclusions								
			1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
			Frequency Test (Monobit)	*Frequency Test within a Block*	*Run Test*	*Longest Run of Ones in a Block*	*Binary Matrix Rank Test*	*Discrete Fourier Transform (Spectral) Test*	*Non-Overlapping Template Matching Test*	*Overlapping Template Matching Test*
RDO	67 M	16/16	Random	Random	Random	Random	Random	Random	Random	Random
SDO	60 M	12/16	Non-Random	Random	Non-Random	Random	Random	Random	Random	Random
SDO ligne 0x0b	596 k	8/16	Non-Random	Random	Non-Random	Non-Random	Random	Random	Non-Random	Random
Keccak 1600	60 M	16/16	Random	Random	Random	Random	Random	Random	Random	Random
Keccak 400	60 M	16/16	Random	Random	Random	Random	Random	Random	Random	Random

			9	*10*	*11*	*12*	*13*	*14*	*15*	*16*
			Maurer's Universal Statistical test	*Linear Complexity Test*	*Serial test*	*Approximate Entropy Test*	*Cummulative Sums (Forward) Test*	*Cummulative Sums (Reverse) Test*	*Random Excursions Test*	*Random Excursions Variant Test*
RDO	67 M	16/16	Random	Random	2/2 Random	Random	Random	Random	8/8 Random	18/18 Random
SDO	60 M	12/16	Random	Random	2/2 Random	Random	Non-Random	Non-Random	8/8 Random	18/18 Random
SDO ligne 0x0b	596 k	8/16	Random	Random	2/2 Random	Non-Random	Non-Random	Non-Random	7/8 Random	14/14 Random
Keccak 1600	60 M	16/16	Random	Random	2/2 Random	Random	Random	Random	8/8 Random	18/18 Random
Keccak 400	60 M	16/16	Random	Random	2/2 Random	Random	Random	Random	8/8 Random	18/18 Random

FIGURE 4.4 – Résultats des tests NIST

La figure 4.5 permet de comparer les qualités obtenus par les différentes méthodes de débiaisage.

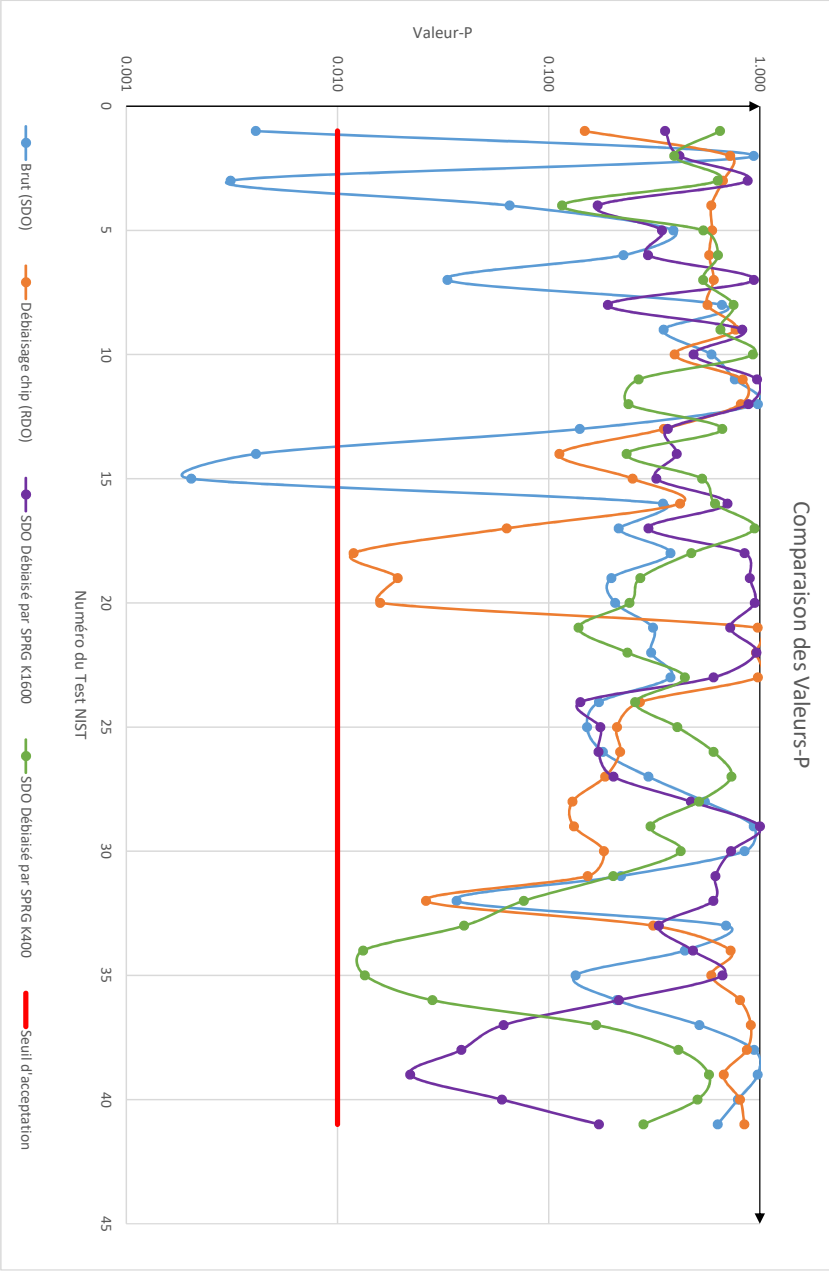


FIGURE 4.5 – Comparaison des valeurs-P

Tout d'abord le graphique confirme bien la présence de biais dans les données générées par le mode SDO. En effet, quatre tests donnent une valeur-P inférieur au seuil de 0.01. Ensuite,

les deux méthodes de débiaisage basé sur le SPRG ont effectivement permis d'améliorer la distribution de la séquence SDO. Enfin, il semble que la métrique fournie par la valeur-P des tests NIST ne permette pas d'identifier une méthode de débiaisage plus performante qu'une autre.

En conclusion, aux vues de la similarité des résultats des séquences générées, il semble intéressant de laisser le choix de la méthode d'utilisation de l'architecture à l'utilisateur. En effet, si le débit de 1.5 Mb/s lui convient , il peut choisir le mode RDO sans débiaisage SPRG. En revanche, si il souhaite obtenir un débit supérieur et qu'il possède les ressources disponibles, il peut utiliser l'algorithme SPRG et choisir les paramètres génériques adaptées. Cependant, le choix de cette méthode ne propose pas de certification des nombres qui pourra alors être réalisé par la suite à l'aide d'un processeur.

CONCLUSION

Ce projet consistait en la création d'un système complet de génération de vrais nombres aléatoires destinés à des applications en sécurité informatique, en particulier dans des systèmes embarqués. Les critères de réussite sont une taille réduite, un haut débit de génération et une faible consommation électrique. Par ailleurs, l'utilisation de paramètres génériques doivent permettre à un utilisateur d'adapter l'architecture aux ressources logiques disponibles.

A l'issue de ce projet, le choix du matériel utilisé à permis de répondre au problématique d'espace et de consommation électrique. Pour la génération des nombres aléatoires, deux modes ont été mis en place. Le premier, RDO permet la récupération de séquences aléatoires certifiées, avec un débit limité à 1.5 Mb/s. L'observation de ce dernier nous a amené à la conception du deuxième mode, SDO débiaisé qui profite à la fois des performances du générateur de nombres pseudo aléatoires SPRG et de l'entropie du QT-Xr. Celui-ci permet un débit de 6 Mb/s mais dont les séquences ne sont pas certifiées. On notera toutefois une consommation en éléments logiques plus importante qui peut être compensée par des paramètres génériques permettant d'adapter l'architecture aux besoins.

Les tests statistiques ont montré que les deux modes sont adapté à la génération de vrais nombres aléatoires. Par ailleurs, l'utilisation des tests NIST n'a pas permis d'identifier un générateur plus performant que l'autre.

Ce projet m'aura premièrement permis de découvrir et de prendre en mains de nombreux outils informatiques tel que "Python", l'environnement "Unix" et le logiciel "Make" pour l'exécution de fichiers sources ou encore les outils de développement FPGA de Lattice en particulier "iCE Cube 2" et "Diamond Programmer". Deuxièmement, il aura renforcé mes connaissances en VHDL, avec la découverte de nouvelles notions telles que les paquets, les fonctions, les paramètres génériques et les "tests bench". Enfin, il m'aura permis d'acquérir une grande quantité de connaissances théoriques concernant, le monde des nombres aléatoires et leur génération, les algorithmes de débiaisage et plus particulièrement les fonctions de hachages cryptographiques. Finalement, il aura fait naître en moi un intérêt tout particulier pour la cryptographie.

BIBLIOGRAPHIE

[1] "Site internet de la société id quantique." [Online]. Available : https://www.idquantique.com/

[2] S. Yicheng, "A ultra-fast quantum random number generator," Ph.D. dissertation, 2016.

[3] *SKT QRNG Chip Specification*, SK telecom, Jun. 2017, version 0.5.

[4] J. Roux, "Détection d'intrusion dans l'internet des objets : Problématiques de sécurité au sein des domiciles," in *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI)*, 2017, p. 4p.

[5] R. Dumont, "Cryptographie et sécurité informatique," *Université de Liège. Belgique*, 2010.

[6] A. Kerckhoffs, "La cryptographic militaire," *Journal des sciences militaires*, pp. 5–38, 1883.

[7] L. Piccot, "Génération de vrais nombres aléatoires sur système embarqué," 2019. [Online]. Available : http://

[8] *Product selector guide*, Lattice semiconductor, May 2018, order : I0211 Rev. 16.

[9] *iCE40 UltraPlus Family Data Sheet*, Lattice semiconductor, November 2018, fPGA-DS-02008-1.6.

[10] *FT2232H Dual High Speed USB to Multipurpose UART/FIFO IC*, Future Technology Devices International Limited, version 2.6.

[11] P. Martin-Löf, "The definition of random sequences," *Information and control*, vol. 9, no. 6, pp. 602–619, 1966.

[12] Wikipédia, "Suite aléatoire — wikipédia, l'encyclopédie libre," 2019, [En ligne ; Page disponible le 5-juin-2019]. [Online]. Available : http://fr.wikipedia.org/w/index.php?title=Suite_al%C3%A9atoire&oldid=159883923

[13] ——, "Générateur de nombres aléatoires — wikipédia, l'encyclopédie libre," 2019, [En ligne ; Page disponible le 3-juin-2019]. [Online].

Available : http://fr.wikipedia.org/w/index.php?title=G%C3%A9n%C3%A9rateur_de_nombres_al%C3%A9atoires&oldid=159819400

[14] A. Cherkaoui, "Générateurs de nombres véritablement aléatoires à base d'anneaux asynchrones : conception, caractérisation et sécurisation," Ph.D. dissertation, Saint-Etienne, 2014.

[15] J.-L. Danger, S. Guilley, and P. Hoogvorst, "Fast true random generator in fpgas," in *2007 IEEE Northeast Workshop on Circuits and Systems*. IEEE, 2007, pp. 506–509.

[16] M. Drutarovsky and P. Galajda, "A robust chaos-based true random number generator embedded in reconfigurable switched-capacitor hardware," in *2007 17th International Conference Radioelektronika*. IEEE, 2007, pp. 1–6.

[17] I. Q. SA, "What is the q in qrng ?" 2019, [En ligne ; Page disponible le 20-juin-2019]. [Online]. Available : https://marketing.idquantique.com/acton/attachment/11868/f-0226/1/-/-/-/-/What%20is%20the%20Q%20in%20QRNG_White%20Paper.pdf

[18] Wikipédia, "Fonction de hachage cryptographique — wikipédia, l'encyclopédie libre," 2019, [En ligne ; Page disponible le 3-juin-2019]. [Online]. Available : http://fr.wikipedia.org/w/index.php?title=Fonction_de_hachage_cryptographique&oldid=159835825

[19] "Site internet de la société sk telecom." [Online]. Available : https://www.sktelecom.com/

[20] *QRNG Chip Application Note Draft*, SK telecom & ID quantique, version 0.1, ID Quantique proprietary and confidential.

[21] C. Paar and J. Pelzl, "Sha-3 and the hash function keccak," *Understanding Cryptography A Textbook for Students and Practitioners, www. crypto-textbook. com*, 2010.

[22] G. Bertoni, J. Daemen, M. Peeters, and G. Assche, "The keccak reference," *Submission to NIST (Round 3)*, vol. 13, pp. 14–15, 2011.

[23] Wikipédia, "Fonction éponge — wikipédia, l'encyclopédie libre," 2018, [En ligne ; Page disponible le 24-octobre-2018]. [Online]. Available : http://fr.wikipedia.org/w/index.php?title=Fonction_%C3%A9ponge&oldid=153333348

[24] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak specifications," *Submission to nist (round 2)*, pp. 320–337, 2009.

[25] ——, "Sponge-based pseudo-random number generators," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2010, pp. 33–47.

[26] P. Gaži and S. Tessaro, "Provably robust sponge-based prngs and kdfs," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 87–116.

[27] L. Steiner, "Herva," 2018.

[28] L. GANTEL, "Herva hardware developments," 2019.

[29] L. E. Bassham, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, S. D. Leigh, M. Levenson, M. Vangel, N. A. Heckert, and D. L. Banks, "A statistical test suite for random and pseudorandom number generators for cryptographic applications| nist," Tech. Rep., 2010.

[30] S. K. Ang, "randomness_testsuite," https://https://github.com/stevenang/randomness_testsuite, 2018.