# Contents

# Introduction

The **privacy** is a fundamental right of each person and it's put to big test in this technological era, thus it is important to understand how to protect our information from being (ab)used or altered.

Random Number Generators (RNGs), are essential components for every *cryptographic* system and application [2]. They can be used for the generation of random session keys, signature keys and signature parameters, challenges and zero knowledge proofs and nonces. Wrong and poor implementation of these modules is a risk that cannot be accepted when we spoke about *security* and *integrity* of precious information. In addition, RNGs can be used, not only for cryptographic application but also for *stochastic simulation*, such as Monte Carlo methods, used to solve problems heuristically, because their intrinsic complexity does not permit to solve them analytically. Consider as an example, the optimal distribution of service times in complex multi-user system or the study of the correlated and uncorrelated variation in analog and digital ICs.

An ideal RNG is capable of generate *ideal secrets*, namely aperiodic, non deterministic bit sequence, thus the random bit produced are independent and uniformly distributed over a certain range. A perfect RNG, however, is a *fiction* [28]. In this work I will focus my attention on the best feasible approximation of an ideal RNG, defined as **True Random Number Generator (TRNG).** This module is the basic primitive to build of every crypto-system. It is mandatory also in the Pseudo Random (Algorithmic) world. Their realization exploits specific *non-deterministic* physical phenomena from various branch of physics.

To face this challenging problems I will show a strong mathematical model, from which I will deduce a collection of statistical tools, like Fourier transformation, Entropy and Hamming distance. Without these instruments it is not possible to analyze a RNG and quantify how much randomness there is in a given bit sequence. Remember that randomness is a *relative* subject.

I will explore "ad hoc" methodologies for design RNG: analog and digital solutions. In the specific context of cryptography, one can be faced with a malicious person, defined as an **attacker** who does not only has knowledge of the RNG's design and the possibility to analyze its outputs, but also he can try to affect or control the RNG. So, for a designer it is not sufficient to have a theoretical abstraction of a RNG in mind, but also its concrete realization and its possible flaws in the design process, e.g. side-channel attack immunity, tamper resistance.

In the past, a lot of research work has been devoted to the development of good physical random sources and a variety of design have been proposed. Less work has been spent in the development of suitable *tests* and assessment criteria. Thus I

will explore this filed and present the theoretical foundations, defined as Statistical Hypothesis Testing (SHT) and I will introduce some test suite: NIST SP800-22, DIEHARD, dieharder.

Finally I will propose the core of this work: the design and the implementation of a CMOS on-chip test module. This circuit is able to perform some tests using only some combinatorial logic, counters and comparators (full hardware implementation) and return some results to check on-line the health of the target RNG.

# 1 Random Number Generators Foundations

## 1.1 Random Number Generators Classification

In cryptographic/security applications the random number generation mechanism falls into two classes:

1. **True Random Number Generators (TRNG)** OR **Physical Random Number Generators (PhRNG)**: exploit Non-Deterministic events and return aperiodic sequences;

2. **Pseudo Random Number Generators (PRNG)** OR **Deterministic Random Bit Generator (DRBG)**: exploit Deterministic events and return periodic sequences;

The first class of RNG is formed by True or Physical RNG. True Random Number Generators harness random physical phenomena to generate random bits. On-chip noise, clock jitter and stray electromagnetic field are some of the sources of randomness for a TRNG. TRNG circuits are often used to periodically seed a PRNG, generate nonces for security protocols and in data encryption/decryption. The most commonly used TRNG circuits are based on Ring Oscillators which sample and digitize on-chip jitter noise to generate random bits. Ring Oscillator based circuits are also popular for FPGA based implementations. Noise amplifiers and Analog-to-Digital Converters (ADC) are used to sample on-chip thermal and shot noise to obtain random samples. Digital circuits for sampling thermal noise use metastable elements like a pair of cross coupled inverters. Power up state of SRAM, read-refresh collision in DRAM and Random Telegraph Noise (RTN) in Contact Resistive RAM [12] are examples of memory-cell based TRNG circuits.

Another example of first class random number generation, not detailed in this work, is what is called, not strictly physical, hence true RNG, and it is based on the intersection of casual chains, remember that the combination of events of several process which are independent of each other may behave completely randomly. Consider, as an example, computer data as time of interrupts, hard-disk seek times or user interactions, like mouse movement or keystroke timings. The intersection of these data can provide a source of randomness. Modern UNIX and Windows Operating System s have OS-level RNGs based on the timing of kernel IO events. Unfortunately, the quality of the entropy collected depends upon the system's configuration and hardware. For example the entropy available from an embedded

device without hard drive or keyboard, such as a smartphone, may be insufficient for security application, so it is better to rely on, first class, True RNG.

The second class is formed by *Pseudo* Random Number Generator or *Deterministic* Random Bit Generator, denoted in this work as **PRNG** and **DRBG**.[1] A PRNG uses deterministic processes, in the specific finite memory algorithms, to generate a *periodic* series of output from an initial *seed* and a *current state*. Because the output is purely a function (e.g., bijection) of the seed data, the unpredictability of the output can never exceed the unpredictability of the seed. Given the same seed, a PRNG will always output the same sequence of values so the possibility of retrieving information about the seed through the observation of output sub-sequences and their repeatable behavior can hardly be desirable in applications such as data security and cryptography. It can, however, be computationally infeasible to distinguish a well-seeded DRBG from an ideal RNG because often it is possible to extend the period with regard to the time scales on which a piece of equipment is employed. The substantial advantage of this class with respect to the first is the *algorithmic* nature, which makes them easily embeddable in any digital circuit or system, instead TRNG often requires dedicated hardware that is difficult to embed. It is important to focus the attention on the seed that by definition requires true random numbers, so even in this case a TRNG unit is required somewhere and it is fundamental for the system.

Chronologically speaking the first two class of RNGs represented the starting point of the design methodologies and so in a certain sense a "classic" scheme. But in the recent years novel methodologies merges the two classes in what can be defined as an **hybrid** RNG. This architecture use both components taking advantage of the best parts of each, e.g., super-secure true random number for the seed, fast and embeddable algorithm for hashing the outputs. In fact re-seeding a DRBG very frequently and with fresh true random number can lead to very good performance, an example will be showed in sec. 3.3. This novel design approach can be classified with the term **Cryptographically Secure Pseudo Random Number Generator (CSPRNG)** and by definition is a RNG capable to pass all statistical tests that run in *polynomial* time asymptotically. All such polynomial time statistical tests, called also *efficient* statistical tests, will be unable to distinguish a CSPRNG from a true random source.

It is important, for the scope of this work, to define a reference RNG that is defined as an **ideal** RNG. This mathematical construction is capable to generate infinitely long sequence formed by random numbers that are *uniformly (identically) distributed* in their range, for digital circuit the interval is simply$[0, 1]$, and *independent* from each other so it can produce unguessable, uncorrelated random output (for the statistics behind that see sec. 2.5). Starting from this model it is possible to analyze which kind of feasible architecture, during normal operation, cannot be distinguished

[1] Notice that, in this case the difference between the word *number* and the word *bit* does not exist because all the applications concern with random numbers generated from digital circuits outputs, that are bits.

from a perfect RNG. This is the fundamental sentence! TRNGs will be the core of this work, because they represent the *best* feasible approximation of an ideal RNG and a fundamental *primitive* for all security-related application.

## 1.2 TRNG General Scheme

The most synthetic, high level block scheme of a True Random Number Generator is in Fig. 1.1.

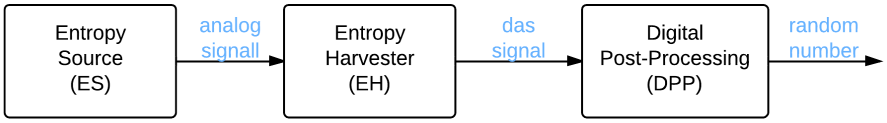



**Figure 1.1:** Basic high-level block scheme of TRNG.

This is a comfortable abstraction used recursively in this work.

### 1.2.1 Where is Randomness? Entropy Source (ES)

Design and Analysis of an Random Number Generator is based on the *understanding* of randomness [28]. Firstly it is needed to *find* randomness, common example of what is defined as an **Entropy Source (ES)** can be:

- Thermal (Johnson-Nyquist) noise: spontaneous voltage fluctuations in a (semi) conductor in thermodynamic equilibrium, that result from the thermal agitation of the electric charges (carriers) in the material of the (semi) conductor. [23, 15];
- Schottky (Shot) noise, which describes the randomness in a current as it begins to flows through a conductor e.g., from a Zener diode [29];
- Time between emissions of radioactive decays;
- Brownian motion [4];
- Quantum photon effect;

All these examples are physical random micro-cosmic process, defined by their nature, as **Non Deterministic**. The word *entropy* has a central meaning because quantify the amount of randomness present in a sequence, its formal definition will be presented in sec. 2.7. This quantity can be extracted in many ways depending on the type of ES, common examples are chaos on deterministic analog signals sec. 3.1,

power-up state of memory cells sec. 3.2, meta-stability of devices sec. 3.3 or random clock jitter samples.

It is possible to model these random sources as a time-continuous analog signal generators. The signals then can be *digitized* after uniform time intervals (e.g., using a comparator). These samples are defined as *digitized analog signal*, briefly denoted as *das random numbers* or as *raw bits*. Such generators are called True or Physical RNGs, denoted in this work as True Random Number Generators (TRNGs) or Physical Random Number Generators (PhRNGs) and they are capable to produce *aperiodic*, *uncorrelated* random outputs.

### 1.2.2 Entropy Harvester (EH)

The Entropy Harvesting stage is the part of the circuit devoted to the extraction of random bit. The first and most common example of EH circuit is the comparator, that returns some constant voltage value if the input fall in a certain thresholds interval. There are also examples in which the EH stage is embedded in the ES:

1. Chaos in a pipeline ADC circuit sec. 3.1;
2. Metastability in SRAM power-up state sec. 3.2;

### 1.2.3 Digital Post-Processing

Usually the das random number are algorithmically post-processed in order to reduce, more precisely *to mask* potential weakness of the ES e.g., *bias*. The outputs of this stage, denoted as Digital Post-Processing (DPP) stage, are called *internal random numbers*. Upon external call the same outputs are usually called *external random numbers.*

There are many ways to post-process a bit sequence, the common tradeoff in digital circuit design is between area/power consumption and strength of the algorithm. The DPP stage always increase the entropy of the system, reducing its throughput (bit-rate) [33]. I briefly present four solutions in ascending order of complexity.

#### 1.2.3.1 XOR Filter

The *XOR function* is a very common entropy extractor used also in sec. 3.1.

**Table 1.1:** XOR Function table of truth.

| **A** | **B** | **A$\dot{\vee}$B** |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Outputs of two or more TRNG circuits XORed (bit wise added) to improve the entropy of the output as shown in Fig. 1.2. Bias in one of the circuits is masked by the other TRNG circuits.

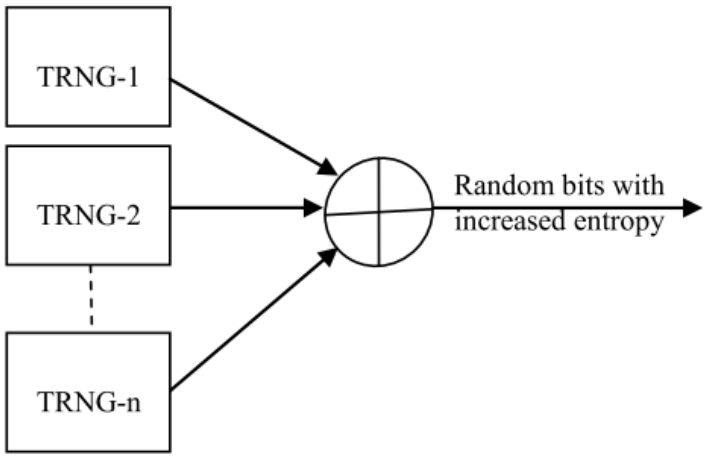



**Figure 1.2:** XOR Filter.

This technique also provides tolerance against device wear out or side channel attacks. Although the XOR function provides a simple implementation for improving the entropy of the design, it leads to overhead due to need for multiple TRNG circuits.

#### 1.2.3.2 von Neumann corrector

In 1951, John von Neumann presented a method for removing all 0/1 bias from a RNG. This method, successively defined as *the von Neumann corrector* produces a balanced distribution of ones and zeros. As can be seen from Tab. 1.2, the function converts bit pairs [0,1] from the TRNG into an output 1 bit and the pairs [1,0] into an output bit 0. The pairs of bits [0,0] and [1,1] are discarded.

**Table 1.2:** von Neumann corrector.

| Input bit pair | Output bit |
|---|---|
| 0 0 | discard |
| 0 1 | 1 |
| 1 0 | 0 |
| 1 1 | discard |

The Von Neumann corrector is very efficient in terms of producing an equal distribution of 1s and 0s. But, since the output rate of the Von Neumann corrector is not constant, the generated bits need to be stored, (e.g., in a shift register) before

using for further processing. Even with very high entropy TRNG, the maximum bit rate achievable is half the bit rate of the TRNG.

#### 1.2.3.3 Secure Hash Algorithm (SHA)

Secure Hash Algorithms are four cryptographic functions distinguished chronologically as: SHA-0, SHA-1, SHA-2, SHA-3. A long explanation is out the scope of this work. Basically a long bit sequence is processed by a complex non-linear function, defined as *compression function*. The hashing process is repeated many times, each *round* the compression function change slightly. The output produced is called *digest* because it is always a reduction of the provided input, notice that even if the complexity of the functions very high, the algorithm is still *invertible* (pseudo-randomness).

SHA-1 (1995) is the most widely used hash function of the recent years. Each round is described in Fig. 1.3.

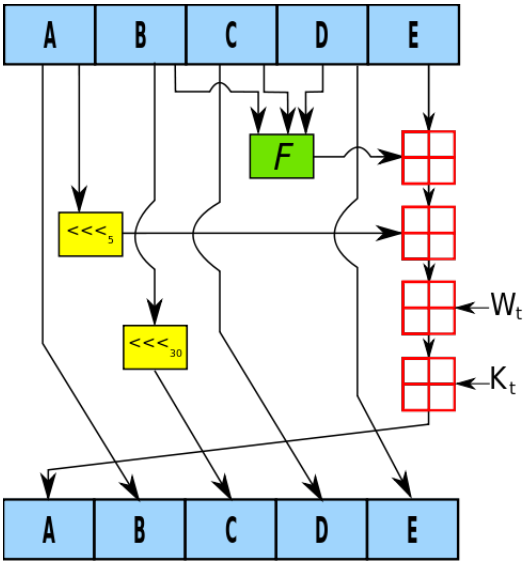



**Figure 1.3:** SHA-1 round.

A B C D and E are 32-bit sequence, F is the compression function that varies each round, $\ll_n$ is the left-bit rotation by $n$ places operator, $n$ varies each operation. $W_t$and $K_t$ are respectively the expanded message word and the constant of round $t$. $\boxplus$ denotes addition modulo $2^{32}$.

After 80 rounds a 160-bit digest output is produced.

AES posso anche non metterlo

### 1.2.4 TRNG Figures of merit & Tradeoffs

As a summary for the end of this chapter, are presented in Tab. 1.3 the usual *tradeoffs* for the design and testing of a TRNG circuit. Notice that, the *classical* metric, namely: power, area and throughput, are not sufficient to classify a TRNG IC, there is the need of additional variable that quantifies the security and the level of randomness of the circuit.

**Table 1.3:** TRNG circuit tradeoffs.

| **Left** | **Right** |
|---|---|
| Area /Power Consumption | Bit Entropy |
| ES Complexity | DPP Complexity |
| Testability /Tamper Resistance | Embeddability /Reusability |
| Mixed Circuit | Fully Digital circuit |
| Throughput /Bandwidth /Entropy Rate | Hardw /Softw complexity |
| Specification /Certification | Dedicated Hardware /Cost |
| Technology Scaling | Process Variation |
| ASIC Implementation | FPGA implementation |
| (Re)Configurability | Application Dependence |

# 2 Randomness Toolbox

## 2.1 Thermal Noise (Johnson–Nyquist noise)

Walter Schottky first postulated the existence of **thermal noise** and **shot noise** in 1918 but J. B. Johnson discover and measure it. This in an extract that came from [15]:

> *"Ordinary electric conductors are sources of spontaneous fluctuations of voltage which can be measured with sufficiently sensitive instruments. This property of conductors appears to be the result of thermal agitation of the electric charges in the material of the conductor.*
>
> *The effect has been observed and measured for various conductors, in the form of resistance units, by means of a vacuum tube amplifier terminated in a thermocouple. It manifests itself as a part of the phenomenon which is commonly called tube noise. The part of the effect originating in the resistance gives rise to a mean square voltage fluctuation V2 which is proportional to the value R of that resistance. The ratio V2/R is independent of the nature or shape of the conductor, being the same for resistances of metal wire, graphite, thin metallic films, films of drawing ink, and strong or weak electrolytes. It does, however, depend on temperature and is proportional to the absolute temperature of the resistance. This dependence on temperature demonstrates that the component of the noise which is proportional to R comes from the conductor and not from the vacuum tube..."*

After Johnson's results, H. Nyquist theoretically deduces the noise electromotive force (thermal noise) from thermodynamics and statistical mechanics [23]. He started from the point that the electromotive force due to thermal agitation in conductors is a universal function of *frequency*, *resistance* and *temperature* and of these variable only. Furthermore this statement is valid independently on the type of conductor e.g., silver, lead or electrolyte.

To determine the form of this function it can be convenient to consider two **ideal** conductors, each of resistance R, connected as showed in Fig. 2.1 by means of a non-dissipative transmission line. In order to avoid radiation one conductor may be internal to the other. The characteristic capacity C and impedance L per unit length of the line is modeled in order to obtain R as characteristic impedance. Under these conditions there are no reflection at either end of the line. Let the length of the line by $l$ and the velocity of propagation $v$.

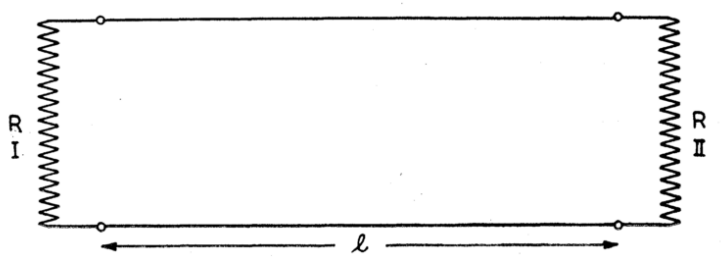



**Figure 2.1:** Resistive network thermal electromotive force.

After thermal equilibrium is established, let the absolute temperature of the system be $T$. There are now, due to the generated thermal electromotive force, two trains of energy traversing the line, one from left to right and viceversa.

At any instant after thermal equilibrium, let the line be *isolated* from the two conductors by application of a short-circuit at both ends. Now there are complete reflection at the two ends and the energy which was on the line at the time of isolation remains trapped. It is possible to model the line as *vibrating at its natural frequency.* The fundamental frequency is

$$\nu_0 = v/2l. \tag{2.1}$$

and it is quantized for an integer value $i = 1, 2, 3, ..., n$ by the relation $\nu_i = i\nu_0$. Consider any $\nu_i$ as a *mode* of vibration or as a *degree of freedom* of the system. Each degree of freedom has an average constant energy associated equal to:

$$E_{thermal} = k_B T. \tag{2.2}$$

where $k_B$ is the Boltzmann constant. Half of $E_{thermal}$ is *magnetic* and half is *electric*. The total energy of the vibrations within a generic frequency interval $\mathrm{d}\nu$ is: $(2lk_BT/v)\,\mathrm{d}\nu$.

But since there is no reflection in the Fig. 2.1 configuration, this is the energy within that frequency interval that the two conductors transferred to the line during the time of transit $l/v$ and the average power within that frequency interval transferred by the two conductors to the line during the same time interval is $k_BT\,\mathrm{d}\nu$.

Using basic formulas of circuit theory, $I = E/2R \quad P = RI^2$, that link current, voltage and power of a conductor is possible to write the square-root of the thermal electromotive force in a conductor of pure resistance R and temperature T, see Eq. 2.3:

$$E_\nu^2\,\mathrm{d}\nu = 4Rk_BT\,\mathrm{d}\nu. \tag{2.3}$$

This expression can be extended for any other network build up of *impedance* $Z_\nu$ members at temperature T:

$$Z_\nu = R_\nu + iX_\nu = 1/Y_\nu.$$

$$E^2 \, \mathrm{d}\nu = 4R_\nu k_B T \, \mathrm{d}\nu. \tag{2.4}$$

Let $Y(\omega)$ be the transfer *admittance* of any network from the member in which electromotive force is generated to the member in which the resulting current is measured. Let $I\omega = 2\pi\nu$ and let $R(\omega) = R_\nu$ be the resistance of the member in which the electromotive force is generated. The squared value of the current measured within the interval $\mathrm{d}\nu$ is :

$$I^2 \, \mathrm{d}\nu = E_\nu^2 \left|Y(\omega)\right|^2 \, \mathrm{d}\nu = (2/\pi) \, k_B T R(\omega) \left|Y(\omega)\right|^2 \, \mathrm{d}\omega.$$

Integrating from 0 to $\infty$

$$I^2 = (2/\pi) \, k_B T \int_0^\infty R(\omega) \left|Y(\omega)\right|^2 \, \mathrm{d}\omega. \tag{2.5}$$

Eq. 2.5 is the same formula obtained in Johnson's paper, confirming Nyquist theoretical proof.

Quantities such *charge, number* and *mass* of the carriers of electricity do not appear explicitly in the formula, therefore they enter indirectly because they influence the value of $R_\nu$.

Notice that electromotive force in engineer typical language is know as *one-sided power spectral density*, or *voltage variance (mean square) per hertz of bandwidth* denoted as:

$$\bar{v_n^2} = 4k_B T R \Delta f. \tag{2.6}$$

The thermal (Johnson-Nyquist) noise in is ideal approximation is defined as **white** noise for reason that will be clear in sec. 2.6.2 with its statistical description. Notice that the noise's color classification includes also *pink*, *red (Brownian)*, *purple* and *gray*. In addition pink noise is often defined as *flicker* noise or $1/f$ noise.

There are two theorems that can be used for thermal noise analysis in *thermal equilibrium*. The first one derives from the **Parseval's theorem** and states that:

**Theorem 2.1.** *The noise mean power is equal to the integral of the Power Spectral Density (PSD) of noise at thermal equilibrium, see Eq. 2.6, over all the frequencies.*

The second one is more powerful because simplify computations in complex systems, and it is called **equipartition theorem** and it comes from statistical mechanics. It states that:

**Theorem 2.2.** *Any energy storage elements (also called "degree of freedom"), such as inductor or capacitor, in thermal equilibrium holds an average noise energy of $kT/2$.*

As an example, using the equipartition theorem, it can be easily computed the noise mean power of an RC network without any integral. The only energy storage element (degree of freedom) in this case is the capacitor, recalling that the mean energy is $E = \frac{1}{2}VQ$ , and the capacitance is defined as $C = Q/V$, it is easy to see that the average noise energy is:

$$\frac{1}{2}CV^2 = \frac{kT}{2}. \tag{2.7}$$

thus the noise mean power is:

$$V^2 = \frac{kT}{C}. \tag{2.8}$$

Notice that, for this circuit, the noise mean power does *not* depend on the value of the resistance R. This result is not valid in all situation, in particular for more complex circuit where they must be introduced *parasitic* capacitance that can lead to a noticeable degradation in noise performance. A common example is Switched Capacitors circuits that implement Track-and-Hold function used for filters, data converters (A/D and D/A), sensor interfaces and dc-dc converters.

Noise mean power and noise Power Spectral Density (also called Power Density Spectrum PDS) are fundamental quantities for engineering applications, their computations involve statistics that helps to map randomness into a measurable set, see sec. 2.5 for more details. In the specific, noise analysis it is a matter of hypothesis and abstractions, it uses sampling technique as explained in sec. 2.4, lumped circuit elements to model noise sources, Channel (Network) Transfer Functions (CTFs) to model input-output relations over frequency and discrete-time domains to model clocked circuits.

## 2.2 Fourier Transform

**Definition 2.1.** For **continuous** variable, given any integrable function $f : \mathbb{R} \to \mathbb{C}$ , can be defined a transformation $\mathcal{F} : \mathbb{R} \to \mathbb{C}$ such that

$$\mathcal{F}\left[f(y)\right] = \hat{f}(x) = \int_{-\infty}^{+\infty} f(y)e^{-2\pi ixy}\,\mathrm{d}y. \tag{2.9}$$

when exists the inverse it has the form:

$$\mathcal{F}^{-1}\left[\hat{f}(x)\right] = f(y) = \int_{-\infty}^{+\infty} \hat{f}(x)e^{2\pi ixy}\,\mathrm{d}x. \tag{2.10}$$

**Definition 2.2.** For **discrete** variable, given a function $x_n : x\left[n\right]$ with $n = 0, 1, 2, ..., N-1$, can be defined its Discrete Fourier Transform (DFT) as:

$$\mathcal{F}\left[x\left[n\right]\right] = X(k) = \sum_{n=0}^{N-1} x\left[n\right] e^{\frac{-2\pi i}{N}kn}. \tag{2.11}$$

As before if exists the Inverse Discrete Fourier Transform (IDFT) and it has the form:

$$\mathcal{F}^{-1}\left[X(k)\right] = x\left[n\right] = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N}kn}. \tag{2.12}$$

For any discrete time function $g\left[n\right]$, that comes from the sampling of a continuous function $g$, sampled $n$ times with period $T$, exists a precise relation between the Fourier transform of the discrete-value samples and the Fourier transform of the continuous sampled function:

$$G(\xi) = \frac{1}{T}\sum_{k=-\infty}^{+\infty} \hat{g}(\xi - \frac{k}{T}). \tag{2.13}$$

**Theorem 2.3.** *so the DFT of the discrete-value samples $G(\xi)$, is equal to the periodic repetition $1/T$ of the Fourier Transform of the continuous sampled function $\hat{g}(\xi)$.*

In both cases, the typical engineering application involves the relation between functions of time time measured in $s$ and their transformation in the frequency domain, measured in $s^{-1} = Hz$.

## 2.3 Probability theory

All aspects that regards randomness, as TRNG, should be encoded in something usable, so a clear mathematical model is necessary. *Probability theory* is an attempt to accomplish this purpose.

**Definition 2.3.** Given $\Omega$ as the set of all possible event and $A \subseteq \Omega$ as a single event. Then it is useful to define a family of subsets of $\Omega$ that satisfy certain properties, this subset is called $\Sigma(\Omega)$ and it is a *$\sigma$-algebra* of $\Omega$ Iff

1. $A \in \Sigma(\Omega)$;
2. if $A \in \Sigma(\Omega)$ then $\bar{A} = \Omega/A \in \Sigma(\Omega)$;
3. if $A_j \in \Sigma(\Omega)$ then $\bigcup_j A_j \in \Sigma(\Omega)$;
    a) applying *De Morgan's laws* can be noticed that 3. is valid also for intersection operation $\bigcap$. [1]

**Definition 2.4.** Given $\Omega$ and one of its $\sigma$-algebra $\Sigma(\Omega)$, it is possible to define a *probability function* Pr such that:

[1] $\overline{A \cup B} \equiv \overline{A} \cap \overline{B}$ and $\overline{A \cap B} \equiv \overline{A} \cup \overline{B}$

1. $\Pr(\Omega) = 1$;
2. if $A_j \cap A_k = \emptyset$ then $\Pr\left(\bigcup_j A_j\right) = \sum_j \Pr(A)$.

This function assumes value in the interval [01] and it is used as a fundamental metric (measure) to model random things.

**Definition 2.5.** Furthermore, can be defined a particular kind of $\sigma$-algebra called **Borel $\sigma$-algebra** on the set of $\mathbb{R}$ numbers. It is represented with $S_y$ and satisfies the following properties:

1. $S_y = \{x | x \leq y\}$;
2. $S_y \in \Sigma(\Omega)$;
3. $\Omega \in \mathbb{R}$.

So $S_y$ is the smallest $\sigma$-algebra in the set of $\mathbb{R}$ numbers that contains *all real intervals*.

**Definition 2.6.** Borel $\sigma$-algebra can be used to define a *Random Variable (RV)* as a function $x : \Omega \to \mathbb{R}^n$ such that:

$$if \quad A \in \Sigma(\Omega) \to x^{-1}(A) \in \Sigma(\Omega).$$

From this definition is possible to develop all the classic probability theory.

**Definition 2.7.** Given any RV x, it can be always defined its *Cumulative Distribution Function (CDF)*, that is a function $F_x : \mathbb{R}^n \to [0, 1]$

$$F_x(y) = \Pr(X \leq y). \tag{2.14}$$

and its *Probability Distribution Function (PDF)*, that is a function $f_x : \mathbb{R}^n \to \mathbb{R}^+$

$$f_x(y) = \frac{\mathrm{d}}{\mathrm{d}y} F_x(y). \tag{2.15}$$

It is also valid the theorem of *Lebesgue*, valid for both discrete-value and continuous-value PDF, that states that:

**Theorem 2.4.** *Given a continuous(discrete)-value RV $x$ and its PDF and CDF. If you have a step in the PDF then put a $\delta$-function ($\delta_{ij}$-function) in the CDF.*[2]

[2] In the context of signal processing, the *Dirac* delta function ($\delta$) is often referred to as the *unit impulse symbol*. Its discrete analog is the *Kronecker* delta function ($\delta_{ij}$) which is usually defined on a finite domain and takes values 0 and 1.

Thus for a continuous-value RV $x$, and its CDF $F_x$. If $f_x$ exists, is a non-negative Lebesgue-integrable function thus are valid the following relations:

$$F_x(y) = \int_{-\infty}^{y} f_x(\xi)\,\mathrm{d}\xi.$$

$$\Pr\left[a \leq X \leq b\right] = \int_{a}^{b} f_x(\xi)\,\mathrm{d}\xi.$$

$$\int_{\infty} f_x = \int_{\infty} PDF = 1.$$

and for a discrete-value RV $x$, if $f_x$ exists, are valid the following relations:

$$F_x(y) = \sum_{i=-\infty}^{j} f_x\left[i\right] = \sum_{i=-\infty}^{j} f_{x_i}.$$

$$\Pr\left[a \leq X \leq b\right] = \sum_{i=a}^{b} f_x\left[i\right].$$

$$\sum_{i} f_x\left[i\right] = \sum_{i} PDF = 1.$$

**Definition 2.8.** Another fundamental quantity is the *expectation* E, of a continuous(discrete) RV $x$, defined as:

$$\mathrm{E}\left[x\right] = \int_{-\infty}^{+\infty} x f_x(x)\,\mathrm{d}x. \tag{2.16}$$

$$\mathrm{E}\left[x\right] = \sum_{j}^{\infty} x_j f_{x_j}. \tag{2.17}$$

From this quantity can be extracted many statistical features of a RV, see Tab. 2.1:

**Table 2.1:** RV Statistical Features.

| | |
|---|---|
| mean | $\mu = \mathrm{m}_x = \mathrm{E}\left[x\right]$ |
| variance | $\sigma_x^2 = \mathrm{E}\left[\left\|x - \mathrm{m}_x\right\|^2\right] = \mathrm{E}[x^2] - (\mathrm{E}[x])^2$ |
| standard deviation | $\sigma_x = \sqrt{\mathrm{E}[(x-\mu)^2]} = \sqrt{\mathrm{E}[x^2] - (\mathrm{E}[x])^2}.$ |
| j$^{th}$ order central moment | $\mathrm{M}_x^j$ =E$[(x-\mu)^j]$ |
| j$^{th}$ order non central moment | $\mathrm{m}_x^j$ =E$[x^j]$ |
| j$^{th}$ order absolute moment | $\mu_x^j$ =E$[\left\|x\right\|^j]$ |

Given these quantities, there exists a theorem called *Chebishev's inequality*, useful to estimate a probability of a certain RV with unknown PDS, using only its mean and variance:

**Theorem 2.5.** *Given a RV $x$, with finite mean $\mu_x$ and finite variance $\sigma_x^2$, then for any real number $k > 0$,*

$$0 \leq \Pr(|x - \mu_x| \geq k\sigma_x) \leq \frac{1}{k^2} \tag{2.18}$$

*or equivalently*

$$0 \leq \Pr(|x - \mu_x| > k) \leq \frac{\sigma_x^2}{k^2}. \tag{2.19}$$

.

**Definition 2.9.** There exists also a *Moment Generating Function (MGF)* of a RV that uses inverse Fourier transform, defined as:

$$\psi_x(\omega) = \mathcal{F}^{-1}\left[f_x\right](\omega) = \mathcal{F}\left[f_x\right](-\omega). \tag{2.20}$$

MGF has two important properties:

1. Can be expanded using Taylor series, this expansion serves to prove the Central Limit Theorem (CLT) see sec. 2.9:

$$\psi_x(\omega) = \sum_{j=0}^{k} \mathrm{m}_x^j \frac{(2\pi i\omega)^j}{j!} + \mu_x^{k+1} \frac{|2\pi\omega|^{k+1}}{(k+1)!}\theta(\omega).$$

2. Deriving it $k$ times and computing its value in 0, can be obtained the correspondent $k^{th}$order non central moment:

$$\frac{\mathrm{d}^k}{\mathrm{d}\omega^k}\,\psi_x(0) = (2\pi i)^k\,\mathrm{m}_x^k.$$

**Definition 2.10.** Given two RV $x_0$ and $x_1$, is possible to define their *covariance* as:

$$\gamma_{x_0x_1} = \mathrm{E}\left[(x_0 - \mathrm{m}_{x_0})^*\,(x_1 - \mathrm{m}_{x_1})\right]. \tag{2.21}$$

The * correspond to the complex conjugation operator or conjugate transposition of a matrix. Notice that, if $x_0 = x_1 = x$, then $\gamma_{xx} = \sigma_x^2$. Furthermore, if $\gamma_{x_0x_1} = 0$, then the two RV are *uncovariated.*

Finally given two RV $x_0$and $x_1$,it is possible to define their *joint PDF* also called *joint probability distribution*

$$f_{x_0x_1} = f_{x_0}\,\text{and}\,f_{x_1}. \tag{2.22}$$

From this definition we can check one of the most important probability of RVs, their **independence.**

**Theorem 2.6.** *Two or more RVs are independent if their joint PDF can be factorized into independent single PDFs.*

$$f_{x_0,x_1,...,x_n}(y_0, y_1, ..., y_n) = f_{x_0}(y_0) f_{x_1}(y_1) ... f_{x_n}(y_n)$$

.

**Definition 2.11.** Given two RV $x_0$ and $x_1$, is possible to define their *cross-correlation*:

$$\rho_{x_0 x_1} = \frac{\gamma_{x_0 x_1}}{\sigma_{x_0} \sigma_{x_1}}. \tag{2.23}$$

While the correlation of a random vector X is considered to be the correlation matrix (matrix of correlations). If $x_o$ and $x_1$ are two independent RVs with PDFs $f_{x_0}$ and $f_{x_1}$, respectively, then $f_{x_1 - x_0}$is formally given by the cross-correlation $\rho_{x_0 x_1}$,it is also possible to compute the PDF of the sum of the two inverting the second RV.

There is a parallel theory of cross-correlation for *signal processing* application. In this case, cross-correlation is a measure of *similarity* of two waveforms. This is also known as a sliding dot product or sliding inner-product. For continuous functions, f and g, the cross-correlation is defined as:

$$(f \star g)(t) \ = \int_{-\infty}^{\infty} f^*(\tau)\ g(t+\tau)\, \mathrm{d}\tau.$$

where f * denotes the complex conjugate of f. Similarly, for discrete functions, the cross-correlation is defined as:

$$(f \star g)[n] \ = \sum_{m=-\infty}^{\infty} f^*[m]\ g[n+m].$$

To compare these quantities see Fig. 2.2

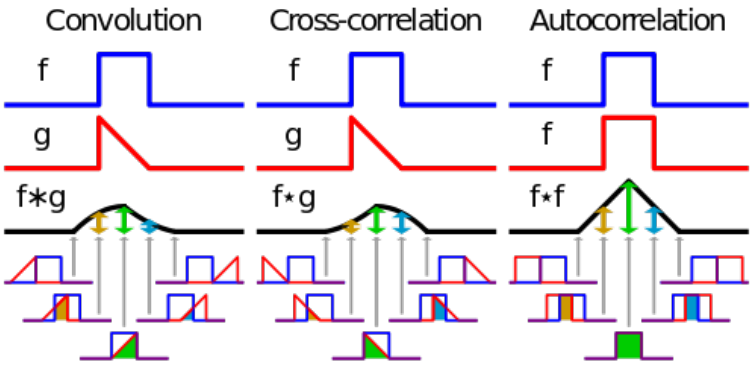



**Figure 2.2:** Convolution, Cross-Correlation and Auto-correlation.

.

The cross-correlation is similar in nature to the convolution of two functions.

Notice that if two RVs are independent, then they are also uncovariated (and also uncorrelated), the inverse is true only for Gaussian(normal) RV, see sec. 2.6.

## 2.4 Quantization

Every digital system presents a quantization circuit responsible of the translation of real-analog signal into processable-digital one. The aim of this circuit is to transform a continuous-time function, that can assumes an infinite set of values into a discrete-time function, that can assume only a *finite* set of values. This operation is not reversible so it produces a loss of information, defined as *error of quantization.* Notice that, every circuit has a saturation point, so in real application the set of infinite values is bounded between two voltage/current values.

The mathematical model of the quantization process is a function $Q : \mathbb{R} \to \mathbb{R}$ such that, defining $X_j = \left]j\Delta - \frac{\Delta}{2}, j\Delta + \frac{\Delta}{2}\right]$, can be obtained

$$Q(x) = j\Delta \quad \text{if} \quad x \in X_j. \tag{2.24}$$

for a certain *quantization step* $\Delta$.

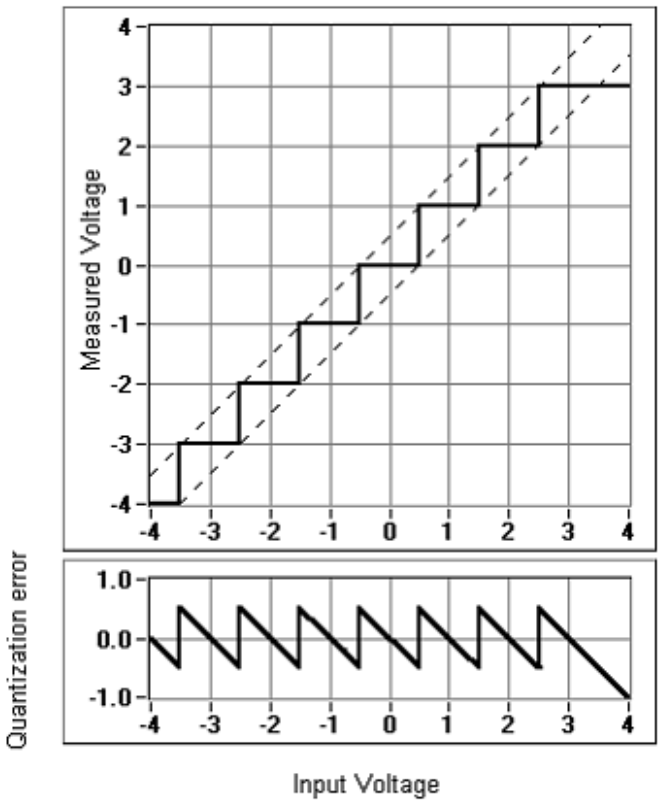



**Figure 2.3:** Input-Output relation of a quantization process $Q(x)$, of step $\Delta$.

Suppose to model the input signal as a RV $x$, the application of $Q$ on it produces two additional RVs:

The RV that model the quantized value: $\omega = Q(x)$. Its characterization is based on the fact that, all the values $x$ that fall in the interval $X_j$ are associated to $j\Delta$.

$$\begin{aligned} f_\omega(z) &= \sum_{j=-\infty}^{\infty} \delta(z - j\Delta) \int_{X_j} f_x(y)\,\mathrm{d}y \\ &= \sum_{j=-\infty}^{\infty} \delta(z - j\Delta) \left[ F_x\left(j\Delta + \frac{\Delta}{2}\right) - F_x\left(j\Delta - \frac{\Delta}{2}\right) \right]. \end{aligned} \tag{2.25}$$

The RV that model the error of quantization: $e = x - \omega$. Its characterization is much important with respect to Eq. 2.25, because after the quantization process, the system conserve $\omega$ and ignores $e$. Thus a statistical model for $e$ is necessary to determine and guarantee the margins of error of the system. There are some assumption that can be made about $e$:

1. $e$ is uniformly distributed in the interval $\left[-\frac{\Delta}{2}, \frac{\Delta}{2}\right]$;
2. two errors measured in different time intervals, defined as $e_1$ and $e_2$, are at least uncorrelated and, in the best case, independent;
3. $e$ and $x$ are uncorrelated and, in the best case, independent.

There are theorems able to verify these conditions the Moment Generating Function (MGF), see Eq. 2.20, of the input RV $x$. The first one permits to verify the uniform distribution of the quantization error in the interval $\left[-\frac{\Delta}{2}, \frac{\Delta}{2}\right]$ (cond 1.):

**Theorem 2.7.** *If $e = x - Q(x)$ then, $e$ is uniformly distributed in $\left[-\frac{\Delta}{2}, \frac{\Delta}{2}\right]$ iff*

$$\psi_x\left(\frac{k}{\Delta}\right) = 0$$

*for all $k \neq 0$.*

This theorem can be interpreted remembering that $\psi_x(z) = \mathcal{F}[f_x](-\omega)$. The PDF of $x$ is a real function.

To verify that two uncorrelated errors of quantization, $e_0$ and $e_1$, are independent (cond 2.) it is valid

**Theorem 2.8.** *If $e_0 = x_0 - Q(x_0)$ and $e_1 = x_1 - Q(x_1)$ are two quantization errors uniformly distributed in $\left[-\frac{\Delta}{2}, \frac{\Delta}{2}\right]$, then they are independent iff*

$$\psi_{x_0,x_1}\left(\frac{k_0}{\Delta}, \frac{k_1}{\Delta}\right) = 0$$

*for each couple $(k_0, k_1) \neq (0, 0)$.*

There is also a theorem for evaluate the correlation between the input RV and the error of quantization (cond 3.):

**Theorem 2.9.** *The input RV $x$ and the error of quantization $e$, uniformly distributed in $\left[-\frac{\Delta}{2}, \frac{\Delta}{2}\right]$, are uncorrelated if*

$$\frac{\mathrm{d}}{\mathrm{d}y}\psi_x\left(\frac{j}{\Delta}\right) = 0$$

*for all $j \neq 0$.*

## 2.5 Stochastic Process (SP)

A **Stochastic Process (SP)** is any process that contains some randomness and information which a user wants to extract. It is a fundamental abstraction because it permits to introduce the concept of *time* as follows. Take a SP and extract from it a number of *profiles* at precise time intervals, also called *realizations* of the SP. Then the SP's realizations can be seen as a vector of RVs, namely a *random vector* and each sample is a function (discrete or continuous depending on the initial hypothesis) and can be modeled as a RV, thus all formulas introduced before are usable, such as in Tab. 2.1. These collection of RVs permits to model a SP.

A SP can involve both continuous-time (c-t) and discrete-time (d-t) RVs. In this work the latter will be used more often. Furthermore its realization can assume $\mathbb{C}$ or $\mathbb{R}$ values, notice that all the theory developed for complex values can be reduced simply for real values RVs, so it is preferred in this work because it is more general.

Remember the ideal RNG showed in sec. 1.1 ? In statistical terms it can be described with a discrete-time stochastic process, formed by a sequence of *independent and uniformly distributed (IID)* random variables, distributed in the set $\{0, 1\}$, with probability $p = 0.5$. This stochastic process is very common in literature and is known as **Bernoulli process**.

### 2.5.1 SP Characterization

Generally exists three ways to model a SP:

1. **Joint PDF characterization.** The first type of char is the most complete, but requires a huge amount of data. Fixing $n$ time instants $t_0 < t_1 < ... < t_{n-1}$, it is possible to get a collection of $n$ profiles $x(t_0), x(t_1), ..., x(t_{n-1})$ from the SP, then find its joint PDF of order $n$: $f_{x(t_0)x(t_1)...x(t_{n-1})}(y_0, y_{1,}, .., y_{n-1})$. For $n \to \infty$, the joint PDS provides any information about the SP but is evidently infeasible in practice; it is usual to steak to *second* order joint PDS characterization.

2. **Autocorrelation-Autocovariance function characterization.** The second type of characterization involves *auto*-function, that in statistical terms means to

evaluate a RV and its evolution in time. To clarify this concept take the covariance between two discrete-time RVs, as showed in Eq. 2.21, $x_0 = x_0[0]$ and $x_1 = x_0[1]$, so $\gamma_{x_0x_1}$ is the covariance of $x_0$ against a time-shifted version of itself, namely the auto-covariance of $x_0$. With this notion in mind, a SP can be characterized from the auto-covariance and auto-correlation functions of its realizations in the time interval $t_0 < t_1 < ... < t_{n-1}$:

$$C_x(t_0, t_1, ..., t_{n-1}) = \mathrm{E}\left[x^*(t_0)x(t_1)x^*(t_2)...x(t_{n-1})\right]. \tag{2.26}$$

auto-correlation is the cross-correlation of a RV with itself in time

$$K_x(t_0, t_1, ..., t_{n-1}) = \mathrm{E}\left[\Big(x(t_0) - \mathrm{m}_{x(t_0)}\Big)^* \Big(x(t_1) - \mathrm{m}_{x(t_1)}\Big) ... \Big(x(t_{n-1}) - \mathrm{m}_{x(t_{n-1})}\Big)\right]. \tag{2.27}$$

As before, it is impossible the extend to infinity the order of characterization so it is sufficient to limit the evaluation at the $2^{ND}$ order, $n = 2$, to extract information about the *energy* of the SP from the knowledge of Eq. 2.26 or Eq. 2.27 that are both *positive definite* functions(matrix). In the Fourier domain, this fact translates into a real, positive power density spectrum.

3. **Projections characterization.** The third type of characterization is the most powerful. Suppose to fix an interval of observation $\left[-\frac{T}{2}, \frac{T}{2}\right]$, given a SP $x$ it is possible to define all of its realization as *vector space* $v$. It is now possible to project (scalar product) the realizations on some function $\varphi_j$, obtaining the **projection** $\rho_j$:$\rho_j = \langle \varphi_j, x \rangle$. The parameter j dictates the order of aproximation. It is possible to reverse the formula $\sum_j \rho_j \varphi_j$. The set of $\varphi_j$ have to be made of *orthogonal* function, such that $\langle \varphi_j, \varphi_k \rangle = 0$ for each $j \neq k$, and *orthonormal* functions, such that $\langle \varphi_j, \varphi_k \rangle = 1$ for $j = k$, to assure that they are linear independent functions. To obtain the maximum information with the minimum value of $j$, it is sufficient that all the projections are uncovariated.

## 2.5.2 SP Stationarity

Another basic concept is *stationarity*. Imagine to model and design a system, the prototype must work not only during the testing phase but also during operation time, that is much and much longer than testing phase. Considering this system as a SP is possible to use this theorem with many *degree* or *order* of stationarity:

**Definition 2.12.** A SP is $m$-order stationary, with $m = 1, 2, 3, ..\infty$, if $\forall m$ and $\forall t_0, t_1, ..., t_{m-1}$, its joint PDF it is:

$$f_{x(t_0)...x(t_{m-1})} = f_{x(t_0+\Delta t)...x(t_{m-1}+\Delta t)}. \tag{2.28}$$

and Eq. 2.28 is valid $\forall \Delta t \in \mathbb{R}$ for continuous-time SP and $\forall \Delta t \in \mathbb{Z}$ for discrete-time SP.

This means that, all the statistical features of the SP are invariant with respect to any translation in time. Claim to reach infinite order stationarity is absurd, a second order ($m = 2$) stationarity is a good tradeoff.

**Definition 2.13.** A SP is said to be *cycle-stationary* with period T if, for each integer $m$ and time interval $t_0, t_{1,}, ..., t_{m-1}$

$$f_{x(t_0)...x(t_{m-1})} = f_{x(t_0+k\mathrm{T})...x(t_{m-1}+kT)}, \tag{2.29}$$

for each integer $k$ .

## 2.6 Gaussianity

### 2.6.1 Jointly Gaussian Vector (JGV)

The Gaussian theory it is a fundamental part security engineering because it permits to model a lot of situations and it has a set of strong properties that simplify a lot the work. Notice that any Gaussian RV can be completely characterized in terms of its mean and variance.

**Definition 2.14.** Given a set of $n$ real RVs that forms what is called a **random vector**, $x = x_0, x_1, ..., x_{n-1}$.

This collection of RVs is a **Jointly Gaussian Vector (JGV)** if:

$$f_x(y) = \frac{1}{2\pi^{n/2}\det(K_x)} e^{-\frac{1}{2}(y-\mathrm{m}_x)^t K_x^{-1}(y-\mathrm{m}_x)}. \tag{2.30}$$

Notice that this definition can be extended also to complex RVs $x = a + ib$ where $a = Re\{x\}$ and $b = Imm\{x\}$, adding the following condition

$$K_{aa} = K_{bb}$$

$$K_{ab} = -K_{ba}$$

Eq. 2.30 is the same with transposition operator $t$ substituted by the conjugate = transpose operator $+$.

The square auto-correlation matrix of a real discrete-value JGV satisfy certain properties:

- $K_x$ is *symmetric* $K_x = K_x^t$. [3]
- $K_x$ is a *positive semi-definite* matrix
- $K_x$ is *non-singular*; so it is *invertible* and $det(K_x) \neq 0$.

[3]
- In case of a complex JGV the auto-correlation matrix is *hermitian*

Another important Gaussian property regards the MGF, recall Eq. 2.20:

**Theorem 2.10.** *Given a JGV $x = x_0, x_1, ..., x_{n-1}$ and its associated MGF $\psi_x$. It valid the following relation:*

$$\mathcal{F}^{-1}\left[f_x\right](w) = \psi_x(w).$$

Thanks to this particular theorem it is possible to say that *any* linear transformation that take as input something that is Gaussian returns as output something Gaussian.

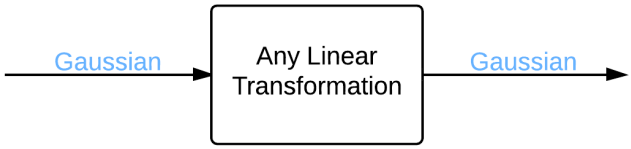



**Figure 2.4:** Linear transformation of a Gaussian quantity.

Given an input JGV $x$ with mean $\mathrm{m}_x$ and auto-covariance matrix $K_x$and a linear transformation $A$ that produces a Gaussian JGV output $w$; then there is a linear relation between the mean/auto-covariance of the input and the output:

$$\mathrm{m}_w = \mathrm{A}\mathrm{m}_x.$$

$$K_w = \mathrm{A}K_x\mathrm{A}^t$$

Given a JGV it is possible to see if it is formed by independent RVs only looking to its $K_x$.

**Theorem 2.11.** *Given a JGV $x = x_0, x_1, ..., x_{n-1}$ , if for each couple $j, k$ with $j \neq k$, the expectation $E\left[\left(x_j - m_{x_j}\right)^t \left(x_k - m_{x_k}\right)\right] = 0$ then $x_0, x_1, ..., x_{n-1}$* are *independent* RVs

## 2.6.2 Additive White Gaussian Noise SP (AWGN)

It is important to mention at least one of the most important type of *discrete-value real* SP.

**Definition 2.15.** A SP can be defined as **Gaussian Stochastic Process (GSP)** if:

$$p = \langle \varphi, x \rangle$$

is a Gaussian RV for each $\varphi \in L^2$

So to characterize a GSP it is sufficient to use a *projection* model and choose a set of orthonormal basis $\varphi$ and know the statistics of the projections. There are at least three fundamental properties of a GSP:

1. Any sample $x[t_n]$ of a GSP is a Gaussian RV, and any linear combination of a finite number of sample is a JGV.

2. A GSP is completely characterize by the mean and the auto-covariance of its projections.

3. Any linear filtering $h$ of an input GSP produces a GSP output.

There is a particular type of GSP called *White Noise GSP* or simply *White Gaussian Noise (WGN)*. It is often incorrectly assumed that Gaussian noise is necessarily white noise, yet neither property implies the other. Gaussianity refers to the probability distribution with respect to the value, in this context the probability of the signal reaching an amplitude, while the term *white* refers to the way the signal power is distributed over time or among frequencies. We can therefore find Gaussian white noise, but also Poisson, Cauchy, etc. white noises. Thus, the two words "Gaussian" and "White" are often both specified in mathematical models of systems.

**Definition 2.16.** A SP x is called **White Gaussian Noise SP (WGN SP)** if for each $\varphi \in L^2$ any projection $p = \langle \varphi, x \rangle$ is a Gaussian RV with mean $\mathrm{m}_p$ and variance $\sigma_p^2$.

A discrete-value second order projection characterization of a WGN SP is a good approximation of many real-world situations such has disturbance during a measurement. In this case it is allowed to use the word *additive* because the noise can be treated as an additive term that sums to the useful one (linear relation). Also notice that:

$$E[p_0 p_1] = \frac{N_0}{2} \langle \varphi_0, \varphi_1 \rangle = 0. \tag{2.31}$$

$$K_p = \begin{matrix} \sigma_{p_0}^2 & 0 \\ 0 & \sigma_{p_1}^2 \end{matrix} . \tag{2.32}$$

So with this characterization it is impossible to linearly predict any terms because the cross-correlation between different projections it is always 0. In fact the terms *White* is used because the Power Density Spectrum (PDS) of a WGN SP, that is simply the Fourier transform of the auto-correlation function, is **flat**, so there are no dominant frequency component that carrier higher amount of energy.

## 2.7 Entropy

The most important metric to analyze and test a RNG is **entropy.** Its definition is taken from *Information Theory* field, and it is abstract and different from the definition of entropy in thermodynamics sense. It is a real number between zero and one, that gives a measure of how random a particular process is. There are several way to measure it, historically the first definition is *Shannon entropy* that comes from by Claude Shannon[31]:

$$H_i = -\sum_{i=i}^{n} p_i log_2 (p_i) \tag{2.33}$$

$H_i$ quantify how much bits of information can be compressed in a *loss-less* bit sequence, in our terms this translates in how much randomness is present in a bit sequence. One perfect coin toss contains one bit of entropy (or one bit of information because of no redundancy). By using a base-2 logarithm the entropy is measured in bit. In the formulas $p_i$ is the probability of the process of being in the $i$th of $n$ possible states, or returning the $i$th of $n$ possible outputs. In Fig. 2.5 there is an example of Shannon's entropy computation[4] :

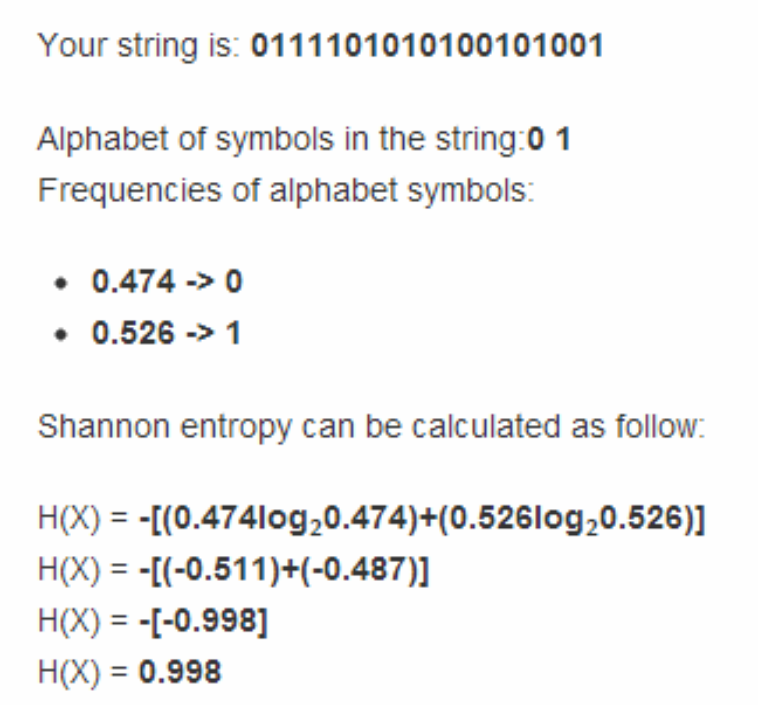



**Figure 2.5:** Shannon entropy computation.

To better understand and also extend the last definition is often useful talk in term of *entropy rate* or *entropy per symbol*, that is applied to bit sequence that

[4]Free Shannon entropy calculator/tutorial http://www.shannonentropy.netmark.pl/

encode a specific alphabet formed by symbols. As an example, consider a source that produces the sequence ABABABAB... If the symbol's length is equal to one, the entropy rate 1 bit per character but, if the symbols are formed by two letter the entropy rate is 0 bit per character, because the same symbol is repeating over time. These simple example underlines how much randomness is subject to the boundary condition.

There is also a lower bound quantity, namely *min-entropy:*

$$H_\infty = min_{i=i}^{n} \left(-log_2 \left(p_i\right)\right) \tag{2.34}$$

that measures the probability that an attacker can guess the state with a single guess. Notice that:

$$H_\infty \leq H_i \tag{2.35}$$

the min-entropy of a process is always less or equal to its Shannon entropy.

As an example, in the case of a RNG that produces a $k$-bit *binary* result, e.g. 010101 for $k = 6$, $p_i$ is the probability that an output will equal $i$, where $0 \leq i < 2^k$. Thus a *perfect* binary RNG has $p_i = 2^{-k}$, so Eq. 2.33 and Eq. 2.34 are both equal to $k$ bits, and all possible outcomes are equally likely (IID outputs). The information present in the output cannot, on average, be represented in a sequence shorter than $k$ bits, and an attacker cannot guess the output with probability greater than $2^{-k}$. A random number generator for cryptographic applications should appear to relative *computationally-bounded* adversaries to be close as possible to a perfect RNG, so for a good analysis result a target RNG, at least in principle, shall not be distinguishable from a perfect RNG that is an useful ideal abstraction.

## 2.8 Hamming Distance

The first error correction code was used in 1950 to detect errors in telephone central offices to provide a better Quality of Service (QoS) in the American Telephone and Telegraph Company (AT&T) and it was used also for digital computer applications by Richard Hamming [9]. While, at that time, *error detecting* codes, such as 2out-of-3 or 3out-of-7 code, and *self-checking* circuit were still available, error *correction* had represented a big step forward in computer science.

As first hypothesis each code *symbol* or *word* is represented in the *binary* form, by a sequence of 0's and 1's. All code used in this subsection are defined as *systematic,* that are codes whose

1. Symbol length are all equal, they are formed by $n = k + m$ binary digits where
   a) $n$ total symbol's bit,
   b) $m$ information bits,
   c) $k$ error detection and correction bits.

2. The position checked are independent of the information contained in the symbol,

3. The checks are independent of each other,

4. Use *Parity check* bit.

Hypothesis 1. produces **redundancy R** in the code, defined as the ratio of total symbol's bit and the information bits.

$$R = n/m \tag{2.36}$$

This serves to measure the *efficiency* of the code, because on one side R can lower the effective channel capacity for sending information plus the need of extra hardware equipment but on the other side, is impossible to detect and correct an errors without introducing redundancy in the code.

To define an error correcting code first it is necessary to pass from an error detecting one because any $d$ error correcting code is also an $d$ error detecting code, obviously the inverse is not true. To check a symbol the most common method used is the *parity check*. As an example consider the *even(odd)* parity check: take a symbol of $n$ bits, put $m = n - 1$ bits of information, and the last either a 0 or a 1 so that the symbol has an *even(odd)* number if 1's. This is clearly a single error detecting code, since any single error in a symbol would leave an *odd(even)* number of bits. The redundancy of this code is:

$$R = \frac{n}{n-1} = 1 + \frac{1}{n-1}$$

Increasing $n$, the probability of at least one error in a symbol increase; and the risk of a *double* error, which would pass undetected, also increase. Furthermore, a parity check need not always involve all positions of the symbol but may be a check over *selected* position.

To extend and control the power of error correcting and detecting code it is useful to introduce a *geometrical* model, that consists in identifying the various sequences of 0's and 1's, which are the symbol (word) of a code, with vertices of a unit $n$-dimensional cube. The code *points*, each one of $n$ bits, are labeled as x,y,z,... and they form a subset of the set of all vertices of the cube. Thus the space is formed by $2^n$ points and can be introduced a **(Hamming) distance D(x,y)**, called also as a *metric*. By definition, a single error in a code point changes one coordinate, two errors two coordinates, and in general $d$ errors produce a difference in $d$ coordinates. So, coming back to the unit cube, the minimum number of edges which must be traversed in going from x to y, represent the distance D(x,y) between the two code points, or in other terms the number or error(s) between the two code symbols, see Fig. 2.6.

**Figure 2.6:** 0100→1001 has distance 3 (red path); 0110→1110 has distance 1 (blue path).

This distance function satisfies the usual three conditions for a metric, namely:

1. $D(x, y) = 0 \quad$ if and only if $x = y$
2. $D(x, y) = D(y, x) > 0 \quad if\ x \neq y$
3. $D(z, y) + D(y, z) \geq D(x, z) \quad$ (triangle inequality)

To continue the geometric language, a sphere of radius $r$ about a point x, is defined as all points which are at a distance $r$ from the point x. As an example consider a code with $n = 3$, and four code points: 001,010,100,111. Each symbol may be chosen as the *center* of a sphere of radius $r = 2$, and the other three symbol lies on the sphere's surface or in simple words the minimum distance between all code words is two. Thus it follows that any single error will carry a code point to a meaningless symbol. This in turn means that any single error is detectable with this code. Additionally, if the minimum distance between code points is three, then any single error will leave the point nearer to the correct code point than to any other code points, and this means that any single error will be correctable. Go deep with this theory permits to link the minimum distance of code points with code's features, as show in Tab. 2.2

**Table 2.2:** Minimum distance $D_{min}(x, y)$ of code points vs Code's features.

| $\mathbf{D_{min}(x, y)}$ | **Code Features** |
|---|---|
| 1 | Uniqueness |
| 2 | Single Error Detection |
| 3 | Single Error Correction |
| 4 | Single Error Correction + Double Error Detection |
| 5 | Double Error Correction |
| | Etc. |

All the distances between code points must equal or exceed the minimum distance listed. Thus the problem of finding suitable codes is the same as that of finding subsets of points in the space, which maintain *at least* the minimum distance condition.

It should be noted that, at a given minimum distance, some of the correctability may be exchanged for more detectability; e.g., a subset of code points with $D_{min}(x, y) = 5$ can be used in different manners:

- double error correcting code, thus also double error correction;
- single error correction + triple error detection;
- quadruple error detection, without correction.

Two codes are said to be *equivalent* to each other if, by a finite number of the following operation, one can be transformed into the other:

1. The interchange of any two positions in the code symbols;

2. The complementing of the values in any position of the code symbols.

This definition is very useful because it permits to study a class of codes to the study of typical members of each *equivalence class*. Furthermore, in terms of the geometric representation, equivalence transformations amount to *rotations* (interchange) and *reflections* (complementing) of the unit cube.

The last useful thing to know regards *minimum redundancy* and can be summarized by a powerful general theorem, that is valid for systematic code, where points means number of code symbols and dimensions means symbol's bit length:

**Theorem 2.12.** *To any minimum redundancy code of $N$ points in $n-1$ dimensions and having minimum distance of $2k-1$, there corresponds a minimum redundancy code of $N$ points in $n$ dimensions having a minimum distance of $2k$, and conversely*

This theorem permits to construct min redundancy code ($N$ is fixed) of any minimum distance starting from the simplest one $k = 1$ .

Hamming distance is used in several disciplines including information theory, coding theory, and cryptography. In this work Hamming weight analysis of bits is used as a metric of RNG performance in sec. 4.2.

## 2.9 Limit Behavior of RVs

Try to find for a particular series of RVs $x_0, x_1, ..., x_{n-1}$ that can be approximated by RV $x$. This study starts from the relation of the PDFs of the series and the PDF of the limit RV. The first approach can be with a punctual convergence for their associated PDFs, but it is not too much. A second approach can be with uniform convergence for their PDFs, but this is a too stringent condition. It is needed something in between:

**Definition 2.17.** Given a series of functions $f_k$ I can define a **weak** convergence to the function $f$ as:

$$f_k \rightarrow^w f$$

if for each function $g \int f_k g = \int fg$.

There is a specialized Theorems for weak convergence of RVs series.

**Theorem 2.13.** ***Lyapunov.*** *Given a series of RVs $x_0, x_1, ..., x_{n-1}$ with PDFs $f_k$ and CDF $F_k$, given a single RV $x$ with PDF $f$ and CDF $F$*
$f_k \rightarrow^w f$‘ iff $\mathrm{F_k} \rightarrow \mathrm{F}$. *and* $f_k \rightarrow^w f \quad if \quad \psi_{x_k} \rightarrow \psi_x$. *Thus* $x_k \rightarrow x$.

The second is much more famous with respect to the Lyapunov theorem

**Theorem 2.14.** ***Central Limit Theorem (CLT).*** *Given a series $\mathbb{R}$ independent RVs $x_0, x_1, ..., x_{n-1}$ with means $\mathrm{m}_{x_j} = 0$ and variance $\sigma^2_{x_j}$. It is possible to define $S_w = \sum_{j=0}^{k-1} x_j$ and $w' = \frac{1}{k}\sum_{j=0}^{k-1} x_j$ and $w'' = \frac{1}{\sqrt{k}}\sum_{j=0}^{k-1} x_j$ .*

1. *If, for each $j$, $lim_{k\to\infty}\frac{1}{k^2}\sum_{j=0}^{k-1}\mu^2_{x_j} = 0$ then $f_{w'} \to^w \delta$ .*
2. *If, for each $j$, $lim_{k\to\infty}\frac{1}{k^{3/2}}\sum_{j=0}^{k-1}\mu^3_{x_j} = 0$ and $lim_{k\to\infty}\frac{1}{k}\sum_{j=0}^{k-1}\sigma^2_{x_j} = \sigma^2 < \infty$. then*

$$f_{w''} \to^w \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2}\frac{z^2}{\sigma^2}}.$$

*So the PDF normalized sum, that is not the joint PDF of the RVs, tends to a Gaussian PDF with known mean and variance.*

**Theorem 2.15.** ***de Moivre-Laplace.*** *From CLT can be deduced a special case stating that: given a binomial SP (Bernoulli) of n samples where the trials are independent and the probability of success is $p$ and probability of insuccess $q = 1 - p$ for each. If $n$ is very large the SP can be asymptotically approximated by a Gaussian RV with mean equal to $np$ and std deviation equal to $\sqrt{npq}$.*

$$\binom{n}{k} p^k q^{n-k} \simeq \frac{1}{\sqrt{2\pi npq}}\, e^{-(k-np)^2/(2npq)}, \quad p+q=1,\ p>0,\ q>0. \tag{2.37}$$

## 2.10 Pearson's Chi-Squared test (P$\chi^2$t)

Pearson's $\chi^2$ statistical test is the best-know of several chi-squared test, that are statistical procedures whose results are evaluated by reference to an ideal **Chi-Squared PDF** with $k$ **degrees of freedom.**

**Definition 2.18.** Chi-Squared PDF with k degrees of freedom.

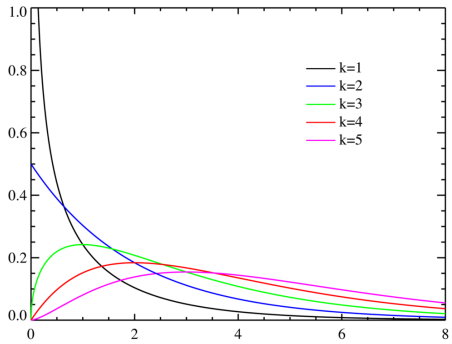
$$\chi^2(k) = \frac{1}{2^{\frac{k}{2}}\Gamma(\frac{k}{2})} x^{(\frac{k}{2}-1)} e^{-\frac{x}{2}}. \tag{2.38}$$

where $\Gamma(n)$ is the **gamma function**[5]

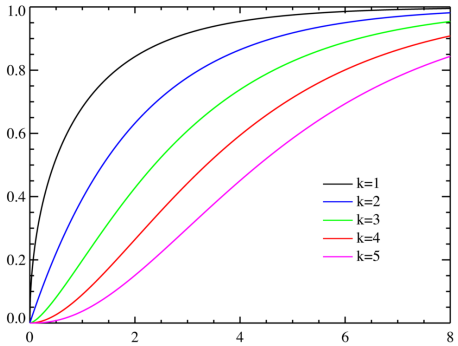
$$\Gamma(z) = \int_0^\infty t^{(z-1)} e^{-t} dt. \tag{2.39}$$

In Fig. 2.7 it is plotted $\chi^2$ PDFs in function of $k$:

[5] not be confused with *gamma PDF*. gamma PDF is a generalization of the chi-squared PDF, or in other term chi-squared PDF is a special case of gamma PDF




**(a) Chi-squared PDFs** in function of some $k$ degrees of freedom.




**(b) Chi-squared CDFs** in function of some $k$ degrees of freedom.

**Figure 2.7:** $\chi^2$ in function of some $k$ degrees of freedom.

Notice that $\chi^2(k)$ is the distribution of a sum of the squares of $k$ *independent Gaussian* RVs.[6]

Generally P$\chi^2$t tests a **null hypothesis** ($H_0$) on an "observed event". In the specific it tests whether outcome frequencies (e.g., number of logical '0' vs logical '1' in a bit sequence follow a specified distribution, also called *goodness-of-fit* testing. This fact is coded in the similarity between the sampled PDF and an ideal chi-squared PDF. The level of similarity reject or not the null hypothesis. Notice that exists also the *alternative hypothesis* associated to the null one, defined as $H_a$.

**Theorem 2.16.** ***Pearson's chi-squared test.*** *Given a null hypothesis $H_0$ and $n$ binomially samples to test it (goodness of fit test) it is possible to compute an observed statistic on it:$\chi^2_{obs}$. If $n$ is large enough, the comparison between $\chi^2_{obs}$ and a ideal chi-squared statistics $\chi^2$ with $k$ degrees of freedom provides a measure of accuracy of $H_0$ over $n$ samples (P-value).*

The application of this kind of test in the filed of TRNG will be extensively explained in sec. 5.2.

[6] called also *normal* or *standard* RVs for their huge use in statistics

# 3 Architecture & Design of RNG

## 3.1 Analog Chaotic ADC

The first TRNG proposed is a mixed non-linear circuit based on an *analog* ES circuit. The implementation refers to [25, 6, 26].

### 3.1.1 Pipeline ADC Converter

The implementation is based on a pipeline ADC converter. The $h+1$ stage converter shown in Fig. 3.1, provides a representation of an input variable $v_{in}$ defined in a interval $X$ into a $l$-bits numerical notation. It's not necessary that all stages are identical but in this example the first $h$ stages are $m$-bit ADCs and they are identical, while the last one, a $n$-bit ADC, has a simpler structure because it's not necessary the computation of its conversion error.

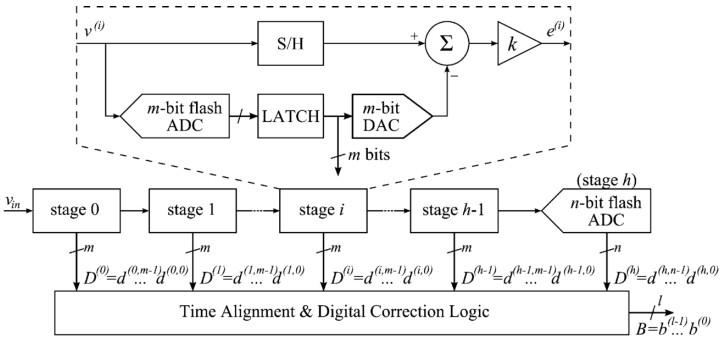



**Figure 3.1:** Basic Structure of a Pipeline ADC.

The input $v^{(0)}$ is sampled by the Sample/Hold (S/H) circuit of the first stage (stage 0), then the sample is passed to a $m$-bit flash ADC converter that computes a *coarse* $m$-bit representation of the input called $D^{(0)}$ and the analog *error conversion* $e^{(0)}$. This error signal is passed to the following stage (stage 1) as input $v^{(1)}$and the same computations are performed. This process is repeated from stage 0 to stage $h$ and provides $h+1$ bits. Observe that, apart from the first stage, which directly provides a coarse digital version of $v_0$, all other stages provide coarse representations of the intermediate conversion errors and not of the whole converter input. Hence the digital output of the various stages must be processed by some *digital correction logic* that provides the digital output $b^{(l-1)}...b^{(0)}$, with $l \leq h * m + n$, that represent the conversion from the analog signal $v^{(0)}$. In addition, the conversion errors

are necessarily *signed quantities* so that all stages in the chain must be capable of converting signed values. Finally, note that the conversion error $e_i$ of the generic $i$-th stage is always smaller than than its input $v_i$, so it must be rescaled in order to obtain identical stages.

The major advantage of a pipeline ADCs is that the various stages can be separated by *sample and hold* (S/H) blocks. This permits to synchronize the flow of information among all the stages. A stage is thus free to start operating on the next piece of data as soon as it has calculated its conversion error, increasing the overall throughput of the ES. However, this also complicates digital correction logic, which now has to be *sequential* (have memory).

### 3.1.2 Modified Single ADC Cell

Assume to take a single stage of the pipeline and to feed back the output to the input as showed in Fig. 3.2.

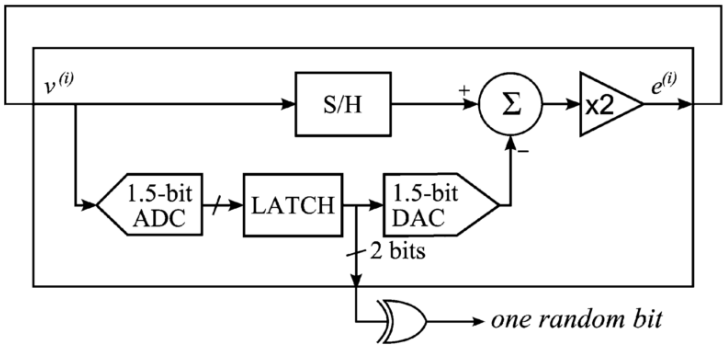



**Figure 3.2:** Modified (feedback added) ADC cell.

Now, dimension the circuit with the following parameters: $m = 1^1/_2$ bit and $X = [-1, 1]$ obtaining a normalized voltage input. Now it is possible to define the A/D *conversion function* $Q(x)$, see Eq. Eq. (3.1):

$$Q(x) = \begin{cases} -1, & \text{for } x < -\frac{1}{2} \\ 0, & \text{for } -\frac{1}{2} \leq x < \frac{1}{2} \\ 1, & \text{for } x \geq \frac{1}{2}. \end{cases} \tag{3.1}$$

So the conversion is obtained by comparing $v^{(i)}$ with the two values $\pm 1/2$ by means of two comparators, whose output is

$$D^{(i)} = d^{(i,1)}d^{(i,0)} = \begin{cases} 00, & \text{for } v^{(i)} < -\frac{1}{2} \\ 01, & \text{for } -\frac{1}{2} \leq v^{(i)} < \frac{1}{2} \\ 11, & \text{for } v^{(i)} \geq \frac{1}{2}. \end{cases} \tag{3.2}$$

Therefore the conversion error is:

$$e^{(i)} = k\left(v^{(i)} - Q(v^{(i)})\right) = M\left(v^{(i)}\right).$$

where $v^{(i)}$ spans in $X = [-1, 1]$; then $e^{(i)}$ spans in $[-k/2, k/2]$ so, to take full advantage of this architecture, the gain of the rescaler is set equal to two: $k = 2$. The generic $i$-th ADC stage is closed into a loop by imposing $v^{(i)} = e^{(i)}$, in this way, due to the intrinsic delay introduced by the S/H circuit, a discrete-time system is formed and the variable $v^{(i)}$ represent its state, and whose evolution is regulated by

$$v_{j+1}^{(i)} = M\left(v_j^{(i)}\right). \tag{3.3}$$

where the function $M$ is defined as $e^{(i)} = M\left(v^{(i)}\right)$ and $j$ is the time step.

The fundamental assumption made for this kind of design is that the *thermal noise* sets the initial conditions on the circuit, acting as an ES, and the quantization process, that is *not* reversible, acts as an entropy harvester. So it is the extreme sensitivity to the initial condition that lead to a long-term unpredictability and two slightly different initial condition diverge very fast in time. Because of ADC, an opponent cannot retrieve any kind of information observing the output of the system. Furthermore the same circuit acts both as ES as a EH, this is a great advantage because the harvester parasitics are included and compensated in the circuit at design time, so an ideal *maximum entropy transfer* is possible and also a less complex implementation of the DPP module i.e., XOR corrector instead of AES block. Here you can see also its circuital CMOS implementation, in Fig. 3.3:

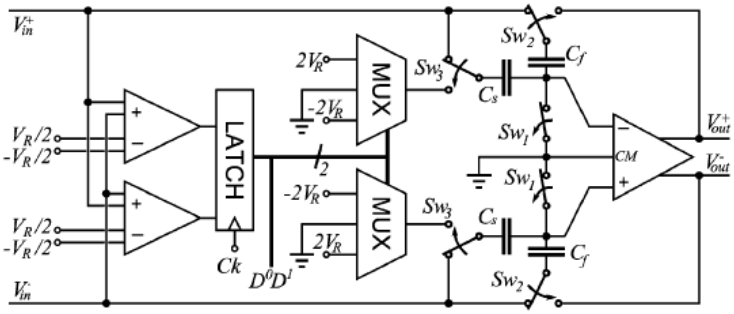



**Figure 3.3:** ADC cell Circuital Implementation.

For reasons that will be more clear in sec. 4.1, the system can be statistically modeled with is a one-dimension discrete-time *chaotic* map, by the function $M$ Eq. Eq. (3.3).

### 3.1.3 Chaotic ADC Pipeline

To complete the design the entire pipeline can be closed into a loop after the $h$ identical stages.

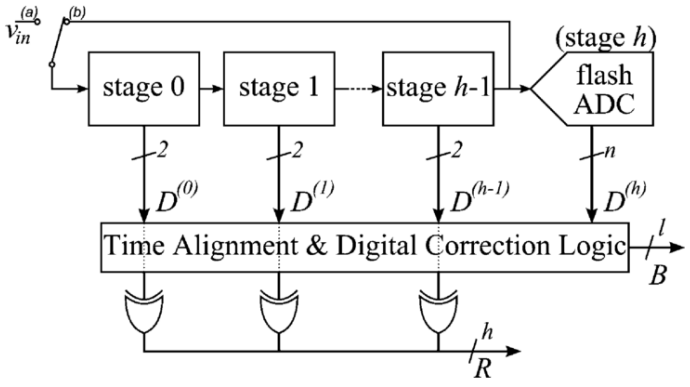



**Figure 3.4:** ADC $h$ stages loop configuration.

As showed in Fig. 3.4, when the switch is in position $(a)$ the circuit operates as a pipeline A/D converter, and the digital correction logic provides a $l$-bit conversion word $B$, while if the switch is in position $(b)$ the closed loop pipeline generates a bit pair for each stage, then a XOR battery is used as simple DPP stage sec. 1.2.3.1. So the throughput is halved and the system each clock cycle generates a $h$-bit random word $R$. The evolution of the system can be mathematically expressed by the following functions:

$$v_{k+1}^{(0)} = M^{(0)}\left(v_k^{(h-1)}\right)$$

$$v_{k+1}^{(1)} = M^{(1)}\left(v_k^{(0)}\right)$$

...

$$v_{k+1}^{(h-1)} = M^{(h-1)}\left(v_k^{(h-2)}\right). \tag{3.4}$$

These equations shows that Eq. Eq. (3.4) is equivalent to $h$ systems (Eq. Eq. (3.3)) running simultaneously. From the single cell implementation the system gain a *throughput* increasing factor of $h$ and the possibility to implement the RNG slightly modifying a normal pipeline ADC (reusability).

### 3.1.4 Chaotic ADC Summary

The main advantages of this TRNG architecture are:

- Reconfigurable Device: TRNG and ADC;
- Reusability/Embeddability: easy ti modify existing ADC circuit.
- Scalable Design: ADC has historically seen many years of refinement. The process of coding designs in re-usable forms is more advanced than for any other complex analog subsystem reaching an high level of design automation (CAD/EDA tools):

- Low Cost Design: no dedicated and/or specialized hardware needed;
- ES Robustness's: no need of too much complex DPP stage;
- Tamper resistant: a power analysis cannot leak information about the generated bits.

## 3.2 Digital Metastable Power-Up SRAM

The second TRNG proposed is *fully digital.* The ES exploits randomness in *volatile memory circuit,* and refers to a work that in principle was targeted for Radio Frequency IDentification (RFID) [12] and then it was discovered and explained also the RNG functionality [13].

The ES architecture uses *volatile* CMOS Static Random Access Memory (SRAM) cells as a Fingerprint Extractor and Random Number Generator, the method is called for convenience *FERNS*. In the specific this implementation uses the high gain of *metastable* cross-coupled CMOS inverters as a mechanism to produce (ES) and detect (EH) thermal noise, that is the (true) source of randomness. So this generator is classified as TRNG. It will be proved that the *power-up* of SRAM reveals a physical fingerprint of the chip and that this fingerprint can provide, under some hypothesis discussed in detail in sec. 4.2, identification and TRNG at low hardware cost, even in application lacking circuits dedicated to either purpose.

### 3.2.1 Static Random Access Memory (SRAM) Circuit

A Static Random Access Memory stores some Bytes till is power down, the systems is composed by the repetition of the standard SRAM cell, that it is a circuit able to store 1 bit Fig. 3.5.

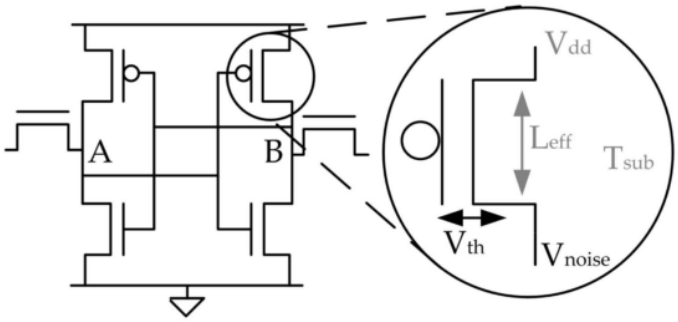



**Figure 3.5:** SRAM cell with relevant process variation and noise shown.

.

Each bit of SRAM is a six-transistor memory cell, consisting of two cross-coupled CMOS inverters and two access transistors. Each of the inverters drives one of the two state nodes, labeled $A$ and $B$. When the circuit is unpowered both state nodes are discharged low ($AB = 0$). Each time the circuit is *powered-up*, this unstable state will transition to one of the two stable states, either logic 0 ($AB = 01$) or logic 1 ($AB = 10$); the $AB = 11$ state is unstable and unreachable. The tendency to transition to one state or the other depends on *process variation* mismatch and *noise.* The impact of common-mode process variation, such as lithography, and

common-mode noise sources, such as supply fluctuations, are minimized because the stabilization of each cell depends only on differences between local devices, see sec. 4.2

### 3.2.2 Skews of SRAM cells

It can be convenient, for illustrative purposes, to define the **skew** of a cell as a continuous quantity used to represent the power-up tendency of a cell. Skew at a given power-up is influenced by noise, so the skew of each cell across many power-ups is described by a PDF. There are two classes and three possible cases of skews, see Fig. 3.6:

1. The first class is represented by that cells that with high probability (regardless of noise), will power-up to the 0 state or 1 state. They can be defined as *0-skewed* or *1-skewed* cells, see Fig. 3.6(a).

2. The second class is represented by those cells who does not have a strong tendency toward either states. They can be defined as *neutral-skewed* cell, see Fig. 3.6(b).

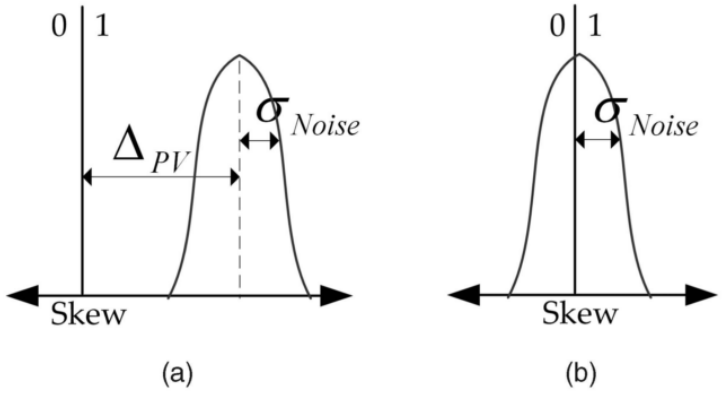



**Figure 3.6:** (a) Tendencies of a 1-skewed cell. (b) Tendencies of a neutral-skewed cell.

It's important to underline that a neutral-skewed cell does not necessarily consist of perfectly matched devices, but instead has some unknowable combination of variations that are approximately offsetting when power-up under nominal conditions. This distinction is significant as it indicates that such a cell may not remain neutral across all operating conditions. If a cell is strongly 0-skewed or 1-skewed, the minor influence of noise is insufficient to sway power-up state; such cell provide *identification*. If a cell is neutral-skewed, the influence of noise can determine its power-up state; such cells provide *randomness*.

### 3.2.3 FERNS method: SRAM fingerprints

Given an SRAM *array*, the power-up state generated by its constituent cells is defined as a *physical fingerprint* of that array. Some of the cells in the array are neutral-skewed, so they are unreliable across power-up trials because they add randomness to the physical fingerprint, but other cells are 0-skewed or 1-skewed acting as reliable identifying features of a fingerprint. Remember also that cells skew is not correlated to the same bits on different chips.

A **latent fingerprint** is an SRAM fingerprint produced at a single power-up. With $l(i)$ denoting the state of a single SRAM cell at power-up $i$, an $N$-bit latent fingerprint is simply the collective state of a specified set of $N$ cells at power up $i$:

$$L_C = \{l_0(i), l_1(i), ..., l_N(i)\}\,. \tag{3.5}$$

As a latent fingerprint is sensitive to noise, and some bit will not power-up the their most probable state, the *same* set of SRAM cells can produce many different latent fingerprint.

A **known fingerprint** is an intentional estimation of the state that a *given* set of SRAM cells is most likely to generate at power-up and this estimation is used as the known identity of a chip. The most likely power-up state of *each* cell is determined by *averaging* across an odd number or trials

$$\mathrm{p} = \mathrm{avg}\,(l(i))\,, \quad \forall i. \tag{3.6}$$

and *rounding* to a binary value $k$

$$k = \begin{cases} 0 & \text{if } \mathrm{p} < 0.5 \\ 1 & \text{if } \mathrm{p} > 0.5 \end{cases}. \tag{3.7}$$

Averaging over multiple power-ups reduces the impact of the noise, making a known fingerprint more representative of the SRAM cells that generate it than a latent fingerprint from the same cells:

$$K_C = \{k_0, k_1, ..., k_N\}\,. \tag{3.8}$$

The difference between latent and known fingerprints imply their usage in the FERNS method of identification and random number generation. Identification is enabled by the similarity between known and latent fingerprints when both are generated by the same chip, compared to the lack of similarity between those generated by different chips. *TRNG* is possible because the minor differences between latent fingerprints generated by the same chip cause large latent fingerprint to be effectively unique.

### 3.2.4 TRNG in SRAM

In the classic metastable realization, because the metastable point is not static in time, some *calibration* circuit is used for dynamic control and time synchronization. This leads to the guaranty that the bits produced are determined “only" by

thermal noise, but the drawback of having additional circuitry provides some extra area/power consumption, that is a very strict requirements in the RFID design (ultra-low power).

FERNS is a sort of "imprecise" version of the classical metastable design because it uses massive *redundancy* that compensates the imprecision and randomness is scattered throughout the SRAM. So no feedback or control logic is required, because there is no need to precisely bias a single cross-coupled cell to perfect metastability. Instead, FERNS relies on the *large* number of cells to ensure that some cells will be influenced by noise when the chip is powered-up, without giving concern to which cells are generating randomness because different conditions of power-up lead to different random cell. Fig. 3.7

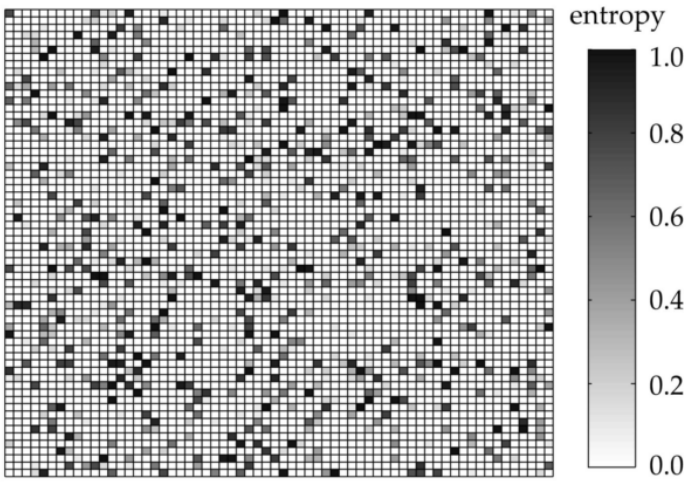



**Figure 3.7:** Shaded dark SRAM cells are those used for RNG, they present unpredictable power-up state.

The privacy amplification (DPP) is performed by hashing a 512-byte fingerprint into a digest of 128-bits sec. 1.2.3.3, using the *PH universal hash function* [35], with each block of message and key comprised of the power-up state of 64-bits of SRAM:

$$PH_K(M) = \sum_{i=i}^{16} (m_{2i-1} + k_{2i-1})\,(m_{2i} + k_{2i})\,. \tag{3.9}$$

$$M = (m_1, ..., m_{32}) \qquad K = (k_1, ..., k_{32})\,. \tag{3.10}$$

$$m_i, k_i \in GF(2). \tag{3.11}$$

PH is designed for low gate count and low-power hardware implementation (only 557 cells needed), with all operations performed over GF(2), so that addition and multiplication reduce to a series of shift and XOR operations.

It is important to notice that this kind of TRNG provides random bits and the physical fingerprints only during power-up state, that is in contrast against the unbounded entropy generation potential of other kinds of TRNG. For this reason, the FERNS method is best suited for application that are *intermittently powered* and do not require large quantities of random number such as contact-less credit cards and peacemakers.

### 3.2.5 Metastable SRAM Summary

The main advantages of this TRNG architecture are:

- Area/Power consumption: imprecision of metastable point compensated by the redundancy of SRAM cells;
- Low Cost Design: no control logic and feedback circuit needed;
- Reconfigurable Device: TRNG and RFID;
- Resiliency against external influence: random bit source are not bounded to a specific portion of the chip;
- Embeddability/Reusablility Design: easy use existing SRAM implementation.

## 3.3 Intel® Ivy Bridge Core™ Bull Mountain DRNG

The first Intel RNG, patented in 1998-2000 [10, 11], was well studied when the company introduced Firmware-Hub chip-set component. This is a ring-oscillator based analog design where intrinsic thermal noise was taken, amplified, and used to pilot a free running oscillator, for further details see the white paper prepared for Intel Corporation by Cryptography Research, Inc. (CRI) [16].

### 3.3.1 Intel Bull Mountain DRNG

The 3rd Intel Core family processor @22$nm$, code-named *Ivy Bridge,* presents an hardware dedicated module Digital Random Number Generator (DRNG) aka *Bull Mountain* [20]. This module is capable to communicate with all the cores of the chip with a new atomic instruction **RDRAND** accessible by direct call from the Operating System or specific application.

The high level schematics in Fig. 3.8 contains the typical macro-blocks of a TRNG. The first block produces (ES) and digitizes (EH) random bit using an asynchronous production pipeline. The second block (DPP) presents a synchronous conditioning part and a Deterministic Random Bit Generator (DRBG). The system synthetically operates as follows:

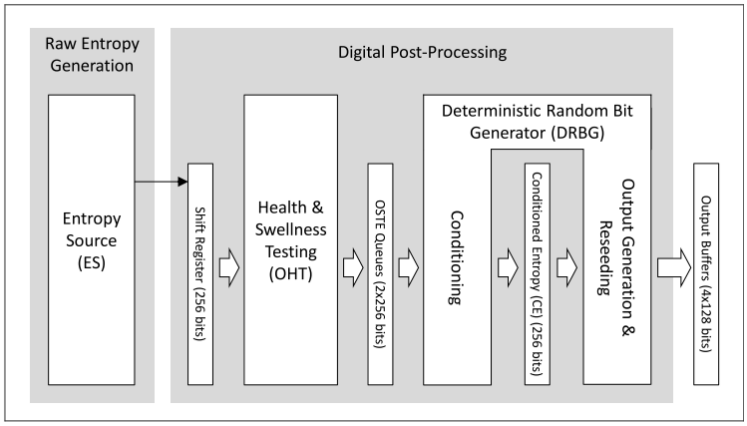



**Figure 3.8:** Block Diagram of the Intel RNG.

1. The ES, that is a self-clocking circuit, operates asynchronously and generates random bits at a high rate (about 3 GHz);

2. Random bit generated by the ES are combined, sampled by the synchronous logic, and grouped into 256-bit blocks in a shift register;

3. Basic statistical test are performed by the so called *Online Health Test (OHT)* unit on each 256-bit block to check for potential failure mode of the ES;

4. The 256-bit block are then passed to the so called *Online Self-Tested Entropy (OSTE)* queue and then they are cryptographically processed into a 256-bit *Conditioned Entropy Pool (CEP)* buffer by the conditioning logic;

5. The CE pool is used to reseed the DRBG that generates the random output bit.

### 3.3.2 ES and Health and Swellness Tests

The ES is a *dedicated* self-oscillating metastable circuit, with feedback and control logic in contrast with the “imprecise" metastable solution proposed in sec. 3.2.

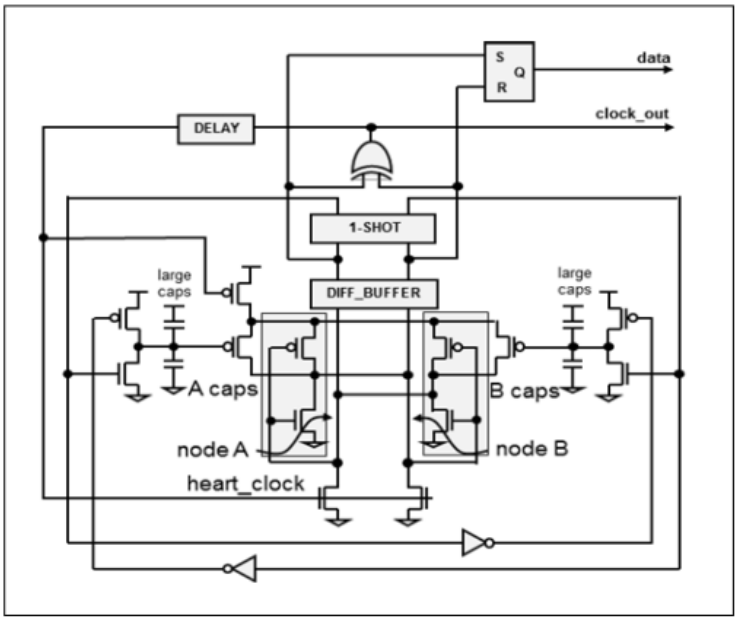



**Figure 3.9:** Entropy Source of the Intel RNG.

The circuit, showed in Fig. 3.9, it is a dual differential (jamb) latch formed by two cross-coupled inverters, the nodes A and B. The circuit is self-clocking and once it is started, enters in a metastable state. Then the ES resolves to one of the possible two states, according to thermal noise randomness, producing in output a random bit. The settling of the circuit is *biased* by the differential in the charges on the capacitors "A caps" and "B caps". The calibration process is based on how the latch resolves: namely a fix amount of charge is drained from one capacitor and added to the other with respect to the last output. So the feedback logic is designed to seek out the metastability, trying to leave the latch oscillating around the metastable region, using the last output as a control signal to determine the charge changes to the capacitors. At normal *Process-Voltage-Temperature (PVT)* conditions, the ES works as a digital noise sensor @ 3 GHz with a throughput of *2.5-3 Gbps*.

Unlike the design proposed in [16], this RNG provides two option after the ES's random bits generation:

1. *Accumulate* samples coming from ES output in single-bit buffers using a XOR circuits *before* the under-sampling by the synchronous region. So at each clock cycle of the asynch region, the bits stored in the buffers are bit-wise summed (XORed) with the next bits coming from the ES, providing a sort of pre-DPP. The buffer is then sampled by the synchronous logic;

2. *Overwrite* the buffer with each new ES output. In this case some output bits are cannot be used.[1]

[1]Future versions of the RNG will use a different synchronization logic, the ES output will be *deserialized*, and then sampled in parallel into synchronous region, thereby preserving all the ES samples for post-processing.

The data sampled into the synch region is passed serially to the OHT unit that performs the so called *"Health & Swellness Tests"*. It evaluates the health of each 256-bit sample using an *heuristic* law choosen by Intel. This *on-line* testing approach is not exhaustive from a statistical testing point of view, details are in Eq. (chapter 5). It simply counts how many times each of the six different bit patterns appears in a 256-bit sample. The sample is considered **healthy** iff the number of times each pattern appears, denoted with $n$, falls within certain bounds; see Tab. 3.1.

**Table 3.1:** Health Bounds for 256-bit samples.

| Bit Pattern | Bounds per 256-bit sample |
|---|---|
| 1 | $109 < n < 165$ |
| 01 | $46 < n < 84$ |
| 010 | $8 < n < 58$ |
| 0110 | $2 < n < 35$ |
| 101 | $8 < n < 58$ |
| 1001 | $2 < n < 35$ |

These bounds are chosen empirically and the probability $\alpha$ of a false positive (probability that a random sample from a uniform distribution fails) is about 1%. It is important to stress that the OHT unit is *not* intended as a measure of entropy, but its task is to check if the ES is badly broken, e.g., it provides continuously simple repeating patterns, such as all zeros.

The OHT unit track also the *health status* of the most recent 256 samples, (each sample is 256-bit). The ES is considered to be **swell** iff at least 128 of the most recent 256 (256-bit) samples are healthy. The on-line tested series of bits fills the first 256-bit OSTE buffer, when it is full all bits are shifted *in parallel* to the second OSTE buffer and then passed to the DRBG subsystem.

### 3.3.3 DRBG: AES CBC-MAC

The DRBG accumulates all the samples, even those not healthy, into a so called *"Conditioned Entropy Pool" (CEP)* buffer, defined also as CE[255 : 0]. The CEP buffer can store 256 random bit and it is used to reseed the DRBG. For security reason, this buffer is formed by two independent parts: the lower and upper parts and these two sub-buffers are updated sequentially and independently.

CE[127 : 0] First the lower half of the CEP buffer is updated by processing the current 256 bits in OSTE with a AES CBC-MAC algorithm [7], using 128-bit non-secret fixed key K' that is identical in all chip. Below is showed the pseudo-code:

**Algorithm 3.1** `CE[127 : 0] Updating process.`

```
Temp[127:0]=AES(K',CE[127:0]);
Temp[127:0]=AES(K',OSTE[127:0] XOR Temp[127:0]);
CE[127:0]=AES(K',OSTE[255:128] XOR Temp).
```

This process is repeated, with new data in the OSTE buffer each time, if the OSTE buffer in not healthy or if the ES is not in a swell state.

CE[255 : 128] Then, the upper half of the CEP buffer is updated using the same process, with the same pseudo-code and same loop criterion and always using *fresh* value from OSTE buffer.

Each update process is designed for fully randomize half of the CEP, even if the OSTE bits are partially random. The complete CEP buffer updating process is showed in Fig. 3.10.

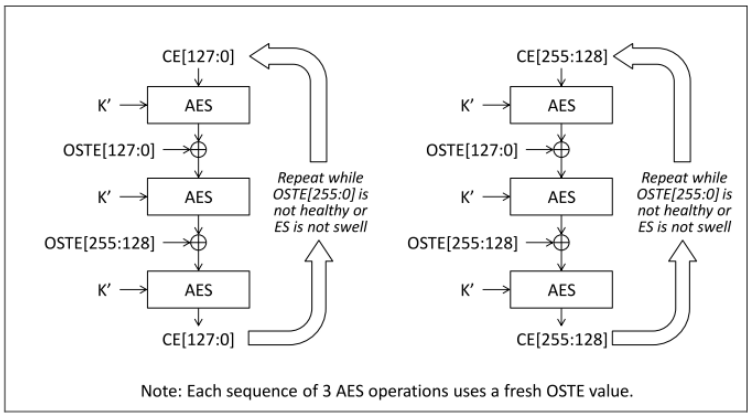



**Figure 3.10:** Left side: CE[127 : 0] updating process. Right side: CE[255 : 128] updating process.

CEP bits are then used in the *reseeding* process, the lower half CE[127 : 0] are used for the DRBG *AES key* K[127 : 0], that is different from the key K' used in the conditioning process in Fig. 3.10. The upper half CE[255 : 128] is used for the DRBG *counter* V[127 : 0]. Below is showed the pseudo-code of the reseeding process:

**Algorithm 3.2** `DRBG Reseeding process.`

```
\\ K is the DRBG key, V is the 128-bit counter and C the
\\ number of outputs produced since the last reseeding
\\ all three quantities are initialized to zero at reset
V[15:0]=(V[15:0]+1)mod 65536;
Temp=AES(K,V);
V[15:0]=(V[15:0]+1)mod 65536;
V=CE[255:128] XOR AES(K,V);
K=CE[127:0] XOR Temp;
C=0;
```

Then the random data are generated using the counter mode CTR_DRBG construction, with AES-128 as the block cipher. First, it fills four 128-bit output buffers with data, and then it generates additional outputs as needed. Below is showed the generation process:

**Algorithm 3.3** `Random bit generation process.`

```
V[15:0]=(V[15:0]+1)mod 65536;
C=C+1;
Output=AES(K,V);
If(update needed) {
 V[15:0]=(V[15:0]+1)mod 65536;
 Temp=AES(K,V);
 V[15:0]=(V[15:0]+1)mod 65536;
 V=AES(K,V);
 K=Temp;
}
```

DRBG requires new 256-bits seed after at least 512 output samples of 128-bits (a total of 65536 bits), but under normal operation it will reseed more frequently. Intel's simulations suggest that as long as the ES is healthy, the generator will reseed within 22 output samples of 128-bit even under heavy load and under light or moderate load, it will reseed before every 128-bit output. Regarding standards, the DRBG reseeds much more often than NIST SP 800-90A [3] requires.

### 3.3.4 BIST and Operating Modes

Upon reset, the DRBG first performs a *Built-In-Self-Test (BIST)* to verify that the system works properly and to initialize the DRBG. BIST is conducted in two phases:

**Phase 1** ES is disconnected and a *Linear Feedback Shift Register (LFSR)* generates a deterministic sequence of bits to test the OHT unit and DRBG circuit. A 32-bit *Cyclic Redundancy Check (CRC)* is computed on the output of the DRBG and compared against an hardwired expected value.

**Phase 2** ES is reconnected and used to generate 256 samples of 256-bits. The sample are used to condition the 256-bit CEP buffer. As before, first the lower half CE[127 : 0] is conditioned till 128 *healthy* samples have been processed. At this point the ES is considered to be *swell* and CE[127 : 0] is marked as available. The upper half CE[255 : 128]is then conditioned, and is marked as available as soon as one sample has been processed while the ES is swell. Once that both half of the CEP buffer are filled, the DRBG can be reseeded so key value (K) and counter value (C) are initialized. Finally four 128-bit samples fill the output buffer.

The RNG *must* pass both phases of BIST to be considered working correctly. If not the RNG will produce no data and the dedicated instruction to access to fresh random data (RDRAND) will return all zeros and *clear* the **carry flag**, indicating that there is a problem.

Instead after the end of a successful BIST, the RNG is ready for *normal operation*, debug and test port are disabled and the RNG operates autonomously. Each

RDRAND call to the RNG returns 64-bits from the output buffer, which is refilled as necessary by the RNG, and a carry flag equal to 1 to indicate good status. Users should check the carry flag after each RDRAND call. Intel claims that even under heavy load @ 800 MHz, the RNG can deliver 800 MBytes/sec post-processed random data.

The Intel RNG supports *eight* different operational modes natively, such as debug and test modes, however most of them are disabled on production parts. In addition to operational modes, the RNG supports also a *FIPS* mode, which can be enabled and disabled independently of the operational modes. FIPS mode, that refers to *Federal Information Processing Standard*, sets additional restrictions on how the RNG operates and is intended *to facilitate* **FIPS-140 certification** [24]. This section and the consequent analysis in sec. 4.3 is only concerned with the behavior of the system in *normal* mode.

### 3.3.5 Intel Bull Mountain Summary

The main features of this TRNG architecture are:

- Throughput/Bandwidth/Entropy Rate: 800 MB/s;
- Hybrid design: try to merge TRNG and PRNG advantages;
- Specification/Certification: CSPRNG, FIPS-140;
- Security Level: hardware dedicated module;
- Granularity: accessible at all system levels.

## 3.4 Two Open Source TRNG

### 3.4.1 HotBits

There are at least two interesting examples of *free* TRNG in the web. The first one is called *HotBits* generator (third-generation).[2] It uses a commercial Geiger-Müller detector, designed for attachment to a computer's serial port, illuminated by a 5 microcurie Cæsium-137 check source. It is an internet resource that brings *genuine* random numbers, generated by a process that basically is governed by the inherent uncertainty in the quantum mechanical laws of nature. It is possible to use these random bits directly from a computer, that acts as an interface from the ES. The random HotBits are generated by timing successive pairs of radioactive decays

[2] `http://www.fourmilab.ch/hotbits/`

detected by a Geiger-Müller tube. Directly from the Web you can order up your serving of HotBits by filling out a request form specifying how many random bytes you want and in which format you'd like them delivered. The data-rate is modest (about 100 Bytes/s) but is free and can be used as seed for some PRNG.

### 3.4.2 LavaRnd™

The second example is *LavaRnd™* a cryptographically sound random number generator. [3]At its heart, it uses a chaotic source to power the generation of very high quality random numbers. Anyone can have their own LavaRnd because its source code is *open* and related algorithms have been released into the public domain. The reference implementation uses low cost consumer parts. LavaRnd turns real world physical chaotic events into random numbers in 3 stages. Firstly a digital snapshot of a physical chaotic process is obtained. Any chaotic source that is sensitive to measurement errors can be used. The digital snapshot containing both structured data and chaotic noise is run through a Digital Blender Algorithm. The combination of n different SHA-1 cryptographic hash operations running in parallel, (discussed in sec. 1.2.3.3) and n different xor-rotate and fold operations on data containing some chaotic noise destroys the structured data portion of the digital snapshot and produces uniformly distributed random data. The uniformly distributed random data is collected into a pool and used only once to produce random values in the form required by the application.

## 3.5 PRNG

### 3.5.1 Wolfram's Rule 30

The genius S. Wolfram, who had invented Wolfram Alpha, Mathematica and New Kind of Science (NKS) introduces in 1983 a particular type of one-dimensional Cellular Automata (CA) called *Rule 30* [34], that can be used as a PRNG. In fact Rule 30 is used as a deterministic random number generator in Mathematica. Notice that this section has an illustrative purpose only, all the theory behind this is out of the scope of this work. CA were introduced by von Neumann and Ulam, under the name of *cellular spaces* with the purpose of modeling biological self-reproduction systems.

**Definition 3.1. Cellular Automata (CA).** Mathematical idealizations of physical systems in which space and time are *discrete*, and physical quantities take on

[3]`http://www.lavarnd.org/`

a finite set of discrete values. It is a completely *deterministic* system because CA discrete-time evolution of each cell value at time step $t$, is dictated by the values of the variables at sites in the "neighborhood" on the time step $t-1$. Thus there is a precise set of local rules that updates synchronously every cell.

**Theorem 3.1.** *Any physical system satisfying differential equation may be approximated as a cellular automation by introducing finite differences and discrete variables.*

It is sufficient to consider one-dimensional CA, with two possible values of the variables at each state (logical 0s and logical 1s) and in which the neighborhood of a given state is simply the site itself and the sites immediately adjacent to it on the left and on the right. This kind of CA are called *elementary.* These kind of CA are classified in terms of rules, because it is possible to describe each one by $2^3 = 8$ binary digit and/or a decimal number btw 0 and 255, e.g. Rule 30 equals to rule 00011110.

The rules must be reflection symmetric, so that 100 and 001 yields to identical values and all the rules that ends with 0 in the binary representation are forbidden. Thus the set of possible rules reduces to **32** legal CA in the form:

$$\alpha_1\alpha_2\alpha_3\alpha_4\alpha_2\alpha_5\alpha_4 0. \tag{3.12}$$

Some elementary CA rules exhibit the important simplifying feature of *additive superposition* or *additivity* property. Evolution according to such rules satisfies the superposition principle:

$$s_0 = t_0 \oplus u_0 \iff s_n = t_n \oplus u_n. \tag{3.13}$$

Only rules 0, 90, 150 and 204 are of this form, where the first and the latter are trivial rules.There are also some elementary CA rules that are *peripheral,* in the sense that the value of a particular site depends on the values of its two neighbors at the previous time step, but not on its own previous state. Rules 0, 90, 160 and 250 exhibit this property.A configuration may be considered disordered or random, if values at different sites are statistically uncorrelated and thus behave as independent RVs. Such configurations represents a discrete form of *white noise.*

**Definition 3.2. Rule 30.** Chaotic and aperiodic configuration that can be seen as a Boolean function of the sites within the neighborhood, defined as

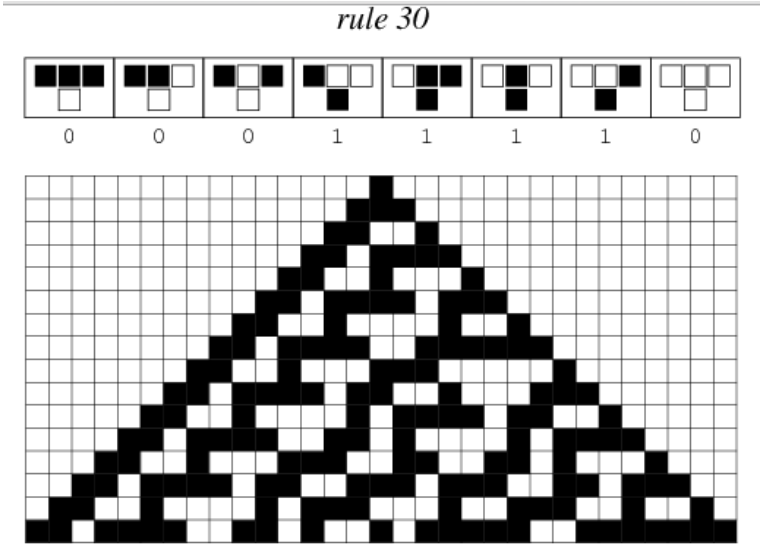
$$x(n+1,i) = x(n,i-1) \oplus [x(n,i) \cdot x(n,i+1)]\,. \tag{3.14}$$

where $n$ indicates the discrete time step and $i$ the cell position (central, left, right).

The rule set which governs the next state is showed in Tab. 3.2:

**Table 3.2:** Rule 30: sets of rules.

| **current pattern** | **111** | **110** | **101** | **100** | **011** | **010** | **001** | **000** |
|---|---|---|---|---|---|---|---|---|
| **new state for central cell** | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |




If the initial state is 1 (black color), you can notice the evolution of the system after some iteration in Fig. 3.11. The vertical axis represents time and any horizontal cross-section represent the state of the system at a given time. You can notice that the left and the right part of the triangle presents periodic pattern but the central column can be used as PRNG .

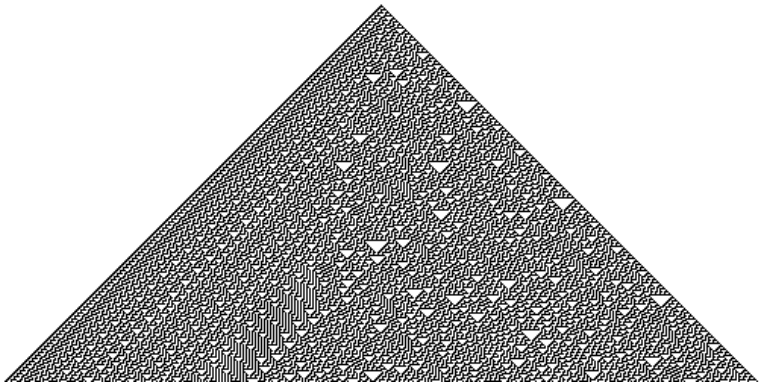


**Figure 3.11:** Rule 30 iteration.

As the RNG presented in sec. 3.1, Rule 30 displays sensitive dependence on initial conditions.

# 4 Theoretical Analysis of RNG

## 4.1 Analog Markov Chaotic sources and Pipeline ADCs

The first TRNG proposed is a mixed non-linear circuit based on an *analog* ES circuit. The analysis refers to the implementation proposed in [25, 6, 26].

### 4.1.1 Discrete-Time Chaotic Model

In general, a *chaotic* system is defined as a dynamical, linear or non linear element that can exhibit non-classical behaviors including very irregular, aperiodic, noise-like trajectories, and an extreme sensitivity to initial conditions, which can make two identical systems apparently starting at identical initial conditions, end up with totally different output. Notice that, the outputs of a chaotic source are generally unevenly distributed and always correlated because of the *deterministic* model responsible of their generation. Hence it's not possible to use them directly. Some signal processing is required to digitize, de-correlate and balancing the sequence like in all other kind of designs. Using some *compressive* DPP algorithms is possible, in principle, to obtain IID sequence. So a chaos-based RNG is no different from a physical RNG and suffers the same liabilities, such as the need for complex signal processing, possibly low output rates and, often, no formal guarantee that completely uncorrelated bits are produced in short term.

As showed in sec. 3.1, the system initial condition are set by intrinsic thermal noise, that acts as an ES and the quantization process act as a EHC and can be modeled as a discrete-time and chaotic. The word *chaos* encloses the main assumption of this approach: micro-cosmic processes employed in physical-RNG, like this one, can be (at least) in principle be expressed deterministically and that it is their extreme *complexity* and *sensitivity to initial conditions* to make them unpredictable to the coarse observer. Under these premises, rather than exploiting "natural" phenomena e.g. thermal noise model, that are well known but hardly controllable and mathematically unmanageable, it would be much better to rely on *artificial* ones that derive unpredictability from complexity. Discrete-time chaotic models exists and they extensively understood so their suitability for RNGs can be *analytically* and not just *heuristically* proven, but it's out of the scope of this thesis.

Intuitively speaking, the uncertainty due to the noise superimposed on the state of the circuit, will trigger the sensitive dependence on initial conditions resulting in a long term unpredictability. The non-reversible quantization (analog to digital)

process does not allow us to retrieve precise information on the actual evolution of the system from the observation of the output (quantized values). If one knows the model of the chaotic system, then he might look for an *informed* DPP algorithm. Even better, dealing with an "artificial" chaotic systems, one can think of co-designing the chaotic model and the post-processing algorithm to get the best possible features under some metric e.g., the output bit-rate. Clearly, all this requires specific mathematical tools to deal with chaos and its properties. As can be seen above, if one sufficiently restricts the range of chaotic systems being considered, many tools become available. In the specific, limiting the analysis to a very precise class of non linear models, represented by the iteration of so called **Piece-Wise Affine Markov (PWAM) maps**. The underlying idea stems from the realization that chaotic systems enjoy a mixed deterministic/stochastic nature and from the consequent adoption of statistical methodologies for their analysis [30].

### 4.1.2 Markov Chain characterization

A typical case of chaotic map is shown in Eq. (Fig. 4.1).

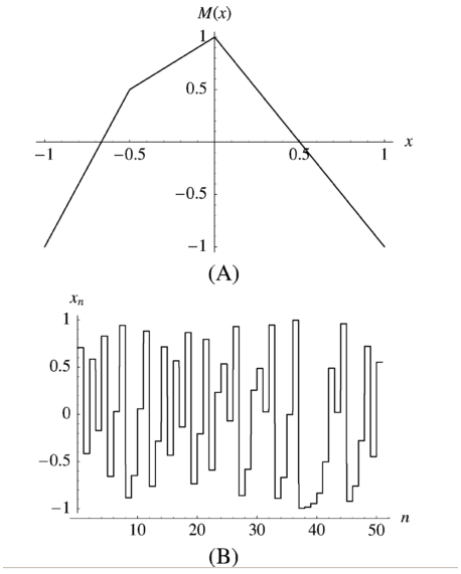



**Figure 4.1:** (A) Example of chaotic map. (B) Typical output trajectory.

Plot A shows a chaotic map, where

$$M(x) = \begin{cases} 3x + 2, & \text{for } x < -\frac{1}{2} \\ x + 1, & \text{for } -\frac{1}{2} \leq x < 0 \\ -2x + 1, & \text{elsewhere.} \end{cases} \tag{4.1}$$

and plot B a typical trajectory in time. As anticipated, if $M$ is chaotic, changing the initial condition $x_0$ even by extremely small quantity leads to a completely different

trajectory, which progressively (and exponentially) increases its distance from the former, as time step $n$ increases. The classical tools of system theory, which are generally based on the observation of system trajectories over time, are insufficient to get *typical* behaviors and general system properties; so it's necessary to introduce *statistical* tools. The intuitive idea is to pretend to be observing a huge number of trajectories at the same time. Suppose that a *very large* number of (different) initial conditions is generated, following a given PDF $\rho_0$ in $[-1, 1]$. When M is applied, an equally large number of $x_1$is obtained. These will distribute in $[-1, 1]$ according to a certain density $\rho_1$. If $M$ is known, it can be expected that $\rho_1$can be computed from $\rho_0$. In other words, one can introduce a *functional operator* $\mathbf{P}$ companion to $M$, called *Perron-Frobenious Operator*. While $M$ transforms points into points, $\mathbf{P}$ transforms probability densities into probability densities. Interestingly, in spite of $M$ being non linear, $\mathbf{P}$ is a linear operator, which makes it more manageable from a mathematical point of view.

In the analysis proposed, the initial probability density $\rho_0$ is stepwise in $\{X_i\}$, all subsequent probability density $\rho_1, ..., \rho_\infty$ are then compelled to be stepwise on the interval partition. So a finite dimension operator $K$ can be substituted for the functional operator $\mathbf{P}$. Being $K$ linear and finite-dimensional in can be represented by a matrix, named *kneading matrix.* It has been proven that the entries of $K$ can be obtained from $M$ as

$$K = \frac{\mu\left(X_i \cap M^{-1}(X_j)\right)}{\mu\left(X_i\right)}. \tag{4.2}$$

where $\mu$ is the common interval (Borel) measure, i.e., if $X_i = [a, b]$, then $\mu\left(X_i\right) = b - a$. For instance the kneading matrix correspondent to the map in Eq. (Fig. 4.1)(A) is

$$K = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{pmatrix}. \tag{4.3}$$

The entry $K_{i,j}$ of $K$ represents the conditioned probability by which a trajectory starting in $X_i$ falls in $X_j$ at the next step. The process can be iterated and can be seen in an infinite tree-graphs reported in Eq. (Fig. 4.2), by which the probability of finding $x_n$ in any of the partition intervals can easily been obtained.

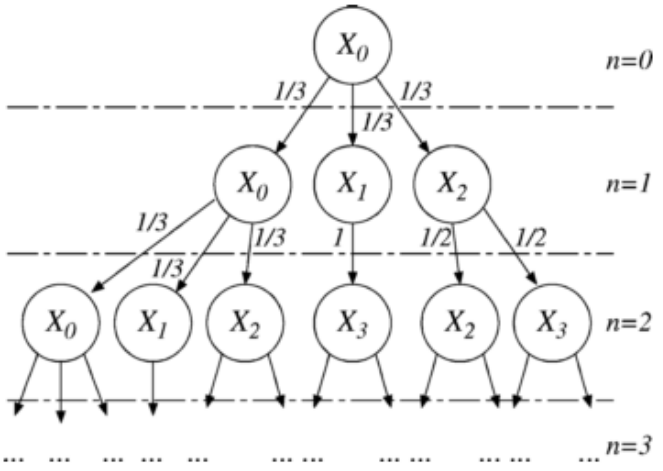



**Figure 4.2:** Probability tree based on the map in Eq. (Fig. 4.1)(A).

It's convenient to compact the tree-graphs such as in Eq. (Fig. 4.3).

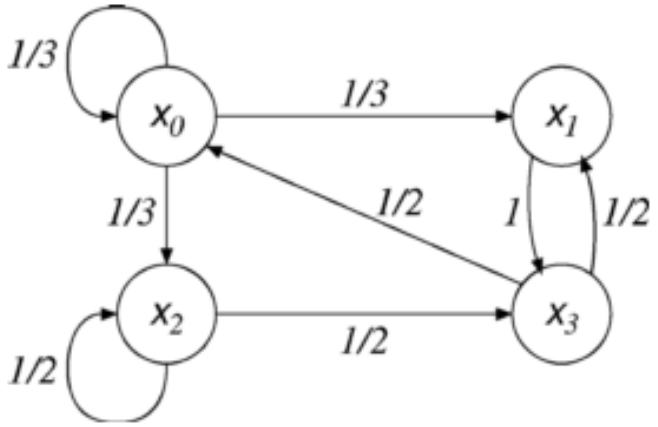



**Figure 4.3:** State chain relative to the map in Eq. (Fig. 4.1)(A).

Note that this graph can be interpreted as a *Markov chain* or a transition graph for a *probabilistic state machine.* The machine is in its discrete state $x_i$ when the chaotic system has its continuous state variable $x$ in the partition interval $X_i$. The weights assigned to the graph arrows represent the probabilities by which the machine travels from a state to another.

### 4.1.3 Ideal chaotic map: Bernoulli Shift

Now, it's interesting to understand how to design a chaotic source *optimized* for the generation of random bits. A common example starts from the Markov chain in Eq. (Fig. 4.4)

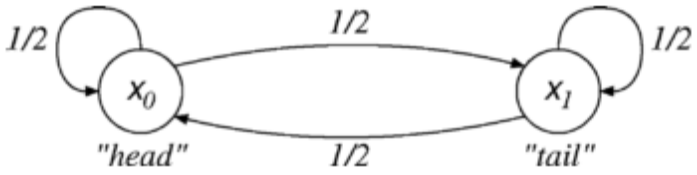



**Figure 4.4:** Markov chain of the fair toss.

This corresponds to the toss of an unbiased coin and is a process generating independently two symbols each with probability $p = 1/2$ so IID bits. Its correspondent kneading matrix Eq. Eq. (4.4) is a $2 \times 2$

$$K = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}. \tag{4.4}$$

Given this $K$, one can design a correspondent PWAM map. One alternative is the *Bernoulli shift*, where $M(x) = 2x \bmod 2 - 1$ which is pictured in Eq. (Fig. 4.5)

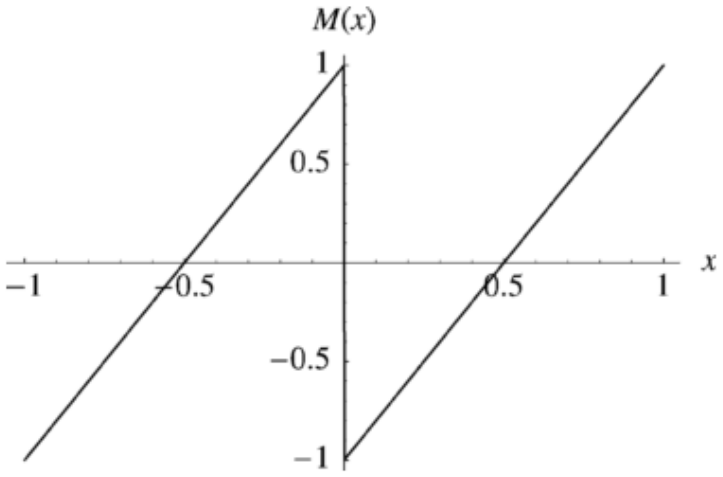



**Figure 4.5:** Bernoulli shift.

This map has only two interval partitions $X_0 = [-1, 0)$ and $X_1 = [0, 1]$. A logical 0 output generated when $x < 0$, and a logical 1 when $x \geq 0$. By construction, this ES is such that the observation of past sequences from the output cannot give *any* information about the future values. Since the probability of being in either of the two states is one-half, the single observation of this Markov chain gives 1 bit of information each time step, so one (maximum) bit of entropy per cycle. Unfortunately, the practical implementation of all the PWAM maps that comes out from kneading matrix shown in Eq. Eq. (4.4) are not well suited for electronic realization. The main problem is that such maps are unable to maintain the state confined into the interval $[-1, 1]$ in presence of noise or implementation errors.

### 4.1.4 Chaotic Model for ADC Pipeline

The implementation presented in [25, 6, 26] is based on a pipeline ADC designed to operate with *one bit and a half* stages as shown in Eq. Eq. (sec. 3.1). From this configuration can be derived for each building block Eq. Eq. (3.3) a new map shown in Eq. Eq. (Fig. 4.6)

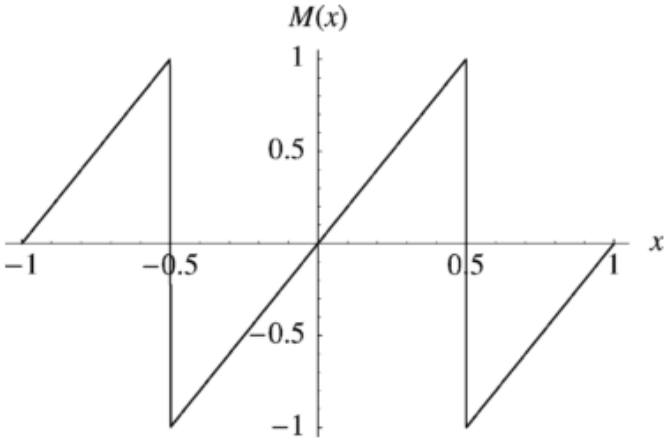



**Figure 4.6:** Map derived from a building block of one and a half bit per stage pipeline ADC .

This map is a *variant* of the Bernoulli shift but does not suffers the state confinement problem: the chaotic behavior is ensured also when considering small. unavoidable implementation errors so this map is suitable for the *practical* implementation. The interval partition on which $M$ is PWAM is $\left\{\left[-1,-\frac{1}{2}\right),\left[-\frac{1}{2},0\right),\left[0,\frac{1}{2}\right),\left[\frac{1}{2},1\right]\right\}$. Because of the noise sets the map initial condition, it's virtually possible to extract all the entropy available at the source. The system exhibits *mixing* properties: independently on initial condition, the probability distribution of state $v_j^{(i)}$, at time step $j$ becomes uniformly distributed in $X$ as $j$ became large. Assuming a partition $\chi$ of $X$ equal to

$$\chi = \{X_0, X_1, X_2, X_3\} = \left\{\left[-1,-\frac{1}{2}\right),\left[-\frac{1}{2},0\right),\left[0,\frac{1}{2}\right),\left[\frac{1}{2},1\right]\right\}$$

and referring to state $\mathsf{x_0}$: if $v^{(i)} \in X_0$, $\mathsf{x_1}$ if $v^{(i)} \in X_1$ and so on and the evolution of the system is described by the four-state Markov chain in Eq. (Fig. 4.7)

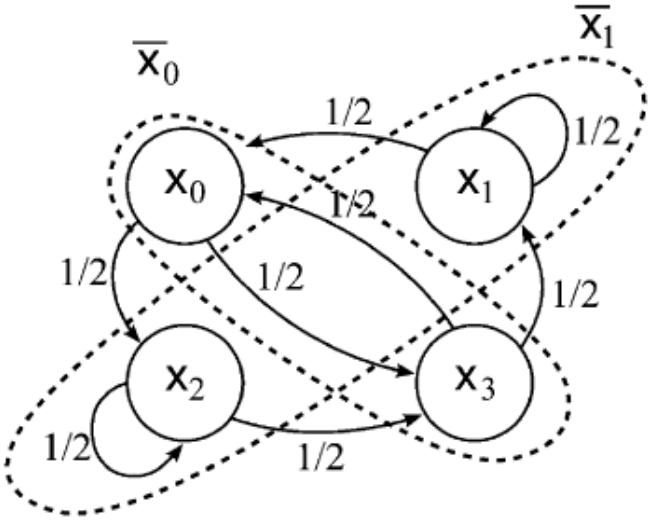



**Figure 4.7:** State chain of a chaotic ADC cell.

This chain is not suitable for the direct generation of random symbols. However, its particularly regular structure allows to *aggregate* the states of the graph into the two macro states $\widehat{x_0} = \{x_0, x_3\}$ and $\widehat{x_1} = \{x_1, x_2\}$, notice that to obtain this result at circuital level it is sufficient to XOR the comparator outputs Eq. 3.2 obtaining a single random bit each clock cycle from two possible configurations namely state $\widehat{x_0}$ and $\widehat{x_1}$. The resulting new state diagram is exactly the Markov chain depicted in Eq. (Fig. 4.4). This prove that the system in Eq. (sec. 3.1) is a TRNG that generates *one random bit* each time step.

The only assumption required to prove this equivalence is that the initial condition of the system (that is $v^{(i)}$ at circuit start-up) is randomly drawn according to a *continuous* probability function. This is verified assuming that the initial condition is set by noise and only few time steps are required for the system to generate digital samples distributed according to Bernoulli process. In the specific, with a Signal-to-Noise Ratio (SNR) as high as $70dB$, the results is obtained after only 25 clock cycles.

## 4.2 Digital Metastable Power-Up SRAM

The second analysis refers to the fully digital TRNG Architecture proposed in sec. 3.2 from [13]. It is based on Fingerprint extraction and Random Number generation is SRAM (FERNS) method.

### 4.2.1 Guessing Probability and Min-Entropy

The system is fully digital and create random numbers using metastable cross-coupled CMOS devices. It exploits the FERNS method: Fingerprint Extraction and Random Numbers in SRAM . After the definition of the basic SRAM cell in Fig. 3.5 and of the *skew* of a cell, in the specific 0-skewed,1-skewed and neutral-skewed cell, that are convenient and illustrative simplification, now it's possible to understand how this RNG works. As explained in sec. 3.2, the neutral-skewed cells in the SRAM can power-up to either state in presence of noise, so they work as tiny, imprecise, six-transistor circuits scattered across the SRAM array, generating and storing random bits at each power-up. This causes latent fingerprints to be randomized.

To extract randomness from a 512-bytes latent fingerprints $X = \{B_1, B_2, ..., B_{512}\}$, *privacy amplification* is employed: the extracted secret is the random number, the body of the information is the latent SRAM fingerprint, generated at power-up, and the (partial) knowledge of the adversary is the tendency of each SRAM cell. Using the metric of *guessing probability* $\gamma(X)$ and *minimum entropy* (min-entropy Eq. (sec. 2.7)) $H_\infty(X)$ is possible to specify bounds on the information that an enemy can posses about an unobserved latent fingerprint.

The *upper bound* is the probability that the SRAM generate a particular latent fingerprint, named the guessing probability of the system. It represents the outcome of the most likely power-up state, that is the best possible guess of an adversary. The guessing probability estimation of a latent fingerprint is based on the assumption that all the 512 bytes are *independent*, so it can be estimated for each byte by observing the most likely outcome across many trials. So the guessing probability of the N-th bit over 100 trials is:

$$\gamma(B_N) = \max\left\{P\left[B_N = b\right] : b \in \{0,1\}^8\right\}. \tag{4.5}$$

The guessing probability of the latent fingerprint, under the assumption of bits independence, is the product of each $\gamma(B_N)$:

$$\gamma(X) = \prod_{n=1}^{512} \gamma(B_N). \tag{4.6}$$

The *lower* bound on the amount of randomness contained in the power-up state of SRAM in the measure of the entropy contained in the most probables power-up state, defined as min-entropy:

$$H_\infty(X) = \log_2\left(\frac{1}{\gamma(X)}\right). \tag{4.7}$$

It is found that min-entropy of SRAM power-up state varies with temperature. Experiments comes from a population of 512-Kbyte SRAM chips, powered and read out using Altera's DE2 development board. They show that 512 bytes of latent fingerprint can be used to create a 128-bit true random number: below we can see results at three different temperatures Tab. 4.1:

**Table 4.1:** Observed min-entropy and associated guessing probability examples.

| **Temp**$(K)$ | **Min-Entropy**(bits) | **Guessing Probability** |
|---|---|---|
| 273 | 189.2 | $2^{-189.2}$ |
| 293 | 202.3 | $2^{-202.3}$ |
| 323 | 218.7 | $2^{-218.7}$ |

### 4.2.2 Supply Voltage and Static Noise Margin (SNM)

The viability of the FERNS method depends on how the TRNG is sensitive to the environments where the circuit will be used. As explained before, the ability of a SRAM cell to hold a state depends on noise and process variation but also on the applied *supply voltage.* Low supply voltage leaves a cell susceptible to noise-induced state changes, while higher voltage makes a cell stable and immune to noise. The minimum supply voltage at which a SRAM cell is able to tolerate "reasonable" noise without changing state falls in the range of 100 to 300 mV. During power-up, it's assumed that the supply voltage begins to 0 V, where all cells can be influenced by noise, and increases to a nominal operating voltage well above 300 mV, where all cells are stable in the "0" or "1" state. The randomness in power-up SRAM state is thus determined by cell behaviors at **low** supply voltage.

The noise immunity of a SRAM cell can be analyzed and quantified using *Static Noise Margin (SNM)* metric, that is defined for a cell at a given supply voltage as the maximum noise voltage that can be tolerated before changing state. Graphically SNM can be seen as the shortest side of the largest box that can be placed inside the eye of the *Voltage Transfer Curves (VTCs)* of the cross-coupled inverters that comprise the cell.

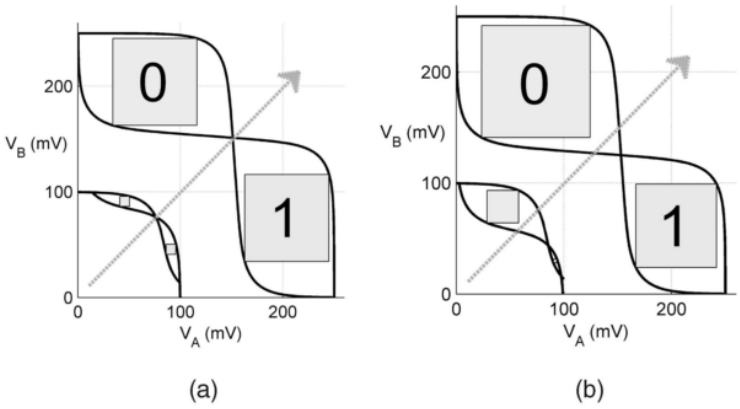



**Figure 4.8:** VTCs of a skewed and neutral SRAM cell at 100 and 250 mV supply voltage. (a) SNMs of unskewed cell. (b) SNMs of 0-skewed cell.

As can be seen in Fig. 4.8, a noise-immune cell has two large eyes between the inverter VTCs. SNM is greatly diminished at low supply voltage as expected. In cell that are not skewed, low supply voltage causes the SNM of each state to be equally small Fig. 4.8(a) and in highly skewed cells, low supply voltage can reduce the SNM of one state to 0 V, indicating the existence of a single noise-immune state.

As showed in Tab. 4.1, also the temperature has an impact on MOSFET devices. An increase in temperature decreases the device threshold voltages (thermal agitation increase):

$$V_{th}(T) = V_{th}(T_0) - \kappa \Delta T. \tag{4.8}$$

while also decrease the electron-hole mobility (number of collisions increase):

$$\mu(T) = \mu_0 \left( \frac{T}{300} \right)^{\alpha}. \tag{4.9}$$

These two trends may counteract each other during power-up, because a lowered threshold voltage will increase sub-threshold current, while reduced mobility will decrease sub threshold-current. Additionally, an increase in temperature increases the magnitude of thermal noise

$$\sigma^2_{NOISE}(T) = \frac{2k_B T}{C}. \tag{4.10}$$

where $k_B$is the Boltzmann constant (equal to 1.3806488x$10^{-23}$ $JK^{-1}$) which could lead to a more random power-up state. The influence of these changing MOSFET characteristic on SRAM power-up state is difficult to model but it can be noticed that this kind of device works on a range of temperature well below the standard high-performance VLSI circuits, that can exceed 400 K, because fingerprint and random numbers are generated at power-up before any self-heating as occurred.

### 4.2.3 Negative Bias Temperature Instability (NBTI)

Storing data in SRAM cells for long periods can cause *burn-in,* allowing the data to be reconstructed long after it was stored. A modern version of burn-in in MOSFET to consider is *Negative Bias Temperature Instability (NBTI).* NBTI is a phenomenon by which deep sub-micron MOSFET threshold voltages *increase* over time due to applied stress conditions of high temperature and vertical electric field caused by the voltage in the MOSFET gate terminal. The effect is dominant in PMOS transistors because they almost always operate with negative gate-to-source voltage but is present also in NMOS transistors. Once the stress is removed, devices begins to recover and in cases where high gate voltage is applied without high temperature, recovery can reach the 100%.

NBTI causes the skew of each SRAM cell to *shift* away from the value previously stored by the cell. As example, consider a 0-skewed SRAM cell that stores a logical 0 (AB=01) as showed in Fig. 4.9.

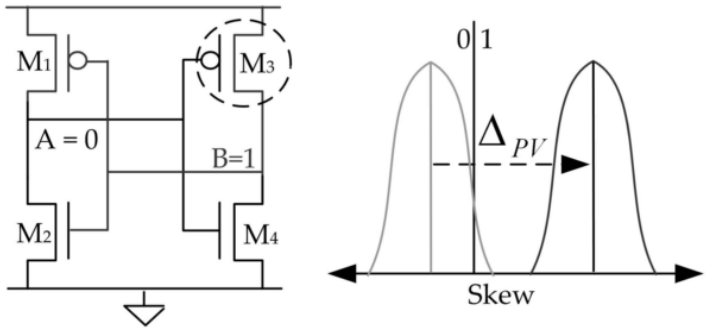



**Figure 4.9:** NBTI raises the threshold of a stressed PMOS device $M_3$ of a 0-skewed SRAM cell: the skew shifts away from the logical 0.

Device $M_3$ (PMOS), experience NBTI stress conditions and so has an increasing threshold voltage afterwards. The next time this cell is powered-up, the higher threshold voltage of $M_3$ causes it to turn on more slowly than normal, making the cell more likely to power-up to the opposing state, logical 1 in this case, as before. Normal usage patterns of intermittently powered devices operating at low temperatures should prevent incidental NBTI from being a significant concern.

## 4.3 3rd Generation Intel® Core™ family processor Ivy Bridge Digital RNG

The analysis refers to TRNG architecture presented in sec. 3.3 that refers to [20]. It is a fully dedicated digital asynchronous pipeline, with a feedback control logic, that exploits metastability in the ES and uses a conditioning circuit to reseed a DRBG to produces random data in outputs. The system is also capable of on-line testing (OHT unit) and BIST upon reset for initialization.

### 4.3.1 Metastable ES

The ES is the most critical component of the system: the state of the system is described as the difference in charge between the two capacitors (A and B caps ) showed in Fig. 3.9, plus the previous output bit. This output bit allows to model some causes of *serial correlation.*

- The distribution and amount of thermal noise affecting the output;
- The amount of charge added(removed) from the capacitors when outputting a logic 0(logic 1), called for convenience right(left) step size;
- The distribution and amount of thermal noise that affect the step size;
- The difference in charge on the capacitors when the system start-up.

The procedure to generate a new output is modeled as follows:

**Algorithm 4.1** `New input routine.`

```
Bias=(Diff_Charge_Caps);
Serial_Adj=if{(Previous_Output_Bit == 1) then 1 else -1} *
Serial_Coefficient;
If(Bias + Serial_Adj + Random_Thermal_Noise > 0) then:
Output=1;
Diff_Charge_Caps=Diff_Charge_Caps-Left_Step_Size + Noise;
Else
Output=0;
Diff_Charge_Caps=Diff_Charge_Caps+Right_Step_Size + Noise;
```

In ideal case (ideal TRNG) the thermal noise model would be:

- *Gaussian SP* with standard deviation $\sigma = 1$;
- Steps in either direction are always 0.1 unit with no noise;
- Serial_Coefficient = 0;
- Starting state with no charge on the capacitors.

However the model presented in [20] is more realistic, thus follows non-ideal conditions :

- Non-Gaussian SP for thermal noise;
- No constant Step sizes in each direction;
- Additive noise also on the step size;
- Positive or Negative Serial_Coefficient;
- Starting state presents some charge on the capacitors.

To perform some statistical analysis on this model, is required to quantize the charge difference and limiting it to a few standard deviation. Doing so, the system can be treated as a Markov Process and from it can be extracted Shannon entropy, min-entropy and also "local" statistics such as auto-correlation and bias.

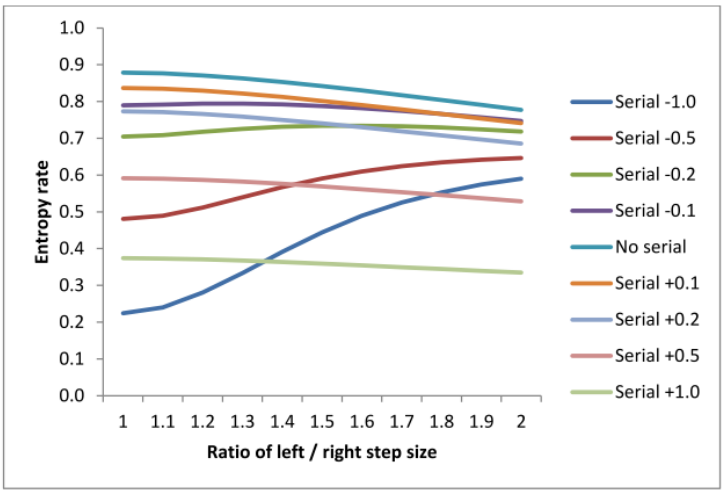



**Figure 4.10:** Effect of bias and serial coefficient on min-entropy with 0.2 mean step size.

In Fig. 4.10 is showed an important result of this kind of modeling: if the serial coefficient is positive or zero, then bias in the step size will decrease min-entropy. However if it is negative then the bias will break the pattern of oscillation, which may increase the entropy.

### 4.3.2 ES Failure Modes

Because of the ES is the most sensitive part in the RNG it is useful to consider its possible *failure modes:*

- ES always shows single-bit bias, serial correlation and other small deviations from perfect randomness. If these biases are severe, they may reduce entropy rate of the ES below acceptable levels;

- ES might take a long time to *warm up* (reach metastable state), and during this time could output mostly 0s or mostly 1s until it settles on the metastable region;
- ES might became *stuck*, always outputting 0 or always outputting 1;
- ES might oscillate between 0 and 1, or in some other short pattern;
- ES might be mostly stuck in one of the preceding patterns, but occasionally deviate from it;
- ES might be influenced by an external circuit, such as chip's power supply, in a way that is predictable or exploitable by an attacker.

Of the possible failure above, most should be detected *reliably* by the Health & Swellness Tests, showed in Tab. 3.1. Furthermore, the presence of a sophisticated DDP stage should mitigate any such unexpected issues. The goal of the OHT unit is to reject ES output with a little actual entropy. Since on *production part* a direct measure of ES's entropy is impossible, the design is intended to catch single points of failure in the ES, such as the failure mode listed above. The question is that the health checks are performed **after** the optional XOR filtering and synchronization logic. The *ratio of the frequencies* ($f_{asynch}/f_{synch}$) between the self-clocking ES and the synchronous region is *not* exact integer, and will drift over time. Hence the number of bits that are included in each sample crossing the *clock boundary* will vary.

As an example, consider an ES that is **"stuck at 1"**, then the output of the XOR filter will toggle between logic 0 and logic 1. The synch region under-sample the XOR output, producing more complicated patterns, depending on how many ES outputs were accumulated in each sample. If the frequency ratio is constant and near 3.3, then the sampled output fails health checks by a margin of 5 samples. If the frequency ratio varies slightly, or the ES is only mostly stuck at 1, then the part may pass the OHT unit tests despite having little entropy. Such a failure will go undetected since on production part it is impossible to examine the ES's raw output, that is fundamental as showed in [18, 28]. In effect, if the health tests were performed on direct ES outputs, no repeating pattern with a period shorter than 12 bits can pass them. It is estimated that, under this failure mode still after the XOR filter and clock domain crossing, the samples have a Shannon entropy rate of nearly 0.4, with a min-entropy slightly lower. Even if this is *less* than the Shannon entropy's design margin of **0.5**, the system's conservative initialization allows it to come up securely with a min-entropy rate of 0.004 that is two order of magnitude less than this failure allows.

As seen in sec. 3.3, the first generation of Intel RNG does not implement the XOR filter. However the ES's outputs are still under-sampled by the synchronous logic but the go directly to the OHT unit, so the health checks are more effective without

the XOR filter e.g., a failure such as "mostly stuck at 1" will certainly be caught.[1] But, without the XOR filter, the OHT unit will not be forgiving of bias in the entropy source. Any part which is biased by more than 57% ones to 43% zeros (or viceversa) is likely to fail BIST.

These concerns can be resolved by having the OHT unit that operates on **all** ES outputs bits directly. In future versions of the RNG, the ES output will be deserialized and then sample *in parallel* into the synchronous region, providing most or all the raw ES output to the health checks.

The swellness check serves for three main purposes:

1. It causes the first 129 healthy (256-bit) samples from the ES to be conditioned into the DRBG's key during BIST. Thus, it saturates the required 128-bit entropy pool even if those samples have min-entropy rate as low as 0.004 design margin;

2. It prevents the RNG from passing BIST unless at least 129 of the first 256 samples are healthy;

3. It prevents the system from remaining mostly unhealthy fro too long.

Furthermore swellness protects *reseed logic* in long term application. As explained in sec. 3.3, reseed happen every few blocks, but if user are not consuming much entropy, then the time between reseeds may be long. During this time, the ES's capacitors might *discharge*, and when the ES is turned back on, it might generate poor data. If most of this data fails the health checks, then the swellness check will eventually fail, preventing the RNG to output weak random bits.

### 4.3.3 DPP and Clock Gating

Entropy conditioning is done via two independent AES-CBC-MAC chains Fig. 3.10, one for the generator's key (K) and one for the counter (C) that works also as entropy extractors. During BIST the conditioner accumulates at least 129 healthy samples for the DRBG's key, so after a restart even if the entropy rate is low, the generator will be in a secure state before it returns any data.

The RNG supports *clock gating* to reduce power consumption. If no application request entropy for a short time (no RDRAND call), the RNG will freeze its clock and stop the ES. This approach may affect the quality of the entropy produced because, the charge on the capacitors may dissipate when the ES is not operating but Intel suggest that is not an issue, as the ES resumes normal operation "quickly", worst case simulation by Intel shows that only the first 256b bit could be affected by a "warm-up effect". In addition, there should be sufficient entropy in the DRBG from the initial seeding during BIST.

[1] Notice that, with sec. 3.3 design is possible to skip the XORing phase and overwrite the buffer before crossing the clock boundary

# 5 Testing of a RNG

## 5.1 Historical Fails

It is better to start remembering some recent historical fails.

### 5.1.1 Cracking Data Encryption Standard (DES)

http://lasec.epfl.ch/memo/memo_des.shtmlThe Data Encryption Standard (DES) is a published federal encryption standard created to protect unclassified computer data and communications. DES has been incorporated into numerous industry and international standards since the Secretary of Commerce first approved DES as a Federal Information Processing Standard during the height of the Cold War in the late 1970s. The encryption algorithm specified by DES is a symmetric, secret-key algorithm. Thus it uses one key to encrypt and decrypt messages, on which both the sending and receiving parties must agree before communicating. It uses a 56-bit key, which means that a user must correctly employ 56 binary numbers, or bits, to produce the key to decode information encrypted with DES.

Although the initial selection of the algorithm was controversial since the US National Security Agency (NSA) was involved in its design, DES had gained wide acceptance and had been the basis for several industry standards, mainly because it was a public standard and can be freely evaluated and implemented. DES technology is readily available worldwide, and several international standards have adopted the algorithm. The process by which DES was developed and evaluated also stimulated private sector interest in cryptographic research, ultimately increasing the variety of commercial security technologies. By 1993, 40 manufacturers were producing about 50 implementations of DES in hardware and firmware that the National Institute for Standards (NIST) had validated for federal use.

Stand to the declarations of the Electronic Frontier Foundation (EFF) at that time, the U.S. government had increasingly exaggerated both the strength of DES and the time and cost it would take to crack a single DES-encrypted message. In fact EFF had proved that a *brute force* attack is possible using a quite simple dedicated machine, defined as DES Cracker.

A 'DES Cracker' is a machine that can read information encrypted with DES by finding the key that was used to encrypt that data. The easiest known way to build a practical DES Cracker is to have it try every key until it finds the right one (brute force). The design of the EFF DES Cracker is simple in concept. It consists of an ordinary personal computer with a large array of custom "Deep-

Crack" chips. Software in the personal computer instructs the custom chips to begin searching for the key, and also functions to interface with the user. The software periodically polls the chips to find any potentially interesting keys that they have located. The hardware's job is not to find the answer, but rather to eliminate most incorrect answers. The software can then quickly search the remaining potentially correct keys, winnowing the "false positives" from the real answer. The strength of the machine is that it repeats a search circuit thousands of times, allowing the software to find the answer by searching only a tiny fraction of the key space. With software to coordinate the effort, the problem of searching for a DES key is "highly parallelizable." A single DES-Cracker chip could find a key by searching for many years. A thousand DES-Cracker chips can solve the same problem in one thousandth of the time. A million DES-Cracker chips could theoretically solve the same problem in about a millionth of the time.

In 1998 DES Cracker won the speed competition posed by RSA Laboratories. Starting at 9:00 AM PST, Monday, July 13, 1998, the EFF DES Cracker began searching for the right key. The machine found the answer at 5:03 PM Pacific PST, Wednesday, July 15. Coincidentally, it took the EFF DES Cracker 56 hours to find a 56-bit key. When the EFF team started the search on Monday morning, they had 35868 search units running on 26 boards (each search unit examines 2.5 million keys per second). The team stopped the search for a few minutes on Tuesday night to improve the software and then again for a few minutes on Wednesday to add a 27th board, which sped up the machine slightly (to 37050 search units). The EFF DES Cracker searched 17,902,806,669,197,312 keys to find the correct answer, which averages out to a rate of 88,803,604,509 keys tested per second (88 billion). The machine was examining 92,625,000,000 keys per second when it found the answer. The key was found after searching almost exactly a quarter of the key space (24.8%).

After DES cracking NIST had started the process of developing an "Advanced Encryption Standard" or AES, which was designed to last for a decade or more after its adoption. Now AES is still evolving and it is the public standard adopted worldwide.

### 5.1.2 OpenSSL DSA and ECDSA

Luciano Bello discovered a serious flaw in the DRBG that shipped with the OpenSSL cryptography library on Debian and Ubuntu Linux systems from September 2006 to May 2008 [1]. All OpenSSL keys generated by the affected systems were compromised, including server certificates, SSH login keys and email signing/encryption keys. More recently, in 2012 a study showed that an unexpectedly large number of Rivest Shamir Adleman (RSA) moduli share common prime factors, which can easily be computed using the Greatest Common Divisor (GCD) algorithm.

One of the most likely causes is poor random number generation processes . The need for strong randomness is not limited to key generation. For example, the popular Digital Signature Algorithm (DSA) and Elliptic Curve DSA (ECDSA) digital signature standards require a random value when each signature is produced. Even

very slight biases in the RNG used to produce this value can lead to exploitable cryptographic weaknesses. Bleichenbacher discovered that the nonce generation method defined in FIPS 186 was slightly biased, and this bias could be used to mount a cryptanalytic attack against DSA and ECDSA.

## 5.2 TRNG Statistical Hypothesis Testing (SHT)

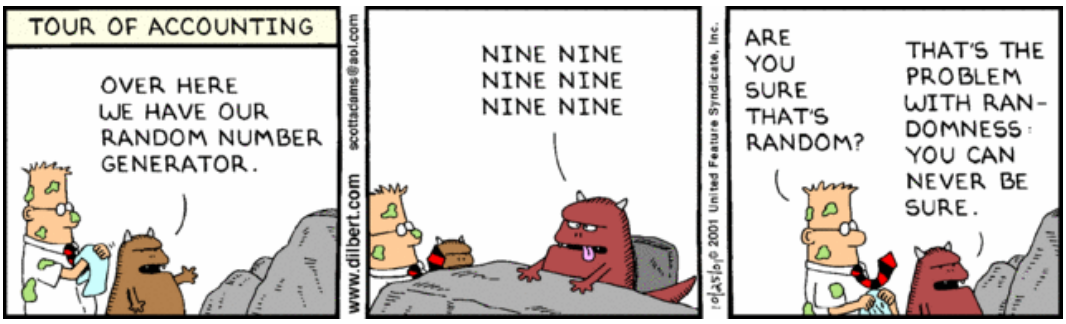



**Figure 5.1:** Testing Randomness.

Fig. 5.1 is a good introduction for the randomness testing field, its statement is clear: randomness is a *relative* subject. [1] Using Probability theory tools it is possible to face this challenge and test a TRNG module. The key to solve this kind of problem is to understand the approach and find the right focus in each situation. The methodology used is under the name of Statistical Hypothesis Testing (SHT) or simply Statistical Test (ST). Recalling the concept of null and alternate hypothesis from sec. 2.10, in this case $H_0$ tests if the occurrence of logical 0 and logical 1 is independent and identically distributed over the sampled sequence. Formally:

**Definition 5.1. Statistical Hypothesis Test (SHT)**. Given a bit sequence $a$ of length $n$ it is possible to test some $H_0$ with some DSP block. Given a fixed threshold value $\alpha$, defined as **significance level** it is possible to extract from the test an output called **P-value** or **P-val** and compare it with the significance level.

Fig. 5.2 shows the block scheme, the flowcharts in this chapter are color coded, the orange part represents the processing phase, the signals are plotted in blue, the significance level is plotted in purple and the acceptance/rejection is plotted in green/red. The scheme of a generic ST is very simple: given a null hypothesis $H_0$, a random bit sequence of length $n$ and a fixed significance level $\alpha$, the system is able to provide an P-value in output.

[1] http://dilbert.com/strips/comic/2001-10-25/

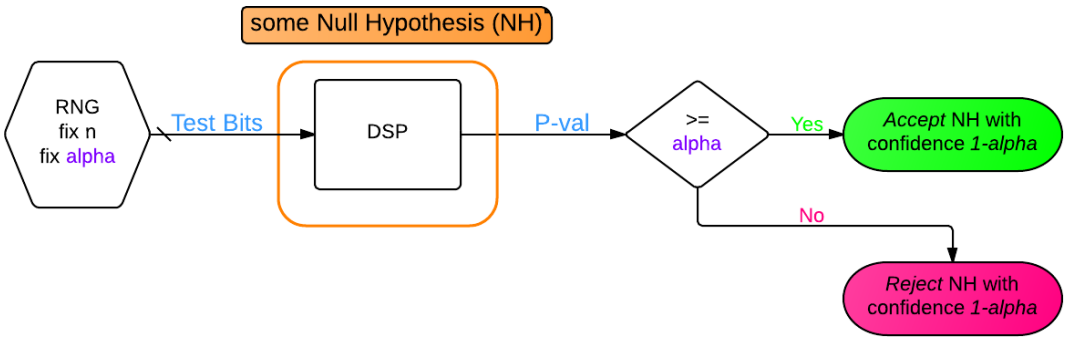



**Figure 5.2:** TRNG Statistical Test .

The meaning of the P-value is *dictated* from the type of test performed and from the alpha's value.[2] Following the usual criterion of an *hypothesis test* is it possible to determine if the null hypothesis is rejected or not with a certain confidence, namely:

$$\text{if } \begin{cases} \text{P} - \text{value} \leq \alpha & \text{then } \text{H}_0 \text{ is rejected with a confidence of } 1-\alpha; \\ \text{P} - \text{value} > \alpha & \text{then } \text{H}_0 \text{ is accepted with a confidence of } 1-\alpha. \end{cases} \tag{5.1}$$

It is important to stress out that, as can be seen from the second statement in Eq. 5.1, that *every* statistical test, to make sense, start from some hypothesis that introduce an error, and ends with an *interpretation* of the results that depends strictly on the initial assumption. E.g., suppose that $\alpha = .001$, and $\text{P} - \text{value} \geq \alpha$ so $\text{H}_0$ is not rejected, or in other terms, is accepted with a confidence of 99.9%. This means that the sequence pass the test with the 0.01% of being completely non-random. For this reason it is possible to define two *correlated* type of errors:

**Definition 5.2. False positive error.** Incorrect rejection of $H_0$. In each test there is always a *non-zero* probability of this type of event:

$$\Pr(p \leq \alpha) = \text{CDF}_{\text{iid}}(\alpha) = \alpha.$$

It is also called *type I* error, and in TRNG testing occurs when you discard a good (presumably IID) sequence of random bits. $\alpha$ it is used as a measure of **significance** and it is a fixed value.

**Definition 5.3. False negative error.** Incorrect acceptance of $\text{H}_0$. In each test there is always a *non-zero* probability of this type of event, denoted with $\beta$. It is also called *type II error,* and in TRNG testing occurs when you accept a bad (presumably non IID) sequence of random bits. $\beta$ is *not* a fixed value and it is more difficult to compute. but it can be estimated from the values of $\alpha$ and the size $n$ of the tested sequence.

$$\min(\beta) = \min(\alpha, n).$$

[2] $\alpha$ usually spans from .05 to .001

Usually the aim is to *minimize* $\beta$ with respect to $\alpha$, because we can tolerate that occasionally a random sequence does not pass a test but it is more dangerous if occasionally a non-random sequence pass test. So to produce the smallest $\beta$ test setting $\alpha$ and $n$ must be set well.

It is important to stress out that randomness is a *relative* subject, so the key of statistical testing is the **interpretation** of results (P-values) with respect to the null hypothesis. To give a perspective remember that an ideal RNG can generate, during normal operating conditions, 100 consecutive logical zeros but under some null hypothesis is considered, rightly, as non random.

There are three test suites of interest, they are general purpose in the sense that they can be applied for TRNG and PRNG:

1. **DIEHARD** [8]: provided by the mathematician G. Marsaglia from Florida State University. It is a battery of twelve tests implemented in many programming language (initially FORTRAN and C). Since 1990 the suite is distributed for free at the beginning as a CD-ROM and now on the internet (link in the reference). It expects input files of the order of 10-11 MB (at least 80Mbits).

2. **dieharder [5]:** The dieharder suite is more than just the DIEHARD tests cleaned up and given a pretty GPL'd source face in native C. Tests from the Statistical Test Suite (STS) developed by the National Institute for Standards and Technology (NIST) are being incorporated. It expects GB file order.

3. **NIST SP 800-22** [27]: provided by the American National Institute of Standards and Technology. It is a fifteen tests battery and represent the state of the art for RNG testing. These tests were used by the *Federal Information Processing Standard (FIPS)* publicly to select, from all the candidates, the actual *Advanced Encryption Standard (AES)* cipher, developed in 2001 by two Belgian cryptographers: J. Daemen and V. Rijmen. The test suit initially consisted of sixteen statistical test, but the Discrete Fourier Transform (Spectral) test and the Lempel-Zip Compression test were disregarded due to problems identified by NIST and other researchers [19].

In this work will be presented the latter

## 5.3 NIST test suite: SP800-22

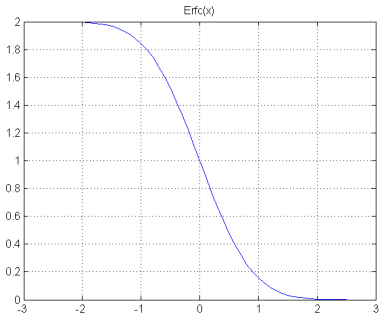
NIST SP800-22 [27] is a fifteen tests battery and represent the state of the art for RNG testing (both Pseudo and True). Notice that this tests suite was used by the *Federal Information Processing Standard (FIPS)* publicly to select, from all the candidates, the actual *Advanced Encryption Standard (AES)* cipher, developed in 2001 by two Belgian cryptographers: J. Daemen and V. Rijmen. The test suit initially consisted of sixteen statistical test, but the Discrete Fourier Transform (Spectral) test and the Lempel-Zip Compression test were disregarded due to problems identified by NIST and other researchers [19].

Each of these tests hypothetically quantifies a certain aspect of randomness i.e. testing a different null hypothesis. Given a bit sequence $a$ of length $n$, the significance level $\alpha$ is fixed *apriori* in the interval $[.001, .01]$. The bit sequence coming from a RNG is analyzed using a specific data processing technique for each tests. The aim of NIST's tests is to minimize $\beta$, namely the probability of accept a non-random bit sequence. All of the subsequent tests assume implicitly that all the discussions and computation are *under the randomness hypothesis* (the null hypothesis).

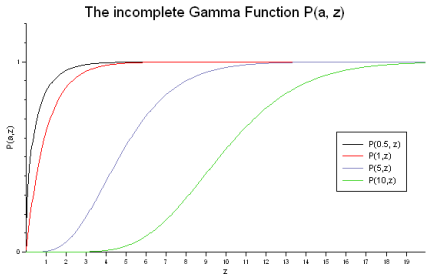
There are two classes of tests:

1. **Erfc**. This class of tests, indicated with the letter **E** in Tab. 5.1, are based on Thm. 2.15 (de Moivre-Laplace) where a Bernoulli SP can be approximated with a Gaussian RV, so the P-val can be computed using the *Complementary Error function* $erfc(x)$, plotted in 5.3a.

2. **Chi-square**. This class of tests, indicated with the letter **C** in Tab. 5.1, are based on Chi-Square $\chi^2$ test Thm 2.16, where the resultant statistics should fit with a theoretical Chi-square distribution, so the P-val can be computed using the *Upper Incomplete Gamma function* $gammainc(a, b,' upper')$, plotted in 5.3b.




**(a)** $erfc(x)$.




**(b)** $gammainc(a, b,' upper')$.

**Figure 5.3:** NIST SP800-22 test's class.

It is useful to show a synthetic review of all the 15 tests in Tab. 5.1:

**Table 5.1:** NIST SP800-22 summary.

| **Test Name** | **Null Hypothesis** | **Code** |
|---|---|---|
| 1 - Frequency Mono-bit Test | Good proportion of 1s and 0s. | E |
| 2 - Frequency Test within a Block. | Good proportion of 1s and 0s within N blocks of M-bit. | G |
| 3 - Run Test (FMt pre-requisite). | No clusters of 1s and/or 0s in a sequence. A run is series of consecutive 1s or 0s. | E |
| 4 - Test for the Longest Run of 1s in a Block (implicitly test also the LR of 0s) | No clusters of 1s and/or 0s in N blocks of M-bit. | G |
| 5 - Binary Matrix Rank Test. | Low linear dependence between sub-strings, that means sub-matrices with high rank (full rank=M). | G |
| 6 - Discrete Fourier Transform (Spectral) Test. | Balanced amount of peaks in the frequency domain, (not a lot of periodic patterns in time domain). | E |
| 7 - Non-Overlapping Template Matching Test. | Not too many non-overlapping equal sequences. | G |
| 8 - Overlapping Template Matching Test. | Not too many overlapping equal sequences. | G |
| 9 – Maurer's Universal Statistics Test. | Sequence cannot be significantly compressed. | E |
| 10 – Linear Complexity Test. | Sequence with long Linear Feedback Shift Registers (LFSRs). | G |
| 11 – Serial Test. | Every m-bit template has equal probability to arise. | G |
| 12 – Approximate Entropy Test. | Not regular occurrence of the same overlapping template. | G |
| 13 – Cumulative Sums (Cusums) Test. | Cumulative sum (considered as a random walk) excursion near zero. | E |
| 14 – Random Excursion Test. | Random Distribution of visits among cycles to eight states (eight P-values provided). | G |
| 15 – Random Excursion Variant Test. | Random Distribution of visits among cycles to eighteen states (eighteen P-values provided). | E |

As you can see the suite contains fifteen tests. Each test in the NIST suite requires a pre-defined minimum sample set of size ranging from 100 bit to 100 Kbits.The majority of them:

- either examine the distribution of zeroes and ones in some fashion:
- or study the *harmonics* of the bit stream utilizing spectral methods:
- or attempt to detect *patterns* via some generalized pattern matching technique.

The *complexity* of the test increase with the number. Generally a good RNG should be able to pass *all* fifteen test. Notice that there are some tests able to provide *multiple* P-values e.g. Cusums. All the material is distributed for free using a framework (not very user friendly) written ANSI C programming language. [3]

In the next subsections I will present some of the most important NIST SP800-22 ST, the explanation will be aided by some flowcharts that I personally create to simplify the treatment.

### 5.3.1 Frequency Mono-bit (FMT)

The first test is called Frequency Mono-bit Test (FMT), it is a sort of *preliminary filter* for the other fourteen, because it is very difficult that a sequence that fails FMT pass the others tests.

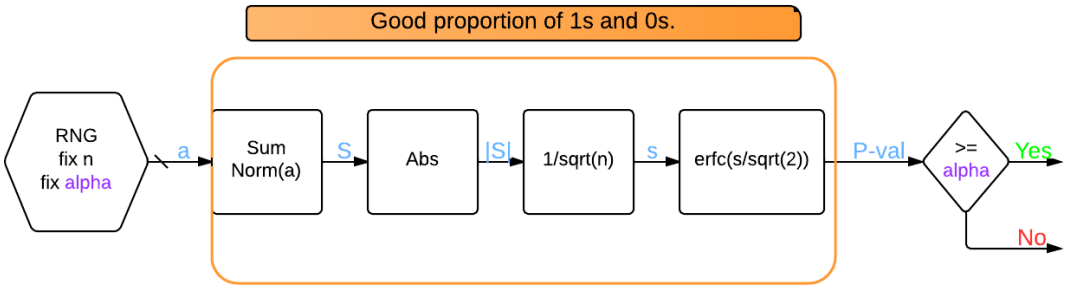



**Figure 5.4:** FMT flowchart.

As you can see from the Fig. 5.4, the aim of this test is to verify a good proportion of logical 1s and logical 0s in the bit sequence. In the specific a bit sequence $a$ is processed by the first block that computes the normalized sum of the input such that:

$$a_n(i) = \begin{cases} 1 & \text{if } a(i) = 1 \\ -1 & \text{if } a(i) = 0 \end{cases} \tag{5.2}$$

$$S = \sum_i a_n(i). \tag{5.3}$$

Then, the test take the absolute value of $S$ and divide it by the square root of $n$, obtaining a constant $s$. This constant is divided by the square root of two and used as argument for the complementary error function. A P-val is computed and compared with the significance level $\alpha$, then a decision is taken.

[3] http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

### 5.3.2 Frequency Test within a Block (FTwB)

The second test is the Frequency Test within a Block (FTwB). This test is based on the same null hypothesis of the FMT, but the bit sequence is splitted into N block of M-bit discarding the remainings, and each block is separately processed.

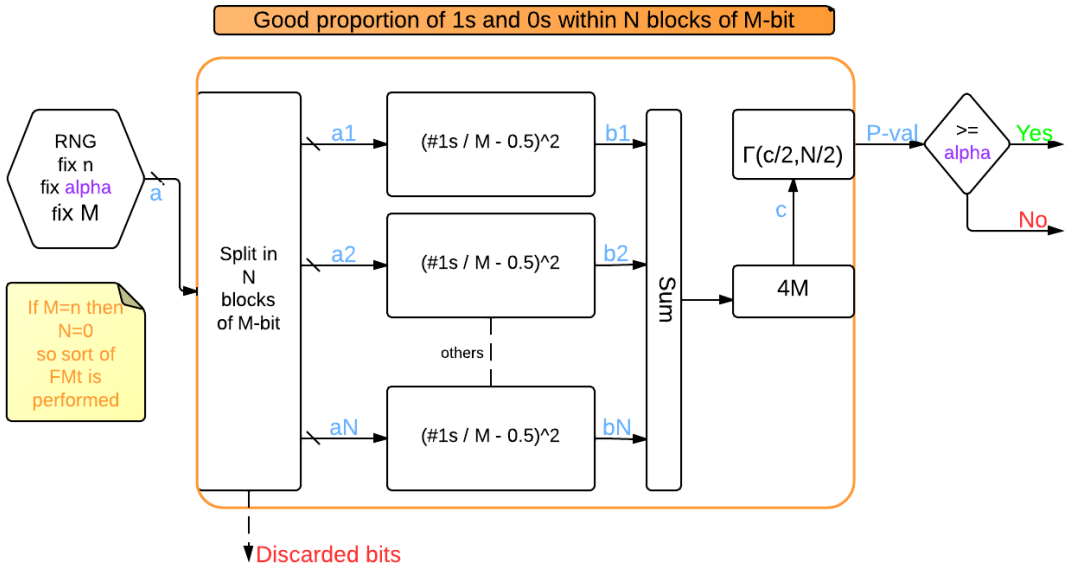



**Figure 5.5:** FtwB flowchart.

As you can see from in Fig. 5.5, for each block is computed the power of the distance between the proportion of ones and the theoretical ideal distance .5; then the results are accumulated in a vector, multiplied by a constant and given as input argument to the gammainc function with N\2 degrees of freedom. A P-val is computed and compared with the significance level $\alpha$, then a decision is taken. Notice that, if M $= n$, this test degenerates in a modified version of Fmt.

### 5.3.3 Runs Test (RT)

The third test presented is the Runs Test (RT). This test has the FMT as a prerequisite. A *run* is defined as a series of consecutive logical 0s or logical 1s and this test study if there are clusters of consecutive 1s and/or 0s in a random bit sequence.

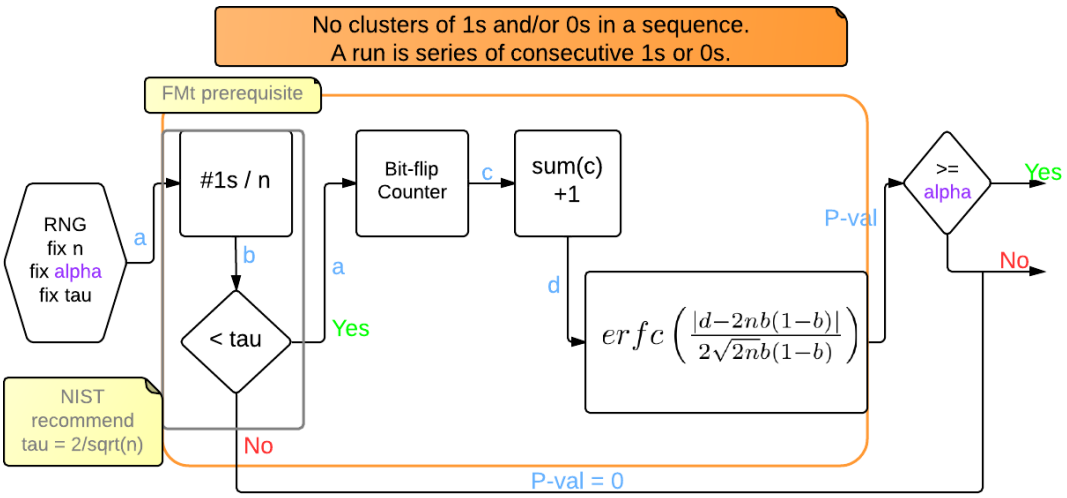



**Figure 5.6:** RT flowchart.

As you can see from Fig. 5.6, if the proportion of 1s in the sequence is *less* then a certain threshold, defined by NIST as $\tau = \frac{2}{\sqrt{n}}$, the RT is not performed and returns a 0 P-val. If the tau condition is met, the bit sequence is scanned such a way that the vector $c$ contains 1s each time the bit sequence **flips** and 0s otherwise. Then $c$ is accumulated, increased by one and used as an argument of the erfc function. A P-val is computed and compared with the significance level $\alpha$, then a decision is taken.

### 5.3.4 Longest Run of 1s Test (LR1sT)

The fourth test proposed is the Longest Run of 1s Test (LR1sT). This test study the presence of clusters like the RT, but it is applied on subsets of the sequence, namely N blocks of M-bit.

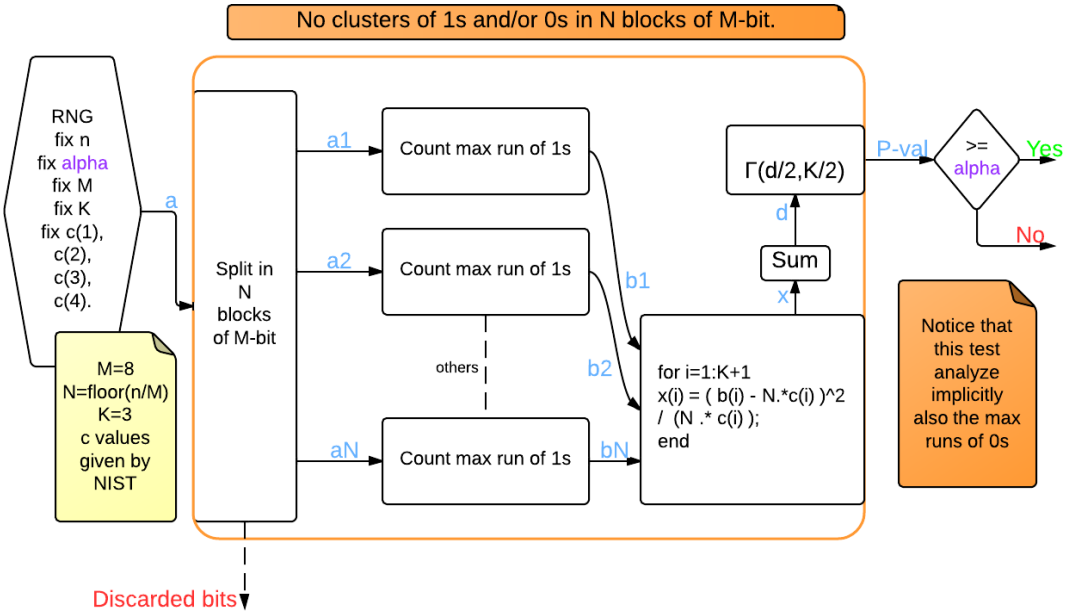



**Figure 5.7:** LR1sT flowchart.

In Fig. 5.7 is presented the configuration for M=8, thus N=floor(n/M). This assumption sets implicitly the degree of freedom of theoretical chi-square distribution K=3. The input random bit sequence is splitted into N blocks of M bit, discarding the remainings. Each block is scanned and return a value equal of its max run of 1s. In this configuration the vector $b$ can count *four* different *runs type* namely:

1. b(0) counts the number of runs with length 0 and 1.
2. b(1) counts the number of runs with length 2.
3. b(2) counts the number of runs with length 3.
4. b(3) counts the number of runs with length 4,5,6,7,8..

Then some computations are performed on $b$ giving out the vector $x$ and then a P-val is computed using gammainc and compared with the significance level $\alpha$, then a decision is taken.

### 5.3.5 Binary Matrix Rank Test (BMRT)

The fifth test proposed is the Binary Matrix Rank Test (BRMT). This test study the linear dependence between sub-strings of random bit using the *rank* of matrix as an index.

**Definition 5.4. Rank** of a nxn square matrix is the number of linear independent columns/rows, the min rank is 0 (all zero entries) and the max rank is n, if a matrix M has max rank, M is defined as a *full rank* matrix.

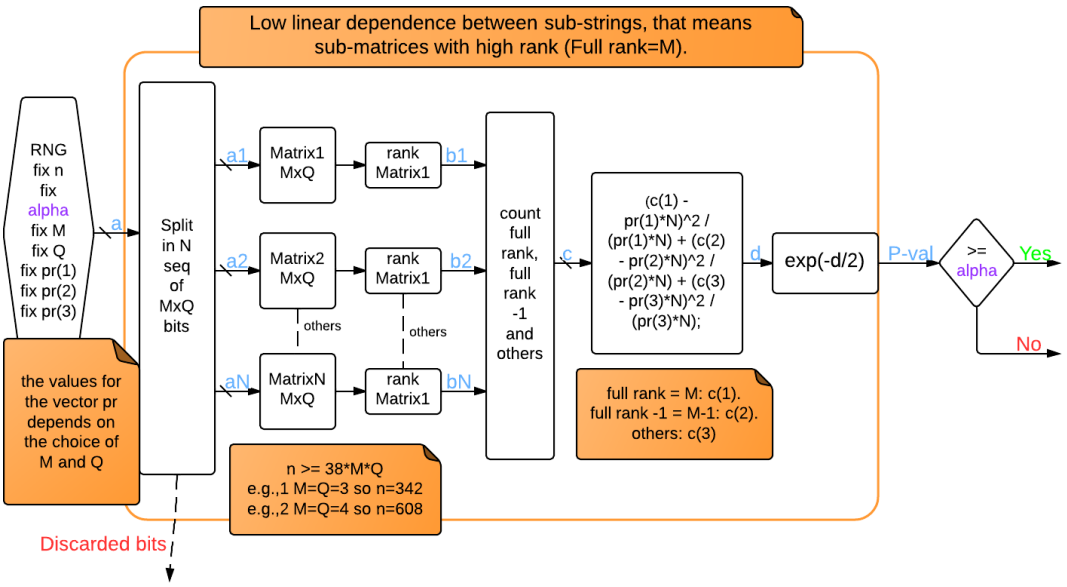



**Figure 5.8:** BMRT flowchart.

As you can notice from Fig. 5.8, the input sequence is splitted into N matrices with dimension MxQ, where M=Q (N square matrices). For each matrix is computed the rank, then a three-dimensional vector $c$ store:

- $c(1)$ number of full rank matrices.
- $c(2)$ number of full rank - 1 matrices.
- $c(3)$ others.

Then some computation are performed using $c$, namely:

$$d = \frac{(c(1) - pr(1)N)^2}{pr(1)N} + \frac{(c(2) - pr(2)N)^2}{pr(2)N} + \frac{(c(3) - pr(3)N)^2}{pr(3E)N}. \tag{5.4}$$

Then the result $d$ is used as argument to compute the P-val. The output is compared with the significance level $\alpha$, then a decision is taken. Notice that this tests appears also in the DIEHARD suite and was invented by a mathematician named Kovalenko [21].

### 5.3.6 Non-overlapping Template Matching Test (NoTMT)

The sixth test presented is called Non-overlapping Template Matching Test (NoTMT). This test study the frequency of pre-defined non-overlapping patterns. Notice each pattern return a P-val.

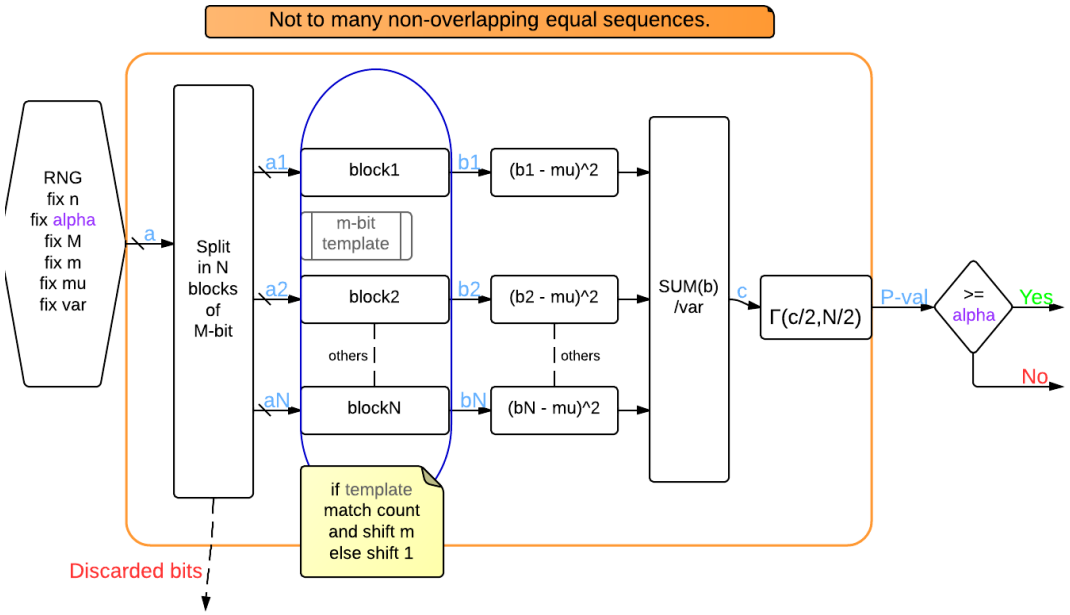



**Figure 5.9:** NoTMT flowchart.

As you can see from Fig. 5.9, the input bit sequence is splitted into N blocks of M-bit discarding the remainder, NIST recommend N = 8. In this case we fix also the input bit length $n = 512$ and the correspondent template length $m = 9$ as recommended by NIST ($m >= 9, 10$ ). Then the remaining constants can be fixed:

$$mu = (M - m + 1)/2^m,$$

$$var = M * (1/2^m - (2 * m - 1)/2^(2 * m)).$$

Given a template, e.g. 101111001, each block is compared with template *bit wise* and has its dedicated counter, if there is a complete matching the count goes up and the template is shifted of m position (non-overlapping). Then each counted value is processed using the constants mu and var obtaining the vector $c$. This vector is used as one argument for the gammainc function. A P-val is computed and compared with the significance level $\alpha$, then a decision is taken. This process is repeated for each template tested.

### 5.3.7 Overlapping Template Matching Test (OTMT)

The seventh test proposed is the Overlapping Template Matching Test (OTMT). This test study the frequency of pre-defined overlapping patterns. Notice each pattern return a P-val.

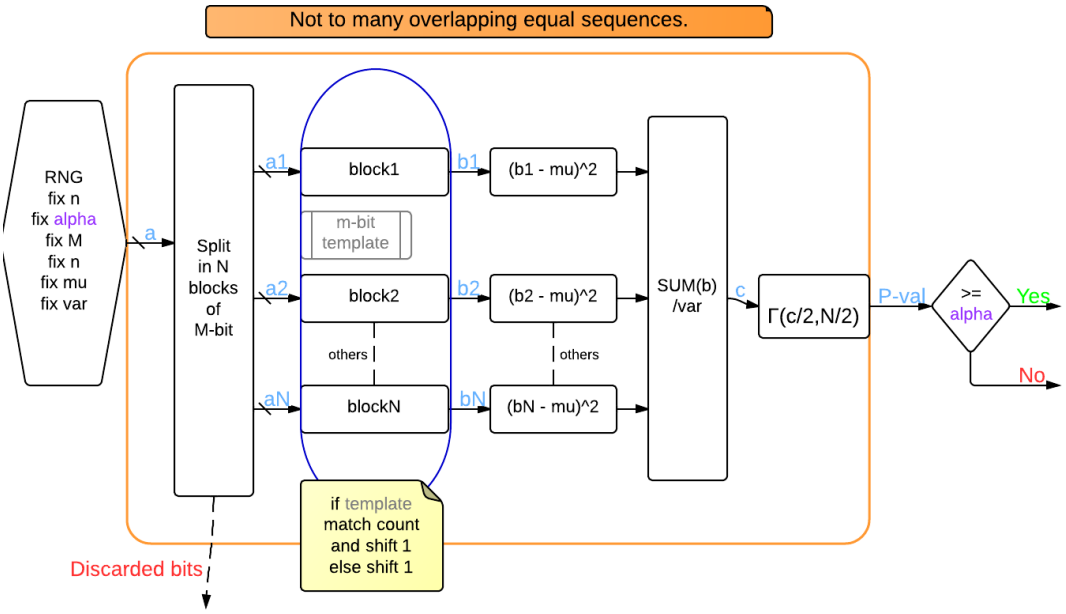



**Figure 5.10:** OTMT flowchart.

As you can see from Fig. 5.10, the procedure is equal to the one described in sec. 5.3.6, with only one difference: it there is a complete matching between the m-bit template and the sub-string analyzed, the patterns is shifted of 1 bit (overlapping) and, as in the non-overlapping case, the counter is incremented by one.

### 5.3.8 Cummulative Sum Test (CST)

The eighth test presented is the Cummulative Sum Test (CST). This test analyze the excursion from 0 of the *random walk*.

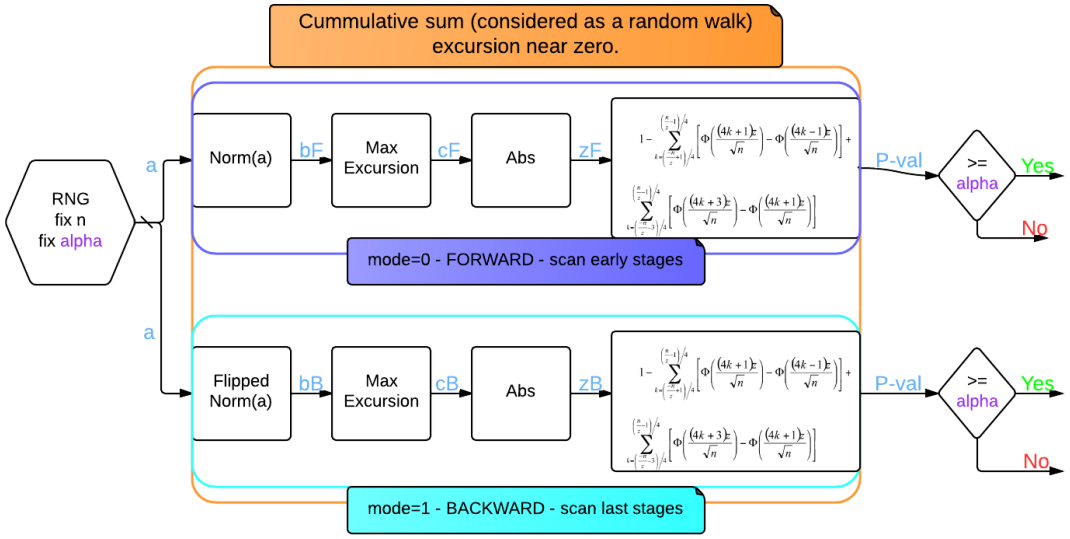



**Figure 5.11:** CST flowchart.

CST performs a double test:

1. **FORWARD** mode. The bit sequence is normalized, see Eq. 5.2. Then it is computed the max excursion of the normalized bit sequence (random walk). Then some processing is performed involving the normal CDF $\Phi$. A forward P-val is computed and compared with the significance level $\alpha$, then a decision is taken. Notice that the forward test focus on the beginning of the input.

2. **BACKWARD** mode. The bit sequence is flipped and then normalized, see Eq. 5.2. Then it is computed the max excursion of the flipped normalized bit sequence (random walk). Then some processing is performed involving the normal CDF $\Phi$. A backward P-val is computed and compared with the significance level $\alpha$, then a decision is taken. Notice that the backward test focus on the last part of the input.

# 6 Hardware Implementation of Reduced NIST SP800-22

This chapter refers to my personal publication, submitted (4 Jan 2013) and accepted (22 Feb 2013) for IEEE *Hardware Oriented Security and Trust (HOST 2013)* Symposium that will take place in Austin TX (June 2-3 2013 ). [1]

The title of the paper is:

> **"On-chip Lightweight Implementation of Reduced NIST Randomness Test Suite [32]"**.
> Keywords: PRNG, TRNG, NIST, Statistical Test.

The work was made between Oct and Dec 2012 at UMass Amherst (MA) in the VLSI/Security Lab collaborating with my advisor: Prof. Riccardo Rovatti and my two co-advisors: Prof. Wayne Burleson and Ing. Vikram Suresh and all the other kind people of the Lab.

## 6.1 Abstract

On-chip Random Number Generators (RNGs), are critical component in lightweight ubiquitous devices like RFIDs and smart cards. The work proposes an *on-chip* implementation of a reduced set of NIST SP 800-22 statistical tests (details in sec. 5.3). The aim is to provide *on-line* RNG testing for low cost security devices along with run-time monitoring of RNG performances. The on-chip NIST module, monitors the effect of dynamic variation of operating condition and time dependent wear-out on RNG circuits. It indicate invasive attacks on RNG and allows the secure system to take protective measure. *Six* NIST test suite are optimized to a hardware design friendly format, but in compliance with the NIST standard.

The *lightweight* implementation reduce complex statistical and arithmetic operations of conventional NIST tests, to a series of bit stream count and compare operations. The need for additional storage is eliminated by the fact that incoming bits from RNG are serially tested cycle-to-cycle. A partial *re-configurable* feature is designed to set the pass/fail threshold for each test, depending on system requirements. The on-chip NIST module, although not exhaustive, is an effective layer of validation and security, for RNG circuits.

[1] http://www.engr.uconn.edu/HOST/

The six 128-bit tests implemented in 45 nm using open-source *North Caroline State University Process Design Kit (NCSU PDK),* have a total synthesized area of $\sim 1926\ \mu m^2$ for an optimized frequency of 2 GHz. The total dynamic power is 3.75 mW and leakage power is 10.5 $\mu$W. At 2 Gbps, the NIST module consumes 1.87 pJ/bit. The lightweight implementation is *scalable* for larger input bit samples.

## 6.2 Motivations

As extensively studied in this work, RNGs are critical blocks in a variety of cryptographic system. As a result, weak RNGs can be a single point of failure of the entire system. Roughly speaking, the weakness of a RNG can be due to weak algorithm (PRNG), or a bad ES and process induced variation in sample and digitize circuit (TRNG). So the reliability of a TRNG circuits depends strictly on physical implementation, therefore the statistics of these circuits have to be validated *post* device fabrication. Unlike conventional VLSI testing methodologies, the statistics of TRNG circuit cannot be tested using fault models, thus large set of data have to be generated and validated using some test suite.

The constant technological scaling provides increasing parameters variation in fabrication process of ICs, thus testing a lager sample of chip does not necessary guarantee a good TRNG in every chip. On the other side, an exhaustive post-fabrication test will increase the test time and hence the *cost per chip.*

We propose a lightweight implementation of six (out of fifteen) statistical tests in the NIST test suite. The proposed implementation is not a replacement to the exhaustive testing provided by the NIST standard, however they encompass the *mandatory* tests listed in *FIPS 140-1* standard [22] and provide an additional layer of security for lightweight TRNG implementation and it can detect abnormal behavior due to variation in fab process, dynamic variation in environmental condition during run-time (e.g. PVT conditions) and direct indirect malicious attack on TRNGs. Further, the NIST module can be turned ON intermittently to reduce power overhead.

Smart cards, RFID tags and sensor nodes are only some example of low cost device that needs TRNG, these devices cannot afford to go through an extensive post-fabrication testing process to validate the TRNG circuits. They are also highly constrained in area, and may passively powered and battery operated. Therefore, a lightweight on-chip NIST module provides efficient statistical validation for the generated random bit stream.

Time-dependent wear-out phenomena like Negative Bias Temperature Instability (NBTI) and Hot Carrier Injection (HCI) can degrade over time the circuit, thus a one-time statistical test performed during the chip testing phase does not provide constant monitoring of circuit behavior. An on-chip test module can be used to continuously monitor the RNG output. The cryptographic system, using the test module can male an *informed* decision about the randomness of inputs in run-time. In [14], a complete NIST test suite implementation has been proposed for real-time

statistical testing. However, such an implementation adds tremendous area overhead to the system, consuming orders of 1000's of flip flops. The authors indicate that only two NIST tests could be ported completely on a Xilinx Virtex II Pro FPGA V2P30.

There are many ways of attacking a RNG. In [17] J. Kelsy, et. al discuss in detail the various cryptanalytic attacks on PRNG. Side channel attack, Electromagnetic wave attacks and eavesdropping attack are common examples been reported in literature. Commercial microprocessor manufacturers use on-chip sensors (temperature, voltage) to detect invasive attacks. An on-chip NIST module can detect variation in randomness, which is the symptom of an attack, thereby protecting the device against all sources of attack. This randomness information can be used by the secure module to take necessary evasive action like a *Denial of service (DoS)* to the attacker. In a *shared-key* protocol for secure communication, the system can test the received key using on-chip NIST module to get a measure of randomness of that key, rejecting or accepting it.

All systems using TRNG employ some form of calibration or DPP to mitigate the effect of physical variation of the ESs, for details see sec. 1.2.3, an on-chip test module allows *multiple* post-processing units to be implemented and then select the appropriate one based on the quality of the TRNG, the other blocks can be powered OFF to reduce the leakage power.

Resuming, on-chip statistical test suite is imperative for secure computation on lightweight devices with scaling technology, cost/chip reduction and increasing invasive attacks. We propose a lightweight implementation of 6 statistical test from NIST test suite. The test are chosen based on the *minimum sample set* recommended by NIST and the complexity of *storage* and *computation* in terms of ultra-low power implementation .

## 6.3 Reduced NIST SP800-22 test suite

Testing the randomness of a bit sequence is as challenging as generating them. Randomness is a *relative* term and there is no definite metric to quantify it. Statistical Hypothesis Testing (SHT) or simply Statistical Test (ST) are the common approach in this field, see sec. 5.2 for details. These tests provide a hypothesis of whether the bit sequence can be considered random or not, with a specific level of *confidence.* Since the focus of this work is to realize a test module for ultra-lightweight applications, we choose six out of fifteen NIST tests, see Tab. 6.1.

**Table 6.1:** NIST SP800-22 candidates.

| **Test Name** | **Null Hypothesis** |
|---|---|
| Frequency Mono-bit Test | Good proportion of 1s and 0s. |
| Frequency Test within a Block. | Good proportion of 1s and 0s within N blocks of M-bit. |
| Run Test (FMt pre-requisite). | No clusters of 1s and/or 0s in a sequence. A run is series of consecutive 1s or 0s. |
| Test for the Longest Run of 1s in a Block (implicitly test also the LR of 0s) | No clusters of 1s and/or 0s in N blocks of M-bit. |
| Binary Matrix Rank Test. | Low linear dependence between sub-strings, that means sub-matrices with high rank (full rank=M). |
| Non-Overlapping Template Matching Test. | Not too many non-overlapping equal sequences. |

Our reduced set includes the four tests mandated by FIPS 140-1 standard [22].

The Complementary Error function and the Upper Incomplete Gamma function are the bottleneck in computing the P-val for each test. To reduce the computation required, we fix the bit-length value $n$ and compute the range of input $x$ to the erfc or gammainc, which lead to a P-val greater than the critical value of $a$. Notice that is possible to use this *back-of-envelope* calculation because the two complex function are strictly monotonic (increasing CDFs), thus the inverse of these function is strictly monotonic (decreasing). We completely eliminate the need of complex computation of error function and gamma function, thereby significantly reducing the complexity of hardware required to implement the tests. In fact, the proposed implementation *shifts* the results dependence from the P-vals to the $x$ values, without losing in accuracy because the two are one-to-one mapped. The partial reconfigurable feature allows the user to set the critical value $\alpha$ depending on the need of the application to make the test more stringent. Modifying the critical value only varies the bounds for the input to the complex function. The bounds can be set one-time by blowing fuses or can be configured using registers to store the values. Furthermore, all the computations are performed serially on the incoming bits from the RNG, this eliminates the need for additional storage devices, except for byte-wide shift registers.

## 6.4 Lightweight Implementation of reduced NIST Test Suite

In this section a detailed description of the lightweight implementation for the Frequency Test within a Block (FTwB) is presented, see sec. 5.3.2. The implementation of the other five tests is also based on similar techniques. For a detailed statistical analysis of each test, interested readers may refer to [27].

As showed in Fig. 5.5, the FTwB checks for a good proportion of 1s and 0s in block of random bits. The M parameter is settable and implicitly sets N, the other bits are discarded. For each block is computed the power of the distance between the proportion of ones and the theoretical ideal distance .5; then the results are accumulated in a vector, multiplied by a constant and given as input argument to the gammainc function with N\2 degrees of freedom. A P-val is computed and compared with the significance level $\alpha$, then a decision is taken. Notice that, if $\mathrm{M} = n$, this test degenerates in a modified version of Fmt.

It is evident that the gammainc computation is the most resource-consuming part. Using the above steps, we have reduced the test complexity for a general case ($n = 128$, $\mathrm{M} = 8$) to accommodate it for a lightweight hardware implementation:

**Algorithm 6.1** `FTwB reduction.`

Let $n = 128$ and $\mathrm{M} = 8$;
Thus $\mathrm{N} = floor\,(n/\mathrm{M}) = 16$ non-overlapping blocks;
$n - \mathrm{MN}$ bits discarded;
Assuming $\alpha = .01$, the test passes for $\mathrm{P} - \mathrm{val} \geq .01$;
Hence $gammainc(c/2, \mathrm{N}/2, '\,upper') \geq .01$;
Computing the inverse of gammainc(), fixing the second argument leads to

$$\mathbf{c \leq 16}$$

Notice that the inverse flips the disequality;
Knowing that $c = 4\mathrm{M}\sum_{\mathrm{i}=1}^{\mathrm{N}} \left(\frac{\#1\mathrm{s}}{\mathrm{M}} - .5\right)^2 \leq 16$; If a counter $x$ is used to count the number of 1s in each block then: $4\mathrm{M}\sum_{\mathrm{i}=1}^{\mathrm{N}} \left(\frac{\mathrm{x}}{\mathrm{M}} - .5\right)^2 \leq 16$; Thus $\mathbf{\sum_{i=1}^{N} (x-4)^2 \leq 32}$, is the new condition for passing FTwB.

The complex computation to estimate the P-val and hence deem a bit stream to pass/fail the FTwB with confidence $1 - \alpha$, can be reduced to a series of counter, offset calculation and squaring operations. As already proved, all computations make sense because the CDF function is monotonic (increasing) and its inverse it is monotonic too (decreasing). All the calculations involve *whole* numbers, allowing a simpler combinatorial logic implementation.

By calculating the bounds of input to gammainc function, the FTwB is optimized to a series of count, accumulate and compare operations, as shown in Fig. 6.1:

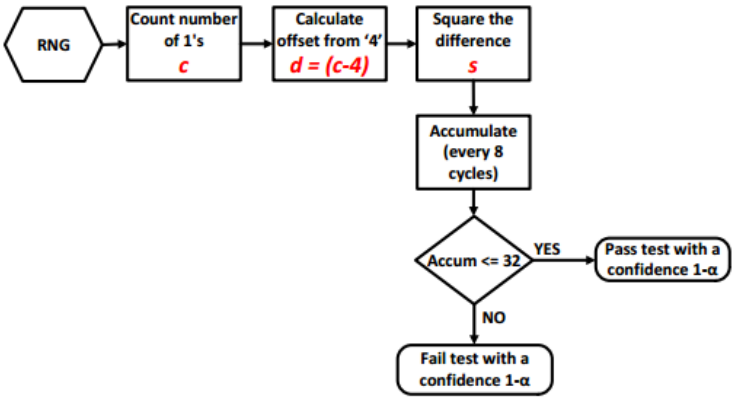
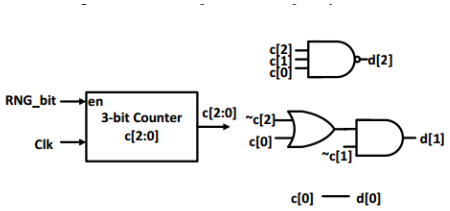
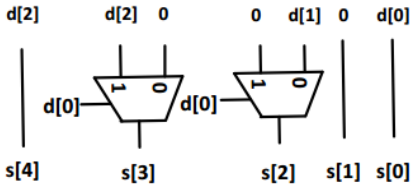



**Figure 6.1:** Flow of optimized FTwB.

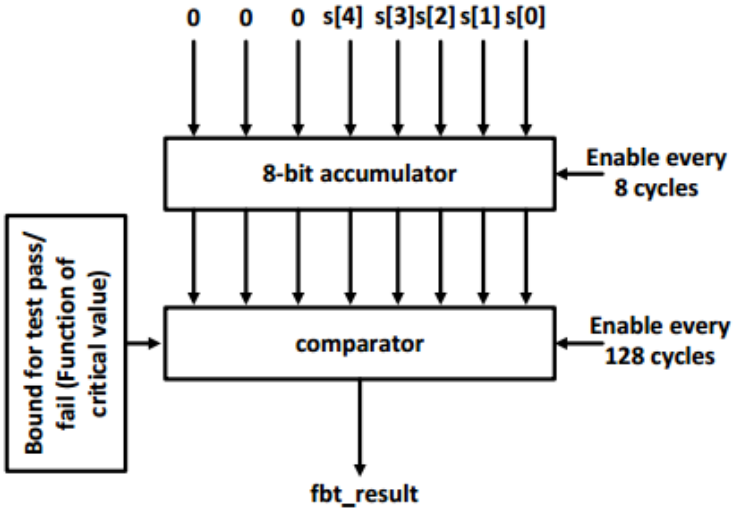
The complex gammainc function is avoided enabling a lightweight implementation. The hardware implementation for each stage of FTwB is as show in Fig. 6.2:




**(a)** Enable counter (c). Variance calculator (d).




**(b)** Squaring the variance (s).




**(c)** Accumulator and Comparator.

**Figure 6.2:** Digital logic for lightweight FTwB implementation.

Similar to FTwB, other NIST tests are also reduced to facilitate optimum hardware implementation. All the tests operate on each incoming bit from the RNG serially, thereby minimizing additional hardware to store the bits. Only for the NoTMT and the BMRT, a 10 bit and 8 bit shift registers are used respectively to store the previous bits. The counters and control logic is shared across different tests to further optimize area and power.

The partial reconfigurable feature of the reduced NIST test module provides the flexibility of choosing a different critical value $\alpha$ for each test, based on the requirement of the platform/application. The reconfigurable bound registers in each test module allows appropriate value of $\alpha$ to be set as the threshold critical value for test pass/fail. The calculated upper bounds for the inputs to erfc/gammainc functions is showed in Tab. 6.2.

| $\alpha =$ | **.001** | **.0025** | **.0040** | **.0055** | **.0070** | **.0085** | **.010** |
|---|---|---|---|---|---|---|---|
| **FMT** | 2.3268 | 2.1378 | 2.0352 | 1.9631 | 1.9070 | 1.8608 | 1.8214 |
| **FTwB** | 19.6262 | 18.2279 | 17.4898 | 16.9802 | 16.5888 | 16.2698 | 16.0000 |
| **RT** | 2.3268 | 2.1378 | 2.0352 | 1.9631 | 1.9070 | 1.8608 | 108214 |
| **LR1sT** | 8.1331 | 7.1602 | 6.6582 | 6.3168 | 6.0574 | 5.8481 | 5.6724 |
| **BMRT** | 6.96078 | 5.9915 | 5.5215 | 5.2030 | 4.9618 | 4.7677 | 4.6052 |
| **NoTMT** | 13.0622 | 11.8872 | 11.2726 | 10.8507 | 10.5280 | 10.2660 | 10.0451 |

**Table 6.2:** Input to erfc/gammainc in function of $\alpha$.

Although the calculated input bounds have fractional values, the final hardware counter/comparator bounds will be whole numbers. The script able to extract these number and all the other essential scripts will be studied in sec. 6.5.

## 6.5 MATLAB Code

A similar calculation is performed for the other five NIST tests to reduce the computation and design a lightweight implementation. I will show the *essential* scripts wrote and tested by me to reach this goal.

- Each time you encounter the word **entropy**, it refers to the Shannon entropy presented in sec. 2.7.
- The meaning of **pass/fail** bit is that the decision taken is encoded in a 1 if the test pass with a certain confidence else it is a 0.
- Every code variable and routine is written in `monospace`.

All the Functions/Scripts presented are pre-commented with the following sentence:

```
% * Copyright 2012 Daniele <Daniele@TX230>
% * This program is free software; you can
% * redistribute it and/or modify
% * it.
% * This program is distributed in the hope that
% * it will be useful, % * but WITHOUT ANY WARRANTY; without even
   the implied warranty of
% * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

## 6.5.1 Input Functions

### 6.5.1.1 `gi`

The first function is called `gi`. It takes as input the parameter `n`, and it is able to return:

- `y`: a pseudo-random input of size $n$ bits.
- `y1`: its normalized sum.

```
% Pseudo random input generator
% return BIT SEQUENCE y and NORMILIZED bit sequence y1
function [y, y1] = gi(n)
y = zeros(1,n);
y1 = zeros(1,n);
for i=1:n
    a = rand;
    if a > .5
        y(i) = 1;
        y1(i) = 1;
    else
        y(i) = 0;
        y1(i) = -1;
    end
end
end
```

### 6.5.1.2 `PRNG2`

The second function is called `PRNG2`. It takes as input the parameter `n` it is able to return:

- `ES`: matrix with all possible entropy input sequence.
- `ES_norm`: matrix with the correspondent normalized sum vectors.
- `max_en_dis`: the max entropy distance between two input sequence.

- `ENT`: vector containing all the entropy values of the input sequences.

```
% Shannon Entropy Input generator
% when n is ODD number is impossible to reach entropy = 1

function [ES,ES_norm,ENT,max_en_dis] = PRNG2 (n)
% generating all possible entropy input depends on the size n.
% input integer must be even

c=n;
step=floor(n/2+1);
ES = zeros(step,n);
ENT = zeros(0,step);
ES_norm = -1 .* ones(step,n);

for i=2:step
        ES(i:step,c)= 1; %matrix of all test input      ES_norm(i:
           step,c) = 1; %matrix of the   normalized test input
        c=c-2;
        ENT(i)=entropy(ES(i,:));
        %row vector of  entropy vaules of each ES       line
end

max_en_dis = abs(ENT(2)-ENT(1));

end
```

#### 6.5.1.3 `permsum2`

The third input script is called `permsum2`. It computes all possible combination of `m` numbers that summed results `s`. Notice that actually the script supports m=2,3,4 only.

```
% This script computes all possible combination of m numbers that
   summed
% results s (0 included). Actually the script support m=2,3,4 only
clear all
s=4;
m=2;
% compute the combination
if m==4
    i=1;
    acc=zeros(1,s);
    acc(1) = 1;
    while i<=s
        i=i+1;
        acc(i) = acc(i-1) + i;
    end
```

```
    n = sum(acc);
                                                                    16
elseif m==3
    acc = 1:1:s+1;
    n=sum(acc);                                                     19
elseif m==2
    n=s+1;
else                                                                22
    error('m can be only 2 or 3 or 4');
end
% Compute the n possible combination an store it in A              25
A = zeros(n,m);
j=1;
a=zeros(1,m);                                                       28
while j <= n
    for i=1:m
        a(i)=floor((s+1)*rand);                                     31
    end
    copy=0;
    if sum(a) == s                                                  34

            for i=1:j
                if a== A(i,:)                                       37
                    copy=1;
                elseif copy==0 && i==j
                    A(j,:) = a;                                     40
                    j = j+1;
                end
            end                                                     43

    end
end                                                                 46
```

## 6.5.2 Output Scripts

### 6.5.2.1 `back_envelope`

This output script is called `back_envelope`. It takes as parameter: `n` that is the bit length and the alpha step `s`. It generates all possible class of inputs based on entropy, using `PRNG2`, see sec. 6.5.1.2, and *reverse-test* it for `s` different value of alpha equally spaced btw .0010 and .0100, that is the common interval used for security application. The back-of-envelope calculation is performed for **FMT**,**FTwB** and **RT**.

The scripts creates and write a txt output files:

- `out_be.txt`: file that collects the results of back-of-envelope calculation, used for the on-chip hardware implementation of the FMT,FTwB and RT.

```
%The script writes matrices using the same row for different inputs but
%equal alpha value
clear all
n=128;

%decide the alpha step between .001 and .01 and create a vector of alphas
%and confidences
s=5;
st=abs((.01-.001)/(s-1));
alpha=.001:st:.01;
al=length(alpha);
conf=100-alpha.*100;

%OUTPUT FILE
fid_out = fopen('out_be.txt','w');
fprintf(fid_out, 'Analisys for:\t%d values of alpha over %d-bit input set.\n', s,n);
fclose(fid_out);

[A B] = PRNG2(n);

%%%%%Mono-bit
fid_out = fopen('out_be.txt','a');
fprintf(fid_out, '\nFrequency Mono-bit test.\n');
fclose(fid_out);
% A contains all possible class of entropy input of lenght n,

x_mono = zeros(1,al); %vector of thresholds
Y_mono = zeros(al,n/2+1); %P-val monobit matrix_mono, each row analyze an alpha
P_mono = zeros(al,n/2+1); %Pass monobit matrix_mono
up_mono=zeros(al,2);
updown_mono=zeros(al,al);
%compute
for j=1:al
x_mono(j)=erfcinv(alpha(j)); %bound if  <= x_mono test pass
    for i=1:(n/2)+1
        Y_mono(j,i) = erfc(abs( sum(B(i,:)) ) / sqrt(2*n));
        if Y_mono(j,i) >= alpha(j) && abs( sum(B(i,:)) ) <= x_mono(j)*sqrt(2*n)
            P_mono(j,i)=1;
        else
            P_mono(j,i)=0;
        end
    end
    %Hardware reduction permits starting from an upcounter to determine if
    %the test is passed or not, if the number of 1s in the sequence is
    %INSIDE the bounds
```

```
    [up_mono(j,1),up_mono(j,2)]=upcounter_FMt(n,alpha(j));
    updown_mono(j) = up_mono(j,2) - up_mono(j,1);
    %Update outputs
    fid_out = fopen('out_be.txt','a');
    fprintf(fid_out, 'alpha=%.4f\tconf=%.2f%%\t', alpha(j),conf(j))
        ;
    fprintf(fid_out,'|S|=%d\tupcount_bounds=[%d %d]\n', updown_mono
        (j),up_mono(j,1),up_mono(j,2));
    fclose(fid_out);
end



%%%%%Block: script di riferimento testblockFTB
M=8; %no multiple block possibility
N=floor(n./M);
fid_out = fopen('out_be.txt','a');
fprintf(fid_out, '\nFrequency Test within %d blocks of %d bit.\n',N
    ,M);
fclose(fid_out);
chi_FTB=zeros(al,n/2+1);
Y_FTB=zeros(al,n/2+1);
P_FTB=zeros(al,n/2+1);
X_bl=zeros(al,N);
c=zeros(1,al);
for j=1:al %j is the alpha index
    for k=1:n/2+1 %k input index
            acc=1;
            for i=1:N
                X_bl(j,i)= ( sum(A(k,acc:acc+M-1)) ./ M - .5) .^ 2;
                acc = acc+M;
            end

            chi_FTB(j,k) = 4 .* M .* sum(X_bl(j,:));
            Y_FTB(j,k) = gammainc( chi_FTB(j,k)./2, N./2, 'upper');
            if Y_FTB(j,k) < alpha(j)
                P_FTB(j,k) = 0;
            else
                P_FTB(j,k) = 1;
            end

    end
    %Hardware implementation part, back envelope calculation
        extract chi_max
    %such that test PASS with the correspondent alpha
    c(j)=chi_FTB( j,n/2+1 - sum(P_FTB(j,:)) + 1 );
    fid_out = fopen('out_be.txt','a');
    fprintf(fid_out, 'alpha=%.4f\tconf=%.2f%%\t',alpha(j),conf(j));
    fprintf(fid_out, 'chi_squared_max=%.4f\t', c(j));
    fprintf(fid_out, 'sum from 1 to %d of (2*#1s - %d)^2  <= %.4f\n
        ', N,M, M*c(j));
    fclose(fid_out);
```

```
end

%%%%%Runs Test
tau = 2 ./ sqrt(n);
fid_out = fopen('out_be.txt','a');
fprintf(fid_out, '\nRuns Test with tau=%.2f\n',tau);
fclose(fid_out);
x_run=zeros(1,al);
D_run=zeros(4,al);
RT_down=zeros(1,al);
RT_up=zeros(1,al);
for i=1:al
    x_run(i)=erfcinv(alpha(i));
    D_run(1,i)= ((x_run(i)*2*sqrt(2*n)+2*n) * up_mono(j,1)*(n-
       up_mono(j,1)) ...
       /n^2); %less or equal lower bound
    D_run(2,i)= ((x_run(i)*2*sqrt(2*n)+2*n) * up_mono(j,2)*(n-
       up_mono(j,2)) ...
       /n^2); %less or equal upper bound
    D_run(3,i)= ((-x_run(i)*2*sqrt(2*n)+2*n) * up_mono(j,1)*(n-
       up_mono(j,1)) ...
       /n^2); %greater or equal lower bound
    D_run(4,i)= ((-x_run(i)*2*sqrt(2*n)+2*n) * up_mono(j,2)*(n-
       up_mono(j,2)) ...
       /n^2); %greater or equal upper bound
   RT_up(i)=floor(D_run(1,i));
   RT_down(i)=ceil(D_run(3,i));
    fid_out = fopen('out_be.txt','a');
    fprintf(fid_out, 'alpha=%.4f\t%.4f\t%.4f\t%.4f\t%.4f\t',alpha(i
       ),D_run(:,i));
    fprintf(fid_out, '--> RT bound=[%d %d]\n', RT_down(i),RT_up(i))
       ;
    fclose(fid_out);
end

%%%%%Final Comments
%this code is NOT parallelized
```

**E.g. output file for `n=128` and `s=5`**

```
Analysis for: 5 values of alpha over 128-bit input set.
Frequency Mono-bit test.
alpha=0.0010 conf=99.90% |S|=36 upcount_bounds=[46 82]
alpha=0.0033 conf=99.67% |S|=32 upcount_bounds=[48 80]
alpha=0.0055 conf=99.45% |S|=30 upcount_bounds=[49 79]
alpha=0.0077 conf=99.22% |S|=30 upcount_bounds=[49 79]
alpha=0.0100 conf=99.00% |S|=28 upcount_bounds=[50 78]
Frequency Test within 16 blocks of 8 bit.
alpha=0.0010 conf=99.90% chi_squared_max=36.5000 sum from 1 to 16 of (2*#1s
- 8)^2 <= 292.0000
alpha=0.0033 conf=99.67% chi_squared_max=34.0000 sum from 1 to 16 of (2*#1s
- 8)^2 <= 272.0000
alpha=0.0055 conf=99.45% chi_squared_max=32.5000 sum from 1 to 16 of (2*#1s
- 8)^2 <= 260.0000
alpha=0.0077 conf=99.22% chi_squared_max=32.5000 sum from 1 to 16 of (2*#1s
- 8)^2 <= 260.0000
alpha=0.0100 conf=99.00% chi_squared_max=28.5000 sum from 1 to 16 of (2*#1s
- 8)^2 <= 228.0000
Runs Test with tau=0.18
alpha=0.0010 78.6608 78.6608 43.2142 43.2142 -> RT bound=[44 78]
alpha=0.0033 76.7892 76.7892 45.0858 45.0858 -> RT bound=[46 76]
alpha=0.0055 75.8905 75.8905 45.9845 45.9845 -> RT bound=[46 75]
alpha=0.0077 75.2796 75.2796 46.5954 46.5954 -> RT bound=[47 75]
alpha=0.0100 74.8113 74.8113 47.0637 47.0637 -> RT bound=[48 74]
```

#### 6.5.2.2 `BMRT342_3`

This output script is called `BMRT342_3`. It implements the BMRT in the case of `alpha=.01`, `M=Q=3` and `n=342`. The parameter `sim` dictates the number of simulation performed, each simulation test a pseudo-random input generated by `gi` function, see sec. 6.5.1.1. The rank computation is performed using the built-in MATLAB function `rank`.

The scripts creates and writes into two txt files

- `in_BMRT342_3`: file that contains all inputs divided in row
- `out_BMRT342_3`: file that contains pass/fail bit for all `sim` value.

```
% This script simulate Binary rank test in the specific case of M=Q
    =3 and
% n equal to 342 bits                                                2
clear all
%%%%%Initialization
n=342; %n must be >= 38MQ                                            5
alpha=.01;
```

```
sim=10;
p = zeros(1,sim);
M=3;
Q=3;
N=floor(n/(M*Q)); %in this case is 38
% For each M=Q three probabilities must be computed following the
   general formula
% the number after a indicate the M value
%a32 = [.288788, .577576, .128350];
a3 = [.328125, .57421875, .095703125];
%a4 = [.3076171875, .5767822265625, .112152099609375];
%INPUT FILE
fid_in = fopen('in_BMRT342_3.txt','w');
fprintf(fid_in, 'INPUT:\t%d samples of %d bits\t%dx%d matrices\n\n
   ',sim, n, M, Q);
fclose(fid_in);
%OUTPUT FILE
fid_out = fopen('out_BMRT342_3.txt','w');
fprintf(fid_out, 'OUTPUT:\t%d samples of %d bits\t%dx%d matrices.\n
   ', sim,n,M,Q);
fprintf(fid_out, '\n1 means test passed with %.2f%% confidence\t',
   100-alpha*100);
fprintf(fid_out, 'alpha = %.2f.', alpha);
fprintf(fid_out, '\nEach row is a test result for a different input
   .\n\n');
fclose(fid_out);

%%%%%Compute
gg=1;
while gg<=sim
a=gi(n); %generate a pseudorandom input
fid_in = fopen('in_BMRT342_3.txt','a');
fprintf(fid_in, '%d ', a);
fprintf(fid_in, '\n');
fclose(fid_in);
%notice that this test discard n-N*Q*M bits
cr = zeros(1,3); %vector that count the ranks cr(1) counts the full
    ranks

%generate the rank vector of N MbyQ matrix
s=0;
R(1) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(2) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(3) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(4) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(5) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(6) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
```

```
s=s+9;                                                                    53
R(7) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(8) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);                         56
s=s+9;
R(9) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;                                                                    59
R(10) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
                                                                          62
R(11) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(12) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);                        65
s=s+9;
R(13) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;                                                                    68
R(14) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(15) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);                        71
s=s+9;
R(16) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;                                                                    74
R(17) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(18) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);                        77
s=s+9;
R(19) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;                                                                    80
R(20) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
                                                                          83
R(21) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(22) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);                        86
s=s+9;
R(23) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;                                                                    89
R(24) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(25) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);                        92
s=s+9;
R(26) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;                                                                    95
R(27) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(28) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);                        98
s=s+9;
R(29) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;                                                                    101
R(30) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
                                                                          104
```

```
R(31) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(32) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(33) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(34) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(35) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(36) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(37) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;
R(38) = rank ([a(1+s:3+s);a(4+s:6+s);a(7+s:9+s)]);
s=s+9;

%A2 = [a(10:12);a(13:15);a(16:18)];

%loop to count the rank type, probably it is more efficient to make
    an if
%after each R formation
for i=1:N
    if R(i) == M
        cr(1)=cr(1)+1; %full rank matrix
    elseif R(i) == M-1
        cr(2)=cr(2)+1;
    else
        cr(3)=cr(3)+1;
    end
end


chi = (cr(1) - a3(1)*N)^2 / (a3(1)*N) + (cr(2) - a3(2)*N)^2 / (a3
   (2)*N) + ...
    (cr(3) - a3(3)*N)^2 / (a3(3)*N);

%y = gammainc(chi/2,1,'upper' );
y = exp(-chi/2);

if y >= alpha
    p = 1;
else
    p = 0;
end
%Print the content in outputs128.txt
fid_out = fopen('out_BMRT342_3.txt','a');
fprintf(fid_out, '%d\t', p);
fprintf(fid_out, '\n');
fclose(fid_out);
%Increment for the next input
gg=gg+1;
```

```
end                                                                    155
```

#### 6.5.2.3 `TLROB_stat`

This output script is called `TLROB_stat`. It implements the reverse LR1sT for `M=8`. It uses the `permsum2` script, see sec. 6.5.1.3, to compute all the possible combination of 4 numbers that sums 16, zero included. Notice that the resultant matrix is a `969x4`, it is loaded inside the script and used for computation.

The scripts returns:

- `R`: matrix where each row contains results for a certain input:
  - The first column contains the P-val.
  - The second column contains the pass/fail bit.
  - The other four columns contain the correspondent quadruplet in input.

```
% Reverse the reduced TLROB M=8                                        1
% Matrix A is generated using permsum2 script
% the is loaded. A contains all possible combination of 4 numbers
    that
% sums 16, zero included that are 969 rows                            4
clear all
load('4 number that sum 16.mat')
                                                                       7
n=128;
M=8;
N=floor(n/M);                                                          10
K=3;
c = [ .2148 .3672 .2305 .1875 ];
alpha=.01;                                                             13
F=zeros(length(A(:,1)),K+1);
chi_obs=zeros( length(A(:,1)) , 1 );
y=zeros( length(A(:,1)) , 1 );                                         16
fault=ones( length(A(:,1)) ,1 );

for j=1:length(A(:,1))                                                 19
    for i=1:K+1
    F(j,i) = ( A(j,i) - N.*c(i) ).^2 ./ (N .* c(i) );
    end                                                                22
    %compute the statistic
    chi_obs(j,1) = sum(F(j,:));
                                                                       25
    %compute the incomplete gamma function
    y(j,1) = gammainc( chi_obs(j)./2,K./2,'upper');
    if y(j,1) < alpha                                                  28
        fault(j,1)=0;
    end
```

```
end

R(:,1)=y;
R(:,2)=fault;

R(:,3:6)=cast(A,'uint8');

%Each row is a test for a particular input. The first column
   contains all
%P-values the second contains a 1 if the P-value is >= alpha and
   the last
%4 columns are all the possible input combination ofr n=128, namely
    all
%the possible 4 numbers, 0 included, that sum gives 16.
```

#### 6.5.2.4 NoTMt3

This output script is called `NoTMt3`. It is able to generate `sim` pseudo-random input of `n=512` bit an test it for `m=9` bit entropy classified templates, using `PRNG2` function, see sec. 6.5.1.2. Each template gives a P-value and a pass/fail bit.

The scripts creates and writes into two txt files:

- `in_no512_9`: file that contains all inputs divided in row
- `out_no512_9`: file that contains the list of all tested templates and a `sim x m` matrix where each column is the pass/fail bit for a certain template and a summary section where they are showed the percentages.

```
clear all
%script able to generate sim pseudo-random input of n bit an test
   it for all
%possible m bit templates and/or entropy classified templates.
%each template gives a P-value

%%%%%Initialization
n = 512;
alpha = .01;
sim=128;
%generate templates
m = 9; %NIST recommend 9,10 tem lenght. With m=9 I will generate 5
   templates
Q = PRNG2(m); %
%Q = all_input(m);
N = 8;
M = floor(n/N);
mu = (M-m+1) / 2^m;
var = M * ( 1/2^m - (2*m-1)/2^(2*m) );
%Generate Pass Matrix
```

```
P = zeros(sim,length(Q(:,2)));

%INPUT FILE
fid_in = fopen('in_no512_9.txt','w');
fprintf(fid_in, 'INPUT:\t%d samples of %d bits\n\n',sim, n);
fclose(fid_in);
%OUTPUT FILE
fid_out = fopen('out_no512_9.txt','w');
fprintf(fid_out, 'OUTPUT:\t%d samples of %d bits.\n', sim,n);
fprintf(fid_out, '\nList of %d-bit used templates:\n',m);
fprintf(fid_out, '%d %d %d %d %d %d %d %d %d\n',Q');
fprintf(fid_out, '\nEach sample provides %d P-values',length(Q(:,2)
   ));
fprintf(fid_out, '\n1 means test passed with %.2f%% confidence\t',
   100-alpha*100);
fprintf(fid_out, 'alpha = %.2f.', alpha);
fprintf(fid_out, '\nColumn order:(1)First temp\t(2)Second temp\t');
fprintf(fid_out, '...\tLast temp\n\n');
fclose(fid_out);

%%%%%Provide sim inputs of n bits
load rep512

%%%%%Compute
gg=1;
while gg <= sim
%a = gi(n); %generate a pseudo-random input
a = cc(gg,:);
%initialize local variables
W = zeros(length(Q(:,2)),N);
C = zeros(length(Q(:,2)),N);
y = zeros(1,length(Q(:,2)));
p = zeros(1,length(Q(:,2)));
chi_obs = zeros(1,length(Q(:,2)));
%save the input in row in in_no512_9.txt
fid_in = fopen('in_no512_9.txt','a');
fprintf(fid_in, '%d ', a);
fprintf(fid_in, '\n');
fclose(fid_in);
%for each block control a template
 for k=1:length(Q(:,2)) %test all possible entropy input
    %first block outside the loop because of shift problem
    ii = 1;
    while ii <= M-m+1
        if (a(ii:ii+m-1) == Q(k,:))
            W(k,1) = W(k,1) + 1;
            ii = ii + m;
        else
            ii = ii + 1;
         end

    end
```

```
    C(k,1) = (W(k,1)-mu)^2;
                                                                    70
    shift = M;
    for i=2:N
    %scan a block                                                   73
        j = 1;
        while j <= M-m+1 %maximum iteration possible
            if (a(j+shift:j+shift +m-1) == Q(k,:))                  76
                W(k,i) = W(k,i) + 1;
                % raws identifies the template number and colums
                   the block number
                j = j + m;                                          79
            else
                j = j + 1;
                                                                    82
            end

        end                                                         85
        C(k,i) = (W(k,i)-mu)^2;
        shift = shift + M;
    end                                                             88
    %compute the P-value for the k templates
    chi_obs(k) = sum(C(k,:))/var;
    y(k) = gammainc(chi_obs(k)/2, N/2, 'upper');                    91
    if y(k) < alpha
        p(k)=0;
    else                                                            94
        p(k)=1;
    end
                                                                    97
 end
%Print the content in outputs128.txt
fid_out = fopen('out_no512_9.txt','a');                             100
fprintf(fid_out, '%d\t', p);
fprintf(fid_out, '\n');
fclose(fid_out);                                                    103
%Increment for the next input
P(gg,:)=p;
gg=gg+1;                                                            106
end

%Write RESULTS SUMMARY                                              109
fid_out = fopen('out_no512_9.txt','a');
fprintf(fid_out, '\n');
fprintf(fid_out, 'RESULTS SUMMARY: %d samples of %d-bit alpha=%.4f  112
   confidence %.2f%%.\n', sim,n,alpha,100-alpha*100);
fprintf(fid_out, 'Testing %d templates of %d-bit.\n\n', length(Q
   (:,2)),m);
for i=1:length(Q(:,2))
    fprintf(fid_out, 'NonOver Temp#%d:[%d %d %d %d %d %d %d %d %d  115
       ]', i,Q(i,:));
    fprintf(fid_out, '\t%d pass over %d\tperc: %%%.4f\n', sum(P(:,i
```

```
        )),sim,sum(P(:,i)) / sim * 100 );
end
%fprintf(fid_out, 'Sequences that pass all the test');                    118
fclose(fid_out);

%%%%%%GENERAL COMMENTS                                                    121
%the output file generated provides a P-value for each templates in
%columns. So one row identifies the result of a single input with
    multiple
%templates tested.                                                        124
% sim time with this config 5 min
% code can be optimized avoiding the declaration of local variables
```

**E.g. output file for `n=512, m=9, sim=128`**

OUTPUT: 128 samples of 512 bits.
List of 9-bit used templates:
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 1
0 0 0 0 1 0 1 0 1
0 0 1 0 1 0 1 0 1
Each sample provides 5 P-values
1 means test passed with 99.00% confidence alpha = 0.01.
Column order:(1)First temp (2)Second temp ... Last temp
1 1 1 1 1
1 1 1 1 1
.....
for reason of space I need to hide some results
.....
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
RESULTS SUMMARY: 128 samples of 512-bit alpha=0.0100 confidence 99.00%.
Testing 5 templates of 9-bit.
NonOver Temp#1:[0 0 0 0 0 0 0 0 0] 128 pass over 128 perc: %100.0000
NonOver Temp#2:[0 0 0 0 0 0 0 0 1] 128 pass over 128 perc: %100.0000
NonOver Temp#3:[0 0 0 0 0 0 1 0 1] 128 pass over 128 perc: %100.0000
NonOver Temp#4:[0 0 0 0 1 0 1 0 1] 128 pass over 128 perc: %100.0000
NonOver Temp#5:[0 0 1 0 1 0 1 0 1] 125 pass over 128 perc: %97.6563

#### 6.5.2.5 `intelHS`

This output script is called `intelHS`. It is able to generate `sim` pseudo-random input of `n=256` bit an test the frequency of six pre-defined patterns using the empirical law by Intel showed in Tab. 3.1.

The scripts creates and writes into two txt files:

- `in_intel`: file that contains all inputs divided in row.
- `out_intel`: file that contains the list of all tested templates and a `sim x 6` matrix where each column is the pass/fail bit for a certain template and a summary section where are showed the percentages.

---

```
%%%%Intel health and sweelness test
                                                                    2
%%%%%Initiali
clear all
close all                                                           5
n=256; %value fixed by Intel
sim = 256;
P = zeros(sim,6); %results matrix                                   8

%INPUT FILE
fid_in = fopen('in_intel.txt','w');                                 11
fprintf(fid_in, 'INPUT:\t%d samples of %d bits\n\n\n',sim, n);
fclose(fid_in);
%OUTPUT FILE                                                        14
fid_out = fopen('out_intel.txt','w');
fprintf(fid_out, 'OUTPUT:\t%d samples of %d bits.\t', sim,n);
fprintf(fid_out, '1 means test passed with %.2f%% confidence\t',    17
   99);
fprintf(fid_out, 'alpha = %.2f.', 0.01);
fprintf(fid_out, '\nColumn order:(1) [1]  (2) [01]  ');
fprintf(fid_out, '(3) [010]  (4) [0110]  (5) [101]  (6) [1001]\n\n  20
   ');
fclose(fid_out);

%%%%%Provide some input and load it into v()                         23
load rep4
v=cc;
%a=zeros(1,n);                                                      26
% for i=1:sim
%     v(i,:)= %some loaded file
% end                                                               29
%%%%%compute
 gg=1;
 while gg <= sim                                                    32
a=v(gg,:);
fid_in = fopen('in_intel.txt','a');
fprintf(fid_in, '%d ', a);                                          35
fprintf(fid_in, '\n');
fclose(fid_in);
c=zeros(1,6); %counting vector                                      38
p=zeros(1,6); %pass of fail vector

%1 bit template                                                     41
c(1)=sum(a);
if  c(1) > 109 && c(1) < 165
```

```
    p(1) = 1; %no else because initialized at 0
end

%2 bits templates
t2 = [0 1];
for i=1:n-1
    if a(i:i+1) == t2
        c(2)=c(2)+1;
    end
end
if c(2) > 46 && c(2) < 84
    p(2) = 1;
end

%3 bits templates
t3 = [0 1 0];
t5 = [1 0 1];
for i=1:n-2
    if a(i:i+2) == t3
        c(3)= c(3) + 1;
    end

    if a(i:i+2) == t5
        c(5) = c(5) + 1;
    end
end
if c(3) > 8 && c(3) < 58
    p(3)=1;
end
if c(5) > 8 && c(5) < 58
    p(5)=1;
end

%4 bits templates
t4 = [0 1 1 0];
t6 = [1 0 0 1];
for i=1:n-3
    if a(i:i+3) == t4
        c(4)= c(4) + 1;
    end

    if a(i:i+3) == t6
        c(6) = c(6) + 1;
    end
end
if c(4) > 2 && c(4) < 35
    p(4)=1;
end
if c(6) > 2 && c(6) < 35
    p(6)=1;
end
P(gg,:)=p;
```

```
%Print the results in a output file
fid_out = fopen('out_intel.txt','a');
fprintf(fid_out, '%d\t', p);
fprintf(fid_out, '\n');
fclose(fid_out);
gg = gg+1;
 end

 %Write RESULTS SUMMARY
fid_out = fopen('out_intel.txt','a');
fprintf(fid_out, '\n');
fprintf(fid_out, 'RESULTS SUMMARY: %d samples of %d-bit alpha
   =0.0100 confidence 99.00%%.\n', sim,n);
fprintf(fid_out, '[1] pattern\t%d pass over %d\tperc: %%%.4f\n',
   sum(P(:,1)),sim,sum(P(:,1)) / sim * 100 );
fprintf(fid_out, '[01] pattern\t%d pass over %d\tperc: %%%.4f\n',
   sum(P(:,2)),sim,sum(P(:,2)) / sim * 100 );
fprintf(fid_out, '[010] pattern\t%d pass over %d\tperc: %%%.4f\n',
   sum(P(:,3)),sim,sum(P(:,3)) / sim * 100 );
fprintf(fid_out, '[0110] pattern\t%d pass over %d\tperc: %%%.4f\n',
    sum(P(:,4)),sim,sum(P(:,4)) / sim * 100 );
fprintf(fid_out, '[101] pattern\t%d pass over %d\tperc: %%%.4f\n',
   sum(P(:,5)),sim,sum(P(:,5)) / sim * 100 );
fprintf(fid_out, '[1001] pattern\t%d pass over %d\tperc: %%%.4f\n',
    sum(P(:,6)),sim,sum(P(:,6)) / sim * 100 );
%fprintf(fid_out, 'Sequences that pass all the test');
fclose(fid_out);

%this code can be parallelized in a single while
% it seems that using Pseudorandom input always pass all the tests,
% weakness of Intel RNG ????
% alpha is implicitly set at .01
% a sample is healthy if passes all the template counting
% Intel scans the last 256 256-bit samples, if at least 128 samples
    are
% healthy then the ES is in a SWELL state
```

**E.g. output file for `n=256, sim=256`**
OUTPUT: 256 samples of 256 bits. 1 means test passed with 99.00% confidence alpha = 0.01.
Column order:(1) [1] (2) [01] (3) [010] (4) [0110] (5) [101] (6) [1001]
1 1 1 1 1 1
1 1 1 1 1 1
.....
for reason of space I need to hide some results
.....
1 1 1 1 1 1
0 1 1 1 1 1
RESULTS SUMMARY: 256 samples of 256-bit alpha=0.0100 confidence 99.00%.
[1] pattern 205 pass over 256 perc: %80.0781
[01] pattern 256 pass over 256 perc: %100.0000
[010] pattern 256 pass over 256 perc: %100.0000
[0110] pattern 256 pass over 256 perc: %100.0000
[101] pattern 256 pass over 256 perc: %100.0000
[1001] pattern 255 pass over 256 perc: %99.6094

#### 6.5.2.6 `arginv`

This output script is called `arginv`. Given the bit length `n`, the block length `M` and the alpha step `s`, the script is able to generate :

- `X`: `6 x s` matrix containing all the threshold values for different alpha values. Notice that this script was used to fill Tab. 6.2.

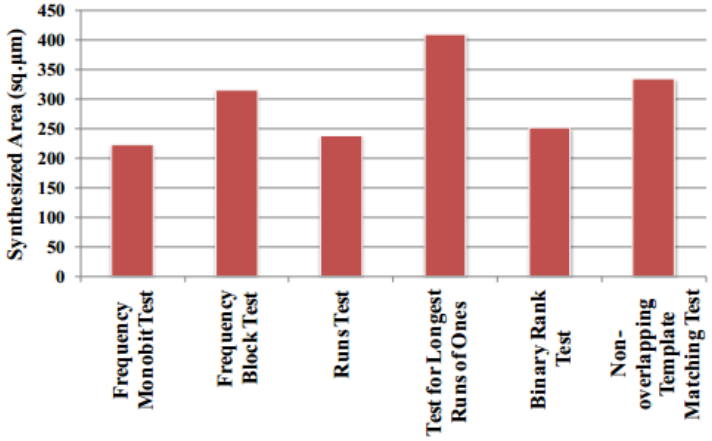
```
%Script that computes the arg of 6 light tests
clear all
nt=6; %number of tests implemented
% Monobit, Block, Run
n=128;
M=8; %Block
N=floor(n/M);
K=3;
%Non overlapping and Binary Matrix
nn=512;
m=9;
NN=8;
MM=floor(nn/NN);
mu = (MM-m+1) / 2^m;
var = MM * ( 1/2^m - (2*m-1)/2^(2*m) );
%decide the alpha step between .001 and .01 and create a vector of
    alphas
%and confidences
s=7;
st=abs((.01-.001)/(s-1));
alpha=.001:st:.01;
```

```
al=length(alpha);
conf=100-alpha.*100;
                                                                    23
x = zeros(nt,al); %vector of thresholds
s = zeros(nt,al);
                                                                    26
for j=1:al

    %Input                                                          29
    x(1,j) = erfcinv(alpha(j)); %Monobit
    x(2,j) = gammaincinv(alpha(j), N./2, 'upper'); %Block
    x(3,j) = erfcinv(alpha(j)); %Run Test                          32
    x(4,j) = gammaincinv(alpha(j), K/2, 'upper'); %Longest Run of
       ones
    x(5,j) = gammaincinv(alpha(j),NN/2,'upper' ); %Non Overlapping
    x(6,j) = gammaincinv(alpha(j),1,'upper' ); %Binary             35
    %Computed statistics
%     s(1,j) = sqrt(2) .* erfcinv(alpha(j)); %Monobit
%     s(2,j) = 2 .* gammaincinv(alpha(j), N./2, 'upper'); %Block   38
%     s(5,j) = 2 * gammaincinv(alpha(j),NN/2,'upper' );
%     s(6,j) = 2 * gammaincinv(alpha(j),1,'upper' );
                                                                    41
end

x(nt+1,:)=alpha;                                                    44
```

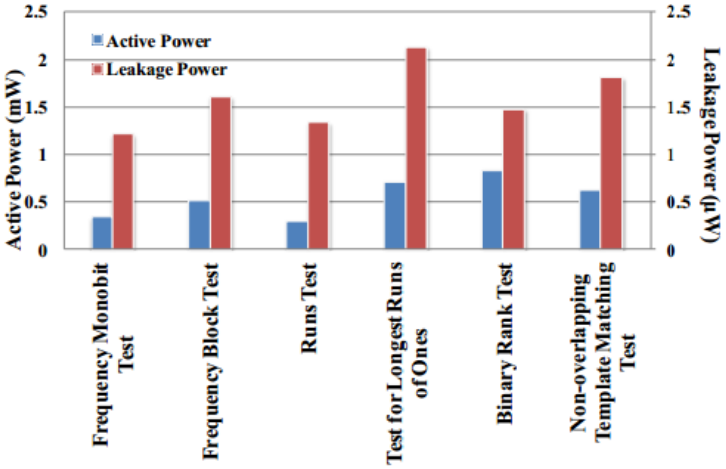
## 6.6 Logic Implementation and Results

The lightweight implementation for reduced NIST test suite ( six tests) was design in Verilog, and verified for functionality using ModelSim. The designs were synthesized in Synopsys Design Compiler using 45 nm SOI NCSU/OSU Open Source Standard Cell Library. The synthesized designs were optimized for a cycle time of .5 ns (2 GHz).

The area and power numbers for each 128 bit test are shown in 6.3a and 6.3b respectively:




**(a)** Synthesized Area $\mu m^2$.




**(b)** Active and Leakage power mW.

**Figure 6.3:** Area and Power.

The synthesized area ranges from 240 $\mu m^2$ to 460 $\mu m^2$ for each test. The shared control logic and counters reduce the overall implementation area. The common counter and control logic consume an area of around 200 $\mu m^2$, resulting the overall NIST module area of 1926 $\mu m^2$. This translates to 1026 NAND gates equivalent in 45 nm technology.

The active power for each test is of the order of 0.4 mW to 0.8 mW. All the tests are designed to operate in parallel, resulting in a overall active power of 3.75 mW for the NIST module operating at 2GHz. The overall cell leakage is $\sim$ 10.5 $\mu$W which is 0.28% of the total power. Since the target applications include passively powered and battery operated devices, like RFIDs and smart cards, *energy/bit* is an important metric. The 128-bit reduced NIST module operating on 2Gbps consumes 1.87 pJ/bit.

The proposed implementation is scalable to larger number of bit sample as well. Depending on the number of bits $n$ and block sizes, the bounds for each test varies. BMRT and NoTMT are implemented with a minimum bit sample size greater that 512 bits. The optimized tests can be scaled for larger bit sequence range at the cost of increased area and power. The area, power and energy/bit for 256-bit and 512-bit implementations are showed in Tab. 6.3:

**Table 6.3:** 256 and 512 bit implementations (45 nm tech).

| Bit lengths | 256 bits | 512 bits |
|---|---|---|
| Area $\mu m^2$ | 2394 | 2787 |
| Power mW | 4.03 | 4.37 |
| Energy/bit @ 2Gbps pJ/bit | 2.01 | 2.18 |

## 6.7 Conclusions and Future Work

A lightweight implementation of reduced set of NIST randomness test suite is proposed in this work. The implementation of the six NIST tests is based on optimized calculations to bypass the computation intensive *erfc* and *gammainc* functions. The back-of-envelope calculation permits to determine the pass/fail bit directly using the argument of the complex functions, *avoiding* their computations. The resource-intensive statistical tests are converted to a series of count, add and compare operations, implemented using combinatorial logic. The design is further optimized for a fixed bit sample size and share global counters and control signals. A partial reconfiguration feature allows the critical value to be changed in the form of modified bounds for each test. The tests operate serially on each incoming bit from the RNG, thereby avoiding additional hardware for storage. The design for 128-bit tests has a synthesized area of 1926 $\mu$m$^2$, for an optimized cycle time of 0.5 ns. The six tests operate in *parallel*, consuming an active power of 3.75 mW and leakage power of 10.5 $\mu$W.

As a part of future work, we propose to explore lightweight implementation of other statistical tests from NIST and DIEHARD test suites, e.g. CST, and expand to *second* level tests for analyzing the distribution of P-vals.

# Conclusions

In chapter 1 I introduced the foundations of RNG. True randomness versus Pseudo randomness. The basic block scheme of a TRNG, defining the concept of ES, EH and DPP, with some examples. The common metrics used to evaluate a security IC.

In chapter 2 I introduced the basic list of tools to threat randomness. Thermal Noise modeling. Fourier transformation. Elements of Probability Theory and Stochastic Process. The Quantization process. The Gaussian statistics. The abstract definition of entropy in Information Theory sense and the definition of Hamming Distance. The statement of the CLT and one particular case: de Moivre-Laplace Thm. . The Pearson's Chi-Squared test.

In chapter 3 I explained some examples of RNG implementation from an architectural point of view. The analog ADC pipeline that exploit chaos used both for random number generation and analog-to-digital signal conversion. The digital SRAM cells used in metastable state each power-up for random number generation and fingerprint extraction. The Intel RNG a.k.a. Bull Mountain integrated in the third gen processors (Ivy Bridge) used as a dedicated, scalable (atomic instruction) and high security RNG. Some examples of open-source TRNGs: HotBits and LavaRnd. The most famous PRNG example: Rule 30 of Wolfram's CAs, used in for pseudo-random number generation in Mathematica.

In chapter 4 I explained how to math model and evaluate some RNG implementation. The analog chaotic ADC modeled by Markovian SP, namely Piece-Wise Affine Markov maps. The digital metastable SRAM cell that exploits FERNS method and the intrinsic process variations that affect its performance. The random number generation mechanism in the Bull Mountain RNG: failure modes, OHT unit, BIST and DPP algorithm.

In chapter 5 I introduced the testing methodologies for RNG. The most famous fail in the history of encryption: cracking DES, OpenSSL and ECDSA. The Statistical Test as basic block for RNG evaluation: scheme, critical value, null hypothesis, related errors and confidence. RNG test suites: DIEHARD, dieharder and NIST SP 800-22. Focus on NIST SP 800-22: erfc based class E and Chi-Square based class C tests. Summary of all NHs and flowcharts for FMT, FTwB, RT, LR1sT, BMRT, NoTMT, OTMT, CST.

In chapter 6 I showed the work made in Oct-Dec 2012 for the publication of the paper titled: **"On-chip Lightweight Implementation of Reduced NIST Randomness Test Suite [32]"** at UMass College (MA), that is the core of my thesis work.

The abstract summarized the paper. The motivations underline why we have made this on-chip test module. The reduction part explains how we work on the

software to fully-hardware conversion of the test suite. The FTwB lightweight implementation is used as a general example to understand in the details all the steps we made and which is the *hack* that we used. The MATLAB code section, focus on my personal work and explains the basic script used to reach the goal. The input functions and the outputs scripts are used to compute threshold values useful in the hardware implementation. The logic implementation part describes the main steps for the realization of the on-chip test module and shows the results in terms of area, throughput, power and energy/bit. The conclusion part review the proposed work, and propose new features and challenges for future works.

# List of Figures

# List of Tables

# List of Algorithms

# Bibliography


[1] *"DSA-1571-1- openssl – predictable random number generator,"*, 2008.

[2] Ross Anderson, *Security Engineering*, 2008.

[3] EB Barker and JM Kelsey, *Recommendation for random number generation using deterministic random bit generators (revised)*, (2012), no. January.

[4] R Brown, *Brownian Motion*, **1** (1827).

[5] R. G. Brown, D. Eddelbuettel, and D. Bauer, *Dieharder: A random number test suite*, 2011.

[6] Sergio Callegari, Riccardo Rovatti, Senior Member, and Gianluca Setti, *Embeddable ADC-Based TRNG for Crypto Applications Exploiting Nonlinear Signal Processing and Chaos*, **53** (2005), no. 2, 793–805.

[7] Morris Dworkin, *Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality*, 2004.

[8] G. Marsaglia, *DIEHARD Statistical Tests.*

[9] RW Hamming, *Error detecting and error correcting codes*, Bell System technical journal (1950).

[10] Eric J. Hoffman, *US5706218*, 1998.

[11] ———, *US6061702*, 2000.

[12] Daniel E Holcomb and Wayne P Burleson, *Initial SRAM state as a fingerprint and source of true random numbers for RFID tags*, of the Conference on RFID (2007).

[13] Daniel E Holcomb, Student Member, Wayne P Burleson, Senior Member, and Kevin Fu, *Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers*, Cell **58** (2009), no. 9, 1198–1210.

[14] Dan Hotoleanu, Octavian Cret, Alin Suciu, Tamas Gyorfi, and Lucia Vacariu, *Real-Time Testing of True Random Number Generators Through Dynamic Reconfiguration*, 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (2010), 247–250.

[15] J B Johnson, *Thermal Agitation of Electricity in Conductors*, Physical Review **32** (1928), no. 1, 97–109.

[16] Benjamin Jun and Paul Kocher, *The Intel random number generator*, Cryptography Research Inc. white paper **27** (1999), no. July 1948, 1–8.

[17] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall, *Cryptanalytic attacks on pseudorandom number generators*, Fast Software Encryption (1998).

[18] Wolfgang Killmann, *Functionality classes and evaluation methodology*, (2001), 1–38.

[19] SJ Kim, Ken Umeno, and Akio Hasegawa, *Corrections of the NIST statistical test suite for randomness*, arXiv preprint nlin/0401040 (2004).

[20] Paul Kocher and Mark E Marson, *ANALYSIS OF INTEL IVY BRIDGE DIGITAL RANDOM NUMBER G ENERATOR*, (2012).

[21] IN Kovalenko, *Distribution of the linear rank of a random matrix*, Theory of Probability & Its Applications (1973), no. 2, 342–347.

[22] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone, *Handbook of Applied Cryptography*, (1996).

[23] H Nyquist, *Thermal agitation of electric charge in conductors*, Physical Review (1928).

[24] C Of, HB Ronald, and HB James, *Security Requirements For Cryptographic Modules*, ukpmc.ac.uk (2002).

[25] Fabio Pareschi, Gianluca Setti, and Riccardo Rovatti, *Implementation and Testing of High-Speed CMOS True Random Number Generators Based on Chaotic Systems*, IEEE Transactions on Circuits and Systems I: Regular Papers **57** (2010), no. 12, 3124–3137.

[26] Riccardo Rovatti, Sergio Callegari, and Gianluca Setti, *On the convergence to regime of ADC-based true random number generators*, 2007 18th European Conference on Circuit Theory and Design **2** (2007), no. 1, 635–638.

[27] Andrew Rukhin, Juan Soto, James Nechvatal, M Smid, and Elaine Barker, *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, (2001), no. April.

[28] Werner Schindler and Wolfgang Killmann, *Evaluation Criteria for True ( Physical ) Random Number Generators Used in Cryptographic Applications*, Informationstechnik **2523** (2002), 431–449.

[29] W Schottky, *Small-shot effect and flicker effect*, Physical Review (1926).

[30] G. Setti, G. Mazzini, R. Rovatti, and S. Callegari, *Statistical modeling of discrete-time chaotic processes-basic finite-dimensional tools and applications*, Proceedings of the IEEE **90** (2002), no. 5, 662–690.

[31] C E Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal **27** (1948), no. July 1928, 379–423.

[32] Vikram B Suresh, Daniele Antonioli, and Wayne P Burleson, *On-chip Lightweight Implementation of Reduced NIST Randomness Test Suite*, appear in IEEE Hardware Oriented Security and Trust (HOST) Symposium, Austin TX (2013).

[33] Vikram B Suresh and Wayne P Burleson, *Entropy Extraction in Metastability-based TRNG*, Computer Engineering (2010).

[34] S Wolfram, *Statistical mechanics of cellular automata*, Reviews of modern physics (1983).

[35] K Yuksel, JP Kaps, and Berk Sunar, *Universal hash functions for emerging ultra-low-power networks*, ... of the Communications Networks ... (2004).

# Nomenclature

| | |
|---|---|
| ADC | Analog to Digital Converter |
| AES | Advanced Encryption Standard |
| AT-and-T | American Telephone and Telegraph |
| AWGN | Additive White Gaussian Noise |
| BIST | Built-In-Self-Test |
| c-t | continous-time |
| CA | Cellular Automata |
| CCD | Charge-Coupled Device |
| CDF | Cummulative Distribution Function |
| CEP buffer | Conditioned Entropy Pool buffer |
| CLT | Central Limit Theorem |
| CMOS | Complementary Metal-Oxide-Semiconductor |
| CRC | Cyclic Redundancy Check |
| CRI | Cryptography Research, Inc. |
| CSPRNG | Criptographically Secure Pseudo Random Number Generator |
| CTF | Channel Transfer Function |
| d-t | discrete-time |
| Das | Digitized analog signal |
| DDP | Digital Post Processing |
| DFT | Discrete Fourier Transform |
| DoS | Denial of service |

| | |
|---|---|
| DRBG | Deteministic Random Bit Generator |
| DSA | Digital Signature Algorithm |
| DSM | Deep SubMicron |
| DSM | Deep-SubMicron |
| ECDSA | Elliptic Curve DSA |
| EFF | Electronic Frontier Foundation |
| EHC | Entropy Harvesting Circuit |
| ES | Entropy Source |
| FERNS | Fingerprint Extraction and Random Numbers in SRAM |
| FIPS | Federal Information Processing Standard |
| FPGA | Field-Programmable Gate Array |
| GCD | Greatest Common Divisor |
| GSP | Gaussian Stochastic Process |
| HCI | Hot Carrier Injection |
| IC | Integrated Circuit |
| IDFT | Inverse Discrete Fourier Transform |
| Iff | If and only if |
| IID | Independent Identically(Uniformly) Distributed |
| LFSR | Linear Feedback Shift Register |
| MGF | Moment Generating Function |
| MOSFET | Metal-Oxide-Semiconductor Field-Effect Transistor |
| NBTI | Negative Bias Temperature Instability |
| NBTI | Negative Bias Temperature Instability |
| NCSU | North Caroline State University |
| NKS | New Kind of Science |

| | |
|---|---|
| NSA | National Security Agency (US) |
| OHT unit | Online Health Test unit |
| OSTE queues | Online Self-Tested Entropy queues |
| PDF | Probability Distribution Function |
| PDK | Process Design Kit |
| PDS | Power Density Spectrum |
| PGP alg | Pretty Good Privacy algorithm |
| PhRNG | Physical Random Number Generator |
| PRNG | Pseudo Random Number Generator |
| PVT | Process-Voltage-Temperature |
| QoS | Quality of Service |
| RSA | Rivest Shamir Adleman |
| RV | Random Variable |
| SHT | Statistical Hypothesis Testing |
| SNM | Static Noise Margin |
| SNR | Signal to Noise Ratio |
| SoC | System on a Chip |
| SP | Stochastic Process |
| SRAM | Static Random Access Memory |
| ST | Statistical Testing |
| TRNG | True Random Number Generator |
| UMC | United Microelectronics Corporation |
| VLSI | Very-Large-Scale Integration |
| VTCs | Voltage Transfer Curves |
| WGN | White Gaussian Noise |
| wrt | with respect to |