POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Design of a true random number generator for post-quantum cryptography

Supervisors

Prof. Guido MASERA

Prof. Maurizio MARTINA

Alessandra DOLMETA

Mattia MIRIGALDI

Candidate

Valeria PISCOPO

April 2024

Abstract

The protection of sensitive data is a paramount concern in a wide variety of fields, making the use of cryptography crucial to ensure confidentiality and information security. This is all the more needed after the advent of quantum computers. Their much higher computational strength with respect to the classical ones allows them to break many of the regularly used public-key protocols.
Post-quantum cryptography (PQC) researches have been investigating robust algorithms that even quantum computer attacks cannot undermine.
The starting point of any cryptographic algorithm is an encryption key generation. A weak random key would allow the attacker to easily decrypt data, exposing the entire cryptosystem to high vulnerability, consequently devaluing the complexity of the PQC algorythm.
While pseudo random number generators (PRNGs) are based on deterministic algorithms, hence producing keys that can be predicted once known the algorithm and the initial state, true random number generators (TRNGs) exploit the inherent randomness of physical phenomena (thermal noise, power supply fluctuations, temperature variations etc.) to generate true random samples that fulfill the requirements needed by a robust key. A solid random key should be highly unpredictable (non-deterministic), aperiodic and characterised by good statistical properties. The National Institute of Standards and Technology (NIST) published a set of recommendations and tests to design and validate a reliable Entropy Source (ES) to be used in cryptographic Random Bit Generators (RGBs).
This work intends to provide a possible hardware implementation of a ring oscillator-based TRNG, aiming to obtain the best trade-off in terms of area, throughput, power and entropy. The original entropy source is also connected to an accelerator implementing an optimized version of the Keccak security primitive, to have the possibility of generating a random key with or without additional conditioning. The whole system has been integrated as an external accelerator in the RISCV-based X-HEEP microcontroller.
The proposed solution passes all the tests of the NIST statistical test suite and shows promising results in terms of entropy, area and throughput, representing an interesting starting point for an integrated RBG for cryptographic algorithms.

Keywords: Post-Quantum Cryptography, PQC Key Generation, True Random Number Generators, Entropy, TRNG Hardware Design, TRNG integration

Table of Contents

List of Tables

List of Figures

Acronyms

CRP

Challenge Response Pair

DCM

Digital Clock Manager

ES

Entropy Source

FiRO

Fibonacci Ring Oscillator

GaRO

Galois Ring Oscillator

LFSR

Linear Feedback Shift Register

Meta-RO

Metastable Ring Oscillator

NIST

National Institute of Standards and Technology

PLL

Phase Locked Loop

PRNG

Pseudo Random Number Generators

PQC	Post-Quantum Cryptography
PUF	Physical Unclonable Functions
RGB	Random Bit Generator
RO	Ring Oscillator
TERO	Transition Effect Ring Oscillator
TRNG	True Random Number Generator

Chapter 1

Introduction

Cryptography can be traced back thousands of years, when, mainly for military reasons, ancient populations developed ciphers and encoding techniques to secretly communicate. As time progressed, these techniques evolved deeply, but the core idea is still the same: a secure exchange of information among the authorized recipients so that the message will be unintelligible to all the external parties, i.e. the adversaries.
With the spread of digital communication, the compelling need to protect sensitive data led to the development of many cryptographic protocols, based on cryptographic primitives. Primitives are low-level cryptographic algorithms, used to compose higher-level algorithms. [1]
Classical cryptography algorithms are based on hard mathematical problems such as the factorization of a large number (Rivest-Shamir-Adleman algorithm, RSA) or the discretization of a logarithm (Elliptic-curve cryptography, ECC). For many years, the robustness of these algorithms was enough to ensure data integrity, given the huge computational cost required to solve them. However, this was no longer true after the advent of quantum computers.
Quantum computers rely on quantum bits instead of the classical binary bits. The state of a quantum bit (qubit) is a superposition of both 0 and 1 at the same time, meaning that, with a series of qubits, more numbers can be simultaneously represented. This results in great computational strength and processing speed, making the classical cryptographic algorithms vulnerable. Even though a great diffusion of quantum computers has not yet occurred, this is still a big problem to manage. For this reason, classical cryptography evolved in **post-quantum cryptography** (PQC), whose focus is to find robust algorithms that even quantum computers cannot crack. Examples might be lattice-based algorithms, where the problem is finding a non-zero vector in a lattice, or multivariate polynomial algorithms, based on multivariate quadratic equations.

Regardless of the type of cryptographic algorithm, the generation of at least one **secret key** is an essential step; keeping this key unintelligible to adversaries is crucial to ensure security. Ideally, an attacker that knows everything about a cryptographic system but the key is still not able to undermine the whole system. In other words, the generation of a robust key is the foundation of a robust algorithm. Conversely, a weak key would jeopardize the complexity of any algorithm: without an effective key generation process, all the work invested in developing a solid PQC algorithm would be completely vain.

1.1 Key generation in cryptographic systems

A robust cryptographic key needs to have specific statistical features, unpredictability and aperiodicity being just two of them. Consequently, not all methods to generate random numbers are suitable options, as they may lack some of these properties.
There are three possibilities:

- **Physical unclonable functions** (PUFs): based on the physical properties of the specific hardware component. They offer the maximum security level.
- **Pseudo random number generators** (PRNGs): based on deterministic algorithms, hence easily predictable. They offer the lowest security level.
- **True random number generators** (TRNGs): based on the inherent randomness of physical processes. They represent a compromise between PUFs and PRNGs, producing robust keys. This thesis will focus on this type of random number generator.

A brief description of the first two classes of generators will now follow. A deeper examination of TRNGs will be then proposed in Chapter 2.

1.1.1 Physical unclonable functions (PUFs)

PUFs harness the manufacturing variability of each IC, which results in unique physical characteristics (gate delays, power-on state of SRAM cells, threshold voltages), from which the secret can be derived. [2]
Classical cryptography authentication relies on a secret key stored in non-volatile memory. If the key applied by the user (*challenge*) is equal to the stored one, then the authentication is successful (Figure 1.1, left). The main problem is that the on-chip memory leaks current. Consequently, an attacker could register the power consumption at each user request (for example, by using a small resistor in series to the power supply), extract the leakage power, and trace the secret key. This

issue can be prevented by using a PUF-based authentication scheme: each PUF is unique, hence the same challenge will produce different responses depending on the chip.
The authentication is composed of 3 steps (Figure 1.1, right):

1. Creation of a **challenge-response pair** (CRP) by user and storage in a trusted environment. The CRP will depend on circuit variability.

2. Authentication request by applying a challenge, even in an untrusted environment. Authentication is successful if there is a match between the generated response and the stored response.

3. CRP cleared from the database.

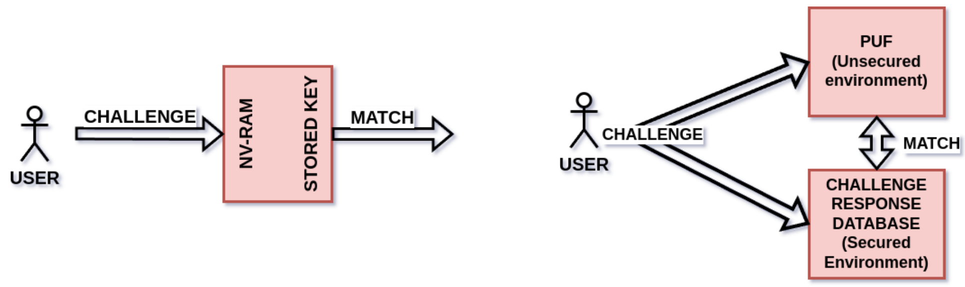

USER
CHALLENGE
NV-RAM
STORED KEY
MATCH
USER
CHALLENGE
PUF (Unsecured environment)
MATCH
CHALLENGE RESPONSE DATABASE (Secured Environment)

Figure 1.1: Classical (left) vs PUF-based (right) Authentication Schemes

Depending on their features, PUFs can be classified as:

- **Weak PUFs**: they provide a small number of CRP, the responses need additional error-correcting circuits to be used as cryptographic keys, and, in general, they are more vulnerable. Examples of weak PUFs are SRAM-based PUFs.

- **Strong PUFs**: they provide a large number of CRP, the responses can be directly used as cryptographic keys and they provide a higher level of security. Examples of strong PUFs are arbiter-PUFs or ring oscillator-PUFs.

Since the key principles underlying the various PUF implementations are quite similar to the ones of the TRNGs, refer to chapter 2 for any additional explanation. [3]

1.1.2 Pseudo random number generators (PRNGs)

PRNGs are based on algorithms that, starting from an initial value (***seed***), produce an apparently random sequence of numbers. As a matter of fact, the output is a

deterministic function of the seed, hence predictable on some level. Therefore, it is crucial that the seed is unknown to the attacker. This is why, usually, a PRNG is coupled with a TRNG: the latter extracts a seed from physical processes and feeds it to the former. This ensures a seed with sufficient entropy and, consequently, the output of the PRNG can be used as a cryptographic key.
There are different types of possible PRNGs implementations: Lagged Fibonacci generators, Blum Blum Shub (BBS), Mersenne Twister being just some of them. The most commonly used PRNG is based on **linear feedback shift registers** (**LFSRs**), i.e. on right-shifts and XOR operations (Figure 1.2). [4], [5]

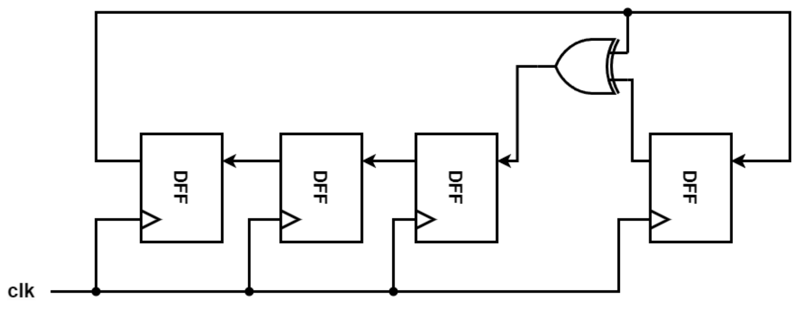


Figure 1.2: Example of a simple 4-bit LFSR

An LFSR is represented by a polynomial (**feedback polynomial**), which defines the mask or tap sequence, i.e. the bit positions that will affect the next state. For example, the polynomial $x^4 + x + 1$ corresponds to a mask equal to 10011. Since the right-most bit is always equal to 1, the final binary mask is truncated and, in this example, is then equal to 1001.
When considering PRNGs, the polynomial is chosen among the so-called **primitive polynomials**, which ensure the maximum length period of shifting. The LFSR derived from a primitive polynomial of degree n will have $2^n - 1$ different states, so it will have $2^n - 1$ different outputs and, after this period, the sequence will be repeated.
The position of the XOR-ed bits depends on the type of LFSR, an example can be the Fibonacci LFSR, where taps are XOR-ed sequentially.
In summary, the LFSR is initialized by the seed, at each clock cycle a shift is performed and the system is brought in one of the $2^n - 1$ possible states.

1.2 Thesis structure

This thesis proposes a possible TRNG implementation and integration as an external accelerator on a RISC-V based microcontroller. The main idea is to find a tradeoff between the classical PPA analysis (power, performance and area) and the statistical requirements of a robust cryptographic key.
After this initial introductory chapter 1, the thesis is structured as follows:

- **chapter 2** introduces the true random number generators and their general characteristics. Each section of the chapter focuses on one of the most commonly used digital implementations: section 2.1 describes the ring oscillator -based configurations, section 2.2 the PLL-based ones, section 2.3 includes a brief discussion on digital clock manager-based and chaotic map-based implementations.

- **chapter 3** presents the concepts of entropy and cryptographic key robustness. Following the NIST recommendations, the most important features of an efficient entropy source are described. Section 3.1 focuses on the different components of an entropy source, whereas section 3.2 provides an overview on the NIST tests to analyse the robustness of a given implementation.

- **chapter 4** focuses on the hardware implementation of the TRNG. Firstly, in section 4.1, the TRNG architecture is presented, each subsection centered on one of the main components. In subsection 4.1.1, the noise source implementation description is also followed by a presentation of the design and simulation methodologies. Secondly, section 4.2 presents the additional conditioning component, i.e. the Keccak accelerator.

- **chapter 5** concerns the integration of the TRNG component as an external accelerator of the X-HEEP microcontroller. After a brief presentation of this platform (section 5.1), sections 5.2 and 5.3 describe the actual integration process of both the TRNG block alone and the TRNG block with Keccak conditioning.

- **chapter 6** reports the final results in terms of entropy, statistical properties and area, power and performance, both considering ASIC and FPGA implementations. A comparison with the state-of-art results is also included, along with a summary of the potential improvements to further develop the proposed TRNG.

- **chapter 7** presents the conclusion of this thesis.

Chapter 2

True random number generators

As previously mentioned, true random number generators exploit the inherent randomness of physical sources to produce unpredictable outputs. Generally, a combination of these sources is used to enhance randomness.
A TRNG can be:

- **analog**; sensors or voltage comparators are required for the digitization of the output signals. Randomness is usually extracted by exploiting:
  - **thermal noise**: resistors can be used for this purpose. Due to its small amplitude, high-gain amplifiers are needed.
  - **chaotic circuits**: they are a particular class of oscillatory circuits that display a non-periodic behavior. An example is the Lorenz chaotic system.
- **digital**; in this case the most commonly used mechanisms to generate entropy are:
  - **jitter**: the presence of inherent semiconductor noise, temperature variations, power supply fluctuations, and cross-talk produces jitter phenomena in clock signals. This means that the rising and falling edges of the signal suffer from lags or advances and thus can be considered as random variables.
  - **metastability**: in this case the randomness is obtained by forcing the violation of setup and hold time of memory elements, consequently leading to outputs in an unknown state. It is often used in combination with jitter.

– **chaotic maps**: they are peculiar mathematical functions (maps) displaying a chaotic behavior, meaning that their output can largely vary by slightly changing their initial state. This can be done by looping the output back into the map.

The focus of the analysis will now be on fully digital noise sources, with classification based on the most commonly used types of TRNG implementations.

2.1 Ring Oscillator-based TRNG

A ring oscillator is a circuit composed of an odd number of inverters, where the output of the last inverter of the chain is connected to the input of the first one, as shown in Figure 2.1.

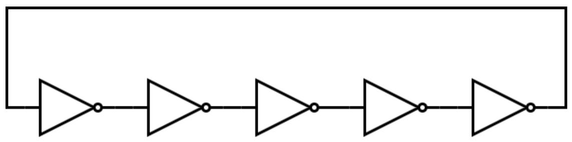


Figure 2.1: Ring Oscillator with N = 5 inverters

Ideally, the output bit of the chain would oscillate with a period equal to

$$T_{RO} = 2N \cdot T_{delay} \tag{2.1}$$

where N is the number of inverters in the chain and T_{delay} is the delay of a single inverter. In the ideal case, each inverter introduces the same delay. In the real case, the output signal of the ring oscillator will suffer a certain lag or advance (**jitter**) due to physical phenomena such as power supply fluctuations, semiconductor noise, temperature variations, and so on. These processes will affect each gate and each internal signal of the RO in a different way, meaning that the real oscillation period will be unpredictable. As a matter of fact, any internal signal of the RO could be taken as the output of the circuit, i.e. as a random bit.
This phenomenon has been widely analyzed in literature ([6], [7], [8]).
The delay of a single gate can be written as:

$$d_i = D_i + \Delta d_i = D_i + \Delta d_{Li} + \Delta d_{Gi} \tag{2.2}$$

where:

- D_i is the nominal gate delay at the nominal supply voltage level and temperature;
- Δd_i is the delay variation causing jitter; it in turn is constituted of:

- Δd_{Li}: the component due to local physical events. It can be expressed as:

$$\Delta d_{Li} = \Delta d_{LGaussi} + \Delta d_{LDeti} \tag{2.3}$$

  with $\Delta d_{LGaussi}$ being the **local Gaussian jitter component** (mean value $\mu_i = 0$ and standard deviation σ_i) and Δd_{LDeti} the local deterministic jitter due to local cross-talks.

- Δd_{Gi}: the component due to global conditions, such as power supply and temperature of the whole system.

$$\Delta d_{Gi} = K_i(\Delta D + \Delta d_{GGauss} + \Delta d_{GDet}) \tag{2.4}$$

  K_i is a coefficient indicating how much the global conditions affect the single delay; ΔD represents the slow global variations; Δd_{GGauss} and Δd_{GDet} are the gaussian and deterministic jitter due to global sources.

The major source of randomness is represented by $\Delta d_{LGaussi}$, which can be consequently considered as the most important contribution of the model. The **global deterministic jitter** source Δd_{GDet} is also generally taken into account, as it can be easily corrupted from the external, hence representing the principal source of injection attacks ([9]): if frequencies are injected into the power supply, the ROs composing the TRNG can experience phase-locking, meaning that they can all be synchronized, resulting in a completely deterministic output.
It was previously stated that the effect of the variable delay contributions of the single inverters is an unpredictable period of the clock generated by an RO. To obtain a random bit, it is sufficient to sample this signal with another clock, that may or not may be generated by another RO (Figure 2.2).

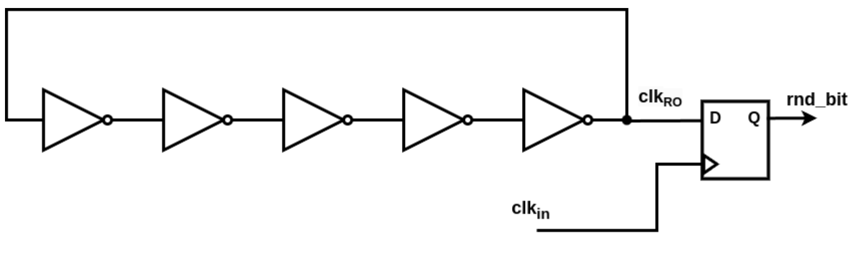


Figure 2.2: RO-based TRNG key principle

Many implementations working with this principle are present in literature. ([10], [11], [12]). In the following subsections, different variations of RO-based TRNGs are briefly presented.

2.1.1 Transition Effect Ring Oscillator (TERO)

The Transition Effect Ring Oscillator [13] is based on the so-called **oscillatory metastability**. This phenomenon can be clarified by analyzing the implementation in Figure 2.3. Table 2.1 displays a partial truth table of the circuit. Whenever `rst = 0` a feedback loop is formed and the transitions of the `ctrl` signal cause oscillations (oscillatory metastability). The even number of inverting elements in the loop makes sure that this metastable state will eventually become stable; however, due to the inherent circuit noise, the number of oscillations needed for this process is unpredictable. The final T flip-flops are used to determine whether the number of oscillations is even or odd. In the first case the output is equal to `1`, in the second case it is equal to `0`.

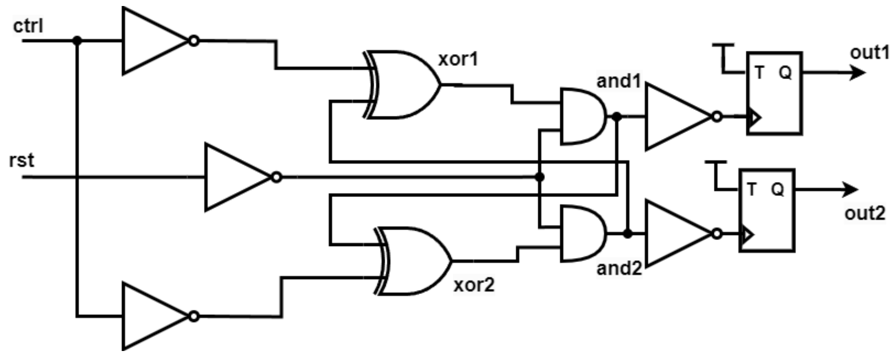


Figure 2.3: Transition Effect Ring Oscillator. XOR gates can also be replaced by NANDs or NORs

`rst`	`ctrl`	`and1`	`and2`	`xor1`	`xor2`	STATE
1	0	0	0	1	1	stable
1	1	0	0	0	0	stable
0	$0 \to 1$	`xor1`	`xor2`	`xor2`	`xor1`	**metastable**
0	$1 \to 0$	`xor1`	`xor2`	`NOT(xor2)`	`NOT(xor1)`	**metastable**

Table 2.1: Partial truth table of TERO in Figure 2.3

2.1.2 Metastable Ring Oscillator (Meta-RO)

Metastable ROs [14] harness the metastable state of inverters, which occurs whenever the output of the inverter is short-circuited to its input. Referring to Figure 2.4, this corresponds to a low clock signal. When `clk` is high, the circuit is a RO and the final D flip-flop can sample a bit. The additional low amplitude noise of the isolated metastable inverters is now amplified by the RO, which means that the system is in an unknown state and the sampled bit will be random. The additional delay line is inserted to tune the sampling instant in a precise way.

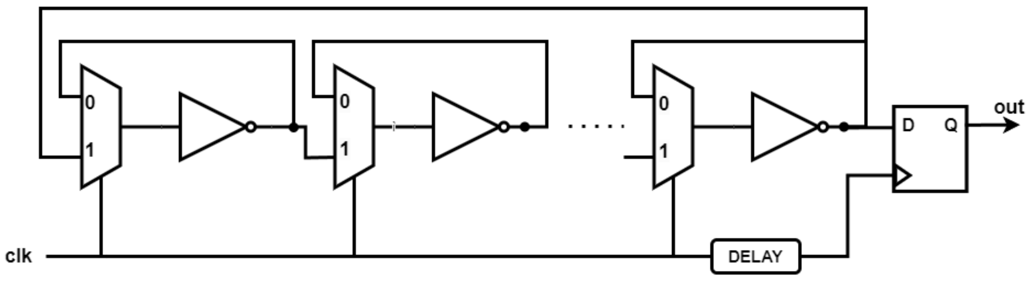


Figure 2.4: Metastable Ring Oscillator

2.1.3 Fibonacci (FiRO) and Galois Ring Oscillators (GaRO)

Fibonacci (FiRO) and Galois (GaRO) ring oscillators ([15]) are directly obtained from Fibonacci and Galois LFSR configurations, where D flip-flops are replaced by inverters (Figures 2.5 and 2.6). Due to noise causing variations in the inverters' delay, the circuit will no longer evolve in a deterministic fashion.
In both cases, the switches are closed if the respective coefficient of the polynomial describing the circuit is equal to 1 (as explained in Subsection 1.1.2). The characteristic polynomials take the form:

$$P(x) = \sum_{i=0}^{n} f_i x^i \qquad with \quad f_0 = f_n = 1 \tag{2.5}$$

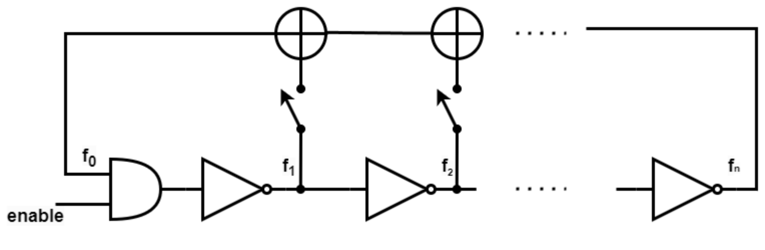


Figure 2.5: Fibonacci Ring Oscillator

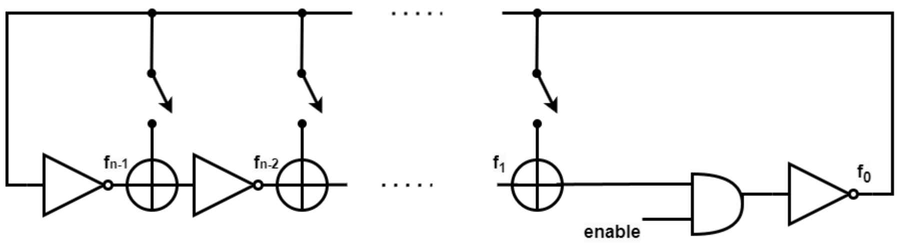


Figure 2.6: Galois Ring Oscillator

FiRO and GaRO can be combined together ([15], [16]) in a new structure (**FiGaRO**), shown in Figure 2.7. N parallel FiROs and N parallel GaROs are sampled at the output and the 2N resulting random bits are XORed to increase the randomness. The higher the number of parallel ROs, the lower the probability of phase-locking. Moreover, entropy can be further enhanced by introducing metastability through the violation of the setup/hold times of the sampling flip-flops.
To have a `n`-bit output, `n` parallel FiGaRO stages can be used.

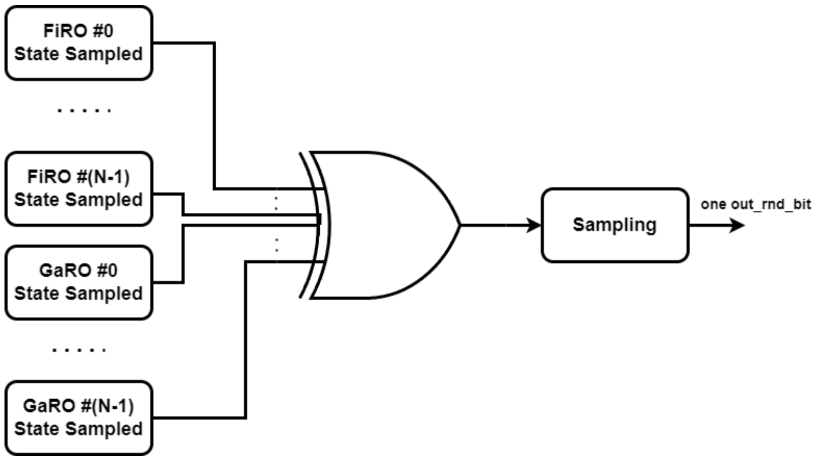


Figure 2.7: Fibonacci-Galois Ring Oscillator stage

2.2 PLL-based TRNG

A Phase Locked Loop (PLL) is a circuit able to generate an output signal with frequency and phase related to the ones of the input signal. A simplified schematic of a digital PLL is depicted in Figure 2.8.

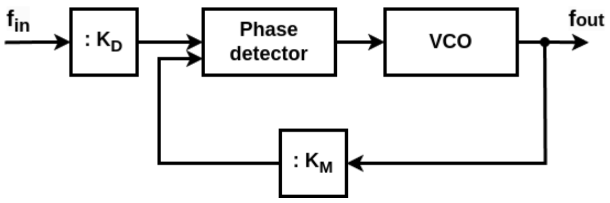


Figure 2.8: Phase Locked Loop (PLL) simplified schematic

Considering the notation of the figure, the formula describing the output frequency of the system is:

$$f_{out} = \frac{K_M}{K_D} f_{in} \tag{2.6}$$

with K_M and K_D respectively the **multiplication** and **division factors** of the PLL.
In the context of PLL applications for TRNGs ([17], [18]), the objective is to generate a clock signal with a higher frequency than the input signal, i.e. the original clock. In fact, as in some of the previous RO-based cases, also in this case the source of randomness is mainly the inherent **jitter** present at the output of the PLL: the key idea is to sample the new generated signal. An example is proposed by [18]; the architecture is shown in Figure 2.9. In this case, just one PLL is used, but a second one can be inserted to obtain a different sampling clock than the initial one.
Ideally, no jitter is present and the output of the flip-flop (the sampler) is periodic with period $T_P = K_D T_0$. Due to jitter, the sampling of `clk1` will contain a certain amount of entropy and the output of the flip-flop can be considered a random variable. The decimator component has the simple function of collecting several samples and reducing them to one output random bit, usually a simple XOR is used.
The accurate choice of K_M and K_D is fundamental to obtain a TRNG with good statistical properties. These two design parameters determine the delay ΔT between the two edges of the clock. The more this value is comparable to the entity of the jitter, the more the entropy of the system is enhanced.

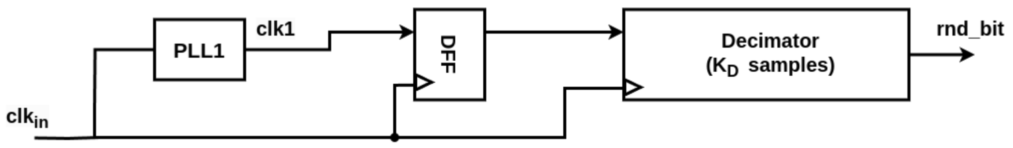


Figure 2.9: PLL-based TRNG architecture

One of the main problems of the PLL-based TRNG architecture is the tradeoff between the **bit rate R** and the **sensitivity to jitter S**:

$$R = \frac{f_{in}}{K_D} \qquad S = \frac{1}{\Delta} = f_1 \cdot K_D \tag{2.7}$$

Both equations depend on K_D, thus the choice of this parameter must be done carefully: the sensitivity to the jitter must be high enough to ensure good statistical properties, without compromising the bit rate. In [19] an accurate mathematical analysis of the problem is proposed.

2.3 Other implementations

While the PLL and RO-based TRNG architectures are the most frequently used in the current state of art, there are also other possibilities. A brief discussion on two additional techniques will be now presented.

2.3.1 DCM-based TRNG

In the case of a purely FPGA implementation of the TRNG, **Digital Clock Managers** (DCMs, also called Clock Managers in some FPGAs) can be used in the same fashion as ROs and PLLs to exploit jitter or metastability phenomena.
DCMs are hardware primitives of some FPGAs, able to produce one or more clock signals with a programmable frequency. They can be considered as more flexible PLLs.
In the case of a TRNG implementation, two approaches are possible:

- **frequency tuning** ([20]). The frequencies of the output clocks from the Dynamic Clock Manager (DCM) can be dynamically adjusted on-the-fly using its Dynamic Reconfiguration Port (DRP). This adjustment can be made without affecting other functionalities of the FPGA. The jitter, representing the source of randomness, is contingent on the DCM design parameters, i.e. the multiplication factor (M) and division factor (D), similar to the case of PLLs. By modifying these parameters at runtime, the level of entropy generated by the TRNG can be varied. This dynamic variation is facilitated through a tuning circuit that stores specific pairs of M and D values, which are determined through mathematical analysis.

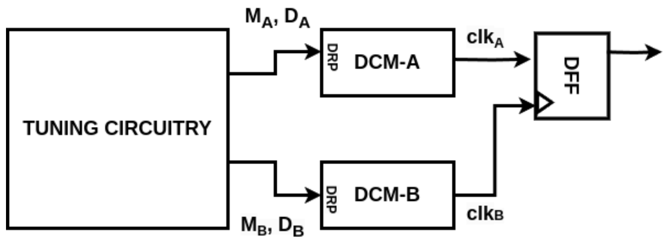


Figure 2.10: DCM-based TRNG - runtime frequency modulation

- **dynamic phase shifting (DPS)** ([21]). The dynamic variations of the DCM parameters can lead to slow TRNGs, as a lot of clock cycles may be needed for two consecutive samplings. This problem can be overcome by varying the phases rather than the frequencies of the output clock signals, forcing the rising or falling edges to occur in the setup or hold times of the sampling flip-flop. Differently from the previous case, metastability is the source of randomness, rather than jitter. The phase shift resolution would normally depend on the voltage-controlled oscillator internal to the DCM, but can be increased by adding tunable elements, like a carry chain primitive.
  Post-processing circuitry is required to achieve a satisfying level of entropy.

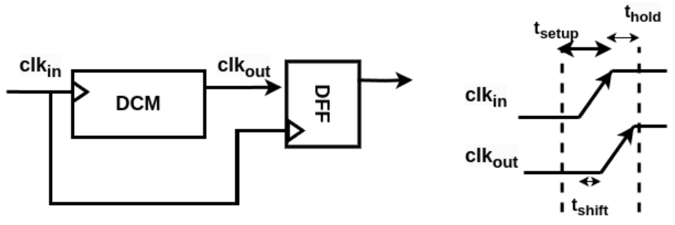


Figure 2.11: DCM-based TRNG - DPS technique

2.3.2 Chaotic map-based TRNG

Chaotic maps are particular mathematical functions that exhibit, apparently, unpredictable behavior. However, this is actually true only for a long but finite period of time. The behavior of a chaotic map is actually dictated by its initial conditions, making these types of functions more suitable for PRNGs applications or TRNG additional conditioning circuits ([22]). Nevertheless, an investigation on direct TRNG applications has also been carried out in literature ([23], [24]).

Different types of maps can be used, with different degrees of complexity and period lengths; examples may be the **tent map** or the **Bernoulli map**. In any case, the analyses conducted on these implementations reveal that, even though the resulting bit stream is unbiased and exhibits a uniform distribution in terms of 0s and 1s, the resulting entropy is relatively low compared to other methods. This implies that chaotic maps are indeed a much smarter choice for conditioning components or PRNGs.

Chapter 3

NIST recommendations and tests

In the first chapter, it was emphasized the paramount importance of a solid key to ensure the efficacy of a cryptographic algorithm. Consequently, concepts like randomness, entropy and key robustness are important just as much as the PPA parameters when talking about the design of a TRNG. For this purpose, NIST provides a series of recommendations ([25]) and statistical tests ([26]) for a thorough design and evaluation of the entropy source. NIST also published an additional document in 2022, [27], which is more focused on the interfaces of a possible RBG implementation and its access through firmware.

Entropy is the core mathematical concept that describes the randomness of a system. It can be defined as a measure of uncertainty and unpredictability. In other words, entropy is an assessment of how much a realization of an experiment can be predicted by observing previous realizations of the same experiment. In cryptography, **min-entropy** and **Shannon entropy** are the most used definitions:

$$H_{min} = -\log_2(\max_{1 \leq i \leq k} p_i) \qquad H_S = -\sum_{i=1}^{k} p_i \log_2 p_i \tag{3.1}$$

with $p_i = Pr(X = x_i)$ for $i = 1, ..., k$, where X is an independent discrete random variable, all its possible values come from the set $A = \{x_1, x_2, ...x_k\}$. The maximum possible min-entropy of a random variable with k distinct values is equal to $log_2 k$.
In TRNG applications, the aim is to obtain an output key characterized by maximized entropy so that, ideally, it is impossible to predict. To achieve this objective, NIST proposes an entropy source model and effective ways to test it. In the following sections, an overview of both these aspects will be presented.

3.1 Entropy source

The recommended entropy source model is depicted in Figure 3.1.

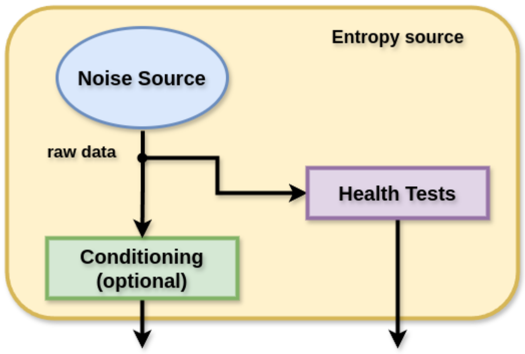


Figure 3.1: Entropy source model ([25])

The different possibilities for an effective noise source have been described in the previous chapter. The focus will be now on the other two components: the health tests and the conditioning.

3.1.1 Health tests

The noise source relies on physical aspects of the circuit, hence necessitating the accurate monitoring of potential alterations in its output, particularly during rapid shifts in operating conditions, such as sudden power supply fluctuations. The health tests serve this purpose: they check on the run-time entropy source operations, signaling any errors in the raw output bitstream of the noise source.
There are three types of health tests:

- **start-up health tests**: they check on the bitstream right after the power-on or the reboot of the device. During these phases, the output of the noise source should not be considered valid.

- **continuous health tests**: they are performed during the normal functioning of the component to signal possible errors and anomalies in the bitstream. Differently from the previous case, the tests are now conducted without invalidating the output key.

- **on-demand health tests**: they are additional tests that may be requested in specific moments, also depending on the chosen type of noise source. When these tests are performed, the bitstream should be discarded and not used as output.

Besides a series of recommendations on the characterization of customized health tests for an RBG, NIST also describes two approved continuous health tests, which are sufficient to evaluate the run-time conditions of the output bitstream of the noise source: the **Repetition Count Test** and the **Adaptive Proportion Test**. Obviously, being statistical tests, they also have a **false positive probability** α, i.e. a probability that a properly functioning noise source will fail the test on a specific output. In many applications, this value is equal to 2^{-20}.

Repetition Count Test

The Repetition Count Test determines whether the output of the noise source is stuck, continuously producing only `0`s or `1`s.
The probability of a noise source of minimum entropy H_{min} to produce n consecutive equal samples is at most $2^{-H_{min}(n-1)}$. In order to have a correct output, without critical repetitions, the false probability α must be greater than or equal to this value. The minimum value of n that satisfies this condition, the **cutoff value** C, is defined as follows:

$$C = 1 + \lceil \frac{-log_2 \alpha}{H_{min}} \rceil \tag{3.2}$$

Algorithm 1 contains the pseudo-code describing how the test works.

Algorithm 1 Repetition Count Test

```
$A = next\_sample$
$B = 1$
while continous_samples do
    $X = next\_sample$
    if $X = A$ then
        $B = B + 1$
        if $B \geq C$ then error
        end if
    else
        $A = X$
        $B = 1$
    end if
end while
```

Adaptive Proportion Test

While the Repetition Count Test only provides a coarse method to detect evident entropy losses in the output bitstream, the Adaptive Proportion Test can track the frequency of a certain value in a **window of samples** of size W. In other words, if a value occurs more frequently than others, the test returns an error. The upper limit for the occurrence of a specific value within W is indicated as C (**cutoff value**), just like in the previous test.
NIST suggests a window size equal to 1024 bits if the noise source is binary, otherwise it should be equal to 512 samples. For what concerns the cutoff value, it should be chosen so that the probability of having C or more occurrences of the same value in W is at most equal to the false positive probability α. Mathematically:

$$Pr(B \geq C) \leq \alpha \tag{3.3}$$

In [25], NIST also presents a set of possible values of C associated to different entropy values.
The Adaptive Proportion Test for binary sources is detailed in Algorithm 2, in the case of non-binary sources, the error is only given by the condition $B \geq C$.

Algorithm 2 Adaptive Proportion Test (binary source case)

```
while continous_samples do
    $A = next\_sample$
    $B = 1$
    for $i = 1$ to $W - 1$ do
        if $A = next\_sample$ then
            $B = B + 1$
        end if
        if $B \geq C$ or $W - B \geq C$ then error
        end if
    end for
end while
```

3.1.2 Conditioning

The entropy source designer might additionally integrate a conditioning component, generally a cryptographic function. Its purpose is to reduce bias and/or increase entropy of the raw output of the noise source; it may also be used to increase the throughput of the system.
Among all the possible conditioning functions (block cipher-based or hash-based), a particular focus will be now placed on the **Keccak** cryptographic function, as it

will be used as a conditioning component in the proposed TRNG implementation (section 4.2). Keccak also serves as the core hash function for SHA-3 (Secure Hash Algorithm), the latest member of the SHA family published by NIST.

Keccak

Keccak is a family of cryptographic hash functions, designed by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche ([28]).
The basic block of the Keccak algorithm is the **sponge construction** ([29]), which consists in an iterative process of permutations (f in Figure 3.2) on a fixed number of bits. The width of the permutation is indicated as b. With the notation **Keccak[r,c]**, the specific Keccak function of **bit-rate r** and **capacity c** is recalled; $r+c$ is fixed and equal to b. In the case of the four SHA-3 hash functions (SHA3-224, SHA3-256, SHA3-384, SHA3-512), this value is equal to b = 1600.
As depicted in Figure 3.2, the sponge construction is based on two phases:

- **absorbing phase**: the r-bit blocks, in which the input message has been divided, are XOR-ed with the first r bits of the state, interleaved with the application of the permutation function f. This phase is completed when all the message blocks are processed.

- **squeezing phase**: the output string is composed of blocks of r bits, once again with the interleaving of the permutation function. The length of the output, i.e. the number of $r - bit$ blocks to be concatenated, is chosen by the user.

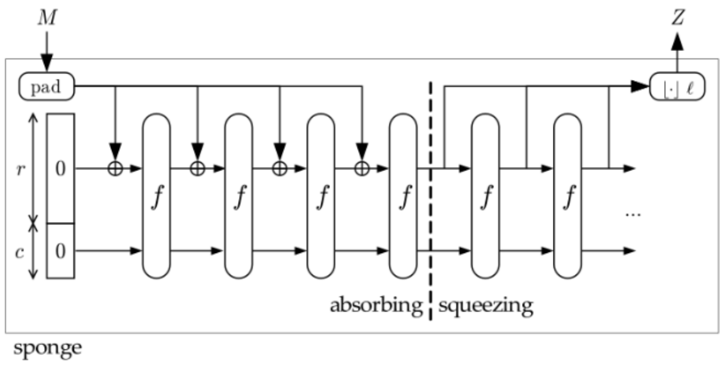


Figure 3.2: Sponge Function

Keccak sponge function is described in Algorithm 3.

Algorithm 3 Pseudo-code description of the Keccak sponge function

```
procedure KECCAK[R,C](M)
    ▷ Padding
    d = 2^|Mbits| + sum for i=0..|Mbits|-1 of 2^i Mbits[i]
    P = Mbytes || d || 0x00 || ... || 0x00
    P = P ⊕ ( 0x00 || ... || 0x00 || 0x80)
    ▷ Initialization
    S[x,y] = 0                                          ▷ ∀ (x,y) in (0...4, 0...4)
    ▷ Absorbing phase
    for each block P_i in P do
        S[x,y] = S[x,y] xor P_i[x+5*y]                  ▷ ∀ (x,y) such that x+5*y < r/w
        S = Keccak-f[r+c](S)
    end for
    ▷ Squeezing phase
    Z = empty string
    while output is requested
    Z = Z || S[x,y]                                     ▷ ∀ (x,y) such that x+5*y < r/w
    S = Keccak-f[r+c](S)
    return Z
end procedure
```

The description of the permutation function, $Keccak - f$, is reported in Algorithm 4. Every 1600-bit state, composed of a 5x5 matrix of 64-bit words, is subjected to 24 compression rounds, each one of them divided in five steps (θ, ρ, π, χ, ι). At the end of each step, the state array A is updated. Algorithm 5 describes the compression procedure, for more details refer to [28].

Algorithm 4 Keccak-f[1600] (A)

```
procedure KECCAK-F[1600](A)
    ▷ A is the state matrix
    for i in 0 ... $n_r - 1$ do
        A=Round[1600] $(A, RC\,[i])$
    end for
    return A
end procedure
```

Algorithm 5 Round[b]

```
procedure ROUND[B]((A,RC))
    ▷ $\theta$ step
    C[x]= A[x,0] ⊕ A[x,1] ⊕ A[x,2] ⊕ A[x,3] ⊕ A[x,4] ⊕,            ▷ ∀x in 0...4
    D[x]= C[x-1] ⊕ ROT(C[X+1],1),                                  ▷ ∀x in 0...4
    A[x,y]=A[x,y]⊕ D[x]                                 ▷ ∀ (x,y) in (0...4, 0...4)
    ▷ $\rho$ and $\pi$ steps
    B[y,2x+3y]=ROT(A[x,y], r[x,y])                      ▷ ∀ (x,y) in (0...4, 0...4)
    ▷ $\chi$ step
    A[x,y]=B[x,y] ⊕ ((NOT B[x+1,y]) AND B[x+2,y]) ▷ ∀ (x,y) in (0...4, 0...4)
    ▷ $\iota$ step
    A[0,0]=A[0,0] ⊕ RC
    return A
end procedure
```

3.2 NIST tests

NIST provides two series of statistical tests to evaluate whether a random number generator is suitable for cryptographic applications or not. Both test suites analyze a stream of output samples, directly from the noise source or after the conditioning. The difference is that the first test suite ([26]) examines specific characteristics of data to evaluate if it can be considered random or not, and the second one (NIST 800-90B, [25]) assesses the entropy value of the bitstream.

3.2.1 Statistical Test Suite

The NIST statistical test suite is composed of 15 tests [1] and its final objective is to estimate whether the analyzed data is random or not.
As already mentioned in subsection 3.1.1, the false positive probability (α, also called ***level of significance*** of a test) must be taken into account in every statistical test. Generally, α is chosen in the range [0.001, 0.01] and, particularly for cryptographic applications, $\alpha = 0.01$. This means that, out of 100 sequences generated by a valid RGB source, one is expected to be rejected.
Another important parameter is the ***P-value***, produced by each one of the 15 tests. It represents the probability that an ideal random number generator would have produced a less random sequence than the one analyzed by the test. In other words, if the *P-value* is equal to 1, then the sequence is perfectly random. Since each one of the 15 tests examines a particular statistical characteristic of the sequence, it is important to notice that the term *randomness* can have various nuances.
Once chosen the level of significance α and having executed a test, two cases are possible:

- *P-value* $\geq \alpha$: the sequence is determined to be random, **test is passed**.
- *P-value* $< \alpha$: the sequence is determined to be non-random, **test is failed**.

To have reliable results, the number of samples n to be analyzed must be in the range from 10^3 to 10^7. Moreover, the number of sequences m, composing n, must be related to the chosen level of significance *alpha*: if for example $\alpha = 0.01$, at least 100 sequences must be examined, otherwise it would be difficult to observe a possible rejection.
A brief overview of the 15 tests is depicted in Table 3.1; for a more detailed description, refer to [26].

[1]https://github.com/terrillmoore/NIST-Statistical-Test-Suite.git

TYPE OF TEST	**IT CHECKS ON:**
Frequency	Equal proportion of 1s and 0s in the entire sequence
Block Frequency	Equal proportion of 1s and 0s in M-bit blocks
Cumulative Sums	Maximal excursion of the cumulative sum random walk (a defined sequence of steps). For the cumulative sum, the digits of the sequence are adjusted ($0 \rightarrow -1$)
Runs	Total number of uninterrupted sequences of identical bits (runs)
Longest Run	Longest run of 1s in M-bit blocks
Rank	Rank of disjoint sub-matrices of the sequence
FFT	Peaks in the Discrete Fourier Transform of the sequence
Non Overlapping Template	Number of occurrences of m-bit target strings
Overlapping Template	Same as the previous test, difference in how the m-bit string slides of 1 bit along the sequence
Universal	Number of bits between matching patterns
Approximate Entropy	Frequency of all possible overlapping consecutive m-bit patterns in the sequence
Random Excursions	Number of cycles having exactly K visits in a cumulative sum random walk
Random Excursions Variant	Number of visits to a specific state in a cumulative sum random walk
Serial	Frequency of all possible overlapping m-bit patterns in the sequence
Linear Complexity	Length of the LFSR that would generate M-bit blocks

Table 3.1: NIST statistical test suite overview

The listing below depicts a possible output file produced by the tests (in this case the Frequency test).

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
  generator is <./results/to_analyze/rnd_out_13INV_32RO_30sigma.txt>
------------------------------------------------------------------------------
 C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 P-VALUE PROPORTION STATISTICAL TEST
------------------------------------------------------------------------------
 16  9 12  6 15 10 13  7  5  7 0.145326   99/100   Frequency
```

For each statistical test, a row of 12 results is reported. The first 10 represent the distribution of P-values of the m analyzed sequences: the unit interval has been divided into ten discrete bins to have a measure of uniformity of results. The next number is the resulting P-value from a chi-square test; as mentioned above, it indicates whether the test is successful or not. Finally, the last result corresponds to the **proportion of sequences passing the test** compared to the total number applied. In order to be acceptable, this value should be in the **confidence interval** defined as $\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$, with $\hat{p} = 1 - \alpha$.

3.2.2 NIST SP 800-90B Tests

Along with the discussed statistical test suite, NIST also provides additional tests[2] to assess the entropy of a RBG. There are four different variations:

- **IID tests**: these tests determine whether the source generates independent and identically distributed samples or not. If just one of the IID tests is failed, then the bitstream can be defined as non-IID and the next set of tests will be applied. If this is not the case, the IID tests will already provide a result of entropy. They are divided into two classes:
  - **permutation testing**: different statistical tests are performed on 10000 permutations of the original dataset. The results are then compared to observe whether the permutation of samples can affect the entropy level of the stream.
  - **chi-square statistical tests**: additional tests to discover dependencies between successive samples and/or regularities in the samples' distribution.

[2]https://github.com/usnistgov/SP800-90B_EntropyAssessment.git

- **non-IID tests**: they provide an entropy assessment after the non-IID declaration, obtained from the previous battery of tests. Ten different estimators are used, each one of them producing an entropy assessment. The lowest resulting min-entropy is taken as the final result.
- **restart tests**: in this case, a series of sequences of samples are collected after the restart of the source and then tested, similarly to what has already been discussed in subsection 3.1.1 for health tests.
- **conditioning tests**: these tests must be applied to the collected samples after the conditioning component. Even though it usually decreases the bias of samples, the conditioning function may however cause a lower final entropy, due to its inherent determinism.

Also in this case, the analyzed data must be composed of a large number of samples, at least equal to 10^6.
For more details on the available specific statistical tests and estimators, refer to [25].

Chapter 4

Hardware Implementation

This chapter will focus on the hardware implementation of the true random number generator, both with and without the additional conditioning.
Among all the possibilities discussed in chapter 2, a **ring oscillator** approach has been selected to implement the noise source. Besides the certified high robustness and security level that arises from the study of literature, the other advantage of this choice is given by the realization of flexibility: ROs can be easily implemented both in ASICs, using standard cell libraries, and FPGAs. Even though most of the papers in the literature focus on TRNGs realized on FPGA, this work aims to propose a versatile design methodology. The goal is an implementation with a good trade-off among all the different PPA and entropy parameters, despite the employed platform.
As previously anticipated, the additional **conditioning** makes use of an accelerator implementing the **Keccak** function, that can be also exploited as a stand-alone component.
The design presents a hierarchical structure and employs the **SystemVerilog** language, with the only exception being the Keccak block, previously developed in VHDL. Furthermore, whenever possible, parameterization has been applied. In this way, not only the design can be adjusted depending on the requirements, but it is also possible to examine how the tuning of the different parameters can affect the final results.
Testing and intermediate analyses have been carried out using both Questasim and Synopsys and by developing auxiliary Python scripts.

4.1 TRNG implementation without conditioning

Figure 4.1 shows the schematic of the complete TRNG block without conditioning (clock and asynchronous active-low reset signals have been omitted).

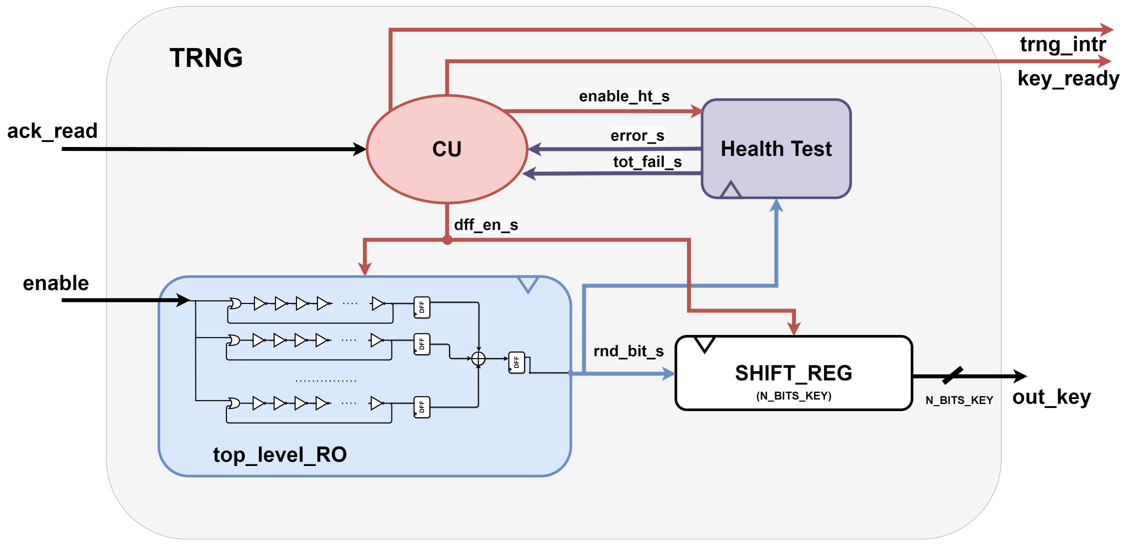


Figure 4.1: TRNG schematic (clock and asynchronous active-low reset signals omitted)

The system takes four input signals: besides `clk` and `rst_n`, the `enable` signal is used to start up the noise source, whereas `ack_read` is received from the external as an acknowledge that the last generated key has been read.
After enabling the TRNG, the RO-based noise source (`top_level_RO`) will start to produce random bits serially. To have a parallel output, a configurable `N_BITS_KEY` shift register is inserted to get a key of the desired number of bits. The correct operation of the noise source is checked on-the-fly by the component implementing the health tests.
The TRNG also provides a flag that notifies whether the key is ready to be used or not (`key_ready`) and an interrupt (`trng_intr`).
In the following subsections, a more detailed description of the single components will be presented.

4.1.1 Noise source

As previously mentioned, the entropy source is based on an RO configuration (Figure 4.2).

The circuit is structured hierarchically: it is composed of a series of parallel ROs, each of them formed by a certain number of inverters. The design has been

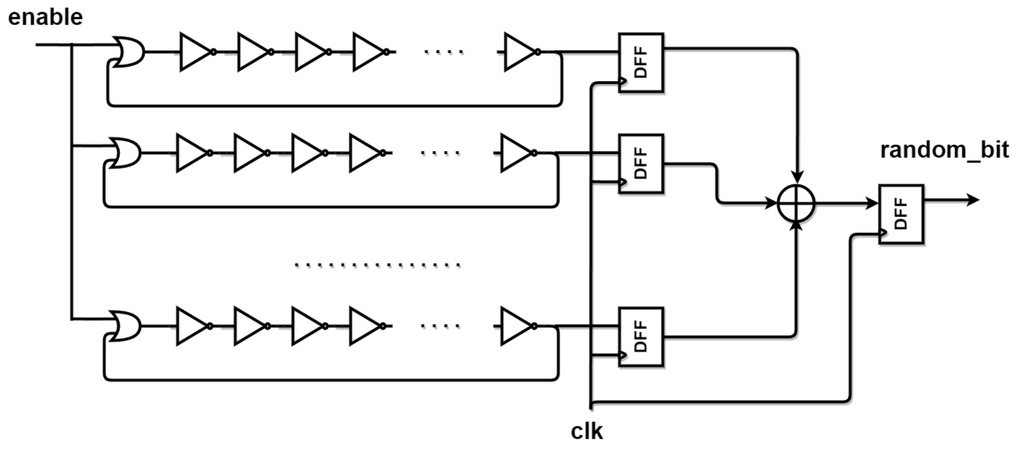


Figure 4.2: Noise source hardware implementation (clock and global reset signals omitted)

parameterized to be able to select both these quantities flexibly. A discussion on the choice of design parameters will be detailed in the next paragraph. The OR gate of each RO is used to connect the `enable` signal, hence starting the oscillation. This signal should not be constantly active, otherwise, the behavior of the ring oscillator would be fixed and the circuit would not work correctly.
A bit of each RO is sampled by a D flip-flop; all the FFs output bits are XORed together to increase randomness. Finally, the XOR output is sampled again and a random bit is then generated.

Design parameters analysis

The number of parallel ROs and the number of inverters for each RO affect not only the area, power, and maximum frequency of the circuit but also the statistical properties of the output bitstream. Consequently, particular attention has been paid in finding the best trade-off among these parameters.
The PPA analysis has been carried out by observing different syntheses' reports on Synopsys' Design Compiler, fixing the number of parallel ROs, and varying the number of inverters for each RO and vice-versa. The minimum considered configuration is composed of 3 inverters and 4 ROs, the maximum one corresponds to 65 inverters and 32 ROs (obviously, for ring oscillators only odd numbers of inverters are taken into account).
The results are reported on the graphs of Figures 4.3 - 4.5.

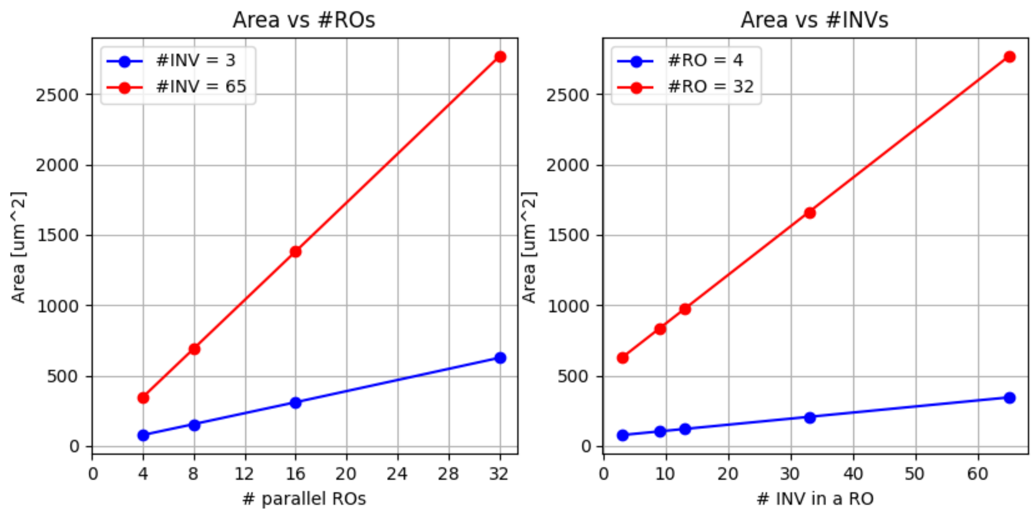


Figure 4.3: Area vs number of parallel ROs, fixing the number of inverters for each RO (left) and vs number of inverters for each RO, fixing the number of parallel ROs (right)

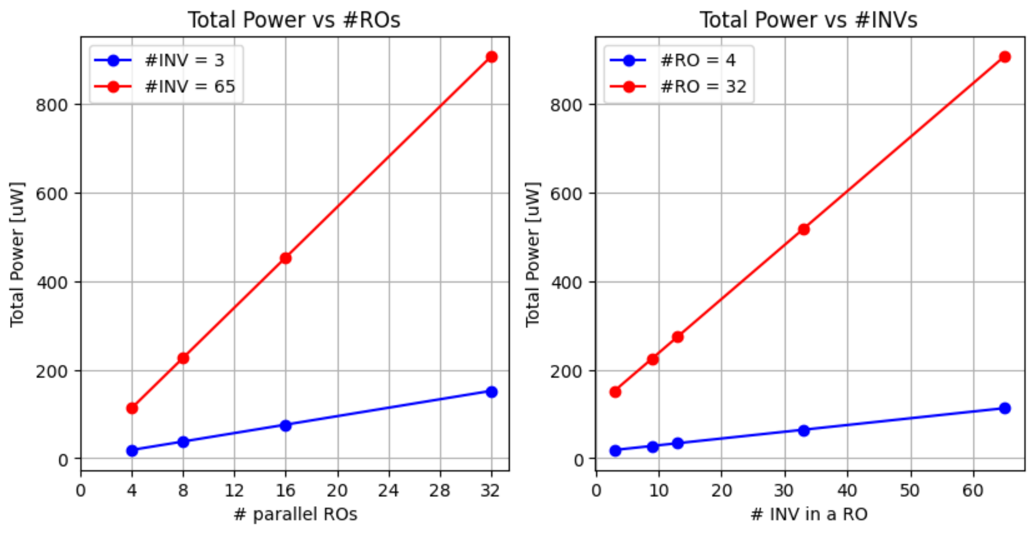


Figure 4.4: Power vs number of parallel ROs, fixing the number of inverters for each RO (left) and vs number of inverters for each RO, fixing the number of parallel ROs (right)

Obviously, area and power grow linearly with the number of ROs and inverters, with an impact of the increasing number of parallel ROs slightly higher with respect

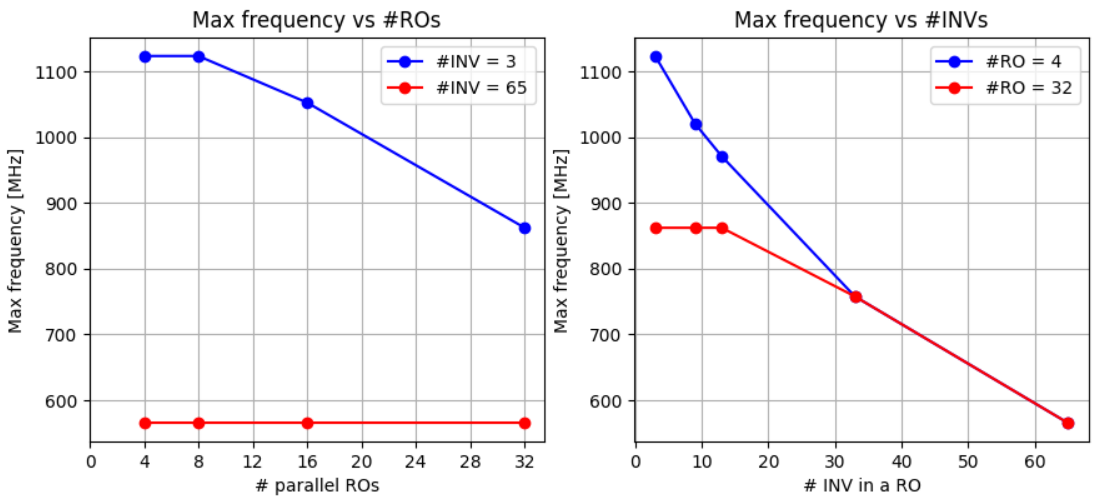


Figure 4.5: Maximum frequency vs number of parallel ROs, fixing the number of inverters for each RO (left) and vs number of inverters for each RO, fixing the number of parallel ROs (right)

to an increasing number of inverters for each chain. The influence of the number of inverters is however prevailing when considering the critical path, whereas the ROs are in a parallel configuration, hence they will not affect this aspect. Nevertheless, Figure 4.5 seems to contradict this last observation. The reason is that, for a number of inverters lower than 33, the critical path of the circuit is not yet given by a single-ring oscillator.
One of the most challenging part of the TRNG design is to combine PPA properties to statistical ones. As a matter of fact, by varying the number of inverters and ROs in the noise source, also the randomness of the output bitstream changes, as well as its entropy. The NIST statistical tests have been applied to output data coming from the same different configurations analyzed with Synopsys' Design Compiler. The proportions of sequences passing the 15 tests of the NIST statistical test suite for three different cases of design parameters are reported in Figure 4.6.
Ideally, the proportion of sequences passing a test should be in the acceptable range indicated by the dotted lines (confidence interval defined in subsection 3.2.1). It can be observed that this is true only for the case with the highest number of inverters and ROs among the reported ones. For this reason, **32 parallel ring oscillators and 13 inverters** have been chosen as final design parameters. In this way, all the NIST tests are passed and a good trade-off in terms of area (approximately 1000 μm^2), power (approximately 280 uW) and maximum frequency (approximately 880 MHz) is achieved.

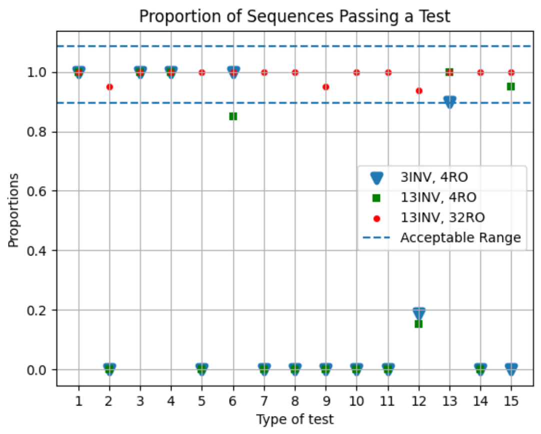


Figure 4.6: Proportions of sequences passing the 15 tests of NIST suite for different design parameters

Simulation

Being the randomness of physical phenomena the core aspect of the true random number generator, the functional simulation is one of the most critical points of the whole design. The main issue is the inherent ideality of a simulation which clashes with the key idea underlying the TRNG.
The adopted strategy to overcome this problem consists in assigning a certain delay to each inverter of each RO, similarly to what is done in [8]. Figure 4.7 displays the basic model.

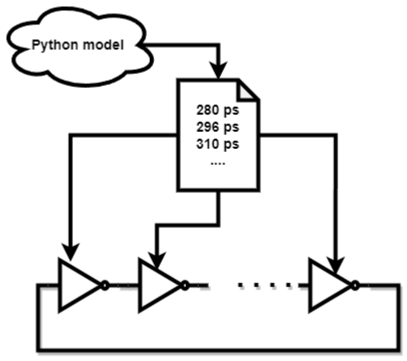


Figure 4.7: Delay simulation model

The different delay values are generated using a Python script, following the jitter model of section 2.1 and the directives of [8]. Only the local Gaussian jitter is taken into account, whereas the global deterministic component is neglected under the initial assumption of not having injection attacks. Consequently, each delay value is obtained by superimposing a Gaussian random variable with mean equal to 0 and standard deviation σ= 30 ps to a nominal inverter delay (typical values reported in [8]). The script is reported in Appendix A.

In SystemVerilog this can be easily done with a line:

```
out <= #delay ~in;
```

Being not synthesizable, this command will only help to assign a certain delay value to the inverter for the simulation. The variable `delay` is read from the Python model file employing a task in the testbench, which simply consists in the parsing of the file. The code is reported in the appendix A.
The different delay values of each inverter of each RO are then assigned hierarchically starting from the top level of the TRNG architecture.
An important note must be made: since the delays are of the order of hundreds of ps, the simulation must be performed with a resolution of 1 ps.

4.1.2 Health Test

As described in subsection 3.1.1, the health test component has the function of checking the correct runtime behavior of the noise source. The two main tests are implemented: the Repetition Count and the Adaptive Proportion tests.
The schematic of the component is shown in Figure 4.8, the `clk` and `rst_ni` signals are omitted. The output bit generated by the noise source (`rnd_bit`) is used in two separate ways to implement both tests.

For the **Repetition Count test**, the bit is used as input for a configurable (`NBITS`) shift register. Its lenght, i.e. `NBITS`, which corresponds to the cutoff value of the test, is parameterized to be changed depending on the chosen false-positive probability α. The shift register parallel output and its negated version is sent to two AND gates to determine whether the sequence of bits is stuck at 1 or 0. If one of these cases is true, the output `error` flag is set to `1` and sent to the control unit.
The other parts of the component are employed to implement the **Adaptive Proportion test**. It only realizes in hardware what is described by Algorithm 2: a window of samples, `W`, properly controlled by means of a counter, is accumulated; if the final value of the accumulator does not belong to the correct range, meaning that there is a prevalence of 1s or 0s, the `error` signal is set to `1`. Since binary data are considered, the window size `W` is fixed to 1024 bits, while the `CUTOFF` value, determining the allowed range, is parameterized so that it can be tuned accordingly to the expected entropy value, as indicated by NIST ([25]).
An additional counter is inserted to count the number of consecutive errors: if this

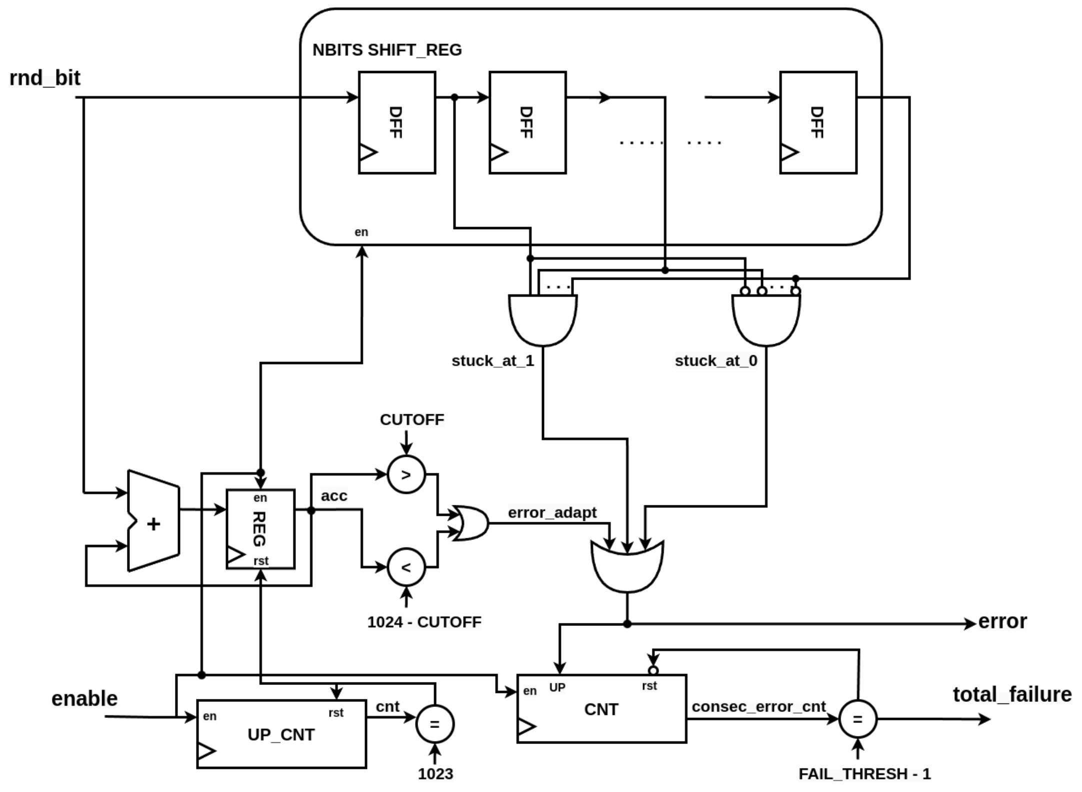


Figure 4.8: Health test component schematic (clock and global reset signals omitted)

value is equal to the constant `FAIL_THRESH`, the `total_failure` output signal is high and the control unit will declare the unrecoverable DEAD state. Also in this case, the `FAIL_THRESH` value is parameterized.

4.1.3 Control Unit

The control unit (CU) synchronizes the whole system. It is organized as a classical finite state machine (FSM), its state diagram is displayed in Figure 4.9.

The FSM is an extended version of the one proposed in the RISC-V Cryptographic Extension documentation [1]. It is composed of six states: `IDLE`, `BIST`, `WAIT`, `ES32`, `WAIT_FOR_ACK` and `DEAD`.
Figure 4.10 shows the CU timing diagram when no errors are detected by the

[1]https://github.com/riscv/riscv-crypto.git

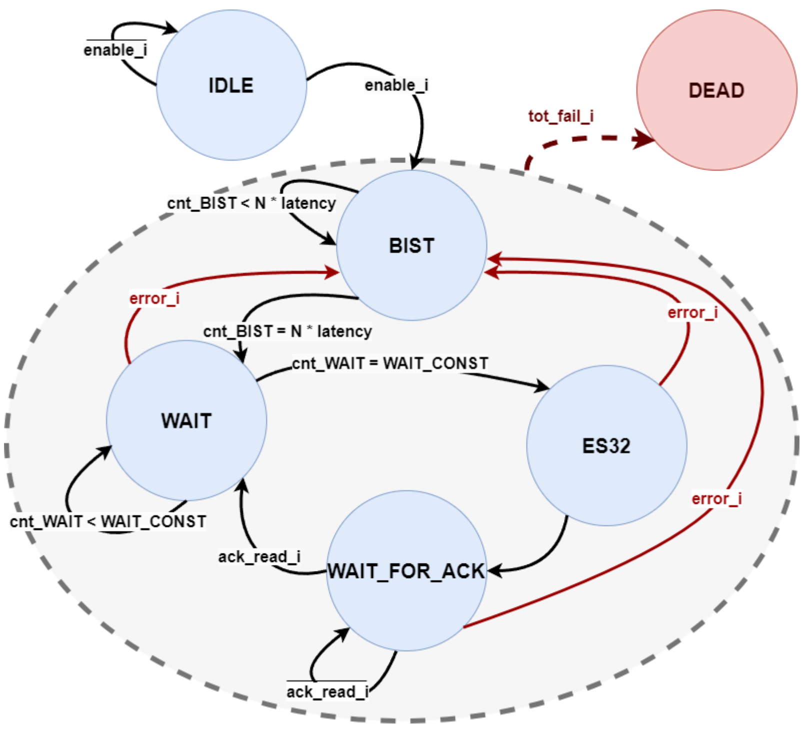


Figure 4.9: Control unit state diagram

health tests. Initially, the system is in a `IDLE` state, everything is switched off. When the `enable` input signal of the TRNG arrives, the FSM will move to the `BIST` state, i.e. the warm-up phase, and all the components of the TRNG are switched on. Since the TRNG relies on physical phenomena, an adjustment phase is useful to have a noise source behaving as expected. The `BIST` state can be seen as a sort of conservative start-up health test, in which the random bits are continuously checked but no output key is produced. The duration of this phase is parameterized through the `latency` constant and can be chosen also considering the specific operating conditions of the circuit.
After `BIST`, the serial random bits of the noise source can be collected into the shift register to produce the output key: this is the `WAIT` state. As in the `BIST` case, a parameterized counter (`WAIT_CONST`) is used to determine how many cycles this phase will last, depending on the output key parallelism.
When this interval of time elapses, the FSM moves into the `ES32` state: the key is ready, and the interrupt and the `rnd_ready` flag can be set to 1. This phase only lasts one clock cycle. The system will then wait for the external acknowledgment of the key (`WAIT_FOR_ACK` state) and, when it is received, it moves back to the `WAIT` state to generate another key.
The `WAIT_FOR_ACK` state may not be necessary: in this case, it has been inserted

to have both the interrupt and the `rnd_ready` flag set to 1 for just one clock cycle, which is convenient for the conditioning component integration (section 4.2). The CU also outputs the `flush_regs_o` signal. It may be useful when the TRNG key needs to be readable by external devices immediately after its generation, only for a specific number of clock cycles. In this case, `flush_regs_o` would allow to reset the buffer containing the key.

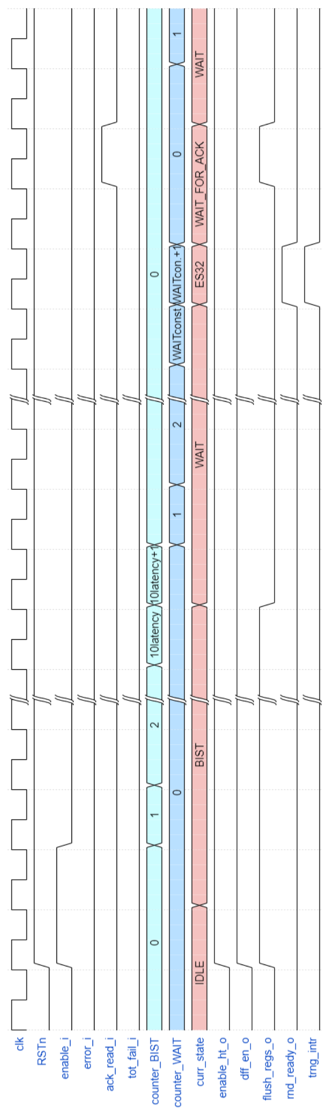


Figure 4.10: Timing diagram of the control unit when no errors arise

The described FSM flow occurs when no errors are detected by the health test component. On the other hand, Figure 4.11 reports the CU timing diagram in case of errors or total failures. Whenever the error flag set by the health tests is equal to 1, the system goes back to the `BIST` state, regardless of the current state. The `counter_BIST` signal, keeping track of the number of cycles that the system must spend in `BIST` state, will start to increment only if the error flag is sent back to 0, meaning that the errors have been recovered. In this case, the system can come back to its normal operation, following the normal FSM flow. However, if the error persists and the health tests detect a `total_failure` condition, the system goes to the unrecoverable `DEAD` state and must be rebooted.

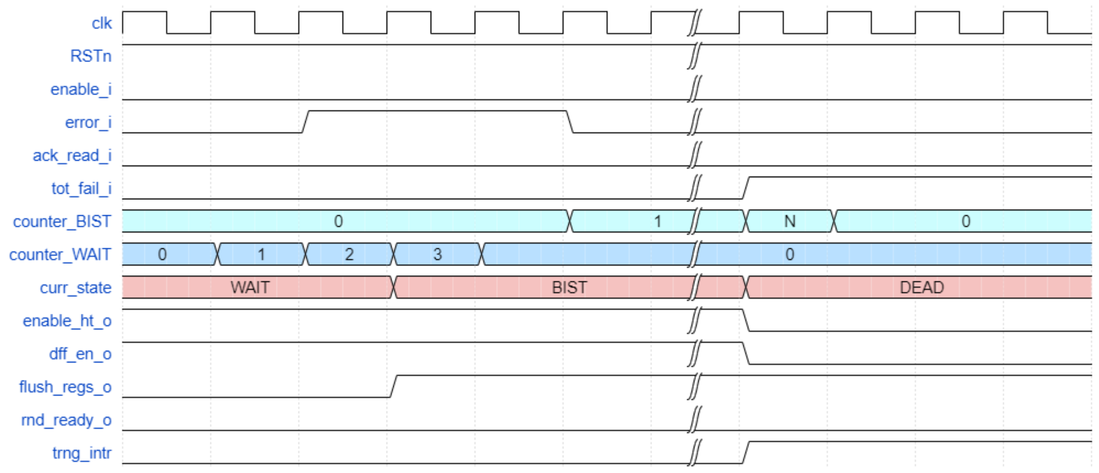


Figure 4.11: Timing diagram of the control unit in case of simple errors or total failure condition

4.2 Keccak conditioning

The Keccak accelerator block developed in [30] has been used as an optional conditioning. The block takes a 1600-bit input and outputs 1600 processed bits. The processing starts after the reception of the `start` pulse; after 24 clock cycles, a `status` flag and an interrupt notify that the output is ready.
The architecture has been developed both with a unique CU for the TRNG and the Keccak blocks and with separate CUs. In this way, the most convenient architecture can be chosen depending on the requirements. This point will be better clarified in the next chapter.
Figure 4.12 shows a schematic with the main signals. In this case, each block has its internal control unit, but with a unique CU the operating principle remains the same.

The system's behaviour is determined by the `conditioning` signal:

- `conditioning = 0`: the TRNG and Keccak blocks are independent of each

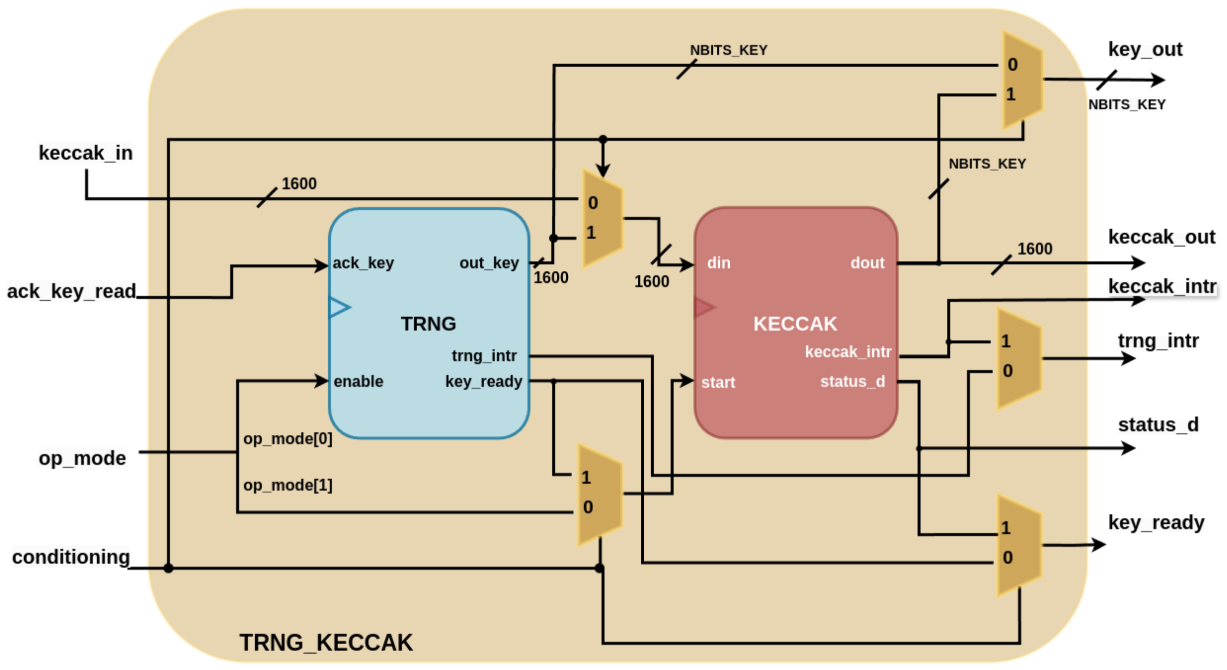


Figure 4.12: TRNG with optional Keccak conditioning

other. The TRNG output is not conditioned, the output key will be produced in the same way as described in the previous section. At the same time, the Keccak unit can be also exploited to process different data (`keccak_in`) as a stand-alone component.

- `conditioning = 1`: the two blocks work together. The output key generated by the TRNG is the input of the Keccak unit and the `key_ready` flag acts as `start` pulse. In this case, the final output key is ready only after the Keccak processing and it must be extracted from its 1600-bit output. Consequently, the `status_d` flag is connected to the final `key_ready` signal of the `TRNG_KECCAK` block. In the same way, the interrupt of the system corresponds to the interrupt of the Keccak unit.

A timing diagram with the essential signals is reported in Figure 4.13.

Two observations must be made. The first one concerns the number of bits of the output key produced by the TRNG, which must be fixed to 1600 or slightly lower. As a matter of fact, the `conditioning` input could be switched from 0 to 1 runtime, meaning that the TRNG output must be ready to be used as input of the Keccak block. The parallelism could also be slightly lower than 1600 bits due to the padding operation performed by the Keccak function. This can be important to increase the throughput since the TRNG produces random bits serially. However,

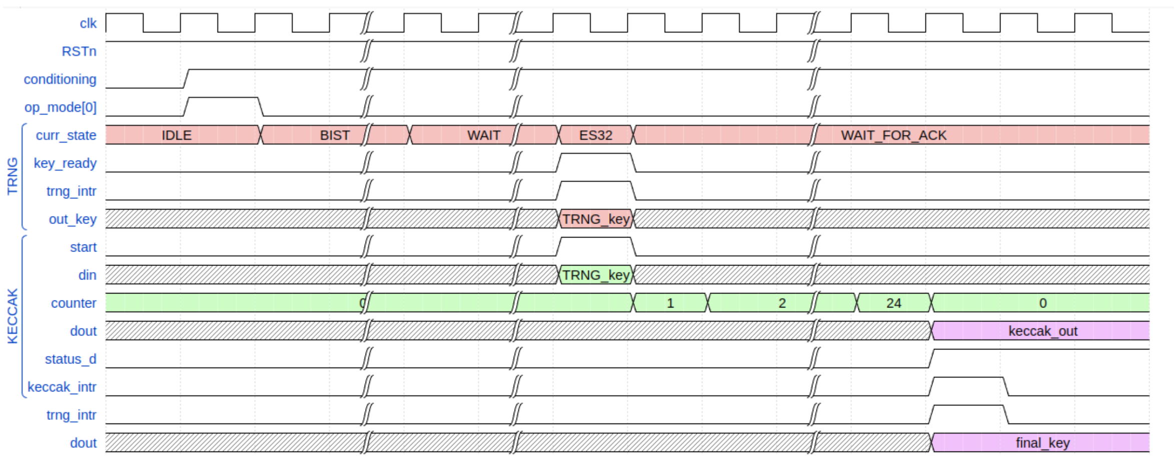


Figure 4.13: TRNG + Keccak timing diagram

too few bits may result in an insufficient randomness of the final key. Therefore, a trade-off analysis between randomness and throughput must be taken into account. The second note is about the extraction of the final output key from the 1600-bit Keccak output. Usually, a lower number of bits are needed by the key, hence a truncation must be performed. Since the objective is to obtain a random output, MSBs, LSBs or different combinations of bits are all valid choices. A study on this additional degree of freedom may also be carried out to increase the entropy of the final key.

Chapter 5

X-HEEP Integration

5.1 X-HEEP

X-HEEP (eXtendable Heterogeneous Energy-Efficient Platform)[1] is a 32-bit RISC-V based customizable microcontroller, developed by the Embedded Systems Laboratory (ESL) at the Swiss Federal Institute of Technology in Lausanne (EPFL). [31] The main purpose of X-HEEP is to offer a flexible platform for the integration of ultra-low-power edge accelerators. Configurability is one of the main features: the user can choose among different CPU micro-architectures, memory sizes, and bus topologies.
Figure 5.1 shows the X-HEEP MCU. The architecture includes:

- **CPU** : the user can choose among three different RISC-V cores (CV32E20, CV32E40X, and CV32E40P), selected from the OpenHW Group Core-V family. They offer different possibilities in terms of power and performance trade-offs.
- **memory subsystem**: the memory size and the number of memory banks can be selected depending on the specifications. The memory models are derived from the PULP platform, a project born with the same intent as X-HEEP.
- **bus subsystem**: the open-bus interface (OBI) is used to ensure compatibility with IPs coming from different projects (OpenHW Group, PULP, OpenTitan). The user can choose between a one-at-a-time topology (only one master allowed) and a fully connected topology (multiple masters).
- **Peripheral subsystem**: it includes an interrupt controller (PLIC), timers, and general-purpose I/O peripherals such as GPIO and I2C. These IPs can be turned off in case of strict power consumption requirements.

[1]https://github.com/esl-epfl/x-heep.git

- **Always-on peripheral subsystem**: in this case, the IPs are always on. The SoC controller, boot ROM, power manager, fast interrupt controller, and DMA are originals of the X-HEEP platform. In particular, the power manager unit implements low-power strategies, such as clock-gating, power-gating, and RAM retention. The whole system is indeed divided into different power domains.

- **Debug subsystem** : inherited by the PULP platform.

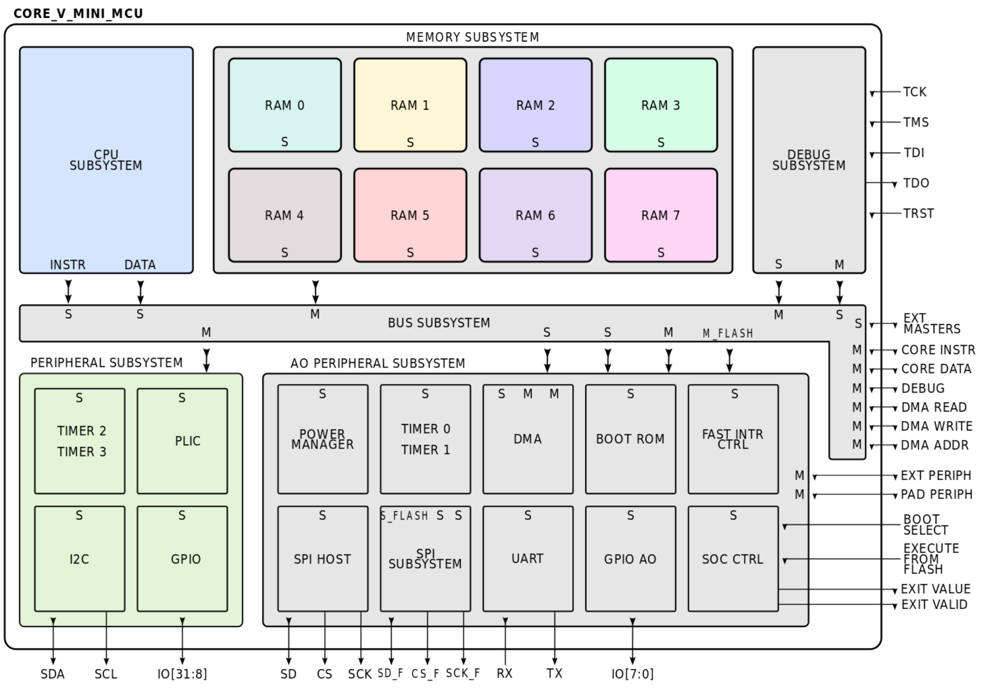


Figure 5.1: X-HEEP MCU

To facilitate the integration of external accelerators, the platform presents the configurable XAIF interface, which allows it to accommodate the X-HEEP system to the specific requirements of the different accelerators. The XAIF interface includes a configurable number of interrupt ports, a configurable number of master and slave ports in the bus subsystem, and a customizable power control interface. X-HEEP makes use of the **FuseSoc** build system to simplify the processes of simulation and FPGA/ASIC implementation with different EDA tools.
The following sections will focus on the description of the integration of the TRNG, alone and with the Keccak block, as an external accelerator of the X-HEEP MCU.

5.2 TRNG integration

The TRNG integration process can be divided in six steps.

1. Generation of a **register file** for the TRNG.
2. Creation of a **wrapper** including the TRNG and the register file.
3. Integration of the TRNG wrapper in the **X-HEEP wrapper**.
4. Creation of the FuseSoc new **core file**.
5. Development of TRNG **drivers**.
6. Creation of the **Makefile**.

5.2.1 Step 1: Register file

The first step for the TRNG integration is to create a **register file** so that the accelerator can be controlled through load and store operations by the X-HEEP core.
In order to have standardization with respect to the X-HEEP parallelism, the width of all registers has been chosen equal to 32 bits. The register file is composed of two registers:

- **Control-Status register**: can be written and read both by the CPU and the TRNG. Only three LSBs are used: the two control bits (`enable` and `ack_read`) are written by the CPU; the status bit (`key_ready`) is written by the accelerator.
- **Data register**: it contains the output key to be read by the CPU. In case of a key larger than 32 bits, it will be segmented into 32-bit chunks and read in successive cycles. Obviously, if the key width is fixed and known in advance, more than one data registers can be directly allocated.

This interface has been created by using OpenTitan's *Register Tool*[2], which employs a simple Python 3 script (`regtool.py`). By describing the registers in hjson format, the tool generates the register file in SystemVerilog and a C header file with the address offset of the registers, useful when creating the driver.
Besides the name and width of the registers, the tool allows to specify different options. Some examples are the primary clock and reset signals, the bus interfaces,

[2]https://github.com/lowRISC/opentitan.git

the hardware and software access permissions, and the different bit fields in which a register is divided. For instance, the hjson description of the data register is shown below:

```
name: "trng_data",
clock_primary: "clk_i",
reset_primary: "rst_ni",
bus_interfaces: [
{ protocol: "reg_iface", direction: "device" }
],
regwidth: "32",
registers: [
    { name: "TRNG_DOUT"
      desc: "Random key"
      swaccess: "ro",
      hwaccess: "hwo",
      hwext : "true",
      fields: [
          { bits: "31:0"
          }
      ]
    }
],
```

The generated SystemVerilog description will present a standardized interface. Specifically, two types of structures are provided to clarify the direction of the input and output signals: from the register to the TRNG (`reg2hw_t`) and vice versa (`hw2reg_t`).

5.2.2 Step 2: TRNG Wrapper

After creating the register file, the next step is to have a wrapper containing the TRNG and the register file to be connected to X-HEEP (`TRNG_WRAPPER` in Figure 5.2).
While the control register will be directly connected to the slave port of X-HEEP, the data registers will exploit the OBI protocol to communicate with an external bus, which will be then connected to the slave port. This difference is due to the possibility of using the DMA for fast data management. Because of this, the additional `periph_to_reg` component has been inserted in the `TRNG_WRAPPER` to allow the correct communication between the data register top level and the

external bus.

5.2.3 Step 3: Integration in X-HEEP wrapper

The `TRNG_WRAPPER` needs now to be connected to X-HEEP, as shown in Figure 5.2. X-HEEP has been configured with one slave port. As just mentioned, the signals of the control register of the `TRNG_WRAPPER` are directly connected to the X-HEEP slave port. On the other hand, the external bus links the data register to the same slave port, which is then connected to the internal X-HEEP bus.
The final operations consists of reserving a slot of memory addresses for the accelerator and assigning the TRNG interrupt to one of the `external_interrupt_vector` empty slots of X-HEEP.

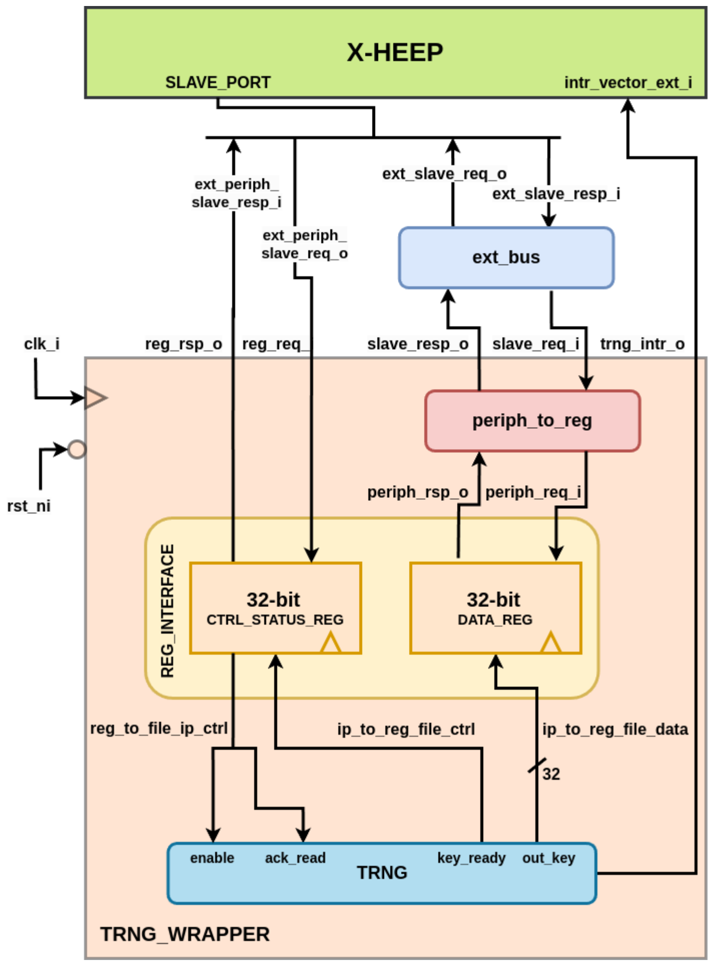


Figure 5.2: TRNG and X-HEEP connection

Additional modifications have been made to the testbench used by X-HEEP (`testharness.sv`). The most important one is the inclusion of the `assign_delays` task to correctly simulate the TRNG.

5.2.4 Step 4: FuseSoc core

Once having completed all the RTL connections to X-HEEP, a new FuseSoc core file is required.
A core file is written in YAML syntax and is used to describe a design in FuseSoc. The first line of the file specifies the CAPI 2 schema, which indicates the structure of the core file. Successively, the name of the design must be indicated. The rest of the file can be divided in different sections:

- `filesets`: first of all, all the involved sources must be specified, along with their type. They can be divided in different groups; dependencies on other cores can be also included. In this case, the TRNG source files have been added to the original X-HEEP filesets.
- `parameters` and `scripts`: parameters that are present in the source files may be specified here with their default value, for example, the type of core to be used by X-HEEP. Moreover, also specific scripts to be used in simulation or synthesis must be declared.
- `targets`: his section allows the configuration of the different simulation/synthesis processes, similar to what one would do within a specific tool. An example of the configuration of the Modelsim simulation is reported below:

```yaml
sim:
  <<: *default_target
  default_tool: modelsim
  filesets_append:
  - tb-harness_x_heep
  parameters:
  - use_cv32e40p_corev_pulp? (COREV_PULP=1)
  - "!use_cv32e40p_corev_pulp? (COREV_PULP=0)"
  - use_jtag_dpi? (JTAG_DPI=1)
  - "!use_jtag_dpi? (JTAG_DPI=0)"
  tools:
    modelsim:
      vlog_options:
      - -override_timescale 1ps/1ps
      - -suppress vlog-2583
```

```
        - -suppress vlog-2577
        - -suppress vlog-2720
        - -pedanticerrors
        - -define MODELSIM
        vsim_options:
        - -voptargs=+acc
```

After specifying the involved files to be compiled, the `vlog` and `vsim` options are provided so that everything is correctly set up for an automatic simulation. The same process can be repeated for different tools. In this work, the target section has been configured for Modelsim, Synopsys' Design Compiler, and Vivado.

5.2.5 Step 5: Driver

At this point, the TRNG will need a driver. Having memory-mapped the accelerator, all the memory locations will be automatically generated in the `trng_x_heep.h` header file and the specific registers offset constants have been already created through the `regtool.py` script.
The driver consists of a simple function called `get_rnd_key`:

```
void get_rnd_key(uint32_t* Dout);
```

The only argument of the function is a pointer to a `uint32_t` variable which will contain the output key.
The first operation performed by the driver is defining the pointers to the TRNG registers; their addresses are defined as macros in the `trng_x_heep.h` file.

```
uint32_t volatile *ctrl_reg = (uint32_t*) TRNG_CTRL_START_ADDR;
uint32_t volatile *Dout_reg = (uint32_t*) TRNG_DOUT_START_ADDR;
uint32_t volatile *status_reg = (uint32_t*) TRNG_STATUS_START_ADDR;
```

After this initialization phase, the TRNG is triggered by writing a 1 in the correct bit of the control register:

```
*ctrl_reg = 1 << TRNG_CTRL_CTRL_TRNG_EN_BIT;
```

In the next line, a 0 is written again in the same position to just have an enabling pulse. Memory barriers are also inserted to ensure the correct succession

of operations to be executed.
At this point the status register is polled up to generation of the key, notified by the dedicated flag of the TRNG.

```
    do {
        key_ready = (*status_reg) & (1 << TRNG_CTRL_STATUS_TRNG_BIT);
    } while (key_ready == 0);
```

When the key is ready, the data register contains the 32-bit random key to be read.

```
    *Dout = Dout_reg[0];
```

The final line consists in writing a 1 in the proper bit position of the control register, to indicate the acknowledgement of the key.
Another version of the same function has been developed by exploiting the TRNG **interrupt** instead of **polling** the status flag. The dedicated functions of the interrupt controller peripheral of X-HEEP (header file `rv_plic.h`) are used for the initialization. The correct interrupt line is the one that was previously assigned in the RTL, in the third step.

```
plic_Init(); // Init the PLIC
plic_irq_set_priority(EXT_INTR_0, 1); // Set the priority of the TRNG interrupt
plic_irq_set_enabled(EXT_INTR_0, kPlicToggleEnabled); // Enable the interrupt
```

To enable the TRNG interrupt, a few write operations in the control registers of the system are needed, as explained in the X-HEEP documentation [3].

```
// Enable global interrupt for machine-level interrupts
CSR_SET_BITS(CSR_REG_MSTATUS, 0x8);
// Set mie.MEIE bit to one to enable machine-level external interrupts
const uint32_t mask = 1 << 11; //IRQ_EXT_ENABLE_OFFSET;
CSR_SET_BITS(CSR_REG_MIE, mask);
```

Now that the interrupt is correctly set, the triggering of the TRNG can be done in the same way as before. The polling is now replaced by the `wait_for_interrupt()` X-HEEP library function.

[3]https://github.com/esl-epfl/x-heep.git

```c
while(plic_intr_flag==0) {
    wait_for_interrupt();
}
```

The `wait_for_interrupt()` function works as a NOP, meaning that the system will wait for the TRNG interrupt. The remaining lines of the driver are the same as in the polling case.

5.2.6 Step 6: Makefile

The final step to test the TRNG application is the creation of the new makefile, which is directly inherited from the original one of X-HEEP. This makefile, in turn, is built upon a hierarchy of different makefiles.
After including the new header files and the drivers, the remaining operation is to integrate the new rules to compile, link, and run the new applications.

5.2.7 TRNG accelerator test

The application to test the drivers is very simple. It just recalls the two functions and prints the output hexadecimal keys:

```c
int main()
{
    static uint32_t Dout;

    get_rnd_key(&Dout);
    printf("Key: %08X \n", Dout);
    get_rnd_key_intr(&Dout);
    printf("Key: %08X \n", Dout);

    return EXIT_SUCCESS;
}
```

Furthermore, performance counters in each function are used to trace the number of required clock cycles to execute the driver.
Figure 5.3 and Figure 5.4 show two snippets of the simulation on Questasim, proving that the accelerator and the driver work properly.
By recalling the functions, one can obtain an output similar to the one displayed in Figure 5.5.

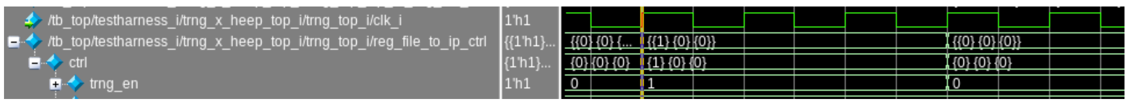

/tb_top/testharness_i/trng_x_heep_top_i/trng_top_i/clk_i
/tb_top/testharness_i/trng_x_heep_top_i/trng_top_i/reg_file_to_ip_ctrl
ctrl
trng_en
1'h1
{{1'h1}...
{1'h1}...
1'h1

Figure 5.3: Questasim simulation - write operation in the control register to enable the TRNG

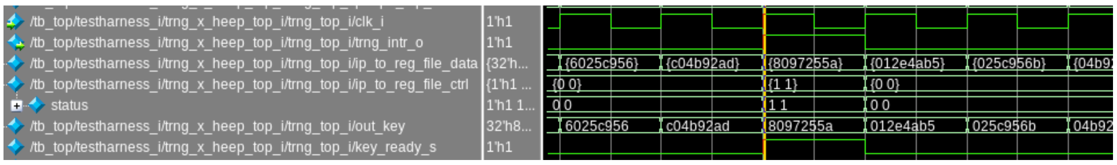
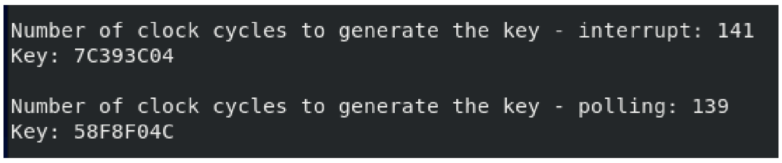

/tb_top/testharness_i/trng_x_heep_top_i/trng_top_i/clk_i
/tb_top/testharness_i/trng_x_heep_top_i/trng_top_i/trng_intr_o
/tb_top/testharness_i/trng_x_heep_top_i/trng_top_i/ip_to_reg_file_data
/tb_top/testharness_i/trng_x_heep_top_i/trng_top_i/ip_to_reg_file_ctrl
status
/tb_top/testharness_i/trng_x_heep_top_i/trng_top_i/out_key
/tb_top/testharness_i/trng_x_heep_top_i/trng_top_i/key_ready_s
6025c956
c04b92ad
8097255a
012e4ab5
025c956b

Figure 5.4: Questasim simulation - the key is ready, the respective flags are set to 1 and the bits will be collected from the data register



```
Number of clock cycles to generate the key - interrupt: 141
Key: 7C393C04

Number of clock cycles to generate the key - polling: 139
Key: 58F8F04C
```

Figure 5.5: Output of the TRNG test application

After analysing the results of several tests, it has been found that no more than **150 cycles** are required by the drivers.

5.3 TRNG and Keccak integration

The integration of the TRNG with the Keccak conditioning has been realised following the same procedure described in the previous section. An overview on the **main differences** will be now presented.
The first one concerns the **register file**. Since the TRNG and Keccak components can also be used as stand-alone accelerators (section 4.2, case `conditioning = 0`), additional data registers must be added to the register file for the 1600-bit Keccak input and output. Keeping the 32-bits parallelism, 50 registers are allocated for the input data and 50 more are allocated for the output data, as depicted in Figure 5.6. Therefore, the output data register structure has been conceived to have 51 32-bits registers, 50 for the `keccak_out` signal and the last one for the output random key. For what concerns the Keccak control and status bits, they are included in the same control/status register used for the TRNG, as only three bits were actually used. An additional bit is also reserved for the `conditioning` signal.
The structure of the wrapper remains unchanged, as well as the connections

to X-HEEP. The only modification consists in adding another element to the `external_interrupt_vector`, because of the presence of the **additional Keccak interrupt signal**. Furthermore, a larger slot of memory locations may be needed, meaning that also the memory-mapping might require a modification.
Obviously, the FuseSoc core has been updated in order to include also the Keccak component source files.

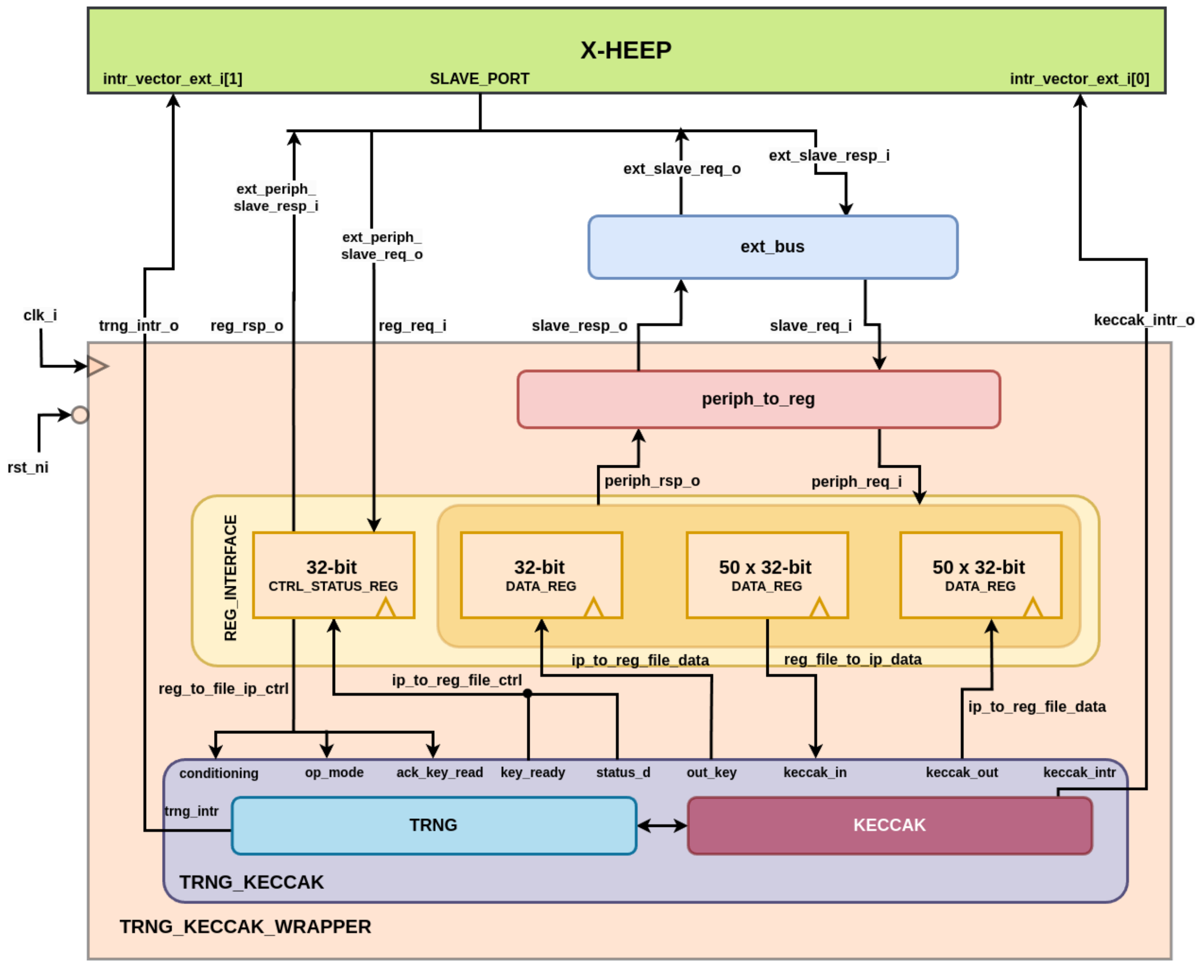


Figure 5.6: TRNG-Keccak and X-HEEP connection

Even for the driver, only a minor change is needed. It is important to specify that the driver for the standalone Keccak component will not be discussed, as the case study of this work is the TRNG.
The previous driver has been modified by adding another argument:

```
void get_rnd_key(uint8_t conditioning, uint32_t* Dout);
```

Depending on the value of this parameter, the `conditioning` bit of the control register will be set to 1 or 0:

```
if (conditioning == 1)
    *ctrl_reg = 1 << TRNG_KECCAK_CTRL_CTRL_CONDITIONING_BIT;
else
    *ctrl_reg = 0 << TRNG_KECCAK_CTRL_CTRL_CONDITIONING_BIT;
```

The rest of the code remains unchanged, as the correct signal routing depending on the conditioning bit has already been made in hardware. The only difference is in the data register containing the key: as previously mentioned, the 51st register must be read.

```
  *Dout = Dout_reg[50];
```

Chapter 6

Results and comparison

This chapter will provide an overview of the final results in terms of randomness, power, area, and performance. A comparison with other recent works in literature is also included. The last section presents a summary of potential improvements and ideas to further develop the proposed TRNG.

6.1 NIST Tests Results

The TRNG before and after conditioning has been tested by means of the NIST tests described in chapter 3.
For what concerns the entropy, the NIST SP 800-90B has been applied to the raw bitstream of the TRNG, identifying a non-iid distribution with a **Shannon Entropy** per bit equal to **0.9995**.
The results of the NIST SP 800-22 statistical test suite are reported in table 6.2, for multiple outcomes of the same test (Non Overlapping Template, Random Excursions, Random Excursions Variant) the average values have been considered. $2 \cdot 10^7$ bits have been analyzed: in the first part of the table the output stream is directly produced by the noise source (Figure 4.2), in the second part it is the result of the Keccak block post-processing. The block length parameters of the tests have been chosen following the NIST directives and are reported in Table 6.1.

Since a level of significance $\alpha = 0.01$ has been considered, a test is passed if its P-value is higher than 0.01, as explained in chapter 3. Therefore all tests are passed. Moreover, the P-values after the Keccak conditioning are on average higher, indicating a more uniform distribution and a less biased output bitstream.

Statistical Test	**Block Length**
Block Frequency Test	128
Non Overlapping Template Test	9
Overlapping Template Test	9
Approximate Entropy Test	10
Serial Test	12
Linear Complexity Test	500

Table 6.1: Parameters for NIST SP 800-22 test suite

Test	**No conditioning**			**Conditioning (Keccak)**		
	P-value	**Prop.**	**Result**	**P-value**	**Prop.**	**Result**
Frequency	0.3505	1	PASS	0.7399	1	PASS
BlockFreq.	0.0127	0.95	PASS	0.1223	1	PASS
Cumulat.Sums	0.6885	1	PASS	0.5078	1	PASS
Runs	0.9114	1	PASS	0.4373	0.95	PASS
LongestRun	0.8343	1	PASS	0.7399	0.95	PASS
Rank	0.3505	1	PASS	0.7399	1	PASS
FFT	0.3505	1	PASS	0.1626	1	PASS
NonOver.Templ.	0.4598	0.99	PASS	0.4945	0.99	PASS
Over.Templ.	0.2757	0.95	PASS	0.6371	1	PASS
Universal	0.5341	1	PASS	0.9114	0.95	PASS
Approx.Entr.	0.4373	1	PASS	0.6371	1	PASS
Rand.Excurs.	0.1091	0.984	PASS	0.3714	0.990	PASS
Rand.Exc.Var.	0.0838	0.993	PASS	0.2367	0.995	PASS
Serial	0.3505	1	PASS	0.3943	0.975	PASS
LinearComplex.	0.9643	1	PASS	0.4373	1	PASS

Table 6.2: NIST SP 800-22 test results

Also, the proportion of sequences passing each test is in the acceptable range (Figure 6.1).

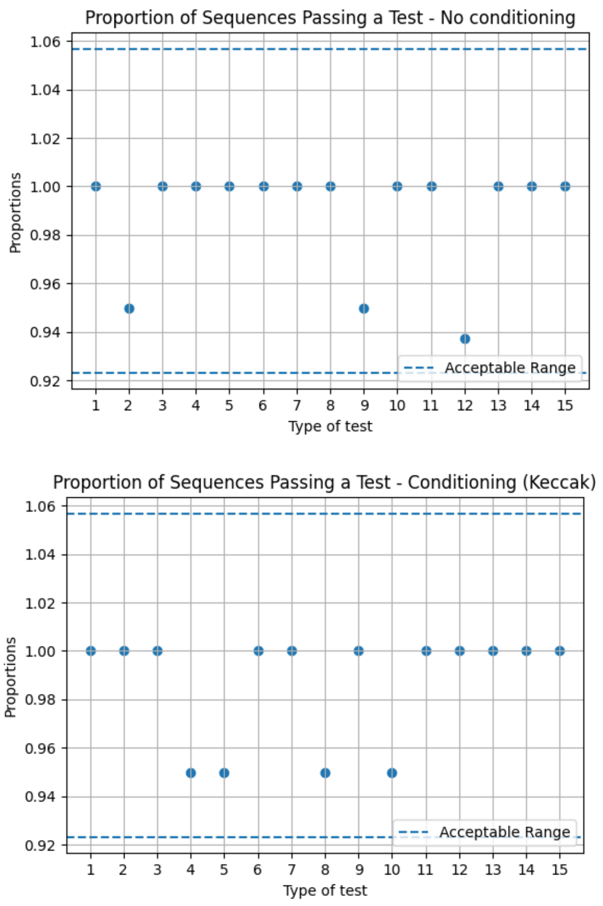


Figure 6.1: Proportions of sequences passing the 15 NIST tests - before (up) and after conditioning (down)

6.2 Modelsim Results

The proposed TRNG implementation has been tested by exploiting the C implementation of the **CRYSTALS-Kyber algorithm** proposed by PQClean [1].
CRYSTALS-Kyber is a KEM (Key Encapsulation Mechanism), based on the Learning with Errors (LWE) problem. ([32])
CRYSTALS-Kyber algorithm presents three different versions, with different security levels and lengths (Table 6.3).

[1]https://github.com/PQClean/PQClean.git

Kyber Length [bytes]			
Kyber version	Secret Key	Public Key	Cipher Text
Kyber512	1632	800	768
Kyber768	2400	1184	1088
Kyber1024	3168	1568	1568

Table 6.3: Kyber Crypto Length

Following the algorithm, the PQClean implementation is composed of three main functions:

- `crypto_kem_keypair`: the public and private keys are generated.
- `crypto_kem_enc`: it generates the cipher text and the shared secret for the given public key (encryption phase).
- `crypto_kem_dec`: it generates the shared secret for the given cipher text and private key (decryption phase).

All the random values are generated by means of the `randombytes` function which actually produces a predictable sequence of bytes. As a matter of fact, it is declared that the function is used only for testing purpose, without any reliable cryptographic qualities. Therefore, by replacing this function with the TRNG driver, there is the a double advantage: having a faster execution and robust random keys.
The three variants have been tested on the X-HEEP MCU. Initially, the average number of clock cycles required for each original version has been measured. Subsequently, the TRNG accelerator has been used to replace the `randombytes` function. To get more than 32 bits, a minor modification to the driver has been made, such that an output buffer of the required length is filled 32 bits at a time. The code is reported in Appendix A. The results are presented in Table 6.4.

PQC Function	Avg. number of clock cycles		**Speed-up**
	SW	**With TRNG**	
Kyber512	3 083 978	3 036 592	**1.015**
Kyber768	4 994 839	4 938 277	**1.012**
Kyber1024	7 608 400	7 517 755	**1.012**
`randombytes`	27209	925	**29.415**

Table 6.4: Number of cycles required by PQClean Kyber functions, before and after the TRNG accelerator

The TRNG acceleration provides a speed-up higher than 29 for the single `randombytes` function. This value could be increased even more by modifying the

`N_BITS_KEY` value of the TRNG, reducing the software operations and directly obtaining the key of the desired length. At the same time, if considering the complete Kyber functions, the speed-up is minimal: the majority of the computational cost is indeed required by the Keccak function permutations.

6.3 ASIC Results

The system has been synthesized by means of **Synopsys' Design Compiler**, exploiting the **65nm** library. The results, reported in Table 6.5, refer to the case of the maximum achievable frequency. This means that the clock has been progressively reduced in order to achieve a slack equal to 0. This is reached for a clock equal to 1.10 ns in the TRNG without conditioning and to 1.4 ns in the presence of the Keccak block.

	Max freq. [MHz]	**Area [μm^2]**	**Power [mW]**
TRNG	909.1	3912.84	2.61
TRNG + Keccak	714.3 (÷1.27)	84130.92 (x21)	39.20 (x15)

Table 6.5: Synthesis results with 65 nm library

The synthesis with the Keccak accelerator produces an area and power overhead and a reduction of the maximum achievable frequency. This can be in turn compensated by the possibility of having a higher throughput: Keccak produces an output of 1600 bits every 24 clock cycles, whereas the TRNG produces a bit at each clock cycle. Therefore, by passing a key of width lower than (1600-24) bits, a higher throughput will be reached.
More detailed comparisons of the power and area contributions are displayed in Figure 6.2 and Figure 6.3.

Table 6.6 shows a hierarchical overview of the area occupied by the TRNG. The most expensive component in terms of resources is the one implementing the health tests, which is more than 2 times bigger than the noise source.

Cell	**Area [μm^2]**	**Percentage**
noise_src	983.880027	25.2%
CU	407.160011	10.4%
health_comp	2204.280038	**56.3**%
shift_reg	317.520009	8.1%
TRNG	3912.840085	100%

Table 6.6: TRNG resources utilization after Synopsys synthesis

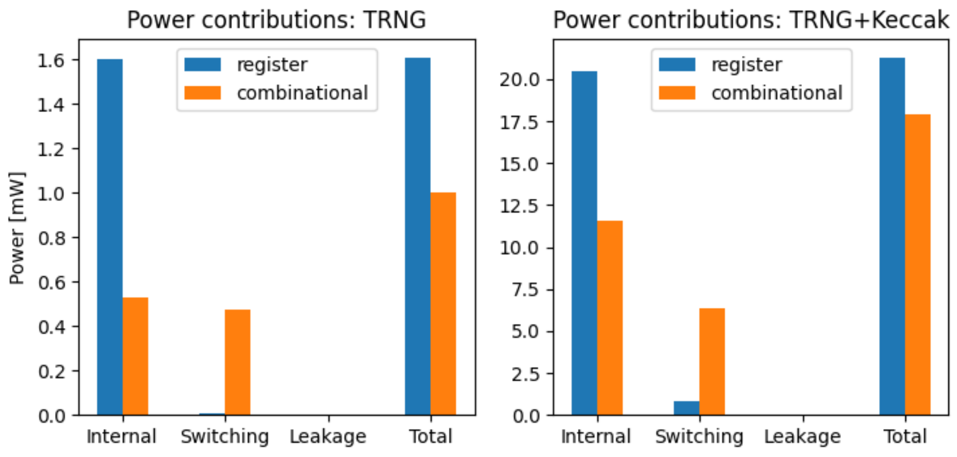


Figure 6.2: Power contributions of the TRNG with and without Keccak

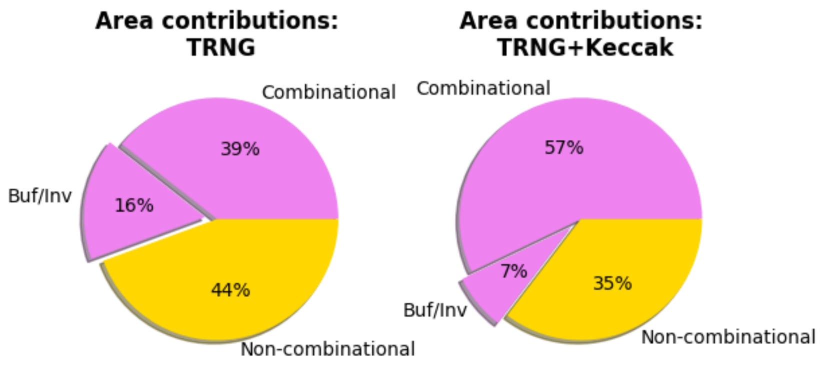


Figure 6.3: Area contributions of the TRNG and TRNG with Keccak

6.4 FPGA Results

To be able to compare the proposed TRNG to the ones presented in other papers, the FPGA synthesis and implementation have been performed. In particular, two Xilinx FPGA families have been considered, Artix and Spartan, as they are the most widely used in literature. The specific devices are the **Spartan 7 xc7s100fgga676-1Q** and the **Artix 7 xc7a75tcsg324-3**. By exploiting **Vivado**, the results of Table 6.7 have been obtained.

Device	**Frequency [MHz]**	**Area**			**Power [mW]**
		LUT	FF	Slices	
Artix 7	324.7	106	194	73	99
Spartan 7	230.3	98	194	79	104

Table 6.7: TRNG FPGA post-implementation results

The results of power and area are similar, whereas a slight difference is present in terms of frequency, the Artix 7 device reaching 100 MHz more than the Spartan 7.
Figure 6.4 reports a more detailed overview of the power, highlighting the predominance of static power.
Figure 6.5 displays the different area contributions for the two devices. Even though the most used resources are LUTs, only a minimal part of the total available area is used. The utilization of I/O ports is instead higher with respect to the available ones, reaching 18% in the case of the Artix board.

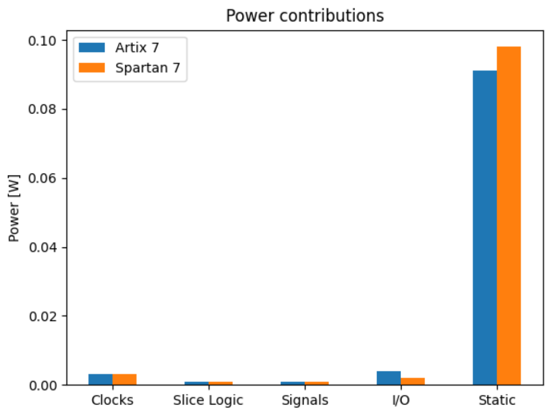


Figure 6.4: FPGA different power contributions

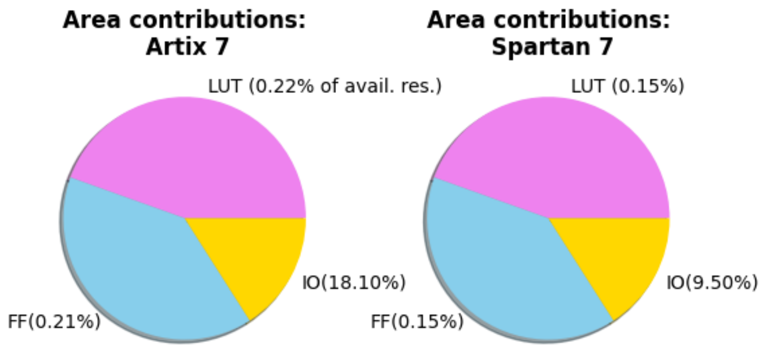


Figure 6.5: FPGA different area contributions, the percentage refers to the total available resources of the device

Table 6.8 shows the hierarchical distribution of the resources: also in this case, as in the 65 nm synthesis, the greater amount of area is reserved to the health tests component.

	Artix 7			**Spartan 7**		
Cell	**Area**			**Area**		
	LUT	FF	Slices	LUT	FF	Slices
noise_src	23	33	8	23	33	10
CU	35	22	16	29	22	15
health_comp	**48**	**132**	**46**	**46**	**132**	**50**
shift_reg	0	7	3	0	7	4
TRNG	106	194	73	98	194	79

Table 6.8: TRNG resources utilization after Synopsys synthesis

6.5 Comparison

The results obtained in this work have been compared to the ones presented in recent literature, the outcome is reported in Table 6.9. An effort has been made to collect common parameters to have a reliable comparison, nevertheless, some specifications are not always disclosed and the Table presents some missing points. For the same purpose, only the results regarding the TRNG, without the Keccak conditioning, are taken into consideration.

	Shann. Entr.	FPGA device	Area LUTs	FFs	Slices	Thrp. [Mbps]	Freq. [MHz]
[10]	0.9999	Artix7	24*	33*	13*	275.8	275.8
[11]	0.9999	Spartan3	528	177	270	6	24
[12]	-	Zynq7	65*	119*	5*	1600	100
[16]	0.9999	Stratix4	288	190	-	400	100
[33]	0.9993	Artix7	$40^{@}$	$29^{@}$	-	1.91	-
[34]	0.997	Spartan6	$16^{@}$	$11^{@}$	-	1.15	100
This Work	**0.9995**	**Artix7**	**106|23***	**194|33***	**73|8***	**324.7**	**324.7**
		Spartan7	**98|23***	**194|33***	**79|10***	**230.3**	**230.3**

Table 6.9: Comparison of different TRNG implementations (* no control logic nor health tests, $^{@}$ no health tests)

To have a valid comparison, **RO-based TRNGs** have been considered.
Z. Lu *et al* ([10]) proposes an architecture composed of only 4 parallel ROs. A higher level of randomness is then given by a multiphase sampler, i.e. another RO, sampling the 4-ring oscillators in different instants of times. The implementation proves to be very robust in terms of statistical properties, also providing low-power and low-area results. However, there is no mention of the control circuitry nor of the health tests.
In [11] programmable delay inverters are exploited to have a higher control on the period of each ring oscillator. To do so, some inputs of the LUTs, implementing the inverters on the FPGA, are used as delay tuning. The solution allows to have great control of the entropy level of the noise source but results in big requirements of the area. Moreover, the Von Neumann post-processing reduces the throughput of the system.
In [12] ROs of different lengths are additionally conditioned by means of Maximal Length Feedback Shift registers. As in this thesis, also in this case a Keccak post-processing block is exploited. To have a fair comparison among the different papers, only the results related to the TRNG without the Keccak block are considered.

Even though no information about the Shannon Entropy is provided, the non-linear logic function used as a pre-processing block allows to achieve high performances in terms of throughput.
P. Nannipieri *et al* ([16]) implement a TRNG based on Fibonacci-Galois ROs (FiGaRO), achieving an entropy per bit very close to 1. By using 4 parallel FiGaRO stages, a throughput of 400 Mbps is reached with an operating frequency of 100 MHz, at the expense of area.
In [33] a configurable TERO-based TRNG is proposed, along with an accurate statistical analysis of the architecture. No information on the used operating frequency is provided, however the resulting throughput is not higher than 2 Mbps. The entropy and area follow the state of art trend.
Mentens *et al* ([34]) present an architecture based on edge sampling, similarly to what is done in [10], with the addition of an accurate mathematical model. Moreover, an implementation of a bit extractor is included to increase randomness. The biggest point in favor of this work is the low area occupation.
From Table 6.9 it is possible to observe that this work follows the trend of the recent literature papers in terms of entropy. Regarding the area, the first result of each field is referred to the complete design, the second one only takes into account the noise source area. Comparing the respective cases of the table, the implementation results are lower than most of the papers. In addition, promising throughput results are also provided. Even though [12] and [16] present better results in this field, it is also true that their area is higher. This work presents a good trade-off between these parameters, with also a contained power consumption.
In summary, the proposed design manages to fulfill the original objective of this thesis, which is finding a TRNG implementation balancing both the entropy level and the PPA metrics.

6.6 Future Improvements

The proposed TRNG shows promising results, however there is still room for improvement.
First of all, the throughput and the latency of the circuit can be enhanced by implementing a **parallel output** for the noise source. An idea to be further investigated could be to sample each ring oscillator at different points along the chain, and then XORing together all the parallel samplings, as depicted in Figure 6.6. An accurate study of the resulting entropy should be carried out to understand if this solution is feasible.

On the other hand, different conditioning circuits can be compared to find the best trade-off in terms of area, frequency and power overhead.

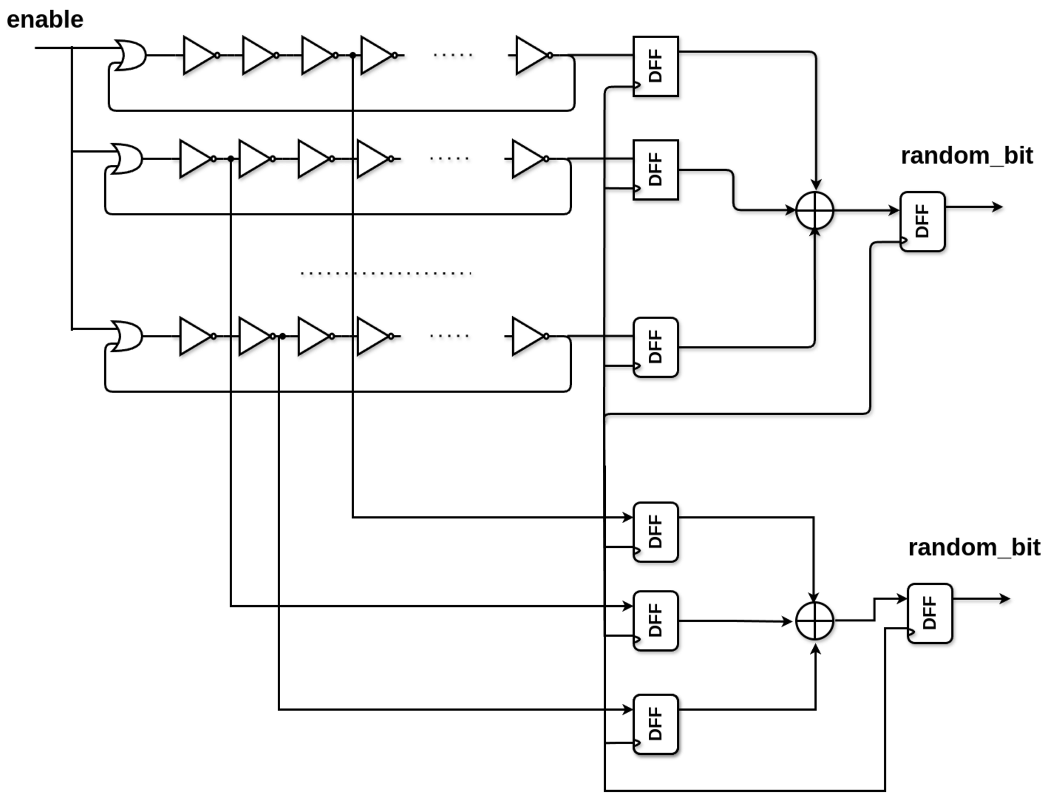


Figure 6.6: Sketch of the noise source schematic to have a parallel output

Another possibility is to implement **power-gating** and retention solutions to reduce power consumption. An idea could be to block the noise source through power gating when unused. At the same time, the actual advantage of this solution with respect to the time required for the warm-up phase at each power-on must be estimated.
Regarding the X-HEEP integration, the register file and output buffer of the TRNG could be partially merged to reduce the latency.
Finally, the proposed TRNG should be **physically tested** on a real FPGA device. A few tests have already been done on the Pynq-z2 board, but there is still a lot of work to do to be able to read the actual output bitstream and have a more realistic entropy estimate.

Chapter 7

Conclusion

The generation of robust unpredictable **random keys** is paramount in any cryptographic algorithm, even more, if considering the complexity of **PQC** ones.
This work presents a possible hardware implementation of a **true random number generator**, along with an efficient design methodology, with the aim of combining statistical properties and power-performance-area (PPA) metrics. The proposed solution passes all the tests of the NIST statistical test suite and presents a Shannon entropy per bit of 0.9995. Syntheses on two different Xilinx FPGA families have proven promising results in terms of area, power, and maximum frequency. An accelerator implementing the Keccak primitive can be used as an optional conditioning component, leading to a more unbiased datastream and a higher throughput.
The whole system has been integrated as an external accelerator in the RISC-V-based X-HEEP microcontroller. The hardware acceleration given by the TRNG has been applied to the PQClean C implementation of the CRYSTALKS-Kyber algorithm: the TRNG driver is more than 29 times faster than the original C function generating pseudo-random bytes.
Encouraging results have been obtained, but there is still room for improvement in terms of the trade-off between throughput, latency, and entropy.
The last years have seen deep research efforts in the field of Post-Quantum Cryptography, leading to novel algorithmic ideas and a focus on the development of dedicated cryptographic cores. There is still work to do, but this thesis can represent a valid starting point for a random number generator for an integrated cryptographic system.

Appendix A

Code snippets

A.1 Simulation scripts

A.1.1 Delay theoretical model

```python
import os.path
import numpy as np
import matplotlib.pyplot as plot

# Time unit : ps

# Parameters
n_RO = 33
n_delay_elem = 13
sigma = 30

# Deterministic jitter supposed = 0
Delta_dGD = 0

with open(os.path.join(os.getcwd(), 'model_13INV_33RO_30sigma.txt'),
                                                    'w') as fileID:
    for j in range(1, n_RO + 2):
        D_i = np.random.randint(275, 282)  # mean delay of a INV (275-281) ps
        # print delays of a RO for each line
        if j != 1:
            n_char = fileID.write(f'RO #{j-1} ')
            for print_var in range(len("RO #xxxx") - n_char):
                fileID.write('  ')
        for i in range(1, n_delay_elem + 1):
            if j == 1:
                if i == 1:
                    for _ in range(len("RO #xxxx")):
                        fileID.write(' ')
                fileID.write(f'I#{i} ')
            else:
                Delta_dLG = np.random.normal(0, sigma)
                # print delay of INV for each column
                fileID.write(f'{int(D_i + Delta_dLG + Delta_dGD)} ')


        fileID.write('\n')
```

The parameters and the name of the files are varied by means of a bash script to have a quick automatic procedure.

A.1.2 assign_delays task

```
task assign_delays;
    output int unsigned delays_vec[N_STAGES][RO_LENGTH];

    string line;
    static string fixed_chars = "RO #xxxx";
    int fID_delays;

    fID_delays = $fopen ("./model_files/model_13INV_33RO_30sigma.txt", "r");
    $fgets(line, fID_delays);

    for(int j = 0; j < N_STAGES; j++) begin
        $fscanf(fID_delays, "%s %s ", line, line);
        for(int i = 0; i < RO_LENGTH; i++) begin
            $fscanf(fID_delays, "%d ", delays_vec[j][i]);
        end
    end

    $fclose(fID_delays);

```

```
endtask
```

A.2 Drivers

A.2.1 `get_rnd_key(uint32_t* Dout)`

```
void get_rnd_key(uint32_t* Dout)
{
    uint32_t volatile *ctrl_reg = (uint32_t*) TRNG_CTRL_START_ADDR;
    uint32_t volatile *Dout_reg = (uint32_t*) TRNG_DOUT_START_ADDR;
    uint32_t volatile *status_reg = (uint32_t*) TRNG_STATUS_START_ADDR;
    uint8_t volatile key_ready;
    // Performance regs variable
    unsigned int volatile cycles = 0;

    // Starting the performance counter
    CSR_CLEAR_BITS(CSR_REG_MCOUNTINHIBIT, 0x1);
    CSR_WRITE(CSR_REG_MCYCLE, 0);

    // trigger
    asm volatile ("": : : "memory");
    *ctrl_reg = 0 << TRNG_CTRL_CTRL_ACK_KEY_READ_BIT;
    asm volatile ("": : : "memory");
    *ctrl_reg = 1 << TRNG_CTRL_CTRL_TRNG_EN_BIT;
    asm volatile ("": : : "memory");
    *ctrl_reg = 0 << TRNG_CTRL_CTRL_TRNG_EN_BIT;

    // poll
    do {
        key_ready = (*status_reg) & (1 << TRNG_CTRL_STATUS_TRNG_BIT);
    } while (key_ready == 0);

    // get key
    *Dout = Dout_reg[0];

    // acknowledge key
    *ctrl_reg = 1 << TRNG_CTRL_CTRL_ACK_KEY_READ_BIT;
    asm volatile ("": : : "memory");
    *ctrl_reg = 0 << TRNG_CTRL_CTRL_ACK_KEY_READ_BIT;

    // stop the HW counter used for monitoring
    CSR_READ(CSR_REG_MCYCLE, &cycles);
    printf("\nNumber of clock cycles to generate the key - polling:
    %d\n", cycles);
}

```

A.2.2 get_rnd_key_intr(uint32_t* Dout)

```
void get_rnd_key_intr(uint32_t* Dout)
{
    uint32_t volatile *ctrl_reg = (uint32_t*) TRNG_CTRL_START_ADDR;
    uint32_t volatile *Dout_reg = (uint32_t*) TRNG_DOUT_START_ADDR;
    uint32_t volatile *status_reg = (uint32_t*) TRNG_STATUS_START_ADDR;
    uint8_t volatile key_ready;
    // Performance regs variable
    unsigned int volatile cycles = 0;

    // Interrupt
    plic_Init();
    plic_irq_set_priority(EXT_INTR_0, 1);
    plic_irq_set_enabled(EXT_INTR_0, kPlicToggleEnabled);

    CSR_CLEAR_BITS(CSR_REG_MCOUNTINHIBIT, 0x1);
    CSR_WRITE(CSR_REG_MCYCLE, 0);

    CSR_SET_BITS(CSR_REG_MSTATUS, 0x8);
    // Set mie.MEIE bit to one to enable machine-level external interrupts
    const uint32_t mask = 1 << 11;//IRQ_EXT_ENABLE_OFFSET;
    CSR_SET_BITS(CSR_REG_MIE, mask);

    // trigger
    asm volatile ("": : : "memory");
    *ctrl_reg = 0 << TRNG_CTRL_CTRL_ACK_KEY_READ_BIT;
    asm volatile ("": : : "memory");
    *ctrl_reg = 1 << TRNG_CTRL_CTRL_TRNG_EN_BIT;
    asm volatile ("": : : "memory");
    *ctrl_reg = 0 << TRNG_CTRL_CTRL_TRNG_EN_BIT;

    while(plic_intr_flag==0) {
        wait_for_interrupt();
    }
    // get key
    *Dout = Dout_reg[0];

    // acknowledge key
    *ctrl_reg = 1 << TRNG_CTRL_CTRL_ACK_KEY_READ_BIT;

    // stop the HW counter used for monitoring
    CSR_READ(CSR_REG_MCYCLE, &cycles);
    printf("\nNumber of clock cycles to generate the key - interrupt:
    %d\n", cycles);
}
```

A.2.3 get_rnd_bytes(size_t nbytes, uint8_t *Dout)

```
void get_rnd_bytes(size_t nbytes, uint8_t *Dout)
{
    uint32_t volatile *ctrl_reg = (uint32_t*) TRNG_CTRL_START_ADDR;
    uint32_t volatile *Dout_reg = (uint32_t*) TRNG_DOUT_START_ADDR;
    uint32_t volatile *status_reg = (uint32_t*) TRNG_STATUS_START_ADDR;
    uint8_t volatile key_ready;

    // trigger
    asm volatile ("": : : "memory");
    *ctrl_reg = 0 << TRNG_CTRL_CTRL_ACK_KEY_READ_BIT;
    asm volatile ("": : : "memory");
    *ctrl_reg = 1 << TRNG_CTRL_CTRL_TRNG_EN_BIT;
    asm volatile ("": : : "memory");
    *ctrl_reg = 0 << TRNG_CTRL_CTRL_TRNG_EN_BIT;

    uint32_t mask = 0x000000FF;
    for(int i = 0; i < nbytes; i=i+4){
        asm volatile ("": : : "memory");
        *ctrl_reg = 0 << TRNG_CTRL_CTRL_ACK_KEY_READ_BIT;
        asm volatile ("": : : "memory");
        do {
            key_ready = (*status_reg) & (1 << TRNG_CTRL_STATUS_TRNG_BIT);
        } while (key_ready == 0);

        for(int j = 0; j < 4; j++)
         Dout[i+j] = (Dout_reg[0] >> (j<<3)) & mask;
        *ctrl_reg = 1 << TRNG_CTRL_CTRL_ACK_KEY_READ_BIT;
    }

}
```

Bibliography

[1] Elaine Barker (NIST). *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. Tech. rep. Gaithersburg, MD, USA, 2020. DOI: https://doi.org/10.6028/NIST.SP.800-175Br1 (cit. on p. 1).

[2] Charles Herder, Meng-Day Yu, Farinaz Koushanfar, and Srinivas Devadas. «Physical Unclonable Functions and Applications: A Tutorial». In: *Proceedings of the IEEE* 102.8 (2014), pp. 1126–1141. DOI: 10.1109/JPROC.2014.2320516 (cit. on p. 2).

[3] Rivaldo Ludovicus Sembiring, Rizka Reza Pahlevi, and Parman Sukarno. «Randomness, Uniqueness, and Steadiness Evaluation of Physical Unclonable Functions». In: *2021 9th International Conference on Information and Communication Technology (ICoICT)*. 2021, pp. 429–433. DOI: 10.1109/ICoICT52021.2021.9527493 (cit. on p. 3).

[4] Than Naing So Aye Myat Nyo. *A New Approach to LFSR based Pseudorandom Numbers Generator*. URL: https://meral.edu.mm/records/7332 (cit. on p. 4).

[5] Analog Devices. *Pseudo Random Number Generation Using Linear Feedback Shift Registers*. URL: https://www.analog.com/en/resources/design-notes/random-number-generation-using-lfsr.html (cit. on p. 4).

[6] Markku-Juhani O. Saarinen. *On Entropy and Bit Patterns of Ring Oscillator Jitter*. Cryptology ePrint Archive, Paper 2021/1363. https://eprint.iacr.org/2021/1363. 2021. URL: https://eprint.iacr.org/2021/1363 (cit. on p. 7).

[7] Boyan Valtchanov, Alain Aubert, Florent Bernard, and Viktor Fischer. «Modeling and observing the jitter in ring oscillators implemented in FPGAs». In: *2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*. 2008, pp. 1–6. DOI: 10.1109/DDECS.2008.4538777 (cit. on p. 7).

[8] Nathalie Bochard, Florent Bernard, Viktor Fischer, and Boyan Valtchanov. «True-Randomness and Pseudo-Randomness in Ring Oscillator-Based True Random Number Generators». In: *International Journal of Reconfigurable Computing* 2010 (Jan. 2010). DOI: `10.1155/2010/879281` (cit. on pp. 7, 33, 34).

[9] A. Theodore Markettos and Simon W. Moore. «The Frequency Injection Attack on Ring-Oscillator-Based True Random Number Generators». In: *Cryptographic Hardware and Embedded Systems - CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 317–331. ISBN: 978-3-642-04138-9 (cit. on p. 8).

[10] Zhaojun Lu, Houjia Qidiao, Qidong Chen, Zhenglin Liu, and Jiliang Zhang. «An FPGA-Compatible TRNG with Ultra-High Throughput and Energy Efficiency». In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 2023, pp. 1–6. DOI: `10.1109/DAC56929.2023.10247746` (cit. on pp. 8, 63, 64).

[11] N. Nalla Anandakumar, Somitra Kumar Sanadhya, and Mohammad S. Hashmi. «FPGA-Based True Random Number Generation Using Programmable Delays in Oscillator-Rings». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.3 (2020), pp. 570–574. DOI: `10.1109/TCSII.2019.2919891` (cit. on pp. 8, 63).

[12] Annapurna Kamadi and Zia Abbas. «Implementation of TRNG with SHA-3 for hardware security». In: *Microelectronics Journal* 123 (2022), p. 105410. ISSN: 0026-2692. DOI: `https://doi.org/10.1016/j.mejo.2022.105410` (cit. on pp. 8, 63, 64).

[13] Michal Varchola and Milos Drutarovsky. «New High Entropy Element for FPGA Based True Random Number Generators». In: *Cryptographic Hardware and Embedded Systems, CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 351–365. ISBN: 978-3-642-15031-9 (cit. on p. 9).

[14] Ihor Vasyltsov, Eduard Hambardzumyan, Young-Sik Kim, and Bohdan Karpinskyy. «Fast Digital TRNG Based on Metastable Ring Oscillator». In: *Cryptographic Hardware and Embedded Systems – CHES 2008*. Ed. by Elisabeth Oswald and Pankaj Rohatgi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 164–180. ISBN: 978-3-540-85053-3 (cit. on p. 9).

[15] J.D.J. Golic. «New Methods for Digital Generation and Postprocessing of Random Data». In: *IEEE Transactions on Computers* 55.10 (2006), pp. 1217–1229. DOI: `10.1109/TC.2006.164` (cit. on pp. 10, 11).

[16] Pietro Nannipieri, Stefano Di Matteo, Luca Baldanzi, Luca Crocetti, Jacopo Belli, Luca Fanucci, and Sergio Saponara. «True Random Number Generator Based on Fibonacci-Galois Ring Oscillators for FPGA». In: *Applied Sciences* 11.8 (2021). ISSN: 2076-3417. DOI: 10.3390/app11083330. URL: https://www.mdpi.com/2076-3417/11/8/3330 (cit. on pp. 11, 63, 64).

[17] Guido Di Patrizio Stanchieri, Andrea De Marcellis, Marco Faccio, and Elia Palange. «An FPGA-Based Architecture of True Random Number Generator for Network Security Applications». In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018, pp. 1–4. DOI: 10.1109/ISCAS.2018.8351376 (cit. on p. 12).

[18] Viktor Fischer and Miloš Drutarovský. «True Random Number Generator Embedded in Reconfigurable Hardware». In: vol. 2523. Aug. 2002, pp. 415–430. ISBN: 978-3-540-00409-7. DOI: 10.1007/3-540-36400-5_30 (cit. on p. 12).

[19] Viktor Fischer, Florent Bernard, Nathalie Bochard, Quentin Dallison, and Maciej Skórski. «Enhancing Quality and Security of the PLL-TRNG». In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.4 (Aug. 2023), pp. 211–237. DOI: 10.46586/tches.v2023.i4.211-237. URL: https://tches.iacr.org/index.php/TCHES/article/view/11164 (cit. on p. 13).

[20] Anju P. Johnson, Rajat Subhra Chakraborty, and Debdeep Mukhopadyay. «An Improved DCM-Based Tunable True Random Number Generator for Xilinx FPGA». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 64.4 (2017), pp. 452–456. DOI: 10.1109/TCSII.2016.2566262 (cit. on p. 13).

[21] Fabio Frustaci, Fanny Spagnolo, Stefania Perri, and Pasquale Corsonello. «A High-Speed FPGA-Based True Random Number Generator Using Metastability With Clock Managers». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 70.2 (2023), pp. 756–760. DOI: 10.1109/TCSII.2022.3211278 (cit. on p. 14).

[22] Je Sen Teh, WeiJian Teng, and Azman Samsudin. «A true random number generator based on hyperchaos and digital sound». In: *2016 3rd International Conference on Computer and Information Sciences (ICCOINS)*. 2016, pp. 264–269. DOI: 10.1109/ICCOINS.2016.7783225 (cit. on p. 14).

[23] Ahmad Beirami and Hamid Nejati. «A Framework for Investigating the Performance of Chaotic-Map Truly Random Number Generators». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 60.7 (2013), pp. 446–450. DOI: 10.1109/TCSII.2013.2258274 (cit. on p. 14).

[24] Chua-Chin Wang, Yih-Long Tseng, Hon-Chen Cheng, and Ron Hu. «Switched-current 3-bit CMOS wideband random signal generator». In: *Southwest Symposium on Mixed-Signal Design, 2003.* 2003, pp. 186–189. DOI: 10.1109/SSMSD.2003.1190424 (cit. on p. 14).

[25] Meltem S. Turan, Elaine Barker, John Kelsey, Kerry McKay, Mary Baish, and Michael Boyle. *Recommendation for the Entropy Sources Used for Random Bit Generation.* Tech. rep. Gaithersburg, MD, USA, 2018. DOI: https://doi.org/10.6028/NIST.SP.800-90B (cit. on pp. 16, 17, 19, 23, 26, 34).

[26] Lawrence E. Bassham et al. *SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* Tech. rep. Gaithersburg, MD, USA, 2010 (cit. on pp. 16, 23).

[27] Elaine Barker, John Kelsey, Kerry McKay, Allen Roginksy, and Meltem S. Turan. *Recommendation for Random Bit Generator (RBG) Constructions.* Tech. rep. Gaithersburg, MD, USA, 2022. DOI: https://doi.org/10.6028/NIST.SP.800-90C.3pd (cit. on p. 16).

[28] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. «Keccak». In: *Advances in Cryptology – EUROCRYPT 2013.* Ed. by Thomas Johansson and Phong Q. Nguyen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 313–314. ISBN: 978-3-642-38348-9 (cit. on pp. 20, 22).

[29] G.M. Bertoni, Joan Daemen, Michael Peeters, and Gilles Assche. «Sponge Functions». In: *ECRYPT Hash Workshop 2007* (Jan. 2007) (cit. on p. 20).

[30] Alessandra Dolmeta, Mattia Mirigaldi, Maurizio Martina, and Guido Masera. «Implementation and integration of Keccak accelerator on RISC-V for CRYSTALS-Kyber». In: *Proceedings of the 20th ACM International Conference on Computing Frontiers.* CF '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 381–382. ISBN: 9798400701405. DOI: 10.1145/3587135.3591432. URL: https://doi.org/10.1145/3587135.3591432 (cit. on p. 38).

[31] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. *X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators.* 2024. arXiv: 2401.05548 [cs.AR] (cit. on p. 42).

[32] Roberto Avanzi et al. *CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation.* 2017 (cit. on p. 57).

[33] Naoki Fujieda. «On the Feasibility of TERO-Based True Random Number Generator on Xilinx FPGAs». In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL).* 2020, pp. 103–108. DOI: 10.1109/FPL50879.2020.00027 (cit. on pp. 63, 64).

[34] N. Mentens B. Yang. «ES-TRNG: A high-throughput, low-area true random number generator based on edge sampling». In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2018) (cit. on pp. 63, 64).