Hardware implementation of a pseudo random number generator based on chaotic iteration

Mohammed Bakiri

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

Thèse de Doctorat

N° 2018UBFCD014

THESIS presented by

MOHAMMED BAKIRI

to obtain the

Doctor of Philosophy Degree of
University of Bourgogne Franche-Comté
Department of Computer Science of Complex Systems (DISC)

Doctoral Faculty nº ED 37
Sciences for the Engineer and Microtechniques

Speciality : **Informatics**

Hardware Implementation of Pseudo Random Number Generator Based on Chaotic Iteration

Publicly supported in Belfort on 8 Janvier 2018 in front the Jury composed of :

M, ENRICO FORMENTI	Professor at University of Nice Sophia Antipolis	President
M, SYLVAIN CONTASSOT-VIVIER	Professor at University of Lorraine	Reporter
M, FRÉDÉRIC MAGOULÈS	Professor at University of Paris-Saclay	Reporter
M, CHRISTOPHE GUYEUX	Professor at University of Bourgogne Franche-Comté	Examiner
M, JEAN-FRANÇOIS COUCHOT	Associate Professor and HDR at l'Université Bourgogne Franche-Comté	Supervisor
M, ABDELKRIM KAMEL OUDJIDA	Master of Research and HDR at Centre de Développement des Technologies Avancées, Alger	Co-Thesis Director

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

Thèse de Doctorat

N° 2018UBFCD014

THÈSE présentée par

MOHAMMED BAKIRI

pour obtenir le

Grade de Docteur de
l'Université de Bourgogne Franche-Comté
Département d'Informatique des Systèmes Complexes (DISC)

Ecole doctorale n° ED 37
Sciences Pour l'Ingénieur et Microtechniques
Spécialité : **Informatique**

Implémentation matérielle de générateurs de nombres pseudo-aléatoires basés sur les itérations chaotiques

Soutenue publiquement à Belfort le 8 Janvier 2018 devant le Jury composé de :

M, ENRICO FORMENTI	Professeur à l'Université Nice Sophia Antipolis	Président
M, SYLVAIN CONTASSOT-VIVIER	Professeur à l'Universités de Lorraine	Rapporteur
M, FRÉDÉRIC MAGOULÈS	Professeur à l'Universite Paris-Saclay	Rapporteur
M, CHRISTOPHE GUYEUX	Professeur à l'Université Bourgogne Franche-Comté	Examinateur
M, JEAN-FRANÇOIS COUCHOT	Maître de Conférences et HDR à l'Université Bourgogne Franche-Comté	Directeur de thèse
M, ABDELKRIM KAMEL OUDJIDA	Maitre de Recherche et HDR à Centre de Développement des Technologies Avancées, Alger	Co-Directeur de thèse

ACKNOWLEDGEMENTS

Je souhaite avant toutes choses remercier deux personnes pour leur encadrement, leur disponibilité et leur amitié: Prof. ***Christophe Guyeux*** mon premier directeur de thèse puis Dr. ***Jean-François Couchot*** présent dès le début en tant que co-directeur puis directeur exclusif sur la dernière période. Ils ont su, malgré un emploi du temps bien chargé, être toujours présents à mes côtés, et me faire profiter de leur expérience, leur intelligence, et leurs connaissances des objets de ma recherche. Le travail que j'ai pu mener et ce document ne seraient pas ce qu'ils sont sans leur motivation et leurs encouragements, leur patience, leur recul, leur regard critique et la pertinence de leurs conseils. Ce fut un grand plaisir de travailler avec eux, et j'espère pouvoir continuer à le faire longtemps encore.

Je tiens également à remercier aussi Dr. *Abdelkrim Oudjida* ainsi que les membre de l'équipe IPLS pour leurs encouragements, leurs conseils, et leur expertise dans le domaine du digital aux sein du Centre de Développement des Technologies Avancées. Je suis infiniment reconnaissant envers *Nouma Izeboudjen*, *Sabrina Titri*, *Samir Tagzout*, *Ibrahim Bouzouia* et *Mohand Tahar Belaroussi* pour leur soutien, leurs encouragements, leur disponibilité et leur confiance. Ces quelques lignes de remerciement ne sont rien par rapport a ce que ce document leur doit.

Je tiens également à remercier les membres de mon jury de thèse particulièrement, Prof. *Sylvain Contassot-Vivier* et Prof. *Frédéric Magoules* qui m'ont fait l'honneur d'être les rapporteurs de cette thèse. Que Prof. *Enrico Formenti* soit aussi remercié pour avoir accepté d'être examinateur. Merci pour leurs suggestions et leurs précieux conseils, qui ont permis de clarifier et donc d'améliorer ce mémoire.

Je tiens aussi à remercier tous les membres de l'équipe AND à Belfort pour leur amitié et la bonne ambiance qu'ils contribuent à créer, spécialement *Amor Lalama*, *Neserine Khernane* et Prof. *Raphaël Couturier*.

Je ne remercierai jamais assez mes parents et mon épouse Radhia qui ont toujours été à mes cotés, mes frère et soeurs (Soumia, Foued, Karima et Walid), pour avoir toujours été présents, m'avoir toujours aidé et soutenu, et pas seulement durant mes études. Sans eux, sans leur gentillesse, leurs encouragements et leur dévouement, je n'en serais pas là.

CONTENTS

ABBREVIATIONS

RNG ..... Random Number Generator

PRNG ..... Pseudo Random Number Generator

TRNG ..... True Random Number Generator

CPRNG ..... Chaotic Pseudo Random Number Generator

LPRNG ..... Linear Pseudo Random Number Generator

CI ..... Chaotic Iteration

CIG ..... Chaotic Iteration Generalized

NIST ..... National Institute of Standard and Technologies

FIPS ..... Federal Information Processing Standard

FPGA ..... Field Programmable Gate Array

ASIC ..... Application-Specific Integrated Circuit

HDL ..... High Description Language

RTL ..... Register Transfert Level

HLS ..... High-Level Synthesis

SDK ..... Software Development Kit

SoC ..... System on Chip

IP ..... Intellectual Property (semiconductor)

CLB ..... Configurable Logic Block

IOB ..... Input Output Block

LUT ..... Look-Up Table

FF ..... Flip-Flop

SR ..... Shift Register

DSP ..... Digital Signal Processing

RAM ..... Random Access Memory

BRAM ..... Block of RAM

FIFO ..... First-In First-Out

LFSR ..... Linear Feedback Shift Register generator

LCG ..... Linear Congruential Generator

MT ..... Mersenne Twister

TGFSR ..... Twisted Generalized Feedback Shift Register

CA ..... Cellular Automata

BBS ..... Blum Blum Shub generator

PLL ..... Phase-Locked Loop

RO ..... Ring Oscillator Generators

VCO ..... Voltage Controlled Oscillator

PS ..... Peripheral System

PL ..... Programmable Logic

AXI ..... Advanced eXtensible Interface

UART ..... Universal Asynchronous Receiver Transmitter

DMA ..... Direct Memory Access

S2MM ..... Slave to Memory Map

MM2S ..... Memory-Map to Slave

GE ..... Gate Equivalent

P&R ..... Place and Route

Gbps ..... Gyga Byte Per Second

I

General Introduction

INTRODUCTION

MOTIVATION AND PROBLEM STATEMENT

Despite its long history, random generation still remains a hot topic, with the emergence of the so-called Random or Entropy as Service [1] needs. It also becomes a key element in lightweight security core for IoT devices. Despite the common use of these generators in many applications as described above, their integration into System on Chip becomes highly desirable, particularly for IoT and Smart Cards. Therefore, the practical purpose of current research works in this field is to provide compact, with high throughput, secure, and reconfigurable pseudorandom generators for hardware applications.

Let us recall that a random number generator algorithm can be defined by the state space S of the generator, the transition mapping function f, the output extractor function g from a given state, and the seed x^0.The random output sequence is $y^1, y^2, \ldots$, where each y^t is generated by the two main steps described thereafter. The first step applies the transition function according to the recurrence $x^{t+1} = f(x^t)$, where x^t and x^{t+1} both belong to S. The mapping function f can be either an algorithm that deterministically produces random-like numbers in a discrete and finite state space. Such generators are denoted as pseudorandom number generators (PRNGs). On the opposite, f can be based on a physical source of entropy to produce randomness, thus making S a continuous space. The whole approach is thus called a "True" random number generator (TRNG). The second step consists in applying the function generator to the new internal state leading to the output y^t, that is, $y^t = g(x^t)$. There is a large variety of such recursive generators, which can be either linear or not, chaotic, and so on.

Pseudorandom number generation is more studied in mathematics and for software aspects, whereas hardware and semiconductor solutions are deeply investigated for true random generation. On the one hand, linear PRNGs (LPRNG) are a special case of linear recurrence modulo 2 (that is, S is $\mathbb{F}_2$). Many research works and solutions are regularly proposed to increase their performance and statistical profile, and their linearity and security are investigated accordingly. Unfortunately, only a few of these linear PRNGs are analyzed in details at the hardware level, such as FPGA and ASIC. On the other hand, chaotic pseudorandom number generators (CPRNGs) are non-linear generators of the form: $x^0 \in \mathbb{R}$ and $x^{t+1} = f(x^t)$, where f is a chaotic map. They are an attractive application of the mathematical theory of chaos. Reasons explaining such an interest encompass their sensitivity to initial conditions, and their unpredictability. Truly chaotic generators are a good demonstration of these characteristics: their period is infinite, hardware resources are compact, and statistical tests are often succeeded quite reasonably [2, 3].

One natural question that arises is: how can we inject disorder in a deterministic digital system, while respecting the mathematical definitions of chaos provided by Devaney [4] on such finite state machines? A usual answer in digital embedded systems is to consider pseudo-chaotic generators instead of truly chaotic ones [5, 6]. In spite of the quality of the

TRNG output based on a chaotic phenomenon, most of these techniques are however produced in a manner that is either slow (*i.e*, in a range of some Kbps to Mbps, to extract noise or jitter from a given component [7]) or costly (*e.g.*, extracting or measuring some noise using oscilloscope or laser [3]). Additionally, to embed these TRNGs in a pure digital platform is an extreme challenge, where the main concern is calibration of the bias phenomenon coming from analog inputs. Digital TRNGs lead thus to an uncontrollable uniformity and performance of the outputs compared to the theory. Conversely, chaotic PRNG (CPRNG) appears as a convenient solution in SoC platforms such as Zynq based FPGA [8]. However, due to the finite precision and quantization of floating point numbers, this latter may exhibit both deflated periods and non uniformly distributed outputs. Additionally, these PRNGs have various drawbacks, particularly they fail some statistical tests, and from a cryptographer point of view, chaos is not related to security [9]. Thus, avoiding floating approximation and its consequences is a major research objective, which has been investigated in various state-of-the-art proposals.

A recent software approach has been developed within the DISC department of the FEMTO-ST Institute. Formally speaking, this is a random walk in the graph of iterations of a specific binary function. The direction to take and the path length are defined by the embedded generator(s). Practically, it can be seen as post processing treatment which adds chaos (as defined by Devaney) to the embedded PRNG. A first application of such an approach was presented in the PRNG framework, leading to the so-called chaotic iterations based pseudorandom number generators (CIPRNG, [10, 11]). Since then, various improved versions have been proposed, one of them being designed specifically for FPGAs.

The objective of this thesis is to study the approach of generating pseudorandom numbers using chaotic iterations, in order to present the widest possible application coverage in terms of hardware implementation. Our interest focuses on using our skills on hardware/-software design with FPGA and ASIC facilities in Microelectronics department of CDTA research center to integrate, and implement on SoC/FPGA/ASIC, new chaotic iterations process as random number generator. In other words, the goal is to propose a series of chaotic post processing of pseudorandom generator, which increases their statistical proprieties, adds chaos, while preserving a large throughput, being cryptographically secure on hardware and software supports, and finally independent on the technology.

Requirements and Specifications

A number of specifications are considered for our research in the field of hardware random number generators. They are summarized as follows:

- An initial FPGA implementation of CIPRNG has been proposed in previous research [11], where the generator is based on PRNGs that have already been proven their cryptographic security and their good behavior faced to statistical tests (BBS, ISAAC).
- Most statistical test analyses of the proposed CIPRNGs are only executed on software level, with a basic FPGA implementation.
- As already stated, PRNGs based on chaotic iterations (CIPRNGs) use a PRNG as a strategy to select which bit(s) are to be iterated. These PRNGs are weakly

investigated and analyzed at hardware level. Moreover, most of them do not pass statistical tests.

- Most ASIC implementations of random number generators are based on true random number generators (TRNG), which use physical sources (laser, transistors, noise, ...). Conversely, pseudorandom generators are also weakly implemented, and have difficulties to pass statistical tests.
- Finally, none of hardware chaotic PRNGs on FPGA can pass the reputed BigCrush statistical test from TestU01 (319 tests), with the exception of our CIPRNGs.

Therefore, the new hardware chaotic PRNGs need to establish these requirements:

- Inject most of the theory of chaotic iterations on digital system, where only fixed point representation and positive numbers are considered.
- The hardware implementations are technology-independent (no DSP or bloc memories) and are easy to integrate on system on chip for FPGA and ASIC application.
- A high throughput, small area, and low power consumption are required.
- Various range of data width (8, 16, 32, and 64 bits), period length, and k-dimension chaotic PRNG.
- Finally, a high rate of statistical test success including the hardest ones (e.g., BigCrush of TestU01).

Contribution of the Thesis

This manuscript reports the design and evaluation of (generalized) Chaotic Iterations based PRNGs as a possible post-processing for hardware PRNGs, demonstrating its benefits compared to other linear and chaotic PRNGs. This proposal focuses on adding chaos on linear PRNGs as a post-processing, in which at each iteration, only a subset of components of the iteration vector is updated.

Our contributions in this thesis are summarized as follows:

- It presents a survey of a large set of selected hardware implementations of random number generators on FPGA. Both pseudorandom and true random generators are investigated, while linear and non-linear generators are discussed in the PRNG case. Each approach is explained in details, and a discussion of the results on both implementations and statistical tests are systematically given. Performance with respect to frequency, area size, weaknesses, and statistical evaluations are presented, if they are available.
- In order to investigate the strategy properties, 18 linear PRNGs belonging to 4 families (xorshift, LFSR, TGFSR, and LCG) have been physically implemented in FPGA and compared in terms of area, throughput, and statistical tests. Therefore, two design flows of conception are used for Register Transfer Level (RTL) and High-level Synthesis (HLS). Based on this study, the relations between linear complexity,

seeds, and arithmetic operations on the one hand, and the resources deployed in FPGA on the other hand, are deeply investigated. To the best of our knowledge, no published work has really deeply investigated hardware implementations of such linear PRNGs.

- It provides two FPGA test platforms based on Zynq and AXI interconnection BUS. They are used for implementation and randomness tests. Additionally, ASIC implementations are proposed using UMC-65nm Low Leakage Technology and Cadence tools.

- Implementation and tests of these new families of post-processing PRNGs are proposed based on chaotic iterations for FPGA and ASIC. The first one is an update of CIPRNG [10], in which three CIPRNG variants for FPGA have been designed, namely the XOR-CIPRNG, the CIPRNG-MultiCycle, and Multi-Cycle Multi-Dimension (CIPRNG-MCMD) (see [12]). These hardware pseudorandom number generations can reach a very large throughput/latency ratio.

- A new chaos-based pseudorandom number generator implemented in FPGA, which is mainly based on the deletion of a Hamilton cycle within the N-cube (or on the vectorial negation) plus one single permutation, is detailed. By doing so, the obtained generator has a better statistical profile than its input, while running at a similar speed. This generation can also reach a very large throughput/latency ratio.

This thesis has led to the submission and/or the publication of the following articles [12–14].

Peer-reviewed International Journals

- Bakiri Mohammed, Jean-François Couchot, Christophe Guyeux. "CIPRNG: A VLSI Family of Chaotic Iterations Post-Processings for $\mathbb{F}_2$-Linear Pseudorandom Number Generation Based on Zynq MPSoC", IEEE Transactions on Circuits and Systems I: Regular Papers (2017), vol.PP, no.99, pp.1-14. doi:10.1109/TCSI.2017.2754650, Accepted (12 Septembre 2017)

- M. Bakiri, C. Guyeux, J.-F. Couchot, and A. K. Oudjida, “Survey on hardware implementation of random number generators on fpga: Theory and experimental analyses,” Computer Science Review, vol. 27, pp. 135–153, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013716302271

Peer-reviewed International Conferences

- Mohammed B., Couchot J. and Guyeux C. (2016). "FPGA Implementation of F2-Linear Pseudorandom Number Generators based on Zynq MPSoC: A Chaotic Iterations Post Processing Case Study". In Proceedings of the 13th International Joint Conference on e-Business and Telecommunications - Volume 4: SECRYPT, (ICETE 2016) ISBN 978-989-758-196-0, pages 302-309. DOI: 10.5220/0005967903020309

- Mohammed B., Couchot J. and Guyeux C. (2017). "One random jump and one permutation: sufficient conditions to chaotic, statistically faultless, and large throughput PRNG for FPGA". .In Proceedings of the 14th International Joint Conference on e-Business and Telecommunications - Volume 6: SECRYPT, (ICETE 2017) ISBN 978-989-758-259-2, pages 295-302. DOI: 10.5220/0006418502950302

National Conferences and Congress

- Bakiri Mohammed, Couchot Jean-François and Guyeux Christophe. "FPGA and ASIC Implementation of a Pseudorandom Number Generator based on Chaotic Iterations". XIIème Colloque du GDR SoC-SiP, 14-16 June 2017, Bordeaux, France

Other Peer-reviewed paper

- A. K. Oudjida, A. Liacha, M. Bakiri and N. Chaillet, "Multiple Constant Multiplication Algorithm for High-Speed and Low-Power Design," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 63, no. 2, pp. 176-180, Feb. 2016. doi: 10.1109/TCSII.2015.2469051

- Ahmed Liacha, Abdelkrim K. Oudjida, Farid Ferguene, Mohammed Bakiri, and Mohamed L. Berrandjia, "Design of High-Speed, Low-Power, and Area-Efficient FIR Filters ," in IET Circuits, Devices & Systems, ACCEPTED MANUSCRIPT, 23/08/2017, DOI: 10.1049/iet-cds.2017.0058, Print ISSN 1751-858X, Online ISSN 1751-8598

- G.Abdellaoui, S.Abeb, A.cheli, H.Adams, d.Ahmad, A.hriche, J-N.Albert, .D.Allard, A.lonso, L.Anchordoqui, V.Andreev, A.Anzalonel, W.Aouimeur, Y.Arain, N.Arsene, K. Asan, R.Attallah, H.Attoui, M.Ave Pernas, S.Bacholle, M.Bakiri, et al, "Meteor studies in the framework of the JEM-EUSO program", Planetary and Space Science 143 (2017): 245-255.

- G.Abdellaoui, S.Abeb, A.cheli, H.Adams, d.Ahmad, A.hriche, J-N.Albert, D.Allard, A.lonso, L.Anchordoqui, V.Andreev, A.Anzalonel, W.Aouimeur, Y.Arain, N.Arsene, K. Asan, R.Attallah, H.Attoui, M.Ave Pernas, S.Bacholle, M.Bakiri, "Cosmic ray oriented performance studies for the JEM-EUSO first level trigger." Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment (2017).

Organization of the Thesis

The remainder of this manuscript is divided in four parts, as detailed below.

The first part consists of two scientific background chapters. The first chapter describes a state-of-the-art of random number generators, from mathematical considerations to FPGA implementation. Linear and non-linear PRNGs are presented, and true random ones too. Additionally, recalls regarding statistical batteries of tests are given, while scores of some well-known RNGs on FPGA are summarized at the end of this survey. In

the second chapter, an overview of the mathematical foundation concerning chaotic iterations based PRNGs, which are the main objects regarded during our thesis, is provided. In particular, we will recall two theoretical schemes of these discrete dynamical systems: the unary scheme [15] and the generalized one [16].

In the second part, Chapter 3 will analyze the FPGA implementation of a set of selected linear pseudorandom number generators. Frequency, area size, weaknesses, and computation complexity are investigated to select which linear PRNGs can be used as a strategy for our post-processing. Then, Chapter 4 reviews two hardware platforms used for all our implementations and test comparison: FPGA and ASIC.

The third part focuses on adding chaos on linear PRNGs studied in Chapter 3 as a post-processing (CIPRNG), in which at each iteration, only a subset of components of the iteration vector is updated. Therefore, three CIPRNGs (namely: CIPRNG-XOR, CIPRNG Multi-Cycle, and CIPRNG Multi-Cycle Multi Dimension) are reviewed, implemented, and tested on FPGA and ASIC. The final part (Chapter 6) describes our new proposed design for a new chaotic PRNG, targeting FPGA and ASIC implementations. It resumes new iterative functions based on chaotic iterations whose graph of generalized iterations is strongly connected and which has been obtained by removing a balanced Hamiltonian cycle in a N-cube following the method suggested in [16].

Finally, this manuscript ends by a conclusion section, in which the contribution is summarized and the intended future work is outlined.

ABBREVIATIONS FOR RANDOM NUMBER GENERATORS

Abbreviation	Definition
RNG	Random Number Generator
TRNG	True Random Number Generator
PRNG	Pseudo Random Number Generator
LPRNG	Linear Pseudo Random Number Generator
CPRNG	Chaotic Pseudo Random Number Generator
CSPRNG	Cryptographically Secure Pseudo Random Number Generator

II

Scientific Background

1

Random Number Generators on FPGA

This chapter is a comprehensive survey on random number generators implemented on Field Programmable Gate Arrays (FPGAs). A rich and up-to-date list of generators specifically mapped to FPGA are presented with deep technical details on their definitions and implementations. A classification of these generators is presented, which encompasses linear and nonlinear (chaotic) pseudo and truly random number generators. A comparison of their statistical evaluation through usual batteries of tests and of their area and speed performances is finally outlined. This chapter is mainly issued from an article in submission to ACM Elsevier Computer Science Review.

1.1/ General presentation

Randomness is a common word used in many applications [17] such as simulations [18], numerical analysis [19], computer programming, cryptography [20], decision making, sampling, etc. The general idea lying behind this generic word most of the times refers to sequences, distribution, or uniform outputs generated by a specific source of entropy. In other words, the probabilities to generate the same output are equal (50% to have "0" or "1"). If we take the security aspect, many cryptosystem algorithms rely on the generation of random numbers. These random numbers can serve for instance to produce large prime numbers which are at the origin of cipher key construction [21] (for example, in RSA algorithm [22], in Memory Encryption [23] or Rabin signatures [24]). Furthermore, when the generators satisfy some very stringent properties of security, the generated numbers can act as stream cyphers in symmetric crytosystems like the one-time pad, proven cryptographically secure under some assumptions [25]. Randomization techniques are especially critical since these keys are usually updated for each exchanged message. Even if an adversary has partial knowledge about the random generator, the behavior of this latter should remain unpredictable to preserve the overall security.

From a historical point of view, numerical tables and physical devices have provided the first sources of randomness designed for scientific applications. On the one hand, random numbers were extracted from numerical tables like census reports [26], mathematical tables [27] (like logarithm or trigonometric tables, of integrals and of transcendental functions, etc.), telephone directories, and so on. On the other hand, random numbers were extracted also from some kind of mechanical or physical computation like the first

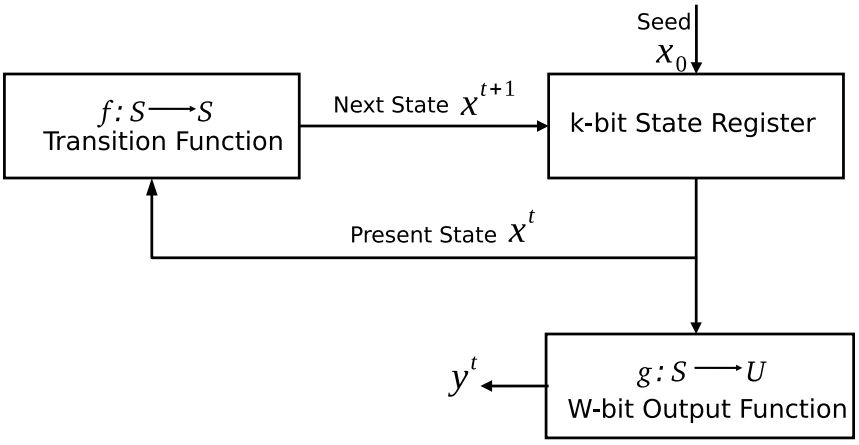


Figure 1.1: General random number generator architecture

machine of Kendall and Babington-Smith [28], Ferranti Mark 1 computer system [29] that uses the resistance noise as a physical entropy to implement the random number instruction in the accumulator, the RAND Corporation [30] machine based on an electronic roulette wheel, or ERNIE (Electronic Random Number Indicator Equipment [31]), which was a famous random number machine based on the noise of neon tubes and used in Monte Carlo simulations [32, 33].

These techniques cannot satisfy today's needs of randomness due to their mechanical structure, size limitation when tables are used [27], and memory space. Furthermore, it may be of importance to afford to reproduce exactly the same "random sequence" given an initial condition (called a "seed"), for instance in numerical simulations that must be reproducible – but physical generation of randomness presented above does not allow such a reproducibility. With the evolution of technologies leading to computer machines, researchers start searching for low cost, efficient, and possibly reproducible Random Number Generators (RNGs). This search historically began with John von Neumann, who presented a generation way based on some computer arithmetic operations. Neumann generated numbers by extracting the middle digits from the square of the previously generated number and by repeating this operation again and again. This method called mid-square is periodic and terminates in a very short cycle. Therefore, periodicity and deterministic outputs that use an operator or arithmetic functions are the main difference with the earlier generators. They are known in literature as "pseudorandom" or "quasirandom" number generators (PRNGs), while circuits that use a physical source to produce randomness are called "true" random number generators (TRNGs).

In most cases a random number generator algorithm can be defined by a tuple (S, f, g, U, x^0), in which S is the state space of the generator, U is the random output space, $f : S \to S$ is the transition mapping function, $g : S \to U$ is the output extractor function from a given state, and x^0 is the seed [34], see Figure 1.1. The random output sequence is $y^1, y^2, \ldots$, where each $y^t \in U$ is generated by the two main steps described thereafter. The first step applies the transition function according to the recurrence $x^{t+1} = f(x^t)$, where f is an algorithm in the PRNG case and a physical phenomenon in the TRNG one. Then, the second step consists in applying the function generator to the new internal state leading to the output x^t, that is, $y^t = g(x^t)$. The period of a PRNG is the minimum number of iterations needed to obtain twice a given output (a PRNG being deterministic, it always finishes to enter into a cycle).

As stated previously, the old hardware manner to build such RNGs was to use a mechanical machine or a physical phenomenon as entropy source, which can thus be based on noise [35], metastability (frequency instability [36]), semiconductor commercial or industrial component circuit (PLL [37], amplifier, filters [38], inverter,...), or a variation in the CMOS/MEMS process technologies (transistor). In spite of the quality of the generated randomness, most of these techniques are however, either slow processes (i.e, extracting noise from a component) or costly (e.g., extracting or measuring noise may require specific equipment like an oscilloscope). All these previous drawbacks are the motivation behind the development of hardware generators based on a software design. The latter consist of developing deterministic algorithms by targeting a specific hardware system, like a Field Programmable Gate Array (FPGA), before automatically deploying it on the hardware architecture by using ad hoc tools.

FPGA devices are reconfigurable hardware systems. They allow a rapid prototyping, i.e., explore a number of hardware solutions and select the best one in a shorter time [39]. The design methodology on FPGA relies on the use of a High Description Language (i.e, Verilog, VHDL, or SystemC) and a synthesis tool. Because of this, FPGA has become popular platforms for implementing random generators or complete cryptographic schemes, due to the possibility to achieve high-speed and high-quality generation of random. The general architecture of a FPGA presented in Figure 1.2 is based on LCA (Logic Cell Array), which is composed of three parts, namely: Configurable Logic Block (CLB) [40], Input Output Block (IOB), and interconnect switches. FPGA could additionally include more complex components like a Digital Signal Processing (DSP), a Random Access Memory (RAM), a Digital Clock Manager (DCM), or an Analog-Digital Converter (ADC/DAC). The nomination of the internal blocks depends on the FPGA vendors (Xilinx, Altera, Actel ...) even they have a similar functionality. The CLB structure is mainly based on Look-Up Tables (LUTs [41]), additionally with a Flip-Flop and some multiplexers. A K-input LUT is a $2^K \times 1$-bit memory array based on a truth table of K-bits inputs. These later can executes any logic functions as XOR/ADD/SHIFT...

Different implementations of RNG on FPGA have diverse characteristics. First of all, does it provide true random or pseudorandom numbers? In the second reproducible case, which algorithm is implemented? The next characteristic is the way each block is deployed on the FPGA, namely by computing or in a hardware manner. For instance, for a polynomial division, there is a choice between look-up table in software and a hardware shift register. Furthermore, the quality of the FPGA model that implements a random number generator can be evaluated according to many criteria. In a statistical perspective, the output has to be verified against some well-known test suite like the NIST [42], DieHARD [43], or TestU01 [44] ones. From the hardware perspective, one objective is to provide the highest frequency per randomly generated bit with less FPGA hardware resources (CLB, IOB, ...).

This chapter surveys a large set of selected hardware implementations of random number generators on FPGA. Both pseudorandom and true random generators are investigated, while linear and non-linear generators are discussed in the PRNG case. Each approach is explained in details, and a discussion on the choices of both implementations and generations are systematically given. Performance with respect to frequency, area size, weaknesses, and statistical evaluations are finally presented, when they are available.

The remainder of the chapter is as follows. Section 1.2 describes FPGA implementation of linear PRNGs (LPRNG), whereas the next section 1.3 focuses on non-linear ones

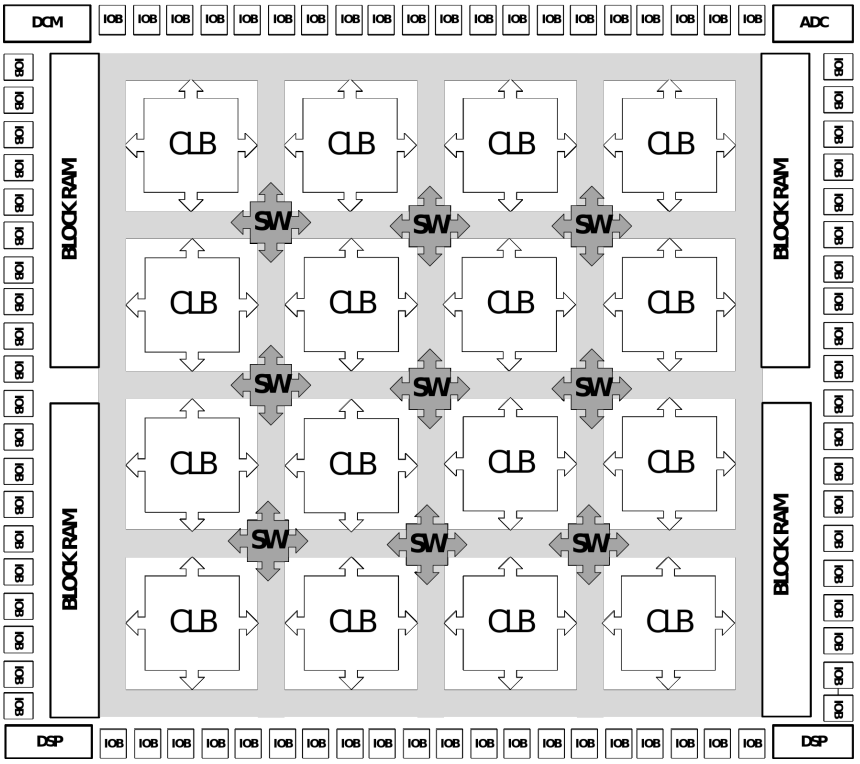


Figure 1.2: General structure of a FPGA by Xilinx

(CPRNG). The true random ones are detailed in Section 1.4. Each of these three sections ends by a FPGA implementations comparison regarding area resources and throughput (see Section 1.5). While, we recalls statistical batteries of tests and scores of some RNGs on FPGA are provided in Section 1.6. This chapter ends by a conclusion section, in which the review is summarized and future investigative directions are outlined.

1.2/ Linear Pseudorandom Number Generators

This section and the next one are devoted to pseudorandom number generators on FPGA. Recall that the latter are defined by a tuple containing a recurrent equation of the form $x^{t+1} = f(x^t)$. This recurrence may be linear or not. The linear case is investigated in the current section, while the non-linear case is detailed in Section 1.3.

Linear PRNGs are a special case of linear recurrence modulo 2. They are convenient for low power and high speed requirement but, due to the limitation of the shift register state (two possibilities: 0 and 1), the period of these generators is usually short. Because of this, many hardware optimizations are proposed to increase the period (they will be detailed thereafter). A linear PRNG of w bits can be defined by the following Equations (1) [45]:

$$\begin{aligned} x^{t+1} &= A \times x^t \quad (a) \\ y^t &= B \times x^t \quad (b) \\ r^t &= \sum_{\ell=1}^{w} y^t_{\ell-1}\, 2^{-\ell} (c) \end{aligned} \tag{1}$$

Indeed the first equation (a) defines the function f, where $x^t = (x^t_0, \ldots, x^t_{k-1}) \in S = \mathbb{F}_2^k$ is

the k-bit vector at step t ($\mathbb{F}_2$ is the finite field of cardinality 2 and S is the internal state space of the generator). The other equations (b) and (c) define the function g, where $y^t = (y_0^t, \ldots, y_{w-1}^t) \in U = \mathbb{F}_2^w$ is the w-bit output vector at step t, and U is the state space of the output. Additionally, A is a $k \times k$ transition matrix, B is a $w \times k$ output transformation matrix, which produces the output bits which corresponds to the internal RNG state, and $r^t \in [0, 1]$ is the output at step t. All the elements of A and of B are in $\mathbb{F}_2$.

In the simplest case we have $w = k$ and B is the identity matrix, which means that the state bits are directly used as random output bits. In case where $w < k$, the output are either propagating in another circuit, or multiple state bits are *XOR*ed together to produce each output bit, as in the case of Mersenne Twister [46]. These linear generators are covering Tausworthe or Linear Feedback Shift Register [47], polynomial Linear Congruential Generators [48], Generalized Feedback Shift Register (GFSR [49]), twisted GFSR [50], Mersenne Twister, linear cellular automaton, and combinations between them. More details will be presented regarding each of these generators in this survey.

1.2.1/ Linear Congruential Generators

Linear Congruential Generators (LCGs) [48] are founded on system of linear recurrence equations defined as:

$$x^{t+1} = (ax^t + b) \mod 2^k, \tag{2}$$

where a (the "multiplier"), b (the "increment"), s.t. $0 \leq a, b \leq 2^{k-1}$ are parameters of the generator,

This latter is often called a Multiplicative Congruential Generator [51] (MCG) if $b = 0$, and Mixed Linear Congruential Generator, otherwise.

In [52], two optimized LCGs are proposed, namely the Ranq1 and Ran [53]. Ranq1 is a MCG working modulo 2^{64}, while its seed is produced by a 64-bits right *XOR*shift [54]. Let us first recall that the *XOR*shift takes an input and iteratively executes an exclusive or (XOR) of the binary number with a bit shifted translation (left and right) of itself. The second one, the Ran generator, combines a LCG generator with two *XOR*shifts, and the results are *XOR*ed by a Multiply with Carry (MWC) generator [55]. In MWC, the equation (2) is modified as follows: the constant b is replaced by the carry b^t which is defined by b^0, the initial carry, is less than a and $b^{t+1} = \lfloor \frac{ax^t + b^t}{2^{32}} \rfloor$.

Authors of [52] optimized the implementation of the 64-bits constant coefficient multiplier $a \times x^t$. However the 64-bit multiplication is problematic due to DSP macro limitations that support only 18-bit operations in Xilinx's FPGA. This is why these authors proposed a pipeline of multiplier-adder architecture, which takes 5 cycles for Ran and 4 for Ranq1, while the output is the least significant 32 bits. Comparisons realized in their article showed that these two new optimized implementations have a lower cost in the area than other PRNGs like the Mersenne Twister [46], which use memories or multiplier macros of the FPGA. But the authors were wrong when they assumed that the multiplication by a constant is similar to the multiplication by a variable. In the former (Oudjida et al [56–58]), the multiplication is implemented in a multiplierless way, i.e., using only additions, subtractions, and left shifts.

Authors of [59] have presented a coupling of two Coupling Linear Congruential Generators (CLCG), further denoted as CLCG-1 and CLCG-2. Each one generates a separate output with different parameters described as follow:

$$\begin{aligned} x_1^{t+1} &= (a_1 x_1^t + b_1) \mod 2^k \\ x_2^{t+1} &= (a_2 x_2^t + b_2) \mod 2^k \\ C_1^{t+1} &= \begin{cases} 1 & \text{if } x_1^{t+1} \geqslant x_2^{t+1} \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{3}$$

$$\begin{aligned} x_3^{t+1} &= (a_3 x_3^t + b_3) \mod 2^k \\ x_4^{t+1} &= (a_4 x_4^t + b_4) \mod 2^k \\ C_2^{t+1} &= \begin{cases} 1 & \text{if } x_3^{t+1} \geqslant x_4^{t+1} \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{4}$$

The first CLCG-1 of [59] is characterized by $\{x_1^{t+1}, x_2^{t+1}, C_1^{t+1}\}$ while the second CLCG-2 is defined with $\{x_3^{t+1}, x_4^{t+1}, C_2^{t+1}\}$ (C_1^t and C_2^t are bit sequences). CLCG-2 aims at selecting which bit must be taken from CLCG-1 as a final output y^t: $y^t = C_1^t$ if $C_2^t = 0$, otherwise the bit C_1^t is ignored. For instance, the authors assume a simple format of the multipliers: $a_1 = a_3 = 2^{\delta_1} + 1$ and $a_2 = a_4 = 2^{\delta_2} + 1$, where $1 < \delta_1, \delta_2 < k$. Indeed, the new format of x_1^{t+1} for CLCG-1 (and similarly for x_2, x_3, and x_4) is as follow:

$$x_1^{t+1} = ((2^{\delta_1} \times x_1^t \mod 2^k) + x_1^t + b_1) \mod 2^k, \tag{5}$$

where $2^{\delta_1} \times x^t$ is the result of shifting x^t exactly δ_1 times to the left, and the modulation is computed as the k least significant bits of what is in parentheses. However, a large value of k leads to a large latency. To solve this problem, an implementation of P stages of addition and comparison for the two CLCGs has been proposed in this article. It divides the k-bits numbers into P-pipeline parts, processes each k/P-bits part in a pipeline stage, and finally generates 1-bit of C_1 and C_2 simultaneously. Additionally, it takes the results of each stage and sends them to both the previous and the next stages, in order to produce the current and the next outputs.

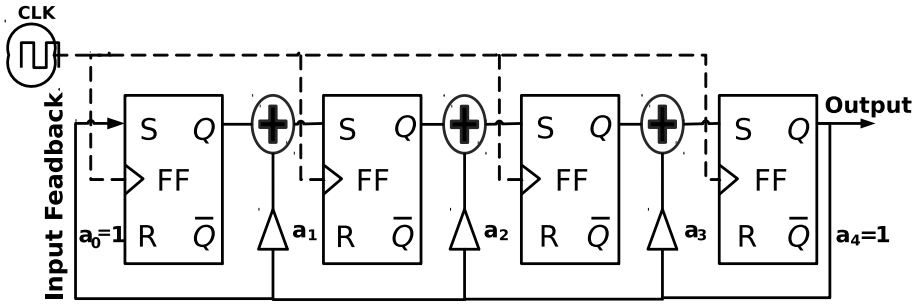
1.2.2/ Linear Feedback Shift Register generators

Linear Feedback Shift Register generators (LFSR) or Tausworthe [47] are linear recurrent generators. An LFSR uses a sequence of Flip-Flop (FF) as shift registers to generate one bit per iteration. Each register is connected to its neighborhoods, the binary value in each register is shifted at each iteration, while the last register produces the output (see Figure 1.3). A *XOR* is operated on some designed registers to build a feeadback input to the first register, which is expressed by a characteristic polynomial. As depicted in Figure 1.3, two configurations are usually considered, namely the Galois and Fibonacci setups. These two aforementioned implementations are synchronized with a main clock (CLK), in which at each edge the maintained data (1-bit) in FF is released and a new input is stored. The matrix A of Equation. (1) is in this case:

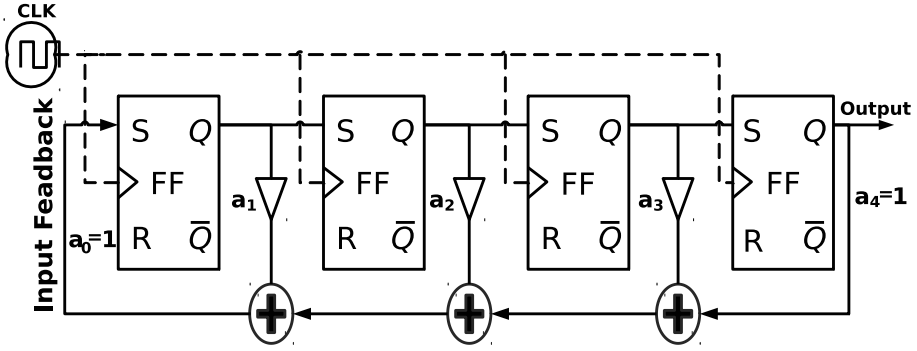
$$A = \begin{pmatrix} 0 & I_{k-1} \\ a_k & a_{k-1}, \ldots, a_1 \end{pmatrix}. \tag{6}$$

The characteristic polynomial of the matrix A is $x^{t+1} = a_1 x^t + \cdots + a_k x^{t+1-k}$. In the above equations, $a_1, \ldots, a_k$ represent the LFSR coefficients, each in $\mathbb{F}_2$. Accordingly, if any of these coefficients exists, it deploys a *XOR* operand on the output (remark that the matrix B of Equation (1) is the identity matrix I).

Even though many FPGA implementations of such LFSRs can be found in the literature, only few of them are really optimized for this architecture. In [60], the authors present two types of LFSRs. The first one, called Shrinking Generator (SG), it uses two LFSRs



(a) A 4-bits Galois LFSR architecture using Flip-Flop (FF)

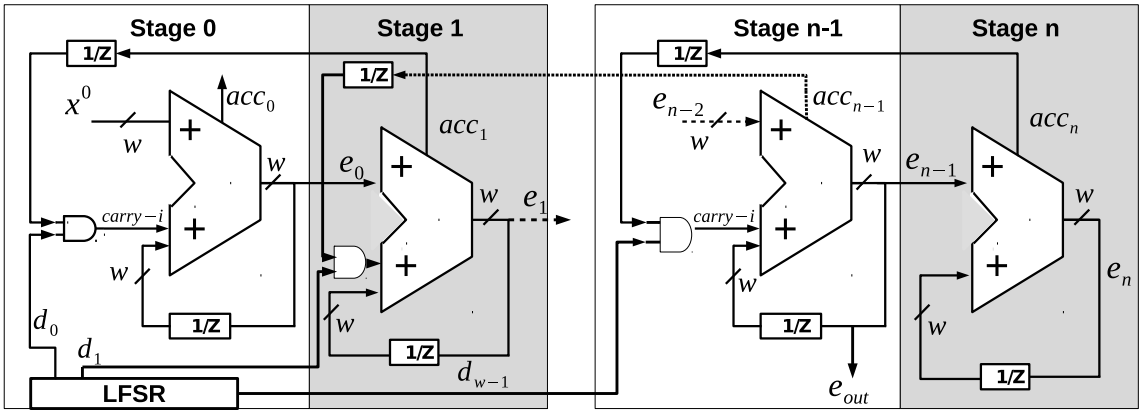


(b) A 4-bits Fibonacci LFSR architecture using Flip-Flop (FF)

Figure 1.3: A 4-bits linear feedback shift register generator with a feedback polynomial $a_0 * X^4 + a_1 * X^3 + a_2 * X^2 + a_3 * X + a_4$ $(a_0 = a_4 = 1)$.

of 67 bits (LFSR-1 and LFSR-2). At each clock cycle, the SG directly takes the value of the output bit which is generated by the second LFSR-2 if the output bit from the first LFSR-1 is equal to 1; if not, both outputs are discarded. The second version, named Alternating Step Generator (ASG), considers a third LFSR-3 of 141 bits in addition of the two previous ones. This latter is used to control which output bit will be taken from the two first LFSRs of $131/137$-bits. For comparison purposes, if T_1, T_2, and T_3 are the periods of LFSR-1, LFSR-2, and LFSR-3 respectively, let us note that the SG has a total period of $T_{SG} = (2^{T_1} - 1) \times (2^{T_2} - 1)$ (length $\simeq 64$ bits), while it is $T_{ASG} = 2^{T_1}(2^{T_2} - 1) \times (2^{T_3} - 1)$ for ASG (length $\simeq 128$ bits).

LFSR based Accumulator Generator proposed in [61] is a PRNG based on Digital Sigma-Delta Modulator [62] made from accumulator circuits. This latter is used to divide the frequency in a Fractional-n Frequency Synthesizer [63]. Authors of [61] propose a pipeline of $N = 9$ serial digital accumulators of $w = 8$ bits based FPGA as described in Figure 1.4. Each accumulator, which can produce $M = 2^w$ possible values, is a self-recursive structure based on quantization error mapping function formalized in Equation. (8), where the accumulator's feedback coefficients are time-varying, using another linear feedback shift register. The accumulator, which is presented in Equation (7), is parameterized by the input seed x^0, the accumulator sum p, the carry output *acc*, the quantization error e, and the feedback coefficients $c = 2^{-w}$ of the accumulators as outputs. The input of each $n = 0, \ldots, (N-1)$ stage during the processing uses the quantization error e^{t-1} of the previous stage. Therefore, the PRNG gives a better uniform outputs by propagating the latter (e) at all stages following the Equation (7). The accumulator feedback coefficient c



source [61]

Figure 1.4: Block-level model of a w-bit digital accumulator PRNG comprising n stages

is implemented with another accumulator. The latter are multiplied by a binary variable $d_w \in \{0, 1\}$ of the LFSR to control the feedback depending on the period of LFSR. The final output $y^t = e_{out}$ is the last generated e^t_{N-1} evaluated in Equation (8).

$$p_n^{t+1} = \begin{cases} x^0 + e_0^t + acc_1^t d_0^t, & n = 0 \\ e_{n-1}^{t+1} + e_n^t + acc_{n+1}^t d_{w-1}^{t+1}, & 0 < n < N-1 \\ e_{n-1}^{t+1} + e_n^t, & n = N-1 \end{cases} \tag{7}$$

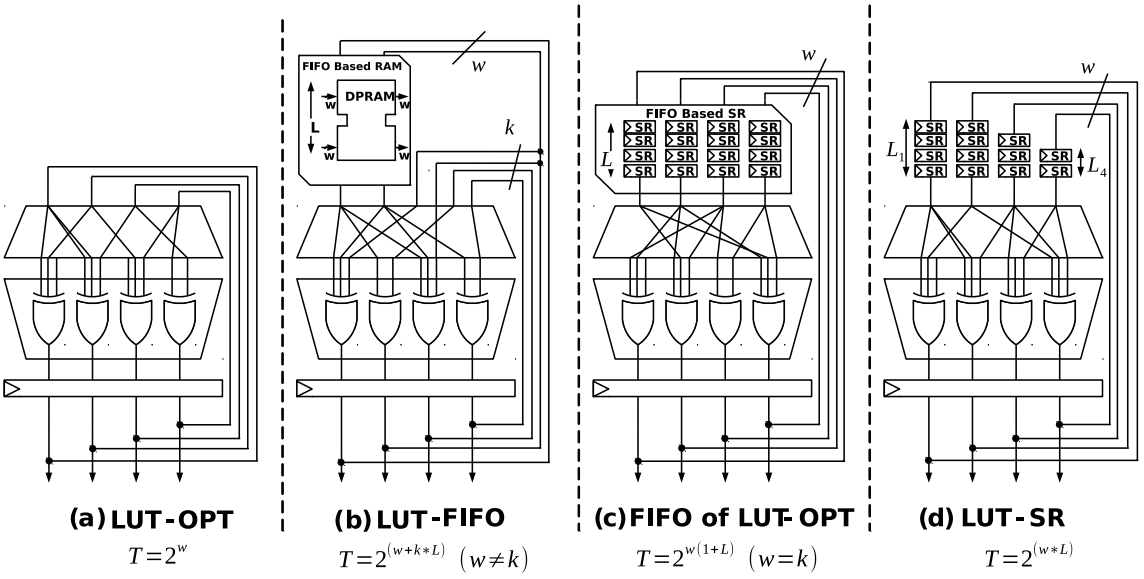
$$e_n^{t+1} = p_n^{t+1} \mod 2^w \text{ and } acc_n^t = \begin{cases} 1 & p_n^t \geq M \\ 0 & p_n^t < M \end{cases} \tag{8}$$

1.2.3/ Look-up Table Optimized Generators

Look-up Table (LUT [41]) optimized generators use logic block as a digital component defined in a CLB provided by the FPGA vendors. It is used for implementing many function and operation generators for a hardware optimization purpose. A LUT consists of a block of RAM (Random Access Memory) implemented as a truth table that is indexed by the LUT inputs.

In [64–66], the authors present a series of LUT PRNGs based on $\mathbb{F}_2$ linear matrix recursive algorithms (see Figure 1.5). The main idea is to produce a maximum efficiency at area level. The authors associate either Flip-Flops (FF), Shift Registers (SR), or block of RAMs (RAM) with LUT to perform shift/multiplication operations in FPGA. However, creating long period sequences $T = 2^w$ with this method is a difficult task. To solve this problem, large optimized LUT based {FF, SR, RAM} pairs are investigated.

The first proposed PRNG is called LUT-OPT (LUT optimized, (a) in Figure 1.5). It maps each row of the recurrence matrix A as a *XOR* gate using just LUT and FF. To generate w bits per cycle requires w LUT-FFs in a single LUT of k-bits during a period of $T = 2^w$ where $w = k$ (see Figure 1.5). Their estimations of the FPGA resources conclude that even if an application requires 64 bits for each cycle, their implementations necessarily use 512 LUT-FFs to produce a period of $2^{512} - 1$. The second one, the LUT-FIFO (b), is used to increase the period up to $T = 2^{(w+k*L)}$ without using the pair LUT-FF, which uses RAM



source [66]

Figure 1.5: LUT based shift-register and FIFO FPGA optimized PRNG: (a) maps each row of the recurrence matrix as a *XOR* gate using LUT-FF, (b) uses RAM block memory as $k \times k$ FIFO to store the recursive sequences, (c) loads the state in FIFO based shift-register SR instead of BRAM, (d) cascading of any number of Xilinx SRL32 to create a k-bit SR

block memory (dual-port RAM) of FPGA as $L \times k$ FIFO to store the recursive sequences. In this case, each new output bit is depending on one bit from the last iteration. They next propose a FIFO based shift-register SR (c) with a fixed length L of 1-bit, to load the w-bit state in parallel instead of using dual-port RAM. They also propose a LUT-SR PRNG (d) that turns the use of LUT as a k-bit Shift-Register using "Xilinx SRL32", with the length of each "SR" varying as follows: $1 < L_i < L$. The "Xilinx SRL32", allows the cascading of any number up to 32-bit shift registers to create a shift register with any size needed.

1.2.4/ TWISTED GENERALIZED FEEDBACK SHIFT REGISTER PRNG

Twisted Generalized Feedback Shift Register (TGFSR) proposed in [50] is an extension of Generalized Feedback Shift Register "GFSR" [49], which uses an array of shift registers to generate more than one bit for each state change. Therefore, a TGFSR is based on recurrence of N sequences of words, $x^0, \ldots, x^{N-1}$, each containing k-bits and two parameters, namely a bitmask size c, such that $c \leqslant k-1$ and a initial median position m with $1 \leq m \leq N$.

TGFSR computes the $t+N$-th word $(t = 0, 1, \ldots)$ by operating with three words: the first two words x^t and x^{t+1} with the median word x^{t+m}. More precisely (see Figure 1.6):

1. It computes the c least significant bits (LSB) of x^{t+1} and the $k-c$ most significant (MSB) ones of x^t. These two vectors are obtained thanks to the two following bit-mask vectors: $\overline{S_c}$ for $(0,\ldots,0,1,\ldots,1)$ and $\underline{S_{k-c}}$ for $(1,\ldots,1,0,\ldots,0)$.

2. These two vectors are further concatenated through $(x^t \& \underline{S_{k-c}}) \mid (x^{t+1} \& \overline{S_c})$.

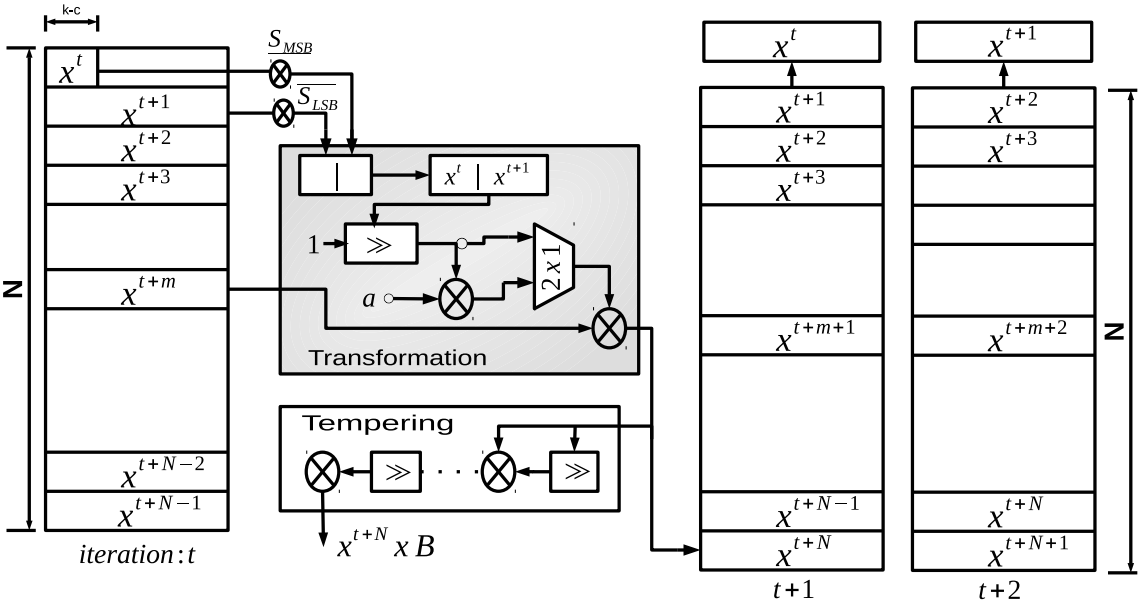


Figure 1.6: Twisted Generalized Feedback Shift Register architecture: at each recurrence operation t, it computes x^{t+N} thanks to the three words x^t, x^{t+1}, and x^{t+m} and generates the output with tempering function

3. The result x' is then "multiplied" with a matrix A, caracterized with values $(a_0, a_1, \ldots, a_{w-1})$ as defined in Equation. (10).

4. The final results of the previous calculations are then XORed with the median word x^{t+m}.

By putting $c = 0$, then the Equation (9) represents the TGFSR PRNG [50], conversely it is Mersenne Twister [46].

$$x^{t+N} = x^{t+m} \oplus (((x^t \& \underline{S_{k-c}}) \mid (x^{t+1} \& \overline{S_c})) \times A), \text{ where} \tag{9}$$

$$x' \times A = \begin{cases} x' \gg 1 & \text{if } x'_0 = 0 \\ (x' \gg 1) \oplus (a_0, a_1, \ldots, a_{w-1}) & \text{otherwise} \end{cases} \tag{10}$$

Consider c_1 and c_2 as given bitmasks and b_1, b_2, b_3, and b_4 are constant integer parameters. At each iteration t, the value x^{t+N} resulting of the recurrence Equation (9) serves as input of the tempering module of the TGFSR. This step improves the equidistribution of the output. This step, which is described a sequence of bitwise/shift computation is equivalent to a matrix product as formaized in Equation. (1)(b).

This one is defined in Equation (11) where c_1, c_2 (resp. b_1, b_2) are tempering bitmasks (resp. bit shifts).

$$\begin{aligned} z &= x^{t+N} \oplus (x^{t+N} \gg b_1), \\ z &= z \oplus ((z \ll b_2) \& c_1), \\ z &= z \oplus ((z \ll b_3) \& c_2), \\ y^t &= z \oplus (z \gg b_4). \end{aligned} \quad . \tag{11}$$

Mersenne Twister (MT) is proposed as a special case of TGFSR that has a long period of $2^{wN-c} - 1$. To achieve this, the authors in [46] propose two MT configurations:

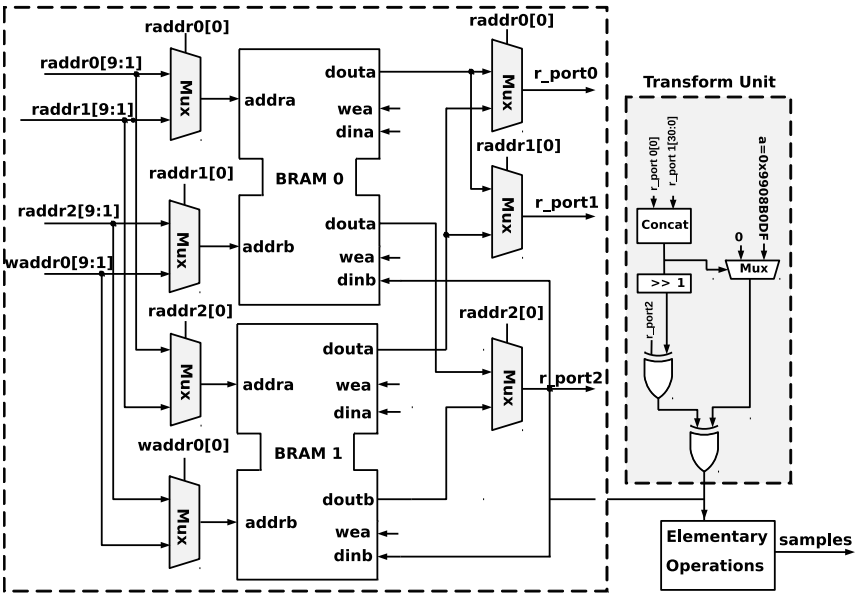
- "MT11213" with a period of $2^{11213} - 1$ that has $w = 32$, $N = 351$, $m = 175$, $c = 19$, and $a = 0xE4BD75F5$ as recurrence parameters, and $c_1 = 0x655E5280$, $c_2 = 0xFFD58000$, $b_1 = 11$, $b_2 = 7$, $b_3 = 15$, and $b_4 = 17$ for tempering ones
- "MT19937", which has a period of $2^{19937} - 1$, has $w = 32$, $N = 624$, $m = 397$, $c = 31$, and $a = 0x9908B0DF$ as recurrence parameters, and $c_1 = 0x9D2C5680$, $c_2 = 0xEFC60000$, $b_1 = 11$, $b_2 = 7$, $b_3 = 15$, and $b_4 = 18$] for Tempering ones.

Two FPGA implementations of Mersenne Twisters MT19937 and MT11213 are proposed in [52] for Monte Carlo applications in finance. The authors implement many Block RAM memory or namely "BRAM" for matrix multiplications: a single dual-port BRAM for MT11213 and two dual-port BRAM for MT19937. The RAM memory, configured in the read-before-write mode, operates like a feedback shift register. In this mode, the new inputs are stored in memory at appropriate write address, while the previous data are transferred to the output ports. This latter coming from BRAM are then processed following the Equation (6). The same approach has been proposed in [67] for MT19937 using 2 BRAM. Authors of [68], for their part, have implemented the MT11213 in three platforms for the sake of comparison, namely: FPGA, CPU, and GPU. Remark that, for testing the FPGA performances, initial and Tempering matrix parameters have been extracted from a PC software, due to the hardware cost consuming by the initialization stage of MT. However, both transformation and Tempering modules are executed in FPGA. In this case, two dual-port BRAMs are necessary for the other stages. This structure reduces the area compared to other MT implementations in FPGA, and the speed up is about $\approx 9\times$ and $\approx 25\times$ compared to GPU and CPU respectively.

In [69], the authors have proposed two parallel PRNG implementations with many levels of three different Mersenne Twisters: the MT19937-32bits, the MT19937-64bits, and the SIMD-oriented Fast Mersenne Twister SFMT19937 [70]. The first one is the Interleaved Parallelization (IP), that generates w-bits for each P memory block separately.

In the IP configuration, the $N = 624$-words state vector is located across multiple memory banks of the same size. Each P memory bank has d input/output ports I/O of w-bits, while each I/O port generates v-bits per clock cycle every q read operation. Therefore, the number of clock cycle τ required to generate a random number is equal to $\tau = (w * (q + 1))/v * d$. The second one is the Chunked Parallelization (CP), that uses the output bits of each RAM bank as the far recurrence input for the next RAM bank. Therefore, the N-words state vector is sequentially split into chunks across a number of banks of different size. Even though the IP version has a better throughput than the CP one, the latter uses lesser RAM blocks compared to the IP version (3 levels of CP use 2 BRAMs while IP uses 3).

Authors of [71] give more hardware details for the deployment of RAM memories. Their MT19937 implementation consists of a transform unit, a Temper Unit, a control unit "Crossbar" implemented using 7 multiplexers, a 3R/1W RAM, and an address unit for RAM access (3 read addresses and 1 address for writing). The main key of the latter is the implementation of 624 states of 32-bits register using BRAM memory of the FPGA (see Figure 1.7). Therefore, instead of fetching the 3 state vectors using 3 BRAM as in [69], two dual-port BRAMs of 312×32-bits can perform in each cycle 3 read operations and 1



source [71]

Figure 1.7: Mersenne Twister MT19937 architecture using 3R/1W BRAM: at each cycle, R/W address is even address for BRAM0 and odd for BRAM1

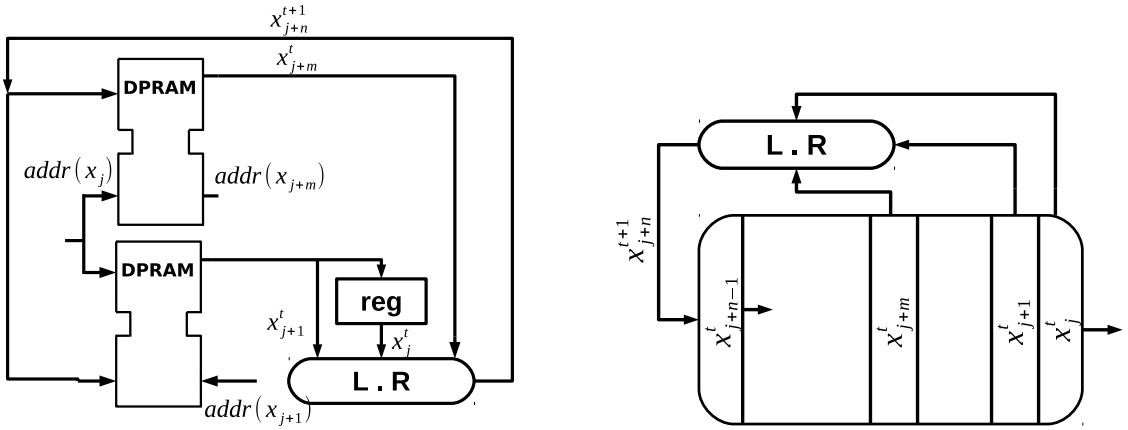
write one. The R/W for the first BRAM operates with an even address, while the second R/W is in the odd address.

In [72], various degrees of parallelization of the MT19937 architecture (of degrees 2, 3, 4, and 6) implemented in [71] and used for Monte-Carlo based simulations are proposed. In this case, the configuration of BRAM is the key of the parallelism, where each degree corresponds to the number of BRAM that are used (4 degrees = 4 implemented RAMs). The authors use one 206×32-bit, two 207×32-bit dual-port BRAM, and four registers to provide state consistency for the given parallelized states for 3 degrees as an example. Here, all I/O ports of three BRAM are in read mode during initialization, while in the runtime just one is in read mode (the others being in write mode).

Finally, a recent FPGA implementation of Mersenne Twisters is presented in [73]. The authors propose an alternative solution of the use of RAMs, which is named Circular Buffer (CB). It is implemented for MT19937 (see Figure 1.8). The solution is based on the fixed relationship between word indices. Words x_j^t, x_{j+1}^t, and x_{j+m}^t written in the buffer are passed to the transform unit. At each iteration, the first word x_j^t is clocked out of the buffer while new data x_j^{t+1} is written to the free location. By this way, the linear recurrence and the buffer of registers can be considered as a circular buffer. The linear recurrence is carried out by some combinational logic between the input and the output of the buffer. Therefore, the architecture is simplified since no logic operation for the table indices is needed.

1.2.5/ Cellular Automata based PRNGs

Cellular Automata (CA) is a discrete model, proposed by John von Neumann and Stan Ulam [74] as formal models of self-reproducing robots. The basic representation of one dimensional CA PRNG includes N cells with an internal state machine that can be a



(a) BRAM configured as 3R/1W (b) Circular Buffer

source [73]

Figure 1.8: Different deployments of the linear recurrence (L.R) for Mersenne Twister PRNG: (a) using BRAM configured as 3R/1W, (b) using Circular Buffer of registers (L.R is linear recurrence of transferring function of MT)

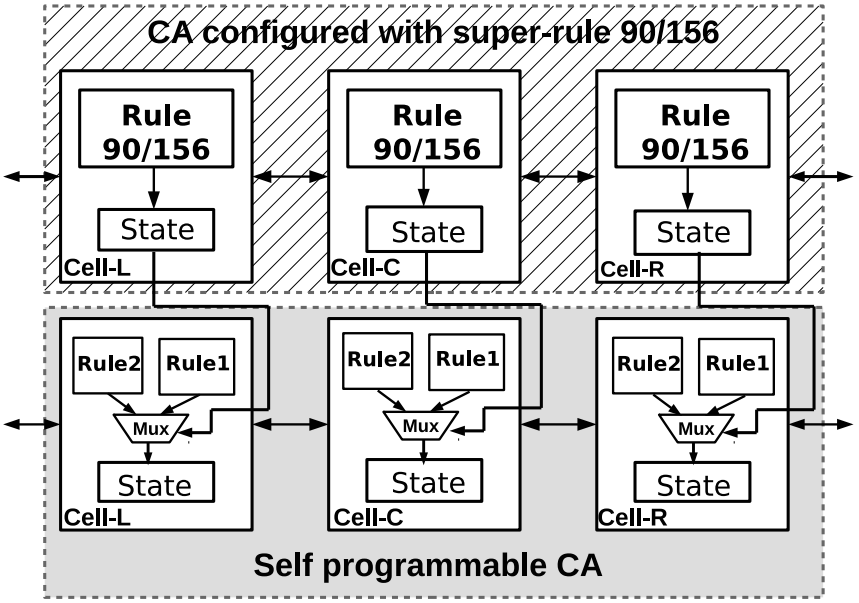
Boolean function rule and $k = 1$-bit output as described in Equation (12). The latter consider the function $f : \{0,1\}^N \rightarrow \{0,1\}$ as the local transition rule, and the cells neighborhood size N is $2*rd+1$, where rd is the radius that represents the standard 1-D cellular neighborhood. Therefore, at each iteration t, the CA structure can hold and update the internal state for each cell, depending on the local rules and the states $x^t \in \{0,1\}$ of their neighborhoods j ($j = 1,\ldots,N$). There are 2^N ($rd = 1$ and $N = 3$) states for a single CA producing 256 (2^8) possible rules classed by the Wolfram code [75].

$$x^{t+1}_j = f(x^t_{j-rd} \ldots x^t_j \ldots x^t_{j+rd}) \tag{12}$$

As an example, let us consider that $N = 3$, which leads to $x^{t+1}_j = f(x^t_{j-1}, x^t_j, x^t_{j+1})$. The 184 rule updates the middle bit x^t_j and then left shifts the input in the next iteration t as follows: $f(111) = 1$, $f(110) = 0$, $f(101) = 1$, $f(100) = 1$, $f(011) = 1$, $f(010) = 0$, $f(001) = 0$, $f(000) = 0$ (i.e. $11101011 \rightarrow 101011$).

Hybrid CA generator (HCA) is defined with more than one rule and can be integrated as a state transition machine of $2^N/2$ cycles between 2^N rules. Each transition cycle has a $2{\times}2^N$ length cycle. Two hybrid CAs are proposed in [76] as part of an encryption system. The first one is a PRNG of single state transition using rules 90, defined by $f(x^t_1, x^t_2, x^t_3) = x^t_1 \oplus x^t_3$, and 150, defined by $f(x^t_1, x^t_2, x^t_3) = x^t_1 \oplus x^t_2 \oplus x^t_3$ to generate an encrypting real-time key stream. The second one is a block cipher of two state transitions, each having 8 cycles length with $51/153/195$ rules. The aim of this application is to use the first HCA to select the transition sequences between rules used by each cell of the second HCA in the block cipher. The FPGA implementation of each CA is done with a logic combinational circuit (LCC) to define the rules. Then it uses LCG to control the loading operation of the CA and stores the data into a D flip-flop. Authors of [77], for their part, create an automatic software tool to generate the RTL code of any HCA configuration. Finally, in [78], authors increase the ratio of frequency/area and the security of their previous PRNG [77] by using a chain of HCAs instead of a single one.

Mixed CA generator is proposed in [79], where the author mixes the outputs of a 37-bits hybrid CA (rules 90 and 150) with a 43-bits LFSR to obtain a large period. However, some



source [81]

Figure 1.9: Self-Programmable cellular automata generator: uses a super-rule $90/156$ to dynamically determines when the rules have to change in each CA cell

drawbacks of this implementation are revealed during statistical evaluation, which can be surpassed only if the two PRNGs are clocked at different clock frequencies. This is why a new solution is presented in [80]. In this article, authors propose to *XOR* the last bit of HCA with the last bit of LFSR to generate 1-bit per clock cycle. As a repercussion, they found that the optimal combination for a PRNG of high quality is 16-bits for CA with a 37-bits LFSR.

Self-Programmable CA (SPA) was presented first as a new rules for CA generator in [81]. The topological behavior of the generator proposed in [82] is the use of a super-rule $90/156$ to dynamically determine when the rules have to change in each CA cell (see Figure 1.9). In practice, the input rules of each neighbor cell are also a second CA which is executed in parallel with the main cellular automata. Remark that, despite SPA gives a better throughput than the LFSR/HCA combination PRNG [80], it fails to pass the statistical tests of DIEhard battery.

Another cellular automata based PRNG is proposed in [83]. This latter combines a CA with a Non-LFSR (NSFR) generator based on A2U2 stream cipher design. Recall that the stream cipher A2U2 [84] was presented as a new key cryptographic generator of 56-bits for RFID tags application. It has a LFSR counter of 7-bits and two non-linear feedback shift registers (NFSRs, 17 and 9-bits). However, NFSR is known for its short period length. Hence, their main contribution is to associate a CA PRNG of 9-bits to increase the period of NFSR, both having feedback between them (which means that the seed of NFSR is provided by CA and vise versa). This approach improves resistance to various forms of cryptanalysis like correlation attacks and algebraic ones. For the sake of completeness, notice that the authors of [85] have proposed a different implementation concept of the usual rules in CA. In their proposal, the initial state configuration of CA and its length depend on the current date.

1.3/ Non-Linear Pseudorandom Number Generators

Chaotic generators (CPRNGs) are non-linear generators of the form $x^0 \in \mathbb{R}$: $x^{t+1} = f(x^t)$, where f is a chaotic map. They are attractive applications of the mathematical theory of chaos. Reasons explaining such an interest encompass their sensitivity to initial conditions, their unpredictability, and their ability of reciprocal synchronization [86]. From a cryptographer point of view, these chaotic PRNGs have major drawbacks often reported [9].

Chaotic Mapping PRNG are based on a polynomial mapping that uses a non-linear dynamic transformation, which is a quadratic mapping. Most of these generators are based on the Logistic Chaotic Map called also "LCG" map [87], defined as follows: $x^{t+1} = \alpha \times x^t(1 - x^t)$, where $0 < x^{t+1} < 1$ and α is the biotic potential $(3.57 < \alpha < 4.0)$. The logistic map is mainly depending on the parameter α: its chaotic behavior is lost when α is out of the range provided above. Moreover, if $\alpha > 4$ and for almost all initial values, the outputs diverge and leave the interval $[0, 1]$. The second most frequently used function is the Hénon chaotic map [88], which takes a point (x^t,y^t) within the plan square unit and maps it into a new point (x^{t+1}, y^{t+1}). This map is defined by these equations: $x^{t+1} = 1 + y^t - a(x^t)^2$ and $y^{t+1} = bx^t$, where a and b are called canonical parameters.

In [89], the authors have used fixed point representation [90] to implement the logistic map using Matlab DSP System Toolbox software. Fixed-point format is an approximation of real numbers, with much less precision and dynamique range than the floating-point format. Nevertheless, it has the merit of being very efficient in high-speed and low-power applications. This unit requires less power and cost to manipulate such kind of numbers than usual floating-point circuitry. They generate many designs with different lengths from 16 to 64 bits, where the resources are depending on the precision $(24$ to 53 bits). The multiplication is implemented with DSP blocks of FPGA that perform 18x25 bits multiplications, while the multiplication by a constant α is a simple series of add and left-shift operations.

Authors of [91] compare the implementation of logistic map with the Hénon one. Unlike the logistic map, the 64 bits multiplication in Hénon [88] map cannot be implemented with a left shift operation, which leads to the use of DSPs blocks of the FPGA for all multiplications needed to implement ax^2. Two optimized versions of PRNGs based on chaotic logistic map are proposed in [92], which aim to reduce resources and increase frequency, unlike in [89, 91]. The first one is based on LUT and DSP blocks of the FPGA. The second one rewrites the logistic map equation as follows: $x^{t+1} = \alpha x^t - \alpha(x^t)^2$. The objective of these two PRNGs is to pipeline the multiplication operations and synchronize them while adding some delays into each stage, in order to ensure a parallel execution of sequences. The outputs are generated for each 8-16 clock cycles and in each cycle a new seed is inserted.

In [93], the authors vary the biotic potential α and observe the divergence of random for almost all initial values. Accordingly, they propose a range of the form $[\alpha, 1 - \alpha]$, where the biotic potential is $\alpha < 0.5$. Another way to select the parameter α is presented in [94]. They propose a couple of two logistic map PRNGs, each having different seed and parameter (x^0, α_1 and y^0, α_2 respectively), where both generates pseudorandom numbers synchronously. The main idea is to recycle the pseudorandom number generated by the first chaotic map, namely x^{t+1}, as the biotic potential α_2 for the second one (y^{t+1}) when either $3.57 < x^{t+1} < 4$ is satisfied or the sequence output is divergent. Another

coupling chaotic map is presented in [95]. In this work, the former is based on the H*é*non map and the latter is an 1-dimension logistic map. The former is used to generate a random sequence, and the latter controls a multiplexer to choose the output of the first one according to the value generated by the logistic map. The output of the logistic map generator is then decomposed in 32 bits; the first most significant bit is *XOR*ed with its neighbor bit. The result is then *XOR*ed again with the next bit until reaching the least significant one.

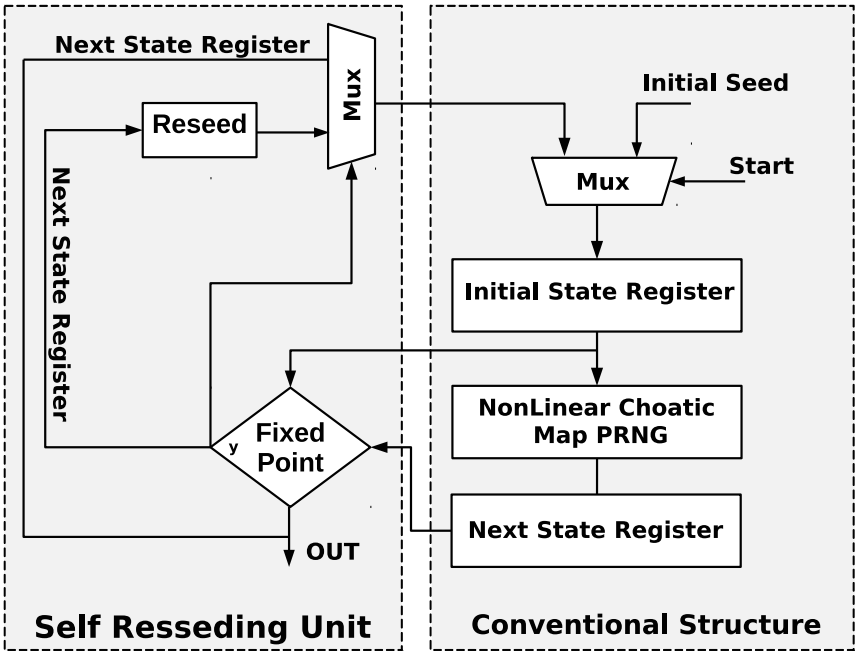
Finally, in [96] four different chaotic maps are implemented in FPGA, namely, the so-called Bernoulli, Chebychev [97], Tent, and Cubic chaotic maps. The implementation is done with and without FPGA's DSP blocks for the multiplication operations. The results show that the Bernoulli chaotic map gives a higher ratio of area/power compared to the other chaotic generators.

Spatiotemporal Chaotic PRNG is a temporally chaotic system which is an extension of chaotic maps. It is also spatially chaotic (many mathematical models can be used to represent this type of generator). For instance, in [98] the authors present a spatiotemporal chaotic PRNG , which is based on a coupled chaotic map lattices defined as

$$x_i^{t+1} = (1-\varepsilon)f(x_i^t) + \frac{\varepsilon}{2}(f(x_{i-1}^t) + f(x_{i+1}^t)). \tag{13}$$

In this equation, t (resp. $i = 1,\ldots,k$) is a temporal (resp. a spatial) index of discrete lattice, ε is the couple parameter, and f is a logistic map. They first deal with continuous domain digitizing of all operands to be suited for hardware implementation. To achieve this, they consider a particular version of Equation (13) where x ranges over $\{0,1,\ldots,2^k-1\}$ and f is a modified logistic map, $f(x) = \lfloor \frac{4x(2^k-x)}{2^k} \rfloor$ for a k-bits precision. Secondly, to avoid the finite precision chaotic map problem, they compute only the insignificant bit which is subject to be an output. Indeed, for each 25 clock cycles, only the $w = 16$ most insignificant bits of the random numbers would be used from each lattice and the computational precision $k = 32$.

Chaotic based Timing Reseeding (CTR) proposed first in [99] aim at removing the short period problem due to the quantization error from a nonlinear chaotic map PRNG. Instead of initializing the chaotic PRNG with a new seed, the seed can be selected by masking the current state x^{t+1} at a specific time (see Figure 1.10). More precisely, the reseeding unit compares the two register states to check whether a fixed point has been reached. In this case x^{t+1} is not streamed out. It is masked with a constant and the result is stored in the initial register state. Additionally, it increases the period each time the condition is true or the reseed period is reached (counter). This main concept of CTR was first implemented in FPGA [100], in which the Carry Lookahead Adder [101] has been used to optimise the critical path of the partial products of the multiplication operation. Unlike [100], authors of [102] present more hardware details for reducing multiplication operation resources. They also mix the output x^{t+1} with an auxiliary generator z^{t+1} to improve statistical tests. The mixer module is a DX generator [103], whose output is as follows: $z^{t+1} = (z^t + (2^{28} + 2^8)z^{t-7}) \mod (2^{31}-1)$. Then, the authors add the MSB-bit of x^{t+1} (32^{th} bit) to the 31 LSB-bits of the final output $y^{t+1}[30:0] = x^{t+1}[30:0] \oplus z^{t+1}[30:0]$, which generates a full 32-bits output state and has a full period. Both uses Circular Left Shift [104] (CLS) and End-Around Carry Adder [105] (ECA) to optimize the multiplication operations. They finally suggest to choose a reseeding period that must be not only prime, but also not a multiple of the nonlinear chaotic map PRNG. The same approach has been used in [106] for



source [100]

Figure 1.10: Chaotic based Timing Reseeding PRNG: masking the current state x^{t+1} at a specific time (fixed point between the two register states is reached)

plaintext encrypting/decrypting application system.

Differential Chaotic PRNG is a digitized implementation of a nonlinear chaotic oscillator system in Rössler format [107]. It uses an approximated numerical solution to solve the dynamic system generalization of the Lörenz hyperchaos. More precisely, a chaotic system with multiple Positive Lyapunov Exponents (PLE [108]) is also known as hyperchaotic system. Lyapunov exponents are a quantity that characterize the rate of separation of infinitesimally close trajectories. That is, a higher order of hyperchaotic system can be generated from a higher number of variables of PLE (order $m_{hyperchaotic}$ needs $n_{PLE} = 2m + 1$ variables [109]). A basic representation of the dynamical system is proposed in [110,111] Equation. (14).

$$\begin{aligned} -\dddot{X} &= \ddot{X} + B(\dot{X}) + X \\ B(\dot{X}) &= \begin{cases} \alpha_1, & \text{if } \dot{X} \geqslant 1 \\ \alpha_2, & \text{otherwise,} \end{cases} \end{aligned} \tag{14}$$

where, α_1, α_2 are integer values in the switch condition. The idea is to create a chaotic system with a unique equilibrium point at the origin. Indeed, to guaranties a chaotic generation, B value must switch between $\alpha_1 >1$ and $\alpha_2 < 1$.

This latter can expand in more than one direction (i.e., Euler approximation where $Y = \dot{X}$ and $Z = \ddot{X}$) and generates a much more complex attractor compared to other chaotic systems.

The resolution of Equation (14) was the main study done in [112] (with other differential systems as the Chen [113] and Elwakil [110] ones). The authors deploy three different numerical methods for each system: 4^{th} order Runge-Kutta [114], mid-point [115], and Euler techniques [116]. Unlike the Euler techniques that only require one calculus per

iteration, the mid-point provides more precise results but longer calculation paths. Additionally, the Runge-Kutta 4^{th}-order have the longest calculation path but it has the most accurate numerical approximation. Obviously, Euler techniques show better results for implementation of differential chaotic methods in FPGA, with respects area and throughput perspectives.

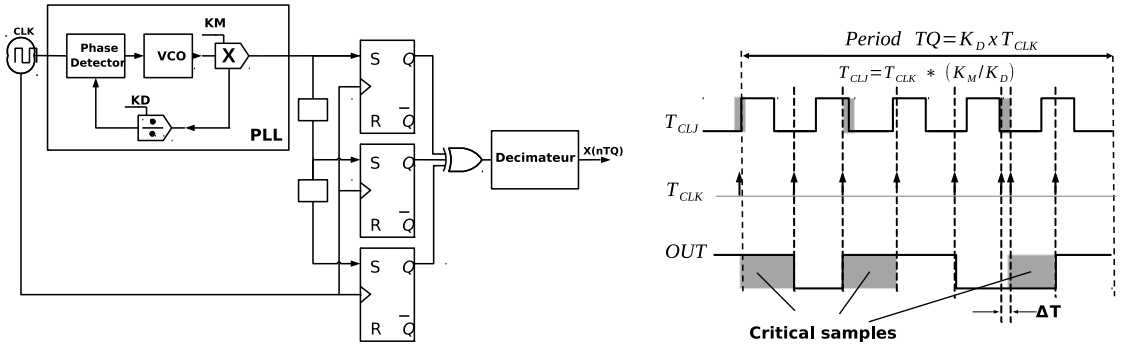
More details regarding implementation and optimization of the multiplication by a constant in Equation (14) are provided in [117]. In this article, authors proposed to use the Euler approximation, as illustrated in Equations (15), where the oscillation margins are within a time interval $[h, \alpha_1]$ (h is the Euler step).

$$\begin{aligned} X^{t+h} &= X^t + hY^t, \quad where \quad Y = \dot{X} \\ Y^{t+h} &= Y^t + hZ^t, \quad where \quad Z = \ddot{X} \\ Z^{t+h} &= Z^t - h(Z^t + Y^t B(Y^t) + X^t) \end{aligned} \tag{15}$$

Their optimization is based on transformation of the parameters $h = 2^{-a}$ and $\alpha_1 = 2^b$, $\alpha_2 = 0$ to simplify the multiplication to a simple shift operation (a and b are positive parameters). They use a Carry Lookahead Adder (CLA) [101] and a Carry Save Adder (CSA) [118] for the multiplication in the first two Equations. (15), and a Carry Propagate Adder (CPA) [119] for the last one. Additionally, a post-processing is integrated for better results in statistical tests, which specifically discards the most significant bits. Authors of [91], for their part, have implemented the so-called Oscillator Frequency Dependent Negative Resistors (OFDNR) [111], which uses the same Euler approximation illustrated in Equations (15). However they have not detailed the resources they used for such multiplication on their FPGA (e.g., DSP, LUT, ...).

In [120] is presented a non-autonomous four-dimensional hyperchaotic PRNG based on Rössler differential equations. In such a chaotic system some undesirable behaviors can appear. Thus, an advanced process-control is necessary in order to delay the occurrence of the hyperchaos. Therefore, the authors used Euler approximation and a control function of 256 bits Linear Feedback Shift Register (LFSR), whose outputs are multiplied by the appropriate coefficient of the control function. However, a post-processing of 256-bits based Fibonacci LFSR is used to remove the short-term predictability of hyperchaotic generator and to successfully undergo the statistical tests of NIST batteries. The post processing combines two loops of rotation and *XOR* feedback loops. The first one uses a fixed 1-bit static rotation to suppress the short-term predictability. The second one is based on a variable rotation controlled by a Fibonacci series of k-bits. The differential sensitivity problem is solved by changing any bit while the other bits is propagating during n-cycles.

Chaotic Iteration based PRNG (CI) is a pseudorandom number post treatment proposed in [10, 11] and based on Chaotic Iterations (CIs [15]). It is based on Devaney's [4] theory of chaos. This theory focuses on recurrent sequences of the form $x_0 \in \mathbb{R}$: $x^{t+1} = f(x^t)$. It tries to find functions f which present elements of complexity and disorder.We will recall the mathematical definition of this theory in the next Chapter 2.



(a) PLL architecture (b) PLL timing

Figure 1.11: Phase-Locked Loop TRNG: detecting the jitter by sampling the reference clock signal T_{CLK} using a correlated signal T_{CLJ} synthesized in the PLL

1.4/ True Random Number Generators

We focus now on FPGA implementations of truly random number generators (TRNGs). FPGA based TRNGs are physical generators that use various hardware components of FPGAs to produce random-like numbers in a faster way than using software. These TRNGs use, as entropy source, either the electronic noise of embedded components or some environmental sensors (temperature, noise, and so on). FPGAs are thus efficient and inexpensive random number generators. Various techniques and hardware optimizations have already been proposed in the literature, while FPGA components have been used in RNG context for optimization, mixing with external components, or as post-processors.

1.4.1/ Phase-Locked Loop TRNGs

The Phase-Locked Loop (PLL) [37] is a circuit derived from an external clock generator source like a quartz or a "Resistor Capacitor" circuit, which can be configured to produce a signal whose phase is associated to the phase of the input signal (see Figure 1.11(a)). This latter depends on the physical environment (power, temperature, or any other physical quantity), and it uses a jitter extraction technique as random stream, which is indeed a short-term variation of the clock propagation. Analog PLLs use the jitter caused by Voltage Controlled Oscillator (VCO) noise, while digital PLL [121] generators extract their randomness from synchronous/asynchronous Flip-Flop components. The most common jitter measurements used by FPGA vendors are, namely, the period jitter and the cycle-to-cycle one. The first jitter is defined as the difference between the n-th clock period and the mean clock period, while cycle-to-cycle jitter consists of the difference between adjacent clock cycles in the collection of sampled clock periods.

The authors of [122] have proposed an analysis about extracting randomness from the jitter of a PLL implemented on an Altera FPGA. Their study is based on detecting the jitter by sampling the reference clock signal T_{CLK} using a correlated signal T_{CLJ} synthesized in the PLL, where $T_{CLJ} = T_{CLK} \times (K_M/K_D)$ with K_M and K_D as PLL multiplier and divider that must be prime number constants. According to [122], the maximum distance, further denoted as $\max(\Delta T_{min})$, between the two clocks CLK and CLJ must satisfy $\max(\Delta T_{min}) < \sigma_{jit}$ to be able to extract randomness. Indeed, according to the authors, in ideal environmental

conditions, we have $\sigma_{jit} = 0$ (we do not have any jitter). In that situation, the sampled outputs are deterministic and can be represented by a series of a bitwise addition of K_D input. According to these authors, the period in that situation is equal to $T_Q = K_D T_{CLK} = K_M T_{CLJ}$. Contrarily, in real case conditions, σ_{jit} is necessarily negative, and so the output loses its deterministic character and becomes random. Indeed, the maximum distance $\max(\Delta T_{min})$ between the two clocks is dependent on the jitter distribution, while the outputs has a direct impact by this latter following the expression [122]:

$$x^t(nT_{CLK}) = x\Big((nT_{CLK}) - \sum_{j=0}^{i} J\tau_j\Big), \tag{16}$$

where τ is the jitter and J is the value of the output influenced by the jitter. The period is changed in $\max(\Delta T_{min}) = T_{CLK} \times GCD(2K_M, K_D)/(4K_M)$, where K_D is odd and $\max(\Delta T_{min})$ is divided by 2.

This research work has been deepened in [123] by combining more than one PLL either in parallel or in series. By doing so and due to this combination, the sensitivity S to the jitter effect is significantly increased according to the formula:

$$S = T_{CLK} \max(\Delta T_{min}). \tag{17}$$

As expected, the lowest sensitivity is achievable by using only one PLL. In that case, the number of random samples and their entropy are low, due to a low value of S. To solve this problem, the authors add a second PLL, either in parallel or in a cascaded configuration, the objective being to increase the entropy without increasing too much the sensitivity.

Authors of [124] have tested the impact of "environmental" PLL conditions (encompassing its temperature, its bandwidth, etc.) on the statistical quality of the produced output. They have deduced that a low bandwidth of PLL causes a higher number of critical samples, which decreases the output jitter, and consequently increases the tracking jitter. Finally, authors in [125] propose two configurations of PLL based TRNGs in embedded systems.

1.4.2/ Ring Oscillator TRNGs

A Ring Oscillator (RO) is a series of an odd number of NOT gates, whose outputs states are balanced between two voltage levels, *i.e.*, between bit 0 an bit 1. The NOT gates, or Inverter Ring Oscillators IROs, are cascaded, while the output of the last inverter is fed back to the first inverter of that chain (see Figure 1.12). In [126, 127], the authors have proposed a TRNG based on two ring oscillators. This latter is rated by different clocks generated by an internal PLL implemented on FPGA. The authors have also extracted the jitter of the 2 ROs implemented in only one CLB slice.

Similarly, the authors of [128], have proposed an approach that combines ROs based on inverters with XOR gates. Their approach is close to the LFSR one, except that they use inverters, the latter being combined either using the Fibonacci setup or the Galois one. The result also has an analog feedback to the input, where the feedback polynomial form is $f(x^t) = \sum_{i=0}^{k} f_i x^{t+i}$, with $f_0 = f_k = 1$. However, the inverter does not reach a fixed state if $f(x^t) = (1 + x^t)h(x^t)$ and the primitive polynomial is such that $h(1) = 1$. Authors have finally demonstrated their ability to extract a better stable state compared to classical RO TRNG, from which randomness can be produced.

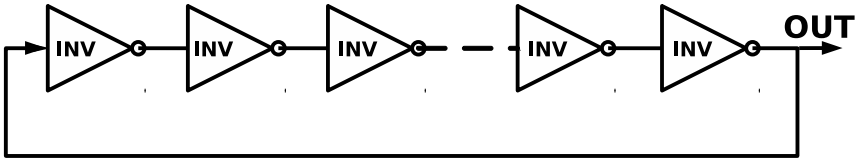
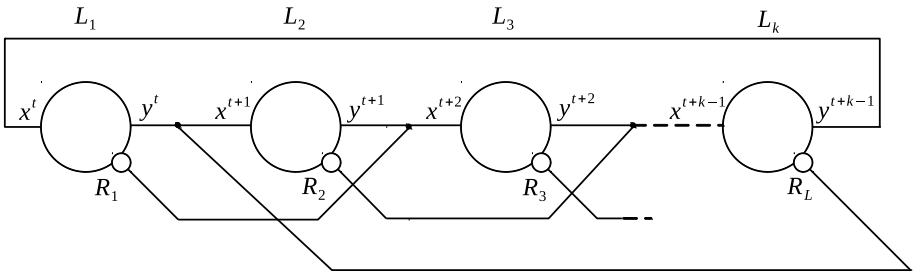


Figure 1.12: Inverters based ring oscillator



source [129]

Figure 1.13: Self-Timed ring architecture: at each ring stage L (Muller gate and an inverter), the jitter is propagated forward if $y^t = y^{t+1}$ or conversely backward, when the output is the XOR of each extracted jitter by a Flip-Flop

1.4.3/ Self-Timed Ring TRNG

Self-Timed Ring (STR) proposed in [129–131] is an alternative approach to generate clock jitter compared to the inverter RO based TRNG. The structure of STR consists of a micropipeline architecture [132], as described in Figure 1.13. In this latter, a ring of L stages can generate k-bits outputs, denoted by y^t $(0 \leqslant k \leqslant L-1)$, at each stage and with a propagation phase equal to $\Delta\varphi = T/2L$. A stage consists of a Muller gate and an inverter. Therefore the jitter period in STRs, for each ring stage, can be considered as an independent entropy source compared to the propagation of one event all around the ring in IRO.

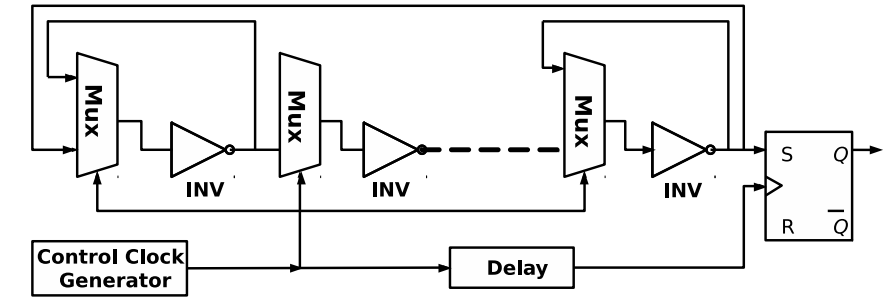
Two situations can occur. If the outputs of two successive stages are equal $(y^t = y^{t+1})$, then the clock jitter is propagated forward. Conversely, in case where $y^t \neq y^{t-1}$, then the jitter is propagated backward. The final output sequence $(y^t)_{1\leqslant k\leqslant L-1}$ is extracted at each ring stage output using a Flip-Flop, and the result is combined according to the following XOR operation: $\psi = y^1 \oplus y^2 \oplus \cdots \oplus y^{t+k-1}$.

1.4.4/ Metastability TRNG

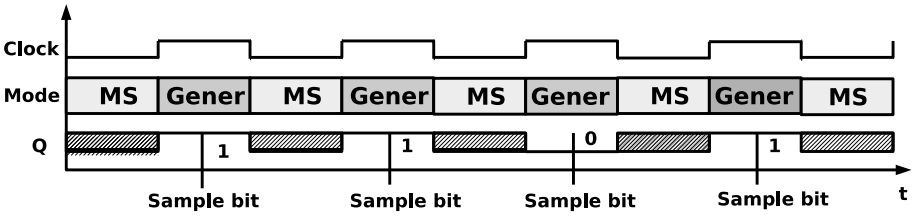
Metastability is a phenomenon that can occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains. This short time phenomenon can cause system failures in digital devices.

Authors of [133] have presented a way to use such metastability phenomena as entropy sources generated by 5 IRO (Ring Oscillator based Inverters) stages. Their goal is to maintain metastability as long as possible, while extracting randomness from this entropy source. To achieve this objective, the authors have firstly implemented inverters as loop rings, and have used a clock generator controller module to switch the connectivity be-

tween the IRO stages following two modes, namely the metastability mode “MS” and the generation one, as described in Figure 1.14. By doing so, the output converges to the metastability level, and it stays a longer time in that state than when using a bi-stable circuit (Flip-Flop), causing thus a high entropy. Secondly, the authors wanted to estimate the robustness of the system after applying the sampling process in various environmental variation modes on FPGA. To achieve this second objective and for a higher quality output, they have added another stage to decrease the operation rate, by applying a Von-Neumann post-processing. Such a post-processing stage influences the loads of the last inverter (RC parasitic). Let us finally note that, operationally speaking, the end of the IRO was implemented in ASIC while the post-processing process was achieved by using a FPGA in order to test the global device.



(a) Metastability based TRNG architecture



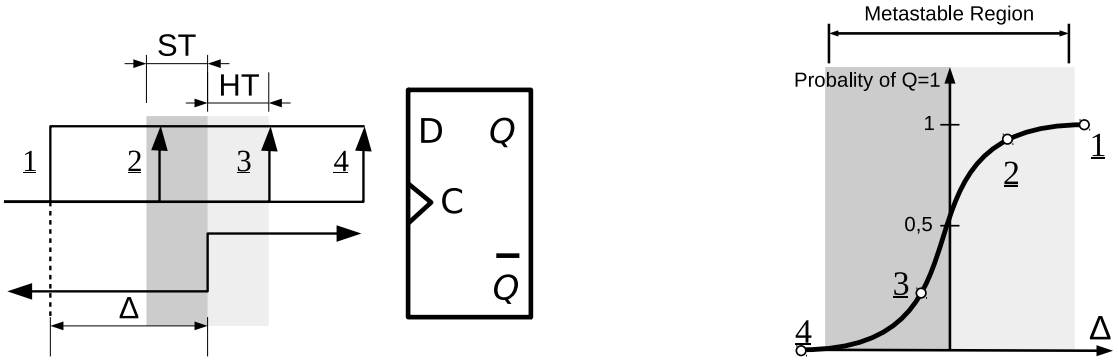
(b) Metastability timing switching based TRNG

source [133]

Figure 1.14: (a) TRNG based on the metastability of multistage architecture inverter ring oscillator, (b) The timing switching connectivity between the IRO stages following the metastability mode “MS” and the generation mode.

Another metastability circuit used as a TRNG has been proposed in [134]. Authors of this article have proposed to use the Flip-Flop metastability when there is a violation in setup/hold time [135] (see Figure 1.15). The setup time ST is the amount of time a synchronous input of the Flip-Flop must be stable before the active edge of the clock. The hold time HT is the amount of time a synchronous input of the Flip-Flop must remain stable after this active edge of the clock (c.f. Figure 1.15). The violation occurs when the input data is between these two times. However, due to their short time (ST/HT), the output must converge rapidly to a stable state 0 or 1 if the input is stable during this two times. Their system is based on a closed-loop feedback mechanism for auto-adjustment on delay Δ, controlled by the Programmable Delay Lines (PDLs) stage based on a LUT, in order to avoid violation and maintain metastability.

The proposed system uses at-speed monitor to keep tracking the output bit probability and the Proportional-Integral (PI) controller, in order to decide to add or subtract the delay difference. As for the updated/corrected delay difference Δ, it is the difference between



(a) Metastability creating in FF

(b) Metastability timing of FF

source [136]

Figure 1.15: (a) The setup (ST) and hold (HT) scenarios operations in Flip-Flop, (b) the output probability depending the delay difference (Δ) of the input signal

the bias/skew caused by the asymmetric routing Δ_b, with delay issued by environment changes (temperature, etc.), and the delay Δ_f corresponds to the "corrected feedback delay difference" injected by PDL, according to the following formula:

$$\Delta = \Delta_p + \Delta_b - \Delta_f. \tag{18}$$

Their method consist of tuning sampling and signal arrival times by setting Δ to 0, which leads to the metastability of the D-Flip-Flop.

An updated version of this TRNG has finally been proposed in [136]. Thanks to an analysis of probability, they reach some metastable states for a long period and prevents deterministic states. To do so, they have used an additional hardware resource as memory for storing the outputs, and a Hamming weight to calculate the history of the probability bits.

1.5/ Experimental Results and Hardware Analysis

1.5.1/ Methodology

Formally speaking, the space represents the allocation cost of most objects used in the algorithm (tables, indexes, loops, etc.). It can also be combination of many PRNG algorithms. In terms of FPGAs, the latter can be translated in memories, registers, and LUT resources, etc. These resources can be a single basic operation (like addition or subtraction, multiplication of variables or constants), algebraic functions (division, modulo, etc.), or any other elementary function. The question raised in this section is thus: how much hardware resources are needed to provide pseudorandom numbers with a good statistical profile? And which algorithms outperform the other ones in terms of internal resources, while providing higher throughput?

Almost aforementioned (P)RNGs have been evaluated regarding their hardware performance according to three parameters: (1) the area, which is the result of $(LUT + FF) \times 8$, (2) the throughput being the frequency (clock-to-setup) multiplied by the RNG output

length for one clock cycle, and (3)the ratio between throughput over area in Mega bits per area unit.

1.5.2/ HARDWARE COMPARISON

Hardware implementation resources required by linear (P)RNGs, their throughput, and the rate area over throughput are presented in Figure 1.16, when nonlinear ones are in Figure 1.17. Finally, the TRNGs are represented in Figure 1.18. We can notice from these figures that not all research papers consulted in our investigation offer the three elements of comparison (area, throughput, and statistical tests). Moreover, most research of PRNG in FPGA level focus on linear PRNG than others, which is explained by an easier implementations of these latter. Indeed, Chapter 3 describes some implementations of these generators on FPGA within a common FPGA platform.

Let us start start to discuss the results obtained with linear PRNGs, as illustrated in Figure 1.16. It appears clearly that the cellular automata has the lowest area, when compared to the other approaches. Such results can be explained by the need of a low amount of resources to store both the states and the rules in the cellular automata. Conversely, the TGFSR family deploys BRAM block memories to read 3 word and write the output in one cycle, whereas LFSR family uses more LUTs in order to parallelize the shifting process based on the polynomial equation. Another parameter is the use of black box as DSP and block memories. The latter optimize the logic operation as multiplication, support the floating point, store internal process in a multidimensional bloc, and finally read and write multiple states in parallel from the BRAM. These advantages, leading to the difficulty to compare such designs to other ones that do not have that, lead naturally to further area bloc consumption in the case of an ASIC implementation. As a consequence, we will consider that (P)RNGs without black boxes are better and more recommended for cryptographic applications.

In terms of area, the PRNGs based on cellular automata [80, 82, 85] have the lowest resource occupation of FPGA, if we compare them to the other ones (see Figure 1.16(a)). By comparison, the PRNG based on LFSR [59] is 76 times larger. We can also remark that most TGFSR implementations do not consider the seed process, while its computing increases the area and decreases the throughput, during the load of 632 words sequentially in the block memories. The throughput, for its part, is completely related to both data path and width (dynamic range) of the design. Additionally, we must take under consideration the fact that most linear PRNGs are 32 bits ones, while the throughput increases with generators manipulating more than 64 bits. However, as stated previously, disabling DSPs and Block memories induces a decrease in the frequency and the throughput respectively. Figure 1.16(b) illustrates opposed results for the throughput, where the LFSR based LUT family has the largest throughput of 343 Gbps, while it is 5 Gbps for the Mersenne Twister. However, the latter are for $1,042$ bits and 128 bits respectively, when for 32 bits, we have 128 Gbps for the LFSR-LUT [66].

Let us focus now on the Throughput/Area ratio (see Figure 1.16(c)). Here, the LUT and shift register based design [66] outperforms all the other linear PRNGs: the ratio is twice as efficient as the second best one [72], which is the Mersenne Twister based PRNG with parallel BRAM.

The performance of PRNGs belonging in the chaotic category are illustrated in Figure 1.17. The one that is based on the logistic map has the lowest area occupation.

Results obtained concerning area (Figure 1.17(a)) can be explained by the use of a basic operation (the shift one), and because the bionic coefficient α can be considered as a constant when implementing the logistic map. PRNGs based on chaotic iterations, for their part, need to embed linear PRNGs for their strategies: on the one hand, CIPRNG-XOR uses 3 PRNGs, while on the other hand ICGPRNG manipulates only one, but with a permutation function. Figure 1.17(b) illustrates a good performance of the first version chaotic iterations family compared to the differential PRNG based Euler optimization [120], an optimized logistic map [92], and these PRNG based chaotic Bernoulli map [96] have the largest throughput in this category of chaotic generators. Regarding the Throughput/Area ratio in Figure 1.17(c), the chaotic PRNG based on LCGM [91] outperforms all the other linear PRNGs (we consider the lowest ratio for comparison): the ratio is 4.1 times more efficient than the second best one [10], which is a chaotic iterations generator based on BBS and XORshift.

Finally, considering the TRNG analysis, only a throughput comparison is provided in Figure 1.18. Indeed, all the considered authors prefer not to discuss about area... which is so low when compared with PRNGs. Hence, even with this main advantage of optimized resources usage, the throughput is too low and it ranges from *Hz* to just a few *kHz*. Compared to PRNGs, TRNGs are probably more secure, while PRNGs can be deploied as fast generators.

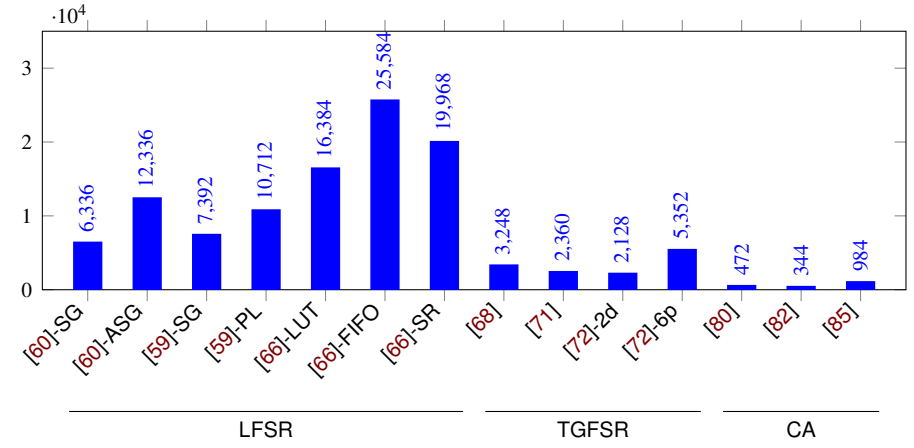
As a conclusion, linear PRNGs can play an important role for FPGA applications, due to their rapidity and parallel generation, if we compare them to other pseudorandom generators. Chaotic PRNGs, for their part, are more secure. They are non linear PRNGs and have low hardware resources compared to the linear ones. Finally, despite the low throughput generated by the TRNG, they are still consuming only a few logic while generating a real random output.

1.6/ STATISTICAL TEST ANALYSIS

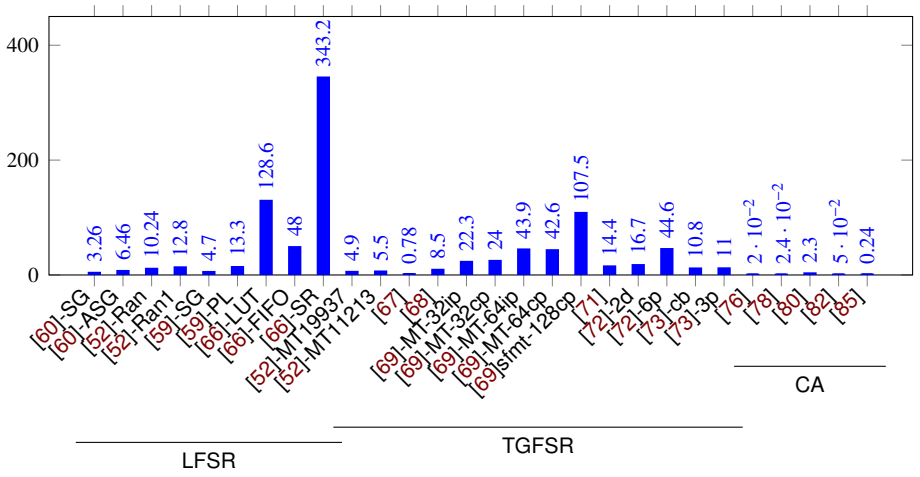
Statistical tests are used to evaluate whether the output of a given RNG can be separated from a real random sequence obtained, for instance, by rolling a dice. Such tests are usually grouped in “Batteries”, like the FIPS [137], DieHARD [43], NIST SP$800-22$ [138], TestU01 [44], or AIS [139] ones. In what follows, the content of these tests is recalled, for completeness purpose so as to make our article self-contained..Indeed, all the aforementioned test batteries generate a common estimated value named the p-value. This latter assesses the statistical behaviors for each test applied in a sequence generated from RNGs.

The National Institute of Standard and Technologies introduced their first test battery namely Federal Information Processing Standard (FIPS) 140-1 [137] in 1994. These quick result tests have been further updated to the FIPS 140-2 [140] version, which covers more complex test batteries (focused for instance on security level).

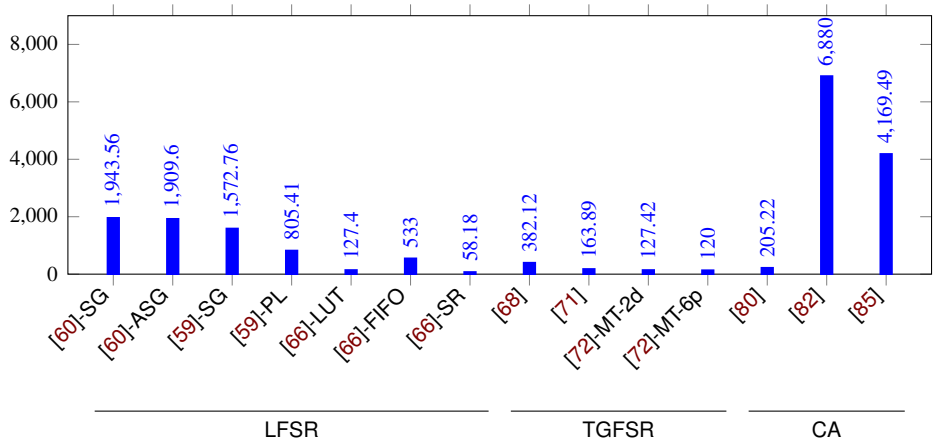
Meanwhile, the DieHARD battery has been proposed by George Marsaglia [43]. It contains 18 tests of randomness. It was designed to provide a better way of analysis in comparison to the previously released NIST tests. Unlike this latter, the p-values have now to belong to some fixed chosen interval $[\alpha, 1-\alpha]$, with a signification level of α for 5% for instance. An example of these batteries are: “Birthday spacings”, “Overlapping permutations”, “Ranks of matrices”, “Monkey tests”, “Count the $1's$”, “Parking lot”, “Min-



(a) Area ((LUT+FF)×8)



(b) Throughput (Gbps)

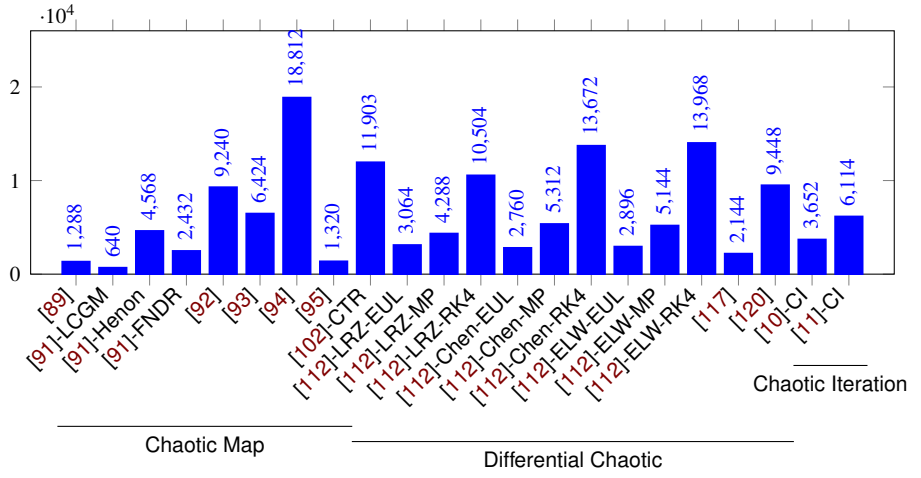


(c) Ratio (Throughput/Area) (*Mbit* per area)

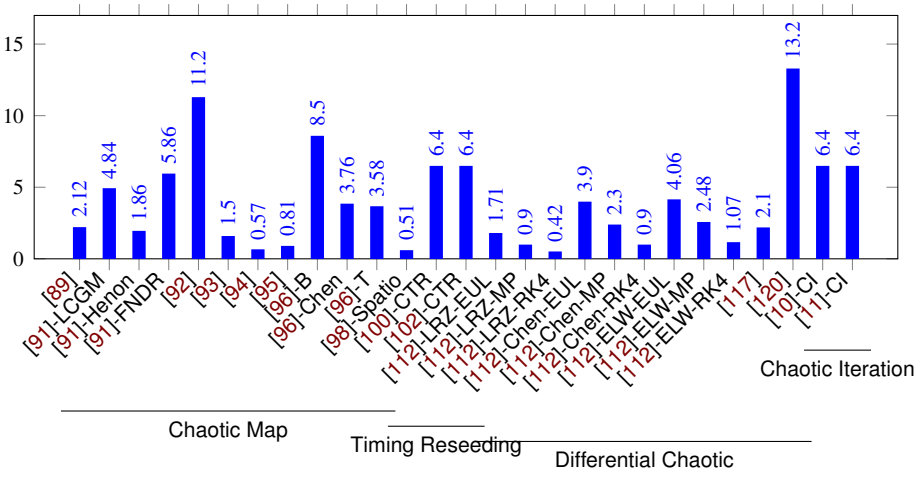
Figure 1.16: Linear PRNGs FPGA hardware analysis.

imum distance", "Random spheres", "The squeeze", "Overlapping sums", "Runs", and "The craps".

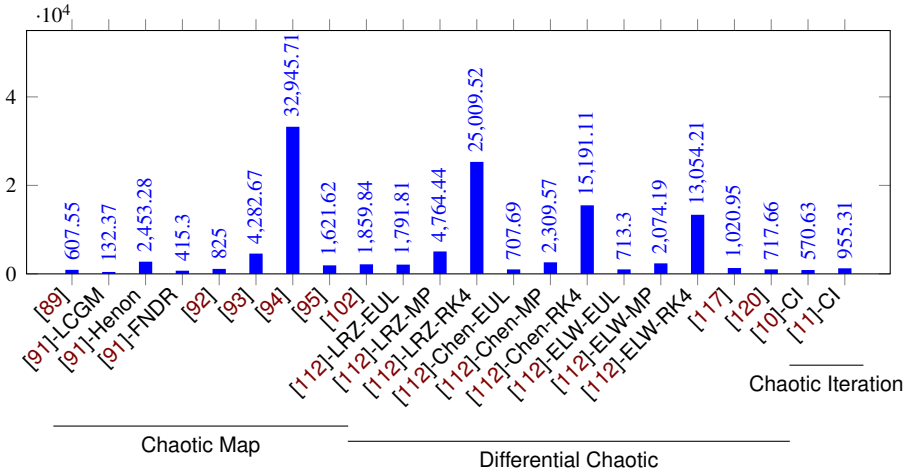
The AIS-31 battery [139] is a German standard to test and evaluate the security properties



(a) Area ((LUT+FF)×8)



(b) Throughput (Gbps)



(c) Ratio (Throughput/Area) ($Mbit$ per area)

Figure 1.17: Non-lineair PRNGs FPGA hardware analysis.

of truly random number generators. It uses 9 statistical tests for the evaluation of a TRNG. AIS can be divided in two categories: the first one consists of T0-T4, which are the same function of FIPS 140-1 [137]. These later are mostly used to test the outputs of a post-

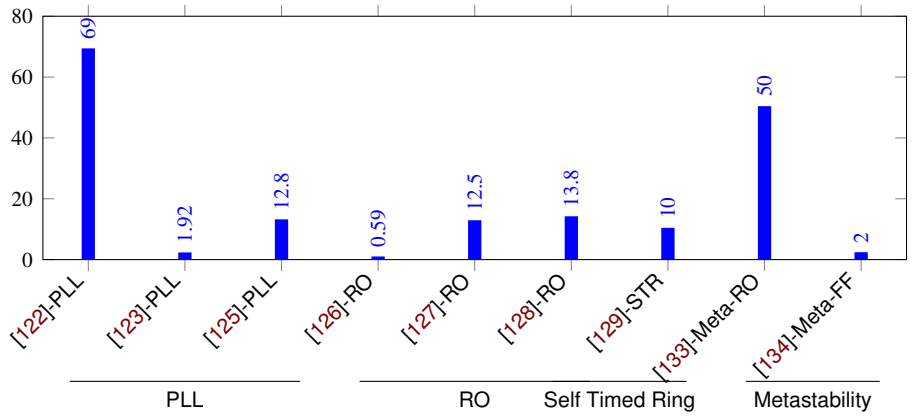


Figure 1.18: TRNGs FPGA implementation analysis: Throughput (Mbps).

processing. T0 is the "disjointedness test", which collects 65536 of 48-bit and verifies that two adjacent values must not be equal. T1 is the monobit test, T2 is the poker test, T3 is the run test, and T4 is the longest run test. The T5 is the auto-correlation test, where T6 is a "uniform distribution test" including of 2 sub-tests. T7 is a "comparative test for multinomial distributions", and finally T8 is an entropy test (Coron's test).

In the other side, National Institute of Standard and Technologies introduces a new test battery known as "NIST SP800 – 22" [138]. This one aims at testing the random profile of a given sequence using 15 tests. More precisely, it evaluates a long binary sequences generated by the RNG for the randomness and a higher security testing level than the FIPS 140-2. The tested sequences must have a fixed length N, where the parameter N is such that $10^3 < N < 10^7$. Then, for each statistical test, a set of s sequences is produced by the RNG under test, and p-values are obtained. They all need to be larger than 0.0001 to reasonably consider the associated sequences as uniformly distributed and cryptographically secure according to NIST standards.

Following "NIST SP800 – 22" [138], the tests are: "Frequency monobit", "Frequency within a block", "Runs Test", "Longest run of ones in a block", "Binary matrix rank", "Discrete Fourier transform (spectral)", "Non-overlapping template matching", "Overlapping template matching", "Maurer's universal statistical test", "Serial test", "Approximate entropy", "Cumulative sums", "Random excursions", and "Random excursions variant".

TestU01, for its part, is currently the most complete and stringent battery of tests for RNGs [44], which groups more than 516 tests inside 7 sub-batteries. In this section, we focus on three major sub-batteries, that encompass 319 tests and which are specific to PRNGs. They are, namely, the SmallCrush, Crush, and BigCrush batteries of tests. Big Crush is the most difficult sub-battery in TestU01. This latter uses approximately 2^{38} pseudorandom numbers and applies 160 statistical tests (it computes 160 p-values, that must belong to $[0.001, 0.999]$ in order to pass the considered test).

The 7 sub-batteries are listed below.

- **Small crush:** The first battery to check, with 15 p-values reported. This is a fast collection
- **Crush:** This battery includes many difficult tests, like those described in [141]. It computes a total of 144 tests and p-values, which resume a total of 2^{35} random

numbers and 96 statistical tests.

- **Big crush:** A suite of very stringent statistical tests, and the most difficult battery to pass. It computes a total of 160 tests and p-values, which resume a total of 2^{38} random numbers and 106 statistical tests.
- **Rabbit:** This battery of tests reports 38 p-values.
- **Alphabit:** Alphabit and AlphabitFile have been designed primarily to test hardware random bits generators. A 17 p-values are reported.
- **Pseudo-DieHARD:** This battery implements most of the tests contained in DieHARD or, in some cases, close approximations to them. It is not a very stringent battery. Indeed, there is no generator that can pass Crush and Big crush batteries and fail Pseudo-DieHARD, while the converse occurs for several defective generators. 126 p-values are reported here.
- **FIPS_140_2:** The NIST battery, recalled previously.

1.6.1/ STATISTICAL RESULTS OF FPGA BASED RNG

In Table 1.1 and 1.2, a number of generators are classified according to the battery test they have undergone. As it can be observed, the most stringent battery (Big crush) has only been applied twice in the literature, namely [10, 127]. Let us notice that most (P)RNGs pass the Diehard and NIST batteries, while only a few PRNGs have been tested using the FIPS that has been integrated latter inside the NIST. Considering the TestU01 one, only crush batteries are usually considered. All generators fail at least one test, with the exception of chaotic iterations generators that can pass the whole battery.

Authors in [12, 14] investigate the related problem for linear PRNGs. They show too that usual chaotic PRNGs are not passing the BigCrush when they consider its non linearity. However, being linear does not lead to a high linear complexity, which is defined by the degree of their polynomial characteristic function. However, most random number generators are linear recursive, and so they fail in the so-called statistical Linear Complexity Test of TestU01 [44]. This test characterizes the (P)RNGs by their longest LFSR model: non randomness is claimed when the model is too short. This model is estimated by using the well-known Berlekamp-Massey algorithm [142]. It determines the shortest polynomial of a linearly recurrent finite output sequence in $G\mathbb{F}_2$. Note that all the other generators fail too the linear complexity test, except for PCG32 and MRG$32K3a$: indeed, only PRNGs based on chaotic iterations are passing TestU01. Under this category, the authors propose too an extended internal space of 64 bits (CIPRNG-XOR) for 32 bits generators, when they increase the number of internal iterations to be uniformly distributed and to pass statistical tests.

Finally, TRNGs are hard to test with TestU01 (specially the BigCrush battery), as it needs 10^{38} random bits for a full test. Figure 1.18 shows a general throughput of the order of $Kbps$, which makes it difficult to collect the minimum amount of data needed in such tests. Under these conditions, only the TRNG of [127] based on ring oscillators has been proven to pass with success the BigCrush battery. Note finally that other batteries offer more flexibility and need a lower amount of bits for their embedded tests (namely, Diehard, NIST, and AIS), but they are less stringent and trustworthy than TestU01.

Table 1.1: Statistical Tests Analysis: Diehard, FIPS, and NIST

RNG	Diehard	FIPS	NIST
PRNG	[143] [144] [145] [71, 72] [52, 69] [80, 82] [83, 85]	[98] [146]	[60] [59] [61, 145] [11, 117] [98, 120] [78, 85] Li and Chen [146, 147] [148]
TRNG	[127] [129]	[124] Cherkaoui [129, 131] [133]	[122] [123] [126] [125] Cherkaoui [129, 131] [134, 136]

Table 1.2: Statistical Tests Analysis: TestU01 Crush and BigCrush, AIS

RNG	TestU01 Crush	TestU01 BigCrush	AIS
PRNG	[143] Thomas [64, 144, 149] [71, 72] [69] [150] [89, 92] [95]	[10]	
TRNG		[127]	Cherkaoui [129, 131] [133]

1.7/ CONCLUSION

We have provided a widespread coverage of the current research in hardware implementation of random number generators on FPGA. We have first recalled well known “linear generators”, encompassing LCGs, LFSRs, look-up table optimised ones, twisted generalized feedback shift registers, and cellular automata. We next have deeply investigated the non-linear ones, based on Blum-Blum-Shub or on chaotic maps. Then a large review of the true random number generators for FPGA has been proposed, encompassing respectively the phase-locked loop, the ring oscillator, the self-timed ring, and the stability TRNG. As a first step, we intents in chapter 3 to implement on the same FPGA platform the most efficient generators with respect to the area [85] [82] [151], wrt. the throughput [149] [120] [92], wrt. the ratio between these two criteria [149] [151] [10], and wrt. the statistical property [10] [149]. A second stage would consist in mixing some of them and to see how their expected properties may be preserved.

2

CHAOTIC ITERATION BASED PRNG

This section introduces the preliminaries and the mathematical proofs used for our proposals based on chaotic iteration. In this chapter, we recall the Boolean domain, the different iteration pattern, and their graph representation. In second part, we present all the mathematical theory related to our contributions for chaotic iterations PRNGs, which is based on unary and parallel pattern. Then, it will be the turn of the mathematical behaviours of the second contribution based on generalized iteration pattern. All these background and the mathematical demonstration of chaos are inspired from the contribution of Jean-François Couchot (HDR reports [152]), Christophe Guyeux (Phd thesis [153]), and our submitted papers.

2.1/ PRELIMINARIES

In what follows, we consider the Boolean algebra on the set $\mathbb{B} = \{0, 1\}$ with the classical operators of conjunction $\ll . \gg$, of disjunction $\ll + \gg$, and of unary negation $\ll \overline{\ } \gg$.

Let N be a natural number. We introduce some notations about elements of $\mathbb{B}^{\mathsf{N}}$. The set $\{1, \ldots, \mathsf{N}\}$ will be denoted by [N]. The i^{th} component of an element $x \in \mathbb{B}^{\mathsf{N}}$ is written as x_i. If the set I is a part of [N], then $\overline{x}^I$ is the $y \in \mathbb{B}^{\mathsf{N}}$ such that $y_i = (1 - x_i)$ if $i \in I$ and $y_i = x_i$ otherwise. We consider the two abbreviations $\overline{x}$ for $\overline{x}^{[\mathsf{N}]}$ (each component of $\overline{x}$ is negated: it is a component to component negation) and $\overline{x}^i$ to $\overline{x}^{\{i\}}$ for $i \in$ [N] (only x_i is denied in $\overline{x}$). For any x and y in $\mathbb{B}^{\mathsf{N}}$, the set $\Delta(x, y)$ contains the $i \in$ [N] such that $x_i \neq y_i$. Indeed, for $f : \mathbb{B}^{\mathsf{N}} \to \mathbb{B}^{\mathsf{N}}$, the i^{th} component if f is named f_i, which is a $\mathbb{B}^{\mathsf{N}}$ function in $\mathbb{B}$. Finally, for each x in $\mathbb{B}^{\mathsf{N}}$, the set $\Delta f(x)$ is defined by $\Delta f(x) = \Delta(x, f(x))$. We can assume that $f(x) = \overline{x}^{\Delta f(x)}$.

Example. *Let consider* $\mathsf{N} = 3$ *and* $f : \mathbb{B}^3 \to \mathbb{B}^3$ *such that* $f(x) = (f_1(x), f_2(x), f_3(x))$ *with*

$$\begin{array}{rcl} f_1(x_1, x_2, x_3) & = & (\overline{x_1} + \overline{x_2}).x_3, \\ f_2(x_1, x_2, x_3) & = & x_1.x_3 \ et \\ f_3(x_1, x_2, x_3) & = & x_1 + x_2 + x_3. \end{array}$$

The Table 2.1 illustrates the mapping of each element of $x \in \mathbb{B}^3$. *For* $x = (0, 1, 0)$, *the following assertions can be deduced directly from the table:*

- $f(x) = (0, 0, 1)$,
- *for* $I = \{1, 3\}$, $\overline{x}^I = (1, 1, 1)$ *and* $\overline{x} = (1, 0, 1)$,

x			$f(x)$		
x_1	x_2	x_3	$f_1(x)$	$f_2(x)$	$f_3(x)$
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	0	1
0	1	1	1	0	1
1	0	0	0	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	1	1

Table 2.1: Map of $(x_1, x_2, x_3) \mapsto ((\overline{x_1} + \overline{x_2}).x_3, x_1.x_3, x_1 + x_2 + x_3)$

- $\Delta(x, f(x)) = \{2, 3\}$.

2.1.1/ BOOLEAN DOMAIN

A Boolean domain maps a function f from Boolean domain to itself and it is defined by:

$$f : \mathbb{B}^{\mathsf{N}} \to \mathbb{B}^{\mathsf{N}}, \qquad x = (x_1, \ldots, x_{\mathsf{N}}) \mapsto f(x) = (f_1(x), \ldots, f_{\mathsf{N}}(x)),$$

and an iterative scheme or an updating mode.

From an initial configuration $x^0 \in \mathbb{B}^{\mathsf{N}}$, the $(x^t)^{t\in\mathbb{N}}$ sequence of the system configurations is constructed according to one of the following schemes:

- Synchronous parallel scheme: based on the recurrence relation $x^{t+1} = f(x^t)$. All x_i, $1 \leq i \leq \mathsf{N}$ are thus updated at each iteration using the previous global state x^t of the system.
- Unary scheme: this scheme is sometimes called chaotic in the literature. It consists in modifying the value of a single element i, $1 \leq i \leq \mathsf{N}$, at each iteration. The choice of the element that is modified at each iteration is defined by a sequence $S = (s^t)^{t\in\mathbb{N}}$ which is a sequence of indices in $[\mathsf{N}]$. This sequence is called unary strategy. This mode is set for any $i \in [\mathsf{N}]$ by:

$$x_i^{t+1} = \begin{cases} f_i(x^t) \text{ if } i = s^t, \\ x_i^t \text{ otherwise.} \end{cases} \tag{19}$$

- Generalized scheme: in this scheme, the values of a set of elements of $[\mathsf{N}]$ are modified at each iteration. In the particular case where the value of a singleton $\{k\}$, $1 \leq k \leq \mathsf{N}$, is modified at each iteration, we find the unary mode. In the second particular case where the values of all the elements of $\{1, \ldots, \mathsf{N}\}$ are modified at each iteration, we find the parallel mode. This mode thus generalizes the two previous modes. More formally, at t^{th} iteration, only the elements of the $s^t \in \mathcal{P}([\mathsf{N}])$ part are updated. The sequence $S = (s^t)^{t\in\mathbb{N}}$ is a sequence of sub-sets of $[\mathsf{N}]$ called generalized strategy. This scheme is based on the relation defined for all $i \in [\mathsf{N}]$ by:

$$x_i^{t+1} = \begin{cases} f_i(x^t) \text{ if } i \in s^t, \\ x_i^t \text{ otherwise.} \end{cases} \tag{20}$$

Where

$$F_f : \{N, \mathbb{B}^{\mathsf{N}}\} \to \mathbb{B}^{\mathsf{N}}, \qquad (i, x) \mapsto (x_1, \ldots, x_{i-1}, f_i(x), x_{i+1}, \ldots, x_{\mathsf{N}}),$$

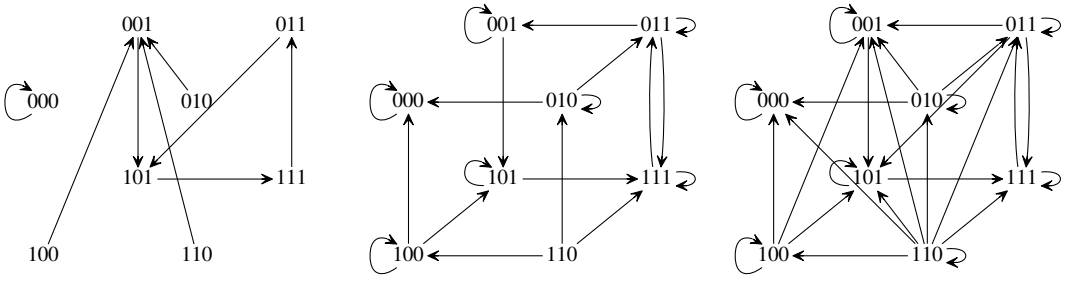
The following section describes how to graph the evolution of such Boolean domain.

2.1.2/ ITERATION GRAPHS

Let N a set of positive integer and $f : B^{\mathsf{N}} \to B^{\mathsf{N}}$, several evolutions are possible according to the iterative scheme retained. These latter are represented by an oriented graph whose nodes are the elements of $\mathbb{B}^{\mathsf{N}}$ (see FIGURE 2.1)

- The graph of the synchronous iterations of f, denoted GIS(f), is the oriented graph of $\mathbb{B}^{\mathsf{N}}$ which contains an edge $x \to y$ if and only if $y = f(x)$
- The graph of unary iterations of f, denoted GIU(f), is the oriented graph of $\mathbb{B}^{\mathsf{N}}$ which contains an edge $x \to y$ if and only if there exists $i \in \Delta f(x)$ such that $y = \overline{x}^i$. If $\Delta f(x)$ is empty, we add the edge $x \to x$.
- The graph of generalized iteration of f, denoted GIG(f), is the oriented graph of $\mathbb{B}^{\mathsf{N}}$ which contains an edge $x \to y$ if and only if there exists a set $I \subseteq \Delta f(x)$ such that $y = \overline{x}^I$. We can notice that this graph contains as subgraph both the one of the synchronous iterations and that of the unary iterations.

Example. *Lets takes the example of the preliminaries section (Section 2.1) as an illustrative example with its mapping table (Table 2.1). The Figure 2.1 shows three iteration graphs associate to f.*



(a) GIS(f) (b) GIU(f) (c) GIG(f)

Figure 2.1: Graphs of iterations function $f : \mathbb{B}^3 \to \mathbb{B}^3$ such that $(x_1, x_2, x_3) \mapsto ((\overline{x_1} + \overline{x_2}).x_3, x_1.x_3, x_1 + x_2 + x_3)$. We notice the cycle $((101, 111), (111, 011), (011, 101))$ in FIGURE (2.1(a)).

2.2/ UNARY AND PARALLEL CHAOTIC SCHEME

In this section, a recent theoretical approach, namely the so-called Chaotic Iterations (CIs [15]) of discrete dynamical systems, is reviewed. It is based on the reputed math-

ematical theory of chaos, historically defined by Devaney [4] and Li-Yorke [154] using mathematical topology and measure theories. This framework focuses on recurrent sequences of the form $x^0 \in \mathbb{R}$, $x^{t+1} = f(x^t)$, and it studies for which function f, such sequences present elements of complexity and disorder. Such chaotic sequences are candidate to provide pseudorandomness, leading to the field of chaotic pseudorandom number generators (CPRNGs). Investigating in which extent topological properties of disorder can lead to random is an attractive application of the mathematical theory of chaos. Reasons explaining such an interest encompass their sensitivity to initial conditions, their unpredictability, and their ability of reciprocal synchronization [86].

With more details, the mathematical theory of chaos, as defined by Devaney [4], takes place into a topological space $(\mathcal{X}, \tau)$. It studies the iterations $x^0 \in \mathcal{X}$, and $\forall t \in \mathbb{N}, x^{t+1} = f(x^t)$, where $f : \mathcal{X} \to \mathcal{X}$ is continuous for the topology τ. A discrete dynamical system is said chaotic when it satisfies the three following properties:

1. Transitivity: For each couple of open sets $A, B \subset \mathcal{X}$, there exists $k \in \mathbb{N}$ such that $f^{(k)}(A) \cap B \neq \varnothing$. In other words, the dynamics is intrinsically complicated, it cannot be studied using a divide and conquer approach that focuses on simpler subsets.

2. Regularity: Periodic points are dense in $\mathcal{X}$. That is, beside transitivity, we still have elements of regularity.

3. Sensibility to the initial conditions: There exists $\varepsilon > 0$ (constant of sensitivity) such that: $\forall x \in \mathcal{X}$, $\exists y \in \mathcal{X}$, $\exists n > 0 \in \mathbb{N}$, such that $d(x, y) < \varepsilon$ and $d(f^{(n)}(x), f^{(n)}(y)) \geqslant \varepsilon$. Due to the two opposite tendencies of the system (transitivity and regularity), two close points can behave in a totally different manner through iterations, the first one having a regular orbit while the second one visits the whole space. Consequently, the effects on a small error on the initial condition cannot be predicted.

Note that mathematical chaos is a very rich and well established theory comprising numerous approaches to define what is a “chaotic” or disordered, unpredictable orbit of a dynamical system. In this framework, the aforementioned definition of chaos is one of the oldest and most reputed notion of such complex dynamics; other approaches are not equivalent but complementary, defining mathematically different ways for a dynamical system to appear as “disordered”, or chaotic.

To realize the junction between this framework and iteration schemes in Section 2.1.1, the following material has been introduced [153, 155].

In the unary scheme, at each iteration t, we update the component whose index is contained in the first term of the strategy $s = (s_t)_{t \in \mathbb{N}}$. Thus, the topological disorder of chaotic iterations can be studied by defining the following:

- the function $F_{f_u} : \mathbb{B}^{\mathsf{N}} \times [\mathsf{N}]$ towards $\mathbb{B}^{\mathsf{N}}$ is defined by:

$$F_{f_u}(x, i) = (x_1, \ldots, x_{i-1}, f_i(x), x_{i+1}, \ldots, x_{\mathsf{N}}). \tag{21}$$

- the shift function $\sigma : [\mathsf{N}]^{\mathbb{N}} \longrightarrow [\mathsf{N}]^{\mathbb{N}}$ which shifts the provided strategy as an element argument to the left by removing the leading element. This is formalized

$$\sigma((s^t)^{t \in \mathbb{N}}) = (s^{t+1})^{t \in \mathbb{N}}.$$

Next, starting from the initial configuration $x^0 \in \mathbb{B}^{\mathsf{N}}$ and the strategy $(s^t)_{t\in\mathbb{N}} \mapsto s^0$. The configurations of x^t are defined by the recurrence

$$x^{t+1} = F_{f_u}(x^t, s^t), \qquad x^t \in \mathbb{B}^{\mathsf{N}}. \tag{22}$$

Thus, the chaotic iteration function G_{f_u} can be modeled by the discrete dynamic system:

$$\mathcal{X}_u = \mathbb{B}^{\mathsf{N}} \times [\mathsf{N}]^{\mathbb{N}}, \qquad G_{f_u}(x, s) = (F_{f_u}(x, s^0), \sigma(s)). \tag{23}$$

That is, performing unary iterations on the function f according to a strategy s amounts to performing parallel iterations of the function G_{f_u} in $\mathcal{X}_u$. Once we have establish the topological disorder of chaotic iterations, we define now the relevant distance d between the points $X = (x, s)$ and $X' = (x', s')$ of $\mathcal{X}_u$ by

$$d(X, X') = d_H(x, x') + d_S(s, s'), \text{ où } \begin{cases} d_H(x, x') = \sum_{i=1}^{N} |x_i - x'_i| \\ d_S(s, s') = \frac{9}{N} \sum_{t\in\mathbb{N}} \frac{|s_t - s'_t|}{10^{t+1}}. \end{cases} \tag{24}$$

Note that in the calculation of $d_H(x, x')$ called the Hamming distance between x and x', the terms x_i and x'_i are considered natural integers equal to 0 or 1 and the calculation is done in $\mathbb{Z}$. In addition, the integer part $\lfloor d(X, X') \rfloor$ is equal to $d_H(x, x')$ be the distance of Hamming between x and x'. We note also that the decimal part is less than 10^{-l} if and only if the first l elements of the two strategies are equal. Moreover, if the $(l+1)^{\text{th}}$ decimal of $d_S(s, s')$ is not zero, then s_l is different from s'_l.

Indeed, it has been proven that d is really a distance on $\mathcal{X}$, and that G_f is continuous on the metric space $(\mathcal{X}, d)$. We are then left to investigate on which conditions the iterations of G_f satisfy the three conditions of chaos. Such work has been achieved in [153, 156], by establishing the inclusion relations between sets $\mathcal{T}$ of topologically transitive functions, regular functions $\mathcal{R}$ and $\mathcal{C}$ of the chaotic functions defined respectively below:

- $\mathcal{T} = \left\{ f : \mathbb{B}^{\mathsf{N}} \to \mathbb{B}^{\mathsf{N}} \text{ such as } G_{f_u} \text{ is transitive} \right\}$,
- $\mathcal{R} = \left\{ f : \mathbb{B}^{\mathsf{N}} \to \mathbb{B}^{\mathsf{N}} \text{ such as } G_{f_u} \text{ is regular} \right\}$,
- $\mathcal{C} = \left\{ f : \mathbb{B}^{\mathsf{N}} \to \mathbb{B}^{\mathsf{N}} \text{ such as } G_{f_u} \text{ is chaotic} \right\}$.

The following successive theorems are given, the proof of which is given in [153].

Theorem 2.2.1. *G_{f_u} is transitive if and only if* GIU(f) *is strongly connected.*

Theorem 2.2.2. *$\mathcal{T} \subset \mathcal{R}$.*

It can be concluded that $\mathcal{C} = \mathcal{R} \cap \mathcal{T} = \mathcal{T}$. We then have the following characterization:

Theorem 2.2.3. *Let $f : \mathbb{B}^{\mathsf{N}} \to \mathbb{B}^{\mathsf{N}}$. The functions G_{f_u} is chaotic if and only if* GIU(f) *is strongly connected.*

At this point, we know which function f makes the output of G_f iterations have a chaotic behaviours, as defined by Devaney. Note that the iteration space $\mathcal{X}$ is constituted by Boolean vectors and sequences of subsets of $[\![1, \mathsf{N}]\!]$. It is thus infinite, while only bounded integers are required to represent its elements. G_f only manipulates Boolean numbers and bounded integers, we can thus achieve a true chaos on finite state machines.

Finally, we have recalled already obtained results regarding the Devaney's chaos of general chaotic iterations. However, this definition is not the only possible approach to formalize unpredictability and disorder aspects of an iterated system. Indeed, since four decades, mathematicians have proposed various other formulations of a chaotic dynamic, and these formulations are complementary but not equivalent: each definition provides a specific description of the complex behavior of such particular "chaotic" discrete dynamical systems. Further investigations of the real chaotic nature of the proposed chaotic iterations as mixing topology and Knudsen's definition of chaos [157] are in Annex A.1 or in [14].

2.3/ GENERALIZED SCHEME

In the generalized scheme, at t^{th} iteration, it is the set of the elements of s^t (included in $[\mathsf{N}]$) that are updated (see Equation 20). The function is defined by:

$F_{f_g} : \mathbb{B}^{\mathsf{N}} \times \mathcal{P}(\{1, \ldots, \mathsf{N}\}) \to \mathbb{B}^{\mathsf{N}}$ with

$$F_{f_g}(x^t, s^t)_i = \begin{cases} f_i(x^t) \text{ if } i \in s^t; \\ x_i^t \text{ otherwise.} \end{cases}$$

In this generalized iteration scheme, for an initial configuration $x^0 \in \mathbb{B}^{\mathsf{N}}$ and a strategy $S = (s^t)^{t \in \mathbb{N}} \in \mathcal{P}(\{1, \ldots, \mathsf{N}\})^{\mathbb{N}}$, the configurations of x^t are defined by the recurrence

$$x^{t+1} = F_{f_g}(x^t, s^t). \tag{25}$$

Then let G_{f_g} be a function of $\mathbb{B}^{\mathsf{N}} \times \mathcal{P}(\{1, \ldots, \mathsf{N}\})^{\mathbb{N}}$ in itself defined by

$$G_{f_g}(x, S) = (F_{f_g}(x, s_0), \sigma(S)),$$

In this definition, the function $\sigma : [\mathsf{N}]^{\mathbb{N}} \longrightarrow [\mathsf{N}]^{\mathbb{N}}$ shifts the provided strategy as an element argument to the left by removing the leading element (as previously).

Again, generalized iterations of f induced by x^0 and the strategy S describe the same orbit as the parallel iterations of G_{f_g} from an initial point $X^0 = (x^0, S)$. This time, the space is $\mathcal{X}_g = \mathbb{B}^{\mathsf{N}} \times \mathcal{P}(\{1, \ldots, \mathsf{N}\})^{\mathbb{N}}$.

Let consider the space $\mathcal{X}_g = \mathbb{B}^{\mathsf{N}} \times \mathcal{P}(\{1, \ldots, \mathsf{N}\})^{\mathbb{N}}$, where we define the new distance d between the points $X = (x, S)$ and $X' = (x', S')$ of $\mathcal{X}_g$ with

$$d(X, X') = d_H(x, x') + d_S(S, S'), \text{ où } \begin{cases} d_H(x, x') = \sum_{i=1}^{\mathsf{N}} |x_i - x'_i| \\ d_S(S, S') = \frac{9}{\mathsf{N}} \sum_{t \in \mathbb{N}} \frac{|S_t \Delta S'_t|}{10^{t+1}}. \end{cases} . \tag{26}$$

where $|X|$ is the cardinality of a set X and $A\Delta B$ denotes the symmetric difference, defined for sets A, B: $A \Delta B = (A \setminus B) \cup (B \setminus A)$.

The function d is a sum of two functions. The function d_H is the Hamming distance; it is also established that the sum of two distances is a distance. Thus, to show that d is also a distance, it is enough to show that d_S is one too.

For $S, S' \in \mathcal{P}(\{1, \ldots, \mathsf{N}\})$, we define

$$d_S(S, S') = \frac{9}{\mathsf{N}} \sum_{t \in \mathbb{N}} \frac{|S_t \Delta S'_t|}{10^{t+1}}.$$

Let us show that d_S is a distance on $\mathcal{P}(\{1, \ldots, \mathsf{N}\})$ and so that d defined in Equation (26) is a distance.

Let S, S' and S'' three parts of [N].

- Obviously, $d_s(S, S')$ is null if and only if S and S' are equal.
- Since the symmetric difference is commutative, the value of $d_S(S, S')$ is equal to that of $d_S(S', S)$.
- Finally, we have the following elements:

$$\begin{aligned}
S \Delta S' &= (S \cap \overline{S'}) \cup (\overline{S} \cap S') \\
&= (S \cap \overline{S'} \cap S'') \cup (S \cap \overline{S'} \cap \overline{S''}) \cup (\overline{S} \cap S' \cap S'') \cup (\overline{S} \cap S' \cap \overline{S''}) \\
&\subseteq (S \cap \overline{S'} \cap S'') \cup (S \cap \overline{S'} \cap \overline{S''}) \cup (\overline{S} \cap S' \cap S'') \cup (\overline{S} \cap S' \cap \overline{S''}) \cup \\
&\quad (\overline{S} \cap \overline{S'} \cap S'') \cup (S \cap S' \cap \overline{S''}) \cup (\overline{S} \cap \overline{S'} \cap S'') \cup (S \cap S' \cap \overline{S''}) \\
&= (\overline{S'} \cap S'') \cup (S \cap \overline{S''}) \cup (\overline{S} \cap S'') \cup (S' \cap \overline{S''}) \\
&= (S \Delta S'') \cup (S'' \Delta S')
\end{aligned}$$

From this, we deduce that $|S \Delta S'| \leq |S \Delta S''| + |S'' \Delta S'|$ and therefore that triangular equality $d_S(S, S') \leq d_S(S, S'') + d_S(S'', S')$ is established.

Similarly to unary iteration, we investigate on which conditions the generalized scheme G_{f_g} in $\mathcal{X}_g$ satisfies the three conditions of chaos. We consider first the following property [152]:

Theorem 2.3.1. *G_{f_g} is transitive if and only if* GIG(f) *is strongly connected.*

Theorem 2.3.2. *$\mathcal{T} \subset \mathcal{R}$.*

It can be concluded that $C = \mathcal{R} \cap \mathcal{T} = \mathcal{T}$. We then have the following characterization:

Theorem 2.3.3. *Let $f : \mathbb{B}^{\mathsf{N}} \to \mathbb{B}^{\mathsf{N}}$. The function G_{f_g} is chaotic if and only if* GIG(f) *is strongly connected.*

2.4/ CONCLUSION

This chapter has shown that the unary and parallel iterations are chaotic if and only if the graph $GIU(f)$ is strongly connected and the generalized iterations are chaotic if and only if the graph $GIG(f)$ is also strongly connected. We thus have a priory an infinite collection of chaotic functions, where negation and generated functions are investigated. The next chapter quantifies hardware performance of PRNGs on FPGA platform, which some of them will be chosen latter as a strategy for our chaotic iteration PRNGs.

III

Quantifying Hardware Performance of PRNGs on FPGA Platform

3

QUANTIFYING HARDWARE PERFORMANCE OF LINEAR PRNGS

This chapter resumes a deep analysis of linear PRNG (LPRNG) stated in chapter 1. This study, will help us to identify the main characteristics and proprieties that contribute to the hardware performance of each linear PRNG, using the two digital flows of High-Level Synthesis and Register-Transfer Level (HLS & RTL) as support. These proprieties are: (1) the space, timing, and computational complexity, (2) the seed and period of the generator, and (3) the arithmetic operators and dynamic range in FPGA resources. Discussion on choices of both implementation and generation are systematically given using FPGA support platform. Performance with respect to frequency, area size, weaknesses, and statistic tests are presented also.

3.1/ METHODOLOGY

The Table 3.2 presents the hardware resources, area, speed, and statistical tests of almost all aforementioned linear PRNGs in chapter 1, which will be used later for comparison with other approaches. The design methodology relies on the use of two high levels of implementation, namely the traditional Register-Transfer Level (RTL) flow and the High-Level Synthesis (HLS [158]). HLS frameworks (Vivado, HandelC, and SystemC) accelerate the semiconductor Intellectual Property (IP) creation by enabling C, C++, and SystemC specifications and by generating the RTL level. The question raised in this chapter is thus: how much space states are needed to provide pseudorandom numbers with a good statistical profile? And which algorithms outperform the other ones in terms of internal resources and time, whilst providing higher complexity?

3.2/ LINEAR COMPLEXITY

Previously presented hardware LPRNGs and CPRNG must be evaluated regarding their randomness, which can be done using statistical tests presented in Section 1.6 (NIST, DieHARD, and TestU01). After applying our experiments, we have obtained that almost all PRNGs pass NIST test but only PCG32, MRG32, and XOR64* generators can pass the Big-Crush of TestU01, the most stringent part of this battery, which is coherent with the literature. Obtained test results have shown that a particular and common test called

linearity complexity is very frequently failed, which is recalled hereafter.

For a given k-length finite binary sequence in $\mathbb{F}_2^k$ issued from a RNG, its linear complexity L_k is defined as the degree of the shortest characteristic polynomial of the LFSR that can generate the same sequence. Intuitively, non linearity is observed when this degree L_k is small. Figure 3.1 presents the linear complexity profiles of some PRNGs when applying the Berlekamp-Massey algorithm [142]. PCG32 and XOR64*, which can pass the whole TestU01, have the linear complexity property. Conversely, other PRNGs like XOR64, WELL512, TT800, and LUT-SR, fail to exhibit such a property.

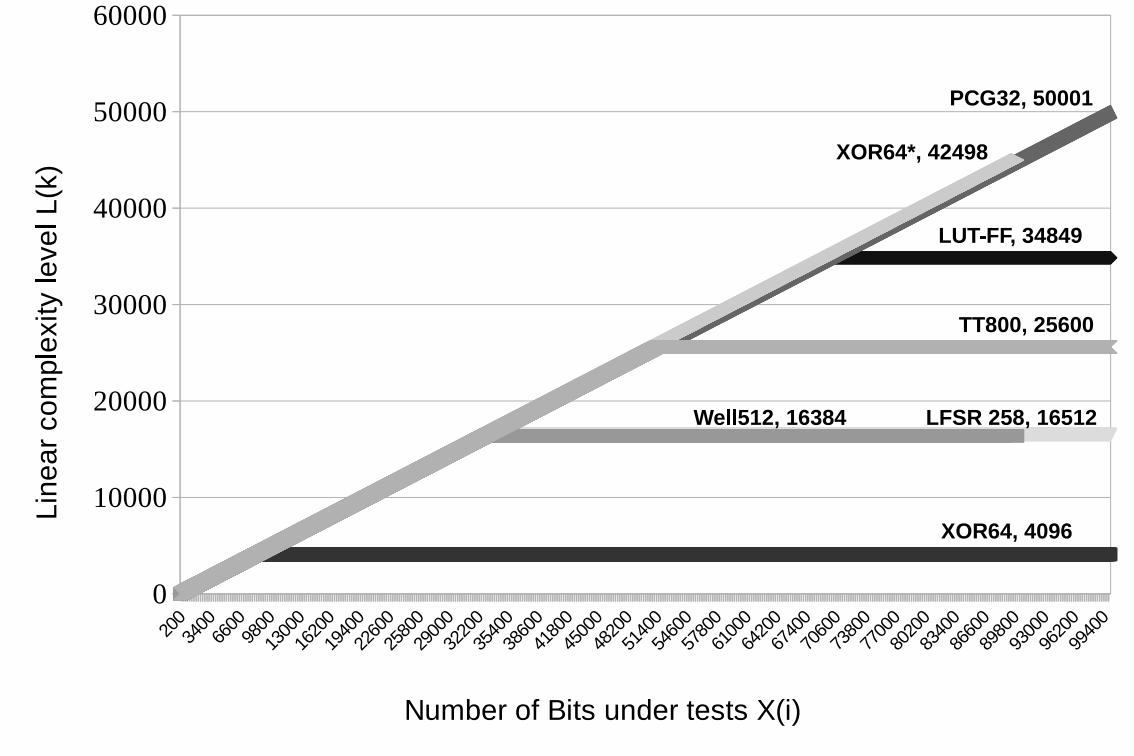


Figure 3.1: Linear Complexity profiles $L_k(x_i)$ using Berlekamp-Massey algorithm

3.3/ JUMP COMPLEXITY

TestU01 battery additionally calculates the number of jumps that occur in the linear complexity for each local subsequence. This number of jumps represents how many bits must be added to the sequence to increase its linear complexity. It has been proven [159] that ideal PRNGs have a linear complexity profile symmetric to the $k/2$-line as in a perfect linear complexity, with maximum jump heights of $k/4$, and close to $\lfloor (k+1)/2 \rfloor$ for each k-length sequence.

Lets us first illustrate some of these properties using Figure 3.2. We compute the linear complexity profiles of the first 32 bits ($k = 32$) of generators LFSR258, XOR64$*$, and PCG32 using the Berlekamp-Massey algorithm, where the complexity level $L_k(x_i)$ is expressed as follows: $L_1(x_i), L_2(x_i), \ldots, L_{k-1}(x_i)$, where $L_1(x_i) = L(x_1), L_2(x_i) = L(x_1, x_2) \ldots$ Each of these PRNGs performs jumps symmetric to the $k/2$-line as illustrated in Fig-

ure 3.2. Let us however explain some differences within these jumps. We first notice that the $L_k(x_0, x_1, x_2, x_3)$ is stable for LFSR258 and XOR64$*$. When we add x_4 to compute $L_k(x_0, x_1, x_2, x_3, x_4)$, LFSR258 jumps from 1 to 4 whereas XOR64* is still stable. PCG32, for its part, is stable for less bits and jump by 2 levels in the same interval, where the first jump happens on the x_7 and with more than 8 levels for XOR64*.

Let us consider a stream of random bits $x_i = x_0, x_1, \ldots, x_n$, in which the perfect jump is the difference between two successive linear complexity levels L_k applied to x_i and that satisfies $0 < L_k(x_i) - L_k(x_{i-1}) \leq 2$ (e.g., PCG32 has $L_k(x_0, x_1) - L_k(x_0) = (1 - 1) = 0$ and $L_k(x_0, x_1, x_2) - L_k(x_0, x_1) = (2 - 1) = 1 \ldots$).

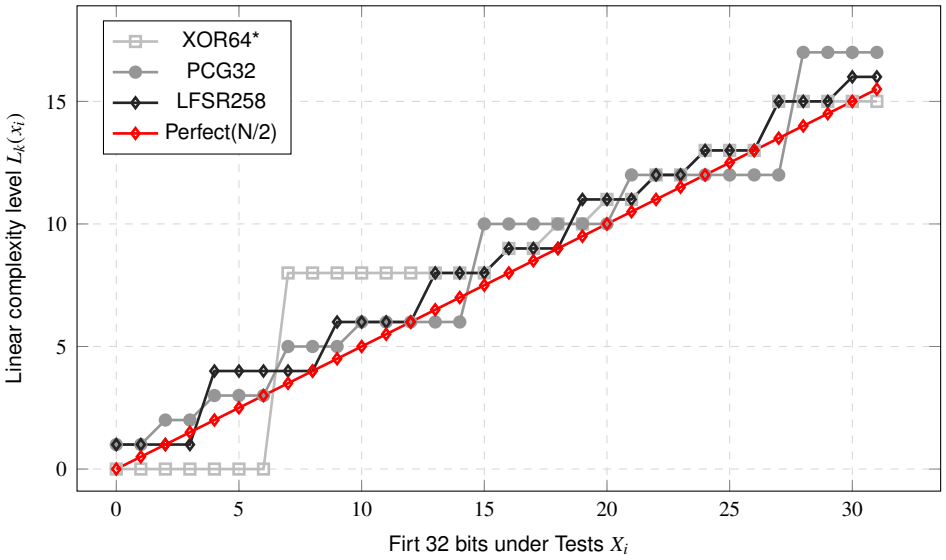


Figure 3.2: Jump Computation for 32 bits of random: number of jumps < 2 lead to a perfect $\lfloor (k+1)/2 \rfloor$) for k-sequences

Regarding FPGAs, these jumps determine how much resources are required in order to have a perfect complexity profile. For illustration purposes, some of these PRNG jumps have been computed in Figure 3.3, by starting from the linear complexity profile L_k illustrated in Figure 3.2. More precisely, we computed the jump complexity of 200 linear complexity degrees $L_k(x)$ (k = 200 bits = 6 words), on the one hand for XOR64* and PCG32 that can pass TestU01, and on the other hand for XOR32, TT800, and LUT-SR, who failed this battery.

Let us take XOR64* and LUT-SR as demonstrators of each category from Figure 3.3. The aforementioned 200 complexity linear levels illustrate that XOR64* needs a minimum of total jumps of 52 to perform a symmetric $k/2$-line (maximum jump heights of $k/4$). However, only 38 jumps are "perfect" (< 2), where $L_k(x)$ can possibly be repeated between jumps. In addition, we consider stable situations where no jump has occurred (streams of repeated $L(x) = L(x-1)$), where unstable jump is repeated only once. Indeed, we conclude that useful bits are the minimum unique bits, which does not present any form of stability in complexity profile L_k.

We can see that LUT-FF is 4 perfect jumps lower in total than XOR64*. The LUT-FF PRNG will be propagated for a long period of time, which conducts to a less useful bits contribution for passing linear tests. It is more obvious for XOR32, which confirms the need to another process to face this issue. Indeed, PRNGs that fail to pass TestU01 have the lowest number of useful bits and of perfect jumps, when compared to successful ones with

the exception of TGSRF family and PCG32. Note that XOR64* uses a multiplication as a kind of output scrambling. PCG32 has the same use in its multiplication use, so why it has less useful bits at the end while passing linearity test? To answer this question, we can focus on Figure 3.1, which illustrates the existence of stability in linear complexity starting from shorter periods of time (XOR64, LFSR258, and LUT-FF with $L_k = 4096, 16512, 34849$ successively).

Some periods can be long, as in the case of PCG32 for instance. When the PRNGs are running, the states space used is constant for any operation. Such property is obvious in 32 bitsLCG generators like the PCG32. The PCG32 deploys 64 bits multiplications (128-bit state), but it uses only 36-bit of state while always dropping the most significant bits (MSB) parts. This fact means a loss of information that can create a new jump in complexity. This is why PCG32 applies a permutation function to scramble the weak least significant bits (LSBs) after the multiplication. In other words, it needs some time to be perfectly linear. In hardware level, doing the same complex operation leads to unnecessary area and power consumption.

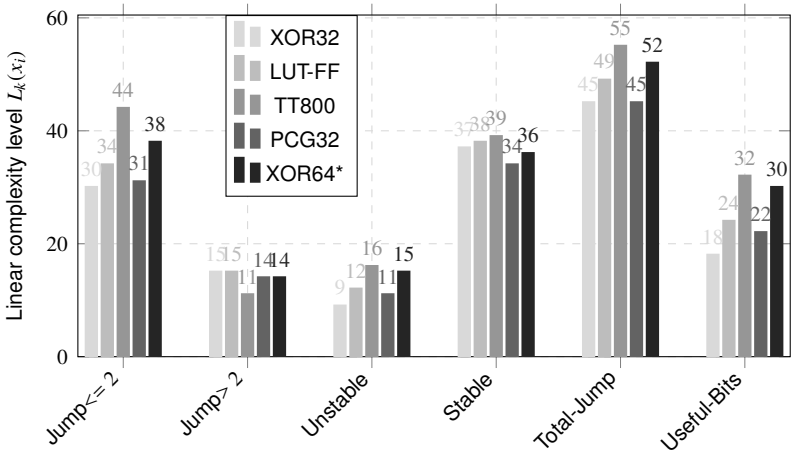


Figure 3.3: Jump computation before TestU01 of 200 linear complexity Level: a) Perfect Jump = $[0 < L(k) - L(k-1) \leq 2]$, b) other Jump =$[L(k) - L(k-1) > 2]$, c) Unstable Jump= $[L(k) - L(k-1) \neq L(k)]$, d) stable jump= $[L(k) - L(k-1) = L(k-1)]$, e) Useful bits=$[L(k) - L(k-1) = 1]$, f) Total Jump

Let us now consider the XOR64* generators, which also use 64 bits multiplications. Their linear complexity is close to the perfect one. The key difference here is the permutation function used for multiplication. In LCG family, this is the main function applied to perform an uniform scrambling operation, whereas in XOR64*, they are deployed to inject bias in randomness. Finally, we can notice the uniform distribution of Mersenne Twister, with an unique maximum perfect jump. But it has the largest stable jumps, that will finally be stable once and for all. This indicates the limitation of tempering unit (similar to XOR32 or LFSR) in terms of performance of transition unit.

3.4/ Arithmetic Operators and Dynamic Range

The arithmetic operators area is a key issue at hardware level, which can be considered as a major factor of the quality of the final implementation. In the binary field $\mathbb{F}_2$, the bi-

nary representation has a direct impact at hardware level, as each bit will turn ON or OFF one transistor. Consequently, in spite of the existence of many coding representations at mathematical level, bit lengths are physical constants[1]. More precisely, most PRNGs use only positive integer values and fixed point representations in hardware level. Regarding arithmetical operators, for instance glue logic defined as Distributed Arithmetic (DA [160]) or Digital Signal Processing (DSP48E1) slices, are usually chosen as optimal implementation of (partial) products. Their size and performance depend on both the word length (addressing the LUT increases the table exponentially) and their binary representations, regarding dynamic range and precision. Indeed, the dynamic range represents the ratio between the largest and the smallest nonzero and positive number that can be represented (integer), which is expressed as follow: $DR_{fxpt} = r^n - 1$ where r is in binary format (Radix-2) and n is the number of digits in fixed-point precision.

Table 3.1 illustrates the multiplication complexity using LUT or DA resources in FPGA, in which $0 < \varepsilon \leq 1$ and d is the digit size (fixed or floating point). Indeed, using LUT for multiplications decreases the number of clock cycles to generate the final output (which means a higher throughput). However, it has the longest critical path (latency), which increases the area $n \times 2^{2^n}$ (FF used to store intermediate results). In the other hand, multiplication based DA reduces the logic deployments, which is depending only on the dynamic range (d) and its binary representation. On the opposite, using DA increases the number of clock cycles for the output (lowest rate of throughput/latency).

Table 3.1: Multiplication Complexity using FPGA

Architecture	Logic gates or RAM cells	Flip Flops	Critical Path Delay	Clock Cycles
Full LUT	$n \times 2^{2^n}$	0	$O(n)$	1
Fully parallel	$O(n^{1+\varepsilon})$	0	$O(log(n))$	1
Digit-sequential	$O(n \times d)$	$O(n)$	$O(log(d))$	n/d
Bit-sequential	$O(n)$	$O(n)$	$O(1)$	n

source [161]

The aforementioned PRNGs in this section have a fixed DR and internal space of 32 or 64 bits. The multiplications are widely implemented with DSP blocks in FPGA, which can be used as a 25×18-bit multiplier and can be pipelined. Its complexity is linear with the "DA", otherwise it jumps higher with the use of more complicated logic as LUT or DSPs. Let us take for instance the KISS127 of $DR = 2^{64}$ as an example, which is implemented with DA (KISS-DA) or DSP blocks (KISS-DSP). It is clear from Table 3.2 that disabling DSP will induce a huge area extension and a drop in frequency while presenting the same latency. As a conclusion, DSP blocks can be a convenient alternative for FPGA application. However when ASIC implementation is targeted, these DSP blocs have to be manually implemented, giving rise to a direct loss of performance.

3.5/ THROUGHPUT AND LATENCY

Let us recall two proprieties based on the frequency, which are namely the latency and the throughput. Many research papers deal with board frequency, clock input, or synthesis

[1] Dynamic allocation is however possible for modern FPGA

frequency, but few of them illustrate the effect of the design frequency in terms of latency and throughput.

Latency is the number of iterations required to compute a new output from a given input. The throughput, for its part, is the number of iterations needed to produce new output or to consume a new input. In FPGA design, an iteration is the clock cycle or the period (the inverse of the frequency). Latency and throughput are linked to the delay expressed as a duration or as the number of clock cycles to have an output. To sum up, latency is the delay from the input to the output, whereas the throughput is the delay to get one new output. Finally, rate is the number of bits that are treated or transferred in each delay unit (Bps). Note that the throughput delay can be equal to the latency, which lead us to use the throughput/latency value to estimate the real throughput of the PRNG. Finally, let us recall that many techniques can improve these properties, like pipelining, parallel computation, and retiming techniques.

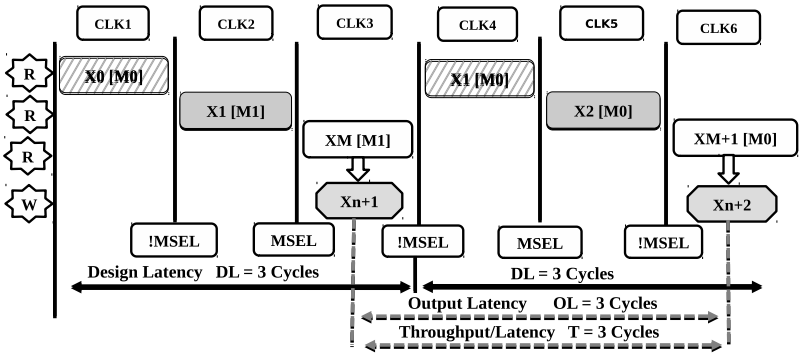
Latency and throughput in the RTL and HLS flows can be formalized as follows.

$$\begin{aligned} \textit{RTL Delay} &= (\#\textit{Clock Cycle}) \times (\textit{Clock Period}) \\ \textit{HLS Delay} &= (\#\textit{Clock Cycle} + 2) \times (\textit{Clock Period}) \end{aligned}$$

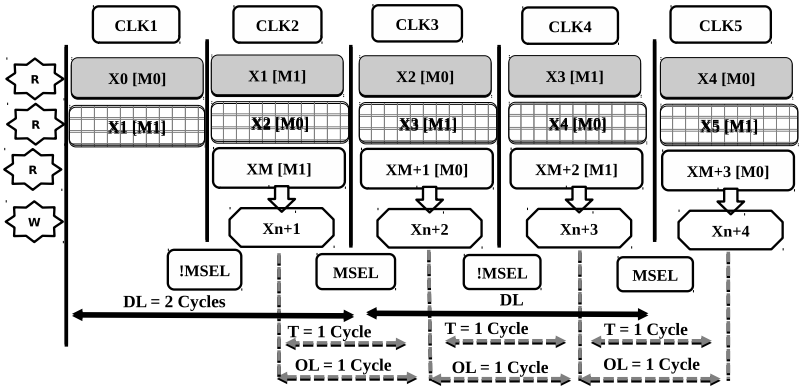
Thus,

$$\begin{aligned} \textit{Design Latency} &: [\textit{Delay from Input to Output}] \\ OutputLatency &: [\textit{Delay for each Output}] \\ Throughput &: \left\lceil \frac{\textit{Output Size}}{\textit{Output Latency}} \right\rceil \end{aligned} \tag{27}$$

Let us explain this fact with the Mersenne Twister implementation in RTL level. Mersenne Twister deploys two dual-port 1×312 BRAM memories, $M0$ and $M1$, which operate like a feedback shift register and which are configured in the read-before-write mode. Additionally, an address controller is used to read the three words ($X0$, $X1$, and XM) and write the result according to Equation (9). The Mersenne Twister follows a permutation mode (Memory SELect "MSEL") to choose on which memory the R/W operation will be operated. In this context, Figure 3.4 presents an example of two R/W modes, illustrating the difference between throughput and latency. The first mechanism reads one word per cycle and writes the output in the third clock cycle. It reads the first word X0 from M0 in the first cycle when MSEL=0, and X1 from M1 in the second cycle when MSEL=1, where final cycle reads the middle word XM and writes the output in the same address of X0 in M0. In this case, the throughput is equal to the latency, and the final rate (throughput/latency) is divided by latency delay (*i.e.*, by a factor 3). Conversely, the second mechanism of Figure 3.4 produces a new output at each clock cycle, which summarizes read and write operations in one clock cycle. Therefore, if the memory select mode MSEL is equal to 0, it reads $X0$ from M0 and $\{X1, XM\}$ from M1 following the read address controller. It writes the output in the same address of the first word and we increment all addresses. The opposite happens when MSEL=1: $\{X2, XM + 1\}$ is read from M0 and $X1$ from M1. So, the rate (throughput/latency) in the last implementation is calculated for one cycle, as reported in Table 3.2.



(a) Throughput/latency is 3 cycle



(b) Throughput/latency is one cycle

Figure 3.4: Latency vs. throughput in two MT implementations: read three words X_i, X_{i+1}, and XM (middle) from two BRAM memories $M0$ and $M1$ and write the output.

3.6/ EXPERIMENTAL RESULTS

The aforementioned PRNGs have been studied according to: (1) the space, timing, and computational complexity, (2) the seed and period, (3) the arithmetic operators and dynamic range FPGA resources, and (4) latency and throughput. All the linear PRNGs have been implemented in RTL flow, except TT800 and PCG2, for which we followed a HLS flow. Concerning the synthesis platform, Vivado synthesis for RTL flow of Xilinx v16.4 and Vivado HLS tool for HLS flow have been used, with the default configuration and without any optimisation. Additionally, the FPGA target was Zybo Zynq-7000 ARM/FPGA SoC Trainer Board from digilent (125Mhz).

Table 3.2 summarizes the obtained results with these PRNG implementations. In RTL implementations, LUT-SR, Taus88, and XOR64 require the lowest amount of area resource, whereas combined PRNGs like KISS124 and MRG32 have a large area consumption. Conversely, LCG and TGFSR families have large area consumption due to their implementation of arithmetic multiplication with complex logic.

The throughput performance depends on two parameters, namely the logic critical path and the output range (32 or 64 bits). On the one hand, for 32 bits generators (resp. for 64 bits ones), Taus88 and LUT-SR with LFSR113 (resp. XOR64 and LFSR258) have the largest throughput performance. On the other hand, the LCG and TGFSR families are

Table 3.2: FPGA implementation of linear PRNG in term of: Area, Speed, and Statistical tests

		Linear PRNG															
Familly		LFSR				xorshift				TGFRS				LCG			
PRNG		LFSR113	Taus88	LFSR258	LUT-SR	XOR128	XOR64	XOR128+	XOR64*	MT_WS	MT_NS	Well512	TT800	KISS-DA	KISS-DSP	MRG	PCG32
Output Rang (n)		32	32	64	32	32	64	64	64	32	32	32	32	64	64	64	32
AREA	LUT	79	68	171	64	36	55	131	298	523	184	94	184	271	2038	1055	345
	FF	162	130	386	64	194	130	194	390	120	179	108	483	746	1277	1359	418
	RAM	0	0	0	0	0	0	0	10	2	2	0	6	0	0	0	10
	DSP	0	0	0	0	0	0	0	4	3	0	2	2	7	0	8	0
	Total Area (LUT+FF)*8	2072	1584	4456	576	1840	1480	2600	5504	5144	3272	1616	5336	8136	26520	19312	6104
SPEED	Frequences (Mhz)	443,26	448,63	396,98	609	429,36	457,45	250,81	231	118	462	213	169	154	112	175	179
	Design Latency	2	2	2	2	2	2	2	21	3	2	5	4	9	9	14	20
	Output Latency	1	1	1	1	1	1	1	21	1	1	5	4	9	9	14	20
	Throughput/Latency (Gbps)	14,18	14,35	12,70	19.5	13,73	14,63	8,02	0.7	3.8	13.2	1.3	1.3	1.1	0.8	0.8	0,286
TESTS	NIST (16 Tests)	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	NO	PASS	PASS	PASS	PASS
	TestU01 (319 Tests)	NO	NO	NO	NO	NO	NO	NO	PASS	NO	NO	NO	NO	NO	NO	PASS	PASS

a HLS Implementation, 1. MT_WS: Mersenne Twister with Seed, 2. MT_NS: Mersenne Twister without Seed.

expected to have the lowest throughput performance, as they operate large arithmetic operations like 64 bits multiplications using DSP. Besides that, using memories for TGFSR automatically drops the PRNG frequency to the half without counting other logic. Once again, the combined generators have the weakest throughput performances.

Additionally, two implementations of Mersenne Twister generators have been designed with and without the seed, respectively denoted as MT_WS and MT_NS. We have remarked that, when considering the seed, frequency is reduced to less than 200MHz compared to the case where we do not use it. Therefore, to increase performances, most PRNGs do not include the seed internally (a software is used).

To put it in a nutshell, if we take the ratio of area/throughput as main criterion, we are balancing between high performance (XOR64 and LFSR113) and the ability to pass statistical tests (PCG32 and XOR64*), which is not surprising. Another result is that combining PRNGs leads to a performance decrease in hardware level. Such combinations do not take into account the Chaotic Iterations post-processing, which appears as promising [10, 11]. Effects of such a post-processing on performances at hardware level are detailed in the following chapters.

3.7/ CONCLUSION

An implementation of various linear PRNGs in FPGA is detailed in this chapter, in which two flows of conception (RTL and HLS) demonstrate the performance level of each PRNG in terms of area throughout and statistical tests. Our study has shown that these performances are related to linear complexity, seed size, and arithmetic operations. On the one hand, for 32 bits generators (resp. for 64 bits ones), Taus88 and LUT-SR with LFSR113 (resp. XOR64 and LFSR258) have outperformed the other candidates when considering hardware performance, while PCG32 and XOR64* are the best when studying statistical ones (they succeeded to pass the whole TestU01 batteries). In order to investigate these parameters, two FPGA test platform (hardware and firmware) has been developed to accelerate the implementation and tests of various PRNGs. These platforms are presented in the next chapter.

4

HARDWARE TEST PLATFORM AND COMPARISON

This chapter presents all hardware platform of FPGA and ASIC deployed to implement, analyze, and test all PRNGs in this thesis.

The first section describes two hardware platforms based on FPGA, in which each has its limitations and advantages during their use in our thesis. The first FPGA platform is based on Xilinx Zynq-7000 Extensible Processing Platform (EPP) (Section 4.1). It is mainly used in the first time to accelerate the hardware implementation and test of the PRNGs in an embedded system (SoC) using an automated flow of tools on FPGA. However the main limitations of the Zynq platform are its ability to produce data in real time for statistical tests, the hardware/software time complexity using embedded SoC, and the transition into ASIC implementations. Therefore, we propose a second platform based an AXI Bus compatible to ARM CPU (Section 4.2) as an alternative of the first platform. The new FPGA platform is indeed more reconfigurable, compatible to any SoC based FPGA, and generic to ASIC applications. Moreover one major reason for using the AXI Platform is the capability of testing in real time any PRNGs on FPGA under TestU01. Finally, this chapter ends with the ASIC platform (Section 4.4) based on the process node of 65-nm UMC, which is an indispensable flow for fabricating a real chip based PRNG.

4.1/ FPGA PLATFORM BASED ON ZYNQ-EPP FOR PRNG

4.1.1/ GENERAL PRESENTATION

Xilinx Zynq-7000 Extensible Processing Platform (EPP) [8] is a silicon system on chip (SoC) for FPGAs, which has been proposed by Xilinx. This SoC deploys the latest technologies of ARM processors with a large set of peripherals (DDR, PCI, etc.). The zynq platform is defined as Peripheral System (PS), which is a sub-system with ARM. The full FPGA is the Programmable Logic (PL) that is connected with PS through an Advanced eXtensible Interface (AXI) protocol interface. We have used the Xilinx Zybo board ($XC7Z010-1CLG400C$) whose clock input is configured at $125Mhz$ as a prototype kit for our experiments.

Figure 4.1 illustrates the Xilinx Zynq-7000 EPP block diagram. Obviously, the AXI interface plays an important role for communication and synchronization between the two parts. This interface can be a master or a slave of the PS, each having a specific port.

Indeed, AXI-4 is part of ARM AMBA Bus 4.0 (Advanced Microcontroller Bus Architecture).

Meanwhile, the CPU initializes and reads/writes data of an IP in PL (i.e., PRNG) over the AXI master using General Purpose (GP) ports. The AXI slave is used for PL master IP over High Performance (HP) ports. Each of these interfaces can handle up to 16 bytes of data.

The AXI interface protocol, for its part, can be configured either as Stream for high-speed streaming data, or as Lite/Full for high-performance memory-mapped requirements (data transactions over an address). We will focus in this manuscript on the stream protocol. Therefore, we have developed a complete infrastructure divided in two parts: hardware and firmware.

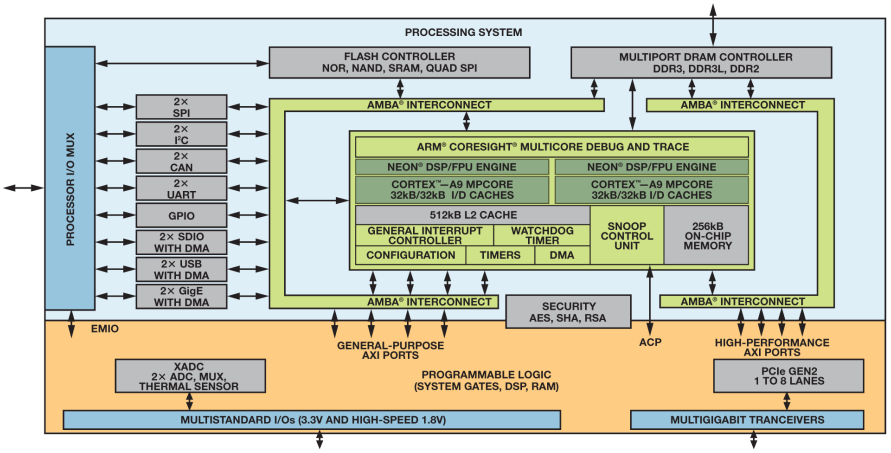


Figure 4.1: Xilinx Zynq-7000 EPP Block Diagram

4.1.2/ HARDWARE PLATFORM

Figure 4.2 and Figure 4.3 show the detailed hardware architecture of our system used to integrate and test PRNGs. It contains, respectively: the ARM Cortex-$A9$ dual cores MPSoC, the high performance DDR3 512Mb, an UART, and finally the PRNGs (RTL or HLS implementation). Additionally, to read the random output on the CPU, we have used both an AXI-PRNG interconnect and an AXI Direct Memory Access controller engine (DMA). The total space of the logic part (PL) on Zybo board is: $2,982$ LUT (19% among the 56658 available ones), $4,071$ FF (11% among the 44781 available ones), 7 DSPs, and 3 memories respectively.

The AXI-PRNG interconnect can handle many PRNGs at the same time and it activates the one that is currently tested. This interconnect component is re-configurable using the firmware, which deploys two GPIO IPs for this task. GPIO-0 is used to select one PRNG at a time, and GPIO-1 is used for the data burst size of the PRNG. As said previously, the AXI bus can handle many master and slave IPs, where the data transaction and the interfacing between IPs can be done following the three protocols cited earlier. For instance, all PRNGs implemented in HLS or RTL including the AXI-PRNG interconnect are AXI Stream Interface, while the CPU is Memory-Mapped Interface. Additionally to CPU, the AXI-DMA engine, which oversees the data transaction between the slave and master IPs, deploys the receiver channel Slave to Memory Map (S2MM) connected to a

slave port and the transmitter channel Memory-Map to Slave (MM2S) connected with the master. The DMA engine needs to be configured first depending on its burst transaction setup, which can handle up to 256 data transfers (total length of 1MB). As can be deduced from the explanations above, the process is unidirectional for pseudorandom generation: the final outputs are displayed in an external terminal via the UART protocol. Indeed, Vivado tools are used for hardware platform, while the firmware will help to synchronize all these processes and transactions using another platform.

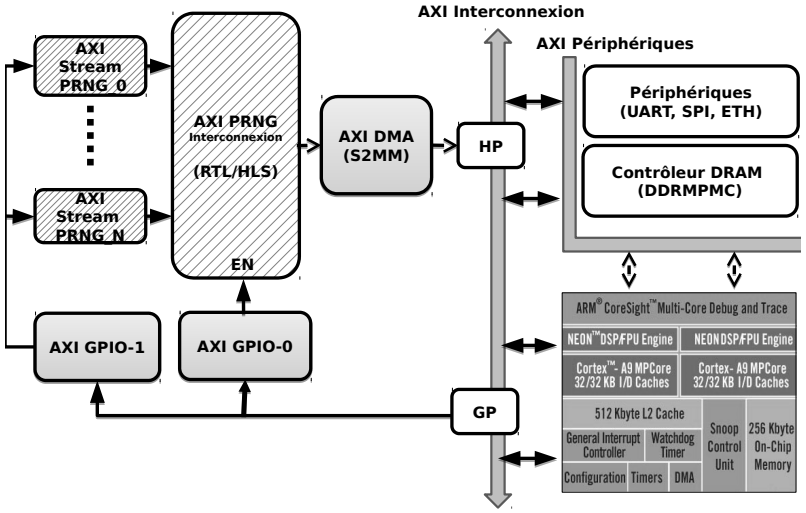


Figure 4.2: PRNG platform based on Zynq FPGA

4.1.3/ SDK FIRMWARE

As what has been said previously, the firmware is used to initialize the system, for transaction synchronization, and for the interface with an external peripheral. SDK software from Xilinx is used for this purpose. It imports first the address map of all hardware IPs that are used, and then it creates their equivalent in software. The system is interrupted to initialize the CPU and the DMA following their base address, and GPIOs are initialized to configure the PRNGs. Finally, we start transferring data using DMA to DDR memory controller and then to UART. The transaction bandwidth can reach 2GBps working with 125MHz FPGA (Zybo), and 4Gbps in DDR.

4.2/ NEW RECONFIGURABLE FPGA PLATFORM FOR CIPRNG

In this section, we propose a new FPGA test platform specially for the proposals based on chaotic iteration CIPRNG (unary and generalized) presented in Chapter 2.

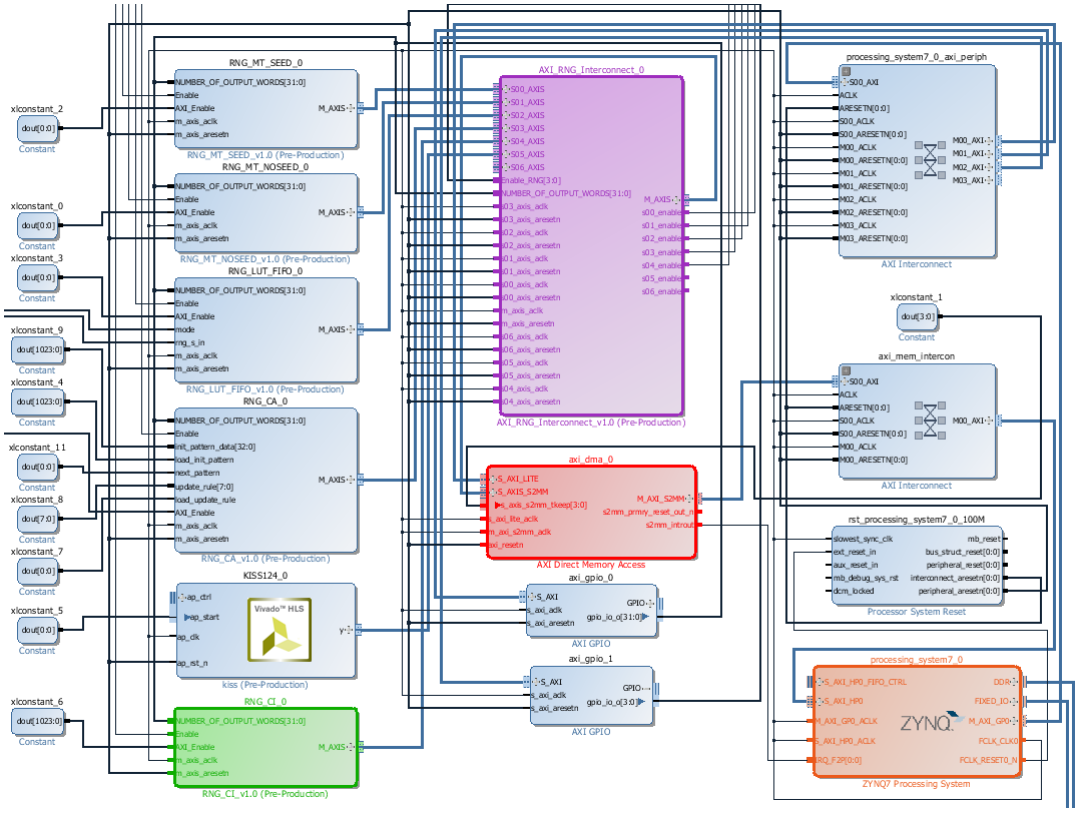


Figure 4.3: Detailed Zynq based SoC implementation for PRNG

4.2.1/ General Presentation

The new platform presents an alternative hardware and test concept of the Zynq one that has been proposed in our previous research work [12]. However, now, the platform becomes fully independent of any CPU (Zynq) and it is fully reconfigurable with a main software. It is also based on AXI-4, which makes it flexible and easy to integrate with Zynq platform for SoC applications (we named it an AXI-test platform). This pool of resources is typically independent of the technology, and can thus be implemented in an ASIC for instance.

4.2.2/ Hardware Platform

Figure 4.4 presents the main architecture of the AXI-test platform, which consists of the following components. A Decoder Command Controller Unit (DCCU), a Design Under Test (DUT) controller based on the GCIPRNG, and an Universal Asynchronous Receiver Transmitter (UART) serial port. All these units are compatible with the AXI-4 Lite bus protocol, which resumes the data transition and handshaking communication between units. Additionally, each of these units has an address map and an identifier (ID), which can be read and reconfigured. On the one hand, the DCCU decodes all commands received from both UART (PC-FPGA-PC) and DUT controller. It also chooses the strategy used in the GCIPRNG. Additionally, the DCCU defines latency of the final outputs, and the read&write operations in the internal registers of the platform (UART, strategy, and GCIPRNG that is tested). On the other hand, the DUT controller is an AXI-4 lite wrapper

of GCIPRNG, which decodes the commands received from the DCCU to enable a strategy (a linear PRNG), read/write to internal registers, and controls internal latency for any GCIPRNG configuration. The UART, for its part, is a simple serial communication to the personal computer, with $115200Kbps$ bandwidth configuration. Finally, a software application is deployed with this platform, to have a full control and access to the GCIPRNGs tested in FPGA. Note that this AXI-test platform allows too the runs of TestU01 statistical tests in real time.

For the experiments, the test platform is designed and implemented using Xilinx Vivado tools and two FPGA prototype boards, namely the ZYBO board and Nexys V.4 Artix-7. The system is embedded with at least 3 strategies for the GCIPRNG core (unary, parallel, or generalized chaotic iterations), where the hardware resources are 2.5 times lower than in the Zynq platform (see Section 4.1.2), with 1211 LUT (1.19%), 1467 FF (1.16%), and $211Mhz$ respectively.

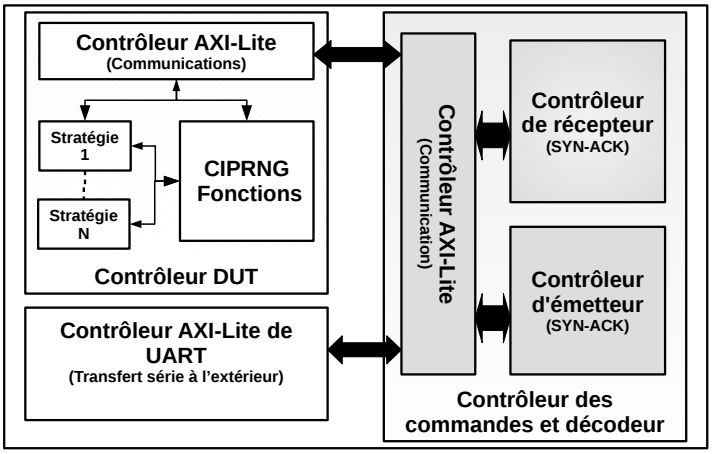


Figure 4.4: CIPRNG platform based on AXI BUS FPGA

4.2.3/ FIRMWARE

Unlike the SDK firmware embedded in Zynq platform, the new firmware is completely independent from third software part (SDK). Having said that, the firmware consists of three main parts and can be summarized as follows. Firstly, the UART communication setups a serial communication between the platform and PC (opens and closes a USB virtual serial port). In order to establish that, the firmware must be synchronized with hardware configuration of UART implemented in FPGA (i.e. same set speed to $115200bps$, 8n1 (no parity)). The second part is the configuration of the platform internal registers for different operations for the initial setup. In other words, it executes a serie of reads and write operations in parallels. Concerning the write operation, it targets the internal registers for configuration, as: defines if it writes or reads operations to internal states, defines latency, selects a strategy (linear PRNG), or resets to initial states. The read operation for instance, captures any responses type from platform after write operation. Finally, the third part runs the PRNG as a random number generator, with in mind, write and read are running in parallel.

Lets takes an example of read an ID for UART. Firstly, we write to the platform to indicate that it is a read operation on the internal register using series of commands. Then, the platform answers back by sending the write address to confirm write operation. Finally, it sends after in one packet the read address to confirm the read operation in one hand and the ID of the UART in terminal in other hand. Each packet concatenates two bytes to indicate the beginning and the end of the packet, in which they are in different size depending on operations (READ&WRITE).

4.3/ FPGA GLOBAL COMPARISON

As stated previously, the objective is to determine the performance of PRNG implementations in terms of area (space) and throughput (speed). Therefore, for all our results, the FPGA based comparison is as follows. The Xilinx tool calculates all resources used in FPGA as logic gates, LUT, Flip-Flop (register), additionally to DSP and memory blocks. Despite the fact that Xilinx calculates the area by counting slices (1 Slice = $4\times$ LUT + $2\times$ FF+interconnection), it uses the same LUT of 6-inputs for all its technologies for modern FPGAs (Virtex5, Virtex6, Virtex7, and Zynq). Hence, for our area comparison, we only calculated LUT and FF as $[(LUT + FF) \times 8]$, since DSPs and RAM memories are hard blocks that can mostly affect time performances. The throughput performance is calculated as in Equation (27). It depends on two parameters, namely the logic critical path that defines the period (design frequency) used and the output range (32 or 64 bits). Additionally to the single throughput, the *throughput over the latency* is a complementary parameter to deeply analyze the speed of the generators.

4.4/ ASIC PLATFORM FOR PRNG

4.4.1/ GENERAL PRESENTATION

Compared to FPGA flow, the ASIC one consists of implementing our design in a specific process technology at transistor level or known as RTL2GDS Flow (see Figure 4.5). The ASIC flow summarizes in digital the use of what is called standard cells, which include combinatorial and sequential basic circuits. Thus, the size and their physical characteristics determines the level of results. In our case, UMC-$65nm$ LL represents the process technology node, where the Cadence tools v14 are the main software for the implementation purpose.

4.4.2/ ASIC ANALYSIS

The figure 4.6 summarizes the ASIC flow for digital circuit, which uses two global flows: the synthesis flow using Cadence RTL Compiler, and physical place and route (P&R) flow in a second step, with Cadence Encounter Digital Implementation. Both flows include Switching Activity Interchange information generated from simulation process for timing and dynamic power estimation (1 million samples). In addition, signoff verification flow is used to close timing and power requirements. The condition operation mode for the technologies deployed in each flow is as follows: the synthesis is based on one mode

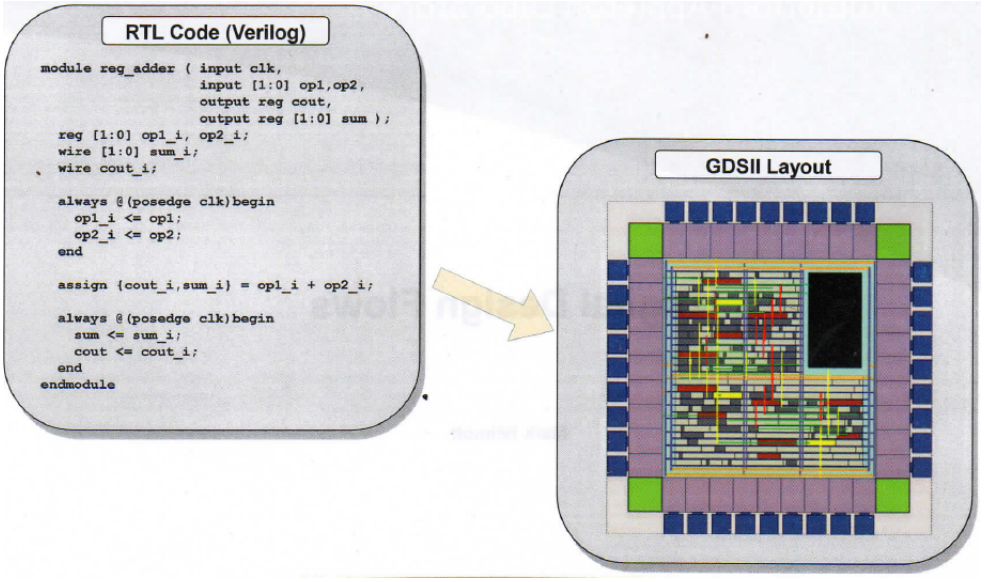


Figure 4.5: ASIC implementation goal

using the Worst Case library (WC=$108°$C and 1.08 Volt), while Multi Mode Multi Corner is applied for P&R flow including both worst and best case library (BC=$-40°$C and 1.32 Volt). The analysis results of ASIC implementation of various PRNG can be summarized as follow.

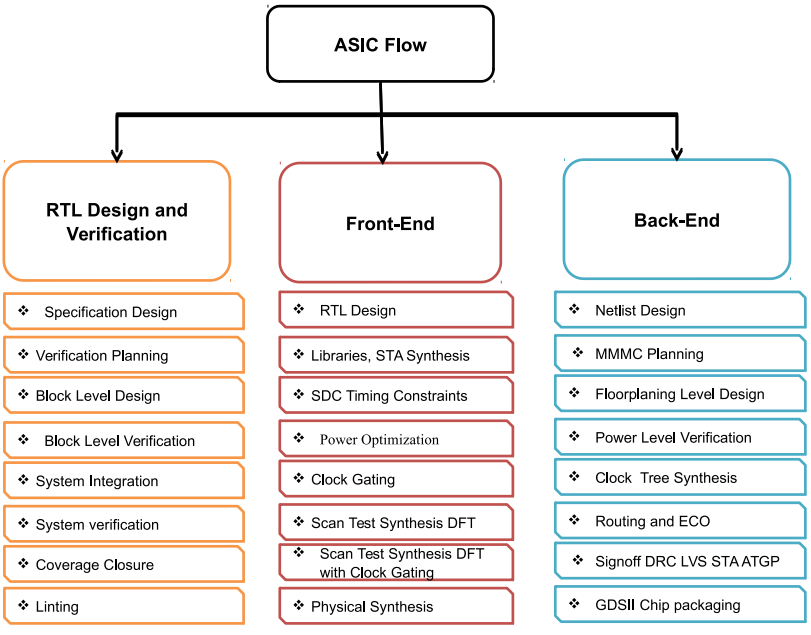


Figure 4.6: General ASIC Flow based on Cadence Tools

4.4.2.1/ AREA ANALYSIS

When dealing with ASIC implementations, two measures can be considered to evaluate area consumption: either the Gate Equivalent (GE) or the number of transistors, which

are recalled hereafter. The former is used as a generic result. The number of GEs is obtained by dividing the area of the circuit in μm^2, obtained using P&R tools, by the area of a basic logic NAND gate (in μm^2 too). The total number of transistors, for its part, is equal to GE$\times 4$, as the number of transistors in an AND gate is 4. UMC-65 Low Leakage being our process technology, its AND area is equal to $1.44 \mu m^2$.

4.4.2.2/ Static Timing Analysis

Physical implementation flow introduces a large amount of changes when compared with RTL design (i.e., datapath transformation). Additionally, static timing analysis (STA) alone does not cover all design aspects, and we cannot be sure that the physical place and route have not broken the design. This limitation cover only the synchronous part and not the asynchronous part of the design. Therefore, the Gate-Level Simulation is a full time one, when accurate net delay is known using back-annotation information delay (SDF). This static timing analysis not only checks the functional mode, but also collects the switching activity information for power analysis.

4.4.2.3/ Power Analysis

Each power pin can carry a limited amount of power. The more power the chip consumes, the more package pins are required to power the chip. This contributes to larger package costs. Also, the higher the power requirements of the chip, the denser the power grid will be in order to distribute power across the chip. This means more power lines, which reduce the wire tracks available for routing signals. Two type of power analysis are used: the Average Power Analysis based on toggling rate (nbr of toggles/time) is collected from simulation level (RTL/Gate); in other hand, the Peak Power Analysis which calculates power for each event collected also from simulation and estimates glitch power. Concerning power analysis, we estimate both static and dynamic power, which computes leakage and switching&internal power of the design, respectively. The leakage power measures each cell (logic) in various states, while dynamic power depends on the initial state of cells, the toggling input, the transition rate, and the output capacitive load.

4.5/ Conclusion

This chapter focuses on differences between hardware platforms used for our implementation and test of all PRNGs presented in this thesis. Despite Zynq FPGA platform accelerates the prototyping and the SoC based ARM integration, it is fully dependent on FPGA and hard to adapt to ASIC applications. In addition, the only way to test with TestU01 the PRNG with the Zynq platform is to generate up to 2^{38} sequences and save them into a file (up to 1̃Terabit for BigCrush). Therefore, the new platform gives that opportunity with less resources and with a simpler software control. Where, the statistical tests with TestU01 is done in real time in software level (each call function for the PRNG from TestU01 is equivalent to a successive read and write to the platform with no file or storage needed). Additionally to FPGA platform, ASIC platform is also deployed to the analysis for all PRNGs in this thesis.

IV

From Unary to Parallel Chaotic Iteration PRNG

5

UNARY CHAOTIC ITERATION PRNG: CIPRNG MULTI-CYCLE AND XOR

In this chapter, we have presented a new family of post-processing PRNGs based on chaotic iterations for FPGA and ASIC. Where, the results of linear PRNG in the previous Chapter 3 are used as sources of information in the design of an hardware post-processing treatment based on chaotic iterations. Therefore, the main contribution of this chapter is to propose two post-processing modules in hardware, to improve the randomness of linear PRNGs while succeeding in passing the TestU01 statistical battery of tests. They are based on chaotic iterations and are denoted by CIPRNG-MC and CIPRNG-XOR. They have various interesting properties, encompassing the ability to improve the statistical profile of the generators on which they iterate. Such post-processing have been implemented on FPGA and ASIC without inferring any blocs (RAM or DSP). Finally, a comparison in terms of area, throughput, and statistical tests, is performed.

5.1/ CIPRNG MULTI-CYCLE

As described in Chapter 2 (Section 2.2), the general chaotic iterations PRNG receives an integer sequence as input (and the first internal state, a binary vector), and it produces a sequence of binary vectors. In other words, chaotic iterations translate a sequence in another sequence. This is a way to obtain a new pseudorandom number generator from a former one. Both the kind of embedded generator and the iteration function f are parameters of this post-treatment, while the first vector x^0 and the first term S^0 act as seeds. Both x^0 and S^0 are the initial condition of the discrete dynamical system of Equation. (19). Finding f such that this dynamical system behaves chaotically seems to be interesting in a pseudorandom generation context. In other words, we hope that chaos bring by the iteration function will lead to a more disordered output $(x^t)_{t\in\mathbb{N}}$ than the input $(S^t)_{t\in\mathbb{N}}$. Even if there is, *stricto sensu*, no theoretical relation between randomness and chaos (similarly, there is no relation between security and chaos), numerous simulations have illustrated [153, 155] that, due to chaos, the output sequence is in general more random than the input one, according to the number of statistical tests they can pass. It is as if this post-treatment rectifies the input signal by adding extreme sensitivity due to the chaotic property of G_f, and that this high sensitivity improves the statistical profile of the output.

Such chaotic iterations based post-treatment over existing PRNGs can be designed as

follows. As we need to generate a sequence $(S^t)_{t\in\mathbb{N}}$ of subsets of $[\![1, \mathrm{N}]\!]$, we can consider two input generators, both producing numbers in $[\![1, \mathrm{N}]\!]$. The aim of the first generator is to provide, at each iterate t, the size of the subset S^t, while the second generator produces the content of S^t. This way to post-operate over the input generators is what we called CIPRNG Multi-Cycles (CIPRNG-MC). The aforementioned CIPRNG-MC is already chaotically proved is previous work in [162]. However, we only concentrate on the hardware implementations with optimization of this version with much larger statistical tests bench.

Indeed, the CIPRNG-MC which is a sub-category of our CIPRNG post-treatment, can be summarized as follows [156]. x^0 is the initial Boolean vector of size N, and $(S^t)_{t\in\mathbb{N}}$ is the sequence resulting from the irregular decimation d_s of $(PRNG1, PRNG2)$. We suppose that this latter produces numbers belonging to $[\![0, 2^{\mathrm{N}} - 1]\!]$. Operating G_f with the vectorial negation on such sequences can be directly rewritten as follows [156]:

$$x^{t+1} = x^t \otimes S^t, \tag{28}$$

where S^t is expressed in the base-2 numeral system as a binary vector of size N, while $\otimes$ is the bitwise XOR operation over binary vectors. In other words, CIPRNG-MC is equal to the chaotic iterations with the vectorial negation and the decimation S of the two inputted generators $m = PRNG1$ and $b = PRNG2$.

The basic design procedure of this latter is summarized in Algorithm 1. The internal state is x, the output state is r. The internal values a and b are computed by the two input PRNGs. Lastly, the value $g(a)$ is an integer, defined as in Equation. 29. To do so, a sequence d_s $(= (d^1, d^2, \ldots, d^N) \in \{0, 1\}^N)$ called a irregular decimation is provided for the second generator b, which insures that we do not have two successive permutations of the same bit within a given iteration. This latter will update the i-th bit of b at iteration m^t, and by using the strategy, if and only if $d^{b^i} \neq 1$, otherwise it is discarded. For instance, let us consider the input $x = \{x^1, x^2, x^3, x^4\}$, the number of iterations $m^t = \{4, 3, 4, 1\}$, and $b = \{2, 3, \underline{\mathbf{1,1}}, \mathbf{4,4}, 3, 1, 2, 3, \underline{\mathbf{2,2}}, \mathbf{4,4}, 1, 2\}$. Due to the first value of m^t, we have to iterate 4 times b, and then 3 and 4 times. We then have to operate the decimation on b so that we will not modify twice a same component in a given iteration: this leads to the strategy $S = \{\{2, 3, 1, 4\}\{4, 3, 1\}\{2, 3, 2, 4\}\{4\}\}$ extracted from b. As can be seen, the duplicated entry 2,2,4,4 has been decimated to 2,4, while it is not the case for the first 4,4, as according to m^0 this duplication falls between two iterates. This constraint explains the general form of m^t provided in Equation. 29.

$$m^t = g(y^t) = \begin{cases} 0 \text{ if } 0 \leqslant y^t < C_{32}^0, \\ 1 \text{ if } C_{32}^0 \leqslant y^t < \sum_{i=0}^{1} C_{32}^i, \\ 2 \text{ if } \sum_{i=0}^{1} C_{32}^i \leqslant y^t < \sum_{i=0}^{2} C_{32}^i, \\ \vdots \quad \vdots \\ N \text{ if } \sum_{i=0}^{N-1} C_{32}^i \leqslant y^t < 1. \end{cases} \tag{29}$$

Note that, most of the time, we need to iterate the second generator more than the cardinality of S^t, as we can obtain twice the same number. This weakness is at the origin of the second proposal, namely the CIPRNG-XOR. We resume hereafter the foundations of this algorithm already presented in [156].

Finally, a new extended version of CIPRNG-MC can be applied as post-processing for TGFSR family (as Mersenne twister and TT800) named CIPRNG Multi-Cycle Multi-

Algorithm 1 CIPRNG-MC proposal. At each iteration, only the S^t-th component of state x^t is updated, as follows: $x_i^{t+1} = x_i^t$ if $i \neq S^t$, else $x_i^{t+1} = \overline{x_i^t}$.

Input: the internal state x (32 bits)
Output: a state r of 32 bits

for $i = 0,\ldots,N$ **do**
  $d_i \leftarrow 0$
$a \leftarrow PRNG_1()$
$m \leftarrow g(a)$
while $i = 0,\ldots,m$ **do**
  $b \leftarrow PRNG2()\ mod$ N
  $S \leftarrow b$
  if $d_S = 0$ **then**
    $x_S \leftarrow \overline{x_S}$
    $d_S \leftarrow 1$
  else
    $m \leftarrow m + 1$
return $(r \leftarrow x)$

Dimention (CIPRNG-MCMD). Therefore, the CIPRNG-MCMD aims to replace the tempering function of the TGFSR PRNG (see Section 11) to pass statistical tests, when it does not without. This latter offers to us a well-uniform and multi-dimensional distribution as illustrated in Section 3.3 and using all techniques described earlier (seed and latency from Section 3.6 3.5).

5.2/ CIPRNG-XOR

Conversely to CIPRNG Multi-Cycles, this CIPRNG-XOR only needs one inputted generator. It operates on it using the vectorial negation. It was establish in [163] that G_f satisfies various properties of chaos with such iteration function, one of them being the notion of chaos according to Devaney. Another interesting property proven in the aforementioned article is that, if the inputted generator is cryptographically secure, then the resulted CIPRNG-XOR generator, obtained after post-processing, still present this property. Such CIPRNG-XOR, which is a sub-category of our CIPRNG post-treatment, can be summarized as follows [156]: x^0 is the initial Boolean vector of size N, and $(S^t)_{t\in\mathbb{N}}$ is the sequence generated by the inputted generator. We suppose that this latter produces numbers belonging into $[\![0, 2^{\mathrm{N}} - 1]\!]$, then operating G_f with the vectorial negation on such sequences can obviously be rewritten as Equation. (28) [156].

The main requirement is to prevent the machine from working in silos, by randomly picking a subset of the inputs at each iteration using another generator S^t. We recall here that the strategy S^t is expressed in the base-2 numeral system as a binary vector of size N, which takes at each iteration a new input from another generator (an entropy source like a physical white noise or some digits in the CPU temperature, can be considered for instance). By doing so, the finite state machine does not necessarily enter into a loop: a same state can be visited twice, but with two completely different future evolution, depending on the inputs.

Algorithm 2 presents details of this approach, which consists of deploying three different PRNGs to compute the strategy. In this version, two inputted PRNGs of 64 bits denoted by x_i and y_i are used for defining the chaotic strategy S. Furthermore, we added a third inputted set generator z_i of 32 bits for more complexity. The z_i generator will pick randomly a subset of the inputs at each iteration, as described in Equation 28, in which only the $\log(\log(n))$ least significant bits (in this case, 3 bits) are used.

Algorithm 2 CIPRNG-XOR proposal: it randomly picks a subset of the inputs at each iteration, whose index is contained in the first term of the strategy

Input: the internal state x (32 bits)
Output: a state r of 32 bits

1: $u_i \leftarrow PRNG1$,
2: $y_i \leftarrow PRNG2$,
3: $z_i \leftarrow PRNG3$
4: **if** $(z_i \& 1) \neq 0$ **then**
5: $\quad x \leftarrow x \otimes (u_i \& 0x0FFFFFFFF)$
6: **if** $(z_i \& 2) \neq 0$ **then**
7: $\quad x \leftarrow x \otimes (u_i \gg 32)$
8: **if** $(z_i \& 4) \neq 0$ **then**
9: $\quad x \leftarrow x \otimes (y_i \& 0x0FFFFFFFF)$
10: $r \leftarrow x \otimes (y_i \gg 32)$
11: **return** r

5.3/ FPGA IMPLEMENTATION

5.3.1/ GLOBAL COMPARISON

Let us recall the three CIPRNG described from previous sections, where each deploys a certain number of embedded PRNGs as strategies. Initially the strategies used for each CIPRNG type is as follows ({PRNG}, {pass or not for TestU01}): the XOR64 (NO), XOR$128+$ (NO), XOR64^* (PASS), KISS$64P124$ (NO), and is LFSR258 (NO) are candidate for 64 bits generators, where MRG$32k3a$ (PASS), MWC1038 (NO), TAUS88 (NO), LFSR113 (NO), MWC256 (NO), TT800 (NO), PCG32 (PASS), WELLRNG512 (NO), Mersenne Twister (NO), XOR128 (NO), CMWC4096 (NO), KISS32P124 (NO) are for 32 bits strategies.

Regarding CIs based post-processing, we tested more than 275 configurations of strategies for CIPRNG-XOR (three of 64 bits and one 32 bits) on our Mésocentre supercomputer facilities (170 were able to pass TestU01) and 169 of CIPRNG-MC/MCMD using two strategies of 32 bits (93 pass the TestU01). Only are recalled hereafter those who pass the recommended statistical TestU01 battery. To reach a fair comparison, we disabled the use of DSP blocs for linear PRNGs. Additionally, having the ASIC implementations in mind, we excluded each CIPRNG combination that deploys BRAM or DSP macros (MT, TT800), to be independent from the technology.

Results concerning CIPRNG-XOR and CIPRNG-MC (respectively CIPRNG-MCMD) are summarized in Table 5.2 and Table 5.1 (resp. in Table 5.3). The FPGA resources are calculated following the Section 4.3. In the former table, we specify which combination

has been studied. In examples contained in these tables, "A" is for XOR64, "B" means XOR128+, and "C" is LFSR258. Values 1,2,3, and 4 correspond to Taus88, LFSR113, XOR128, and XOR32 generators respectively. Additionally, as declared previously, we deploy the new AXI-test platform with Zynq or Atlys board (see Section 4.2 for further information).

Table 5.1: FPGA Implementation of CIPRNG-MC iteration post-processing using different linear PRNG as strategy

CIPRNG (32Bits)		CIPRNG Mult-Cycle [PRNG-32bits, Strategy-32Bits]								
PRNG		[1-2]	[1,3]	[2,1]	[2-3]	[3-1]	[4-1]	[1,1]	[2,2]	[3,3]
AREA	LUT	194	201	187	194	175	119	197	211	175
	FF	386	386	388	418	373	252	356	418	403
	DSP	0	0	0	0	0	0	0	0	0
	RAM	0	0	0	0	0	0	0	0	0
	Total Area (LUT+FF)*8	4640	4696	4600	4896	4384	2968	4424	5032	4744
SPEED	Frequency (Mhz)	304	322	327	307	312	326	300	330	288
	Design Latency	3→330	3→330	3→330	3→330	3→330	3→330	3→330	3→330	3→330
	Output Latency	3→330	3→330	3→330	3→330	3→330	3→330	3→330	3→330	3→330
	Throughput/Latency (Gbps)	3.2→0.03	3.4→0.03	3.5→0.03	3.3→0.03	3.3→0.03	3.478→0.03	3.196→0.03	3.523→0.03	3.069→0.03

a Values 1,2,3, and 4 correspond to Taus88, LFSR113, XOR128, and XOR32.

Table 5.2: FPGA Implementation of CIPRNG-XOR post-processing using different linear prng as strategy

CIPRNG (32Bits)		CIPRNG-XOR [PRNG-64bits, PRNG-64Bits, Strategy-32Bits]						
PRNG		[A,B,2]	[A,B,3]	[B,B,1]	[B,B,2]	[B,B,3]	[B,C,2]	[B,A,2]
AREA	LUT	364	357	237	222	226	502	345
	FF	582	586	454	424	458	838	582
	DSP	0	0	0	0	0	0	0
	RAM	0	0	0	0	0	0	0
	Total Area (LUT+FF)*8	7568	7544	5528	5168	5472	10720	7416
SPEED	Frequency (Mhz)	257.7	250	250.9	251.8	250	266	257.5
	Design Latency	3	3	3	3	3	3	3
	Output Latency	1	1	1	1	1	1	1
	Throughput/Latency (Gbps)	8.246	8.0	8.028	8.057	8.0	8.512	8.24

a "A" is for XOR64, 1. "B" means XOR128+, 2. "C" is LFSR258,
b Values 1,2, and 3 correspond to Taus88, LFSR113, XOR128.

5.3.2/ COMPARISON

CIPRNG Multi-Cycle: As recalled previously, this particular version of chaotic iterations post-treatment is based on two inputted PRNGs. For FPGA implementation, 7 CIPRNG combinations have been selected for their hardware performance. At each iteration, the CIPRNG-MC updates only the bit which is selected by the strategy. The throughput will be

then divided by the number of iterations (cycles) to update a subset of the input m^t. That is, the number of iterations is variable, which corresponds to a range of throughput/latency values. According to results presented in Table 5.1, throughput of CIPRNG Multi-Cycle is larger than those of almost all linear PRNGs that pass TestU01 (PCG, MRG32, and XOR64*). Additionally, the consumed area is globally small, even if 2 PRNGs are embedded and without inferring any blocks (DSPs and BRAM). Regarding statistical evaluation, all the selected combinations succeeded the TestU01, contrary to all other chaotic PRNGs based on Hénon [151], Lörenz & Chen [164], and Tent [165] maps.

CIPRNG-XOR: In this last version, 7 other combinations of CIPRNG-XOR generators have been selected for their hardware performance, when compared with linear PRNGs. The results of Table 5.2 illustrate a throughput to generate 32 bits 2.5 times larger for CIPRNG-XOR than for almost all linear PRNGs that can pass TestU01. Furthermore, if we consider the Thoughput/Latency ratio, CIPRNG is respectively 12 times, 30 times, and finally 7 times faster than XOR64^*, PCG32, and combined PNRGs (MRG32 and KISS124). Additionally, when DSPs blocks are disabled on use, CIPRNG-XOR is 25 times, 44 times, and finally 35 times faster than XOR64^* (0.34Gbps), PCG32 (0.2Gbps), and combined MRG32 (0.25Gbps). The same statement holds for area: CIPRNG-XOR deploys 3 PRNGs, but it is 5 times more efficient than any other linear PRNGs. Compared to all other aforementioned chaotic PRNGs, all configurations of CIPRNG-XOR are more efficient in throughput, area, and ability to face statistical tests. Therefore, for FPGA application, all combinations can contribute in hardware performance and statistical tests compared to linear PRNGs. Finally, compared to CIPRNG-MC, the CIPRNG-XOR is less compact in area resources, but largely more efficient in terms of throughput.

CIPRNG Multi-Cycle Multi-Dimension: As can be seen in Table 5.3, this new post-processing provides the same hardware performance as the original TGFSR PRNGs (see Table 3.2). Additionally, this new post-processing improves generators, in such a way that they are able to pass the statistical TestU01 battery, while providing improved performances with almost all chaotic PRNGs, as the ones of [151, 164, 165]. Due to such qualities, these new types of CIPRNGs can thus contribute to parallel processing and computation applications, like in Monte-Carlo simulation.

5.4/ ASIC IMPLEMENTATION

Table 5.4 and Table 5.5 summarizes the ASIC implementation, which uses two global flows: the synthesis flow using Cadence RTL Compiler, and physical place and route (P&R) flow in a second step, with Cadence Encounter Digital Implementation (see Section 4.4 for further information).

5.4.1/ ASIC COMPARISON

We recall here from ASIC section 4.4 the area, the timing, and the power analysis. The result analyzes of the various ASIC implementation of CIPRNG can be summarized as follows.

Area Analysis: It is obvious that CIPRNG-XOR needs twice the area of CIPRNG-MC, due to the use of three generators. For CIPRNG-MC, $[1, 3]$, $[2, 1]$, and $[4, 1]$ have the lowest area, which uses Taus88 (1) as a strategy. In the case of CIPRNG-XOR, $[A, B, 2]$, $[A, B, 3]$,

Table 5.3: FPGA implementation of Multi-Cycle Multi-Dimension chaotic iteration post-processing based for MT and TT800

	Mersenne Twister		TT800	
Strategy	Taus88	XOR128	Taus88	LFRS113
LUT	434	447	358	357
FF	676	672	830	853
DSP	3	3	6	6
RAM	2	2	2	2
Area (LUT+FF)*8	8880	8952	9504	9680
Design_Latency	3	3	3	3
Output_Latency	1	1	1	1
Throughput/Latency	4.8	4.8	5.2	5.3

Table 5.4: 65nm ASIC Implementation of CIPRNG-MC post-processing using different linear prng as strategy

CIPRNG (32Bits)		CIPRNG Multi Cycle [PRNG-32bits, Strategy-32Bits]								
PRNG		[1,2]	[1,3]	[2,1]	[2,3]	[3,1]	[4,1]	[1,1]	[2,2]	[3,3]
Area	Standard Cells Area μm^2	4718	4739	4791	5267	4437	3229	4372	5136	4831
	Gate Elements (GE μm^2)	3276	3291	3327	3658	3081	2242	3036	3567	3355
	Transistor Area (TE μm^2)	13104	13164	13308	14632	12324	8968	12144	14268	13420
Speed	Frequency (Mhz)	492	427	478	594	473	380	489	427	454
	Output Latency	3→330	3→330	3→330	3→330	3→330	3→330	3→330	3→330	3→330
	Throughput /Latency (Gbps)	5.2→0.03	4.6→0.04	5→0.05	6→0.6	5→0.05	4→0.04	5.2→0.05	4,5→0.04	4.8→0.04
Power	Internal Power (mW)	1.08	1.03	1.05	1.16	0.98	0.72	1	1.11	1.07
	Switching Power (mW)	0.37	0.37	0.39	0.5	0.34	0.25	0.36	0.41	0.38
	Total Power (mW)	1.46	1.4	1.45	1.66	1.32	0.97	1.36	1.52	1.45

a Values 1,2,3, and 4 correspond to Taus88, LFSR113, XOR128, and XOR32.

and $[B, A, 2]$ are selected as best candidates for the lowest area consumption in chaotic iterations based PRNGs.

Static Timing Analysis: Following Table 5.4 and Table 5.5, the CIPRNG-MC throughput is twice better than CIPRNG-XOR, and similarly for the area. However, due to the latency problem, this latter drops the throughput and balance CIPRNG-XOR up to 200 times the ones of CIPRNG-MC and other linear PRNGs who pass TestU01. Finally, combinations $[A, B, 3]$, $[B, A, 2]$, and $[A, B, 2]$ are candidates of chaotic iterations PRNGs who pass TestU01 and with good time performances.

Power Analysis: Various dynamic power analyzes illustrate in both tables low power consumption of both CIPRNG-MC and CIPRNG-XOR. It is clear from Table 5.4 and Table 5.5 that, when we propagate the clock (switching), the switching power of the CIPRNGs is lower than the internal power consumed by the internal cell of CIPRNGs. This is confirmed by the area of both CIPRNG family (GE, and especially FF). Despite such results, CIPRNG-XOR consumes twice the power of CIPRNG-MC, which is balanced by

Table 5.5: 65nm ASIC Implementation of CIPRNG-XOR post-processing using different linear prng as strategy

CIPRNG (32Bits)		CIPRNG-XOR [PRNG-64bits, PRNG-64Bits, Strategy-32Bits]						
PRNG		[A,B,2]	[A,B,3]	[B,B,1]	[B,C,2]	[B,A,2]	[B,B,3]	[B,B,2]
Area	Standard Cells Area μm^2	9070	9165	11240	12579	9104	11867	11168
	Gate Elements (GE μm^2)	6299	6465	7806	8735	6322	8240	7756
	Transistor Area (TE μm^2)	25196	25860	31224	34940	25288	32960	31024
Speed	Frequency (Mhz)	276	340	273	275	281	248	270
	Output Latency	1	1	1	1	1	1	1
	Throughput /Latency (Gbps)	8.8	10.9	8.7	8.8	9	7.9	8.6
Power	Internal Power (mW)	1.72	1.9	2.023	2.45	1.74	2.27	1.83
	Switching Power (mW)	0.83	0.94	0.98	1.2	0.86	1.15	0.93
	Total Power (mW)	2.56	2.73	3.02	3.67	2.61	3.44	2.77

a "A" is for XOR64, 1. "B" means XOR128+, 2. "C" is LFSR258,
b Values 1,2, and 3 correspond to Taus88, LFSR113, XOR128.

frequency and throughput. We can finally select the combinations $[B, A, 2]$, $[A, B, 2]$, and $[B, B, 2]$ as candidates of chaotic iterations based PRNGs who can pass TestU01 and with power performances.

5.5/ STATISTICAL TESTS RESULTS

The statistical batteries of tests considered in this thesis have been evoked in Section 1.6.

During experiments, the test batteries are run in Z-book Intel Core $i7 - 4800MQCPU@2.70GHz \times 8$, working with Ubuntu 16.4 (64bits) and GCC $5.4.0$. For NIST, 100 sequences of 10^6 bits are generated and tested. Tables 5.6 and 5.7 confirm that all the chaotic iterations post-processings for linear PRNGs can pass the NIST, where the minimum passing rate for each statistical test is approximately 96 for a sample size of 100 binary sequences. In the TestU01 case, all CIPRNG configurations for both proposals (MC and XOR) can successfully pass this battery, which is failed when considering the other chaotic PRNGs evoked in this thesis.

5.6/ CONCLUSION

This chapter has introduced the implementation on FPGA and ASIC of the first family of post-processing for PRNGs based on chaotic iteration. The first conclusion that can be outlined is that chaotic iterations post-processing provides an alternative implementation of combined PRNGs without any supplemental cost, which is 2.5 times faster and 5 times more efficient (with a low latency) than almost all the linear PRNGs that can pass TestU01. However, it seems that CIPRNG-MC (multi-cycle) may become more efficient in term of hardware level by reducing the number of cycles.

Table 5.6: Statistical test of NIST for different FPGA implementations of CIPRNG-XOR: a 100 sequences of 10^6 bits are generated and tested and p-value > 0.0001 being required to pass a test [p-value + (minimum pass rate/100)]

Test	[A,B,2]	[A,B,3]	[B,B,1]	[B,B,2]	[B,B,3]	[B,C,2]	[B,A,2]
Frequency (Monobit)	0.366 (0.99)	0.171 (0.99)	0.145 (1.0)	0.911 (1.0)	0.637 (0.96)	0.739 (0.99)	0.779 (0.99)
Frequency within a Block	0.554 (0.99)	0.181 (0.99)	0.987 (0.99)	0.759 (0.99)	0.181 (0.99)	0.719 (1.0)	0.236 (0.99)
Cumulative Sums (Cusum) *	0.381 (0.99)	0.607 (0.99)	0.376 (0.995)	0.312 (1.0)	0.444 (0.965)	0.512 (0.975)	0.626 (0.985)
Runs	0.554 (0.97)	0.637 (0.99)	0.213 (0.97)	0.637 (1.0)	0.262 (0.99)	0.275 (1.0)	0.574 (1.0)
Longest Run of Ones in a Block	0.971 (0.98)	0.249 (0.98)	0.191 (0.97)	0.474 (1.0)	0.637 (1.0)	0.834 (0.99)	0.437 (1.0)
Binary Matrix Rank	0.012 (0.97)	0.162 (1.0)	0.249 (0.98)	0.574 (0.99)	0.798 (0.99)	0.455 (0.98)	0.319 (0.99)
Discrete Fourier Transform (Spectral)	0.883 (1.0)	0.595 (1.0)	0.455 (0.98)	0.319 (0.99)	0.834 (0.97)	0.455 (0.98)	0.275 (1.0)
Non-overlapping Template Matching*	0.509 (0.990)	0.472 (0.989)	0.494 (0.990)	0.477 (0.990)	0.482 (0.99)	0.490 (0.990)	0.498 (0.989)
Overlapping Template Matching	0.401 (1.0)	0.983 (1.0)	0.911 (0.99)	0.275 (0.99)	0.946 (0.99)	0.657 (0.99)	0.816 (1.0)
Maurer's "Universal Statistical"	0.474 (1.0)	0.055 (0.99)	0.983 (0.99)	0.637 (0.99)	0.075 (1.0)	0.494 (0.99)	0.678 (1.0)
Approximate Entropy (m=10)	0.616 (0.99)	0.066 (0.99)	0.834 (1.0)	0.249 (0.99)	0.554 (0.99)	0.494 (1.0)	0.437 (0.99)
Random Excursions *	0.514 (0.979)	0.439 (0.990)	0.522 (0.991)	0.559 (0.997)	0.490 (0.993)	0.436 (0.988)	0.562 (0.994)
Random Excursions Variant *	0.457 (0.967)	0.389 (0.980)	0.269 (0.996)	0.525 (0.988)	0.519 (0.997)	0.335 (0.989)	0.455 (0.985)
Serial* (m=10)	0.352 (1.0)	0.718 (0.99)	0.616 (0.99)	0.352 (0.985)	0.531 (0.995)	0.319 (1.0)	0.441 (0.98)
Linear Complexity	0.897 (0.99)	0.514 (1.0)	0.289 (0.99)	0.779 (0.98)	0.014 (0.99)	0.616 (1.0)	0.616 (1.0)
Total Tests	15/15	15/15	15/15	15/15	15/15	15/15	15/15

a The minimum pass rate for each statistical test is approximately = 96 for a sample size of 100 binary sequences.
b "A" is for XOR64, 1. "B" means XOR128+, 2. "C" is LFSR258, 3. Values 1,2,3, and 4 correspond to Taus88, LFSR113, XOR128, and XOR32.

Table 5.7: Statistical test of NIST for different FPGA implementation of CIPRNG-MC: a 100 sequences of 10^6 bits are generated and tested and p-value > 0.0001 being required to pass a test [p-value + (minimum pass rate/100)]

Test	[1-2]	[1,3]	[2,1]	[2-3]	[3-1]	[4-1]	[1,1]	[2,2]	[3,3]
Frequency (Monobit)	0.236 (0.99)	0.236 (0.99)	0.534 (0.99)	0.055 (1.0)	0.383 (1.0)	0.016 (0.99)	0.719 (0.99)	0.935 (0.98)	0.419 (0.97)
Frequency within a Block	0.129 (1.0)	0.129 (1.0)	0.319 (0.99)	0.834 (1.0)	0.595 (0.98)	0.383 (0.98)	0.851 (1.0)	0.883 (0.99)	0.935 (1.0)
Cumulative Sums (Cusum) *	0.696 (0.995)	0.696 (0.995)	0.326 (0.99)	0.687 (1.0)	0.785 (1.0)	0.438 (0.995)	0.345 (0.99)	0.478 (0.985)	0.895 (0.98)
Runs	0.514 (1.0)	0.514 (1.0)	0.739 (0.99)	0.494 (1.0)	0.946 (1.0)	0.964 (0.99)	0.137 (0.99)	0.699 (0.98)	0.275 (0.99)
Longest Run of Ones in a Block	0.162 (0.99)	0.162 (0.99)	0.213 (0.98)	0.955 (1.0)	0.366 (1.0)	0.719 (1.0)	0.494 (1.0)	0.366 (0.96)	0.798 (0.99)
Binary Matrix Rank	0.021 (0.98)	0.021 (0.98)	0.739 (0.99)	0.867 (0.98)	0.657 (0.99)	0.719 (0.99)	0.019 (1.0)	0.366 (1.0)	0.289 (0.98)
Discrete Fourier Transform (Spectral)	0.075 (0.98)	0.075 (0.98)	0.162 (1.0)	0.554 (1.0)	0.616 (0.98)	0.062 (0.98)	0.262 (1.0)	0.437 (0.96)	0.191 (0.98)
Non-overlapping Template Matching*	0.512 (0.990)	0.512 (0.990)	0.524 (0.990)	0.519 (0.991)	0.510 (0.991)	0.460 (0.992)	0.500 (0.990)	0.524 (0.989)	0.519 (0.991)
Overlapping Template Matching	0.213 (1.0)	0.213 (1.0)	0.834 (1.0)	0.978 (1.0)	0.019 (1.0)	0.883 (0.97)	0.066 (1.0)	0.455 (0.99)	0.494 (1.0)
Maurer's "Universal Statistical"	0.474 (0.98)	0.474 (0.98)	0.657 (1.0)	0.595 (0.98)	0.304 (0.99)	0.437 (0.99)	0.350 (0.99)	0.834 (0.99)	0.616 (1.0)
Approximate Entropy (m=10)	0.289 (1.0)	0.289 (1.0)	0.657 (1.0)	0.946 (0.98)	0.401 (0.99)	0.798 (1.0)	0.534 (0.97)	0.494 (0.98)	0.455 (1.0)
Random Excursions *	0.296 (0.988)	0.296 (0.988)	0.385 (0.991)	0.289 (0.978)	0.562 (0.995)	0.471 (0.984)	0.351 (0.988)	0.395 (0.985)	0.310 (0.993)
Random Excursions Variant *	0.287 (0.993)	0.287 (0.993)	0.327 (0.996)	0.385 (0.983)	0.650 (0.991)	0.514 (0.986)	0.390 (0.980)	0.235 (0.993)	0.418 (0.996)
Serial* (m=10)	0.054 (0.995)	0.054 (0.995)	0.530 (0.995)	0.504 (1.0)	0.118 (0.99)	0.575 (0.985)	0.351 (0.99)	0.427 (0.985)	0.796 (0.965)
Linear Complexity	0.834 (0.99)	0.834 (0.99)	0.474 (1.0)	0.946 (0.99)	0.574 (1.0)	0.851 (1.0)	0.455 (0.98)	0.071 (0.97)	0.739 (1.0)
Total Tests	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15

a The minimum pass rate for each statistical test is approximately = 96 for a sample size of 100 binary sequences.
b Values 1,2,3, and 4 correspond to Taus88, LFSR113, XOR128, and XOR32.

V

Generalized Chaotic Iteration PRNG

6

GENERALIZED CHAOTIC ITERATION

This chapter investigates the hardware point of view of PRNG implementation. Contributions can be summarized as follows.

In order to reduce the number of embedded PRNGs used as strategies in chaotic iterations proposal, a new pseudorandom number generator (GCIPRNG), specifically designed for FPGA, is proposed. At each iteration, it receives a new input from another given generator (P)RNG, which is called the strategy. However, this generator includes the two modes of iterations unary and parallel (CIPRNG) already detailed in the previous Chapter 5. Therefore, we say that PRNG based on generalized chaotic iterations, is a generalization of the two iterations other modes. Thanks to an embedded Boolean function and a mixing function, our generator, which can be considered as a post-treatment on the entrance one, has usually a better statistical profile than its input, while running at a similar speed. Finally, we mathematically prove that the produced output has a chaotic dependence to the input. The mid-term effects of a slight modification of the seed or of the strategy cannot be predicted. The idea has been fully deployed on FPGA and it runs completely in parallel while consuming as few resources as possible.

6.1/ GENERAL IDEA

Formally speaking, a Chaotic Iteration based PRNG (CIPRNG) is a random walk in the graph of iterations of a specific binary function. The direction to take and the path length are defined by the embedded generator(s) [163]. A first application of such an approach was presented in the PRNG framework [12, 14]. Meanwhile, in [166], the authors have proposed to remove an Hamilton Cycle, satisfying some balance properties, from the Markov chain on the N-cube, while in [16], authors proposed new functions without an Hamilton cycle, and studied the length of the walk in their cube, until having an associated Markov graph close enough to the uniform distribution. In these first studies, the minimum length of the chain between two uniform outputs is larger than 109 in [166] and is equal to 9 in [16] for a Boolean function of 8 binary variables. These works end with the idea that it is hard to have together the three properties of: chaos, hardware efficiency, and statistically trustworthy.

Let us first discuss on how we tackle this problem. The first key idea is to have a short internal state, possibly split into parallel blocs. This divide and conquer approach aims at ensuring hardware efficiency but is in conflict with statistical quality. Chaotic iterations [10, 11] can be used to achieve chaos objectives. However, as noticed in [16] the general

formulation of the chaotic iterations [156] should be preferred than the original one when efficiency is needed. Finally, permutation techniques [167] have presented a convenient way to ensure statistical faultless, in a fast manner. Our proposal is based on these three main ideas and is summarized in Figure 6.1.

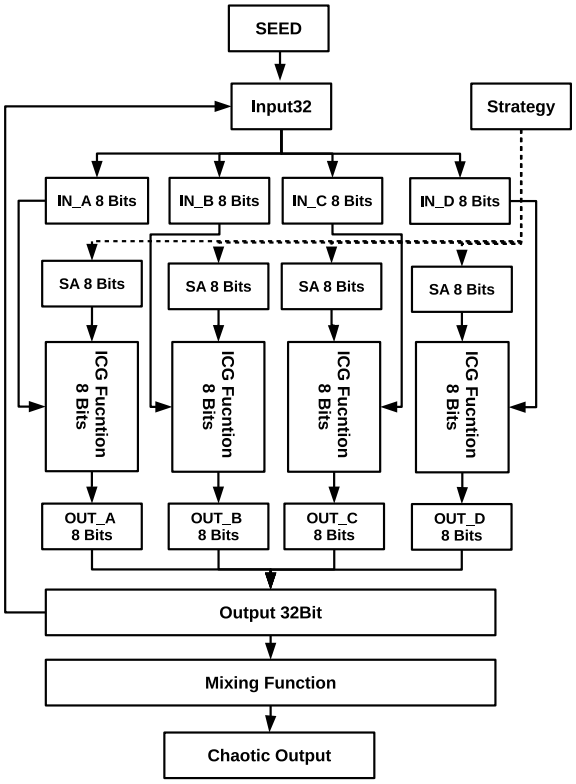


Figure 6.1: The proposal

At first, it can be seen that the seed x^0, the internal state x^t, and the output x^{t+1} are all expressed with the same number N of bits. Without loss of generality, we consider hereafter that N $= 32$. Let us show how to produce a new output x^{t+1} for a given input x^t. This one is first split into n blocs of equal length. We consider here that $n = 4$ and we thus have $x^t = (x^t_A, x^t_B, x^t_C, x^t_D)$ where x^t_l is of size 8 for $l \in \{A, B, C, D\}$. The next step consists in obtaining a N bits number s^t from the embedded generator, which is called *the strategy*. Similarly to x^t, the vector s^t is split into n blocs. Here we thus obtain $s^t = (s^t_A, s^t_B, s^t_C, s^t_D)$. Each s_l, $l \in \{A, B, C, D\}$, can be interpreted as a set of elements in $\{1, 2, \ldots, 8\}$. Each block x^t_l is modified separately as the result of the general formulation of the Chaotic Iterations [156] applied on x^t_l, s^t_l and a specific GCI function $f : \mathbb{B}^8 \to \mathbb{B}^8$, as described hereafter. The i-th component of x^{t+1}_l is the i-th one of $f(x^t_l)$ if i is within the set s^t_l, else this component is the i-th one of x^t_l (*i.e.*, only the components indicated by the set s^t_l are updated). This results x^{t+1}_l. All the x^{t+1}_l are concatenated hereafter, producing the new internal state x^{t+1}. Finally, a permutation over the N bits is applied on x^{t+1} to produce the new output.

The choice of the function f executed inside the GCI iteration, of the embedded PRNG, and of the chosen final permutation function has a great influence on the quality of the generator. It is discussed in the next sections.

6.1.1/ ITERATED FUNCTION

Let $s \in (\mathbb{B}^8)^{\mathbb{N}}$ be a sequence of subests of $\{ 1, \ldots 8\}$, x^0 be a vector in $\mathbb{B}^8$, and f be a function from $\mathbb{B}^8$ to $\mathbb{B}^8$. The sequence $(x^t)^{t\in\mathbb{N}}$ of vectors in $\mathbb{B}^8$ defined according the general formulation of the Chaotic Iterations [156] is

$$x^{t+1} = (x_1^{t+1}, \ldots, x_{\mathsf{N}}^{t+1}) \text{ where } x_i^{t+1} = \begin{cases} f_i(x^t) \text{ if } i \in s^t, \\ x_i^t \text{ otherwise.} \end{cases}$$

Five functions from $\mathbb{B}^8$ to $\mathbb{B}^8$ are mainly studied in this article. The former is the negation function, further denoted as NG. In this one, each f_i is defined with $f_i(x) = \overline{x_i}$. For instance, the image of 5 = 00000101 is 250 = 11111010. We denoted as F1, F2, F3, and F4, which are the GCI functions whose graph of generalized iterations is strongly connected and which has been obtained by removing a balanced Hamiltonian cycle in a N-cube following the method suggested in [16]. The choice of these five functions is motivated by the objective to obtain a chaotic behavior.

Table 6.1: Boolean functions

Function	$f(x)\ for\ x \in [0, 1, 2, 3, 4, 5 \ldots, 2^n - 1]$
NEG	[255, 254, 253, 252, 251, 250 . . . , 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
F1	[223, 190, 249, 236, 243, 234, 241, 252, 183, 244, 229, 245, 179, 178, 225, 248, 237, 254, 173, 232, 171, 202, 201, 200, 247, 198, 228, 230, 195, 242, 233, 160, 215, 220, 205, 216, 218, 154, 221, 208, 213, 210, 212, 148, 147, 211, 217, 209, 239, 238, 141, 140, 235, 203, 193, 204, 135, 134, 199, 197, 131, 226, 129, 224, 63, 174, 253, 184, 251, 250, 189, 176, 191, 246, 180, 182, 51, 50, 185, 240, 47, 46, 175, 188, 139, 42, 161, 172, 231, 164, 181, 165, 227, 130, 33, 32, 31, 222, 153, 158, 219, 26, 25, 156, 159, 214, 151, 149, 146, 18, 144, 152, 207, 206, 157, 136, 138, 170, 169, 8, 133, 6, 5, 196, 3, 194, 137, 192, 255, 110, 109, 120, 107, 126, 125, 112, 103, 114, 116, 118, 123, 98, 121, 96, 79, 78, 111, 124, 75, 122, 97, 108, 71, 100, 117, 101, 115, 66, 113, 64, 127, 90, 89, 94, 83, 91, 81, 92, 95, 84, 87, 85, 82, 86, 80, 88, 77, 76, 93, 72, 74, 106, 105, 104, 69, 102, 68, 70, 99, 67, 73, 65, 55, 58, 57, 44, 187, 186, 49, 60, 119, 52, 37, 53, 35, 54, 177, 56, 45, 62, 61, 40, 59, 10, 9, 168, 167, 166, 36, 38, 163, 162, 41, 48, 23, 28, 13, 24, 155, 30, 29, 16, 21, 150, 20, 22, 27, 19, 145, 17, 143, 142, 15, 14, 43, 11, 1, 12, 39, 4, 7, 132, 2, 34, 0, 128]
F2	[223, 190, 249, 254, 243, 186, 233, 252, 183, 182, 247, 228, 242, 226, 240, 224, 237, 206, 173, 232, 203, 250, 169, 248, 167, 246, 245, 164, 235, 227, 241, 192, 215, 158, 157, 216, 218, 222, 221, 152, 213, 210, 149, 214, 219, 211, 217, 209, 239, 202, 207, 236, 139, 138, 193, 136, 231, 230, 199, 197, 194, 130, 225, 200, 63, 188, 253, 184, 251, 58, 189, 56, 191, 54, 165, 244, 51, 179, 161, 177, 47, 238, 175, 140, 163, 234, 41, 172, 39, 134, 229, 36, 162, 178, 129, 176, 31, 154, 29, 220, 147, 26, 145, 24, 159, 148, 151, 212, 146, 150, 144, 208, 141, 14, 205, 204, 171, 142, 201, 128, 133, 198, 132, 196, 195, 2, 137, 0, 255, 124, 109, 120, 122, 106, 125, 104, 117, 102, 101, 118, 123, 115, 97, 113, 79, 126, 111, 76, 99, 74, 121, 108, 71, 70, 103, 116, 98, 114, 65, 112, 127, 90, 89, 94, 83, 91, 81, 92, 95, 84, 87, 85, 82, 86, 80, 88, 77, 110, 93, 72, 107, 78, 105, 64, 69, 66, 68, 100, 75, 67, 73, 96, 55, 46, 57, 62, 187, 59, 185, 60, 119, 52, 181, 180, 50, 34, 48, 32, 45, 174, 61, 40, 11, 170, 9, 168, 37, 166, 53, 4, 43, 35, 49, 160, 23, 28, 13, 156, 155, 30, 153, 16, 21, 18, 20, 22, 27, 19, 25, 17, 143, 10, 15, 44, 3, 42, 1, 12, 135, 38, 7, 5, 131, 6, 33, 8]
F3	[223, 238, 189, 254, 243, 251, 233, 184, 183, 230, 229, 245, 242, 246, 177, 224, 237, 174, 253, 204, 203, 170, 201, 248, 247, 226, 197, 164, 235, 227, 241, 192, 215, 158, 205, 216, 155, 222, 221, 208, 151, 210, 212, 214, 219, 211, 145, 209, 143, 202, 207, 206, 139, 234, 193, 232, 135, 134, 199, 228, 194, 198, 129, 200, 63, 188, 61, 252, 186, 250, 249, 168, 191, 178, 180, 244, 187, 179, 49, 240, 239, 46, 175, 236, 163, 138, 185, 136, 231, 38, 181, 36, 162, 166, 225, 176, 31, 30, 153, 220, 147, 218, 217, 24, 159, 148, 213, 149, 19, 150, 144, 152, 141, 140, 13, 12, 171, 142, 9, 8, 133, 130, 5, 196, 195, 2, 137, 160, 255, 124, 109, 120, 122, 106, 125, 104, 103, 114, 116, 100, 123, 115, 97, 113, 79, 126, 111, 110, 99, 74, 121, 72, 71, 118, 117, 68, 98, 102, 65, 112, 127, 90, 89, 94, 83, 91, 81, 92, 95, 84, 87, 85, 82, 86, 80, 88, 77, 76, 93, 108, 107, 78, 105, 64, 69, 66, 101, 70, 75, 67, 73, 96, 55, 190, 57, 62, 51, 59, 41, 60, 119, 182, 37, 53, 50, 54, 48, 32, 45, 44, 173, 172, 11, 58, 169, 56, 167, 34, 165, 52, 43, 35, 161, 0, 23, 28, 157, 156, 26, 154, 29, 16, 21, 18, 20, 22, 27, 146, 25, 17, 47, 10, 15, 14, 3, 42, 1, 40, 39, 4, 7, 132, 131, 6, 33, 128]
F4	[223, 250, 249, 254, 243, 234, 185, 232, 183, 244, 229, 180, 242, 178, 240, 248, 237, 206, 253, 252, 171, 170, 201, 224, 247, 246, 165, 230, 195, 227, 161, 192, 215, 220, 205, 216, 218, 222, 153, 208, 151, 150, 212, 214, 219, 211, 217, 209, 239, 202, 207, 236, 235, 138, 137, 204, 135, 196, 199, 228, 194, 130, 225, 128, 63, 188, 61, 172, 251, 190, 189, 176, 191, 166, 245, 182, 187, 50, 241, 177, 143, 238, 175, 140, 43, 42, 233, 184, 231, 164, 37, 132, 35, 226, 33, 168, 31, 154, 221, 158, 27, 155, 145, 156, 159, 22, 213, 149, 146, 210, 144, 152, 141, 14, 157, 136, 203, 142, 9, 200, 7, 198, 197, 4, 163, 131, 193, 0, 255, 124, 109, 108, 107, 126, 125, 112, 103, 102, 116, 118, 123, 115, 97, 113, 79, 106, 111, 76, 75, 122, 121, 120, 71, 100, 117, 68, 98, 114, 65, 104, 127, 90, 89, 94, 83, 91, 81, 92, 95, 84, 87, 85, 82, 86, 80, 88, 77, 110, 93, 72, 74, 78, 105, 64, 69, 66, 101, 70, 99, 67, 73, 96, 55, 58, 57, 62, 51, 186, 41, 40, 119, 52, 181, 53, 179, 34, 48, 56, 45, 174, 173, 60, 59, 46, 169, 32, 167, 54, 5, 38, 3, 162, 49, 160, 23, 28, 13, 24, 26, 30, 29, 16, 21, 18, 20, 148, 147, 19, 25, 17, 47, 10, 15, 44, 139, 11, 1, 12, 39, 134, 133, 36, 2, 6, 129, 8]

6.2/ MIXING FUNCTION

First of all, our proposal is a parallel execution of 4 blocks, each one producing 8 bits. The internal state x is next produced as the concatenation of the results of the 4 blocks. This design is guided by the goal of reducing the required resources. However, such an approach suffers from decreasing the statistical complexity of the PRNG: without any post

treatment it would be dramatic, because it has to deal with 8 bits only. A final step which scrambles the internal state is thus necessary to tackle this problem.

This can be practically implemented with a mixing function (which allows to obtain a uniform output) provided it does not break the chaos property (as proven in the next section). Among the large choice of mixing functions (such as rotation, dropping, xoring...), we inspire from a detailed work in [167]. This work indeed proposes a bench of mixing functions allowing to succeed statistical tests.

This mixing function is implemented as in Algorithms 3. It is not hard to see that it is mainly a composition of three subfunctions. Let *In32* be the internal state. The first function scrambles between 17 and 28 rightmost bits (*i.e.* middle bits) with a xor function. The number of selected elements depends on the value of $In32$. Then, the second function applies a modular multiplication in the cyclic group of elements in $\{1,\ldots,2^{31}-2\}$. The chosen multiplier b is a primitive root of the modulus $2^{31}-1$. However, in [167] they need more than 36 bits of internal state to pass TestU01, which is equivalent to a modulus of $2^{37}-25$. Therefore, b is set to "277803737", but any primitive root of $2^{37}-25$ is convenient for their work in [167]. The mixing function is a simple right xorshift on the lowest bits to scramble them.

Finally, the same concept can be applied for any data width of CIPRNG based generalized scheme associated to mixing function. Where, in 64 bits version of the generator, we deploy a 64 bits strategy divided with the internal space x^t into 8 blocs as for 32bits case, then each of these blocs will pass through 8 CGI functions (same or combination). Algorithms 4 illustrates the 64 bits mixing function used for the 64 bits CIPRNG based generalized scheme.

Algorithm 3 A 32 bits Random Xorshift mixing Function

Input: the internal state $In32$ word (32 bits)
Output: a state $Out32$ of 32 bits

1: $word1 \leftarrow (In32 \gg ((In32 \gg 28u) + 4u)) \otimes In32$
2: $word2 \leftarrow word1 * b$
3: $word3 \leftarrow (word2 \gg 22u) \otimes word2$
4: **return** $Out32 \leftarrow word3$

Algorithm 4 A 64 bits Random Xorshift Permutation Function

Input: the internal state $In64$ word (64 bits)
Output: a state $Out32$ of 64 bits

1: $word1 \leftarrow (In64 \gg ((In64 \gg 59u) + 5u)) \otimes In64$
2: $word2 \leftarrow word1 * b$
3: $word3 \leftarrow (word2 \gg 43u) \otimes word2$
4: **return** $Out64 \leftarrow word3$

The next section shows that such a proposal provides a chaotic PRNG.

6.3/ CHAOTIC BEHAVIOR OF OUR GENERATOR

Let us recall or specify first some notations and definitions in use in this section. In what follows, $\mathbb{B}$ is the Boolean set, while $\mathbb{N}$ is the usual sets of integer numbers. For $a, b \in \mathbb{N}$,

$[\![a, b]\!]$ is the set of integers: $\{a, a+1, \ldots, b\}$, $X^{\mathbb{N}}$ is the set of sequences belonging in X and s^k is the k-th term of a sequence $s = \left(s^k\right)_{k\in\mathbb{N}}$, which may be a vector (thus explaining the use of an exponent). Finally, f^n means the n-th composition of the function f (*i.e.*, $f^n = f \circ f \circ \ldots \circ f$).

In the proposal, the internal function h_f is iterated on the current internal state, and with a new generated sequence taken from the outer strategy. Then, the output is a permutation p of the internal state, which is not internally modified. The topological framework proposed in [12] for the CIPRNG-XOR and in [16] can be applied, *mutatis mutandis*, to this generator. It is then possible to state that iterations of the internal function are chaotic on its iteration space, denoted as $\mathcal{X}_{32} = \mathbb{B}^{32} \times [\![0, 31]\!]^{\mathbb{N}}$. And, using a topological semi-conjugacy, that the permutation does not alter such an unpredictable behavior. After having established that the 8-bits ICG function, denoted as g_f, is strongly transitive on its iteration space $\mathcal{X}_8$, we can first deduce that the discrete dynamical system $x^0 \in \mathcal{X}_8$, $x^{n+1} = g_f(x^n)$ is chaotic, and then that h_f is chaotic according to Devaney.

The proof of the whole generator G_f is chaotic can be found in Annex A.2.

Finally, the whole generator with the permutation p must be integrated inside the iterations, to see if the output has a chaotic behavior when modifying the input (internal state or strategy). To write the generator as a discrete dynamical system, we need to introduce the reverse permutation p^{-1}. To do so, let us define

$$\begin{array}{rccc} \mathsf{p}: & \mathcal{X}_{32} & \longrightarrow & \mathcal{X}_{32} \\ & (e, s) & \longmapsto & (\mathsf{p}(e), s), \end{array}$$

its inverse being

$$\begin{array}{rccc} \mathsf{p}^{-1}: & \mathcal{X}_{32} & \longrightarrow & \mathcal{X}_{32} \\ & (e, s) & \longmapsto & (\mathsf{p}^{-1}(e), s). \end{array}$$

We can now introduce the following diagram:

$$\begin{array}{ccc} \mathcal{X}_{32} & \xrightarrow{h_f} & \mathcal{X}_{32} \\ \uparrow{\scriptstyle \mathsf{p}^{-1}} & & \mathsf{p}\downarrow \\ \mathcal{X}_{32} & \xrightarrow{G_f} & \mathcal{X}_{32} \end{array}$$

p^{-1} and p are obviously continuous on $(\mathcal{X}_{32}, d_{32})$, which can be directly deduced by the sequential characterization of the continuity. So the commutative diagram depicted above is a topological conjugacy, and the generator

$$G_f = \mathsf{p} \circ h_f \circ \mathsf{p}^{-1}$$

thus inherits the chaotic behavior of h_f on $(\mathcal{X}_{32}, d_{32})$.

6.4/ FPGA Implementation

Table 6.2 and Table 6.3 present the results of two different implementations of our proposal on FPGA with their TestU01 statistical test evaluations. During these implementations, we considered five distinct Boolean functions, namely the negation and GCI F1, F2 as mentioned in Section 6.2 (F3 and F4 have the same behaviors than F1 and F2).

To pass TestU01, the multiplication constant "b" of the permutation function (see Algorithm (3)) must be great or equal to 811 for all strategies based 32 bits generators and 995 for 64 bits (we note here that there are other small constants that allow to pass TestU01 but with not all strategies). Therefore, in term of FPGA, these small constants represent a compact implementation resource compared to the original constant initially proposed (*i.e.*, 277803737 for PCG32). Linear PRNGs are used as strategies (inputted generators), which are LFSR113, Taus88, xorshift128, and LFSR258. All the design is synchronized with a main clock of $125Mhz$ and reset. Obtained results are described hereafter.

Negation Function. Three implementations have been realized and evaluated for each GCIPRNG width (32/64 bits). Notice that, in these 3 evaluations, the value of the minimum modular multiplication operand b used in the mixing function (see Section 6.2) is the same. We have obtained that the negation has similar performances compared to other GCI functions in terms of throughput and area.

For 32 bits generators, it is obviously more efficient than its best competitors (linear or chaotic PRNGs), as its throughput is between 1 and 6 times larger than the other chaotic PRNGs (that cannot pass TestU01). However, the exception comes from [92] using the logistic map and Berouili [165]: it is true that the latter has a throughput of 7.5 and 8.5 Gbps for 32 bits (we discarded [92], as this latter is fully dependent on Matlab Simulink macros, which is not relevant for ASIC implementation). Similarly, our three implementations using the negation function exhibit lower results compared to XOR-CIPRNG [12] for throughput compared to the area.

Finally, compared to the 32 bits PRNGs that can also pass TestU01, our 64 bits generators with the negation function use less area and are faster. To conclude this part, and when considering the negation, our proposal using LFSR113 as strategy is our best candidate for 32 bits generating 6.96 Gbps, which is increased to 11.5 Gbps for 64 bits generators.

GCI functions. We performed similar experiments than for the negation function. We obtained a lower performance in terms of throughout when compared with the negation function, which is due to the function implementation, and because in the negation we iterate a very simple logical operation (see Algorithm 3 to compare). However, despite its use of a bigger constant, which leads to a longer data path, the proposal with GCI functions does not consume any DSP block of FPGA: logic operators are sufficient. Additionally, results show that the three implementations with GCI functions perform better than all the other chaotic PRNGs that can pass the TestU01, if we except both our proposal with the negation and the XOR-CIPRNG [12]. Their performances are close to what has been obtained with the negation function, or to [12] with Taus88 as strategy, while GCIPRNG makes harder to reverse the process without knowing the internal transition function. However, XOR-CIPRNG [12] is limited to 32 bits ($8.5Gbps$) and uses three different strategies (two 64 bits and one 32 bits), when GCIPRNG is up to 64 bits ($\approx 12Gbps$) while less strategies are used (for 64-bit GCIPRNG version, it uses a 64-bits strategy based on two 32 bits generators or a linear PRNG of 64 bits). Indeed, GCIPRNG is more efficient in term of area, throughput, and integrity for embedded cryptographical applications and system with a flexibility of integration in SoC for different sizes.

Table 6.2: FPGA Implementation of 32-bits GCI PRNG using different linear PRNG as strategy

	PRNG	Negation			F1			F2			F3			F4		
AREA	Strategy	Taus88	LFSR113	xorshift128	Taus88	LFSR113	xorshift128	Taus88	LFSR113	xorshift128	Taus88	LFSR113	xorshift128	Taus88	LFSR113	xorshift128
	LUT	263	273	252	421	433	395	421	415	399	403	394	411	421	418	401
	FF	305	344	344	412	518	444	412	444	444	412	444	444	412	462	444
	RAM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	DSP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Total Area (LUT+FF)*8	4544	4936	4936	6664	7608	6712	6664	6872	6744	6520	6704	6840	6664	7040	6760
SPEED	Frequency (ns)	2.983	3.398	3.191	3.017	2.932	2.815	2.87	3.14	2.85	3.188	3.22	2.969	2.967	2.72	2.864
	Frequency (Mhz)	199.3	217.3	209.94	200.68	197.32	192.86	194.93	205.76	194.17	207.81	209.2	198.77	198.69	198.39	194.7
	Design Latency	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	Output Latency	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	Throughput/Latency (*Gbps*)	6.38	6.95	6.65	6.42	6.31	6.17	6.24	6.58	6.21	6.65	6.69	6.36	6.35	6.06	6.23
TEST	NIST	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
	TestU01	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS

Table 6.3: FPGA Implementation of 64-bits GCI PRNG using different linear PRNG as strategy

	PRNG	Negation			F1			F2			F3			F4		
AREA	Strategy	Taus88 LFSR113	LFSR258	xorshift128 taus88	Taus88 LFSR113	LFSR258	xorshift128 taus88	Taus88 LFSR113	LFSR258	xorshift128 taus88	Taus88 LFSR113	LFSR258	xorshift128 taus88	Taus88 LFSR113	LFSR258	xorshift128 taus88
	LUT	625	614	665	869	856	906	877	864	914	873	857	906	878	864	914
	FF	812	812	804	779	812	850	812	812	850	812	812	850	812	812	850
	RAM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	DSP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Total Area (LUT+FF)*8	11496	11408	11752	13184	13344	14048	13512	13408	14112	13480	13352	14048	13520	13408	14112
SPEED	Frequency (ns)	2.426	2.479	2.386	2.434	2.597	2.557	2.687	2.613	2.413	2.621	2.47	2.228	2.627	2.354	2.541
	Frequency (Mhz)	179.40	181,12	178,12	179.66	185,08	183,72	188.22	185,63	178,98	185.9	180,83	173,25	186.12	177,11	183,18
	Design Latency	3	3	3	1	3	3	3	3	3	3	3	3	3	3	3
	Output Latency	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	Throughput/Latency (*Gbps*)	11.48	11,59	11.4	11.49	11,84	11,76	12.04	11,88	11,45	11.9	11,57	11,08	11.91	11,33	11,72
TEST	NIST	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS
	TestU01	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS

6.4.1/ STATSISTICAL TESTS RESULTS

To conclude this experiment section, we have verified that all what we proposed can pass all statistical tests of TestU01, from SmallCruh to BigCrush. Let us recall that the permutation function [167] does not pass Crush and BigCrush when the space is lower than 36 bits, while in our case it does with only 32 bits and a lower modular multiplicative constant. Note that these results are naturally improved when we consider 64 outputted bits, leading to a better throughput.

6.5/ CONCLUSION

In this chapter, we have introduced a new chaotic PRNG implemented in FPGA, which is based on the combination of parallel executions of generalized chaotic iterations and on an efficient mixing scheme. Five Boolean functions have been iterated: the vectorial negation and GCI functions issued from removing a Hamilton cycle in the N-cube. Three interesting strategy builders have been evaluated with each GCI functions. These functions variations lead to a hardware generator with one of the best throughput of the literature, and that can pass the most stringent statistical batteries of tests. If we consider the two conditions of throughput and statistics, we have thus obtained one of the best existing hardware generator. Last, but not least, we have also rigorously proven the chaotic

behaviors of the whole proposal for the Boolean functions.

VI

General Conclusion

7

General Conclusion

Random Number Generators, whether they are software or hardware, are of paramount importance in terms of computer security (generation of encryption keys, masks in symmetric ciphers), simulations in various branches of science, as well as software and hardware tests. First of all, these generators must be as basic as possible. And, secondly, it must be impossible to distinguish in practice, through statistical tests, between the series of numbers produced by these algorithms of and those of real random sequences. Other properties are expected, depending on the scope of application, the most frequent one being to be cryptographically secure, and/or to be chaotic.

A recent software approach developed within the DISC/AND department of the FEMTO-ST Institute, is based on chaotic iterations (CI). This new approach allows a post-processing on a pre-existing random flow, in order to improve their statistical properties, and to add chaos property, while preserving the speed and the cryptographic character of the provided flow. This thesis is in line with the work of FEMTO-ST Institute through chaotic iterations. However, it opens many new perspectives thanks to the hardware implementations of these PRNGs.

In this context, this conclusion chapter firstly presents a synthesis of the key elements proposed in this field (Section 7.1) and further outlines some perspectives (Section 7.2).

7.1/ Contribution Synthesis

Chapter 1 has presented a deep analysis of FPGA implementations of well known PRNGs or TRNGs. This survey helped us to define the prerequisites for our new generators. To the best of our knowledge, up to now, no research work has really deeply investigated hardware implementations of random number generators. From a statistical point of view, we pointed out in this survey that only the first version of chaotic iterations based PRNGs (namely, the XOR-CIPRNG) is able to pass the TestU01 battery. Furthermore, this XOR-CIPRNG was the lowest one in terms of FPGA resources.

Chapter 2 recalled various elements related to chaotic iterations. In these discrete dynamical systems, only a subset of components is updated at each iteration. In addition, it is also reminded that the iterations of such PRNGs are chaotic if and only if the graph of the iteration function is strongly connected. Therefore, we a priori have an infinite collection of chaotic functions.

The second part of this manuscript focuses on quantifying hardware performance of lin-

ear PRNGs on FPGA and ASIC platforms. Chapter 3 selects a set of linear PRNGs to be investigated as possible strategies or for future comparisons with CI. These linear PRNGs are indeed analyzed regarding their main performances in: (1) space, time, and computational complexity, (2) statistical tests, (3) seed and period, (4) arithmetic operators and dynamic range, and (5) throughput *vs.* latency. In this analysis, the chaotic PRNGs are used for throughput comparison, while linear PRNGs are considered for statistical tests analysis.

During this thesis, a FPGA platform and an ASIC one have been realized to accelerate the implementations of RNGs and their statistical evaluation, and to allow the analysis of hardware resources used by these generators. The first platform uses either a Zynq ARM CPU based on AXI BUS communication integrated inside the FPGA or is compatible with AXI Bus and is reconfigurable. The ASIC one deploys an advanced process node as UMC $65 - nm$ and a complete digital synthesis with physical flow of cadence. This has allowed us to have a complete RNG production line: from specification to mathematical level, then to software and RTL implementations with SoC applications, until a final physical circuit.

The third part of this manuscript presented the application of this theory at the hardware level. Chapter 5 shown the implementation on FPGA/ASIC of chaotic iterations CIPRNG based on unary and parallel schemes. Therefore, three different CIPRNGs have been proposed: CIPRNG-XOR, CIPRNG-MC (Multi-Cycle), and CIPRNG-MCMD (Multi-Cycle Multi-Dimension). We have illustrated that all of these configurations pass TestU01 and NIST statistical tests using different strategies. In FPGA platform, it was observed that CIPRNG-XOR performs better in terms of area, throughput over latency, and statistical tests than linear PRNGs that pass TestU01, and than other chaotic PRNGs. As far as we know, only a few articles in the literature have provided both FPGA and ASIC implementation results, in the case of chaotic PRNGs. Therefore, such new results presented in this thesis can be considered as an additional contribution to chaotic PRNGs research domain.

Finally, Chapter 6 proposes a new chaotic iterations PRNG based on a generalized scheme (GCIRNG) for FPGA support. Five Boolean functions have been investigated: the vectorial negation and 4 other functions produced by removing a Hamilton cycle in the N-cube. With such functions, a family of 32 bits and 64 bits GCIPRNGs based on the generalized scheme is proposed. The underlying theory has been regarded, since our proposal has been completely studied in the rigorous framework of chaos theory. We verified that only one iteration (or equivalently one jump) in the N-cube and one mixing function are enough to pass the whole TestU01. If we consider the two criteria of throughput and statistics, we have thus obtained one of the best existing hardware chaotic generator on FPGA. We can consider these versions as complements of the oldest one (that is, the CIPRNG-XOR), but that have a larger throughput over latency ratio.

7.2/ PERSPECTIVES

The experiences, results and knowledge acquired during this work lead to new perspectives for pseudorandom number generators based on generalized chaotic iteration. They are presented below.

- We plan to test and implement other GCI functions obtained by removing a Hamil-

ton cycle in the N-cube. This will aim at finding the relationship between strategies, seed, and other mixing functions, in order to find combinations with maximal periods that lead to a large number of successes in statistical tests. Additionally, more process node and analysis for ASIC (TSMC-180nm and ST-130nm) can be deployed, which leads to a full optimization and comparison at transistor level, if we compare it with the algorithm (logic) level.

- Another perspective deals with proving the cryptographic security of all generators based on generalized scheme with a mixing function. This study should help us to enforce our statements about the superiority of our generators for hardware and software applications.

- Another important work is to propose a TRNG version of our approach. This TRNG is planned to use our GCIPRNG as post processing either with an existing TRNG as strategy, or by creating a new one on both FPGA and ASIC. Two approaches are possible. On the one hand, we can generate randomness by doing a transistor stressing, and the Random Telegraph Noise (RTN) signal of a MOSFET transistor can be used for that. This signal is used during the trapping and detrapping charge carriers. It is very interesting to increase the RTN signal of a transistor by an electrical stress to increase the density of the trap. Finally, each electrical stress resumes the strategy call function of our post processing (these experiments can be done in sub-micrometer transistor MOS already Tapout with TSMC-180nm and characterized in the CDTA/DMN team). On the other hand, the second perspective deploys a laser equipment in relaxing mode, which can produces oscillation as impulsion of a period of femtosecondes, and induces true random number in high speed (Gbps). These experiments can be done either in the Femto-ST Institute, optic department, or in the physical department of our CDTA.

- Applications using PRNGs will also to be investigated (cryptography, watermarking, simulation, etc.) We plan to use the AXI platform (Chapter 4) as a support to facilitate the hardware implementation of these applications in a first stage, and then to integrate them into a system on chip (SoC) with Zynq FPGA as a second stage. Indeed, there is already an ongoing work on chaotic bloc cipher cryptography using different mode of operations as CBC, for which a SoC platform will be welcome.

VII

ANNEXES

A

MATHEMATICAL PROOFS

A.1/ FURTHER INVESTIGATIONS OF THE CHAOTIC BEHAVIOR OF "CHAOTIC ITERATIONS"

In Section 2.2 of the Chapter 2, we have recalled already obtained results regarding the Devaney's chaos of general chaotic iterations. Although well-recognized, this definition is not the sole possible approach to formalize unpredictability and disorder aspects of an iterated system. Indeed, since four decades, mathematicians have proposed various other formulations of a chaotic dynamic, and these formulations are complementary but not equivalent: each definition provides a specific description of the complex behavior of such particular "chaotic" discrete dynamical systems. Let us now investigate other qualitative descriptions of the complex behavior of these chaotic iterations. We first begin with the following result:

Proposition A.1.1. *Let us consider f such that the graph Γ_f is strongly connected. Then, for all open ball $\mathcal{B}$ of $\mathcal{X}$, we can find an iteration $n \in \mathbb{N}$ such that $G_f^n(\mathcal{B}) = \mathcal{X}$.*

Proof. Given $x \in \mathcal{X}$ and $r > 0$, let us recall that the open ball $\mathcal{B}(x, r)$ is the set $\{y \in \mathcal{X} \mid d(x, y) < r\}$. For these x and r, we intend to show that $\exists N_0 \in \mathbb{N}$ such that $G_f^{N_0}(\mathcal{B}(x, r)) = \mathcal{X}$. Without limitation, we can assume that $r < 1$, because if $r' < min(1, r)$ satisfies $\exists N_0 \in \mathbb{N}$ s.t. $G_f^{N_0}(\mathcal{B}(x, r')) = \mathcal{X}$, then as $\mathcal{B}(x, r') \subset \mathcal{B}(x, r)$, we can a fortiori deduce that $G_f^{N_0}(\mathcal{B}(x, r)) = \mathcal{X}$.

Consider the point $y^{(0)} = (\{1\}, \{1\}, \{1\}, \ldots), (0, \ldots, 0))$ of $\mathcal{X}$. As the iteration graph is strongly connected, then G_f is strongly transitive. So there is a point $x^{(0)}$ in the neighborhood $\mathcal{B}(x, r)$ of x and an integer $n^{(0)}$ such that $G_f^{n^{(0)}}(x^{(0)}) = y^{(0)}$. This point $x^{(0)}$ is necessarily of the following form:

- Being inside $\mathcal{B}(x, r)$, with $r < 1$ its second coordinate (the Boolean vector) must be the same than x, due to the Hamming distance in d. In other words, $x_2^{(0)} = x_2$.
- Let $n_0 = -\lfloor \log_{10}(r) \rfloor$. Having regard to the definition of d and as $x^{(0)} \in \mathcal{B}(x, r)$, we necessarily have an equality between the n_0 first terms of the sequence x_1 and the n_0 first terms of the sequence $x_1^{(0)}$.
- As $G_f^{n^{(0)}}\left(x^{(0)}\right) = y^{(0)}$, it is a necessity that after $n^{(0)}$ shifts of the sequence of $x^{(0)}$, we obtain the sequence $(\{1\}, \{1\}, \{1\}, \ldots)$ of $y^{(0)}$.

- Finally, terms of the sequence of $x^{(0)}$ between positions $n_0 + 1$ and $n^{(0)}$ are the ones required for f to transform the Boolean vector of $x^{(0)}$ to the one of $y^{(0)}$ (this is the path to follow in Γ_f to reach $y_2^{(0)}$, starting from $x_2^{(0)}$).

Let us now consider a point $Y^{(0)}$ of the form:

- its Boolean vector $Y_2^{(0)}$ is equal to $y_2^{(0)}$;
- its sequence $Y_1^{(0)}$ is of any kind.

Then the point $X^{(0)}$ defined by:

1. $X_2^{(0)} = x_2^{(0)}$: same Boolean vector than $x^{(0)}$;
2. $\forall k \in [\![0, n_0]\!], X_{1,k}^{(0)} = x_{1,k}^{(0)}$: the $n_0 + 1$ first terms of subset sequences of $X^{(0)}$ and $x^{(0)}$ are equal;
3. $\forall k \in [\![n_0 + 1, n^{(0)}]\!], X_{1,k}^{(0)} = x_{1,k}^{(0)}$: idem for the $n^{(0)} - n_0$ following ones;
4. $\forall k > n^{(0)}, X_{1,k}^{(0)} = Y_{1,k-n^{(0)}}^{(0)}$: the last terms in sequence of $X^{(0)}$ are the whole terms of sequence of $Y^{(0)}$;

is such that:

- $X^{(0)} \in \mathcal{B}(x, r)$, due to the two first items above;
- the Boolean vector of $G_f^{n^{(0)}}\left(X^{(0)}\right)$ is the one of $Y^{(0)}$, due to the third item;
- the sequence of $G_f^{n^{(0)}}\left(X^{(0)}\right)$, which is the one of $X^{(0)}$ after $n^{(0)}$ shifts, is too the sequence of $Y^{(0)}$, due to the forth item.

In other words, for each $Y^{(0)}$ in $\mathcal{X}$ that has $(0, 0, \ldots, 0)$ as Boolean vector, we have found a point $X^{(0)}$ in $\mathcal{B}(x, r)$ and $n^{(0)} \in \mathbb{N}$ such that $G_f^{n^{(0)}}\left(X^{(0)}\right) = Y^{(0)}$.

We now proceed similarly for points having $(0, \ldots, 0, 1)$ as Boolean vector. Consider now the point $y^{(1)} = (\{\{1\}, \{1\}, \{1\}, \ldots), (0, \ldots, 0, 1)) \in \mathcal{X}$. For the same reasons than previously, there exists a point $x^{(1)}$ of $\mathcal{B}(x, r)$ and an integer $n^{(1)}$ such that $G_f^{n^{(1)}}(x^{(1)}) = y^{(1)}$. Let us now consider a point $Y^{(1)}$ of the form:

- its Boolean vector $Y_2^{(1)}$ is equal to $y_2^{(1)}$;
- its sequence $Y_1^{(1)}$ is of any kind.

Then the point $X^{(1)}$ defined by:

1. $X_2^{(1)} = x_2^{(1)}$;
2. $\forall k \in [\![0, n_1]\!], X_{1,k}^{(1)} = x_{1,k}^{(1)}$;
3. $\forall k \in [\![n_1 + 1, n^{(1)}]\!], X_{1,k}^{(1)} = x_{1,k}^{(1)}$;

4. $\forall k > n^{(1)}, X^{(1)}_{1,k} = Y^{(1)}_{1,k-n^{(1)}}$;

is such that $X^{(1)} \in \mathcal{B}(x,r)$ and $G_f^{n^{(1)}}\left(X^{(1)}\right) = Y^{(1)}$: any point of $\mathcal{X}$ having $(0,\ldots,0,1)$ as Boolean vector can be reached from $\mathcal{B}(x,r)$ with $n^{(1)}$ iterations of G_f.

This process can be extended accordingly until the point $y^{(2^{\mathsf{N}}-1)} = (((\{1\},\{1\},\{1\},\ldots),(1,\ldots,1,1)))$, which leads to the definition of $n^{(2^{\mathsf{N}}-1)}$, of $x^{(2^{\mathsf{N}}-1)}$, of $Y^{(2^{\mathsf{N}}-1)}$, and finally of $X^{(2^{\mathsf{N}}-1)}$.

At this stage, we can claim that, for all y of $\mathcal{X}$, it is possible to find $x' \in \mathcal{B}(x,r)$ and a certain integer $N \in \{n^{(0)},\ldots,n^{(2^{\mathsf{N}}-1)}\}$ such that $G_f^N(x') = y$. The last issue to solve is that the iteration number N depends on the Boolean vector y_2, which should not be the case.

Let us consider $N_0 = max(\{n^{(k)}, k = 0..2^{\mathsf{N}}-1\})$. In each sequence of subsets $X^{(k)}_1, k \in [\![0, 2^{\mathsf{N}}-1]\!]$, it is possible to incorporate $N_0 - k$ times the empty set $\varnothing$ between terms $X^{(k)}_{1,n^{(k)}}$ and $X^{(k)}_{1,N_0}$, in such a way that:

$$Y^{(k)} = G_f^{n^{(k)}}\left(X^{(k)}\right) = G_f^{n^{(k)}+1}\left(X^{(k)}\right) = \ldots = G_f^{N_0}\left(X^{(k)}\right),$$

which is equivalent to be on a treadmill once reaching the target $Y^{(k)}$ and until having iterated N_0 times. Thanks to that, for all $y \in \mathcal{X}$, it is possible to find $x' \in \mathcal{B}(x,r)$ such that $G_f^{N_0}(x') = y$, which is the expected result. □

Therefore, however small the starting open ball, we finish to reach the whole $\mathcal{X}$ space by iterating G_f. Using this result, we can deduce the following proposition related to chaos.

Proposition A.1.2. *General chaotic iterations G_f are topologically mixing: for all couple of nonempty open sets U and V, there is $n_0 \in \mathbb{N}$ such that $\forall n \geqslant n_0$, $G_f^n(U) \cap V \neq \varnothing$.*

Proof. Let us consider U and V, two disjoint nonempty open sets of $\mathcal{X}$. U being a nonempty open set, we can find $x \in \mathcal{X}$ and $r > 0$ such that $\mathcal{B}(x,r) \subset U$. Due to Proposition A.1.1, $\exists n_0$ s.t. $G_f^{n_0}(\mathcal{B}(x,r)) = \mathcal{X}$. As $\mathcal{B}(x,r) \subset U$, we have too $G_f^{n_0}(U) = \mathcal{X}$, and so $G_f^{n_0}(U) \cap V \neq \varnothing$. Let us consider $Y \in G_f^{n_0}(U) \cap V$. It exists $X^{(0)} \in U$ such that $G_f^{n_0}(X^{(0)}) = Y$. The point $X^{(1)}$ defined by:

- $X^{(1)}_2 = X^{(0)}_2$;
- $\forall k \leqslant n_0, X^{(1)}_{1,k} = X^{(0)}_{1,k}$: the two sequences start by the same terms;
- $X^{(1)}_{1,n_0+1} = \varnothing$: we insert an empty set at position n_0+1 in sequence $X^{(1)}_1$;
- $\forall k > (n_0+1), X^{(1)}_{1,k} = X^{(0)}_{1,k-1}$;

is such that $G_f^{n_0+1}(X^{(1)}) = Y$, and so $G_f^{n_0+1}(U) \cap V \neq \varnothing$. Similarly, by incorporating l empty sets between positions n_0+1 and n_0+l inside the sequence of $X^{(0)}$, we are able to define a point $X^{(l)}$ which is such that$G_f^{n_0+l}(X^{(l)}) = Y$, proving so that $G_f^{n_0+l}(U) \cap V \neq \varnothing$. This inequality being valid for all $l > 0$, we can deduce the topological mixing of G_f. □

Proposition A.1.3. *When considering the vectorial negation for f, the general chaotic iterations satisfy the Knudsen's definition of chaos [157]: they are sensible to the initial condition and they have a dense orbit.*

Proof. The sensibility to the initial condition of G_f has already been stated in [156]. We are then left to construct a point $x \in \mathcal{X}$ such that the set $\{G_f^n(x) \mid n \in \mathbb{N}\}$ is dense in $\mathcal{X}$: iterations of $G_f^n(x)$ must be as close as possible to any point $y \in \mathcal{X}$.

Let us denote by $s_0, s_1, \ldots, s_{2^{\mathsf{N}}-1}$ the list of each subset of $[\![1, \mathsf{N}]\!]$: $s_0 = \varnothing$, $s_1 = \{\mathsf{N}\}$, $s_2 = \{\mathsf{N}-1\}$, $s_3 = \{\mathsf{N}-1, \mathsf{N}\}$, ..., $s_{2^{\mathsf{N}}-1} = \{1, 2, ..., \mathsf{N}\}$. Let us now consider a point $y \in \mathcal{X}$. Its Boolean vector y_2 can be associated to a given s_k, namely the subset of $[\![1, \mathsf{N}]\!]$ that contains the coordinates of 1's in y_2. The first term $y_{1,0}$ of sequence y_1, for its part, is a given $s_{k'}$, while the second term $y_{1,1}$ is too a given $s_{k''}$, with $k, k', k'' \in [\![0, 2^{\mathsf{N}} - 1]\!]$.

Let us now remark that, when iterating G_f on the point $((s_k, s_{k'}, s_{k'}, s_k, ...), (0, 0, \ldots, 0))$, with f the vectorial negation:

- We start on the Boolean vector $(0, 0, \ldots, 0)$;
- As s_k indicates the 1's in vector y and we use the vectorial negation, we thus have, after one iteration of G_f:
  - the Boolean vector $(0, 0, \ldots, 0)$ is changed in y_2;
  - the sequence is shifted of one position, so it now starts by $(s_{k'}, s_{k'}, s_k, ...)$.

  Having the same Boolean vector and the same first term in the sequence, we are thus at a distance lower than 10^{-1} to y after one iterate.
- Iterating G_f another time switches the binary digits in positions $s_{k'}$ in the Boolean vector, while shifting the sequence so that it becomes $(s_{k'}, s_{k'}, s_k, ...)$.
- Iterating twice G_f will operate a second time the negation on Boolean digits at position $s_{k'}$ and s_k,

and so the new Boolean state is $(0, 0, \ldots, 0)$ again after these 4 iterations. To sum up, iterating four times starting from $((s_k, s_{k'}, s_{k'}, s_k, ...), (0, 0, \ldots, 0))$ will first leave the $(0, 0, \ldots, 0)$ vector to be at a distance 10^{-1} to y, and then come back to $(0, 0, \ldots, 0)$ after shifting 4 times the sequence.

Let us consider now the point:

$$
\begin{array}{ll}
((& s_0, s_0, s_0, s_0, \\
& s_0, s_1, s_1, s_0, \\
& ..., \\
& s_0, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, s_0, \\
& s_1, s_0, s_0, s_1, \\
& s_1, s_1, s_1, s_1, \\
& ..., \\
& s_1, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, s_1, \\
& s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, s_{2^{\mathsf{N}-1}}, ...), \quad (0, 0, \ldots, 0))
\end{array}
$$

By iterating G_f on it, we will be at one time at 10^{-1} of any point of $\mathcal{X}$, while recovering the null Boolean vector at each 4 iterates.

Let us now remark that if, instead of considering $((s_k, s_{k'}, s_{k'}, s_k, ...), (0, 0, \ldots, 0))$ in our first explanations, we have regarded $((s_k, s_{k'}, s_{k''}, s_{k''}, s_{k'}, s_k, ...), (0, 0, \ldots, 0))$, then after one iterate the dynamical system is at a distance lower than 10^{-2} to y, as we have applied the

negation on bits at positions s_k, and because the sequence starts by $(s_{k'}, s_{k''}, s_{k''}, s_{k'}, s_k, ...)$ after one shift. So, if we concatenate all the possible (s, s', s'', s'', s', s), $s, s', s'' \subset [\![1, \mathsf{N}$, in the point defined above, we will then be at a distance lower than 10^{-2} of any point of $\mathcal{X}$ during iterations of G_f. Continuing the process with patterns of length 8, 10, etc., will define a unique point x whose iterates are as close as possible to any point of $\mathcal{X}$, leading to a dense orbit. □

Let us consider the point x constructed in the previous proof, which has a dense orbit in $\mathcal{X}$. It is easy to see that the points $G_f(x)$, $G_f^2(x)$, $G_f^3(x)$... have too a dense orbit. And so, points having a dense orbit are dense in $\mathcal{X}$, leading to another aspect of chaos, as targetted by Knudsen.

A.2/ MATHEMATICAL CHAOS OF THE PROPOSED DESIGN OF GCIPRNG

A.2.1/ FIRST CONSIDERATIONS

We want now to characterize how much chaotic is our proposal for PRNG based on generalized chaotic iterations as described in Chapter 6. By chaos, we mean that the effects, on the output sequence, of any slight alteration of either the seed or the embedded PRNG cannot be predicted in the short or medium term. Note that both inputs and outputs are constituted in binary (sequences of) vectors: each element is constituted by a finite number of bounded integers, but the set of all possible inputs/outputs is infinite (countable). Additionally, with two different strategies or seeds, we can reach twice the same x^t, but with two different x^{t+1}: this finite state machine does not necessarily enter into a loop, as at each iteration we take a new value from the "outside world", that is, from the strategy (this machine does not iterate in a vacuum). Furthermore, this strategy can be non-periodic, if we consider a TRNG like a physical white noise or any physical source of entropy of that kind.

To sum up, by designing a finite state machine that only manipulate integers (no floating point issue), but which takes a new input from the outside world at each iteration, we thus obtain an iterative process working on an infinite space, and which does not enter necessarily within a loop. As the whole algorithm may be described as an iteration function, we can thus quantify how much dependent is the output to a slight alteration of the input, using the well reputed and rigorous mathematical theory of chaos. We are then left to evaluate the chaotic behavior of the proposal depicted in Figure 6.1.

A.2.2/ PROOF OF CHAOS: THE INTERNAL PROCESS

Let us recall or specify first some notations and definitions in use in this section. In what follows, $\mathbb{B}$ is the Boolean set, while $\mathbb{N}$ and $\mathbb{R}$ are the usual sets of real and integer numbers, with the notations of $\mathbb{N}^*$ and $\mathbb{R}_+$ for their positive (or equal to zero) subsets. For $a, b \in \mathbb{N}$, $[\![a, b]\!]$ is the set of integers: $\{a, a+1, \ldots, b\}$. $X^{\mathbb{N}}$ is the set of sequences belonging in X and s^t is the t-th term of a sequence $s = (s^t)_{t\in\mathbb{N}}$, which may be a vector (thus explaining the use of an exponent). Finally, f^n means the n-th composition of the function f (*i.e.*, $f^n = f \circ f \circ \ldots \circ f$), while v_i is the i-th component of a vector v. In case where this i-th

component v_i of vector v is itself a vector, the notation $v_{i,j}$ stands for the j-th coordinate of the latter. Similarly, if s is a sequence of sequences, the j-th term of sequence s^i will be denoted by $s^{i,j}$.

We first focus on the proposal without the mixing, which is referred as the internal process.

For $\mathsf{N} \in \mathbb{N}^*$, let us define the set

$$\mathcal{X}_\mathsf{N} = \mathbb{B}^\mathsf{N} \times \left(\mathbb{B}^\mathsf{N}\right)^\mathbb{N},$$

and the following functions:

$$\begin{array}{rccc} i_\mathsf{N} : & \left(\mathbb{B}^\mathsf{N}\right)^\mathbb{N} & \longrightarrow & \mathbb{B}^\mathsf{N} \\ & (s^t)_{t\in\mathbb{N}} & \longmapsto & s^0, \end{array}$$

and

$$\begin{array}{rccc} \sigma_\mathsf{N} : & \left(\mathbb{B}^\mathsf{N}\right)^\mathbb{N} & \longrightarrow & \left(\mathbb{B}^\mathsf{N}\right)^\mathbb{N} \\ & (s^t)_{t\in\mathbb{N}} & \longmapsto & (s^{t+1})_{t\in\mathbb{N}}. \end{array}$$

They respectively extracts the first term of an entrance sequence (i_N), and shifts it to the left by removing the first term (σ_N).

We now define the distances:

$$\begin{array}{rccc} d_{\mathsf{N},E} : & \mathbb{B}^\mathsf{N} \times \mathbb{B}^\mathsf{N} & \longrightarrow & \mathbb{R}_+ \\ & (e, e') & \longmapsto & \sum_{k=1}^\mathsf{N} |e_i - e'_i|, \end{array}$$

which is the Hamming distance on $\mathbb{B}^\mathsf{N}$, and

$$\begin{array}{rccc} d_{\mathsf{N},S} : & \left(\mathbb{B}^\mathsf{N}\right)^\mathbb{N} \times \left(\mathbb{B}^\mathsf{N}\right)^\mathbb{N} & \longrightarrow & \mathbb{R}_+ \\ & (s, s') & \longmapsto & \dfrac{9}{\mathsf{N}} \sum_{t=0}^\infty \dfrac{d_{\mathsf{N},E}(s^t, s'^t)}{10^{t+1}}. \end{array}$$

It has been already proven in [156] (by identifying, *mutatis mutandis*, the set of subsets of $[\![1, \mathsf{N}]\!]$ with $\mathbb{B}^\mathsf{N}$) that, with the distance $d_\mathsf{N} = d_{\mathsf{N},E} + d_{\mathsf{N},S}$, $\mathcal{X}_\mathsf{N}$ becomes a metric space. We define

$$\begin{array}{rccc} F_{\mathsf{N},f} : & \mathbb{B}^\mathsf{N} \times \mathbb{B}^\mathsf{N} & \longrightarrow & \mathbb{B}^\mathsf{N} \\ & (b, e) & \longmapsto & F_{\mathsf{N},f}(b, e), \end{array}$$

where $\forall i \in [\![1, \mathsf{N}]\!]$,

$$F_{\mathsf{N},f}(b, e)_i = \begin{cases} e_i & \text{if } b_i = 0, \\ f(e)_i & \text{else,} \end{cases}$$

and

$$\begin{array}{rccc} g_f : & \mathcal{X}_8 & \longrightarrow & \mathcal{X}_8 \\ & (e, s) & \longmapsto & \left(F_{8,f}(i_8(s), e); \sigma_8(s)\right). \end{array}$$

It has already been established, in [156], that such general iterations are continuous on the metric space $(\mathcal{X}_8, d_8)$, and that the discrete dynamical space $x^0 \in \mathcal{X}_8$, $x^{t+1} = g_f(x^t)$ is chaotic, according to Devaney, on $(\mathcal{X}_8, d_8)$. It is shown also that this dynamical system is strongly transitive [156].

Given $n, N \in \mathbb{N}, N \geqslant n$, we define:

$$\begin{array}{rccc} \psi_{n,N} : & [\![1, N-n+1]\!] \times \mathbb{B}^N & \longrightarrow & \mathbb{B}^n \\ & (t, e) & \longmapsto & (e_t, \ldots, e_{t+n-1}), \end{array}$$

and, similarly, $\Psi_{n,N}$, as follows:

$$\begin{array}{ccc} [\![1, N-n+1]\!] \times \left(\mathbb{B}^N\right)^{\mathbb{N}} & \longrightarrow & (\mathbb{B}^n)^{\mathbb{N}} \\ (t, (s_i)_{i\in\mathbb{N}}) & \longmapsto & \left(\Psi_{n,N}(S_i)\right)_{i\in\mathbb{N}}, \end{array}$$

and, finally,

$$\begin{array}{rccl} h: & \mathcal{X}_{32} & \longrightarrow & \mathcal{X}_{32} \\ & (e,s) & \longmapsto & \Big(\big(g_f\left(\psi_{8,32}(1,e), \Psi_{8,32}(1,s)\right)_{1,1}, \\ & & & \vdots \\ & & & g_f\left(\psi_{8,32}(1,e), \Psi_{8,32}(1,s)\right)_{1,8}, \\ & & & g_f\left(\psi_{8,32}(9,e), \Psi_{8,32}(9,s)\right)_{1,1}, \\ & & & \vdots \\ & & & g_f\left(\psi_{8,32}(9,e), \Psi_{8,32}(9,s)\right)_{1,8}, \\ & & & g_f\left(\psi_{8,32}(17,e), \Psi_{8,32}(17,s)\right)_{1,1}, \\ & & & \vdots \\ & & & g_f\left(\psi_{8,32}(17,e), \Psi_{8,32}(17,s)\right)_{1,8}, \\ & & & g_f\left(\psi_{8,32}(25,e), \Psi_{8,32}(25,s)\right)_{1,1}, \\ & & & \vdots \\ & & & g_f\left(\psi_{8,32}(25,e), \Psi_{8,32}(25,s)\right)_{1,8}), \\ & & & \sigma_{32}(s)\Big). \end{array}$$

We can remark that the h function is what is iterated inside our proposal, if we except the mixing. Indeed, the four blocks (binary digits ranging from 1 to 8, and then from 9 to 16, from 17 to 24, and finally from 25 to 32) appear well in the first component of the output of h.

We will show that iterations of h are chaotic on $\mathcal{X}_{32}$ and, using a topological semi-conjugacy, that the mixing does not alter such an unpredictable behavior. In order to do so, we must first check that,

Proposition A.2.1. *h is a continuous map on the metric space $(\mathcal{X}_{32}, d_{32})$.*

. Let us consider a sequence $x^n = (e^n, s^n)_{n\in\mathbb{N}} \in \mathcal{X}_{32}^{\mathbb{N}}$, which is convergent to an element $x = (e, s) \in \mathcal{X}_{32}^{\mathbb{N}}$. As

$$\begin{array}{rl} d_{32}(x^n, x) & \longrightarrow 0 \\ & = d_{32,E}(e^n, e) + d_{32,S}(s^n, s), \end{array}$$

and due to the fact that $d_{32,E}$ produces only integers, we have $\exists n_1 \in \mathbb{N}, n \geqslant n_1 \Rightarrow e^n = e$.

Similarly, $d_{32,S}(s^n, s) \longrightarrow 0$, so $\exists n_2 \in \mathbb{N}$ such that $n \geqslant n_2 \Rightarrow d_{32,S}(s^n, s) < \dfrac{1}{10^{32}}$. Due to the way we defined $d_{\mathrm{N},S}$, we can conclude that $\forall n \geqslant n_2$, the sequence s^n has the same 32 first terms than the sequence s. And we can conclude from these two facts that

$$\forall n \geqslant \max\{n_1, n_2\}, h(e^n, s^n)_1 = h(e, s)_1.$$

Finally, for $n \geqslant n_2$, we have $\forall i < 32$, $s^{n,i} = s^i$. So, $\forall n \geqslant n_2$,

$$\begin{aligned} d_{32,S}(s^n, s) &= \frac{9}{\mathsf{N}} \sum_{t=0}^{\infty} \frac{d_{\mathsf{N},E}(s^{n,t}, s^t)}{10^{t+1}} \\ &= \frac{9}{\mathsf{N}} \sum_{t=0}^{\infty} \frac{d_{\mathsf{N},E}(\sigma(s^n)^t, \sigma(s)^t)}{10^{t+1}} \\ &= \frac{d_{32,S}(\sigma_{32}(s^n), \sigma_{32}(s))}{10}. \end{aligned}$$

As $d_{32,S}(s^n, s) \longrightarrow 0$, we can deduce that $d_{32,S}(\sigma_{32}(s^n), \sigma_{32}(s)) \longrightarrow 0$, and so:

$$h(e^n, s^n)_2 \longrightarrow h(e, s)_2.$$

As a conclusion, for all sequence $x^n = (e^n, s^n)_{n\in\mathbb{N}}$ of $\mathcal{X}_{32}^{\mathbb{N}}$, if $x^n \longrightarrow x = (e, s) \in \mathcal{X}_{32}$, then $h(x^n) \longrightarrow h(x)$. This is the sequential characterization of the continuity, and so h is continuous on $(\mathcal{X}_{32}, d_{32})$. □

Let us now show that:

Proposition A.2.2. *If g_f is strongly transitive on $\mathcal{X}_8$, then h_f is chaotic according to Devaney on $(\mathcal{X}_{32}, d_{32})$.*

Proof. Let us first prove that,

Lemma A.2.3. *If g_f is strongly transitive on $\mathcal{X}_8$, then h_f is strongly transitive on $(\mathcal{X}_{32}, d_{32})$.*

Proof. Let $x = (e, s)$ and $\check{x} = (\check{e}, \check{s})$ two points of $\mathcal{X}_{32}$, and $\varepsilon > 0$. We must find $x' = (e', s')$ inside the open ball $\mathcal{B}(x, \varepsilon) = \{u \in \mathcal{X}_{32}, d_{32}(u, x) < \varepsilon\}$ such that:

$$h_f^n(x') = \check{x}.$$

Let us consider:

$$\begin{aligned} p_1 = (\psi_{8,32}(1, e); (\quad & \Psi_{8,32}(1, s), \Psi_{8,32}(32 + 1, s), \\ & \Psi_{8,32}(2 \times 32 + 1, s), \ldots, \\ & \Psi_{8,32}(k \times 32 + 1, s), \ldots)), \\ q_1 = (\psi_{8,32}(1, \check{e}); (\quad & \Psi_{8,32}(1, \check{s}), \Psi_{8,32}(32 + 1, \check{s}), \\ & \Psi_{8,32}(2 \times 32 + 1, \check{s}), \ldots, \\ & \Psi_{8,32}(k \times 32 + 1, \check{s}), \ldots)), \end{aligned}$$

in which the second components are infinite sequences of $\mathbb{B}^8$. p_1 and q_1 belong to $\mathcal{X}_8$ and g_f is strongly transitive, so there exist $\widetilde{p_1} = ((\widetilde{e_1}, \ldots \widetilde{e_8}), (\widetilde{s_1}, \widetilde{s_2}, \ldots))$ in $\mathcal{B}(p_1, \varepsilon)$ and $n_1 \in \mathbb{N}$ such that:

$$g_f^{n_1}(\widetilde{p_1}) = q_1.$$

We can apply the same process on points:

$$\begin{aligned} p_2 = (\psi_{8,32}(9, e); (\quad & \Psi_{8,32}(9, s), \Psi_{8,32}(32 + 9, s), \\ & \Psi_{8,32}(2 \times 32 + 9, s), \ldots, \\ & \Psi_{8,32}(k \times 32 + 9, s), \ldots)), \\ q_2 = (\psi_{8,32}(9, \check{e}); (\quad & \Psi_{8,32}(9, \check{s}), \Psi_{8,32}(32 + 9, \check{s}), \\ & \Psi_{8,32}(2 \times 32 + 9, \check{s}), \ldots, \\ & \Psi_{8,32}(k \times 32 + 9, \check{s}), \ldots)), \end{aligned}$$

leading to the existence of $\widetilde{p_2} \in \mathcal{X}_8$ and $n_2 \in \mathbb{N}$ such that $g_f^{n_2}(\widetilde{p_2}) = q_2$. The process if finally applied on the last two "quarters" of $\mathcal{X}_{32} = \mathbb{B}^{32} \times \left(\mathbb{B}^{32}\right)^{\mathbb{N}}$, dividing the first (resp. second) set of the Cartesian product in 4 vectors of 8 bits (resp. in sequences belonging in $\mathbb{B}^8$) thanks to $\phi_{8,32}$ (resp. $\Phi_{8,32}$). This leads to the points of $\mathcal{X}_8$ defined below:

$$\begin{array}{rl}
p_3 = (\psi_{8,32}(17,e);(& \Psi_{8,32}(17,s), \Psi_{8,32}(32+17,s),\\
& \Psi_{8,32}(2\times 32+17,s),\ldots)),\\
q_3 = (\psi_{8,32}(17,\check{e});(& \Psi_{8,32}(17,\check{s}), \Psi_{8,32}(32+17,\check{s}),\\
& \Psi_{8,32}(2\times 32+17,\check{s}),\ldots)),\\
p_4 = (\psi_{8,32}(25,e);(& \Psi_{8,32}(25,s), \Psi_{8,32}(32+25,s),\\
& \Psi_{8,32}(2\times 32+25,s),\ldots)),\\
q_4 = (\psi_{8,32}(25,\check{e});(& \Psi_{8,32}(25,\check{s}), \Psi_{8,32}(32+25,\check{s}),\\
& \Psi_{8,32}(2\times 32+25,\check{s}),\ldots)).
\end{array}$$

As previously, due to the strong transitivity of g_f, we have the existence of $\widetilde{p_3}, \widetilde{p_4} \in \mathcal{X}_8$ and of $n_3, n_4 \in \mathbb{N}$ such that $g_f^{n_3}(\widetilde{p_3}) = q_3$ and $g_f^{n_4}(\widetilde{p_4}) = q_4$.

Let us introduce the following notation: $\widetilde{p_i} = (\widetilde{e_i}, \widetilde{S_1})$ for $i = 1,\ldots,4$, and $n_0 = \max_{i=1}^4\{n_i\}$. We define:

$$\widetilde{s_i}^t = \begin{cases} S_i^t & \text{if } t \leqslant n_i,\\ 0 & \text{if } t \in [\![n_i+1, n_0]\!],\\ S_i^{t-n_0+n_i} & \text{else.}\end{cases}$$

Indeed, for each quarter of $\mathcal{X}_{32}$ we have four different $n_i, i = 1..4$, number of iterations to reach a given point of $\mathcal{X}_8$ by starting to a neighborhood of another given point of this quarter. By adding 0's in the iteration sequence, we thus allow to iterate in a vacuum the required number of times in each quartet, so that after n_0 iterations each 4 parts of the targeted point x' are reached. Let us do it with details.

Let us consider the point $p' = (e', s') \in \mathcal{X}_{32}$ defined by:

- $e' = (\widetilde{e}_{1,1},\ldots,\widetilde{e}_{1,8},\widetilde{e}_{2,1},\ldots,\widetilde{e}_{2,8},\widetilde{e}_{3,1},\ldots,\widetilde{e}_{3,8},\ \widetilde{e}_{4,1},\ldots,\widetilde{e}_{4,8}) \in \mathbb{B}^{32}$,
- $s' = ((\widetilde{s}_{1,8k+1},\ldots,\widetilde{s}_{1,8k+8},\widetilde{s}_{2,8k+1},\ldots,\widetilde{s}_{2,8k+8},\ \widetilde{s}_{3,8k+1},\ldots,\widetilde{s}_{3,8k+8},\widetilde{s}_{4,8k+1},\ldots,\widetilde{s}_{4,8k+8}))_{k\in\mathbb{N}}$, which is a sequence of $\left(\mathbb{B}^{32}\right)^{\mathbb{N}}$.

By construction, this point of $\mathcal{X}_{32}$ is such that $h_f^{n_0}(x') = \check{x}$ and $x' \in \mathcal{B}(x,\varepsilon)$. □

Let us now finalize the proof of Prop. A.2.2. h_f being strongly transitive on $(\mathcal{X}_{32}, d_{32})$, it is thus transitive. We are then left to establish the regularity of h_f.

Let us consider $x = (e,s) \in \mathcal{X}_{32}$, and $\varepsilon > 0$. We need to find a periodic point $x' = (e',s')$ inside $\mathcal{B}(x,\varepsilon)$. As ε may be lower than 1, and due to the definition of d_{32}, we must choose $e' = e$. Let $k_0 = -\lfloor \log_{10}(\varepsilon)\rfloor + 1$ the integer such that any point of the form $(e,(s^0,s^1,\ldots,s^{k_0},a,b,c,\ldots))$ is inside $\mathcal{B}(x,\varepsilon)$.

Let us denote by $\check{x} = (\check{e},\check{s})$ the point $h_f^{k_0}(x)$. Due to the strong transitivity of h_f (Lemma A.2.3), there is a point $\widetilde{x} = (\widetilde{e},\widetilde{s})$ in $\mathcal{B}(\check{x}, 0.1)$ and $k_1 \in \mathbb{N}$ such that $h_f^{k_1}(\widetilde{x}) = x$. Finally, the point $x' = (e,(s^0,s^1,\ldots,s^{k_0},\widetilde{s}^0,\ldots,\widetilde{s}^{k_1},s^0,s^1,\ldots,s^{k_0},\widetilde{s}^0,\ldots,\widetilde{s}^{k_1},\ldots)$ is k_0+k_1 periodic and inside the neighborhood $\mathcal{B}(x,\varepsilon)$ of x, which proves the regularity, and then the chaotic behavior of h_f. □

B

PRNG IMPLENTED ON FPGA

B.1/ LINEAR PRNG ON FPGA

Listing B.1: LFSR113

```
1  `timescale 1ns / 1ps
2  module lfsr113(/*AUTOARG*/
3     // Outputs
4     lfsr113_prng,
5     // Inputs
6     CLK, reset,enable_p
7     ) ;
8
9
10    input CLK;
11    input reset,enable_p;
12    output reg [31:0] lfsr113_prng;
13
14    reg [31:0] z1, z2, z3,z4;
15
16     //== State variables
17    reg      state;
18    reg         next;
19    // End of automatics
20
21    parameter [0:0]
22      CI_S0      = 1'b0,
23       CI_IDLE   = 1'b1;
24
25 reg [78:0]           state_ascii_r;          // Decode of current
26    always @(state) begin
27       case ({state})
28             CI_S0:   state_ascii_r = "CI_S0  ";
29          CI_IDLE:   state_ascii_r = "CI_IDLE  ";
30        default:   state_ascii_r = "CI_IDLE  ";
31       endcase
32    end
33
34    //== assign prng1 and prng2 register
35    //== Initialisation FSM one-hot encoding
36    always @(posedge CLK) begin
37       if (!reset) begin
38             state = CI_IDLE;
39         end else begin
40             state=next;
41         end
42     end
43
44    //== Initialisation FSM one-hot encoding
45     always @(*) begin
46         next = CI_IDLE;
47       case (state)
48
49         CI_IDLE: begin
50             if (enable_p==1'b0) begin
51                 next=CI_IDLE;
52             end else begin
53                 next=CI_S0;
54             end
55         end
56         CI_S0: begin
57             next=CI_S0;
58         end
59         default: next=CI_IDLE;
60         endcase
61     end
62
63 //always @(state or enable or enable_tmp or mci or countci or j or reset) begin
```

```
64      always @(posedge CLK) begin
65      if (!reset) begin
66              z1= 32'd987654321;
67              z2= 32'd987654321;
68              z3= 32'd987654321;
69              z4= 32'd987654321;
70      end else begin
71        case (state)
72              CI_IDLE : begin
73                  //xorshift64_prng = xorshift64_prng;
74                  z1= z1; z2= z2;z3= z3;z4= z4;
75              end
76              CI_S0 : begin
77                  z1 = (((z1 & 32'd4294967294) << 18) ^ (((z1 << 6) ^ z1) >> 13));
78                  z2 = (((z2 & 32'd4294967288) << 2) ^ (((z2 << 2) ^ z2) >> 27));
79                  z3 = (((z3 & 32'd4294967280) << 7) ^ (((z3 << 13) ^ z3) >> 21));
80                  z4 = (((z4 & 32'd4294967168) << 13) ^ (((z4 << 3) ^ z4) >> 12));
81              end
82          endcase
83      end
84  end
85
86     //assign lfsr113_prng =  (state == CI_S0)? z1 ^ z2 ^ z3 ^ z4: 32'b0;
87      always @(posedge CLK ) begin
88        if (!reset) begin
89              lfsr113_prng <= 32'b0;
90          end else if (state==CI_S0) begin
91              lfsr113_prng <= z1 ^ z2 ^ z3 ^ z4;
92          end else begin
93              lfsr113_prng <= lfsr113_prng;
94          end
95     end
96
97  endmodule
```

Listing B.2: Taus88

```
1   module taus88(
2      // Outputs
3      taus88_prng,
4      // Inputs
5      CLK, reset, enable_p
6      ) ;
7
8      input CLK;
9      input reset, enable_p;
10     output reg [31:0] taus88_prng;
11
12
13
14     reg [31:0]    s1, s2, s3;
15
16
17      //== State variables
18     reg       state;
19     reg         next;
20     // End of automatics
21
22     parameter [0:0]
23       CI_S0      = 1'b0,
24        CI_IDLE   = 1'b1;
25
26  reg [78:0]           state_ascii_r;          // Decode of current
27     always @(state) begin
28        case ({state})
29              CI_S0:   state_ascii_r = "CI_S0  ";
30            CI_IDLE:   state_ascii_r = "CI_IDLE  ";
31          default:   state_ascii_r = "CI_IDLE  ";
32        endcase
33     end
34
35     //== assign prng1 and prng2 register
36     //== Initialisation FSM one-hot encoding
37     always @(posedge CLK) begin
38        if (!reset) begin
39              state = CI_IDLE;
40          end else begin
41              state=next;
42          end
43      end
44
45     //== Initialisation FSM one-hot encoding
46      always @(*) begin
47          next = CI_IDLE;
48        case (state)
49
50          CI_IDLE: begin
51              if (enable_p==1'b0) begin
52                  next=CI_IDLE;
53              end else begin
54                  next=CI_S0;
55              end
56          end
```

```
57          CI_S0: begin
58              next=CI_S0;
59          end
60          default: next=CI_IDLE;
61          endcase
62      end
63
64  //always @(state or enable or enable_tmp or mci or countci or j or reset) begin
65      always @(posedge CLK) begin
66      if (!reset) begin
67               s1 <= 32'd12345;
68               s2 <= 32'd12345;
69               s3 <= 32'd12345;
70      end else begin
71        case (state)
72              CI_IDLE : begin
73                  //xorshift64_prng = xorshift64_prng;
74                  s1<= s1; s2<= s2; s3<= s3;
75              end
76              CI_S0 : begin
77               s1 <= (((s1 & 32'd4294967294) << 12) ^ (((s1 << 13) ^ s1) >> 19));
78               s2 <= (((s2 & 32'd4294967288) << 4) ^ (((s2 << 2) ^ s2) >> 25));
79               s3 <= (((s3 & 32'd4294967280) << 17) ^ (((s3 << 3) ^ s3) >> 11));
80              end
81          endcase
82      end
83  end
84
85  always @(posedge CLK ) begin
86        if (!reset) begin
87              taus88_prng <= 32'b0;
88          end else if (state==CI_S0) begin
89              taus88_prng <= s1 ^ s2 ^ s3;
90          end else begin
91              taus88_prng <= taus88_prng;
92          end
93     end
94
95  endmodule
```

B.2/ SOFTWARE PART OF SOC BASED ZYNQ

Listing B.3: SDK Main function for PRNG based Zynq

```
1   #define PACKET_SIZE 256
2   int main(){
3       // Initialize the zynq platform (PS)
4       init_platform();
5
6       // enable the PL.
7       ps7_post_config ();
8
9       // Initialize AXI DMA
10      xil_printf ("initializing axi dma ...\n\r");
11      InitializeAXIDma ();
12
13      // Enable PRNG or CIPRNG Controller
14      // End of frame will come after 128 bytes (32 words) are transferred.
15      xil_printf ("setting up SampleGenerator unit...\n\r");
16      EnableSampleGenerator ( PACKET_SIZE / 4 );
17
18      // set the interrupt system and interrupt handling
19      // clear interrupt. just perform a write to bit no. 12 of S2MM_DMASR
20      xil_printf ("enabling the interrupt handling system...\n\r");
21      InitializeInterruptSystem ( XPAR_PS7_SCUGIC_0_DEVICE_ID );
22
23      // Start DMA Transfer
24      // write destination address to S2MM_DA register.
25      // write length to S2MM_LENGTH register.
26      StartDMATransfer ( 0xa000000, PACKET_SIZE );
27
28      // Data is in the DRAM ! do your processing here !
29        u32 tt,i;
30          for(i=0; i<(PACKET_SIZE/4); i++) {
31              tt=Xil_In32(0xa000000 + 0x04*i);
32              xil_printf("these is the first outputs i:%10lu  DDR:%10lu = %10lu\n\r", i, (0x04*i), tt);
33          }
34      return 0;
35  }
```

B.3/ SOFTWARE PART OF AXI-PLATFORM

To give an illustration, Listing B.4 resumes the main function of the new firmware. The write operation consist of configuration the platform register to select which RTL IP to be configure first, reset, and define latency for PRNG or the final outputs. While the read operation is used to capture response from the platform to confirm any read and write operation is completed by receiving their addresses from FPGA.

Listing B.4: New Firmware Main function for PRNG and CPRNG (AXI platform)

```
// Opens the USB serial port and continuously attempts to read from the port.
// On receiving data, looks for a defined command.
static const char *PORT_NAME = "/dev/ttyUSB1";
int main(){

    // Opens a USB virtual serial port at ttyUSB0
    serial_port_open();

    // Reset, Read ID, define latency, select a strategy
    serial_configure_core_write();

    // Enable CPRNG to generate random
    serial_port_write();

    // Capture response from the platform
    serial_port_read();

    // Resets the terminal and closes the serial port
    serial_port_close();

    return 0;
}
```

Listing B.5: Configure the internal register of FPGA

```
void serial_configure_core_write(){

    // Reset
    .......

    // Read CORE ID
    .......

    // Cycle configuration with ['\x00', '\x00', '\x00', '\x01']
    unsigned char write_buffer_2[9] = { 0x55, 0x11, 0x10, 0x41, 0x00, 0x00, 0x00, 0x01, 0xaa};
    write_buffer_0[sizeof(write_buffer_0)] = 0;
    write(serial_port, write_buffer_2, sizeof(write_buffer_2));
    usleep((sizeof(write_buffer_2) + 25) * 1000);
    usleep(500*1000);
    serial_port_read();
}
```

Listing B.6: Write operation to FPGA: Exemple

```
void serial_port_write(){
    .......

    char write_buffer[5] = {0x55,0x10,0x10,0x20,0xaa};
    len = sizeof(write_buffer);

    bytes_written = write(serial_port, write_buffer, sizeof(write_buffer));

}
```

Listing B.7: Read DATA and R&W address confirmation

```
void serial_port_read(){
    ........
    memset(&read_buffer[0], 0, MAX_COMMAND_LENGTH);
    int chars_read = read(serial_port, &read_buffer[0], MAX_COMMAND_LENGTH);
    usleep(500*1000);
    ........
    ........
    if (read_buffer[1] == 0x7f) {
        // Capture the address read send it from FPGA to confirm read operation from PRNG
        sprintf(readBuff, "0x%X%X", read_buffer[2], read_buffer[3]);
        read_addr = strtol(readBuff, NULL, 0);
        // Capture the DATA of PRNG send it from FPGA
        sprintf(dataBuff, "0x%X%X%X%X", read_buffer[4], read_buffer[5], read_buffer[6], read_buffer[7]);
        read_data = strtol(dataBuff, NULL, 0);
        // Confirm read operation
        printf ("READ_Complete. address 0x%02x  =0x%04x.\n", read_addr, read_data); Confirm read operation
```

```
17      }
18      else if (read_buffer[1] == 0x7e) {
19          // Capture the write address send it from FPGA to confirm write operation to FPGA register
20          sprintf(readBuff, "0x%X%X", read_buffer[2], read_buffer[3]);
21          read_addr = strtol(readBuff, NULL, 0);
22          // Confirm Write operation
23          printf ("WRITE_Complete. address 0x%02x.\n", read_addr);
24      }
25      .......
26  }
```

BIBLIOGRAPHY

[1] Apostol Vassilev and Timothy A. Hall. The importance of entropy to information security. *Computer*, 47(2):78–81, February 2014.

[2] S. Callegari, R. Rovatti, and G. Setti. Embeddable adc-based true random number generator for cryptographic applications exploiting nonlinear signal processing and chaos. *IEEE Transactions on Signal Processing*, 53(2):793–805, Feb 2005.

[3] X. Fang, B. Wetzel, J. M. Merolla, J. M. Dudley, L. Larger, C. Guyeux, and J. M. Bahi. Noise and chaos contributions in fast random bit sequence generated from broadband optoelectronic entropy sources. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(3):888–901, March 2014.

[4] Robert L. Devaney. *An Introduction to Chaotic Dynamical Systems, 2nd Edition*. Westview Pr., March 2003.

[5] T. Stojanovski and L. Kocarev. Chaos-based random number generators-part i: analysis [cryptography]. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 48(3):281–288, Mar 2001.

[6] T. Addabbo, A. Fort, L. Kocarev, S. Rocchi, and V. Vignoli. Pseudo-chaotic lossy compressors for true random number generation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(8):1897–1909, 2011.

[7] Y. Liu, R. C. C. Cheung, and H. Wong. A bias-bounded digital true random number generator architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(1):133–144, Jan 2017.

[8] Vidya Rajagopalan, V Boppana, S Dutta, B Taylor, and R Wittig. Xilinx zynq-7000 epp–an extensible processing platform family. In *23rd Hot Chips Symposium*, pages 1352–1357, 2011.

[9] Stephen Wiggins. *Introduction to applied nonlinear dynamical systems and chaos*, volume 2. Springer Science & Business Media, 2003.

[10] Xiaole Fang, Qianxue Wang, Christophe Guyeux, and Jacques M Bahi. Fpga acceleration of a pseudorandom number generator based on chaotic iterations. *Journal of Information Security and Applications*, 19(1):78–87, 2014.

[11] Jacques M Bahi, Xiaole Fang, Christophe Guyeux, and Laurent Larger. Fpga design for pseudorandom number generator based on chaotic iteration used in information hiding application. *Appl. Math*, 7(6):2175–2188, 2013.

[12] Bakiri Mohammed, Jean-Francois Couchot, and Christophe Guyeux. Fpga implementation of f2-linear pseudorandom number generators based on zynq mpsoc:

A chaotic iterations post processing case study. In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications - Volume 4: SECRYPT,*, pages 302–309, 2016.

[13] Mohammed Bakiri, Jean-François Couchot, and Christophe Guyeux. One random jump and one permutation: Sufficient conditions to chaotic, statistically faultless, and large throughput prng for fpga. In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications - Volume 6: SECRYPT, (ICETE 2017)*, pages 295–302. INSTICC, SciTePress, 2017.

[14] M. Bakiri, J. F. Couchot, and C. Guyeux. Ciprng: A vlsi family of chaotic iterations post-processings for f-2 linear pseudorandom number generation based on zynq mpsoc. *IEEE Transactions on Circuits and Systems I: Regular Papers*, PP(99):1–14, 2017.

[15] Jacques Bahi, Christophe Guyeux, and Qianxue Wang. A novel pseudo-random generator based on discrete chaotic iterations. In *INTERNET'09, 1-st Int. Conf. on Evolving Internet*, pages 71–76, Cannes, France, August 2009.

[16] Sylvain Contassot-Vivier, Jean-François Couchot, Christophe Guyeux, and Pierre-Cyrille Heam. Random walk in a n-cube without hamiltonian cycle to chaotic pseudorandom number generation: Theoretical and practical considerations. *International Journal of Bifurcation and Chaos*, 27(01):1750014, 2017.

[17] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[18] James E Gentle. *Random number generation and Monte Carlo methods*. Springer Science & Business Media, 2003.

[19] Carl-Erik Froberg and Carl Erik Frhoberg. *Introduction to numerical analysis*. Addison-Wesley Reading, Massachusetts, 1969.

[20] Michael George Luby. *Pseudorandomness and cryptographic applications*. Princeton University Press, 1996.

[21] Shimon Even and Yishay Mansour. A construction of a cipher from a single pseudorandom permutation. *Journal of Cryptology*, 10(3):151–161, 1997.

[22] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, pages 10–18. Springer, 1985.

[23] Michael Henson and Stephen Taylor. Memory encryption: a survey of existing techniques. *ACM Computing Surveys (CSUR)*, 46(4):53, 2014.

[24] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International Palo Alto, 1979.

[25] C.E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, The*, 28(4):656–715, Oct 1949.

[26] L.H.C. Tippett. *Random Sampling Numbers. Arranged by L.H.C. Tippett, Etc*. [Tracts for Computers. no. 15.]. 1927.

[27] Martin Campbell-Kelly, Mary Croarken, Raymond Flood, and Eleanor Robson. The history of mathematical tables. *AMC*, 10:12, 2005.

[28] Maurice G Kendall and B Babington Smith. Randomness and random sampling numbers. *Journal of the royal Statistical Society*, pages 147–166, 1938.

[29] Simon Hugh Lavington. *A history of Manchester computers*. NCC Publications, 1975.

[30] George W Brown. History of rand's random digits, summary. Technical report, DTIC Document, 1949.

[31] WE Thomson. Ernie–a mathematical and statistical analysis. *Journal of the Royal Statistical Society. Series A (General)*, pages 301–333, 1959.

[32] Nicholas Metropolis. The beginning of the monte carlo method. *Los Alamos Science*, 15(584):125–130, 1987.

[33] Harald Niederreiter and NSF-CBMS Regional Conference on Random Number Generation. *Random number generation and quasi-Monte Carlo methods*, volume 63. SIAM, 1992.

[34] Pierre L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53(1):77–120, 1994.

[35] Curtis D Motchenbacher and Joseph Alvin Connelly. *Low-noise electronic system design*. Wiley New York, 1993.

[36] Lindsay Kleeman and Antonio Cantoni. Metastable behavior in digital systems. *Design & Test of Computers, IEEE*, 4(6):4–19, 1987.

[37] Guan-Chyun Hsieh and James C Hung. Phase-locked loop techniques. a survey. *Industrial Electronics, IEEE Transactions on*, 43(6):609–615, 1996.

[38] A. Liacha, A. K. Oudjida, F. Ferguene, M. Bakiri, and M. L. Berrandjia. Design of high-speed, low-power, and area-efficient fir filters. *IET Circuits, Devices Systems*, 12(1):1–11, 2018.

[39] A. K. Oudjida, D. Benamrouche, and M. Liem. Front-end ip development: Basic know-how. In *2007 International Conference on Design Technology of Integrated Systems in Nanoscale Era*, pages 60–63, Sept 2007.

[40] Ross H Freeman and Hung-Cheng Hsieh. Distributed memory architecture for a configurable logic array and method for using distributed memory, August 30 1994. US Patent 5,343,406.

[41] Philip M Freidin. Logic block with look-up table for configuration and memory, May 9 1995. US Patent 5,414,377.

[42] E. Barker and A. Roginsky. Draft NIST special publication 800-131 recommendation for the transitioning of cryptographic algorithms and key sizes, 2010.

[43] G Marsaglia. The diehard test suite, 1995. *URL http://stat. fsu. edu/˜ geo/diehard. html*, 1995.

[44] Pierre L'Ecuyer and Richard Simard. Testu01: Ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):22, 2007.

[45] P. L'Ecuyer and F. Panneton. Fast random number generators based on linear recurrences modulo 2: overview and comparison. In *Proceedings of the Winter Simulation Conference, 2005.*, pages 10 pp.–, Dec 2005.

[46] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[47] Robert C Tausworthe. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19(90):201–209, 1965.

[48] Donald E Knuth. Deciphering a linear congruential encryption. *IEEE Transactions on Information Theory*, 31(1):49–52, 1985.

[49] Theodore G Lewis and William H Payne. Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM (JACM)*, 20(3):456–468, 1973.

[50] Makoto Matsumoto and Yoshiharu Kurita. Twisted gfsr generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 2(3):179–194, 1992.

[51] George S Fishman and Louis R Moore, III. An exhaustive analysis of multiplicative congruential random number generators with modulus 2ˆ31-1. *SIAM Journal on Scientific and Statistical Computing*, 7(1):24–45, 1986.

[52] Simon Banks, Philip Beadling, and Andras Ferencz. Fpga implementation of pseudo random number generators for monte carlo methods in quantitative finance. In *Reconfigurable Computing and FPGAs, 2008. ReConFig'08. International Conference on*, pages 271–276. IEEE, 2008.

[53] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[54] George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[55] George Marsaglia and Arif Zaman. A new class of random number generators. *The Annals of Applied Probability*, pages 462–480, 1991.

[56] A.K. Oudjida and N. Chaillet. Radix-2^r arithmetic for multiplication by a constant. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 61(5):349–353, May 2014.

[57] A. K. Oudjida, A. Liacha, M. Bakiri, and N. Chaillet. Multiple constant multiplication algorithm for high-speed and low-power design. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(2):176–180, Feb 2016.

[58] A.K. Oudjida, N. Chaillet, and M.L. Berrandjia. Radix-2r arithmetic for multiplication by a constant: Further results and improvements. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 62(4):372–376, April 2015.

[59] Raj S Katti and Sudarshan K Srinivasan. Efficient hardware implementation of a new pseudo-random bit sequence generator. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 1393–1396. IEEE, 2009.

[60] E. Erkek and T. Tuncer. The implementation of asg and sg random number generators. In *System Science and Engineering (ICSSE), 2013 International Conference on*, pages 363–367, July 2013.

[61] Victor R Gonzalez-Diaz, Fabio Pareschi, Gianluca Setti, and Franco Maloberti. A pseudorandom number generator based on time-variant recursion of accumulators. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 58(9):580–584, 2011.

[62] Vladimir Friedman. The structure of the limit cycles in sigma delta modulation. *Communications, IEEE Transactions on*, 36(8):972–979, 1988.

[63] Franco Maloberti, Edoardo Bonizzoni, and Antonio Surano. Time variant digital sigma-delta modulator for fractional-n frequency synthesizers. In *Radio-Frequency Integration Technology, 2009. RFIT 2009. IEEE International Symposium on*, pages 111–114. IEEE, 2009.

[64] David Barrie Thomas and Wayne Luk. Fpga-optimised high-quality uniform random number generators. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 235–244. ACM, 2008.

[65] David B Thomas and Wayne Luk. Fpga-optimised uniform random number generators using luts and shift registers. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 77–82. IEEE, 2010.

[66] David B Thomas and Wayne Luk. The lut-sr family of uniform random number generators for fpga architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(4):761–770, 2013.

[67] Shrutisagar Chandrasekaran and Abbes Amira. High performance fpga implementation of the mersenne twister. In *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, pages 482–485. IEEE, 2008.

[68] Xiang Tian and Khaled Benkrid. Mersenne twister random number generation on fpga, cpu and gpu. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 460–464. IEEE, 2009.

[69] Ishaan L Dalal, Jared Harwayne-Gidansky, and Deian Stefan. On the fast generation of long-period pseudorandom number sequences. In *Systems, Applications and Technology Conference, 2008 IEEE Long Island*, pages 1–9. IEEE, 2008.

[70] Mutsuo Saito and Makoto Matsumoto. Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer, 2008.

[71] Yuan Li, Jiang Jiang, Hanqiang Cheng, Minxuan Zhang, and Shaojun Wei. An efficient hardware random number generator based on the mt method. In *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on*, pages 1011–1015. IEEE, 2012.

[72] Shengfei Wu, Jiang Jiang, and Yuzhuo Fu. Hardware architecture for the parallel generation of long-period random numbers using mt method. In *Computer Engineering and Technology*, pages 8–15. Springer, 2013.

[73] Pedro Echeverría and Marisa López-Vallejo. High performance fpga-oriented mersenne twister uniform random number generator. *Journal of Signal Processing Systems*, 71(2):105–109, 2013.

[74] John Von Neumann, Arthur W Burks, et al. Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1966.

[75] James Gleick. *Chaos: Making a new science*. Random House, 1997.

[76] Petre Anghelescu, Emil Sofron, and Silviu Ionita. Vlsi implementation of high-speed cellular automata encryption algorithm. In *Semiconductor Conference, 2007. CAS 2007. International*, volume 2, pages 509–512. IEEE, 2007.

[77] Ioana Dogaru and Radu Dogaru. Algebraic normal form for rapid prototyping of elementary hybrid cellular automata in fpga. In *Electrical and Electronics Engineering (ISEEE), 2010 3rd International Symposium on*, pages 277–280. IEEE, 2010.

[78] Dogaru Ioana and Dogaru Radu. Fpga implementation and evaluation of two cryptographically secure hybrid cellular automata. In *Communications (COMM), 2014 10th International Conference on*, pages 1–4. IEEE, 2014.

[79] Thomas E Tkacik. A hardware random number generator. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 450–453. Springer, 2003.

[80] Juan C Cerda, Chris D Martinez, Jonathan M Comer, and David HK Hoe. An efficient fpga random number generator using lfsrs and cellular automata. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 912–915. IEEE, 2012.

[81] Sheng-Uei Guan and Syn Kiat Tan. Pseudorandom number generator–the self programmable cellular automata. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 1230–1235. Springer, 2003.

[82] Jonathan M Comer, Juan C Cerda, Chris D Martinez, and David HK Hoe. Random number generators using cellular automata implemented on fpgas. In *System Theory (SSST), 2012 44th Southeastern Symposium on*, pages 67–72. IEEE, 2012.

[83] Lakshman Raut and David HK Hoe. Stream cipher design using cellular automata implemented on fpgas. In *System Theory (SSST), 2013 45th Southeastern Symposium on*, pages 146–149. IEEE, 2013.

[84] Mathieu David, Damith C Ranasinghe, and Torben Larsen. A2u2: a stream cipher for printed electronics rfid tags. In *RFID (RFID), 2011 IEEE International Conference on*, pages 176–183. IEEE, 2011.

[85] Leonidas Kotoulas, Demetrios Tsarouchis, Georgios Ch Sirakoulis, and Ioannis Andreadis. 1-d cellular automaton for pseudorandom number generation and its reconfigurable hardware implementation. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–pp. IEEE, 2006.

[86] Louis M Pecora and Thomas L Carroll. Synchronization in chaotic systems. *Physical review letters*, 64(8):821, 1990.

[87] Robert M May et al. Simple mathematical models with very complicated dynamics. *Nature*, 261(5560):459–467, 1976.

[88] Michel Hénon. A two-dimensional mapping with a strange attractor. *Communications in Mathematical Physics*, 50(1):69–77, 1976.

[89] Pawel Dabal and Ryszard Pelka. A chaos-based pseudo-random bit generator implemented in fpga device. In *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 151–154. IEEE, 2011.

[90] W.T. Padgett and D.V. Anderson. *Fixed-Point Signal Processing*. Synthesis lectures on signal processing. Morgan & Claypool, 2009.

[91] Pawel Dabal and Ryszard Pelka. Fpga implementation of chaotic pseudo-random bit generators. In *Mixed Design of Integrated Circuits and Systems (MIXDES), 2012 Proceedings of the 19th International Conference*, pages 260–264. IEEE, 2012.

[92] P. Dabal and R. Pelka. A study on fast pipelined pseudo-random number generator based on chaotic logistic map. In *17th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, pages 195–200, April 2014.

[93] Amit Pande and Joseph Zambreno. Design and hardware implementation of a chaotic encryption scheme for real-time embedded systems. In *Signal Processing and Communications (SPCOM), 2010 International Conference on*, pages 1–5. IEEE, 2010.

[94] Shubo Liu, Jing Sun, Zhengquan Xu, and Zhaohui Cai. An improved chaos-based stream cipher algorithm and its vlsi implementation. In *Networked Computing and Advanced Information Management, 2008. NCM'08. Fourth International Conference on*, volume 2, pages 191–197. IEEE, 2008.

[95] Lahcene Merah, Adda ALI-PACHA, and Naima HADJ SAID. Coupling two chaotic systems in order to increasing the security of a communication system-study and real time fpga implementation.

[96] Pascal Giard, Georges Kaddoum, François Gagnon, and Claude Thibeault. Fpga implementation and evaluation of discrete-time chaotic generators circuits. In *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*, pages 3221–3224. IEEE, 2012.

[97] T Geisel and V Fairen. Statistical properties of chaos in chebyshev maps. *Physics Letters A*, 105(6):263–266, 1984.

[98] Yaobin Mao, Liu Cao, and Wenbo Liu. Design and fpga implementation of a pseudo-random bit sequence generator using spatiotemporal chaos. In *Communications, Circuits and Systems Proceedings, 2006 International Conference on*, volume 3, pages 2114–2118. IEEE, 2006.

[99] J Černák. Digital generators of chaos. *Physics letters A*, 214(3):151–160, 1996.

[100] Chung-Yi Li, Jiung-Sheng Chen, and Tsin-Yuan Chang. A chaos-based pseudo random number generator using timing-based reseeding method. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–pp. IEEE, 2006.

[101] R.G. B. Simultaneous carry adder, December 27 1960. US Patent 2,966,305.

[102] Chung-Yi Li, Yuan-Ho Chen, Tsin-Yuan Chang, Lih-Yuan Deng, and Kiwing To. Period extension and randomness enhancement using high-throughput reseeding-mixing prng. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(2):385–389, 2012.

[103] Lih-Yuan Deng. Efficient and portable multiple recursive generators of large order. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(1):1–13, 2005.

[104] Takeshi Oshiba. Closure property of family of context-free languages under cyclic shift operation. *ELECTRONICS & COMMUNICATIONS IN JAPAN*, 55(4):119–122, 1972.

[105] John J Shedletsky. Comment on the sequential and indeterminate behavior of an end-around-carry adder. *IEEE Transactions on Computers*, 26(3):271–272, 1977.

[106] NagaDeepa Hariprasad et al. Fpga implementation of a cryptography technology using pseudo random number generator. In *International Journal of Engineering Research and Technology*, volume 2. ESRSA Publications, November 2013.

[107] O.E. Rössler. An equation for continuous chaos. *Physics Letters A*, 57(5):397 – 398, 1976.

[108] René Thomas, Vasileios Basios, Markus Eiswirth, Thomas Kruel, and Otto E Rössler. Hyperchaos of arbitrary order generated by a single feedback circuit, and the emergence of chaotic walks. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 14(3):669–674, 2004.

[109] Hu Guo-Si. A hyperchaotic attractor with multiple positive lyapunov exponents. *Chinese Physics Letters*, 26(12):120501, 2009.

[110] Ahmed S Elwakil and Michael Peter Kennedy. Construction of classes of circuit-independent chaotic oscillators using passive-only nonlinear devices. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 48(3):289–307, 2001.

[111] AS Elwakil and MP Kennedy. Chaotic oscillator configuration using a frequency dependent negative resistor. *International journal of circuit theory and applications*, 28(1):69–76, 2000.

[112] M Affan Zidan, Ahmed Gomaa Radwan, and Khaled Nabil Salama. The effect of numerical techniques on differential equation based chaotic generators. In *Microelectronics (ICM), 2011 International Conference on*, pages 1–4. IEEE, 2011.

[113] G Chen and J Lü. Dynamics of the lorenz system family: analysis, control and synchronization. *SciencePress, Beijing*, 2003.

[114] PY Tsai, CL Merkle, and TT Huang. Euler equation analysis of the propeller-wake interaction. In *Symposium on Naval Hydrodynamics, 17th*, 1900.

[115] Preston C Hammer. The midpoint method of numerical integration. *Mathematics Magazine*, 31(4):193–195, 1958.

[116] John C Butcher. Numerical methods for ordinary differential equations in the 20th century. *Journal of Computational and Applied Mathematics*, 125(1):1–29, 2000.

[117] M Affan Zidan, Ahmed Gomaa Radwan, and Khaled Nabil Salama. Random number generation based on digital differential chaos. In *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pages 1–4. IEEE, 2011.

[118] E.J. G. Latched carry save adder circuit for multipliers, September 5 1967. US Patent 3,340,388.

[119] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, Feb 1964.

[120] Abhinav S Mansingka, Mohamed L Barakat, M Affan Zidan, Ahmed G Radwan, and Khaled N Salama. Fibonacci-based hardware post-processing for non-autonomous signum hyperchaotic system. In *IT Convergence and Security (ICITCS), 2013 International Conference on*, pages 1–4. IEEE, 2013.

[121] Basab Bijoy Purkayastha and Kandarpa Kumar Sarma. Digital phase-locked loop. In *A Digital Phase Locked Loop based Signal and Symbol Recovery System for Wireless Channel*, pages 103–126. Springer, 2015.

[122] Viktor Fischer and Miloš Drutarovskỳ. True random number generator embedded in reconfigurable hardware. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 415–430. Springer, 2003.

[123] Martin Šimka, Miloš Drutarovskỳ, and Viktor Fischer. Embedded true random number generator in actel fpgas. In *Workshop on Cryptographic Advances in Secure Hardware–CRASH*, pages 6–7, 2005.

[124] Martin Simka, Milos Drutarovskỳ, Viktor Fischer, et al. Testing of pll-based true random number generator in changingworking conditions. *RADIOENGINEERING,*, 20:94–101, 2011.

[125] Michal Varchola, Milos Drutarovsky, Robert Fouquet, and Viktor Fischer. Hardware platform for testing performance of trngs embedded in actel fusion fpga. In *Radioelektronika, 2008 18th International Conference*, pages 1–4. IEEE, 2008.

[126] Paul Kohlbrenner and Kris Gaj. An embedded true random number generator for fpgas. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 71–78. ACM, 2004.

[127] Cristian Klein, Octavian Cret, and Alin Suciu. Design and implementation of a high quality and high throughput trng in fpga. *arXiv preprint arXiv:0906.4762*, 2009.

[128] Markus Dichtl and Jovan Dj Golić. *High-speed true random number generation with logic gates only*. Springer, 2007.

[129] Abdelkarim Cherkaoui, Viktor Fischer, Alain Aubert, and Laurent Fesquet. A self-timed ring based true random number generator. In *Asynchronous Circuits and Systems (ASYNC), 2013 IEEE 19th International Symposium on*, pages 99–106. IEEE, 2013.

[130] Abdelkarim Cherkaoui, Viktor Fischer, Alain Aubert, and Laurent Fesquet. Comparison of self-timed ring and inverter ring oscillators as entropy sources in fpgas. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 1325–1330. IEEE, 2012.

[131] Abdelkarim Cherkaoui, Viktor Fischer, Laurent Fesquet, and Alain Aubert. A very high speed true random number generator with entropy assessment. In *Cryptographic Hardware and Embedded Systems-CHES 2013*, pages 179–196. Springer, 2013.

[132] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[133] Ihor Vasyltsov, Eduard Hambardzumyan, Young-Sik Kim, and Bohdan Karpinskyy. Fast digital trng based on metastable ring oscillator. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 164–180. Springer, 2008.

[134] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. Fpga-based true random number generation using circuit metastability with adaptive feedback control. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 17–32. Springer, 2011.

[135] Emre Salman, Ali Dasdan, Feroze Taraporevala, Kayhan Kucukcakar, and Eby G Friedman. Exploiting setup–hold-time interdependence in static timing analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(6):1114–1125, 2007.

[136] Donggeon Lee, Hwajeong Seo, and Howon Kim. Metastability-based feedback method for enhancing fpga-based trng. *International Journal of Multimedia & Ubiquitous Engineering*, 9(3), 2014.

[137] NIST. National institute of standards and technology (nist): Fips$140-1$: Security requirements for cryptographic modules @ONLINE, January 1994.

[138] NIST. National institute of standards and technology (nist): A statistical test suite for random and pseudorandom number generators for cryptographic applications @ONLINE, 2010.

[139] Wolfgang Killmann and Werner Schindler. A proposal for: Functionality classes and evaluation methodology for true (physical) random number generators. *T-Systems debis Systemhaus Information Security Services and Bundesamt für Sicherheit in der Informationstechnik (BSI), Tech. Rep*, 2001.

[140] NIST. National institute of standards and technology (nist): Fips$140-2$: Security requirements for cryptographic modules @ONLINE, May 2001.

[141] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[142] Anne Canteaut. Berlekamp–massey algorithm. In *Encyclopedia of Cryptography and Security*, pages 80–80. Springer, 2011.

[143] I Zarei Moghadam, Ali Shokouhi Rostami, and Mohammad Rasoul Tanhatalab. Designing a random number generator with novel parallel lfsr substructure for key stream ciphers. In *Computer Design and Applications (ICCDA), 2010 International Conference on*, volume 5, pages V5–598. IEEE, 2010.

[144] D. B. Thomas and W. Luk. Fpga-optimised uniform random number generators using luts and shift registers. In *2010 International Conference on Field Programmable Logic and Applications*, pages 77–82, Aug 2010.

[145] Kuen Hung Tsoi, KH Leung, and Philip Heng Wai Leong. Compact fpga-based true and pseudo random number generators. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 51–61. IEEE, 2003.

[146] Chung-Yi Li, Jiung-Sheng Chen, and Tsin-Yuan Chang. A chaos-based pseudo random number generator using timing-based reseeding method. In *2006 IEEE International Symposium on Circuits and Systems*, pages 4 pp.–3280, May 2006.

[147] C. Y. Li, Y. H. Chen, T. Y. Chang, L. Y. Deng, and K. To. Period extension and randomness enhancement using high-throughput reseeding-mixing prng. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(2):385–389, Feb 2012.

[148] Ziqi Zhu and Hanping Hu. A dynamic nonlinear transform arithmetic for improving the properties chaos-based prng. In *Intelligent Control and Automation (WCICA), 2010 8th World Congress on*, pages 7055–7060, July 2010.

[149] D. B. Thomas and W. Luk. The lut-sr family of uniform random number generators for fpga architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(4):761–770, April 2013.

[150] Vanderlei Bonato, Bruno F Mazzotti, Marcio Merino Fernandes, and Eduardo Marques. A mersenne twister hardware implementation for the monte carlo localization algorithm. *Journal of Signal Processing Systems*, 70(1):75–85, 2013.

[151] P. Dabal and R. Pelka. Fpga implementation of chaotic pseudo-random bit generators. In *Proceedings of the 19th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES 2012*, pages 260–264, May 2012.

[152] Jean-François Couchot. *Modèles discrets pour la sécurité informatique: des méthodes itératives à l'analyse vectorielle*. Hdr, Université de Bourgogne Franche-Comté, january 2017.

[153] Christophe Guyeux. *Le désordre des itérations chaotiques et leur utilité en sécurité informatique*. Theses, Université de Franche-Comté, December 2010.

[154] T. Y. Li and J. A. Yorke. Period three implies chaos. *Amer. Math. Monthly*, 82(10):985–992, 1975.

[155] Sylvain Contassot-Vivier, Jean-Francois Couchot, Christophe Guyeux, and Pierre-Cyrille Heam. Random walk in a n-cube without hamiltonian cycle to chaotic pseudorandom number generation: Theoretical and practical considerations. *International Journal of Bifurcation and Chaos*, *:*, 2016.

[156] Jacques Bahi, Raphaël Couturier, Christophe Guyeux, and Pierre-Cyrille Héam. Efficient and cryptographically secure generation of chaotic pseudorandom numbers on gpu. *The journal of Supercomputing*, 71(10):3877–3903, oct 2015.

[157] Knudsen. Chaos without nonperiodicity. *Amer. Math. Monthly*, 101, 1994.

[158] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, 2011.

[159] Rainer A Rueppel. Linear complexity and random sequences. In *Advances in Cryptology—EUROCRYPT'85*, pages 167–188. Springer, 1985.

[160] Uwe Meyer-Baese and U Meyer-Baese. *Digital signal processing with field programmable gate arrays*, volume 65. Springer, 2007.

[161] M Anwar Hasan and Christophe Negre. Sequential multiplier with sub-linear gate complexity. *Journal of Cryptographic Engineering*, 2(2):91–97, 2012.

[162] Jacques M Bahi, Jean-François Couchot, Christophe Guyeux, and Qianxue Wang. Class of trustworthy pseudo-random number generators. *arXiv preprint arXiv:1112.0950*, 2011.

[163] Christophe Guyeux and Jacques Bahi. An improved watermarking algorithm for internet applications. In *INTERNET'2010. The 2nd Int. Conf. on Evolving Internet*, pages 119 – 124, Valencia, Spain, sep 2010.

[164] M. A. Zidan, A. G. Radwan, and K. N. Salama. The effect of numerical techniques on differential equation based chaotic generators. In *ICM 2011 Proceeding*, pages 1–4, Dec 2011.

[165] P. Giard, G. Kaddoum, F. Gagnon, and C. Thibeault. Fpga implementation and evaluation of discrete-time chaotic generators circuits. In *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pages 3221–3224, Oct 2012.

[166] J-F Couchot, P-C Heam, Christophe Guyeux, Qianxue Wang, and Jacques M Bahi. Pseudorandom number generators with balanced gray codes. In *Security and Cryptography (SECRYPT), 2014 11th International Conference on*, pages 1–7. IEEE, 2014.

[167] MELISSA E O'Neill. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Trans. Math. Softw.(submitted)*, pages 1–46, 1988.

List of Figures

List of Tables

Document generated with LaTeX and:
the LaTeX style for PhD Thesis created by S. Galland — http://www.multiagent.fr/ThesisStyle
the tex-upmethodology package suite — http://www.arakhne.org/tex-upmethodology/

Abstract:

In this thesis, we designed pseudo-random number generators (PRNGs) based on chaotic iterations to be deployed on hardware support such as FPGA or ASIC. These generators can be seen as post-processing of existing generators and thus transform a sequence of numbers, the input, into another, the output. The dependency between these two sequences has been proven chaotic according to Devaney: the effects of one bit change in the input cannot be predicted in the long term on output. Through the hardware implementations, we have been able to provide compact, very high speed, secure and reconfigurable PRNGs.
A state of the art of the hardware implementations of PRNG's was first carried out. All of them have been compared, after being fully implemented in FPGA, in a complete platform that we created. This last one allowed to compare the different hardware PRNGs, and in particular to carry out statistical tests on the outputs. New generators based on chaotic iterations (CI) were then designed and integrated into this platform. The embedded iterated function is built by removing an Hamiltonian cycle from an N-cube, the whole being followed by a permutation. Resulting generators generally have a better statistical profile than embedded ones, while running at a similar speed. Among the PRNGs able to pass the most difficult battery of statistical tests (TESTU01), those ones are the fastest in the world and the only ones to be chaotic. We have finally implemented them on numerous hardware supports: 65-nm ASIC circuit and FPGA Zynq.

Keywords: Random number generators, Chaotic circuits, Discrete dynamical systems, Statistical tests, Cryptography hardware and implementation, Applied cryptography, FPGA, ASIC

Résumé :

Dans cette thèse, nous avons conçu des générateurs de nombres pseudo-aléatoires (PRNGs) basés sur des itérations chaotiques devant être déployés sur des supports matériels comme FPGA ou ASIC. Ces générateurs peuvent être vus comme des post-traitements de générateurs existants et transforment donc une suite de nombres, l'entrée, en une autre, la sortie. La dépendance entre ces deux suites a été prouvée chaotique selon Devaney : les effets d'un moindre changement sur l'entrée ne peuvent être prédits à long terme sur la sortie. Au travers des implantations matérielles, nous avons pu fournir des PRNGs compacts, à très haut débit, sécurisés et reconfigurables.
Un état de l'art des implantations matérielles des PRNGs a été tout d'abord effectué. Celles-ci ont toutes été comparées, après avoir été intégralement programmées en FPGA, dans une plate-forme complète que nous avons créée. Cette dernière a permis de comparer les différents PRNGs matériels, et notamment d'effectuer des tests statistiques sur les sorties. De nouveaux générateurs à base d'itérations chaotiques (IC) ont ensuite été conçus et intégrés à la plate-forme matérielle. La fonction itérée est construite en supprimant un cycle hamiltonien d'un N-cube, l'ensemble subissant ensuite une permutation. Les générateurs obtenus ont généralement un meilleur profil statistique que ceux embarqués, tout en s'exécutant à une vitesse similaire. Parmi les PRNGs capables de passer la batterie de tests statistiques la plus difficile (TESTU01), ceux proposés sont les plus rapides au monde et les seuls à être chaotiques. Nous les avons finalement implantés sur de nombreux supports matériels: 65-nm circuit ASIC et FPGA Zynq.

Mots-clés : Générateur des nombre aléatoire, Circuit Chaotique, Systèmes dynamiques discrets, Test Statistiques, Cryptographie matérielle et implantation, Cryptographie appliquée, FPGA, ASIC

Ecole Doctoral SPIM, Université Bourgougne Franch-Comté
32, Avenue de l'observatoire, 25000, Besançon Cedex
ed-spim@univ-fcomte.fr, www.ed-spim.univ-fcomte.fr, secretariat@ubfc.fr

UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ