



**Politecnico
di Torino**

Politecnico di Torino

Master's Degree in Electronic Engineering

A.y. 2025/2026

Graduation Session July 2026

Design and FPGA implementation of an ASCON-based true random number generator for cryptography

Relator:

Maurizio Zamboni

Candidate:

Andrea Faccio

Abstract

The confidentiality and security of information in a wide range of fields depend critically on the protection of sensitive data through cryptography. This requirement has become even more pressing after quantum computers were introduced, as their substantially increased computational strength compared to classical computers allows the breaking of numerous widely-used cryptographic protocols. A True Random Number Generator (TRNG) serves as a fundamental device for generating random number sequences essential for cryptographic encryption and entropy production. A high-quality random sequence determines the security strength of cryptographic systems, as insufficient randomness makes keys predictable and weakens encryption schemes.

While pseudo-random number generators (PRNGs) rely on deterministic algorithms producing reproducible sequences from a given initial state, TRNGs harness the inherent randomness of physical phenomena such as thermal noise, power supply fluctuations, and metastability in electronic circuits to generate truly unpredictable random samples. The National Institute of Standards and Technology (NIST) has established the SP 800-22 test suite as the primary benchmark for evaluating entropy source quality. Passing these tests certifies that a TRNG meets rigorous cryptographic requirements.

Furthermore, post-processing and conditioning play a crucial role in enhancing TRNG output quality. Raw entropy often contains biases or correlations corrected through conditioning algorithms such as cryptographic hash functions, transforming biased raw entropy into statistically clean, uniformly distributed random bits. This work proposes a ring oscillator-based TRNG design for performance evaluation against existing devices, evaluating throughput, speed, and resource utilization. To investigate post-processing impact, an ASCON-based hashing block was designed and integrated, enabling comparative analysis with data files generated through diverse methods.

An automated testing environment was developed to streamline validation and archive data. The system was implemented on a Cyclone 10 GX FPGA using VHDL. The implemented solution achieves compliance with all NIST statistical test suite requirements while exhibiting favorable entropy, area utilization, and throughput performance. These outcomes constitute a promising foundation for an integrated RBG for cryptographic applications. A notable application involves embedding

the TRNG within an HSM (Hardware Security Module) for secure cryptographic key generation.

Keywords: Cryptography, True Random Number Generators, Entropy, TRNG Hardware Design, TRNG FPGA integration, ASCON algorithm, Hash functions

Table of Contents

List of Tables	IV
List of Figures	V
1 Introduction	1
1.1 Entropy generation in cryptography	2
1.1.1 Pseudo Random Number Generators (PRNGs)	3
1.1.2 Physical Unclonable Functions (PUFs)	4
1.2 Struttura della tesi	5
2 True Random Number Generators in Cryptographic Systems	7
2.1 TRNG Architectural Model	8
2.1.1 Entropy source classification	9
2.1.2 Conditioning stage	10
2.2 Existing approaches to TRNG devices	11
2.2.1 Ring Oscillator-based TRNG	11
2.2.2 PLL-based TRNG	16
2.2.3 DCM-based TRNG	18
3 Statistical Validation and Post-Processing	21
3.1 Conditioning block	23
3.1.1 ASCON-based Conditioning Model	24
3.2 Health Tests	27
3.3 NIST test suites	28
3.3.1 NIST Statistical Test Suite	28
3.3.2 NIST SP 800-90B suite	30
4 Complete TRNG System Design and FPGA Implementation	33
4.1 TRNG Core and Input Memory Interface	34
4.1.1 Noise Source (TRNG core)	35
4.1.2 Interface component	37

4.1.3	Interface Control Unit	38
4.2	ASCON Conditioning Block	39
4.2.1	ASCON Core Block	40
4.2.2	Conditioning Control Unit	41
4.3	Complete System Synchronization	42
4.3.1	Synchronizer FSM Controller	44
4.3.2	Memory Architecture and Design Considerations	45
5	Testing Environment for Data Generation and Statistical Analysis	47
5.1	Entropy Sources Used	49
5.2	Hash Functions Employed	51
5.3	Experimental Workflow	53
6	Results Analysis and Comparison	56
6.1	NIST Suite Test Results	56
6.2	FPGA Performance Analysis	60
6.2.1	Device Characterization	60
6.2.2	Comparative Tables	63
6.3	Possible Improvements and Future Developments	65
7	Conclusions	67

List of Tables

2.1	State evolution of the circuit varying the inputs.	13
3.1	Overview of the tests of the NIST Statistical Test Suite.	29
5.1	Summary of the main SP800-22 results for the <code>/dev/urandom</code> file. .	54
6.1	Parameters chosen for NIST test suite.	57
6.2	NIST test suite results for the HW device.	57
6.3	NIST test suite results for the bitsream generated, conditioning is software hash function SHA-3.	58
6.4	NIST test suite results for the bitsream generated, conditioning is software hash function ASCON.	58
6.5	Performance comparison of the implemented architecture's blocks. .	61
6.6	Device performance parameters for the TRNG plus ASCON complete block architecture.	61
6.7	Area utilization of the standalone TRNG block.	62
6.8	Area utilization of the complete TRNG-ASCON system.	62
6.9	Comparison of different TRNG implementations.	63

List of Figures

1.1	Simple scheme of a 4-bit LFSR vhdl implementation for reference. .	3
2.1	Generic TRNG architecture showing the separation between raw entropy generation, post-processing, and output validation.	8
2.2	Block scheme of a RO with N=3.	11
2.3	Key principle of a RO-based true random number generator.	12
2.4	Scheme of a TeRO true random number generator device.	13
2.5	Scheme of a MeRO true random number generator device.	14
2.6	FiRO trng implementation scheme.	15
2.7	GaRO trng implementation scheme.	15
2.8	FiGaRO trng implementation scheme.	16
2.9	Complete chain of a FiGaRO device.	16
2.10	PLL block scheme.	17
2.11	A possible implementation of a PLL-based TRNG.	17
2.12	Frequency tuning DCM trng scheme.	18
2.13	DSP-based DCM trng scheme.	19
3.1	Block scheme of the system that generates the entropy (TRNG device). .	22
3.2	General scheme of the permutation where the internal state is presented and the logical expressions used in the layers are depicted. .	24
3.3	Hashing mode phases for ASCON-Hash 256 bit.	25
4.1	Interface TRNG block scheme. Clock and asynchronous active-low reset signals were omitted.	34
4.2	Scheme of a Muxed RO-based true random number generator.	35
4.3	Interface block scheme. Clock and reset signals are omitted.	37
4.4	FSM Control for the interface TRNG-Memory source.	38
4.5	Block scheme of the ASCON conditioning implemented component.	39
4.6	FSM representation for the conditioning process.	40
4.7	FSM representation for the controller for the ASCON algorithm	41
4.8	Complete system block scheme. Only significant signals are depicted.	42

4.9	FSM representation for the complete system synchronization.	44
5.1	Conceptual scheme of the environment of test.	48
5.2	First part of the NIST 15-test report for the <code>/dev/urandom</code> file. . .	54
5.3	First part of the frequency report for the <code>/dev/urandom</code> file.	54
6.1	Proportions of sequences passing the 15 NIST tests without ASCON conditioning (top) and after ASCON conditioning (bottom).	59

Chapter 1

Introduction

Cryptography plays a crucial role in the protection of information in modern digital systems where integrity and confidentiality are fundamental requirements for secure communication and data storage. The effectiveness of cryptographic mechanisms, does not depend only on the strength of the adopted algorithms, but also on the quality of the random values employed during key generation, initialization, authentication, and other procedures that can affect the behaviour of the complete system. Randomness is therefore an essential resource in the design of secure systems, as weak or predictable values may compromise the overall robustness of an otherwise well-designed cryptographic scheme. This is especially relevant in hardware oriented security applications, where the reliability of the physical implementation directly affects the quality of the generated data and the final purpose of the design.

In this matter True Random Number Generators (TRNGs) represent a fundamental building block, since they exploit physical sources of entropy to generate highly unpredictable bit sequences suitable for cryptographic use. In contrast to pseudo-random generators, which are deterministic, TRNGs provide non-deterministic outputs derived from physical phenomena and therefore they guarantee strong security in lots of applications.

An important aspect to consider is that classical cryptographic algorithms are based on hard mathematical problems, such as integer factorization or the discrete logarithm problem. For many years, the computational difficulty of these problems was considered sufficient enough to ensure secure communication and data protection. This assumption has been challenged by the emergence of quantum computing, that thanks to its increased computing capacity and due to its intrinsic functioning principle has made several conventional cryptographic schemes potentially vulnerable. Although large-scale quantum computers are not yet widely available, their future impact has motivated the development of post-quantum

cryptography (PQC), which aims at designing algorithms resistant to quantum attacks.

In any cryptographic system, the generation of secret keys remains a fundamental step. The security of the entire scheme depends not only on the algorithm itself, but also on the unpredictability of the generated key. For this reason, entropy extraction and post-processing techniques such as whitening play an important role in improving the quality and uniformity of random data before it is used in cryptographic applications.

1.1 Entropy generation in cryptography

In order to obtain a robust cryptographic scheme it is needed to guarantee some statistical properties to the data generated and it will be required to be able to quantify the levels of randomness of the keys and sequences generated. Therefore several methods and devices are used to fulfill this goal, the most used are:

- **Pseudo Random Number Generators (PRNGs)**: an algorithmic generator that produces sequences of numbers that appear random, starting from a seed value. The output is deterministic, so the same seed always produces the same sequence in output.
- **True Random Number Generators (TRNGs)**: a generator that derives randomness from physical, non-deterministic phenomena, such as noise or jitter. Its output is not predictable in the same way as a PRNG.
- **Physical Unclonable Functions (PUFs)**: hardware primitives that exploit unavoidable manufacturing variations to produce device-specific and hard-to-replicate responses. They are mainly used for identification, authentication, and secure key-related applications.

Overall, PRNGs, TRNGs, and PUFs represent three important concepts in the design of secure digital systems, the suitability of these blocks depend on the application's constraints required. PRNGs provide efficiency and scalability, TRNGs provide physical entropy, and PUFs provide device-specific uniqueness. Even without entering into implementation details, understanding their roles and differences is essential for introducing hardware-oriented cryptographic architectures and secure random generation mechanisms.

Before focusing in detail on the possible TRNG implementations that will lead to the main objective of this work, it is useful to briefly review the characteristics of the other two methods introduced above.

1.1.1 Pseudo Random Number Generators (PRNGs)

Pseudorandom Number Generators (PRNGs) are deterministic algorithms designed to generate sequences of bits or numbers that exhibit statistical properties similar to those of random sequences. Their behavior depends on an initial seed or internal state, from which the subsequent values are recursively derived. As a result, the output is reproducible and not truly random in a physical sense, although it may be sufficiently random-looking for many practical applications.

A classical and intuitive example is the Linear Feedback Shift Register (LFSR). An LFSR is composed of a shift register whose input bit is obtained through a linear feedback function, typically implemented by means of exclusive-OR operations over selected register taps. At each clock cycle, the register shifts its contents and generates a new output bit. This architecture is particularly simple, compact, and efficient in hardware, which makes it well suited for educational purposes and for non-cryptographic applications where low area and high speed are important.

Overall PRNGs are state-based deterministic devices that are lightweight and efficient in implementation that can expand a short initial state into a long output sequence, with several statistical and entropic limitations.

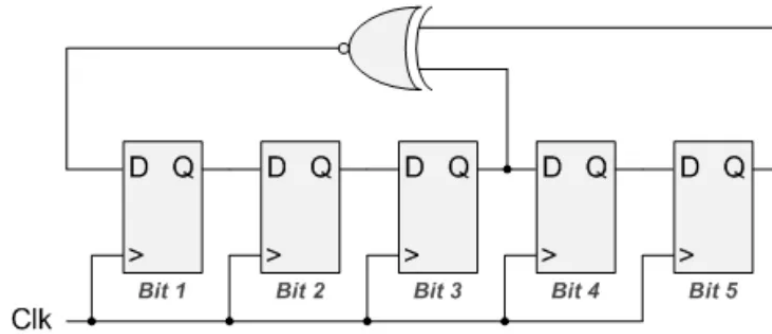


Figure 1.1: Simple scheme of a 4-bit LFSR vhd implementation for reference.

However, the linear structure of an LFSR also exposes its main limitation. Since the generator follows a predictable recurrence relation, observing a sufficient amount of output can allow an attacker to reconstruct the internal state and predict future values. For this reason, basic LFSR-based PRNGs are generally unsuitable for cryptographic use unless they are combined with additional mechanisms that improve their unpredictability.

In this sense, LFSR-based generators provide a useful introduction to the concept of pseudorandomness: they show how a small internal state can be expanded

into a longer output sequence that appears random, while remaining fully deterministic. This distinction is essential for understanding why cryptographic applications require stronger randomness sources, such as TRNGs or more robust cryptographically secure generators.

1.1.2 Physical Unclonable Functions (PUFs)

PUFs are hardware security primitives that exploit the intrinsic and unpredictable manufacturing variations of electronic devices to generate unique responses. Since even nominally identical chips exhibit small physical differences, these variations can be used as a hardware fingerprint for identification, authentication, and secure key-related applications.

From a functional point of view, a PUF maps an input challenge to an output response that depends on the physical properties of the specific device. An effective PUF should provide responses that are unique across devices, reliable under repeated measurements, and difficult to predict or reproduce. In practice, however, its behavior can be affected by environmental conditions, noise, and aging effects, which makes reliability an important design concern.

PUFs are commonly classified as weak or strong.

- Weak PUFs provide a limited number of challenge-response pairs and are mainly used for key generation, key storage, or device binding.
- Strong PUFs, instead, support a much larger challenge space and are therefore more suitable for authentication protocols.

The main advantage of PUFs lies in the possibility of deriving security features directly from the hardware without storing fixed secrets in memory, making devices with PUFs implemented pretty robust against external cryptographic attacks. On the other hand, their responses may require stabilization or error-correction techniques, and some PUF architectures can be vulnerable to modeling or physical attacks.

Therefore, PUFs are a promising security primitive, but their practical adoption requires careful design and evaluation of many factors. They are mostly used for authentication purposes because they provide critical advantages with respect to the classical approach.

1.2 Thesis Structure

The aim of this thesis was to develop a complete TRNG device system, starting from a known entropy source, implemented on an FPGA development board, and subsequently integrating the hardware implementation of an ASCON-based conditioning algorithm into the system. The main objective was to study the performance of the device in relation to the possible alternatives currently available in the state of the art, to quantify the impact of the conditioning block, and to verify the statistical validity of the generated data according to the directives and standards established by NIST.

After the first chapter the thesis is structured as follows:

- **chapter 2** introduces the theoretical background required to understand the context of the work, with particular attention to the main concepts related to random number generation, entropy sources, and statistical validation.
- **chapter 3** introduces the statistical validation framework adopted in this thesis, with particular emphasis on the NIST test suite and the additional SP800-22 tests used to evaluate the randomness properties of the generated data. The chapter also outlines the criteria used to interpret the test outcomes and the role of these results in the overall experimental analysis.
- **chapter 4** describes the hardware and software architecture developed for the thesis, including the design choices adopted for the implementation of the conditioning device and the overall experimental environment built on FPGA. Every component block of the device and its function are described in detail.
- **chapter 5** presents the environment created for the generation of the test datasets and the experimental workflow used to prepare the input files, apply the optional conditioning stage, and organize the validation process to validate the device.
- **chapter 6** reports the final results in terms of entropy, statistical properties, area and performance, considering the FPGA implementations. It also includes a comparison with the state-of-the-art results, together with a summary of possible improvements for further development of the proposed TRNG-Conditioning created device.
- **chapter 7** presents the conclusions of the thesis, with final considerations regarding the work carried out.

Chapter 2

True Random Number Generators in Cryptographic Systems

As previously stated True Random Number Generators (TRNGs) are a fundamental component of modern cryptographic systems, since the security of many operations depends on the availability of unpredictable random data. Secret keys, nonces, initialization vectors, and other security-critical parameters require randomness of sufficiently high quality to prevent predictability and reduce the risk of attacks deriving from weak entropy sources.

Unlike pseudorandom number generators, which produce deterministic outputs starting from an initial seed, TRNGs derive randomness from physical phenomena that exhibit intrinsic unpredictability. Their function is therefore closely related to the quality of the entropy source and to the ability of the system to extract usable random data from it. In practical implementations, the raw output of a physical source is often affected by bias, correlation, and environmental sensitivity, which may reduce its direct suitability for cryptographic use.

For this reason, TRNGs are usually described as multi-stage systems including an **entropy source**, a **digitization or extraction stage**, a **conditioning block**, and suitable **validation mechanisms**. Conditioning techniques, including whitening transformations and cryptographic post-processing functions, are commonly employed to improve the statistical quality of the generated sequence, even though they do not create additional entropy by themselves. The effectiveness of a TRNG must therefore be assessed by considering both the physical origin of randomness and the observable behavior of the final bitstream.

This chapter presents the main classes of entropy sources, discusses the role of conditioning and randomness extraction, and provides a brief overview of the most

used and significant TRNG implementations, classified with respect to the source of randomness they exploit. These elements provide the theoretical background needed to frame the implementation choices and evaluation methodology adopted in the present work.

2.1 TRNG Architectural Model

A True Random Number Generator (TRNG) is generally organized as a pipeline of functional blocks rather than as a single monolithic component. Moreover the generator itself is often used as a functional block for a bigger system such as an HSM (Hardware Security Module). At the core of the architecture lies a physical entropy source, which exploits a non-deterministic phenomenon such as jitter, metastability, thermal noise, or phase variation in an oscillator. The raw output generated by this source is typically not suitable for direct cryptographic use, because it may contain bias, temporal correlation, or sensitivity to external operating conditions. For this reason, the architecture of a TRNG must be understood as a complete chain that converts physical generated randomness into a stable and usable bitstream.

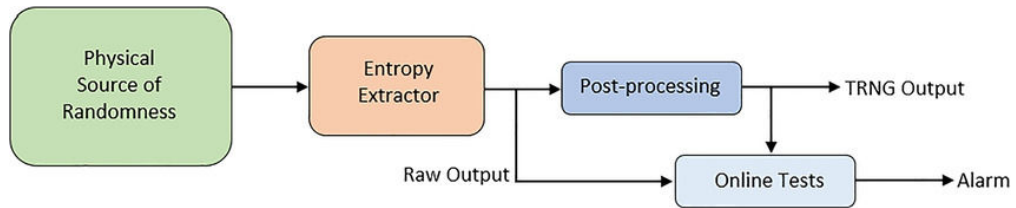


Figure 2.1: Generic TRNG architecture showing the separation between raw entropy generation, post-processing, and output validation.

The first stage of the architecture is the **entropy source** itself, which produces the raw physical behavior from which randomness is extracted. In many hardware implementations, this source generates an analog or quasi-analog signal that must then be sampled and digitized. The **sampling stage** transforms the physical phenomenon into raw bits, but these bits often do not yet satisfy the statistical properties expected from a cryptographic random source. As a result, a second stage is usually required to reduce imperfections and improve the quality of the output.

This second stage is known as **conditioning** or **whitening**. Its purpose is to reduce bias, decorrelate the raw sequence, and produce an output that is more uniform and more suitable for further cryptographic processing. Conditioning does not create entropy; instead, it reorganizes and concentrates the entropy already present in the raw source.

Depending on the implementation, this step may rely on simple combinational logic, bit mixing, cryptographic hash functions, or other post-processing mechanisms.

In addition to entropy extraction and conditioning, a complete TRNG architecture must include monitoring and validation mechanisms. These **health tests** are used to detect abnormal operating conditions, source failure, or excessive drift in the entropy-producing process. In practice, startup tests and continuous tests are often introduced to ensure that the generator remains within acceptable operating bounds during execution. This aspect is especially important in cryptographic hardware, where a failure in the random source can directly compromise the security of the entire system.

From a design perspective, the architecture of a TRNG must also be evaluated in terms of throughput, area, power consumption, and robustness to environmental variations. FPGA and ASIC implementations may require different trade-offs, depending on the target application and the available hardware resources. For this reason, a TRNG should not be described only as a source of random bits, but as an integrated subsystem whose performance depends on the interaction between the entropy source, the digital processing stages, and the validation logic.

2.1.1 Entropy source classification

True Random Number Generators can be implemented using different physical entropy sources, depending on the target platform, the available hardware resources, and the required level of robustness; generally a combination of multiple sources is exploited to generate randomness;. The main classes of entropy sources can be grouped as follows:

- **Thermal noise**, which exploits the random motion of charge carriers in electronic components.
- **Shot noise**, derived from the discrete nature of electrical current and widely used in analog circuits.
- **Avalanche noise**, generated by stochastic processes in reverse-biased semiconductor junctions.
- **Oscillator jitter**, which uses phase or timing fluctuations in ring oscillators or clocked systems.
- **Metastability**, where a circuit remains temporarily in an undefined state before resolving unpredictably.
- **Quantum phenomena**, such as photon arrival times, beam splitting, or other inherently probabilistic processes.

- **Device startup behavior**, for example SRAM power-up states or other hardware-dependent initialization effects.
- **Chaotic maps**, which exploit the extreme sensitivity to initial conditions of nonlinear dynamical systems; in some designs, they are used to generate complex and apparently random sequences, although their suitability as a true entropy source must be carefully evaluated because the underlying evolution is still deterministic unless it is coupled with a genuine physical source of randomness.

From an architectural point of view, TRNGs are often classified as *analog* or *digital*. Analog TRNGs rely on continuous-time physical effects such as noise sources, current fluctuations, or quantum processes, and they typically require analog front-end circuitry for amplification, filtering, and digitization. Digital TRNGs, on the other hand, exploit timing uncertainty, metastability, or oscillator jitter using mostly digital logic, which makes them easier to integrate in FPGA and ASIC implementations.

In practice, digital TRNGs are often preferred in resource-constrained hardware platforms, while analog TRNGs may offer a more direct physical entropy source but usually require more complex conditioning and interface circuitry.

2.1.2 Conditioning stage

The conditioning stage is a critical component of a TRNG architecture because the raw output of the entropy source is typically not statistically ready for direct cryptographic use. Even when the underlying physical process is genuinely non-deterministic, the sampled bitstream may still contain bias, correlation, or other structural imperfections introduced by the acquisition circuitry, limited resolution, or environmental sensitivity. The purpose of conditioning is to map this imperfect raw sequence into a more uniform output with improved statistical properties, while preserving as much entropy as possible. In hardware-oriented implementations, this function is often carried out through cryptographic post-processing, including hash functions, bit-mixing networks, or other diffusion mechanisms. Lightweight primitives such as ASCON can also be considered in this role, especially when compactness and efficiency are important design constraints. In this thesis one of the main objectives was to properly design an hardware block that could perform this type of purpose in the complete TRNG device.

It is important to distinguish conditioning from health tests: conditioning modifies the output data to improve its usability, whereas health tests are diagnostic procedures that monitor the entropy source and detect faults, degradation, or abnormal operating conditions. Therefore, conditioning is part of the data-processing chain, while health tests are part of the reliability and security monitoring framework.

2.2 Existing approaches to TRNG devices

True Random Number Generators have been implemented using several hardware strategies, each of which exploits a different physical or architectural source of unpredictability.

In FPGA-oriented and logic-based designs, the most common families are ring-oscillator-based TRNGs, DCM-based TRNGs, PLL-based TRNGs, and more specialized implementations that rely on oscillator jitter, timing uncertainty, or dedicated synchronization structures. Although these approaches share the same goal of producing unpredictable bits, they differ significantly in terms of entropy source, implementation complexity, sensitivity to placement and routing, throughput, and need for post-processing.

2.2.1 Ring Oscillator-based TRNG

A ring oscillator (RO) is a simple feedback-based circuit formed by an odd number of inverter stages connected in a closed loop. Because the number of inversions in the loop is odd, the circuit cannot settle to a stable logic level and instead produces a periodic oscillation.

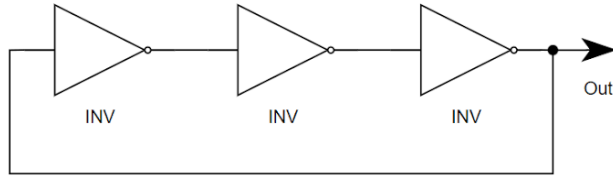


Figure 2.2: Block scheme of a RO with $N=3$.

In practical CMOS implementations, the oscillation frequency is determined mainly by the propagation delay of each stage and by the total number of inversions in the ring. For an ideal ring oscillator with N inverter stages and average propagation delay t_{pd} , the oscillation frequency can be approximated as

$$f_{RO} = \frac{1}{2Nt_{pd}} \quad (2.1)$$

$$T_{RO} = 2Nt_{pd} \quad (2.2)$$

This simple relation shows that the oscillation frequency increases when the number of stages decreases or when the propagation delay becomes smaller.

In real hardware, however, t_{pd} is not constant and varies with supply voltage,

temperature, process variation, and local routing effects, which makes the oscillator suitable as a source of timing uncertainty.

The key idea behind RO-based TRNGs is to exploit this timing uncertainty, especially **jitter**, as an entropy source. If one or more ring oscillators are sampled by a reference clock, the instantaneous phase differences between the oscillators and the sampling edge may vary in a non-deterministic manner. This variation can be converted into random bits by observing the sampled state of the oscillator, by comparing the outputs of multiple oscillators, or by counting oscillator transitions over a fixed time interval. In many designs, the output of several oscillators is combined through XOR logic in order to increase entropy and reduce structural bias.

From a stochastic point of view, the oscillator period can be modeled as

$$T_k = T_0 + \Delta T_k,$$

where T_0 is the average period and ΔT_k represents the random fluctuation due to jitter and environmental perturbations. Since the sampled output depends on these small variations, the resulting bitstream can exhibit unpredictability even if the nominal oscillator topology is fully deterministic. At the same time, the raw output is usually affected by bias and correlation, so a conditioning stage is often required before the generated bits can be used in cryptographic applications. Therefore, RO-based TRNGs should be seen as systems in which the entropy source is not the oscillator frequency itself, but the fine-grained timing noise associated with its instability.

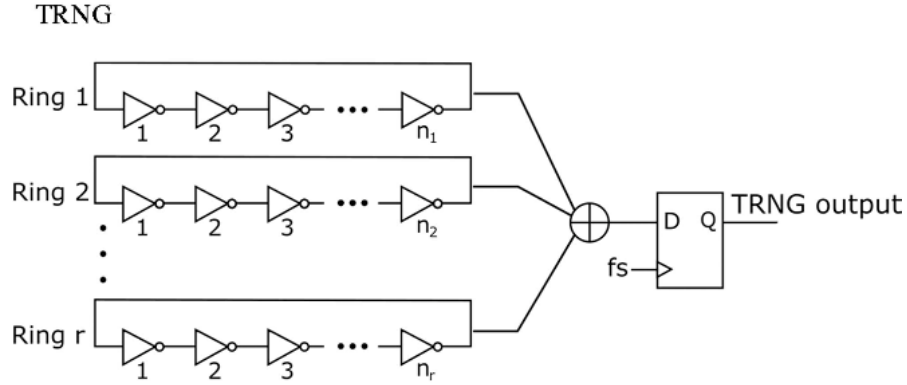


Figure 2.3: Key principle of a RO-based true random number generator.

In FPGA implementations, RO-based TRNGs are particularly attractive because ring oscillators can be realized using only a small amount of logic resources. However, their behavior is strongly influenced by placement and routing, and the actual entropy quality may vary across devices and operating conditions.

For this reason, practical designs usually include careful physical constraints, post-processing, and statistical validation to ensure that the generated sequence meets cryptographic requirements. Many implementations exploit this principle in literature, the following subsections will present a brief overview of them.

Transition Effect Ring Oscillator (TERO)

A TeRO-based TRNG exploits the transient behavior that appears when a feedback loop is enabled for a limited time. In a typical TeRO structure, the oscillation does not remain permanently active; instead, the circuit is triggered so that a burst of oscillations occurs, and the number of transitions observed during this burst depends on the intrinsic noise of the circuit. If the counter associated with the oscillator records the number of rising edges during one activation interval, the output entropy can be derived from the variability of that count.

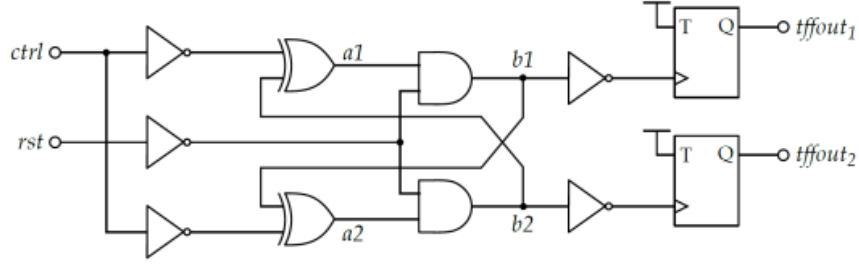


Figure 2.4: Scheme of a TeRO true random number generator device.

In a simplified model, if N_k denotes the number of oscillations generated during the k -th activation period, then the randomness is extracted from the fluctuating sequence $\{N_k\}$, which is influenced by jitter, delay variation, and metastability. The least significant bits of the counter are often used as raw random bits because they are the most sensitive to small timing perturbations.

rst	ctrl	a1	a2	b1	b2	STATE
1	0	0	0	1	1	stable
1	1	0	0	0	1	stable
0 → 1	0	xor1	xor2	xor2	xor1	metastable
0	1 → 0	xor1	xor2	NOT(xor2)	NOT(xor1)	metastable

Table 2.1: State evolution of the circuit varying the inputs.

Metastable Ring Oscillator (MERO)

The MeRO approach, on the other hand, modifies the classical ring oscillator structure to increase the observable effect of timing uncertainty. In a modified ring oscillator, the feedback network may include additional logic elements, altered inverter arrangements, or asymmetric paths that amplify the influence of delay variations. The purpose of this modification is to produce a source whose oscillation is still deterministic in theory, but whose fine-grained timing behavior is more sensitive to process variation and environmental noise.

If T_{RO} is the nominal oscillation period of a classical ring oscillator, the modified structure can be viewed as generating a perturbed period

$$T_{MeRO} = T_{RO} + \delta T,$$

where δT models the additional timing uncertainty introduced by the modified topology. As in other oscillator-based TRNGs, the randomness is not produced by the nominal oscillation itself, but by the stochastic variation around that nominal behavior.

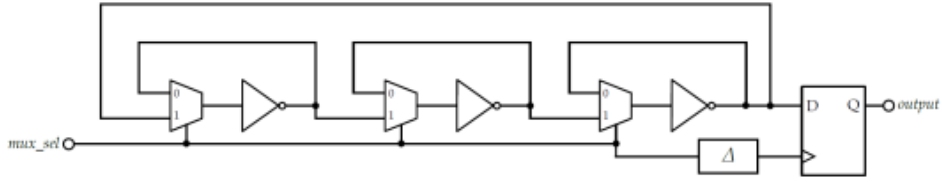


Figure 2.5: Scheme of a MeRO true random number generator device.

From a TRNG perspective, both TeRO and MeRO are attractive because they remain compatible with FPGA implementations and can often be realized with limited hardware resources. However, they also inherit the main limitations of oscillator-based generators: sensitivity to placement and routing, dependence on environmental conditions, and the need for post-processing or conditioning to remove bias and correlation. For this reason, their raw output is usually followed by statistical validation and, if necessary, by a cryptographic conditioning stage before the bits are used in security applications.

Fibonacci and Galois Ring Oscillator (FiRO and GaRO)

Fibonacci and Galois ring oscillators are two closely related families of oscillator-based entropy sources used in FPGA-oriented TRNG designs. In both cases, the basic idea is to build a feedback structure from simple logic elements so that the circuit never reaches a static stable state and instead produces continuously varying transitions. The main difference lies in the organization of the feedback path: in Fibonacci-style structures, the feedback is collected in a centralized way and applied to one stage, while in Galois-style structures the feedback is distributed across several stages. This structural distinction affects the way switching activity propagates through the circuit and therefore influences the timing uncertainty available for entropy extraction.

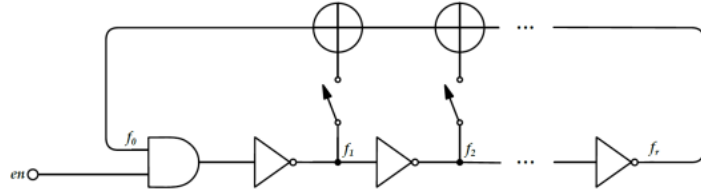


Figure 2.6: FiRO trng implementation scheme.

In a Fibonacci ring oscillator (FiRO), the feedback logic is concentrated in a way that resembles the classical Fibonacci configuration of shift registers. The global feedback condition determines the evolution of the internal states, and the resulting transitions can be sampled to obtain random bits. In a Galois ring oscillator (GaRO), by contrast, the feedback is distributed more locally across the chain, which may lead to different switching dynamics, improved layout flexibility, and potentially better control of propagation delay variations.

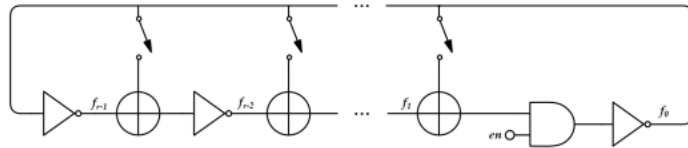


Figure 2.7: GaRO trng implementation scheme.

The FiGaRO architecture combines both Fibonacci and Galois ring oscillator instances in the same TRNG design. The motivation behind this combination is to exploit the complementary timing characteristics of the two oscillator families and to increase the entropy available at the output. In the implementation described in

the literature, multiple FiRO and GaRO modules are instantiated, their sampled outputs are XORed, and the resulting bitstream is then validated statistically. If x_i denotes the sampled output of the i -th oscillator, the combined raw bit can be represented as

$$b = x_1 \oplus x_2 \oplus \cdots \oplus x_m.$$

The XOR combination helps to reduce bias and to decorrelate the individual oscillator outputs, provided that the oscillators are sufficiently independent.

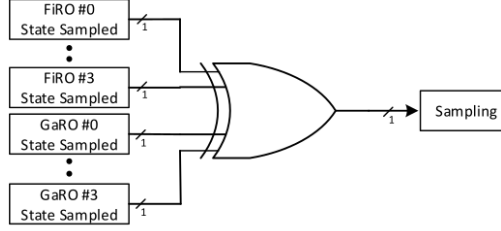


Figure 2.8: FiGaRO trng implementation scheme.

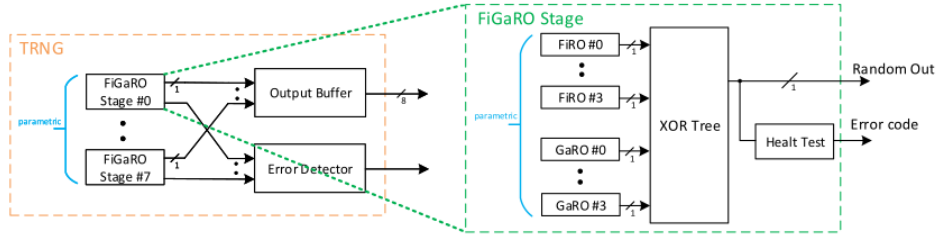


Figure 2.9: Complete chain of a FiGaRO device.

From a practical perspective, FiGaRO designs are attractive because they remain fully synthesizable in FPGA logic and can be scaled by changing the number of oscillator instances. At the same time, they inherit the usual limitations of oscillator-based TRNGs, including sensitivity to placement and routing, dependence on operating conditions, and the need for careful post-processing and statistical validation. For this reason, FiGaRO should be understood as a structured way of combining multiple oscillator-based entropy sources rather than as a fundamentally different physical phenomenon.

2.2.2 PLL-based TRNG

PLL-based True Random Number Generators exploit the timing uncertainty that can be observed in phase-locked loop structures. In contrast to purely oscillator-based designs, where randomness is usually extracted from the jitter of free-running

ring oscillators, PLL-based architectures rely on the phase noise and residual temporal fluctuations present in the output of the PLL or in the interaction between multiple clock domains. The basic idea is that, although the PLL is designed to generate a stable clock, its output still exhibits small non-deterministic variations that can be sampled and converted into random bits.

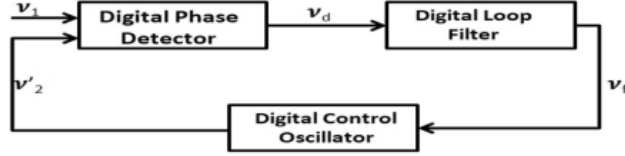


Figure 2.10: PLL block scheme.

A typical implementation compares, samples, or counts events derived from the PLL output, so that the accumulated timing uncertainty becomes the entropy source of the generator.

A simple way to model this behavior is to represent the sampled period as

$$T_k = T_0 + \Delta T_k,$$

where T_0 is the nominal period and ΔT_k is the random fluctuation caused by jitter and operating-condition variations. The raw output is then extracted from the resulting timing differences, often through edge sampling or coherent sampling between reference and PLL-derived signals.

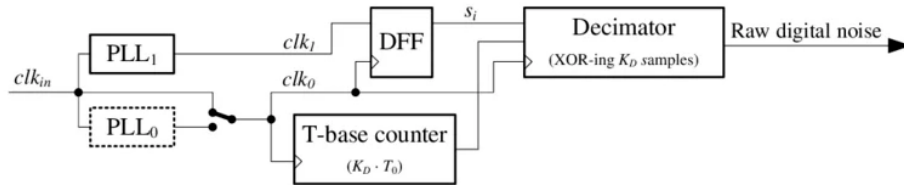


Figure 2.11: A possible implementation of a PLL-based TRNG.

Since the useful entropy is contained in the small deviations around the nominal clock behavior, PLL-based TRNGs usually require careful parameter selection, robust sampling strategies, and post-processing to remove bias and correlation. For this reason, the design of a PLL-based TRNG is not only a matter of using a low-jitter clock source, but also of creating an extraction mechanism that makes the residual uncertainty measurable and cryptographically useful.

From a practical perspective, PLL-based TRNGs are attractive because PLLs are commonly available in FPGA and embedded platforms, which makes them easy to integrate in hardware designs.

2.2.3 DCM-based TRNG

There are several possible implementations of a random number generator that exploit *Digital Clock Managers* (DCMs) available in some FPGA families. These hardware primitives offer a flexible and fully digital solution, which is particularly attractive in comparison with many analog entropy sources that may be sensitive to environmental variations, supply noise, or temperature changes. In a DCM-based TRNG, the key idea is to create timing uncertainty by controlling the relative phase between two or more clock signals, so that one or more flip-flops are driven into the metastability region. The random behavior is then extracted from the unstable resolution of these elements, which depends on extremely small timing perturbations that cannot be deterministically predicted.

The DCM can be configured so that the generated clock signal has programmable frequency, phase, or delay characteristics, and in some implementations these parameters may also be adjusted on-the-fly. This makes the architecture suitable for tunable TRNG designs, since the entropy source can be optimized according to the target FPGA, the operating conditions, and the desired balance between throughput and randomness quality. In the literature, two main DCM-based approaches are commonly considered:

- **Frequency tuning approach:** the output clock frequency is modified by configuring the DCM parameters, so that the interaction between the reference clock and the generated clock increases sampling uncertainty and can lead some flip-flops into metastability.

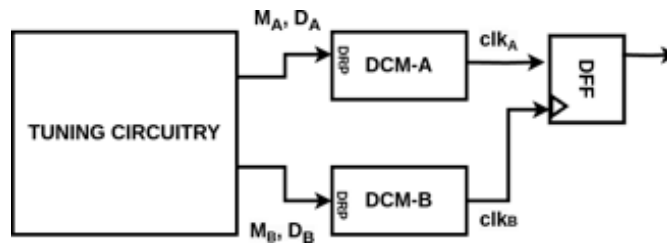


Figure 2.12: Frequency tuning DCM trng scheme.

- **Dynamic phase shifting (DPS) approach:** instead of changing the output frequency, the DCM exploits its phase-shifting capability to introduce

controlled phase offsets between clock domains, forcing one or more flip-flops to operate near the metastability boundary.

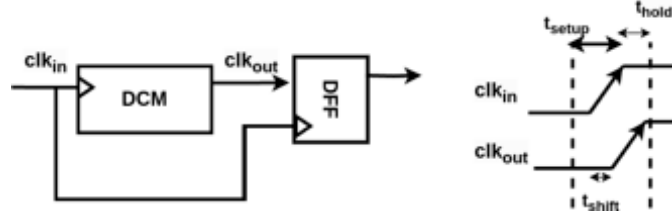


Figure 2.13: DSP-based DCM trng scheme.

In both cases, the entropy does not come from the DCM itself, but from the unpredictable resolution of the sampling elements when they are exposed to finely controlled timing differences. For this reason, the quality of a DCM-based TRNG depends on the choice of the operating point, the stability of the digital routing, and the post-processing stage used to reduce bias and correlation in the raw output.

Although this chapter has focused on the most common TRNG families used in hardware-oriented cryptographic systems, several other implementation strategies have also been proposed in the literature. These include designs based on chaotic maps, metastability-based circuits, delay-line structures, sampler-based architectures, and hybrid approaches that combine multiple entropy sources. Each solution presents different trade-offs in terms of area, throughput, robustness, and sensitivity to environmental variations. For this reason, no single architecture can be considered universally optimal, and the choice of the most suitable TRNG strongly depends on the target platform and application requirements. The implementations discussed in this chapter therefore represent only a subset of the broader landscape of true random number generation techniques.

Chapter 3

Statistical Validation and Post-Processing

A True Random Number Generator must be assessed not only according to the physical origin of its entropy, but also on the basis of the statistical properties of its output. Even when the underlying source is genuinely non-deterministic, the raw bitstream may still exhibit imperfections that make post-processing necessary before cryptographic use. Statistical validation and post-processing therefore constitute essential steps in the evaluation of practical TRNG implementations. The former is used to characterize the observed randomness and identify structural weaknesses, while the latter is employed to reduce bias and improve the uniformity of the output sequence.

In this chapter, the statistical tools used to analyze the data produced by the FPGA-implemented device are presented. The conditioning algorithm selected to integrate a post-processing block into the system is then introduced, together with its main advantages and limitations. Finally, the NIST-compliant test suites used for the statistical validation of the generated bitstreams are described. A short overview of health tests is also provided to frame their role as an essential safeguard for entropy-source operation.

In the previous chapters, the concepts of randomness, reliability, and robustness were introduced. Among the key parameters used to assess the quality and compliance of a TRNG, **entropy** plays a crucial role. Entropy is a fundamental metric for characterizing the unpredictability of a random source in the context of True Random Number Generators. For a discrete random variable X with probability mass function $p(x)$, the classical Shannon entropy is defined as

$$H(X) = - \sum_{x \in \mathcal{X}} p(x) \log_2 p(x),$$

and it represents the average amount of information generated by the source. This quantity is useful to describe the overall uncertainty of the output, but it does not fully capture the worst-case predictability of a TRNG bitstream.

For cryptographic applications, min-entropy is often a more appropriate metric. It is defined as

$$H_{\infty}(X) = -\log_2 \left(\max_{x \in \mathcal{X}} p(x) \right),$$

and it measures the maximum probability that an attacker can guess the most likely output symbol. In this sense, min-entropy provides a conservative estimate of the usable randomness, especially when the raw output is affected by bias, correlation, or other non-idealities. A TRNG may therefore exhibit a reasonable Shannon entropy while still showing a lower min-entropy, which makes conditioning and post-processing essential in practical implementations.

Entropy estimation and statistical validation should be considered complementary. The NIST test suites that will be described later in this chapter consent to identify visible deviation from ideal random behaviour and to provide a proper min-entropy estimation. For this reason, a TRNG must be evaluated both statistically and in terms of its entropy content, since passing randomness tests alone is not sufficient to guarantee cryptographic suitability.

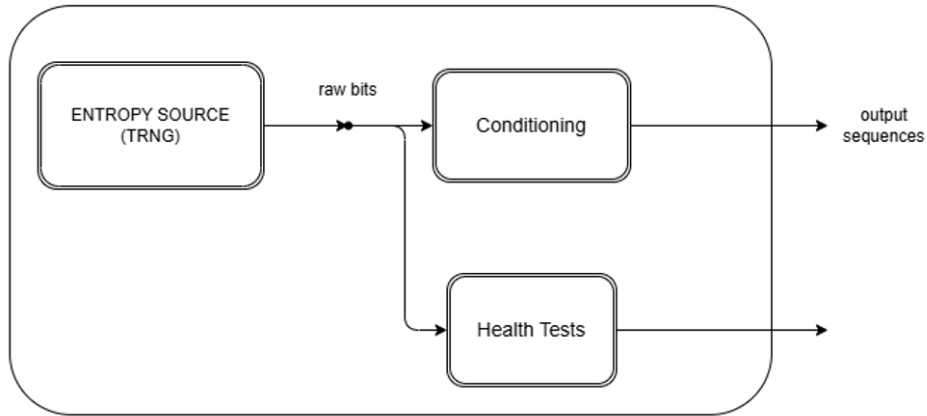


Figure 3.1: Block scheme of the system that generates the entropy (TRNG device).

After discussing the various possible implementations of noise or entropy sources in the previous chapter, this chapter focuses on conditioning and on validation through statistical tests, while also briefly addressing the role of health tests.

3.1 Conditioning block

In order to eliminate bias and correlation from the raw output of a TRNG a conditioning stage is commonly included in practical architectures to improve the statistical quality and the usability of the generated data. Its role is not to create entropy by itself, but rather to process the available raw bits in a way that reduces visible non-idealities and produces a more balanced output.

Several conditioning techniques have been proposed in the literature, each with different trade-offs in terms of simplicity, throughput, and cryptographic strength. Common approaches include:

- **Von Neumann correction**, which removes first-order bias by discarding equal pairs of bits and keeping only alternating pairs, at the cost of a significant reduction in throughput.
- **Hash-based conditioning**, which compresses and diffuses the input sequence using a cryptographic hash function, making the output more uniform and harder to predict.
- **Permutation-based or sponge-based methods**, which exploit a cryptographic permutation or sponge construction to mix the input bits and extract a fixed-size output.
- **Linear mixing techniques**, such as XOR folding, which are simple to implement but usually provide weaker guarantees than dedicated cryptographic primitives.
- **Cryptographic extractors**, which are designed to transform a weakly random source into a shorter sequence with improved statistical properties.

Among these alternatives, lightweight cryptographic primitives are particularly attractive for hardware-oriented implementations, since they can provide strong diffusion while keeping area and power consumption under control.

As described in detail in the following chapters, the data files generated by the device developed in this thesis were compared with data files produced by other entropy sources and post-processed using alternative conditioning algorithms, such as the SHA-3 hash function. In this work, ASCON is used as the conditioning mechanism, and its role is discussed in detail in the following subsection.

3.1.1 ASCON-based Conditioning Model

ASCON is used in this work as the conditioning mechanism applied to the raw output of the TRNG. The choice of ASCON is motivated by its lightweight nature, its suitability for hardware implementation, and its cryptographic design, which makes it well adapted to embedded and FPGA-based systems. It belongs to the family of permutation-based lightweight cryptographic algorithms and is particularly attractive because it provides a compact yet robust structure for data mixing. Its sponge-like design allows input bits to be absorbed into an internal state and then transformed through a sequence of nonlinear operations and permutations.

As a result, even limited variations in the raw input can influence the final output in a nontrivial manner, improving diffusion and reducing visible patterns in the generated bitstream.

The ASCON family of cryptographic functions is based on a lightweight permutation that iteratively applies an SPN-based round transformation, with a variable number of rounds up to a maximum of 16. The standard permutation used in the selected conditioning block employs 12 rounds and is organized into three separate steps operating on an internal 320-bit state divided into 64-bit words.

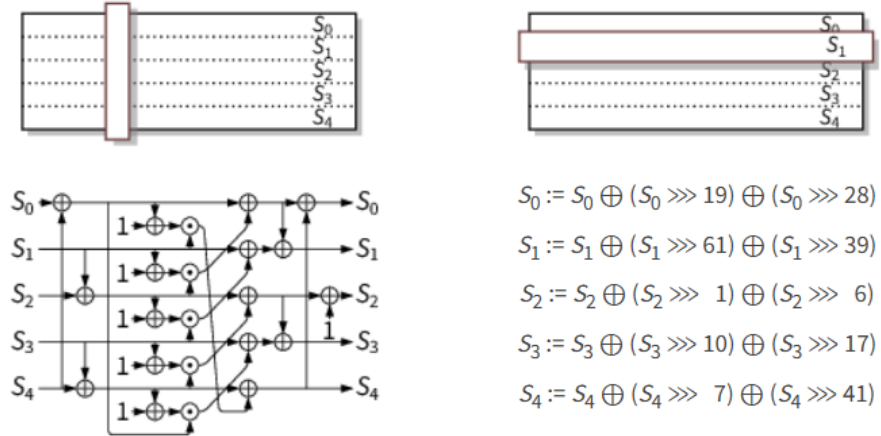


Figure 3.2: General scheme of the permutation where the internal state is presented and the logical expressions used in the layers are depicted.

The three steps are the following:

- **Addition of Round constants:** this is the operation that xors a round-specific 1-byte constant to the second word of the internal state; the choice of the constant depends on the number of rounds of the algorithm.
- **Nonlinear Substitution layer:** applies a 5-bit S-box 64 times in parallel in

a bit-sliced fashion (vertically, across words).

- **Linear Diffusion Layer:** xors different rotated copies of each word (horizontally, within each word).

To perform the whitening process required by the TRNG system developed in this thesis, a hash function from the ASCON family was used. This function adopts a sponge-based mode of operation and provides 128-bit security when the hash output size is 256 bits.

The sponge-based ASCON hash function operates through three main phases:

- **Initialization:** the internal state is initialized with a fixed parameter set and the input data is incorporated into the state structure.
- **Absorbing:** the message blocks are sequentially XORed into the state and processed through the permutation, allowing the input information to be mixed and diffused.
- **Squeezing:** the final hash output is extracted from the internal state, producing the digest that represents the processed input data.

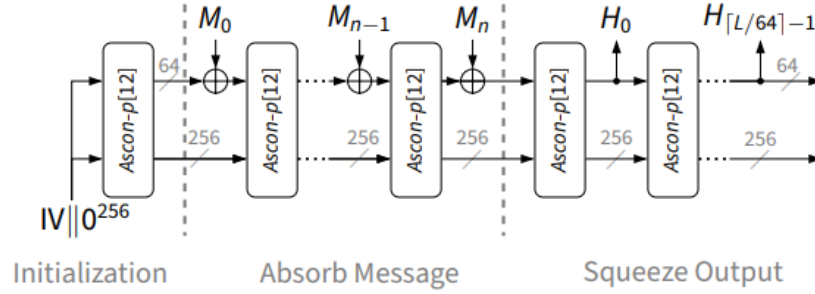


Figure 3.3: Hashing mode phases for ASCON-Hash 256 bit.

The following figure illustrates the pseudocode of the hash algorithm used, which served as the starting point for the implementation of the hardware conditioning block on FPGA that will be presented and discussed in detail in the next chapter.

Algorithm 5 Ascon-Hash256(M)

Input: Bitstring $M \in \{0, 1\}^*$
Output: Digest $H \in \{0, 1\}^{256}$

```

 $IV \leftarrow 0x0000080100cc0002$  ▷ Initialization
 $\mathcal{S} \leftarrow \text{Ascon-p}[12](IV \parallel 0^{256})$ 

 $M_0, \dots, M_{n-1}, \widetilde{M}_n \leftarrow \text{parse}(M, 64)$  ▷ Absorbing
 $M_n \leftarrow \text{pad}(\widetilde{M}_n, 64)$ 
for  $i = 0$  to  $n - 1$  do
     $\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_i$ 
     $\mathcal{S} \leftarrow \text{Ascon-p}[12](\mathcal{S})$ 
end for
 $\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_n$ 

 $\mathcal{S} \leftarrow \text{Ascon-p}[12](\mathcal{S})$  ▷ Squeezing
for  $i = 0$  to  $2$  do
     $H_i \leftarrow \mathcal{S}_{[0:63]}$ 
     $\mathcal{S} \leftarrow \text{Ascon-p}[12](\mathcal{S})$ 
end for
 $H_3 \leftarrow \mathcal{S}_{[0:63]}$ 

 $H \leftarrow H_0 \parallel H_1 \parallel H_2 \parallel H_3$ 
return  $H$ 
    
```

In conclusion ASCON presents several advantages that make it particularly suitable as a conditioning algorithm in a TRNG architecture. Its lightweight design allows efficient hardware implementation with limited area and power consumption, which is especially important in FPGA-based systems. In addition, its sponge-based structure provides strong diffusion and mixing capabilities, helping to reduce bias and correlation in the raw output.

ASCON also offers a good balance between cryptographic robustness and implementation simplicity, making it more reliable than purely linear post-processing techniques. Finally, its standardized nature and compact permutation-based construction make it a practical choice for embedded and resource-constrained applications. For these reasons, ASCON represents an effective solution for the whitening stage of the TRNG developed in this thesis.

3.2 Health Tests

Health tests are an essential part of practical True Random Number Generator designs because they provide a means of continuously checking whether the entropy source is operating correctly. Unlike conditioning, which acts on the generated data to improve its statistical quality, health tests are diagnostic mechanisms whose purpose is to detect failures, abnormal operating conditions, or severe degradations in the behavior of the source. In a TRNG, a malfunctioning entropy source may produce biased, frozen, or highly correlated outputs, and such conditions can compromise the security of the entire cryptographic system. For this reason, health tests are usually integrated alongside the entropy generation and post-processing stages to ensure that the output remains trustworthy during normal operation. Their role is not to make the sequence more random, but to verify that the generator is still producing data consistent with the expected model of operation.

From a practical point of view, health tests are commonly classified into different categories depending on when and how they are applied:

- **Start-up tests:** executed when the device is powered on or initialized, in order to verify that the entropy source begins operation in a valid condition.
- **Continuous tests:** performed during normal operation to detect persistent faults, stuck outputs, or unexpected long-term failures.
- **On-the-fly or on-the-run tests:** lightweight checks carried out during runtime, often with minimal overhead, to identify abnormal source behavior without interrupting the generator.
- **Repetition count tests:** used to detect sequences of repeated identical values that may indicate loss of entropy.
- **Adaptive proportion tests:** used to monitor the statistical balance of bits within a moving window and detect excessive bias.

In this context, health tests should be seen as a security safeguard rather than as a statistical improvement mechanism. When a test fails, the generator is typically forced into an error state, disabled, or reinitialized, depending on the system architecture and the required security policy.

3.3 NIST test suites

Several statistical test suites are available in the literature for the evaluation of random bitstreams. The most appropriate framework depends on the characteristics of the generator under test and on the type of validation required. In this thesis, the adopted methodology is based on two NIST recommendations: the **NIST Statistical Test Suite**, which includes 15 tests targeting different statistical properties of binary sequences, and **NIST SP 800-90B**, which is used for entropy-source evaluation and min-entropy estimation. These two tools are complementary, since SP 800-22 assesses the observable statistical behavior of the output sequence, whereas SP 800-90B focuses on the intrinsic quality of the entropy source.

3.3.1 NIST Statistical Test Suite

The suite is composed of 15 statistical tests in order to detect the level of randomness of the analyzed bitstream. The evaluation of the tests is performed through standard hypothesis testing. The null hypothesis H_0 assumes that the sequence under examination is random with respect to the property analyzed by the test, while the alternative hypothesis H_1 indicates a departure from randomness.

For each statistical test, the outcome is evaluated by comparing the computed p-value with a predefined significance level α , which represents the maximum acceptable probability of rejecting a truly random sequence. In most practical contexts, α is chosen to be small, typically 0.01, although stricter thresholds may also be adopted depending on the desired confidence level. The p-value is obtained from the test statistic and quantifies how likely it is to observe a result at least as extreme as the measured one under the assumption that the null hypothesis H_0 is true. If the p-value is greater than or equal to α , there is insufficient evidence to reject H_0 , and the sequence is considered to have passed the test. Conversely, if the p-value is lower than α , the null hypothesis is rejected and the sequence is deemed to fail that specific statistical criterion.

When multiple sequences are tested, the evaluation should consider both the proportion of passing sequences and the uniformity of the p-values. In the NIST Statistical Test Suite, a sequence passes a test when its p-value satisfies $p \geq \alpha$, where α is the chosen significance level. The observed pass rate is then compared with a confidence interval determined by the sample size and α , and the p-values are checked for uniformity through a chi-square goodness-of-fit analysis. This approach ensures that the validation does not rely on a single outcome, but on the overall statistical behavior of the generator.

Type of test	Description
Frequency	Checks whether the number of zeros and ones is approximately equal in the entire sequence.
Block frequency	Verifies the balance of zeros and ones within fixed-size blocks of the sequence.
Cumulative Sums	Measures the extent to which the cumulative sum of bits deviates from zero.
Runs	Tests whether the number of runs of consecutive identical bits is consistent with randomness.
Longest Run	Examines the length of the longest run of ones in a block of bits.
Rank	Checks the linear independence of fixed-size binary submatrices.
FFT	Detects periodic patterns by analyzing peaks in the discrete Fourier spectrum.
Non Overlapping Template	Searches for the occurrence frequency of a specified bit pattern without overlaps.
Overlapping Template	Counts repeated occurrences of a specific pattern allowing overlaps.
Universal	Estimates the compressibility of the sequence as a measure of randomness.
Approximate Entropy	Measures the frequency of similar patterns in the sequence to estimate regularity.
Random Excursions	Evaluates the distribution of visits to different states in a cumulative random walk.
Random Excursions Variant	Tests the frequency of visits to states in a random walk-based sequence.
Serial	Assesses the occurrence frequency of overlapping patterns of length m and $m - 1$.
Linear Complexity	Estimates the length of the shortest linear feedback shift register that can generate the sequence.

Table 3.1: Overview of the tests of the NIST Statistical Test Suite.

The table 3.1 depicts a brief overview of the 15 tests used by the suite, with a brief description of them. So a possible output of the suite on an analyzed bitsream is reported in the following picture.

RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES												
C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
6	12	9	12	8	7	8	12	15	11	0.616305	0.9900	Frequency
11	11	12	6	10	9	8	9	17	7	0.474986	0.9900	Cusum
6	10	8	14	16	10	10	6	5	15	0.129620	0.9900	Cusum
7	9	9	11	11	11	8	12	12	10	0.978072	0.9900	Serial
13	6	13	15	9	7	3	11	13	10	0.171867	0.9600	Serial

For each statistical test, the results are summarized in a row containing 12 values. The first ten values describe the distribution of the p-values obtained from the m analyzed sequences: in particular, the unit interval is divided into ten bins in order to assess the uniformity of the results. The eleventh value is the p-value produced by a chi-square test, which indicates whether the distribution is consistent with uniformity and therefore whether the test is passed or failed. The last value represents the proportion of sequences that pass the test with respect to the total number of sequences considered. For the result to be acceptable, this proportion should lie within the corresponding confidence interval.

3.3.2 NIST SP 800-90B suite

NIST SP 800-90B is a standard specifically devoted to the evaluation of entropy sources used in random bit generators. Unlike output-oriented statistical suites, this recommendation focuses on the source itself, with the aim of verifying whether the produced data can be considered suitable for entropy estimation. The analysis is based on a set of preliminary tests designed to identify structural weaknesses, dependencies, or deviations from the assumptions required for a reliable entropy source.

In particular, the suite includes the following checks:

- **IID assessment:** verifies whether the observed sequence can be modeled as independent and identically distributed.
- **Restart test:** checks the behavior of the source across multiple acquisitions or restarts, in order to detect possible non-stationarities or dependencies between runs.

- **Compression and predictability analysis:** evaluates whether the sequence shows patterns that would reduce its effective entropy.
- **Entropy estimation:** provides a lower bound on the min-entropy of the source by applying several statistical estimators.

The purpose of SP 800-90B is therefore not to prove randomness in a strict sense, but to characterize the entropy source and determine whether it can support secure random bit generation. For this reason, it is commonly used together with output-based statistical tests such as those defined in SP 800-22, so that both the source quality and the observable randomness of the output can be assessed.

Chapter 4

Complete TRNG System Design and FPGA Implementation

This chapter focuses on the design and implementation of the system integrating the TRNG and the ASCON-based conditioning block. The idea is to start from a random number generator that exploits a known entropy source, in our case an approach based on ring oscillator (RO) structures, and to build a device capable of collecting the generated binary data. These data can first be analyzed without applying any conditioning, and then again after the inclusion of the block implementing the whitening function.

The selected RO-based TRNG has the advantage of offering a level of security and robustness that is well established in the literature. In addition, it provides a high degree of implementation flexibility, which makes it relatively easy to realize the system on an FPGA.

A further goal is to create a small database that can be used to statistically compare the obtained results with data files generated from other known noise sources; in order to achieve this need different blocks will be synchronized and accessible with an automated system drawn in SystemConsole. The proposed design follows a hierarchical architecture, in which the various components have been described in VHDL and made compatible with the Cyclone 10 GX FPGA development kit on which the device has been implemented.

The following sections present in detail the blocks that have been designed and developed and highlight the advantages of the choices made during the design process. For intermediate analysis and simulation, tools such as ModelSim, Quartus System Console, and additional TCL scripts were used.

4.1 TRNG Core and Input Memory Interface

The first component of the considered system is shown in Figure 4.1 and consists of the TRNG, which generates the raw input bitstream for the system, together with an interface block. This block is responsible for connecting the generator to the memory used for collecting the binary data and for synchronizing data sampling with the subsequent processing stages.

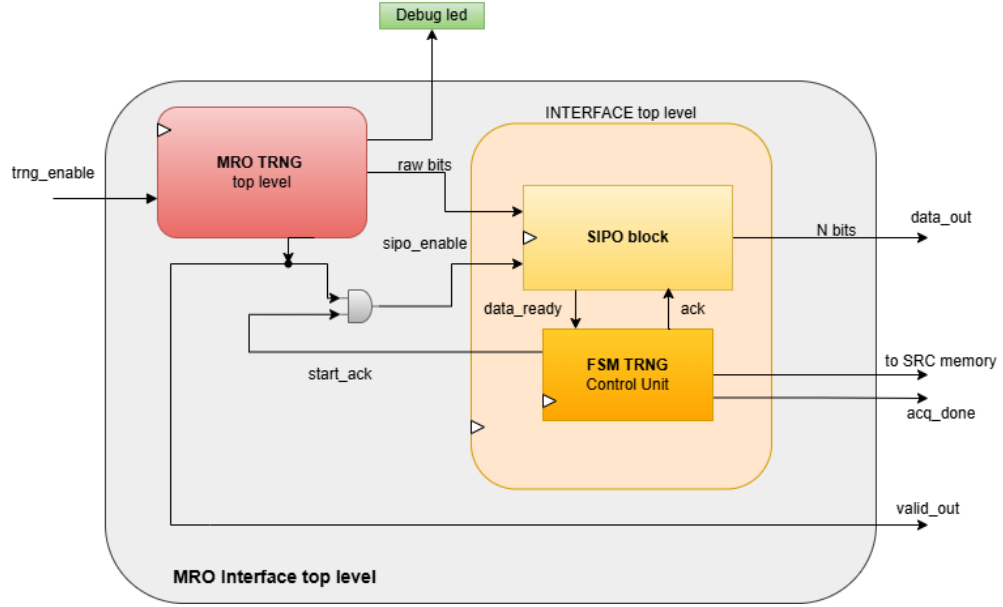


Figure 4.1: Interface TRNG block scheme. Clock and asynchronous active-low reset signals were omitted.

In addition to the clock signal and the asynchronous reset rst_n , the interface block receives an enable signal used to start the noise generator (MRO top level) and consequently begin sampling the output data. The purpose of the interface block is to parallelize the TRNG output into a variable number of bits. The management of the multi-bit output data and the synchronization mechanism with the memory used to store the generated data are handled by the interface block finite state machine.

In addition to the output data bus, the system provides a signal indicating that a K -bit datum is ready ($data_ready$) and a signal indicating that data acquisition has been completed (acq_done). The following subsections present the various components in detail.

4.1.1 Noise Source (TRNG core)

As mentioned previously, the entropy source is based on an RO configuration (Figure 4.2). However, it is not a conventional RO-based TRNG; rather, it is a hierarchical implementation of a muxed ring oscillator (MRO) architecture, which differs from the classical approach in the way the entropy generated by the oscillators is collected and propagated to the output.

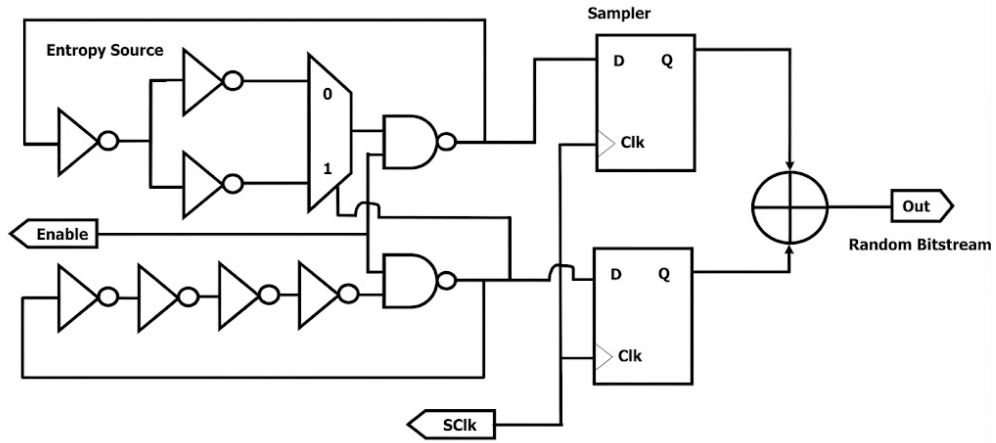


Figure 4.2: Scheme of a Muxed RO-based true random number generator.

In a standard RO-based generator, the output bitstream is obtained by directly sampling the jitter of one or more ring oscillators, typically combined through simple logic structures such as XOR networks or sampling flip-flops. This solution is attractive because of its simplicity, compactness, and ease of FPGA implementation, but it may be more sensitive to bias, correlation, and supply-induced effects if the sampling strategy is not carefully designed.

In contrast, an MRO architecture introduces a multiplexer stage that dynamically selects among different oscillators or oscillator paths, allowing the designer to distribute the sampling process more evenly and improve the flexibility of the overall entropy collection mechanism.

From a practical point of view, the muxing approach can offer better robustness and controllability, since it reduces the dependence on a single oscillator and may help mitigate undesired deterministic behavior. For these reasons, this was the architecture employed in our design.

Design parameters

The number of ring oscillators employed in the MRO architecture has a direct impact not only on the area, power consumption, and maximum operating frequency of the circuit, but also on the statistical behavior of the generated bitstream. For this reason, particular attention has been devoted to identifying a suitable trade-off between implementation cost and randomness quality.

In general, increasing the number of oscillators tends to enhance the amount of available entropy and to improve the diversity of the sampled jitter contributions, but it also enlarges the hardware footprint and may introduce additional routing and timing constraints. On the other hand, using too few oscillators can simplify the design, yet it may reduce the variability of the entropy source and limit the effectiveness of the muxing mechanism. This is especially relevant in an MRO-based implementation, where the multiplexer dynamically selects among multiple oscillator paths and the overall quality of the output depends on how effectively these paths are diversified.

Based on the information reported in the reference paper included in the bibliography, and considering the specific requirements of the proposed design, an architecture composed of 8 ring oscillators operating in parallel, each made up of 13 inverters, was selected for this work. This configuration was chosen as a suitable compromise between implementation cost and the statistical quality of the generated bitstream, while also remaining compatible with the FPGA-based realization of the system. The selected structure provides a sufficiently rich source of entropy and, at the same time, keeps the overall design compact and manageable from both an architectural and a synthesis perspective. Moreover with this structure all the 15 NIST tests are passed and the quality of the output bitstream is sufficient. A brief note on the VHDL implementation of the TRNG is in order. In order to

preserve the metastability generated by the ring oscillators, it was necessary to add the *keep* attributes shown below to the VHDL code, so as to prevent the automatic optimization performed by the Quartus RTL synthesis tools.

```
1      -- Attributi su tutti i segnali critici
2      attribute keep      : boolean;
3      attribute keep      of ro_outputs    : signal is true;
4      attribute keep      of ro_debug_xor : signal is true;
5      attribute preserve  : boolean;
6      attribute preserve  of ro_outputs    : signal is true;
7      attribute preserve  of ro_debug_xor : signal is true;
8      attribute noprunes  : boolean;
9      attribute noprunes  of ro_outputs    : signal is true;
10     attribute noprunes  of ro_debug_xor : signal is true;
```

4.1.2 Interface component

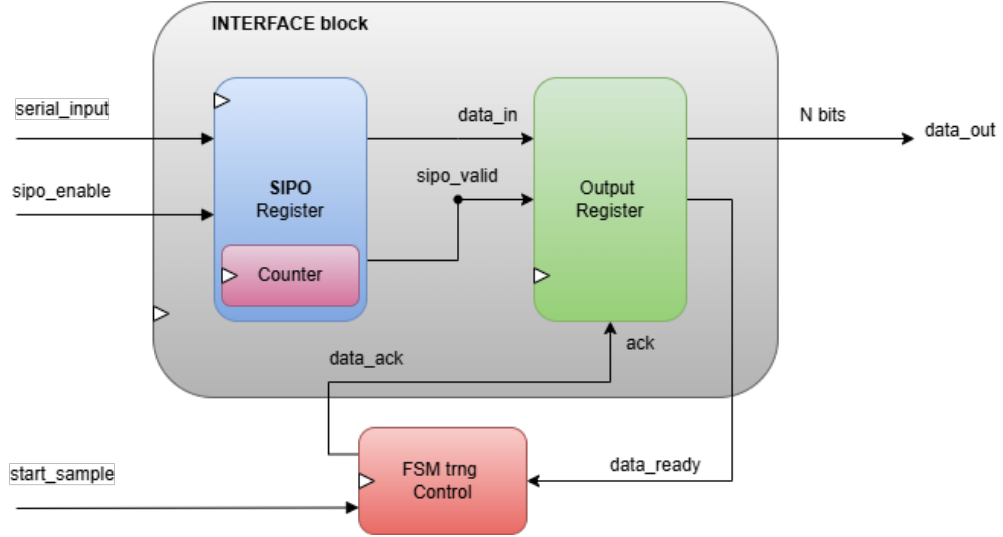


Figure 4.3: Interface block scheme. Clock and reset signals are omitted.

The interface block was designed with the goal of parallelizing the raw-bit output produced by the generator. By sampling the noise source at a given frequency, it avoids consecutive bit repetitions, thereby improving the overall entropy of the system and providing an N -bit parallel output. These data are then stored in a source memory, from which binary data files are extracted and saved for subsequent statistical analyses.

At the hardware level, the implemented block consists of a SIPO register, which is enabled every time a bit is sampled from the noise source, and an output shift register that stores the sampled bits and produces a valid output word when enabled by an internal SIPO counter. The entire process is managed by a control finite state machine, which also regulates the writing of the generated data into the source memory while synchronizing the operation with the other components of the system.

The external signals *start_sample* and *sipo_enable* are driven by TCL scripts that automate the complete system. Communication between the FSM and the registers is handled through signals such as *data_ack* and *sipo_valid*. Since the memory parallelism adopted in this work will be discussed in detail at the end of this chapter, the generated output data are organized into 64-bit words, matching the word size of the input memory.

4.1.3 Interface Control Unit

The finite state machine shown in Figure 4.4 governs the process of writing the data generated by the TRNG into the source memory. Its behavior is structured so as to wait for the availability of data with the required degree of parallelism and to enable the interface registers at the appropriate time. Moreover, the memory write operation and the address update logic are managed in such a way that the synchronization FSM, introduced in the following sections, is informed when the memory has been completely filled.

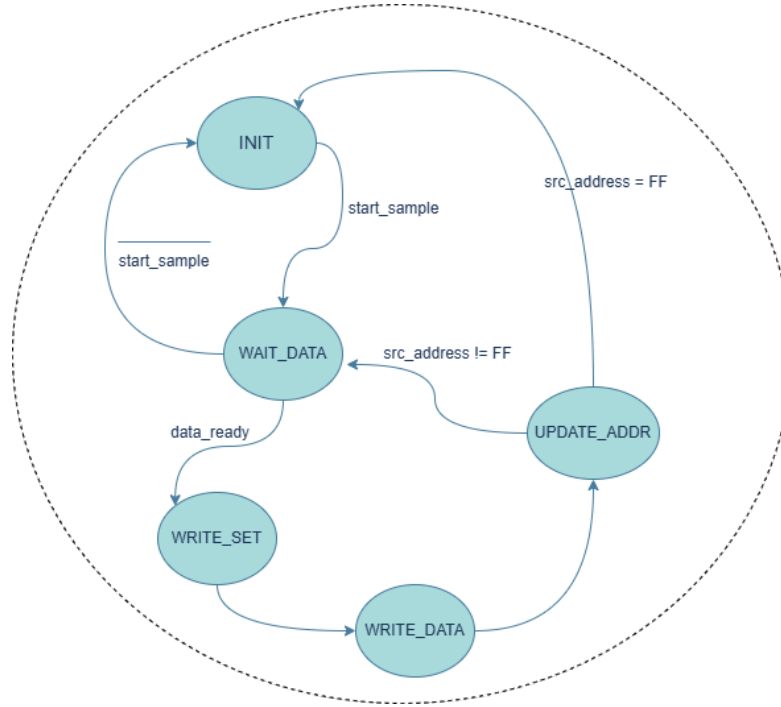


Figure 4.4: FSM Control for the interface TRNG-Memory source.

In the complete system, managed through Quartus System Console, a form of health testing can be performed to verify that the generator is operating correctly and that the memory writing procedure is proceeding as expected. By means of properly designed TCL scripts, snapshots of the memory state are acquired under specific conditions, and blocks of 64-bit data words generated by the noise source can be analyzed in real time to assess their statistical validity.

4.2 ASCON Conditioning Block

In the following picture the Conditioning block is presented, along with its state machine able to control the core of the hashing process in the total system.

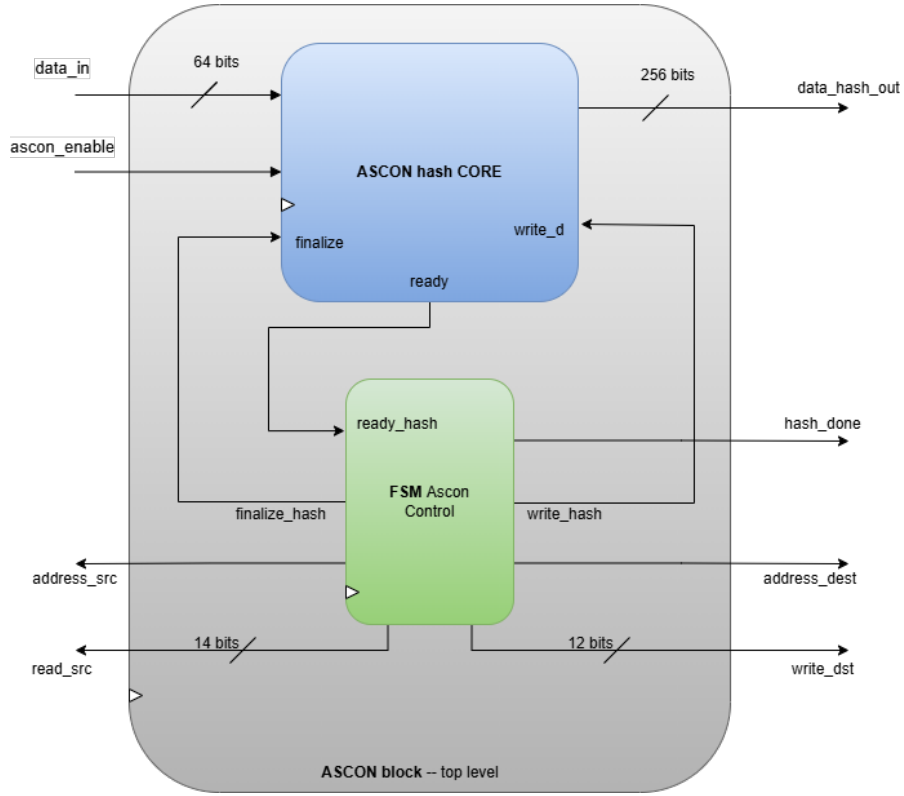


Figure 4.5: Block scheme of the ASCON conditioning implemented component.

The implemented block is based on the ASCON hash algorithm described in the previous chapter. The design of the core component started from the software implementation of the hashing function available in the ASCON GitHub documentation. In order to properly develop the conditioning system on the FPGA board, the entire sponge-like algorithm was manually translated into VHDL and implemented from scratch. The operating principle is as follows: the input data (*data_in*) consist of 64-bit words coming from the source memory. Once a block of four words is available, the hashing function is executed, producing a 256-bit hashed output. The output data bus then fills the destination memory, from which binary data files can later be extracted once the memory is full. All timing issues and synchronization between the different parts of the algorithm are managed by the ASCON control FSM, which also supervises the reading of the input data from the source memory and the writing of the hashed data into the destination memory.

4.2.1 ASCON Core Block

The conditioning algorithm is described in detail in Chapter 3, where it is shown to be divided into three phases: permutation, absorption, and squeezing. The objective is therefore to implement a working hardware block capable of performing this hash algorithm on input data received as a stream of 64-bit words. The internal state of the function is 320 bits wide, while the output of the block is 256 bits.

The process is managed by a finite state machine designed to correctly carry out all the required steps. Its graphical representation is shown below.

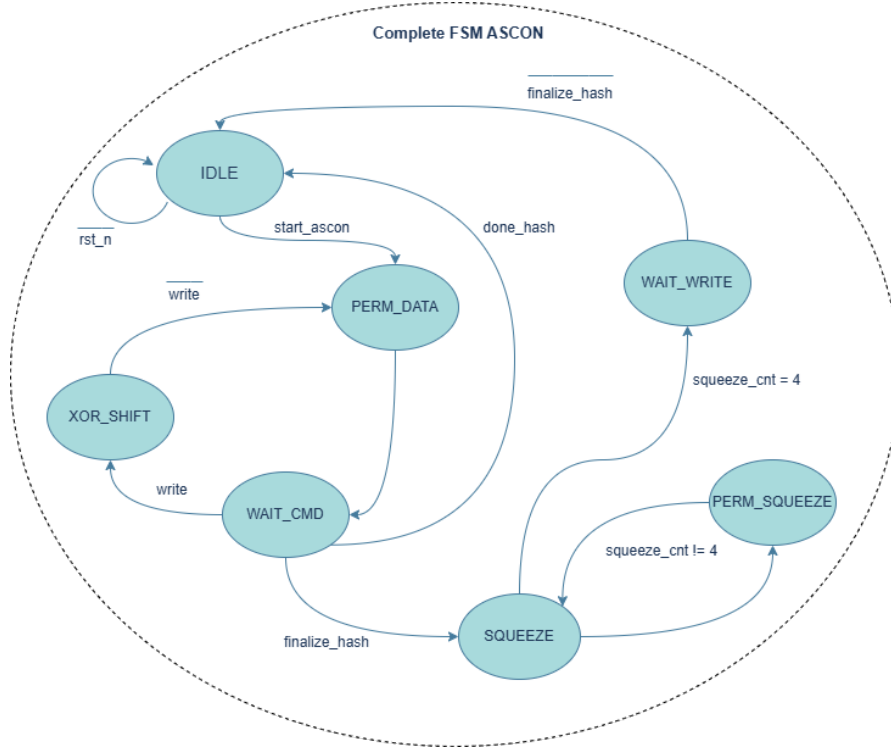


Figure 4.6: FSM representation for the conditioning process.

Let us now describe the conditioning process in more detail. The initial phase corresponds to the logical permutations introduced in the previous section. In the IDLE or INIT state, the internal state is initialized by loading the HASH IV constant into the first word, as specified in the documentation for the 12-round version of the algorithm, while all the remaining words are set to zero.

When the ASCON block is enabled through the *start_ascon* signal, the FSM moves to the PERM_DATA state, where the first permutation phase is executed. The WAIT_CMD state acts as the main decision point of the machine. If the *write* signal is asserted, a new 64-bit input word is available and is absorbed into the internal state after being XORed with the initialization constant in the

XOR_SHIFT state. If, instead, the *finalize* signal is asserted, the FSM transitions to the final squeezing phase.

During this phase, the SQUEEZE and PERM_SQUEEZE states complete the necessary operations and produce the 256-bit output block, which is then ready to be written into memory in the WAIT_WRITE state. Finally, when *done_hash* is asserted in the WAIT_CMD state, indicating that the hashing process has been completed, the FSM returns to the IDLE state.

Once the block had been fully defined in VHDL, it was tested using ModelSim and Signal Tap, with particular attention paid to signal timing. After verifying that it operated correctly, the output results were compared with the certified software implementations available in the documentation used as a reference for the design of the entire block. Since all results matched, the block could then be successfully integrated into the complete system.

4.2.2 Conditioning Control Unit

The management of the signals that activate the different working phases of the conditioning algorithm, as well as the handling of read and write operations to the source and destination memories and the consequent access to the data to be hashed, are entrusted to a second control finite state machine implemented within the ASCON block.

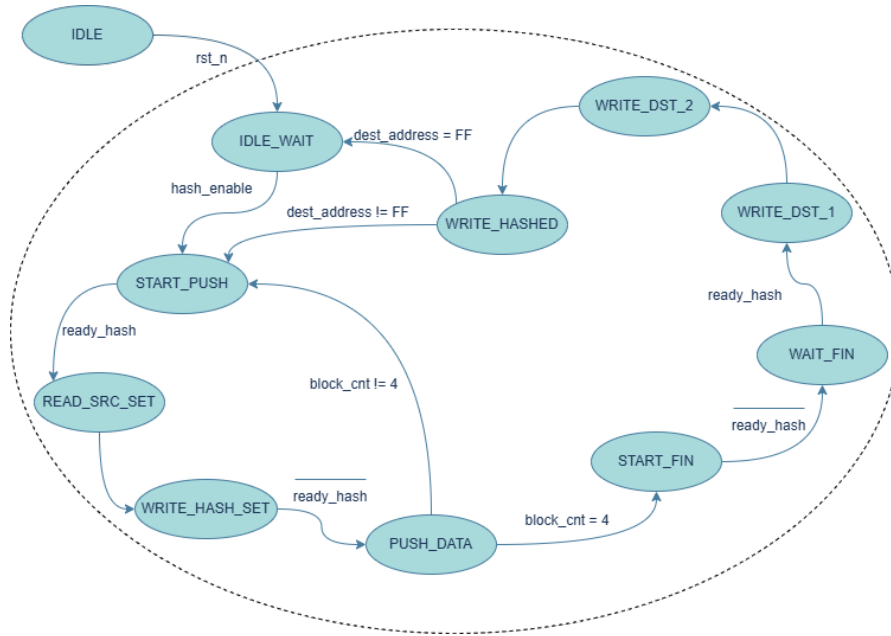


Figure 4.7: FSM representation for the controller for the ASCON algorithm

In summary, the IDLE_WAIT state is responsible for initiating the conditioning block. It waits for the enable signal generated by the external synchronization state machine and authorizes the start of the push phase for the words read from memory.

From START_PUSH to PUSH_DATA, the data required to execute the hash algorithm are read one word at a time. During this phase, the *read_enable* signals and the corresponding memory access addresses are properly managed.

The START_FIN state marks the beginning of the final squeezing phase of the function and enables the writing of the correctly formed output data into memory. Finally, in the WRITE_HASHED_DATA state, the address of the output memory is monitored in order to stop the process once the memory is full.

4.3 Complete System Synchronization

In the following figure (4.8), the complete system is illustrated. The TRNG block and the ASCON conditioning block are interconnected through an FSM control unit, introduced to manage synchronization issues and to ensure the correct switching between the different operating phases.

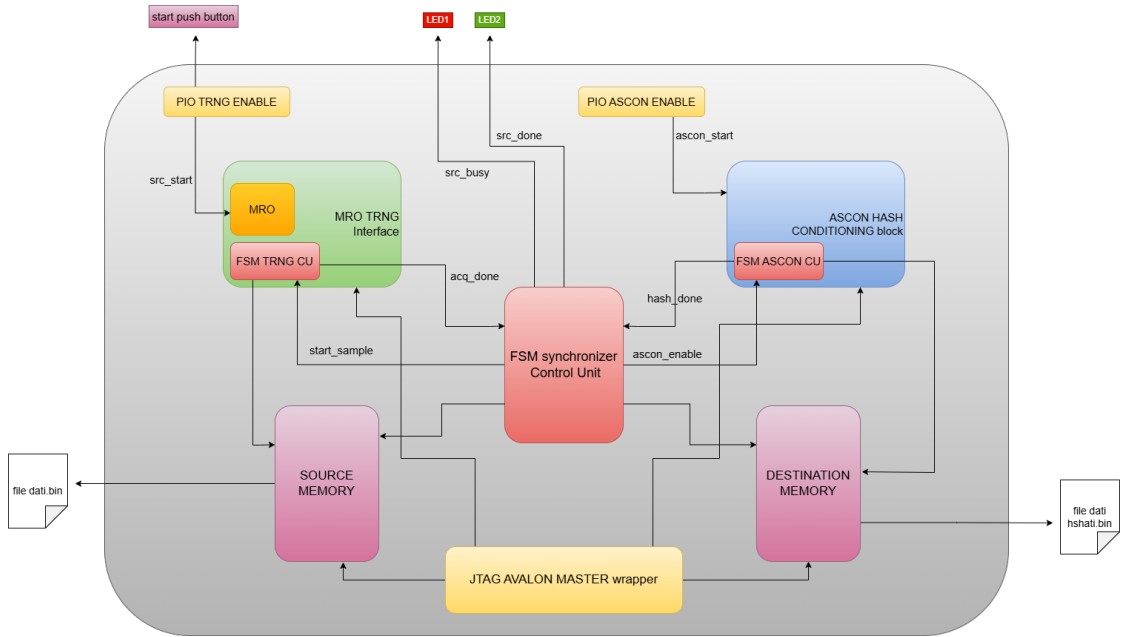


Figure 4.8: Complete system block scheme. Only significant signals are depicted.

The complete system was therefore implemented on an FPGA using a Cyclone 10 GX development board. To manage its operation, the entire flow was automated through a set of TCL scripts used to control the registers associated with the different system components.

Using Quartus Platform Designer, a system was built in which the two memories were instantiated and configured according to the required signals and parallelism, which will be discussed in more detail in the next subsection. In addition, a set of PIO registers was created to allow the software layer to control the activation signals that enable the various stages of the system. The JTAG Avalon wrapper makes it possible to establish a master-slave hierarchy connecting the VHDL-designed blocks to the memories and to the full System Console development environment. A hierarchical system of interconnected modules was then properly created, linking all the VHDL components to the master interface.

The use of TCL scripts in System Console also makes it possible to access the contents of memory at any time and to save the results in binary data files, which can then be transferred to the statistical testing and validation environment described in detail in the next chapter.

We will now examine in detail the main parts of the complete system, in order to provide a comprehensive and concise overview of the operation of the device that is the central subject of this thesis.

- The system is started by means of a push-button on the board, which enables the *src_start* signal and activates the random generator, thereby starting the entire interface block and the first data acquisition phase in the source memory. A second push-button acts as an asynchronous reset and restores the system and all its components to their initial state.
- The enabling of the individual blocks, namely the TRNG, the conditioning block, and the memories, is managed through the registers of the JTAG wrapper and by commands issued via TCL scripts.
- The synchronization of the blocks, the management of all internal signals, and the processes handled internally by the control FSMs of the MRO and ASCON components are coordinated by the FSM synchronizer, which is described in detail in the following subsection.
- As can be seen from the diagram in the figure, the two memories are accessible in order to store binary data files. This is essential not only for monitoring the system behavior during operation by taking snapshots of the memory contents, but also for collecting files to be included in the database used to compare the results obtained from the physical hardware device with noise files generated through different simulation-based and theoretical approaches.

- finally some leds of the board were used to signal the state of operations during the whole workflow and for debugging issues (signals *src_busy* and *src_done*).

4.3.1 Synchronizer FSM Controller

The Finite State Transition diagram of the complete synchronization block is reported in the figure below.

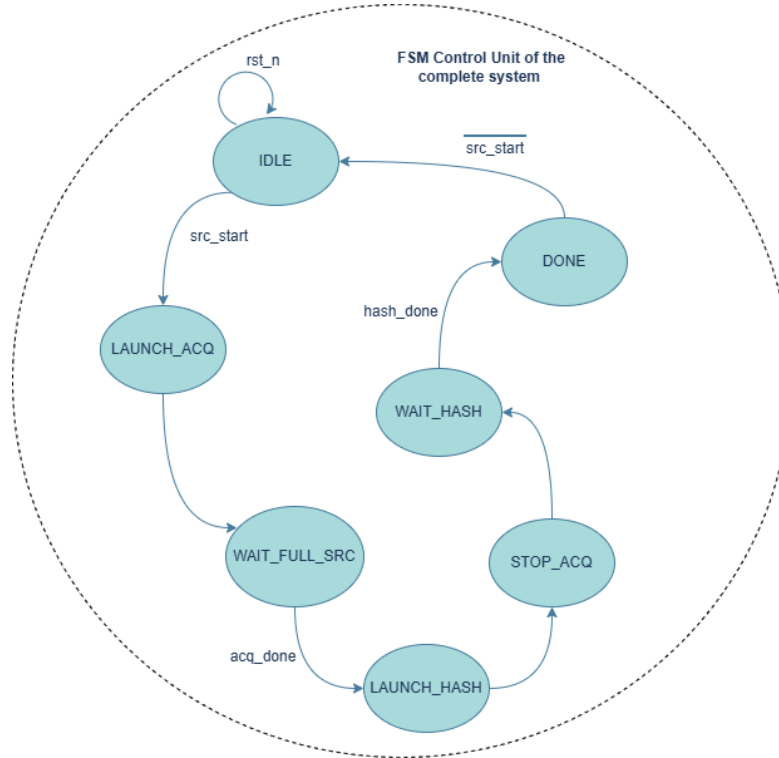


Figure 4.9: FSM representation for the complete system synchronization.

The synchronization FSM manages the various operating phases of the device. In the *LAUNCH_ACQ* state, it enables the start of data sampling from the true random number generator and stores the data in the source memory in the correct format. It then waits until the data have been properly written into the source memory and enables the hashing conditioning process in the *LAUNCH_HASH* state, stopping data acquisition before proceeding.

The FSM then waits until all data produced after the whitening process have been written into the destination memory and finally stops the device, returning to the idle state and waiting for a new start signal.

4.3.2 Memory Architecture and Design Considerations

For completeness, we conclude with a few considerations regarding the chosen level of parallelism for managing the generated data. The conditioning algorithm adopted in this work is based on ASCON-Hash-256 and therefore produces hashed output words of 256 bits. Consequently, the destination memory was designed with a word size of 256 bits and with sufficient capacity to store an adequate amount of data, so that the resulting files could be compared in a statistically meaningful way according to the requirements of NIST and of the selected test suite. In particular, in order to obtain data files of at least 10^6 or 10^7 bits, which represents the minimum required size, the destination memory was implemented with 4096 slots of 256-bit words, accessible through a 12-bit address bus.

The source memory, on the other hand, stores 64-bit words and provides a larger number of available slots (16386) in order to maintain the same overall output-file size while allowing better management of the interaction between the noise-generation block and the conditioning block. In this way, the correct proportion between the data generated by the TRNG and the data stored at the output of the device is preserved.

In conclusion, this chapter has presented the hardware implementation of the complete device, including the TRNG core, the interface logic, the synchronization mechanisms, and the ASCON-based conditioning block. The system has been developed with a hierarchical structure and successfully mapped onto the FPGA platform, ensuring correct operation through simulation, timing verification, and hardware testing. Particular attention has been devoted to the interaction between the different modules, as well as to the management of memory access and data synchronization. The implemented architecture therefore provides a complete and flexible framework for the generation, conditioning, and storage of random data, which will be further analyzed in the following chapter from a statistical validation perspective.

Chapter 5

Testing Environment for Data Generation and Statistical Analysis

This chapter presents the experimental environment developed to generate, transform, store, and compare binary data obtained from different entropy sources. The main objective is to assess the statistical behavior of the produced bitstreams under a controlled and reproducible framework, so as to obtain a more comprehensive evaluation of their quality. In particular, the proposed environment allows raw data to be collected from the TRNG-based hardware platform, processed through different conditioning functions, and then organized into binary files that can be analyzed and compared using a common validation procedure.

The need for such an environment arises from the fact that the statistical quality of a random sequence cannot be fully characterized by a single test or by a single acquisition mode. For this reason, the framework has been designed to support multiple sources of entropy and to make it possible to compare their outputs under homogeneous conditions. In addition to the raw sequences generated by the physical device, the environment also handles data obtained from alternative noise sources and from theoretical or simulation-based models, thereby enabling a broader and more meaningful statistical comparison. This approach is particularly useful when evaluating whether the behavior of the implemented generator is consistent with that of other known sources of randomness.

A second key aspect of the environment is the possibility of applying hashing and conditioning operations before the statistical analysis. In this context, the hash functions are not considered as cryptographic primitives only, but also as tools for whitening the raw data and improving their distributional properties. By comparing unprocessed and conditioned files, it becomes possible to observe how

much the post-processing stage affects the randomness-related metrics and whether it helps in removing residual bias, correlation, or other structural effects introduced by the source. This makes the framework suitable not only for validating the output of the device, but also for studying the effectiveness of the conditioning stage itself.

The overall structure of the environment has therefore been organized around a complete test flow, from data acquisition to final validation. Binary files are generated, stored, and then inserted into a dedicated database used for subsequent analysis. The comparison of results is performed according to the selected statistical suite, allowing the same evaluation criteria to be applied uniformly to all datasets. As previously stated all the reports generated by the statistical analyses are saved in folders of the database and are used to fill tables of comparisons that will be reported in the next chapter.

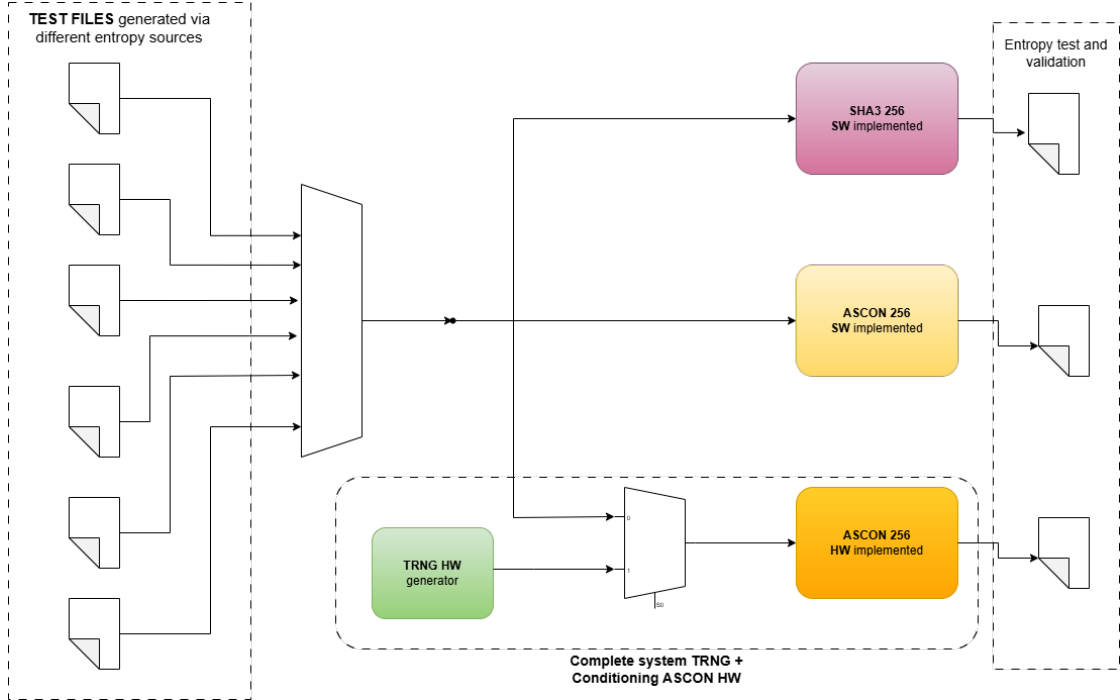


Figure 5.1: Conceptual scheme of the environment of test.

In this way, the chapter provides the methodological basis for the following sections, where the individual components of the environment, the data processing pipeline, and the validation criteria will be described in detail.

5.1 Entropy Sources Used

In this subsection the entropy sources used within the experimental framework developed for the statistical validation of the generated data will be described. In order to perform a meaningful comparison, the test environment has been designed to handle binary files obtained from different sources, including the implemented TRNG-based hardware generator and additional reference datasets derived from alternative noise models or externally available bitstreams.

Using multiple entropy sources makes it possible to analyze not only the raw output of the device under test, but also the effect of different acquisition and conditioning strategies on the statistical quality of the resulting sequences. For this reason, the datasets considered in this work provide a useful benchmark for understanding how the implemented generator behaves with respect to other sources of entropy and whether its output remains consistent with the expected statistical requirements.

In summary, the test files generated from different entropy sources are listed below:

- **Linux `/dev/urandom`:** a pseudo-random source provided by the operating system and commonly used as a benchmark for generating high-quality random data.
- **Bash-generated PRNG sequence:** a pseudo-random file produced through Bash commands, used as a controllable synthetic dataset for testing purposes.
- **Ramp-distribution file:** a file whose values follow a ramp-like progression, introduced to assess the behavior of the system on clearly non-random structured data.
- **Gaussian-distribution file:** a file generated according to a Gaussian distribution, used to evaluate how the statistical tests respond to structured but non-uniform data.
- **Repeated random bit sequences:** a file obtained by repeating random bit sequences in a pseudo-random order, designed to introduce controlled correlations and partial regularities.
- **PNG binary file:** the binary representation of a PNG image, used as a real-world dataset with an inherent file structure and limited randomness.

To generate the test files and subsequently perform the statistical tests, as well as to store the generated reports in the database, Bash scripts were developed to make the entire process hierarchical and automated. The following figure (5.1) shows the portion of the workflow that perform the software generation of the test files that are provided as input to the system.

```

1  # Genera i file binari da testare
2  mkdir -p $EXEDIR || exit 1
3  make -C $BINDIR
4
5  mkdir -p $DATADIR || exit 1
6
7  #-----
8  # File random con urandom
9  if [ ! -f $DATADIR/dev_urandom.bin ]; then
10     echo "Genero un file da /dev/random di lunghezza 1MB"
11     dd if=/dev/urandom of=$DATADIR/dev_urandom.bin bs=1024 count
        =1024 || exit 1
12 fi
13 # Converte in esadecimale, 32 byte per riga.
14 xxd -p -g 32 -c 32 <$DATADIR/dev_urandom.bin >$DATADIR/
        dev_urandom.txt || exit 1
15
16 #-----
17 # File con distribuzione a rampa
18 if [ ! -f $DATADIR/ramp.bin ]; then
19     echo "Genero un file sequenziale(0...255) di 1MB"
20     $EXEDIR/generate_ramp 1048576 >$DATADIR/ramp.bin || exit 1
21 fi
22 # Converte in esadecimale, 32 byte per riga.
23 xxd -p -g 32 -c 32 <$DATADIR/ramp.bin >$DATADIR/ramp.txt || exit
        1
24
25 #-----
26 # File pseudo randomico da 1Mbit
27 if [ ! -f $DATADIR/prng.bin ]; then
28     echo "Genero un file da PRNG di lunghezza 1MB"
29     $EXEDIR/generate_prng 1048576 >$DATADIR/prng.bin || exit 1
30 fi
31 # Converte in esadecimale, 32 byte per riga.
32 xxd -p -g 32 -c 32 <$DATADIR/prng.bin >$DATADIR/prng.txt || exit
        1
33
34 #-----
35 # File con distribuzione gaussiana normale
36 if [ ! -f $DATADIR/gaussian.bin ]; then
37     echo "Genero un file con distribuzione gaussiana, media 2^31, su
        32 bit"
38     $EXEDIR/gaussian 500000 >$DATADIR/gaussian.bin || exit 1
39 fi
40 # Converte in esadecimale, 32 byte per riga.
41 xxd -p -g 32 -c 32 <$DATADIR/gaussian.bin >$DATADIR/gaussian.txt
        || exit 1

```

Listing 5.1: Code snippet of the bash script generating random test files.

5.2 Hash Functions Employed

Now we will present the hash functions employed in the experimental framework and explain their role in the validation process. In this work, hashing is applied as a post-processing step before the statistical analysis in order to condition the raw data and improve their distributional properties. More specifically, the hash functions are used as a whitening mechanism as we have already stressed in previous chapters. This makes it possible to assess whether the transformation reduces residual bias, weak correlations, or other structural effects that may still be present in the unprocessed data.

The choice of multiple hash functions is motivated by the need to observe how different algorithms affect the resulting bitstreams in terms of output length, diffusion capability, processing efficiency, and impact on the final statistical quality. For this reason, the comparison is not limited to the raw generated files, but also includes the corresponding hashed versions, so that a more complete picture of the conditioning process can be obtained. In the following, the selected functions are introduced together with the main characteristics that make them suitable for this experimental setup.

The hash functions considered in this work are SHA3-256 and ASCON, the latter evaluated in both software and hardware implementations. SHA3-256 is included as a reference solution, since it is a well-established and widely adopted cryptographic hash function characterized by a fixed output size of 256 bits and strong diffusion properties. In the context of this experiment, it provides a useful baseline for observing the effect of a standard high-security hashing procedure on the statistical quality of the generated data.

ASCON is tested in two different configurations in order to compare the impact of the implementation environment on the conditioning process. The software version allows a direct software-based evaluation, while the hardware version is intended to assess the same hash function when implemented in a dedicated processing architecture. This comparison is particularly relevant in terms of execution speed, resource usage, and practical suitability for embedded or low-power systems.

From an experimental point of view, the three variants differ mainly in output format, processing efficiency, and implementation cost, while sharing the same role as conditioning functions applied before statistical validation. Their use makes it possible to compare not only the quality of the resulting bitstreams, but also the trade-off between diffusion effectiveness and computational overhead.

```

1      ##### Genera hash dei file di input
2  for f in dev_urandom ramp prng sequences gaussian image ; do
3
4      # SHA3
5      if [ ! -f $DATADIR/sha3-256_$f.txt ]; then
6          $EXEDIR/hash_hex_sha3-256 $DATADIR/$f.txt || exit 1
7          mv output.txt $DATADIR/sha3-256_$f.txt || exit 1
8          rm -f output.bin
9      fi
10     #Genera la versione binaria
11     xxd -r -p -g 32 -c 32 <$DATADIR/sha3-256_$f.txt >$DATADIR/sha3
        -256_$f.bin || exit 1
12
13     #SHA3 concatenato
14     if [ ! -f $DATADIR/concat_sha3-256_$f.txt ]; then
15         $EXEDIR/hash_hex_concat_sha3-256 $DATADIR/$f.txt || exit 1
16         mv output.txt $DATADIR/concat_sha3-256_$f.txt || exit 1
17         rm -f output.bin
18     fi
19     #Genera la versione binaria
20     xxd -r -p -g 32 -c 32 <$DATADIR/concat_sha3-256_$f.txt >$DATADIR
        /concat_sha3-256_$f.bin || exit 1
21
22     # ASCON
23     if [ ! -f $DATADIR/ascon-256_$f.txt ]; then
24         $EXEDIR/hash_hex_ascon-256 $DATADIR/$f.txt || exit 1
25         mv output.txt $DATADIR/ascon-256_$f.txt || exit 1
26         rm -f output.bin
27     fi
28     #Genera la versione binaria
29     xxd -r -p -g 32 -c 32 <$DATADIR/ascon-256_$f.txt >$DATADIR/ascon
        -256_$f.bin || exit 1
30 done

```

Listing 5.2: Code snippet of the bash script performing the hash of the input generated files.

5.3 Experimental Workflow

Now that the structure of the generated test environment has been defined, this paragraph describes the workflow adopted in this thesis. As mentioned in Chapter 3, the statistical validation suites used in this work are the official NIST Test Suite, which includes 15 tests, and the SP800-22 suite, employed for additional tests and entropy evaluation.

The workflow of this test environment, developed in parallel with the construction and implementation of the complete FPGA-based device described above, is as follows.

1. **File generation:** each test file is generated from one of the selected entropy sources or synthetic data patterns, according to the procedures described in the previous sections. All generated files satisfy the minimum size required by the NIST specifications, ensuring that the statistical tests can provide reliable results.
2. **Optional hashing:** when needed, the generated file is processed through one of the hash functions considered in this work to obtain a conditioned version of the data stream. At first, the software implementations of ASCON and SHA-3 were used; later, once the complete hardware device had been designed and implemented, it was integrated into the experimental environment.
3. **Statistical testing:** both the original and, when available, the hashed files are submitted to the selected statistical tests to evaluate their randomness properties.
4. **Report collection:** the reports produced by the testing tools are collected and stored in a structured manner, allowing the results to be compared across the different datasets.
5. **Result comparison:** the final step consists in comparing the statistical outcomes obtained from the various files, in order to highlight the impact of the input source and the conditioning process, as well as to assess the validity of the tested entropy sources.
6. **Automation support:** whenever possible, the workflow is automated through scripts and supporting tools, such as Bash scripts, C/C++ generation programs, and Makefiles, in order to reduce manual intervention and improve repeatability.

For completeness, the following figures show examples of the reports generated by these types of tests. These reports will then be analyzed in detail in the next chapter, where their results will be used as a basis for the subsequent discussion and comparison.

File	Mean	Median	Chi-square	iid Perm.	LRS
/dev/urandom	127.51	128	Passed	Passed	Passed

Table 5.1: Summary of the main SP800-22 results for the /dev/urandom file.

```

-----
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
-----
generator is </home/x1085967/Tesi_update/AMBIENTE_TEST/hw_generated_data/bin_hashed_files/dev_urandom_hash.bin>
-----
C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  P-VALUE  PROPORTION  STATISTICAL TEST
-----
9   6   4   4   4   3   5   5   5   5   0.851383  50/50       Frequency
1   2   7   5   7   3   3   4   9   9   0.096578  50/50       BlockFrequency
8   5   3   5   2   4   3   9   5   6   0.455937  50/50       CumulativeSums
7   8   3   4   5   1   7   3   8   4   0.319084  50/50       CumulativeSums
4   9   6   3   5   4   3   3   9   4   0.383827  49/50       Runs
2   7   7   6   6   4   6   8   2   2   0.383827  49/50       LongestRun
3   7   7   6   6   4   6   8   2   2   0.122325  49/50       Rank
9   11  3   5   3   2   1   9   2   5   0.008879  49/50       FFT
4   6   4   3   4   6   3   9   3   8   0.494392  50/50       NonOverlappingTemplate
7   3   4   5   4   2   9   7   5   4   0.534146  48/50       NonOverlappingTemplate
4   2   9   6   4   5   4   7   8   1   0.236810  49/50       NonOverlappingTemplate
7   6   4   5   3   8   0   8   5   4   0.289667  50/50       NonOverlappingTemplate
3   4   1   7   6   8   5   7   1   8   0.171867  49/50       NonOverlappingTemplate
4   4   4   3   8   4   2   13  2   6   0.017912  50/50       NonOverlappingTemplate
5   5   5   9   5   2   5   4   4   6   0.779188  50/50       NonOverlappingTemplate
7   3   4   5   3   6   4   6   10  2   0.350485  48/50       NonOverlappingTemplate

```

Figure 5.2: First part of the NIST 15-test report for the /dev/urandom file.

```

-----
FILE = /home/x1085967/Tesi_update/AMBIENTE_TEST/hwsamples/hashes_20260612_150136.bin
-----

BITSREAD = 10000 0s = 4976 1s = 5024
BITSREAD = 10000 0s = 5111 1s = 4889
BITSREAD = 10000 0s = 5042 1s = 4958
BITSREAD = 10000 0s = 4982 1s = 5018
BITSREAD = 10000 0s = 5112 1s = 4888
BITSREAD = 10000 0s = 4992 1s = 5008
BITSREAD = 10000 0s = 4969 1s = 5031
BITSREAD = 10000 0s = 4996 1s = 5004
BITSREAD = 10000 0s = 4995 1s = 5005
BITSREAD = 10000 0s = 4985 1s = 5015
BITSREAD = 10000 0s = 4979 1s = 5021
BITSREAD = 10000 0s = 5022 1s = 4978
BITSREAD = 10000 0s = 5054 1s = 4946
BITSREAD = 10000 0s = 5014 1s = 4986

```

Figure 5.3: First part of the frequency report for the /dev/urandom file.

This chapter has introduced the experimental framework adopted in this work, including the generation of the test files, the conditioning stage based on hash functions, and the statistical validation workflow applied to the resulting data streams. The overall methodology has been designed to ensure reproducibility and to allow a consistent comparison between raw and conditioned data under the selected test suites. In the next chapter, the results of the statistical tests will be presented and discussed in detail, together with the technical characteristics of the hardware device developed for the conditioning stage. This will make it possible to analyze both the quality of the generated bitstreams and the practical impact of the implemented architecture on the experimental outcomes.

Chapter 6

Results Analysis and Comparison

This chapter presents the final results of the work, with a focus on randomness evaluation, performance analysis and comparison with related solutions from the literature. It also discusses the main findings obtained from the experimental testing environment and outlines possible future developments and optimizations of the proposed approach in the last section.

6.1 NIST Suite Test Results

First, after completing the TRNG development and implementing the ASCON conditioning block, the system was evaluated using the NIST tests described in Chapter 3. This test campaign was applied initially to the raw TRNG output and subsequently to the version hashed by the ASCON block. The estimated entropy of the raw output bitstream is 0.9965, expressed as Shannon entropy per bit.

The results of the NIST SP 800-22 test suite are reported in Tables 6.2, 6.3 and 6.4, comparing first the complete device implemented in hardware on FPGA and then the effect of software-based conditioning through SHA-3 and ASCON applied to the bitstreams generated by the noise sources. For tests producing multiple outcomes, the average value was considered. Files of approximately 2×10^6 bits (about 2 Mbit) were analyzed. The test parameters were selected in compliance with the NIST guidelines and are reported in the following table.

A significance level of $\alpha = 0.01$ was adopted, and a test was considered passed when its p -value was greater than α , as described in detail in the previous section.

Statistical test	Block Length
Block frequency test	128
Non Overlapping Template	9
Overlapping Template	9
Serial test	12
Approximate Entropy test	10
Linear Complexity test	500

Table 6.1: Parameters chosen for NIST test suite.

Test	No whitening			Whitening (ASCON)		
	P-value	Prop.	Result	P-value	Prop.	Result
Frequency	0.05898	0.98	PASS	0.3191	0.98	PASS
BlockFreq.	0.02381	0.88	PASS	0.5341	0.98	PASS
Cumulat.Sums	0.3951	0.98	PASS	0.5619	0.98	PASS
Runs	0.3838	0.98	PASS	0.4559	1	PASS
LongestRun	0.3416	0.98	PASS	0.7399	0.98	PASS
Rank	0.1223	0.98	PASS	0.3191	1	PASS
FFT	0.0887	0.98	PASS	0.4559	0.98	PASS
NonOver.Templ.	0.5467	0.99	PASS	0.4802	0.98	PASS
Over.Templ.	0.0157	1	PASS	0.6579	1	PASS
Universal	0.4351	0.95	PASS	0.3419	0.94	PASS
Approx.Entr.	0.6342	1	PASS	0.5124	1	PASS
Rand.Excurs.	0.0214	0.95	PASS	0.1285	0.98	PASS
Rand.Exc.Var.	0.0185	0.93	PASS	0.0856	1	PASS
Serial	0.7399	1	PASS	0.2133	1	PASS
LinearComplex.	0.4943	0.94	PASS	0.1223	0.94	PASS

Table 6.2: NIST test suite results for the HW device.

As shown in the table, all tests are passed. Moreover, it can be observed that the p -values vary slightly after conditioning and, on average, increase. This suggests that the ASCON block has a positive effect on the output bitstream, producing a more uniform distribution and reducing the bias of the resulting sequence, as expected from an implemented hash function.

Test	No whitening			Whitening (SHA-3)		
	P-value	Prop.	Result	P-value	Prop.	Result
Frequency	0.5749	1	PASS	0.4897	1	PASS
BlockFreq.	0.4944	0.96	PASS	0.8514	0.96	PASS
Cumulat.Sums	0.7131	1	PASS	0.6163	1	PASS
Runs	0.9357	0.96	PASS	0.5749	1	PASS
LongestRun	0.9114	1	PASS	0.4559	0.98	PASS
Rank	0.02354	0.98	PASS	0.0457	1	PASS
FFT	0.1089	1	PASS	0.8165	0.98	PASS
NonOver.Templ.	0.4968	0.98	PASS	0.5089	0.96	PASS
Over.Templ.	0.4190	1	PASS	0.9357	1	PASS
Universal	0.3761	0.98	PASS	0.4521	1	PASS
Approx.Entr.	0.4512	1	PASS	0.5682	0.94	PASS
Rand.Excurs.	0.6248	0.98	PASS	0.6047	0.98	PASS
Rand.Exc.Var.	0.5149	1	PASS	0.5864	1	PASS
Serial	0.9357	0.98	PASS	0.2897	0.96	PASS
LinearComplex.	0.3191	0.94	PASS	0.7792	0.98	PASS

Table 6.3: NIST test suite results for the bitsream generated, conditioning is software hash function SHA-3.

Test	No whitening			Whitening (ASCON)		
	P-value	Prop.	Result	P-value	Prop.	Result
Frequency	0.5749	1	PASS	0.8879	1	PASS
BlockFreq.	0.4944	0.96	PASS	0.3911	1	PASS
Cumulat.Sums	0.7131	1	PASS	0.6984	1	PASS
Runs	0.9357	0.96	PASS	0.8863	1	PASS
LongestRun	0.9114	1	PASS	0.8513	0.96	PASS
Rank	0.02354	0.98	PASS	0.2368	0.98	PASS
FFT	0.1089	1	PASS	0.0757	0.98	PASS
NonOver.Templ.	0.4968	0.98	PASS	0.5280	0.96	PASS
Over.Templ.	0.4190	1	PASS	0.3537	0.96	PASS
Universal	0.3761	0.98	PASS	0.4025	0.96	PASS
Approx.Entr.	0.4512	1	PASS	0.5227	1	PASS
Rand.Excurs.	0.6248	0.98	PASS	0.6023	1	PASS
Rand.Exc.Var.	0.5149	1	PASS	0.5894	0.98	PASS
Serial	0.9357	0.98	PASS	0.9558	0.98	PASS
LinearComplex.	0.3191	0.94	PASS	0.6993	0.98	PASS

Table 6.4: NIST test suite results for the bitsream generated, conditioning is software hash function ASCON.

Tables 6.3 and 6.4 provide an overview indicating that the hardware implementation of the conditioning stage is consistent with the expected behavior. The software simulation of the selected hash function reproduces the statistical and randomness properties that are also observed in the hardware implementation. Although no substantial difference can be identified with respect to SHA-3, which is a widely adopted standard hash function, and therefore the proposed work does not constitute a radically new alternative, the obtained statistical results are nevertheless fully consistent with the expected outcomes.

The graphs presented below further confirm that the proportion of sequences passing the tests remains within an acceptable range.

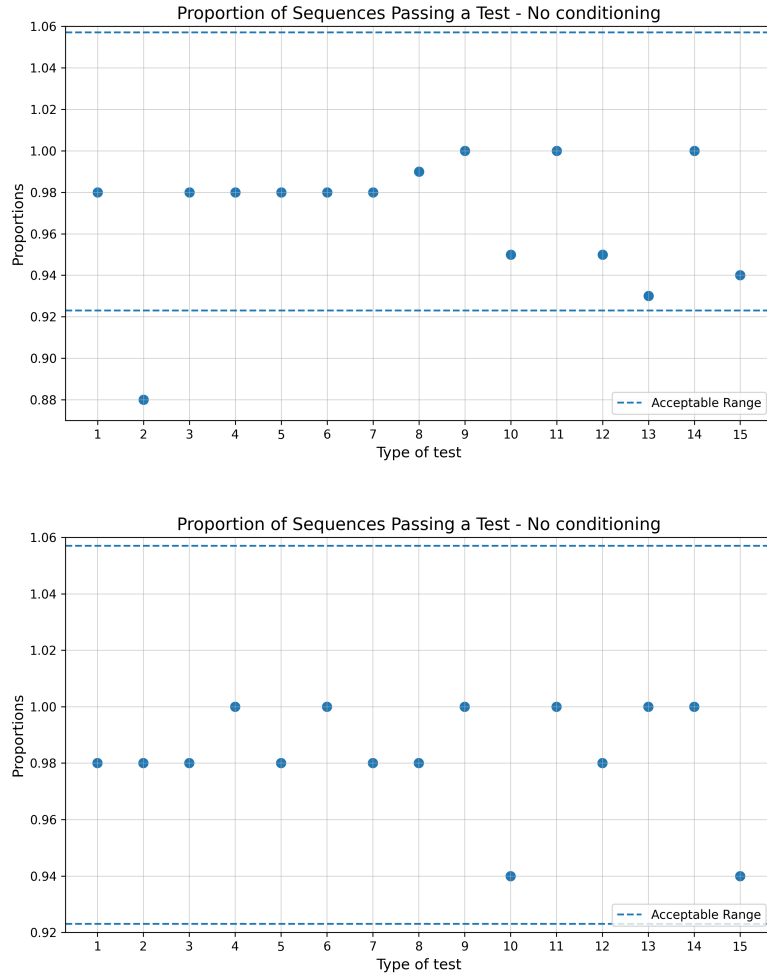


Figure 6.1: Proportions of sequences passing the 15 NIST tests without ASCON conditioning (top) and after ASCON conditioning (bottom).

6.2 FPGA Performance Analysis

To quantify the performance of the device implemented on the FPGA board, and to characterize it for comparison with similar works reported in the literature, a set of analyses was carried out using the tools compatible with the Cyclone 10GX0000477 device. These analyses were aimed at evaluating the maximum achievable operating frequency, throughput, area utilization, and power consumption.

6.2.1 Device Characterization

Initially, a timing analysis was performed using Quartus' Timing Analyzer tool in order to optimize the implemented architecture and the critical paths affecting system timing and latency, and to achieve the highest possible operating frequency while ensuring system stability and avoiding violations of the imposed timing constraints. These analyses were first carried out on the individual blocks, namely the TRNG without conditioning and the ASCON block, and subsequently on the complete system. The obtained results are reported in the following tables.

Once the maximum achievable frequency had been determined, the number of cycles required to complete one operation was estimated, making it possible to calculate the latency and throughput of the complete system, as well as the impact of the added conditioning block on these metrics.

So mathematically, the latency of the system was estimated as the time required to complete a single operation. It can be expressed as:

$$L = \frac{N_{\text{cycles}}}{f_{\text{max}}} \quad (6.1)$$

where N_{cycles} is the number of clock cycles required to complete the operation and f_{max} is the maximum achievable clock frequency.

The throughput was then calculated as the amount of useful output produced per unit of time. For a system generating N_{out} bits in N_{cycles} clock cycles, it is given by:

$$T = \frac{N_{\text{out}} \cdot f_{\text{max}}}{N_{\text{cycles}}} \quad (6.2)$$

This allowed a quantitative evaluation of the impact of the conditioning block on both latency and throughput of the complete architecture.

Then all the informations related to memory occupation, type of blocks used in the architecture and other PPA parameters were obtained by the reports generated after the compilation in quartus.

Device	$F_{\max}$ [MHz]	Area ALMs	Regs	Throughput [Mbps]
TRNG	151.75	1638	3678	127.37
ASCON block	162.70	4342	6031	297.51
Complete System	194.91	9210	26806	112.47

Table 6.5: Performance comparison of the implemented architecture's blocks.

Device performance parameters		
Parameter	Measured value	Notes
$F_{\max}$ (MHz)	194.91	Maximum achievable operating frequency of the complete architecture.
Latency (cycles)	139	Number of clock cycles required to complete one full operation.
Latency (μ s)	0.973	Execution time of one complete operation expressed in microseconds.
Throughput	112.47	Effective data processing rate of the complete system.
Min latency	0.713	Minimum observed latency over the analyzed operating conditions.
Max latency	0.973	Maximum observed latency over the analyzed operating conditions.
TRNG output rate (Mbps)	127.37	Raw random bit generation rate of the TRNG block.
TRNG alone $F_{\max}$	151.75	Maximum operating frequency of the TRNG block without conditioning.

Table 6.6: Device performance parameters for the TRNG plus ASCON complete block architecture.

The results in Tables 6.5 and 6.6 show a clear trade-off between clock frequency, area, and functional integration. The standalone TRNG achieves an $F_{\max}$ of 151.75 MHz with 1,638 ALMs and 3,678 registers, producing a raw throughput of 127.37 Mbps, while the standalone ASCON conditioning block reaches 162.70 MHz, with an area footprint of 4,342 ALMs and 6,031 registers, and a throughput of 297.51 Mbps.

When the two blocks are combined into the complete system, the architecture becomes substantially larger, with 9,210 ALMs and 26,806 registers, and the maximum operating frequency increases further to 194.91 MHz. This increase in $F_{\max}$ is consistent with the optimized integration of the final design, where control, buffering, and interconnection logic are arranged to sustain a faster clock than either standalone block.

At the same time, the complete system delivers the highest throughput, 112.47 Mbps, which reflects the performance of the full TRNG-ASCON data path rather than the raw output rate of the individual blocks. The latency figures further confirm this behavior: the complete architecture requires 139 cycles per operation, corresponding to 0.973 μ s, with the measured latency spanning from 0.713 μ s to 0.973 μ s across the analyzed operating conditions. Overall, the tables indicate that the complete TRNG-ASCON solution is not optimized for minimum area alone, but instead achieves a stronger operating frequency at the cost of a significantly larger resource footprint, while maintaining a well-defined system-level throughput.

Design	ALMs	Combinational ALUTs	Registers	M20Ks
TRNG block	1,637.5	1,965	3,628	549

Table 6.7: Area utilization of the standalone TRNG block.

Design	ALMs	Combinational ALUTs	Registers	M20Ks
Complete system	9,210	6,526	26,806	511

Table 6.8: Area utilization of the complete TRNG-ASCON system.

The comparison between the standalone TRNG and the complete system shows a very clear increase in resource usage once the block is integrated into the final architecture. The TRNG alone requires 1,638 ALMs and 3,678 registers, whereas the complete system reaches 9,210 ALMs and 26,806 registers, which is roughly 5.6 times more combinational logic and more than 7 times more sequential elements than the generator by itself.

Memory usage also increases substantially: the standalone TRNG occupies 9,114,112 block memory bits across 549 M20K blocks, corresponding to 76% of the device memory, while the complete system reaches 9,880,064 block memory bits across 511 M20K blocks, corresponding to 82% of the total available memory. This indicates that the TRNG block is already strongly memory-intensive, but the full integration mainly adds control logic, buffering, and interfacing overhead, which explains the sharp rise in ALM and register usage. In particular, the jump from 1,638 to 9,210 ALMs shows that the overall system complexity is driven not only by memory, but also by the additional logic required to manage the data flow and the interaction between the sub-blocks.

6.2.2 Comparative Tables

The outcome of the comparison of the results obtained for this work and other TRNGs implementations presented in recent literature is reported in Table 6.9. Only RO-based TRNGs have been considered and we tried to obtain significant parameters to confront with the ones collected for the device of this thesis; when this was not possible it is indicated in the table.

Work	Shann.	FPGA	Area	Area	Thrp. [Mbps]	Freq. [MHz]
	Entr.	device	LUTs	FFs		
	0.9999	Artix7	24	33	275.8	275.8
	0.9999	Spartan3	528	177	6	24
	–	Zynq7	65	119	1600	100
	0.9999	Stratix4	288	190	400	100
	0.9993	Artix7	40	29	1.91	–
	0.997	Spartan6	16	11	1.15	100
This Work	0.9965	Cyclone10	1638*	3678*	112.47	194.91

Table 6.9: Comparison of different TRNG implementations.

The asterisks in the *Area* columns indicate that, for this work, the reported resources are given in terms of Intel Quartus **ALMs** and dedicated registers rather

than the LUT/FF representation commonly used in the Xilinx-oriented literature. This distinction is important because an ALM is not directly equivalent to a single LUT: in Intel Cyclone 10 GX devices, each ALM is a composite logic element that includes LUT-based combinational resources together with embedded registers and carry-chain support. In this Table the work depicted are the ones cited in the references at fonts [13], [6], [11], [14], [10] and [19] respectively.

For this reason, the values reported for this work, namely 1,638 ALMs and 3,678 registers, should be interpreted as the native Quartus area metrics of the implemented design, whereas the LUT and FF figures reported in the other references correspond to a different FPGA architecture and synthesis reporting style. The asterisk therefore serves as a reminder that the comparison is architectural rather than one-to-one at the resource primitive level, even though it remains meaningful for a high-level assessment of area efficiency.

From an overall perspective, the table highlights that the proposed Cyclone 10 GX implementation achieves a competitive Shannon entropy of 0.9965 while also providing a markedly higher operating frequency of 194.91 MHz than most of the RO-based TRNG implementations considered.

At the same time, its throughput of 112.47 Mbps places it above the low-throughput designs reported in the table, although below the most aggressive implementations targeting very high throughput. The comparison is restricted to *ring-oscillator-based TRNGs* in order to ensure that the architectures remain conceptually comparable, since different entropy sources and extraction mechanisms would otherwise make the performance and area figures difficult to interpret in a consistent way.

Overall, the results suggest that the presented design offers a balanced trade-off between entropy quality, frequency, and implementation cost within the RO-based TRNG category.

6.3 Possible Improvements and Future Developments

The proposed TRNG already shows encouraging results; however, several possible directions for future improvement remain open. A first and particularly relevant extension would be the implementation of **health tests** directly in hardware, so that the entropy source can be monitored in real time during normal operation. This would allow abnormal conditions, degradation of the noise source, or unexpected bias to be detected with minimal latency, improving the overall robustness of the generator.

Compared with a software-only approach, an on-chip solution would also reduce the dependence on external monitoring logic and would make the architecture more suitable for continuous use in embedded and security-critical contexts. In addition, the conditioning stage could be further optimized by evaluating alternative hash-based solutions or by refining the existing implementation in terms of area, latency, and power consumption. Such an analysis would make it possible to identify the most effective trade-off between diffusion quality and hardware cost, while preserving the statistical properties required by the validation process. Moreover, it would also be possible to design a system in which multiple conditioning blocks coexist and can be selected for different applications according to their specific constraints. Another possible improvement would be to introduce a more configurable architecture, capable of supporting different operating modes or conditioning strategies, so that the device could be more easily adapted to future experimental setups and application requirements.

From an application perspective, the proposed TRNG could also be integrated into a **Hardware Security Module (HSM)** for instance, where high-quality randomness is essential for the generation of cryptographic keys, nonces, seeds, and other internal security parameters. In this scenario, the TRNG would not be considered as a standalone block, but rather as a fundamental building element of the overall security chain, contributing directly to the reliability and trustworthiness of the module. The presence of a validated entropy source, possibly combined with hardware health monitoring, would strengthen the resilience of the HSM against faults, entropy loss, and other unpredictable operating conditions. This makes the proposed solution particularly interesting for secure embedded platforms and protected cryptographic devices, where both randomness quality and runtime dependability are essential. For these reasons, the integration of the proposed TRNG into an HSM represents a natural and promising direction for future development.

Chapter 7

Conclusions

The generation of robust and unpredictable random bitstreams is a fundamental requirement in cryptographic applications, where the quality of the produced data directly affects the security of the overall system. This work presents a hardware implementation of a true random number generator on an Intel Cyclone 10 GX FPGA platform, together with a lightweight and efficient conditioning stage based on the ASCON hash function. The proposed architecture is deliberately simple from a design perspective, yet it satisfies the essential requirements expected from a cryptographic random source, namely statistical robustness, acceptable performance, and a controlled hardware footprint. The experimental results show that the generated bitstreams pass the selected statistical tests and exhibit a high level of Shannon entropy per bit, confirming the effectiveness of the proposed approach. In particular, the inclusion of the ASCON block improves the uniformity of the output stream while preserving a compact and practical implementation. Although additional work is still required to refine the architecture and investigate further optimizations, the proposed solution already constitutes a solid starting point for future developments. Its simplicity, efficiency, and cryptographic relevance make it potentially valuable in a wide range of security-oriented applications, where a reliable hardware source of randomness is a key enabling component.

References

- [1] Analog Devices. *Pseudo Random Number Generation Using Linear Feedback Shift Registers*. Design note. URL: <https://www.analog.com/en/resources/design-notes/random-number-generation-using-lfsr.html>.
- [2] Mathieu Degré et al. “New Models for the Cryptanalysis of ASCON”. In: *IACR Cryptology ePrint Archive* 2024.298 (2024). URL: <https://ia.cr/2024/298>.
- [3] Xiaoyang Dong et al. “Generic MitM Attack Frameworks on Sponge Constructions”. In: *Advances in Cryptology – CRYPTO 2024*. 2024. DOI: 10.1007/978-3-031-68385-5_1. URL: https://doi.org/10.1007/978-3-031-68385-5_1.
- [4] Viktor Fischer and Miloš Drutarovský. “True Random Number Generator Embedded in Reconfigurable Hardware”. In: *Cryptographic Hardware and Embedded Systems – CHES 2002*. Vol. 2523. Lecture Notes in Computer Science. 2002, pp. 415–430. ISBN: 978-3-540-00409-7. DOI: 10.1007/3-540-36400-5_30. URL: https://doi.org/10.1007/3-540-36400-5_30.
- [5] Viktor Fischer et al. “Enhancing Quality and Security of the PLL-TRNG”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.4 (2023), pp. 211–237. DOI: 10.46586/tches.v2023.i4.211-237. URL: <https://tches.iacr.org/index.php/TCHES/article/view/11164>.
- [6] Fabio Frustaci et al. “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. URL: <https://doi.org/10.1109/TCSII.2022.3211278>.
- [7] Charles Herder et al. “Physical Unclonable Functions and Applications: A Tutorial”. In: *Proceedings of the IEEE* 102.8 (2014), pp. 1126–1141. DOI: 10.1109/JPROC.2014.2320516. URL: <https://doi.org/10.1109/JPROC.2014.2320516>.
- [8] Intel Corporation. *Intel textsuperscript Cyclone textsuperscript 10 GX Device Datasheet*. Device datasheet. Feb. 2022. URL: <https://www.intel.com/content/www/us/en/docs/programmable/683828/current/device-datasheet.html>.
- [9] Intel Corporation. *Quartus textsuperscript Prime Pro Edition Version 24.1 Software and Device Support Release Notes*. Release notes and software support information. Jan. 2024. URL: <https://www.intel.com/content/www/us/en/docs/programmable/683706/24-1/version-software-and-device-support.html>.

- [10] Anju P. Johnson, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. “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. URL: <https://doi.org/10.1109/TCSII.2016.2566262>.
- [11] Annapurna Kamadi and Zia Abbas. “Implementation of TRNG with SHA-3 for hardware security”. In: *Microelectronics Journal* 123 (2022), p. 105410. DOI: 10.1016/j.mejo.2022.105410.
- [12] Dharendra Kumar et al. “Design of Energy Efficient True Random Number Generator using MUX-Metastable Approach”. In: *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*. 2021, pp. 446–451. DOI: 10.1109/SPIN52536.2021.9566149.
- [13] Zhaojun Lu et al. “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. URL: <https://doi.org/10.1109/DAC56929.2023.10247746>.
- [14] Pietro Nannipieri et al. “True Random Number Generator Based on Fibonacci-Galois Ring Oscillators for FPGA”. In: *Applied Sciences* 11.8 (2021), p. 3330. DOI: 10.3390/app11083330. URL: <https://www.mdpi.com/2076-3417/11/8/3330>.
- [15] Andrew Rukhin et al. *A Statistical Test Suite for Random and Pseudo-random Number Generators for Cryptographic Applications*. NIST Special Publication 800-22 Rev. 1a. National Institute of Standards and Technology, Apr. 2010. DOI: 10.6028/NIST.SP.800-22r1a. URL: <https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf>.
- [16] Guido Di Patrizio Stanchieri et al. “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. URL: <https://doi.org/10.1109/ISCAS.2018.8351376>.
- [17] Mario Stipčević and Çetin Kaya Koç. “True Random Number Generators”. In: *Open Problems in Mathematics and Computational Science*. Springer, 2014, pp. 99–124. DOI: 10.1007/978-1-4614-6063-1_5. URL: https://doi.org/10.1007/978-1-4614-6063-1_5.
- [18] Meltem Sönmez Turan et al. *Ascon-Based Lightweight Cryptography Standards for Constrained Devices: Authenticated Encryption, Hash, and Extendable Output Functions*. NIST Special Publication 800-232. National Institute of Standards and Technology, 2025. DOI: 10.6028/NIST.SP.800-232. URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-232.pdf>.

- [19] Bohan Yang et al. “ES-TRNG: A High-throughput, Low-area True Random Number Generator based on Edge Sampling”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018), pp. 267–292. DOI: 10.13154/tches.v2018.i3.267–292. URL: <https://doi.org/10.13154/tches.v2018.i3.267–292>.