Universidad Carlos III de Madrid
Escuela Politécnica Superior

Electronic Technology Department

BACHELOR'S DEGREE IN TELECOMMUNICATION TECHNOLOGIES

Final Degree Project

Implementation and Study of a True Random Number Generator

Author: Elena Martínez López
Tutor: Honorio Martín González

Leganés, September 2015

Title: Implementation and Study of a True Random Number Generator
Autor: Elena Martínez López
Tutor: Honorio Martín González

THE TRIBUNAL

President: Ignacio Soto Campos

Vocal: Pablo Acedo Gallardo

Secretary: Jose Luis González de Suso Molinero

Substitute: Pedro Martín Mateos

Finished the defense and lecture of the Final Degree Project on the ..... of ..... of ... in ....., in the Polytechnic School of Universidad Carlos III de Madrid, accord to GRADE it:

.....

VOCAL

SECRETARY PRESIDENT

Acknowledgements

Thanks to my parents and friends who have supported and encouraged me to get to where I am today.

Special gratitude goes also to my tutor whose advice and assistance were key to complete this project.

Contents

List of Figures

List of Tables

Abstract

Securing information has been a concern throughout history. Especially nowadays since many user applications such as smart cards or Internet connections deal with sensible data. To protect this information different cryptography protocols are used. These are algorithms that encapsulate the data by ciphering it. However, this is done by programming an application to run a digital mathematical function. This means that it is also possible to program malign applications to decode the cipher. In order to avoid this it is necessary to add unpredictability or randomness to the encoding process which can be done by employing a Random Number Generator.

A RNG can be implemented in both software and hardware; however, a truly unpredictable sequence is not achieved through a digital process governed by mathematical formulae. This results in most RNGs producing a form of pseudo-randomness. A True Random Number Generator must be implemented on a technology that allows it to harvest entropy from an unpredictable or even chaotic physical process. This is why TRNGs are designed and implemented for hardware. In fact, it is possible to gather entropy through integrated circuits like ASICs or FPGAs. The objective of this project is to design and implement a TRNG on FPGA technology because its pre-defined logic blocks that only require a small amount of resources make it an appealing solution.

First, an analysis of typical RNG designs is presented to understand the between a pseudo-RNG and a TRNG. Once this is stablished, the specific ways of designing TRNGs for integrated circuits are delved into. Moreover, the need for evaluation of the quality of randomness is also stated. This is ensured by a battery of tests that study the statistical properties of the output of a RNG.

Secondly, the TRNG design proposals by Böhl on which this project is based on are introduced and analyzed before creating the design and implementation. Afterwards, the four experiments per-

formed are explained. It was decided to first test the behavior of the TRNG at different frequencies to decide which provided randomness with the best quality. Afterwards, the TRNG was placed in different areas of the FPGA at the optimal frequency to test the variability of the device. A third experiment consisted of comparing these results in more devices to further study the variability. The final experiment consisted on forcing a reset of the circuit to ensure that the TRNG was resilient against this type of attacks.

Last but not least, the results are summarized and several future developments are presented. After this the legal aspects and management of the project are explained.

Keywords: random number generator, true random number generator, FPGA, hardware security.

Abstract

La protección de información ha sido una constante preocupación a lo largo de la historia. Especialmente hoy en día debido a las muchas aplicaciones que manejan datos confidenciales como tarjetas inteligentes o conexiones a Internet. Para proteger esta información diferentes protocolos criptográficos son usados. Estos son algoritmos que cifran los datos para encapsularlos. Sin embargo, esto se hace programando una aplicación que corre una formula matemática digital. Esto significa que también es posible programar aplicaciones maliciosas para decodificar el cifrado. Para poder evitar esto es necesario añadir aleatoriedad o un elemento impredecible al proceso de codificación. Esto puede hacerse empleando un Generador de Números Aleatorios cuyas siglas en inglés son RNG.

Es posible implementar un RNG tanto en software como en hardware; sin embargo, una secuencia realmente impredecible no se puede generar a través de un proceso digital basado en la computación de fórmulas matemáticas. Esto es lo que hace que la mayoría de RNGs produzcan una especie de pseudo-aleatoriedad. Un Generador de Números Realmente Aleatorios (True Random Number Generator o TRNG) debe ser implementado en una tecnología que le permita extraer entropía de un proceso físico impredecible o caótico. Es por esto que los TRNG se implementan en hardware. De hecho, es posible obtener entropía a través de circuitos integrados como ASICs o FPGAs. El objetivo de este proyecto es diseñar e implementar un TRNG en tecnología FPGA puesto que sus bloques lógicos predefinidos que solo necesitan unos recursos reducidos la convierten en una solución atractiva.

Se empieza por presentar un análisis de los diseños de RNG típicos para comprender la diferencia entre generadores pseudo aleatorios y TRNGs. Tras esto, se especifica la forma en la que los TRNGs se diseñan para circuitos integrados. Además, se expone la necesidad de evaluar la calidad de la aleatoriedad que se genera. Esta se comprueba a través de una batería de tests que estudian las propiedades

estadísticas del output del TRNG.

A continuación, las propuestas de diseño de TRNGs de Böhl en las que este proyecto se basa son introducidas y analizadas seguidas del diseño e implementación propios. Tras lo cual se explican los cuatro experimentos realizados. Primero se decidió comprobar el comportamiento del TRNG a diferentes frecuencias con el fin de determinar a cuál de ellas se producía la aleatoriedad de mayor calidad. Segundo, el TRNG fue posicionado en diferentes áreas de la FPGA a la frecuencia óptima para evaluar la variabilidad de la placa. El tercer experimento explora aún más la variabilidad al realizar el experimento anterior en otras placas. El último experimento consistió en forzar un reset del circuito para comprobar la resistencia TRNG ante ataque de este tipo.

Finalmente, los resultados obtenidos se presentan resumidos junto con varias propuestas de mejoras futuras. Tras ello se muestran los aspectos legales del proyecto y su gestión.

Keywords: generador de numeros aleatorios, generador de numeros realmente aleatorios, FPGA, seguridad hardware.

Summary

Throughout history many different forms of protecting sensible information have been designed but nowadays the most commonly used are cryptographic protocols. These protocols consist of algorithms that attempt to encapsulate the data by ciphering it. However, the algorithms are digital functions computed by a machine, therefore, the protocol is as reliable as the computer that generated it. If a software application was pre-programmed to encode a secure environment, another malign application can be programmed to decode it. An extra layer of security that an external application would find unpredictable is required. This can be achieved by employing a Random Number Generator.

There are several ways to create random numbers but the most desirable is to design a generator that produces an uniformly distributed sequence in which there is no discernible pattern. In other words, any element taken from it must be statistically independent from the rest. This can be implemented in both hardware and software but, again, the mathematical formulae on which digital processes are based on are not reliable to produce a truly unpredictable sequence. This is why many RNGs have a deterministic component that results in the generation of a sequence that can only be considered pseudo-random.

True randomness originates from the entropy of an aleatory or chaotic physical phenomenon such as noise or nuclear decay. A True RNG will harvest this entropy and handle it to generate a sequence of numbers. In order to do this a hardware device is required. Despite this seemingly complicated process, many small-sized devices can implement a TRNG such as integrated circuits like ASICs or FPGAs. For this project FPGA boards seemed an appealing solution because not only they are highly affordable but also because their resource consumption is reduced by the use of pre-defined logic blocks.

In FPGA devices, there are several sources from which randomness can be extracted. First, through analog peripherals integrated circuits can gather entropy directly from a chaotic event, however most models do not include such technology. Second, there is metastability which refers to unpredictable voltage oscillations of flip-flops when their hold time and setup times are violated. However, FPGA technology is designed so that these effects can be reduced. Third, using SRAM memories to either simulate metastability or through its start-up state that generates non-deterministic noise. Fourth, Open Delay Chains that XOR many delay stages originated by D-latches. Last but not least, jittery clock that deviate from the ideal behavior of a signal. This has proven to be the best method for FPGAs since it only requires simple components that can be found in any type of FPGA.

There are two ways to exploit jitter in RNGs: coherent sampling and jittery clock sampling. In coherent clock sampling several jittering clocks are necessary. A jittery clock takes samples from another one, since none of them are ideal the samples results in a random sequence. Meanwhile, jittery clock sampling takes samples from oscillators. An oscillator can either be a Ring Oscillator, a feedback loop of an odd number of oscillators, or a Self-Timed Ring, a loop of adjacent stages.

It is very likely that the quality of the randomness generated by any of these means may not be perfect. The output of any TRNG must be analyzed by any of the official batteries of statistical tests dedicated to estimate whether the randomness is truly unpredictable and uniformly distributed or not. If a RNG does not pass these tests it means that it has a deterministic component. A module can be added to the design of the TRNG to correct biases in the numbers generated: a post-processing mechanism. The two most common are: the XOR corrector which simply compresses the sequence by performing an XOR operation of n consecutive bits and the von Neumann corrector that takes pairs of bits and either returns the first bit if they are different and discards them otherwise. There are other mechanisms such as hash and resilient functions, linear feedback registers, encrypting the noise signal, etc.

The TRNG implemented and studied in this project is based on Böhl's proposal for a fault attack robust generator. It consists of a Ring Oscillator of nine inverting components: an initial NAND gate and eight inverters. The random sequence is obtained by sampling at three points of the ring through D flip-flops. Each of these are placed after three inverting elements.

FPGAs, such as the Xilinx Spartan-3E board used, reduce the difficulties in designing circuits by routing the signals and elements in the hardware automatically. Nevertheless, this means that for each measurement that would be taken there is a chance that the hardware would be routed differently. Thus, it would be impossible to guess if the quality results obtained each time a measurement would be taken would depend on the variables controlled during the experiments or because a different routing was more or less effective. To avoid this ambiguous situation, it was decided to turn the TRNG into a Hard Macro. In other words, custom analog block surrounded by digital logic that fixes the components and routing it requires in every implementation.

It was found after a first attempt at evaluating this TRNG that a post-processing mechanism is required to correct its weaknesses. An XOR corrector has been chosen for this purpose. It was desirable to test the performance the post-processor at different lengths: 3-bit, 5-bit, 7-bit and 9-bit. Originally, this was going to be implemented on the hardware. However, it was soon realized that the time that would be wasted in doing so and capturing the data afterwards would not be advantageous. Instead, the corrector was implemented as a digital Python script which greatly reduced the time spent in post-processing. The script treats the random sequence captured by XORing as many bits as specified. As such, it only took a couple of minutes to post-process the sequence with the four XOR correctors.

At the beginning of the project, due to the time constraints, only two experiments had been scheduled but because of the Python optimization another two experiments could be performed. The four experiments are: an evaluation of the TRNG at different frequencies of oscillation, an intradevice test, an interdevice test and a forced restart of the oscillations. The four of them as explained in the following paragraphs.

The experiment is the evaluation at different frequencies. In the reference proposal, it was declared that the frequency of oscillation at which the TRNG produces optimal results is 1MHz. It was decided to prove or disprove this by measuring the random sequence produced by the TRNG in a fixed position of the FPGA at several oscillation frequencies: 50KHz, 100KHz, 500KHz, 1MHz, 25MHz and 50MHz. The XOR correctors of 3 and 5 bits did not produce satisfactory results as they failed many statistical tests. Nevertheless, at 7 bits the post-processing started to correct the bias effectively, although the best results were obtained with a 9-bit XOR. It was found that at higher frequencies there is not enough time for

jitter accumulation which reduces the quality of randomness. Meanwhile, while at low frequencies the time for accumulation is greater, at some point the penalty in throughput becomes too high also affecting the performance of the TRNG negatively. It was therefore proven that at 1MHz the best quality was produced.

The second experiment consists of an intradevice test. This time, using the best frequency of oscillation (1MHz) and the best post-processing (9-bits) the quality of the TRNG is evaluated in different locations of the FPGA board. The objective of this is to test the hardware variability. Nine positions were evaluated: the center of the board, the four outer corners and the four inner corners. The best results were obtained at the center. The reason for this is that, despite the Hard Macro, the connections in surrounding components can affect the jitter. At the center the board has more space to place other elements further away from the TRNG so they would not affect its performance.

The third experiment consists of an interdevice test to further test the hardware variability. This has been done by evaluating the performance of the TRNG in several devices under the established variables: the optimal position, frequency of oscillation and post-processing. It is found that the four boards passed the same number of statistical test but an oddity occurred: the quality of results of the three the newly acquired FPGAs were significantly worse. It has been deduced that these measurements were taken during a heat wave which indicates that the TRNG is not resilient against environmental changes.

Lastly a restart experiment was performed. This consisted on bringing the oscillations to a halt by force and restarting them in order to evaluate whether or not the TRNG could still generate a robust random output. The initial NAND gate of the oscillator is fed by a reset signal. If this signal becomes zero for some reason while the TRNG is running the oscillations will stop. This reset is directly connected to a trigger of the FPGA board. All that is needed to restart the circuit is to pull down this trigger and immediately pull it up again. After many restarts were performed sequence generated was analyzed. No patterns were found that would indicate that the sequence produced after a restart could be guessed. Therefore it has been concluded that the TRNG is secure against this type of attack.

To sum up, it has been found that for this TRNG on a Spartan-3E board, 1MHz is the optimal frequency of oscillation but a post-processing mechanism is required. An XOR corrector of 9-bits yields

the best quality. The TRNG can be implemented anywhere within the FPGA but the best results are obtained at a centered position. Moreover, the TRNG is resilient against restart attacks but not to environmental changes.

There have been previous attempts at implementing this proposal of a TRNG in several technologies. These found that it was robust with a 3-bit XOR corrector and also fault and attack resilient. From this it can be deduced that the quality and weaknesses of this implementation may be caused by the variability of the FPGA model chosen that could be influencing the Ring Oscillator negatively. It would be desirable to repeat these experiments under a controlled environment in other FPGA technology in the future to be able to assure that the Xilinx Spartan-3E is to blame.

Nevertheless, since the TRNG was proven to generate a good quality output given the appropriate conditions, several future developments could be implemented to improve the quality of the output of the TRNG.

Originally, in his proposal, Böhl suggested that the random output of the TRNG could be built in another way. Instead of directly outputting the three bits stored in the three DFFs consecutively, the three of them could be XORed. The actual random sequence would be the output of this last operation. The first modification to the design would have to be testing the quality of the design slightly modified in this way.

The next possible modification to the design would be to add more inverting elements to the ring oscillator. The randomness would have higher quality because the more stages the more jitter that would accumulate. This change to the design would probably change the optimal frequency of oscillation so the first experiment would need to be re-evaluated. There would be two possible ways of modifying the design according to this: first, add more inverters before each sampling stage since it could be assumed that the weaknesses could be appearing because three inverters before sampling are not enough; second, add more sampling stages preceded by three inverters although this would most likely drain more resources. In case any of these two possibilities were implemented, it would be recommended to evaluate the quality of the output first from just outputting the sampled bits and afterwards by XORing them before generating the sequence.

Another completely different approach would be to evaluate other post-processing mechanisms in hopes that there might be a better

correction of bias such as a Linear Feedback Shift Register, the von Neumann algorithm or a hash or resilient function. However, it would also be interesting to test the quality of the current XOR corrector with the previous proposals either using a new technology or by changing the current design.

A last consideration would be to compare the results obtained by this TRNG with a different type of feedback loop. The Ring Oscillator of inverting elements would be replaced by a Self-Timed Ring since that kind of implementation has been proven to be more resistant to variabilities and generate jitter with higher quality.

In conclusion, the TRNG implemented did manage to produce a considerably reliable random sequence given the appropriate post-process was appended. In spite of this, the TRNG is not without its faults and weaknesses which may have been caused by the hardware limitations. It is recommended to repeat these experiments in several FPGAs but it would also be possible that by slightly modifying the current design the deficiencies could be corrected.

Chapter 1

Introduction

Sensible data and information has always been secured employing a wide variety of ways all through history. In fact, nowadays many applications use some form of security: smart cards, Internet routers, home computers and all sorts of mobile and wireless devices. All of these applications demand that the information is handled and stored in a secure way. Modern cryptography attempts to serve this purpose through codes that cipher the data and the transmission channel. However, cryptographic protocols are usually generated by software as a digital function which is not enough to store the data securely, to achieve it, hardware security is required.

Security in hardware usually consists on the use of circuits or other devices that add the conditions necessary for the application to run over a secure environment. In the case of embedded systems, random number generation is a very useful process for addressing this issue. There are several ways to create random numbers but the most desirable is to design a generator that produces a sequence that cannot be guessed because it is uniformly distributed and unpredictable, in other words, a generator that can achieve true randomness.

1.1 Motivation

The motivation of this project is to study how TRNGs can be designed and implemented on integrated circuits and particularly on Field Programmable Gate Arrays.

Random numbers are used in many different types of applications from gambling to data encryption and more. In cryptographic ap-

plications, it is very beneficial to have a generator that can produce a truly random sequence to cipher and/or secure data. As such, many Random Number Generators have been designed in software and hardware.

A sequence is considered truly random when it cannot be predicted. However, especially in software RNGs this is difficult to achieve because they create a sequence according a pre-programmed mathematical formula therefore only obtaining a form of pseudo-randomness. True randomness is not accomplished through an algorithm but it is produced as a result of a physical process such as noise or nuclear decay. True random number generators are implemented on hardware devices that have the ability to harvest entropy from these processes.

Despite the apparent difficulty that gathering randomness presents, it is possible to do so through very small integrated circuitry like ASICs or FPGAs. ASICs are very advantageous since they can integrate tailored blocks including many sorts of analog elements. However, tuning all the parameters and components that the generator requires can make implementing a TRNG on ASICs quite expensive. In contrast, despite their constrained resources FPGAs are a very popular hardware on which to implement TRNGs due to their pre-defined logic blocks which greatly reduces their price.

1.2 Objectives

The main objective of this project is to implement a TRNG on integrated circuitry and study its behavior. More specifically, this project tests the behavior of a TRNG based on Böhl's attractive proposal in [1] on a FPGA board.

According to [1], the a jittery clock such as the one included in FPGAs can be used at an oscillation of 1MHz to produce true randomness but it also states that a post-processing technique is required to enhance entropy. In [2], another paper based in [1], a simple post process is used: an XOR operation of 3 bits. This is tested on a couple of different hardware solutions. One of them is a FPGA which is a very interesting type of hardware it allows to place the TRNG almost anywhere within it. However, this feature is not tested in [2] as it only provides data based on frequency samples. Another important reason to try to emulate their approach of implementing the TRNG in FPGA technology is that, FPGAs also introduce the great advantage of reducing costs in materials.

There are several things that it was desirable to demonstrate as they were not delved into by neither [1] or [2]. The experiments that were decided to be carried out are:

1. The reliability of the TRNG at several frequencies and prove or disprove that the optimal frequency of oscillation is 1MHz.
2. Expand on the use of the TRNG on FPGA technology and check if its placement in different areas of the same hardware affects its performance.
3. Test its output with and without post processing.
4. Post process of the output with a XOR operation of not just 3 bits but also several more bits to test how this affects randomness.

All of these experiments will be performed by passing the data captured through the Statistical Test Suite provided by the National Institute of Standards and Technology. This battery of tests has been especifically designed to examine the quality of a random sequence. [3].

1.3 Structure of this Document

This document is structured as follows:

- **Chapter 1** provides the basic introductory information to the project that will be discussed in later chapters.
- **Chapter 2** expands the theory introduced in the previous chapter as well as providing information on the technology used.
- **Chapter 3** is an in depth explanation of the solution created to implement and design the True Random Number Generator and how it has been tested.
- **Chapter 4** presents the final conclusions that have been reached after the results of the tests have been analyzed and provides future possible developments.
- **Chapter 5** exposes the legal conditions that the project is required to meet.

- **Chapter 6** includes a initial economic management of the project as well as its estimated duration of the project along with the final real money and time spent.

Chapter 2

State of the Art

In this chapter we present an overview of the theory of random number generation as well as the hardware technology that is used in the project to this end.

2.1 Security and Hardware

The creation of secure environments has always been an issue throughout all the history of humanity. The greatest advance in security occurred with the discovery of electricity which led to a new form of security: hardware security. Its first incarnation was introduced in 1853 with the creation of electro-magnetic alarms used to protect businesses and residences. By the end of the 19^{th} century this groundbreaking technology had advanced to be able to electronically controlled vaults [4].

However, protecting physical property is not the only use for hardware security. Securing sensible information has been an issue dating back to ancient times. Cryptography fulfilled this necessity through the use of codes and ciphers generated in a manual process. In contrast, modern cryptography focuses not only on coding the data to be transmitted but also on creating a secure transmission channel. The first instances of these new methods of encryption appeared in military applications by the end of the 19^{th} century but they became a major security measure during World War II with the creation of the German Enigma rotor machine. Breaking the code of this and other encryption machines also became a military priority which resulted in a constant flow of revolutionary hardware security techniques such as *computing*. It was only a matter of time that these technological advances were applied to civil applications lead-

ing to the first computing mainframes in the 1960s, supercomputers in the 1960s and finally personal microcomputers in the 1970s. [5] [6]

Nowadays, hardware security can be found practically every aspect of our daily lives as it is necessary in many consumer applications such as: ID-cards, Internet routers, sim cards, smart cards, game controllers, car electronics. All these applications require hardware security not only to handle information in a secure way but also to store it anywhere at any time with small costs in energy and resources. Modern cryptographic protocols are generally a series of mathematical digital functions that must be made resistant to algorithmic attacks. These protocols generally rely on private or session keys to secure data before exchanging it. An attacker would attempt to eavesdrop on the transmission and guess the key. It is necessary to make the channel resilient against malicious listeners but, in case that line of protection failed, the keys must also be very hard to decode. One way to ensure this is to employ a Random Number Generator: a device that can generate a key so complicated so that an attacker cannot guess it or even produce a sequence to envelop the channel and protect the data to be transmitted from eavesdroppers.

RNGs have been implemented in several types of technologies. Modern integrated circuits (IC), such as Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs), are very popular technology to implement RNGs precisely because they are highly affordable and they only require a small amount of resources.

2.1.1 Random Number Generators

Generating random numbers has always been required in many situations. In ancient times, divination and card or dice games were the main areas where randomness was applied which entailed a very elaborated and difficult calculation process. However, due to the processing capacity of computational or physical devices nowadays random number generation has radically changed. New applications for it have also appeared such as new forms of massive national and international gambling games as well as many security measures such as producing secure data encryption keys.

These modern Random Number Generators (RNGs) in order to be considered secure and trustworthy are expected to produce a se-

quence of numbers in which there is no pattern: any element taken from it must be statistically independent from the rest. [7] There exist statistical tests for randomness to study how successfully a RNG produces an indiscernible pattern. However, many generators that pass them do not actually produce a truly random sequence. In computer applications this is due to the fact that, after all, since a computer simply follows the commands that have been given to it, its behavior is predictable. Hardware RNGs pose a better solution over software applications since they are based on a, usually, unpredictable physical phenomenon although that does not mean that they are without fail.

Therefore, there exist two kinds of RNGs [8]:

- Pseudo-random Number Generators based on a deterministic bit generation process.
- True Random Number Generators based on non-determinisctic random bit generation.

In the following sections of this document these RNGs are explained in greater detail.

2.2 Pseudo-random Number Generators

A Pseudo-random Number Generator or PRNG consists of a deterministic mathematical algorithm that produces output sequences of numbers that appear to be random. This is because all of them have the same probability of appearing and can satisfy statistical tests. However, this output, also known as a unbiased random output, is not truly random. The output is obtained through a pre-calculated list or mathematical formulae that contains all possible values the PRNG can take. In order to simulate randomness, this list is traversed at an unknown starting point known as 'seed' which is fed to the generator [9] [10]. This quality means they can generate long sequences of numbers quite fast which makes them very efficient but it is also what makes them deterministic, any time you feed the same seed to the PRNG it will return the same output sequence. It can also be deduced from this that PRNGs are generally periodic, while this at first may seem like a hindrance to randomness it should be taken into account that the period would be so long that it would not be noticed in most applications in which PRNGs are used. Moreover, there may be certain applications in which it might

be useful to repeat a previous sequence easily such as in simulation and modelling [7].

There are many PRNGs being used but some of the most well-known are:

- **Mersenne twister** is the most widely used algorithm in PRNGs for software systems such as Python, PHP, MATLAB, Maple, Apache, etc. It was developed in 1997 and was designed to correct the many flaws in previous PRNGs by generating pseudo-random integers with speed and high-quality. Although there is a 64-bit word length implementation, the standard is to produce a sequence of a 32-bit integers based on the Mersenne prime that produces a period of $2^{19937} - 1$. In any case, for a w-word length, integers are generated in a range of $[0, 2^w - 1]$ which is k-distributed to v-bit accuracy by the following equation: $(trunc_v(X_i), trunc_v(X_{i+1}), ..., trunc_v(X_{i+k-1}))$ $(0 < i < P)$ where P is the period of a kv-vector and $trunc_v(x)$ denotes the first v bits of x, each of the 2^{kv} combinations occur the same number of times during a period.[11]

- **Blum-Blum Shub** generates a bit sequence according to the formula: $X_{n+1} = X_n^2 mod_M$ where M is the product of two very large secret and distinct prime numbers each congruent to 3 modulo 4. X_0 is calculated through the seed which is in the random interval $[1, n-1]$ and must meet the condition $gcd(seed, n) = 1$, thus $X_0 = seed^2 mod_M$ [9]

- **Linear Congruential Generator (LCG)** is one of the oldest PRNGs. It is based on the recurrence relation: $X_{n+1} = (aX_n + c) mod_m$ in which a, c, m and the seed X_0 are constants that, if selected properly can make the PRNG a maximal period generator. LCGs have been proven to be quite efficient when m is selected to be a power of 2, generally 2^{32} or 2^{64}. However, since X_n and X_{n+1} are not independent, the sequence generated could be guessed by an attacker, therefore, LCGs are not a type of CSPRNG and cannot be used in cryptographic applications. [10]

- **Linear Feedback Shift Register (LFSR)** is a shift register which is fed an input bit that is actually a linear function of its previous state. As a PRNG this feedback function consists of a combination of XOR gates as can be seen in figure 2.1 which results in: $(C(x) = 1 + c_1x^1 + c_2x^2 + ... + c_nx^n)$. While LFSR are

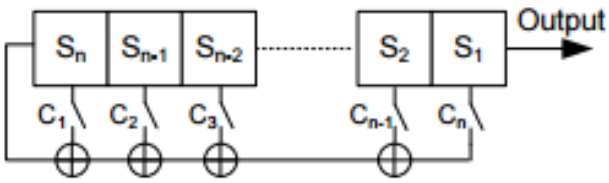

Figure 2.1: LFSR scheme

a very popular PRNG, the fact that they are linear presents a serious weakness[10].

- **Xorshift** is an specification of the LFSR that introduces the advantange that the needed state bits are a multiple of the length of the word. The number of state bits can be as large as 4096, this ensures that the memory cache of a computer can process the PRNG without slowing down while also generating a quite robust output [12].

Other algorithms commonly used as PRNGs include: Wichmann-Hill, Inversive congruential generator, ISAAC, Lagged Fibonacci generator, Maximal periodic reciprocals, Multiply-with-carry, RC4 PRGA, etc.

2.3 True Random Number Generators

Unlike PRNGs, True Random Number Generators (TRNGs), also known as Hardware Random Number Generators, provide randomness from a physical phenomenon rather than a pre-calculated list. There are many possible physical sources of randomness but all of them must allow the TRNG to generate unpredictable data, which in turn means that, as opposed to PRNGs, this kind of generators are nondeterministic because any sequence generated must not be possible to be reproduced at will and thus the random sequence must not be periodic. However, all of this also means that a TRNG will be more inefficient than a PRNG since it takes much more time to produce random numbers. [7]

TRNGs must be very reliable and robust as they are a very important element in cryptography since they are typically used in the generation of secure keys, nonces, padding or even shielding with random noise the data transmitted at a certain bandwidth in order to prevent modifications that can range from natural disruptive

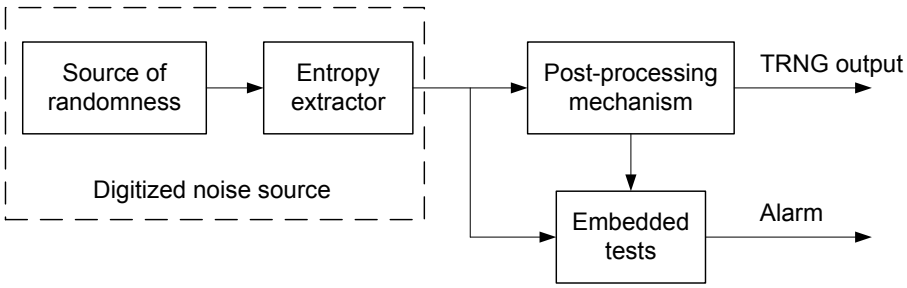

Figure 2.2: TRNG scheme

environmental changes to intended malicious attacks to the channel. Embedded systems also require the security a TRNG provides, however, these hardware devices tend to be designed to implement deterministic functions which may lead to costly undesired results and an overconsumption of the available resources. In the specific case of FPGAs, since the resources are limited to the logic blocks it contains the flexibility to implement a TRNG is reduced. [10]

2.3.1 Structure of a TRNG

The structure of any TRNG depends on the quality of the four basic components that can be seen in figure 2.2.

- **Source of Randomness:** also known as Entropy source, consists on a physical process that can be used to create unpredictable behaviour. These generally are: chaos, electronic noise, nuclear or quantum decay and free running oscillators [13]. However, most logic devices, such as FPGAs, do not contain analog blocks that can reproduce those physical phenomena. Instead, the phenomena used in those devices are:
  - **Metastability** refers to performance of logic gates between two different logic levels. This occurs when the hold time of flip-flops and the setup are violated. The reaction of the internal gates of the flip-flop is unpredictable oscillating at a voltage that is neither the expected logical low or high. After a while the oscillations recede and it outputs a logical low or high randomly [14].
  - **Jitter** is the short-term variation in signals between logic gates from their ideal position. This causes deviations and delays in the propagation of signals. The jitter is also seen as an instability of the clock period due to clock generators

that contain elements in a closed loop that produce a delay such as ring oscillators [15].

- **Thermal Noise** produced by the components inside the device, commonly resistors and capacitors. The noise is used as the frequency of a free running oscillator which to be extracted needs first to be converted to the time domain. However, FPGAs cannot implement resistors and capacitors therefore it is not possible to use this as a source of randomness in FPGAs [10].

- **Entropy Extractor** is a sampler that collects and digitalizes the maximum entropy of the source of randomness without tampering with the analog physical process. Moreover, a particular type of extractor will be used depending on the source employed. In the case of logic devices without analog blocks the extraction is done through synchronous or asynchronous flip-flops that sample the rising or falling edges of the desired signal. The synchronous case consists on taking samples at fixed time intervals of a random signal while asynchronous flip-flops take samples at random time intervals of a regular signal .

- **Post-processing Mechanism** is used to conceal the weaknesses or bias of the random signal that has been obtained. Imperfections in the signal can cause it to produce very weak random numbers that can even be considered non-random by the statistical tests. These weaknesses can be produced by a not high enough entropy source (metastability), a faulty extraction mechanism, environmental changes or tampering [15] [14]. Post-processing may not be applied if the TRNG does not suffer from this bias, otherwise it is required. By employing a post-processing mechanism that compresses the signal, entropy is increased and correlation and bias are reduced. The most common mechanisms are:

  - **XOR corrector:** applies a simple linear exclusive-OR operation on blocks of n consecutive bits to the signal outputting one bit. If the bits of the original signal were independent this compression of the data can reduce the bias the signal had very effectively. However, if they were correlated the bias won't be corrected. Not only this post-processing is very simple to implement in hardware but also keeps a constant throughput. [16]

  - **Von Neumann corrector:** probably the most well-known post-processing mechanism not only because it produces an unbiased output but also because it's a simple non-linear function. It processes pairs of successive bits: it outputs the first one if they are different and discards the pair if they are equal. As a result, about a 75% of the bits are discarded. In spite of this, its output will be unbiased even if the original was stationary and biased. [17]
  - **Hash functions:** an almost uniform and collision-resistant stream can be generated by a hash function it its fed a high entropy signal. However, these functions consume many resources so they are not easy to implement in hardware. [18]
  - **Linear Feedback Shift Registers:** not only are they aesy to implement in hardware but also they are able to output a sequence with a quite long period and very stastiscally robust. [19]
  - **Good Linear Codes:** also known as **Resilient functions** (a derivative of boolean functions [20]). If any attacker has discovered n bits of the random sequence they still cannot guess the full sequence adding robustness to the sequence. Moreover, these codes do not consume many resources and have a very good throughput. [21] [22] [23]
  - **Encryption of the digitalized noise signal:** based on the cryptographic diffusion and confusion properties to dynamically modify the output of the generator. However, this mechanism is not easy to implement and is quite expensive. [15]

- **Embedded tests** are required before the TRNG can be implemented in secure cryptographic applications. The tests examine the quality of the random signal generated by evaluating that it is uniformly distributed. Some of these tests are also used for PRNGs.

  - **DIEHARD:** one of the toughest test batteries, it has two versions: the original consisting of 15 tests [24] and a newer lightweight suite of 3 tests with updated versions of the three selected tests [25].
  - **ENT:** calculates five statistical meassures of the input stream: entropy, chi-square, arithmetic mean, Montecarlo value for Pi and the serial correlation coefficient. It is used

not just for testing the quality of RNGs but also for testing algorithms or applications that mask information density such as compression algorithms [26].

- **FIPS:** developed by the National Institute of Science and Technology (NIST), the Federal Information Processing Standard is used by the government of the United States. It specifies 4 levels of security for a cryptographic application designed to secure unclassified but sensitive data. [27]
- **AIS31:** exclusive to test TRNGs and developed by the Bundesamt für Sicherheit in der Informationstechnik (BSI) or the German Federal Office for Information Security. It consists of two levels of security: the first one, consisting of 6 tests (disjointness, autocorrelation and 4 FIPS tests), evaluates the final signal after post-processing while the second level that consists of three tests (distribution, comparative of multinomial distributions and entropy) evaluates the output signal of the source of randomness. The AIS31 also establishes that an alarm should be generated when the minimum entropy by bit is not achieved, this can be done through an on-line test. [28]
- **On-line Test:** According to [29] these tests should fulfill the next requirements:
  - It must detect almost immediately breakdowns of the source of randomness.
  - It must detect fatal statistical imperfections of the random sequence being tested.
  - If the sequence has a slight deviation from ideal randomness it must not raise an alarm.
  - It must barely consume any memory, run fast and be easy to implement.
- **NIST Statistical Test Suite for Random Number Generators:** a statistical package that includes 15 tests and developed by the National Institute of Science of Technology. It is generally used on PRNGs but is also suitable for TRNGs since it tests the quality of the distribution of large binary sequences as well as their unpredictability [3]. This is the battery of tests that will be used to test our TRNG. Take into account that out of the 15 tests only 13 will be required since the Random Excursions and the Random Excursions Variant tests are only suitable to test

PRNGs [30]. It should be noted that for all of these tests, only if the P-value obtained is > 0.01 the sequence tested is considered random. The P-value of each test is calculated from a test statistic individual for each test. For more information see the documentation provided by the NIST. [31]

* **Frequency Test** This test focuses on the frequency in which zeroes and ones appear in the sequence being tested. In order to pass the test, the proportion of zeros and ones must be approximately equal as is expected from a sequence that is truly random. Passing this test is crucial as all the others depend on this one. To compute the test statistic it is first required that the binary numbers are converted to ± 1. The zero values of the input sequence are converted to -1 while the ones are considered as $+1$. Then, they are added to produce the function: $S_n = X_1 + X_2 + ... + X_n$ in which $X_i = 2\epsilon_i - 1$, where ϵ_i is either a -1 or a $+1$. If S_n results in a large positive value it means that in the sequence there were far more ones than zeros and viceversa if S_n results in a large negative number.
* **Frequency Test within a Block** In this test, the frequency of ones is calculated by the proportion in which they appear in non-overlapping blocks of M-bits. Any bits that cannot fill a full block are discarded. Since the previous test stablished that the number of zeros and ones in the sequence should be approximately equal, for the sequence to pass this test, it is expected that the number of ones in the block should approximately be $\frac{M}{2}$, otherwise the sequence is considered non-random. In case that $M = 1$ the test works just like the previous monobit test. It is recommended that the sequence to be tested should at least be of 100 bits.
* **Cumulative Sums Test** This test first computes the cumulative sum, also known as a random walk, of partial sequences of the input sequence (ϵ). To do so, it is necessary to form a normalized sequence by turning the bits of ϵ into ± 1 according to $X_i = 2\epsilon_i - 1$. Afterwards, the test determines how large or small the random walk is compared to the one expected from a random sequence. If the statistic results in a small value it means that there is a similar amount of ones

and zeros and is therefore random. Otherwise, a large value indicates that there are either too many ones or zeroes at the beginning of the sequence or at the latter stages.

* **Runs Test** A run is the name given to an uninterrupted sequence of the same bits, therefore, a run of k bits also has length k and is preceded and followed by the opposite bit value. The goal of this test is to check how many runs are there in the input sequence whether their lengths are suitable for a random sequence as well as how fast the oscillation between bits takes place. This test requires a previous calculation of the Frequency test to work, if it failed this one is not performed and its P-value is set to 0. In order to be considered random there must be a lot of changes in the sequence.
* **Tests for the Longest-Run-of-Ones in a Block** Expanding upon the previous test, the sequence is divided in blocks of M bits from which the longest run of ones will be the focus. In order to be considered a random sequence, the length of the longest run of ones should be consistent with that of a random sequence. The same could be tested taking the runs of zeroes but it is not required as if the length of the longest run of ones does not meet the criteria it implies that the same is most likely to happen with the longest run of zeroes. The test has been prepared to work at 3 different sizes of M: 8, 128 and 10^4 for which the minimum length of the sequence is 128, 6272 and 750000 bits respectively.
* **Binary Matrix Rank Test**
The goal of this test is to determine whether or not there exists a linear dependence among substrings of the input sequence. These substrings are organized as sub-matrices of MxQ bits and from them the rank of disjoint is calculated. The test has been configured to set $M = Q = 32$. The bits that cannot fill an entire matrix are discarded.
* **Discrete Fourier Transform Test**
This test detects periodicity in the form of repetitive patterns in the input sequence. This is done by calculating the Discrete Fast Fourier Transform of the sequence and checking its peak heights. The number of

peaks that exceed the threshold by 95% should considerably differ from 5%.

* **Overlapping Template Matching Test**
  In this test, the focus is the number of times target strings occur. It will consider the sequence as random if the number of ones of a certain length does not deviate from the number it expects from a random sequence. As in the Longest-Run test, it is not necessary to test the number of zeros since an anomaly in the number of ones also means an anomaly in the zeros. This process is done through a window of m bits that used to search an m-bit pattern throughout the whole sequence to check how often that pattern appears. The window slides by one bit whether the pattern is found or not.

* **Non-overlapping Template Matching Test**
  As in the previous test, this test focuses on the number of times target strings occur. However, unlike the Overlapping Template Test, this tes only considers random those sequences that do not contain many occurrences of the pre-defined aperiodic strings. As such something very important to mention is that in this one many P-values are computed because it depends on several variables and not just the test statistic. This test works similarly to the previous one with the exception that the window only slides one bit when the pattern has not been found, otherwise, the window slides m bits or, in other words, to the next position to where the pattern was found.

* **Maurer's "Universal Statistical" Test**
  In this test randomness is related to the compression of the sequence. A sequence can be compressed as long as no information is lost. If a sequence can be compressed greatly it is not random because there were many matching patterns and thus they did not add any new information. To determine this, the input sequence has to be partitioned as seen in figure 2.3 so that $K = \frac{n}{L} - Q$ If there are any remaining bits that cannot form a L-bit block they are discarded. This test is also based in a series of pre-computed values, it should be noted that if the test statistic differed considerably from these pre-computed expected values, the P-value will not fall in the desired range. In that case,

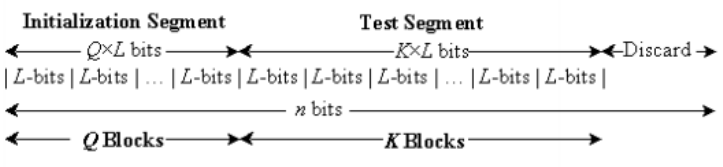

Figure 2.3: Universal test sequence partitioning

the sequence can be compressed and thus not random.

* **Approximate Entropy Test**
  The goal of this test is to check the frequency of every overlapping blocks of similar lengths in the sequence, one of m bits and the other of m+1 bits.
* **Serial Test**
  Since a random sequence should be uniform, any of the 2^m m-bit overlapping string or pattern must appear the same number of times as any other m-bit pattern. As such, this test checks if the frequency of appearance of any possible overlapping pattern of m bits throughout the entire sequence is approximately the same. If m = 1 this test operates just like the Frequency test.
* **Linear Complexity Test**
  This test establishes the complexity of the sequence comparing it to a LFSR. The longest the LFSR is the more complex the sequence and the better the randomness. If the length is too short the test fails. The shortest possible LFSR is computed through the Berklekamp-Massey algorithm [32].

2.4 FPGA Technology

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that contain generic logic cells in a two dimensional array along with programmable switches forming a matrix of Configurable Logic Blocks (CLBs) whose interconnections are also programmable (see section 2.4.1 for more details). Unlike Application Specific Integrated Circuits (ASICs) devices that are designed for a specific design, the programmable capacity of FPGAs means they can be reprogrammed to run several applications with very different circuit layouts without having to re-design the application nor changing the hardware [33].

The main manufacturers of this hardware are Xilinx and Altera which have controlled the market for years. While Altera was the first of the two in producing a truly reprogramable industrial logic device in 1984, it was Xilinx who a year later released a commercial FPGA known as the XC2064 (whose name refers to its 64 CLBs). These two companies were unchallenged in the market during a decade until the mid-1990s when other companies started to become serious competitors, among these are: Microsemi, SilliconBlue Technologies, Achronix, QuickLogic, Lattice Semiconductor, Tabula, etc

Nowadays, some areas in which FPGAs are commonly used in [34] are:

- **Aerospace and Defence:** avionics, missiles and munition, security, space, etc.
- **ASIC Prototyping:** it is possible to verify an embedded system.
- **Audio:** with Digital Signal Processing FPGAs provide a wide range of applications in this field.
- **Automotive:** driver assistance systems, vehicle networking and connectivity and in-vehicle infotainment.
- **Broadcast:** Encoders, decoders, displays, switches and routers, Real-Time Video Engines, etc.
- **Consumer Electronics:** portable electronics, digital cameras, printers, home networking, etc.
- **Data Center:** storage in Servers for Cloud Computing.
- **Industrial:** imaging, networking, motor control, etc.
- **Medical:** Ultrasound, CT scanning, X-ray, etc
- **Video and Image Processing:** video over IP, high resolution video, digital displays, etc.
- **Wired and Wireless Communications:** optical networks, network processing, radio, connectivity interfaces,compatible with standards such as HSDPA and WiMax

The FPGA behaviour is programmed or described through a design in a schematic or it is written in a hardware description language (HDL), the most well know among these languages are VHDL (VHSIC hardware description language) and Verilog although there are

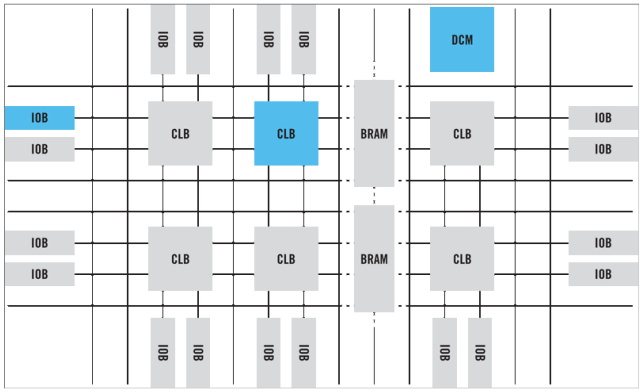

Figure 2.4: FPGA Block Structure

newer graphical programming languages like National Intrument's LabView. For this project, as we have chosen a Xilinx Spartan-3E Board their official design suite has been used and therefore, our design is written in VHDL as it is the recommended option for this application and our target device.

Generally, before synthesizing and running an application on the FPGA hardware, circuit designs are first tested by simulating them with software programs such as Modelsim however, this is not possible for testing True Random Number Generators because it is impossible for the software to emulate the source of entropy that the TRNG requires.

2.4.1 FPGA Block Structure

Every model has different components that make it unique however, as seen in figure 2.4, there are some that can always be found in the block structure of a FPGA board:

- **Logic Blocks.** Formed by a switching matrix, flip-flops and selection circuits like multiplexers or demultiplexers. This Configurable Logic Block can have a synchronous or asynchronous output thanks to these components and moreover, it is always accompanied by several I/O pads as well as flexible interconnection routes between them and other blocks of the board. To reduce design complexity the routing is an automatic task generally hidden from the designer of FPGA applications however this may cause a wrong use of the FPGA resources and take more routing tracks than the design might actually need.

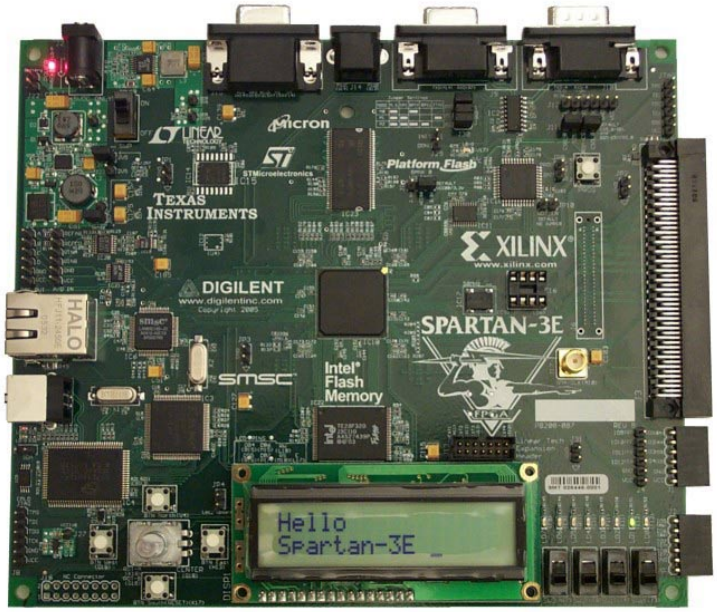

Figure 2.5: Spartan-3E

- **Memories.** FPGA designs can have on-chip memory as most boards contain some sort of Embedded Block RAM (BRAM) memory that can also work in a dual-port operation with diferent clocks.
- **Clocking.** Many components will require a synchronous clock signal for them to work therefore a Digital Clock Manager (DCM) will always be present alongside with routing dedicated exclusively to clock and reset signals all throughout the board as well as clock frequency synthetization through analog PLLs or DLLs and sometimes even a Digital Signal Processor (DSP). A DCM can manipulate clock signals by eliminating their skew, adding a different phase shift, multiplying or diving the signal, etc.

2.4.2 Xilinx Spartan-3E

This project uses a board from the Xilinx Spartan-3E family devices which is an improvement over the previous successful Spartan-3 family. This is done by improving the configuration costs and system performance since the amount of I/O pads per logic cell is higher. Moreover, it comes with newer, cheaper and more advanced 90-nanometer technologies which not only allow for more bandwidth

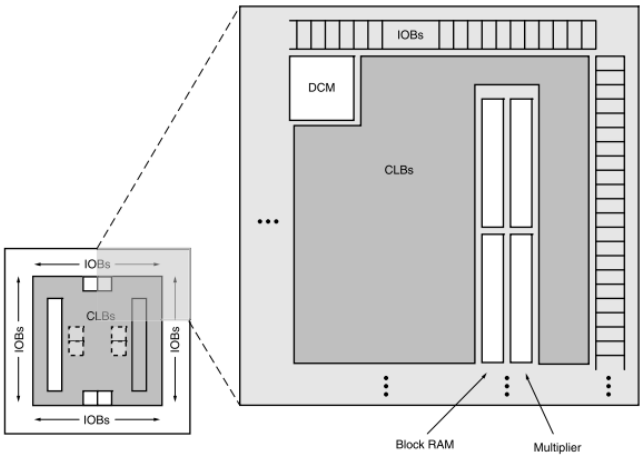

Figure 2.6: Spartan-3E Family Architecture

but also many functionalities that make them ideal for their use in consumer electronics such as home networking with broadband access and digital displays for television. [35]

There are five main programmable types of elements in this family of FPGAs, their distribution in a device are as shown in figure 2.6:

- **CLBs:** store data in latches and flip-flops as well as implement the logic of the circuit.
- **I/O Blocks:** manage the data transmitted and received bidirectionally in between I/O pins and the rest of the device.
- **Block RAM:** stores data in dual-port blocks of 18Kb. Each BRAM has their dedicated multiplier.
- **Multiplier Blocks:** can calculate the product of two binary numbers of 18 bits.
- **DCM Blocks:** to digitally self-calibrate clock signals. It is possible to multiply, divide, delay, phase-shift and distributing the signal.

The board that has been used in this project, the Xilinx Spartan-3E Starter Board (see figure 2.5), also has as some its main features [36]:

- 232 I/O pins

- A Xilinx 4Mbit PROM
- Parallel 16MB NOR Flash configuration
- 16Mbit SPI serial Flash configuration
- MicroBlaze^{TM}32-bit embedded RISC processor
- PicoBlaze^{TM}8-bit embedded controller
- 64MB DDR SRAM interfaces.

2.5 Implementing TRNGs on FPGAs

As mentioned in section 2.4, when designing a solution for integrated circuits it is important to take into account that the resources they provide are limited. In spite of this, there are several ways to implement TRNG on FPGA technology. These can be presented according to the method employed to extract randomness or their source of entropy.

1. **Metastability.** As seen in section 2.3.1 , it refers to the ability of a circuit to endure an unstable state during a prolonged time. FPGAs are designed to avoid situations that can produce metastability which makes it difficult to use this mechanism. However, a couple of solutions employing metastability have been implemented on FPGAs, the most well-known is the one designed by Vasyltsov et al. [37] As presented in figure 2.7, it is a ring oscillator (RO) formed by inverters whose output is connected to their input by a multiplexer that acts as a switch. The state of those switches determines if the circuit works in entropy accumulation mode (metastability) or in oscillating mode. When the inverters are disconnected from each other metastability occurs and the output voltage fluctuates. Therefore, when they are connected again the state of the ring oscillator depends on the entropy created in each inverter.

2. **Open Delay Chain.** Through a chain of delay elements an n-bit signal is obtained by using D-latches in the LUTs. Afterwards, it is XOR-ed to produce a random bit. There are two advantages to this method: first, it is possible to produce high rates of data at the maximum clock frequency and second, it is a simple structure without the need of PLLs or ROs.

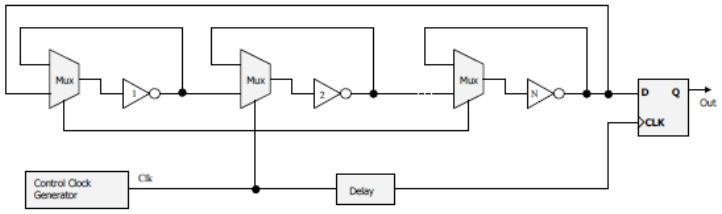

Figure 2.7: Vasyltsov et al. TRNG

3. **Chaos.** Obtaining randomness from a chaotic event is only possible through an analog device. Most FPGAs lack such technology but certain models have several analog blocks or peripherals. However, these cannot produce the chaotic behavior required for a TRNG. Although it is possible to produce this behavior through an Analog to Digital converter which is only a matter of time until FPGAs are manufactured with one.

4. **Use of SRAM Memories.** There are two methods to make use of the SRAM in an FPGA to generate random numbers: first, through the start-up state of the SRAM that generates non-deterministic noise [38] and second, by the *write collisions* method which consist of trying to write opposite values at the same time which results in a process similar to metastability [39].

5. **Jittery Clock Sampling.** The deviation from ideal behavior of a signal or jitter has been proven to be the best method for FPGAs since the possible implementations are technology independent and require components that can be found in any type of FPGA. The jittery signals are composed of thermal noise and flicker noise. The former results in the non-deterministic behavior while the latter adds a deterministic behavior dependent of the technology on which the TRNG is implemented. The two techniques of using jitter are:

    - **Coherent Sampling.** This method employs two or more clocks with output frequencies or phases that are related. One of the clock signal samples the others in the edges but since none of them are ideal the result will be a random sequence. There are two well-known implementations of this principle: Fischer and Drutarovsky's [40] and Kohlbrenner and Gaj's [41] . In the former, PLLs are used to generate the jittery clock signals, no post-processing is required

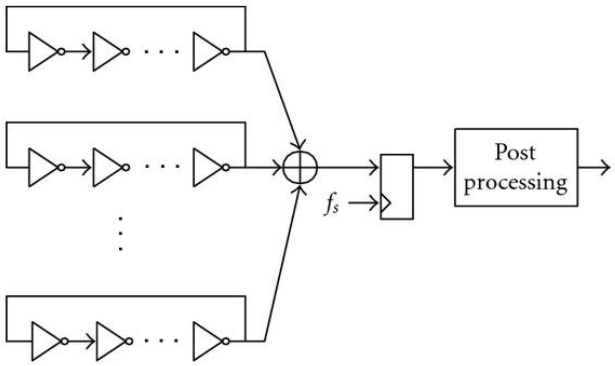

Figure 2.8: Sunar et al. TRNG

and the output bit rate, while considerably low, is constant. However, many FPGA families do not contain PLLs and therefore this TRNG cannot be implemented in them. Kohlbrenner and Gaj solve this problem by replacing the PLLs by ROs that can be implemented in all FPGAs. The frequencies of the oscillators must be very similar but not the same in order to obtain the desired random sequence which will have a very long period.

- **Jittery Clock Signals.** This method uses a synchronous or asynchronous D flip-flop (DFF) to sample the high frequency signals generated by oscillators. A reference clock (generally obtained from a quartz oscillator) is used to obtain the sampling frequency. This technique was originally coined by Sunar et al. (see figure 2.8) [42] who samples several ROs at the same time. Their outputs are XOR-ed to obtain a high-frequency random signal which is sampled by the DFF. Afterwards a resilient function is used as post-processing that makes the output bit-rate constant. This implementation, however, has two disadvantages: not only the power consumed by this TRNG is substantial since many ROs are required to oscillate simultaneously and continuously but also the XOR and the DFF are not capable of handling all the transitions. A solution to this problem is proposed in [43] and can be seen in figure 2.9. It consists of putting a DFF after each RO. Moreover, it claims that by doing this the post-processing would become unnecessary and the number of ROs can be reduced. It has been argued however that the behavior it produces is not true randomness but pseudo-randomness. Another

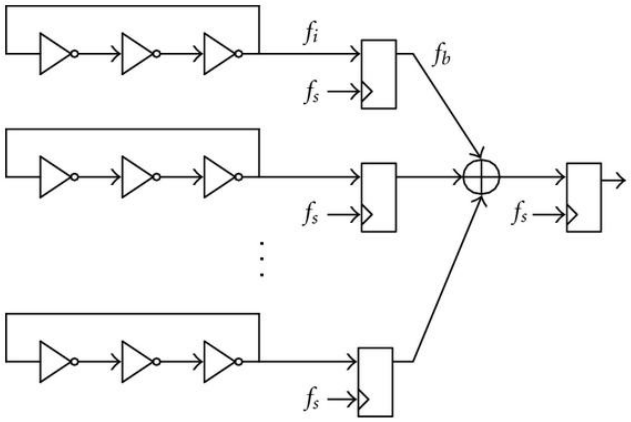

Figure 2.9: Wold et al. TRNG

improvement to this was proposed in [44] where instead of ROs, Self-Timed Rings are used.

In this project, a jittery clock has been used as entropy source. It has just been stated above that this can be implemented by using ROs or STRs, therefore an explanation of both is in order.

- **Ring Oscillators:** consist of a chain of an odd number of elements connected in a feedback loop to form a ring. These elements are either inverters as seen in figure 2.10 or an initial inverter and delay elements. They are widely used in ASICs and FPGAs in jittery clock sampling due to their many useful features [45]:
  - They are easy to design in integrated circuit technology.
  - High frequency oscillations with dissipating low power are achieved at a low voltage.
  - Their frequency of oscillation can be tuned with a wide range of possibilities.
  - It is possible to obtain a multiphase output.

  The frequency of oscillation depends on the number of delay elements and the propagation delay they produce. A self-sustained oscillator must have a 2π phase shift. For a RO of n elements to achieve this, if each element adds a $\frac{\pi}{n}$ phase shift and a propagation delay of τ_d (its actual value depends on the parameters of the circuit), the RO completes a first revolution in a time of $n\tau_d$ with a phase shift of π. Another revolution

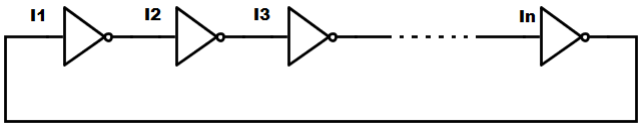

Figure 2.10: Ring Oscillator basic stucture

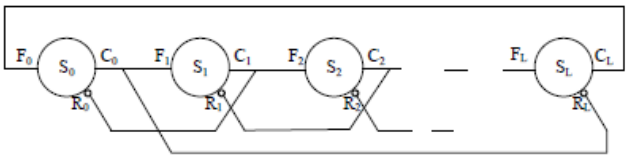

Figure 2.11: Self-Timed Ring basic stucture

is required to reach the 2π phase and thus the time period of the whole process becomes $2n\tau_d$. Therefore, the frequency of oscillation on an RO is $f_o = \frac{1}{2n\tau_d}$.

- **Self-Timed Rings:** are a handshake protocol that controls a sequence of events by distributing them through different adjacent stages (each consisting of a Muller C-element and an inverter) forming a micropipeline FIFO as can be seen in figure 2.11. The clocks of synchronous designs are replaced by these handshakes and therefore the oscillator is highly configurable allowing to choose phase and frequency between signals. The jitter does not depend on the length of the ring but on the one that each stage generates [46].

As was stated in section 1.2, the nature of this project is to expand on Böhl's RO. However, it must be taken into account that, despite all of their advantages, ROs are very sensitive towards voltage and process variability. In [46] and [47] comparisons between ROs and STRs are presented. They conclude that, while in ROs the jitter has a greater magnitude and propagates throughout the whole ring, the jitter generated by STRs has a better quality as its deterministic noise component is much lower than the one a RO generates.

Chapter 3

Characterization of the TRNG

In this chapter, the implementation and design of the TRNG is presented along with an analysis of its results.

First, the detailed design of the TRNG is shown, followed by its implementation on the Xilinx Spartan-3E as well as how the random numbers it generates are captured and handled prior to being tested. Afterwards, the different experiments performed on the hardware are explained. First, testing for which frequency and post-processing the best results are obtained, second, on which area of the chosen FPGA the TRNG is the most robust, third, a comparison between different FPGAs on the optimal position and, last but not least, a forced reset of the generator to prove whether or not it is still reliable.

3.1 TRNG Under Study

This project studies the TRNG proposed in [1]. Back in section 2.5, it was stated that common designs for TRNGs on FPGAs consist of ring oscillators. ROs are a feedback loop of an odd number of inverting elements. Most RO-based TRNGs employ several ROs, however [1] and [2] state that a robust TRNG can be implemented with just one RO. This is achieved with multiple taps or sample points within it. Typical approximations use instead of a single sample point after the last inverting element before the feedback takes place. This was previously shown in figures 2.8 and 2.9 from [42] and [43] respectively. In any case, it is required that these taps

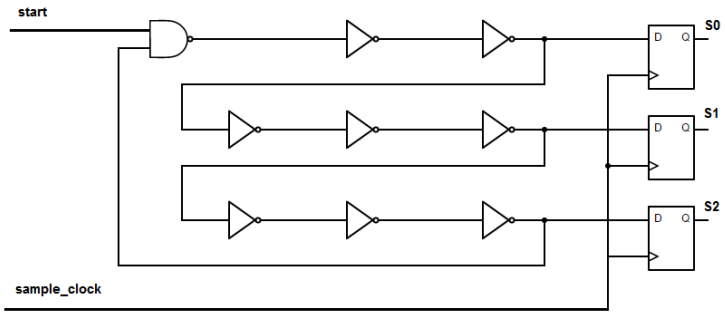

Figure 3.1: TRNG design

are arranged after an odd number of elements for this approach to work.

Therefore, the TRNG design is as shown in figure 3.1: a RO of 9 inverting elements and 3 sample points. The initial inverter is replaced by a NAND gate to control the start of the oscillations as well as stop them. The sampling is done through D flip-flops placed after the third, sixth and ninth inverters respectively. The DFFs store the retrieved bits.

This design extracts entropy from the jitter or deviation from the ideal behavior of the signal travelling through the RO. This occurs because it is being delayed by the inverting elements that add to it flicker and thermal noise. The noise is the non-deterministic component that produces randomness. Therefore the bits stored in the DFFs can be XORed or simply processed independently because they are already random.

On the one hand, [1] simply states that a sampling frequency of 1MHz is enough to accumulate sufficient jitter. Our first goal will be to prove the veracity of this statement by controlling the *sample_clock* signal. On the other hand, [2] recommends to use a 3-bit XOR post-processing, however, the results of the statistical tests are not provided. Thus, the actual quality of the output of the RO is unknown. As such, it was decided to first test the quality of the TRNG (see section 3.3) as is and afterwards, if necessary, design a post-processing unit.

3.2 Experimental Framework

Now that the theoretical design of the TRNG has been introduced, this section deals with its actual physical design as well as its im-

plementation on the chosen FPGA. Once this is done, it focuses on the procedures taken to capture the output and handling it before and after the statistical tests were applied.

3.2.1 FPGA Implementation

A design is implemented in the Xilinx Spartan-3E family by describing the required hardware elements in a VHDL application which is loaded to the FPGA. For a design to work the target FPGA must have the elements described in it. This would not be a constraint for the TRNG since its elements are just logic gates and DFFs which are elements featured in every FPGA technology.

First, the three basic elements of the design (inverter, NAND gate and DFF) were described in three separate VHDL files. Secondly, the actual RO is implemented by adding these three as its components. Lastly, since we want to control and change the sample frequency a last file is required in which the RO will be a component. The clock of the Spartan-3E runs at 50MHz so in order to make the RO work in a different oscillation frequency a counter is created. This counter will create the sample_clock signal that will be fed to the DFFs. The desired frequencies are obtained according to the length of the counter: the largest the counter is the smaller the oscillating frequency. Nevertheless, there is a third element in the final application: the RS232 protocol connection. This connection is an important part of the design since it is through it that the generated random bits of the TRNG are transmitted to be captured.

In the previous section, 3.1, it was stated that this design offered two different possibilities to produce a random sequence:

1. XOR the 3-bits stored in the DFFs to easily produce a single signal
2. Process the 3 bits independently, each forming a bit of the random sequence.

It was decided to make use of this second possibility so a longer sequence will be created more quickly. The bits created in these stages can be considered random because the phase shift produced by the accumulated jitter has been through an odd number of elements after each sample is taken, a requirement that was specified in section 2.5. The 3-bits are then stored in a 8-bit register before they are overwritten in the next oscillation. The contents of this

register are shifted and sent through the RS232 transmission to be captured.

All the VHDL code used in the design is available in appendix A.

There is however a crucial intermediate step in the implementation that has not yet been mentioned. In section 2.4.1 it was declared that the difficulty of designing applications for FPGAs is greatly reduced because the routing of the circuit is done automatically which is quite convenient for most designs. It would also be convenient for this TRNG if it was not for the nature of the experiments to be performed.

Since we want to place the TRNG in several positions all throughout the FPGA it is most likely that the routing that will be assigned in each case will also be different. Theoretically, this should not affect the process, however it cannot be assured. Therefore, we have to ensure that the routing of the RO is always the same. This is done by turning the RO into a Hard Macro before adding it to the final design. A Hard Macro is a custom analog block surrounded by digital logic. This means that for an application the distribution of blocks of the FPGA with the required components as well as the routes used to connect them are fixed. Now, wherever the oscillator needs to be placed its routing will remain constant. The designer simply needs to be careful and place the reference component of the RO in an area of the FPGA that allows that distribution. An in-depth explanation of how the Hard Macro of the TRNG is created can be found in appendix B.

Once the design has been implemented in VHDL, the target device, the Xilinx Spartan 3-E, has to be configured to run the application. This has been done in ISE Project Navigator an official application distributed by Xilinx. Notice that, as was already said in section 2.4, before programming an FPGA with a program, the application should be tested in simulating software but since a TRNG uses physical processes to generate entropy it cannot be simulated.

3.2.2 Treatment of the Data

While the TRNG is running on the FPGA, the random sequence has to be captured before its quality can be tested. Any application that captures data from the prolific serial port of a computer would suffice.

The statistical tests demand that a sufficiently large sequence

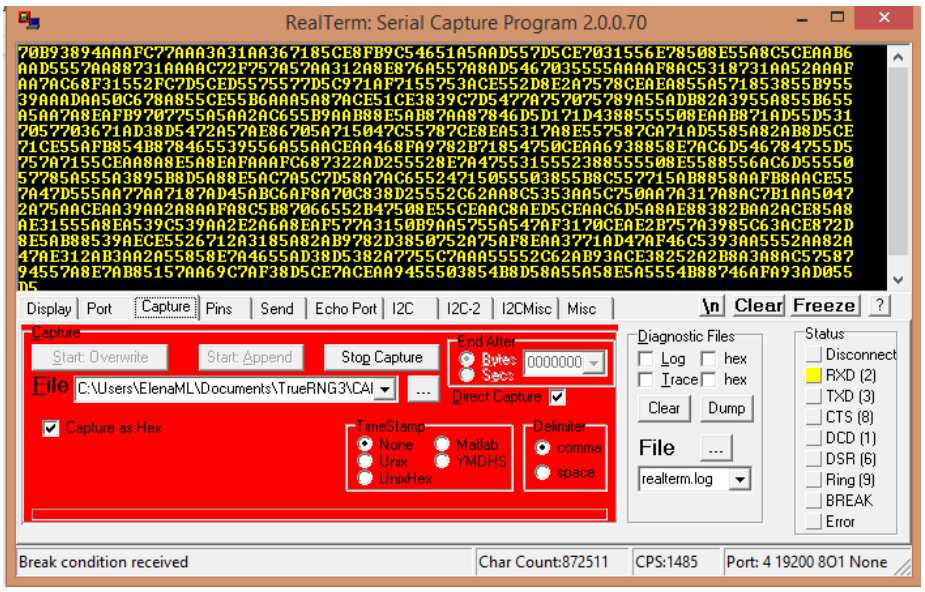

Figure 3.2: Capturing software

must be evaluated, the larger the sequence the more reliable the results of the tests. It was considered that capturing around 10MBs of data would be more than enough. Moreover, the chosen software does not capture the data received from the RS232 in binary, as is necessary for the battery of tests, but in hexadecimal. Therefore, the 10MB hexadecimal file after conversion conveniently becomes a binary file of approximately 40MBs. The hexadecimal to binary converter is a Python script developed specifically for this project whose code can be found in appendix C, it only requires to specify the input hexadecimal file and the name or location of the new generated binary file.

The NIST allows a free download of its Statistical Test Suite as a zip which we have run on a virtual linux machine to save us time as it requires to run a *makefile* to install it. To run the tests the first step is to assess the number of traces and the input file with the binary sequence. The application then prompts for the tests to be passed, even though the results of the Random Excursions and the Random Excursions Variant are specific to PRNGs and would not be useful in this context it was chosen to run all the tests as it is easier to simply ignore the results of these two later on. Back in section 2.3.1 several parameters required in the tests were introduced, if we wished to modify them it would be possible at this point. Nevertheless, it was decided to use the default values:

1. Block Frequency Test - block length(M): 128
2. NonOverlapping Template Test - block length(m): 9
3. Overlapping Template Test - block length(m): 9
4. Approximate Entropy Test - block length(m): 10
5. Serial Test - block length(m): 16
6. Linear Complexity Test - block length(M): 500

The results of the tests are stored in an ASCII file named *finalAnalysisReport.txt*. This file nicely lays out the numerical statistical data and marks which tests have failed but it does so in a way that it is not easy to work with it. Another Python script has been created to remove the unnecessary info of the results file and instead turn it in a comma-separated CSV file. The script is presented and detailed in appendix C. From the CSV file the p-values can be easily extracted afterwards in Matlab to compare graphically each measurement.

After taking the measurements for the first experiment it was found that the TRNG was not generating reliable random numbers because it did not pass the tests. A post-processing mechanism is required.

3.2.2.1 Post Processing

In Böhl's original paper [1], no specific post-processing mechanism was specified. In [2], however, a simple serial XOR corrector is used as post-processing. As it was already mentioned in section 2.3.1, this operation compresses the original sequence which results in an effective reduction of bias or deviation from the expected $\frac{1}{2}$ number of ones and zeroes.

The basic bitwise XOR operation is shown in table 3.1 and can be represented as $A \otimes B$. It will return 1 if A or B are 1 and it will return 0 if they are equal. This also means that the operation is commutative: $A \otimes B = B \otimes A$. An XOR can operate more than two bits as can be seen in tables 3.2 and 3.3. These tables also help to realize why this operation is also known as a *parity check*: a sequence of n bits has even parity ($XOR = 0$) if the number of 1s in it is even, otherwise, if the number of ones is odd the sequence also has odd parity ($XOR = 1$). From this it can also be deduced

Table 3.1: XOR operation of 2 bits

A	**B**	**XOR**
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.2: XOR operation of 3 bits

A	**B**	**C**	**XOR**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

that a XOR operation is also associative, for example, for 3 bits: $A \otimes (B \otimes C) = (A \otimes B) \otimes C$ [16].

An XOR compresses a binary sequence according to the number of bits the operation takes, for example: an XOR operation of 2 bits is mapping 2 bits into 1 and thus reduces the length of the sequence in half. A 3-bit operation has a compression rate of 3:1 bits, for 5-bit operation it is 5:1 bits and so on. Moreover, in a n-bit XOR all possible combinations of those bits are 2^n, which always consist of an equal amount ones and zeroes: $\frac{2^n}{2}$. For example: in table 3.2, the 3-bit XOR yields $2^3 = 8$ possible results, out of these, one half, $\frac{2^3}{2} = 4$, are ones and the other half are zeros. Meanwhile, in table 3.3, the 5-bit XOR returns $2^5 = 32$ possible results, 16 1s and 0s.

To sum up, the greater the number of bits that the XOR takes, the greater the compression rate as well as the more difficult for an attacker to guess which was the original sequence of bits, therefore not only reducing the bias of the sequence but also making it more resistant to attacks. While the reference work used simply and XOR of 3-bits, for this project it has been decided to also evaluate the performance of other three post-processing XOR correctors that operate with a higher number of bits.

In spite of how advantageous the post-processing is, one problem has been encountered: the time constraint. Capturing the data of a single measure takes up to approximately an hour and a half, if instead of measuring each experiment once for each frequency and position it were performed 4 times, the time required to finish

Table 3.3: XOR operation of 5 bits

A	**B**	**C**	**D**	**E**	**XOR**
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	0
1	1	1	0	0	1
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	1

the project would have increased greatly leaving no margin for errors and possible re-measurements. Since at this point the physical process required for the TRNG has already been handled, it was decided to implement the post-processing mechanism as a Python script outside the hardware to optimize the running time. This optimization allows us to compare the exact same measurement with different post-processing. The code of this script has been included in appendix C.

Through this optimization, the XOR correctors used in the project result in:

1. 3-bit XOR:
   - Compression rate: 3:1, reduces 40MB file to approximately 13MBs
   - Possible combinations of bits: $2^3 = 8$, number of ones and zeros: $\frac{2^3}{2} = 4$.
2. 5-bit XOR:
   - Compression rate: 5:1, reduces 40MB file to 8MBs
   - Possible combinations of bits: $2^5 = 32$, number of ones and zeros: $\frac{2^5}{2} = 16$.
3. 7-bit XOR:
   - Compression rate: 7:1, reduces 40MB file to approximately 5.7MBs
   - Possible combinations of bits: $2^7 = 128$, number of ones and zeros: $\frac{2^7}{2} = 64$.
4. 9-bit XOR:
   - Compression rate: 9:1, reduces 40MB file to approximately 4.45MBs
   - Possible combinations of bits: $2^9 = 512$, number of ones and zeros: $\frac{2^9}{2} = 256$.

3.3 Experiments: Results and Analysis

In [1] it is stated that at 1MHz sufficient jitter accumulates to generate a robust random sequence. Therefore the first experiment performed consisted on proving whether this is truly the optimal oscillation frequency or not. Once that experiment concluded it was also desirable to evaluate if the quality of the TRNG was dependent on the position on which it was placed in the hardware. To find this, nine representative positions of the FPGA layout were chosen.

Originally, those were the only experiments planned because it was considered that they would take more time than they actually did. This was partly due to the new computer on which the capturing software was run but mostly thanks to using python for the XOR corrector. With this extra time two more experiments were executed: a test of the optimal position in several FPGAs and a restart of the circuit while the TRNG is running to make sure it is still working.

The analysis of all four tests is described in the subsequent sections. Notice that since the generator did not pass the tests without post-processing these sections only provide the results of the 4 XOR correctors.

3.3.1 Frequencies

The first experiment focuses on proving the claim from [1] that the optimal frequency of oscillation for the design is 1MHz. To do so, a range of frequencies including 1MHz has been chosen to evaluate the behavior of the TRNG. This range goes from 50KHz to the oscillation frequency of the clock of the FPGA, 50MHz. Inside this range, six representative frequencies were tested: 50KHz, 100KHz, 500KHz, 1MHz, 25MHz and 50MHz.

Moreover, for this experiment the placement of the TRNG within the Spartan-3E board has been fixed to a central position. Due to the Hard Macro it is assured that the routing and components used in each measurement are exactly the same so it is possible to conclude that the results obtained are completely dependent on the frequency being tested.

Tables 3.4 to 3.9 show the p-values obtained from the statistical test suite while in figure 3.3 a graphic view of these is presented.

For 50KHz (table 3.4):

- **3-bit post-processing:** only passes 4 tests out of 15
- **5-bit post-processing:** only passes 6 tests out of 15
- **7-bit post-processing:** passes 9 tests out of 15
- **9-bit post-processing:** passes 14 tests out of 15

For 100KHz (table 3.5):

- **3-bit post-processing:** only passes 4 tests out of 15
- **5-bit post-processing:** passes 5 tests out of 15
- **7-bit post-processing:** passes 10 tests out of 15
- **9-bit post-processing:** passes 13 tests out of 15

For 500KHz (table 3.6):

- **3-bit post-processing:** only passes 4 tests out of 15

Table 3.4: Central Position 50KHz

Test	PP3	PP5	PP7	PP9
Frequency	0.000000	0.000000	0.236810	0.699313
BlockFrequency	0.000000	0.000076	0.000954	0.366918
CumulativeSums	0.000000	0.000000	0.071177	0.897763
CumulativeSums	0.000000	0.000005	0.437274	0.759756
Runs	0.000000	0.000000	0.000000	0.867692
LongestRun	0.000000	0.000001	0.002559	0.739918
Rank	0.616305	0.289667	0.090936	0.090936
FFT	0.000000	0.798139	0.275709	0.042808
OverlappingTemplate	0.000000	0.003447	0.005762	0.883171
NonOverlappingTemplate	0.029305	0.45492	0.396955	0.473554
Universal	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.000000	0.001296	0.006661	0.162606
Serial	0.000000	0.514124	0.719747	0.012650
Serial	0.401199	0.003996	0.867692	0.779188
LinearComplexity	0.102526	0.085587	0.798139	0.334538

- **5-bit post-processing:** passes 9 tests out of 15
- **7-bit post-processing:** passes 12 tests out of 15
- **9-bit post-processing:** passes 13 tests out of 15

For 1MHz (table 3.7):

- **3-bit post-processing:** only passes 3 tests out of 15
- **5-bit post-processing:** passes 9 tests out of 15
- **7-bit post-processing:** passes 10 tests out of 15
- **9-bit post-processing:** passes 13 tests out of 15

For 25MHz (table 3.8):

- **3-bit post-processing:** only passes 6 tests out of 15
- **5-bit post-processing:** passes 13 tests out of 15
- **7-bit post-processing:** passes 14 tests out of 15
- **9-bit post-processing:** passes 14 tests out of 15

For 50MHz (table 3.9):

- **3-bit post-processing:** only passes 2 tests out of 15

Table 3.5: Central Position 100KHz

Test	PP3	PP5	PP7	PP9
Frequency	0.000000	0.000000	0.759756	0.657933
BlockFrequency	0.000000	0.000000	0.759756	0.213309
CumulativeSums	0.000000	0.000000	0.911413	0.935716
CumulativeSums	0.000000	0.000000	0.006196	0.494392
Runs	0.000000	0.000000	0.000000	0.005358
LongestRun	0.000000	0.000000	0.004301	0.964295
Rank	0.924076	0.249284	0.304126	0.037566
FFT	0.000000	0.009535	0.419021	0.058984
OverlappingTemplate	0.000000	0.000000	0.115387	0.108791
NonOverlappingTemplate	0.021242	0.3089	0.480633	0.501248
Universal	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.000000	0.000000	0.000097	0.455937
Serial	0.000000	0.028817	0.289667	0.779188
Serial	0.851383	0.019188	0.202268	0.834308
LinearComplexity	0.334538	0.595549	0.129620	0.289667

- **5-bit post-processing:** only passes 6 tests out of 15
- **7-bit post-processing:** only passes 7 tests out of 15
- **9-bit post-processing:** passes 13 tests out of 15

The p-values for each frequency are represented in figure 3.3 through boxplots. A boxplot is a tool used in descriptive statistics to depict groups of numerical data graphically without being parametric. This means that the data is represented without assuming that it follows any type of distribution. This is interesting because classical statistics is usually based on the normal distribution. In that situation, the *mean* of the data is used to describe the central tendency of the data set. However, when a normal distribution cannot be assumed this is no longer a useful value and it is instead replaced by the *median*. If, as in our case, the number of data to be represented is odd, the median is the middle observation when the data is ranked from smallest value to largest or vice versa.

A boxplot is also based on the *interquartile range* of the data which is the range between the lower quartile and the upper quartile values of the data. The median of the whole data divides it in half, then the median of the half with the smaller values is also the lower quartile and the median of the other half becomes the upper quartile. The IQR is then the difference between the upper and lower quartiles. The median of the whole data and the IQR are used to draw the box: its width is not relevant but its height is

Table 3.6: Central Position 500KHz

Test	PP3	PP5	PP7	PP9
Frequency	0.000000	0.000000	0.779188	0.678686
BlockFrequency	0.000000	0.003712	0.834308	0.129620
CumulativeSums	0.000000	0.000002	0.474986	0.032923
CumulativeSums	0.000000	0.000000	0.062821	0.851383
Runs	0.000000	0.000000	0.000002	0.419021
LongestRun	0.000000	0.262249	0.162606	0.816537
Rank	0.455937	0.010237	0.002374	0.115387
FFT	0.000000	0.191687	0.798139	0.494392
OverlappingTemplate	0.000000	0.911413	0.851383	0.419021
NonOverlappingTemplate	0.023762	0.481288	0.475329	0.438327
Universal	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.000000	0.014550	0.014550	0.003712
Serial	0.000000	0.834308	0.574903	0.437274
Serial	0.137282	0.678686	0.971699	0.289667
LinearComplexity	0.897763	0.334538	0.616305	0.289667

Table 3.7: Central Position 1MHz

Test	PP3	PP5	PP7	PP9
Frequency	0.000000	0.000000	0.637119	0.911413
BlockFrequency	0.000000	0.122325	0.000406	0.437274
CumulativeSums	0.000000	0.000000	0.249284	0.494392
CumulativeSums	0.000000	0.000000	0.474986	0.897763
Runs	0.000000	0.000000	0.000000	0.003201
LongestRun	0.000000	0.437274	0.000002	0.935716
Rank	0.071177	0.262249	0.494392	0.924076
FFT	0.000000	0.319084	0.001030	0.978072
OverlappingTemplate	0.000000	0.455937	0.236810	0.153763
NonOverlappingTemplate	0.007652	0.472842	0.444217	0.501438
Universal	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.000000	0.000347	0.000000	0.304126
Serial	0.000000	0.616305	0.202268	0.616305
Serial	0.191687	0.637119	0.129620	0.401199
LinearComplexity	0.366918	0.129620	0.616305	0.759756

Table 3.8: Central Position 25MHz

Test	PP3	PP5	PP7	PP9
Frequency	0.000000	0.366918	0.616305	0.171867
BlockFrequency	0.000000	0.224821	0.366918	0.066882
CumulativeSums	0.000000	0.236810	0.419021	0.153763
CumulativeSums	0.000000	0.275709	0.964295	0.015598
Runs	0.000000	0.035174	0.514124	0.616305
LongestRun	0.000000	0.637119	0.798139	0.062821
Rank	0.401199	0.419021	0.191687	0.129620
FFT	0.554420	0.171867	0.759756	0.455937
OverlappingTemplate	0.037566	0.699313	0.779188	0.122325
NonOverlappingTemplate	0.239008	0.479693	0.526375	0.495125
Universal	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.000000	0.008266	0.007160	0.153763
Serial	0.000000	0.911413	0.514124	0.137282
Serial	0.153763	0.911413	0.719747	0.834308
LinearComplexity	0.637119	0.304126	0.678686	0.997823

Table 3.9: Central Position 50MHz

Test	PP3	PP5	PP7	PP9
Frequency	0.000000	0.000000	0.000000	0.534146
BlockFrequency	0.000000	0.006661	0.000000	0.401199
CumulativeSums	0.000000	0.000000	0.005762	0.055361
CumulativeSums	0.000000	0.000000	0.000004	0.275709
Runs	0.000000	0.000000	0.000000	0.554420
LongestRun	0.000000	0.000060	0.000000	0.289667
Rank	0.009535	0.075719	0.401199	0.006661
FFT	0.000000	0.108791	0.574903	0.262249
OverlappingTemplate	0.000000	0.008879	0.798139	0.657933
NonOverlappingTemplate	0.047825	0.449661	0.432256	0.49566
Universal	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.000000	0.000000	0.000000	0.010237
Serial	0.000000	0.181557	0.554420	0.494392
Serial	0.000005	0.401199	0.494392	0.275709
LinearComplexity	0.719747	0.678686	0.616305	0.350485

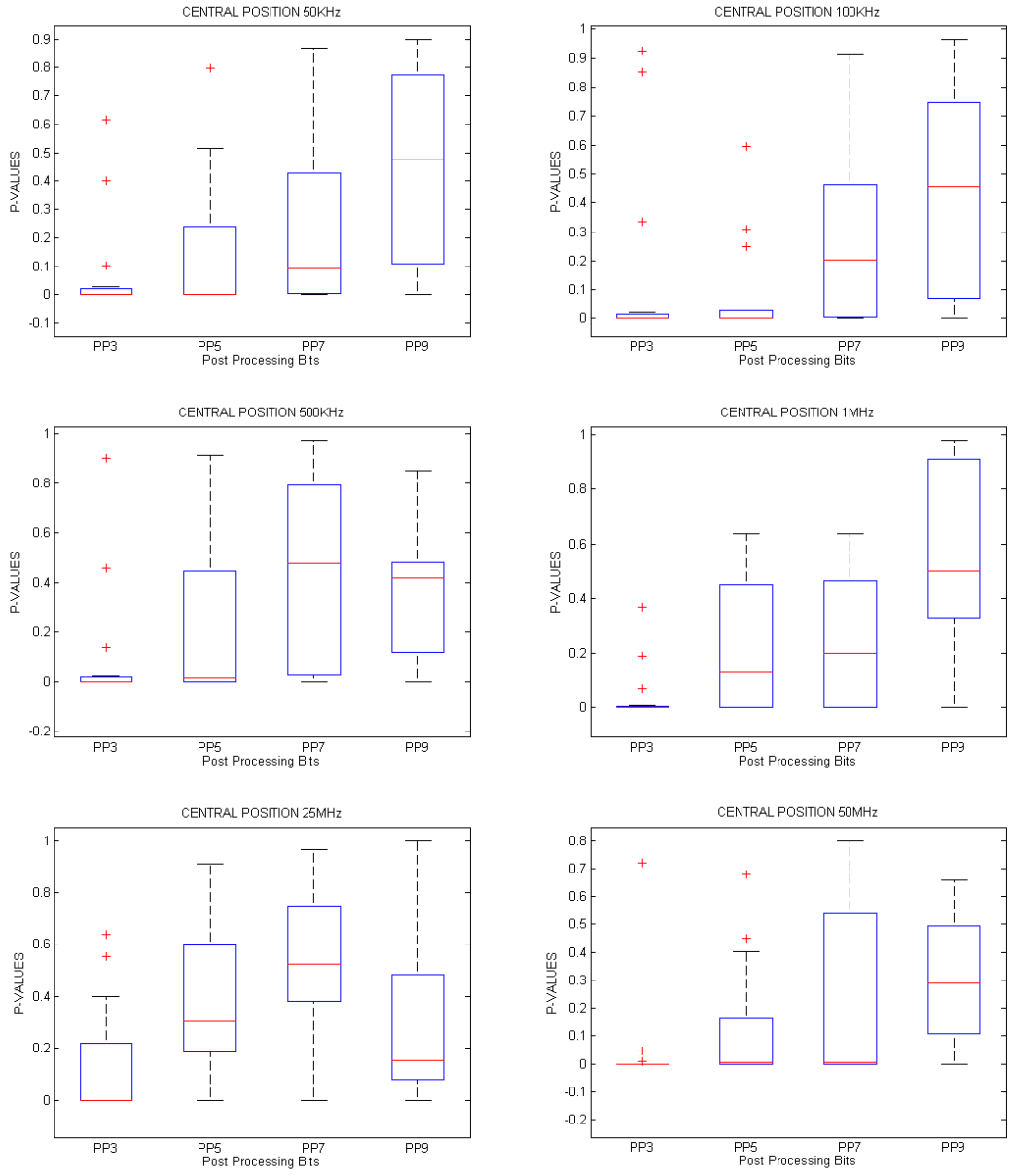

Figure 3.3: Tested Frequencies in Central Position

determined by the lower and upper quartiles and a horizontal bar is drawn at the position of the median. The values that do not fall within the IQR are represented as whiskers. The whiskers however, do not represent the actual data but are dependent on the IQR, their range is: $[-1.5IQR; +1.5IQR]$. The box should more or less be centered in the whiskers. If this does not happen and there are value exceedingly larger or smaller than the limits of the whiskers they are represented as asterisks and referred as extremes [48].

The values being represented are the p-values obtained for each statistical test at each frequency with the four XOR correctors. A p-value is expected to take a value from 0 to 1 but for the TRNG to pass the tests the p-value must be greater than 0.01. The furthest the lower quartile is from 0 the better the results of the tests. The assumption made in 3.2.2.1 that the greater the bits of the XOR corrector the better the randomness can be observed.

The results obtained with a 3-bit XOR are still defective for all frequencies. The 5-bit post-process starts yielding better results for those frequencies near 1MHz but they are still insufficient. At 7-bits of post-processing the results become considerably satisfactory but it is still found that the lower quartile is too close to zero (if not zero) for most cases. With an XOR corrector of 9-bits it is finally seen that the frequencies higher than 1MHz degrade while the lower quartile of those below it is still near 0.

Next, a closer comparison of the post-processing of 7-bits and 9-bits is provided in tables 3.10 and 3.11 and figures 3.4 and 3.5.

A couple of anomalies can be noted:

1. Graphically, at 500KHz it seems that a 7-bit XOR produces better results than a 9-bit one. However, as can be seen in the p-values of table 3.6, the 9-bit post-process does pass more tests than the 7-bit XOR.

2. At 25MHz and a post-process of 7-bits the p-values obtained are drastically improved and rival those of the 9-bit XOR at 1MHz. However, the fact that the results for 25MHz quickly degrade at a higher post-processing makes this frequency unreliable. It can be concluded that what is being gained in randomness does not compensate the throughput penalty.

Theoretically, at lower frequencies better results should be obtained since there is more time for accumulating jitter. In spite of this, at some point this backfires because the lower the frequency

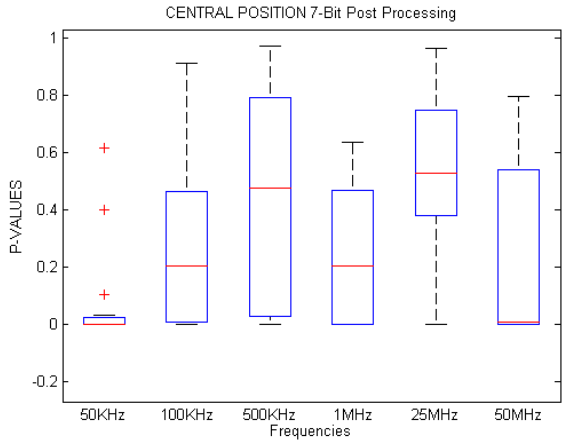

Figure 3.4: Comparison of 7-bit XOR of central position

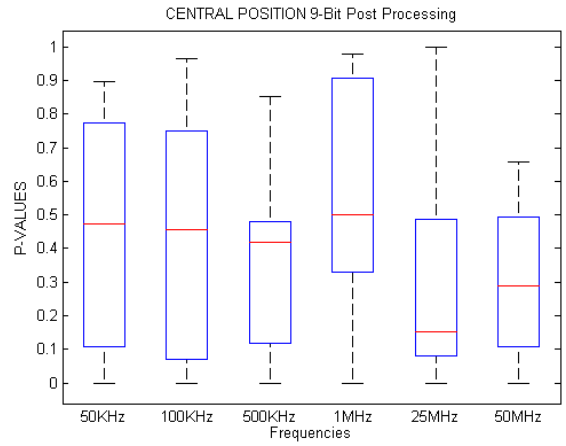

Figure 3.5: Comparison of 9-bit XOR of central position

the greater the loss of throughput. Therefore, it was then concluded that the optimal frequency was indeed 1MHz with a 9-bit XOR corrector.

Table 3.10: Central Position Post-Processing of 7-bits

Test	50KHz	100KHz	500KHz	1MHz	25MHz	50MHz
Frequency	0.000000	0.759756	0.779188	0.637119	0.616305	0.000000
BlockFrequency	0.000000	0.759756	0.834308	0.000406	0.366918	0.000000
CumulativeSums	0.000000	0.911413	0.474986	0.249284	0.419021	0.005762
CumulativeSums	0.000000	0.006196	0.062821	0.474986	0.964295	0.000004
Runs	0.000000	0.000000	0.000002	0.000000	0.514124	0.000000
LongestRun	0.000000	0.004301	0.162606	0.000002	0.798139	0.000000
Rank	0.616305	0.304126	0.002374	0.494392	0.191687	0.401199
FFT	0.000000	0.419021	0.798139	0.001030	0.759756	0.574903
OverlappingTemplate	0.000000	0.115387	0.851383	0.236810	0.779188	0.798139
NonOverlappingTemplate	0.029305	0.480633	0.475329	0.444217	0.526375	0.432256
Universal	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.000000	0.000097	0.014550	0.000000	0.007160	0.000000
Serial	0.000000	0.289667	0.574903	0.202268	0.514124	0.554420
Serial	0.401199	0.202268	0.971699	0.129620	0.719747	0.494392
LinearComplexity	0.102526	0.129620	0.616305	0.616305	0.678686	0.616305

Table 3.11: Central Position Post-Processing of 9-bits

Test	50KHz	100KHz	500KHz	1MHz	25MHz	50MHz
Frequency	0.699313	0.657933	0.678686	0.911413	0.171867	0.534146
BlockFrequency	0.366918	0.213309	0.129620	0.437274	0.066882	0.401199
CumulativeSums	0.897763	0.935716	0.032923	0.494392	0.153763	0.055361
CumulativeSums	0.759756	0.494392	0.851383	0.897763	0.015598	0.275709
Runs	0.867692	0.005358	0.419021	0.003201	0.616305	0.554420
LongestRun	0.739918	0.964295	0.816537	0.935716	0.062821	0.289667
Rank	0.090936	0.037566	0.115387	0.924076	0.129620	0.006661
FFT	0.042808	0.058984	0.494392	0.978072	0.455937	0.262249
OverlappingTemplate	0.883171	0.108791	0.419021	0.153763	0.122325	0.657933
NonOverlappingTemplate	0.473554	0.501248	0.438327	0.501438	0.495125	0.49566
Universal	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.162606	0.455937	0.003712	0.304126	0.153763	0.010237
Serial	0.012650	0.779188	0.437274	0.616305	0.137282	0.494392
Serial	0.779188	0.834308	0.289667	0.401199	0.834308	0.275709
LinearComplexity	0.334538	0.289667	0.289667	0.759756	0.997823	0.350485

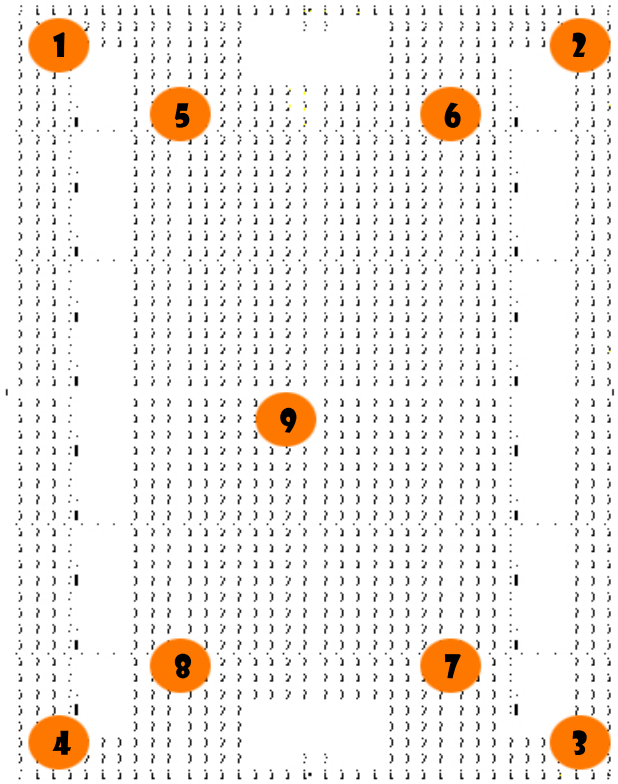

Figure 3.6: Tested Positions of the FPGA

3.3.2 Intradevice Testing

The second experiment evaluates the quality of the TRNG in several positions within the same FPGA. As explained previously, the routing for the design is always the same because of the Hard Macro, thus it would be possible to conclude that the results obtained for each position depend solely on its specific position on the hardware.

The chosen positions needed to be representative of the layout of the Spartan-3E board.Therefore, it was decided to test the nine different positions shown in figure 3.6 by selecting the required pins for the Hard Macro. The exact locations used are specified in the *ucf* file of the implementation of the design in appendix A. Notice that the ninth central position is also where the TRNG was placed for the previous experiment, as such its measurements have already been obtained and there is no need to repeat them.

In the previous experiment it was confirmed that the optimal oscillation frequency was 1MHz as [1] claimed. Moreover, it was

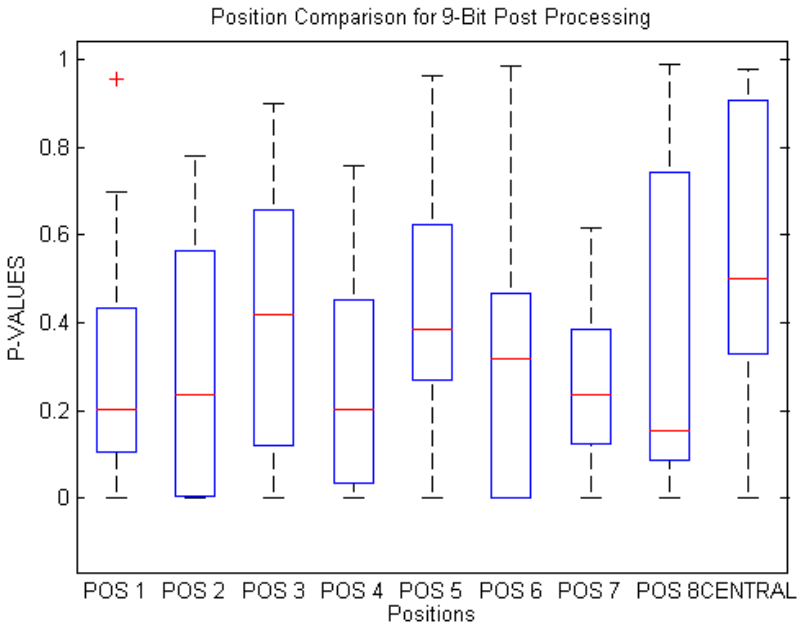

Figure 3.7: Intradevice Test: 1MHz and Post-Procressing of 9-Bits

found that employing a 9-bit XOR operation in the post-process provided the best results. As such, this experiment used a frequency of oscillation of 1MHz for the RO to generate random numbers and a XOR corrector of 9 bits as its post-processing mechanism.

The results obtained are presented in figure 3.7, for the numerical values obtained for the p-values refer to table 3.12 from which we can see that:

- **Position 1:** passes 13 tests
- **Position 2:** passes 11 tests
- **Position 3:** passes 13 tests
- **Position 4:** passes 12 tests
- **Position 5:** passes 10 tests
- **Position 6:** passes 10 tests
- **Position 7:** passes 13 tests
- **Position 8:** passes 14 tests
- **Position 9:** passes 13 tests

However, it has to be taken into account that at this point, we are not just looking at the number of tests that the TRNG passes but also the quality of those p-values. While the central ninth position does not pass the Runs test and the eighth position does, in general, the p-values of the central position are far better than any other. This can be noticed in figure 3.7 where, with the exception of the fifth and ninth positions, all the other positions present a very small lower quartile. In spite of this, it can be seen that for the fifth position although the p-values obtained for the tests that pass are considerably good, it fails several tests. This can be explained according to the research in [41] that discovered a wide variation in the intrinsic speed of FPGAs of approximately a 7% between the normalized speeds of the slowest CLB and the fastest CLB.

It is not surprising that centering the TRNG in the board yields better results. In spite of using Hard Macros, the connections with other elements can influence the jitter. At the center, the synthesizer has more space to place other elements in a natural way that would not interfere with the TRNG. Therefore, it has been concluded that the central position of the FPGA is the best one to place the TRNG.

Table 3.12: All positions Post-Processing of 9-bits

Test	P1	P2	P3	P4	P5	P6	P7	P8	P9
Frequency	0.401199	0.145326	0.897763	0.202268	0.366918	0.419021	0.236810	0.816537	0.911413
BlockFrequency	0.213309	0.616305	0.102526	0.514124	0.350485	0.080519	0.145326	0.085587	0.437274
CumulativeSums	0.040108	0.534146	0.678686	0.455937	0.759756	0.000700	0.181557	0.798139	0.494392
CumulativeSums	0.419021	0.494392	0.108791	0.759756	0.383827	0.983453	0.115387	0.304126	0.897763
Runs	0.122325	0.000000	0.162606	0.000000	0.000000	0.000000	0.319084	0.153763	0.003201
LongestRun	0.699313	0.040108	0.474986	0.096578	0.437274	0.004301	0.171867	0.987896	0.935716
Rank	0.145326	0.699313	0.798139	0.016717	0.334538	0.419021	0.000031	0.090936	0.924076
FFT	0.437274	0.779188	0.851383	0.437274	0.514124	0.319084	0.474986	0.759756	0.978072
OverlappingTemplate	0.955835	0.574903	0.595549	0.437274	0.249284	0.946308	0.080519	0.145326	0.153763
NonOverlappingTemplate	0.491438	0.490924	0.461254	0.487463	0.48546	0.48408	0.479589	0.49665	0.501438
Universal	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.102526	0.000043	0.213309	0.002971	0.013569	0.000001	0.319084	0.010237	0.304126
Serial	0.202268	0.000757	0.032923	0.304126	0.657933	0.090936	0.401199	0.699313	0.616305
Serial	0.066882	0.236810	0.401199	0.122325	0.883171	0.816537	0.334538	0.080519	0.401199
LinearComplexity	0.202268	0.021999	0.419021	0.153763	0.964295	0.350485	0.616305	0.153763	0.759756

3.3.3 Interdevice Testing

Having proved that the best place to implement the TRNG on the Xilinx Spartan-3E is its central area with an oscillation frequency of 1MHz and a 9-bit XOR post-processing mechanism, the third experiment consists on evaluating the TRNG with these characteristics in several FPGAs of the same model.

In figure 3.8 the results of this experiment are presented. Again, for the numerical resulting p-values of the statistical test suite, refer to table 3.13.

Table 3.13: Several FPGA, 1MHz, Post-processing of 9-bits

Test	Original	FPGA1	FPGA2	FPGA3
Frequency	0.911413	0.574903	0.224821	0.883171
BlockFrequency	0.437274	0.514124	0.946308	0.006196
CumulativeSums	0.494392	0.991468	0.637119	0.616305
CumulativeSums	0.897763	0.474986	0.616305	0.924076
Runs	0.003201	0.085587	0.319084	0.334538
LongestRun	0.935716	0.304126	0.115387	0.637119
Rank	0.924076	0.202268	0.350485	0.224821
FFT	0.978072	0.798139	0.202268	0.071177
OverlappingTemplate	0.153763	0.867692	0.419021	0.319084
NonOverlappingTemplate	0.501438	0.468086	0.524918	0.518739
Universal	0.000000	0.000000	0.000000	0.000000
ApproximateEntropy	0.304126	0.000051	0.007160	0.037566
Serial	0.616305	0.137282	0.108791	0.108791
Serial	0.401199	0.350485	0.978072	0.971699
LinearComplexity	0.759756	0.062821	0.514124	0.350485

The four tested devices pass the same number of tests however, while the other three newly tested devices generate similar results the quality of the obtained p-values is far from that of the first device evaluated. The fact that they are similar among themselves leads to believe that the difference with the last one is not due to these devices being faulty but because of other external sources. The environment of which the experiments were conducted was not a controlled one. Moreover, as was stated back in section 2.3, a TRNG can suffer modifications not just from a malicious attack but also from environmental changes. This was not taken into account for the duration of this project which was only focused on the quality of the generator in ideal conditions. It is most likely that the measurements of these other three devices were taken during one of the many heat waves that struck Spain the summer of 2015.

It would be recommended to repeat this experiment in the fu-

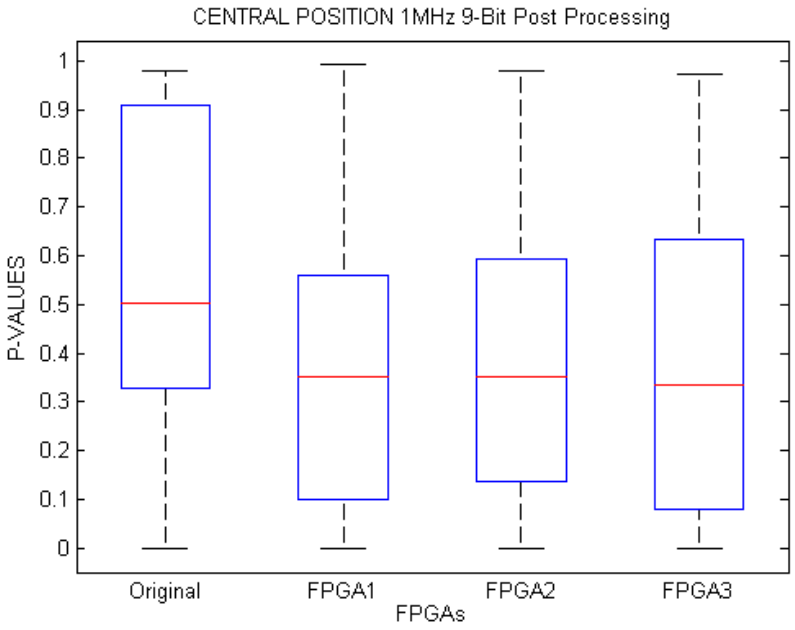

Figure 3.8: Interdevice Test: 1MHz and Post-Procressing of 9-Bits

ture with more devices and in a controlled environment to be able to assure that random numbers generated are consistent in this technology.

3.3.4 Restart Experiment

The final experiment is separate from the results obtained in the previous ones and instead applies the idea proposed in [49]. The only conditions necessary were that the TRNG was captured while it ran at a position and a frequency where the tests had been confirmed to pass. The TRNG is restarted from the same initial conditions before it started oscillating. This can be performed easily because in the design of the TRNG (see section 3.1 above) the first inverting element is a NAND gate rather than an inverter. This was done so that the oscillations could be controlled and stopped if necessary. This property added to the RO by the NAND gate is the focus of study of this experiment.

As can be seen in the VHDL code in appendix A, the oscillation only occurs when the *reset* trigger of the circuit is active. If the reset trigger is turned off the oscillation stops and the TRNG simply

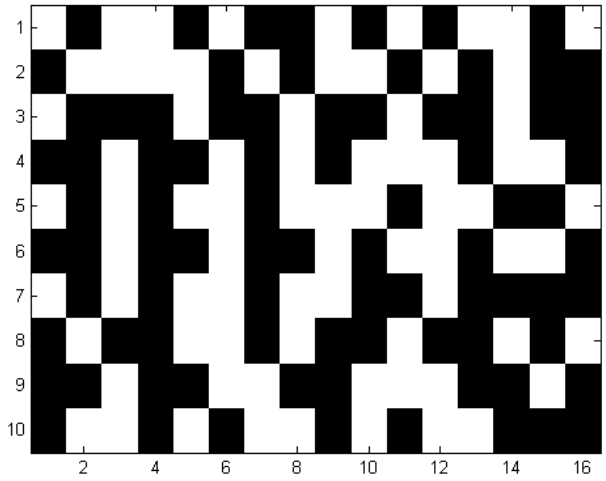

Figure 3.9: Restart Experiment Results

produces a *null* result. When the trigger is turned on again and thus the *reset* signal is set to 1, the RO starts oscillation again. The question is: is the quality of the numbers generated after restarting the oscillation still adequate?

To appraise the quality of the TRNG through the statistical test suite is pointless. The reset trigger was pulled many times while the sequence generated was being captured, therefore if the statistical tests were to be applied several problems would have been encountered:

1. Many long substrings of zeroes would be found in the whole sequence. This would immediately make the tests fail.

2. Deleting the long substrings of zeroes would not we useful. Erasing them from the sequence would mean feeding the tests fake data.

3. Simply partitioning the sequence is also out of the question. The file captured is much smaller than those used in the previous experiments, if the sequence was divided its parts would most likely not be long enough for the statistical test.

4. Capturing longer periods of data in between restarts is also ineffective. To produce a sequence nearly as long as those captured in the previous experiments the time required would be enormously higher. Not just that, the number of restarts for

the experiment would have had to be reduced which would have also led to an unreliable result.

To avoid all of these problems, the approach taken to test the behavior of the generator after the reset becomes active again has been a graphical one. The first substrings of bits generated after each restart have been represented in a couple of ways in search of a pattern that would indicate that the sequence could be guessed. By paying close attention to figures 3.9 and 3.10, which are not also a different way of representing the bits but also different iterations of the restart experiment, no such pattern has been observed. Another way of ensuring this would be to capture a large sample of bits and then check if there is a correlation between them. This can be done by modifying the design of our TRNG to program automatic restarts and then only send a certain number of bits after each reset occurs. Nevertheless, it can be concluded that the TRNG is secure against any number of restarts.

3.4 Final Analysis

These experiments have proven that [1] was indeed correct when it claimed that 1MHz was an optimal frequency for this sort of TRNG however, using a 3-bit XOR corrector in the post-processing like in [2] is not enough for this technology.

What's more, from the results of the second experiment, it is not suggested to implement this generator in any other location of the Xilinx Spartan-3E but in its central area. While the fourth experiment proves that the RO is resilient against possible restart attacks, it is, however, not possible to assess it reliability against environmental changes according to the results obtained in the third experiment.

In the following chapter, this analysis is expanded along with possible future developments to solve the weaknesses of this TRNG.

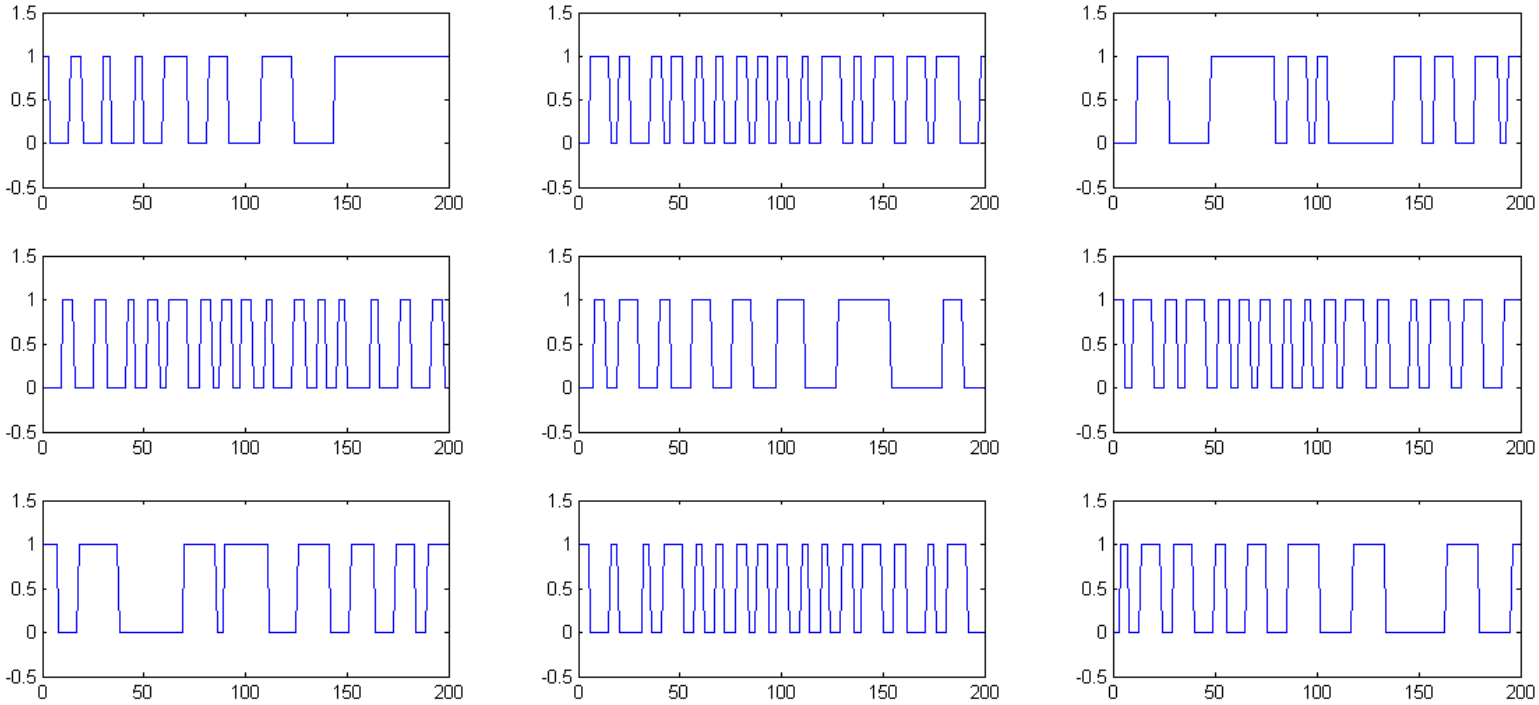

Figure 3.10: Another Restart Experiment Results

Chapter 4

Conclusions

Random Number Generators are a vital component in cryptographically secure applications nowadays, even more if those numbers can be attested to be the result of *true* randomness. A true random number generator is however not so easy to implement which is why it was so appealing to evaluate Böhl's simplified design of a ring oscillator-based TRNG.

Certain properties of this generator were considered to be worth evaluating and at the beginning of this project some goals were set (as was seen in section 1.2). Not only have the experiments performed fulfilled those goals but also, due to the optimizations in the post-processing mechanism, it has been possible to expand on those goals and test other behavior of the TRNG that had not originally been contemplated.

4.1 Conclusion

As stated above, a certain number of goals were set and achieved. Here is a summary of the results obtained:

- **Original Goals:**

    1. Prove whether or not the optimal frequency of oscillation for the RO is 1MHz by evaluating the quality of the output at several frequencies of oscillation → It is expected that the lower the frequency the better the results because there is more time for jitter accumulation. However, for frequencies lower that 1MHz the throughput losses become too high which reflects negatively in the performance generat-

ing less random numbers per second. As such, it has been concluded that 1MHz is certainly the optimal frequency.

2. Expand on the use [2] gives to the TRNG on FPGA technology by testing its performance in different areas of the same device → The generator is the most reliable when it is placed in the central area of the Spartan-3E. The reason for this is that there is a variation of about a 7% in the intrinsic speed between the slowest and the fastest CLBs of an FPGA. In the implementation of the TRNG, this occurs because the connections with other nearby elements can affect jitter. At the center of the board the TRNG is more likely to avoid being influenced by other elements because there is more space to place them.
3. Evaluate the quality of the TRNG with and without a post-processing mechanism → Post-processing is necessary as the generator does not pass the statistical tests without it.
4. Use XOR correctors of 3-bits and more in the post-process to check their effect on the sequence → The greater the number of bits of the XOR corrector, the better results. It results in a much more effective reduction of bias due to the higher compression rate. Moreover, an attacker will find all the more difficult to guess the original bit sequence.

- **Extra Goals:**

5. Test the quality of results in different FPGAs of the same model → It could be guessed that the generator produces a similar output, however, a definitive answer cannot be provided due to environmental changes that have most likely altered the jitter of the RO.
6. Test how the TRNG behaves when a restart occurs and whether or not it is still robust → After any number of restarts of the circuit the generator does not produce any distinguishable patterns. Therefore, the output of the generator cannot be guessed.

The design and implementation of this TRNG on FPGA technology is not only quite simple but also saves many resources. This is because of its reduced number of elements which are also very basic: just eight inverters, a NAND gate and three DFFs.

It has been possible to prove that this generator is resilient against possible restart attacks, however, as was demonstrated in the third

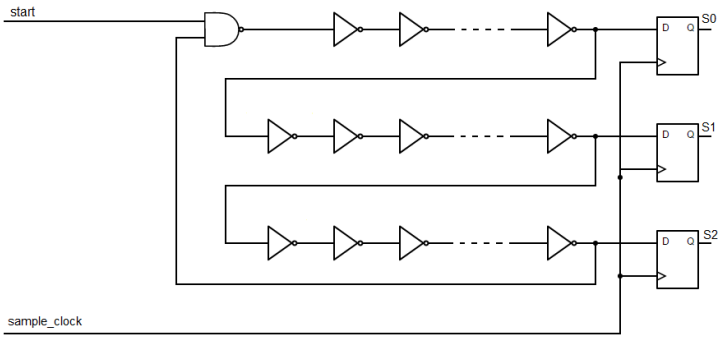

Figure 4.1: Modified TRNG design with more inverters

experiment in section 3.3.3, it seems weak against other kinds of external tampering such as those produced by environmental changes. Moreover, the TRNG by itself is not enough to provide randomness as it did not pass the tests by itself and required a post-processor. In [2] the post-processing mechanism consisted on a XOR corrector of 3 bits; however the results obtained in this project for that corrector were disastrous. An XOR operation that took a longer string of bits was required to cover the bias of the generator. It was found that with a 7-bit XOR the results started to become acceptable however at least 9-bits are required to obtain a truly reliable random sequence at the optimal oscillation frequency of 1MHz.

Nevertheless, it cannot be assured that these weaknesses that the TRNG has presented are entirely because of its design since the results obtained in [1] and [2] present it as a robust solution. It is entirely possible that these vulnerabilities have appeared as a result of the variability of the target device which in turn has affected the performance of the RO. This is demonstrated through the results obtained in experiment 3.3.2. It can be seen that the behavior of the TRNG varies greatly depending on the area of the hardware on which it is placed. By turning the TRNG into a Hard Macro it was assured that the distribution of the RO and its routing were constant, therefore it has to be concluded that the hardware in which the TRNG was implemented may not advantageous.

4.2 Future developments

It has been concluded that, while the TRNG has been proved to be easy to implement and can provide a reliable random sequence, it

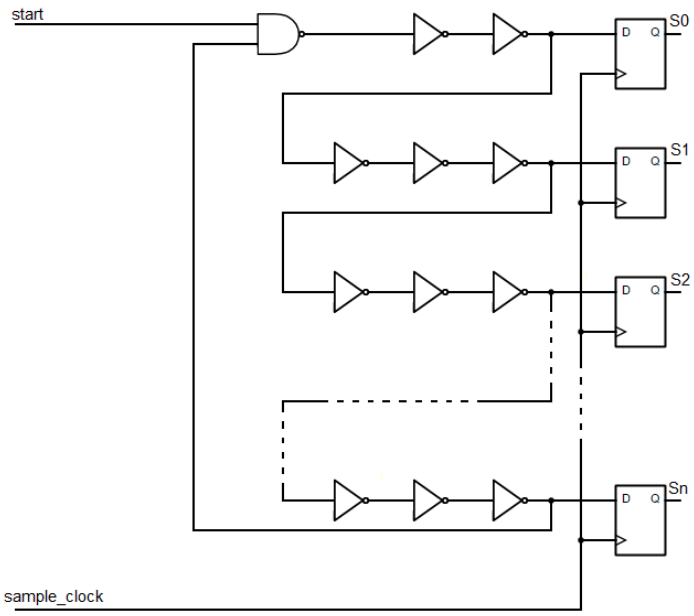

Figure 4.2: Modified TRNG design with more inverters and more sampling stages

is not flawless. It is very likely that the vulnerabilities it presents are caused by the properties and variability of the FPGA model on which it has been tested. To be able to ascertain that the hardware is to blame for the vulnerabilities found, it would be desirable to implement this design in several FPGA devices and perform the same experiments to be able to compare the behavior of the TRNG.

Taking into account the results of experiment 3.3.3, performing said experiments in a controlled environment and test the behavior of the TRNG at several temperatures would also be recommended. That way modifications in the random sequence by external natural processes could be analyzed and counteract if necessary. Moreover, as was stated at the end of section 3.3.4 it would be interesting to perform the Restart experiment again by modifying the design to automatically restart the oscillation and then send a certain number of bits instead of doing it manually.

However, as the TRNG has been proven a good solution given that a strong enough post-processing stage is implemented as well as very resistant against resetting its oscillation, there might still be room for improvement.

In 3.1 it was mentioned that it was possible to either XOR the stored bits in the three DFFs or simply output them because each

inverting element would have already added enough phase shift to produce a random bit in each inverting stage. It was chosen to take this second approach and it may be the source of many weakness encountered. Consequently, several possible solutions to solve this can be implemented by changing slightly the design:

1. Try the other approach and XOR the 3 bits in the DFFs. The resulting bit will be the output of the TRNG.
2. Add more inverting elements to the design. The more inverting stages the higher the phase shift and jitter accumulated which results in better randomness with much better quality. And it would also be possible to XOR the bits sampled by the DFFs if necessary. Although it needs to be taken into account that the optimal frequency of oscillation would no longer be 1MHz. There is a couple of ways to change the design through this approach:
   (a) Add more inverters before each sampling stage. This would ensure that the jitter accumulated before sampling has a better quality. Currently the design has only 3 inverting elements before a bit is stored in a DFF which might be also the reason for an unsatisfactory output. This would change the design from figure3.1 to 4.1.
   (b) Add more inverters so that also more sampling stages can be implemented. It could be possible to do this by still keeping three inverting stages in between each DFF, changing the design to fit figure 4.2, or adding more inverting stages like in the previous proposal to accumulate more jitter. This approach however would drain more resources than the previous ones.

A last consideration would be to study the behavior of the post-processing mechanism. From the reference work in [2], it was decided to try a 3-bit XOR corrector but it turned out that to obtain appropriate results over 7-bits were required. There are also some possible future developments to derive from this:

1. An XOR corrector might not be the most adequate mechanism and without changing the circuit another post-processing methodology could be studied. Instead of an XOR, in [1] the von Neumann algorithm and resilient functions are mentioned as possible post-processing techniques. Moreover, the author

offers a possible novel design for a post-processing circuit that would raise an alarm if an attack is detected.

2. It is likely that if the TRNG generated a better output by itself the 3-bit XOR would suffice. From what has been proposed previously, if the design of the circuit was to be changed and its quality improved the 3-bit XOR would have to be evaluated again but it can be expected to yield better results.

3. It also has to be considered that if implementing the current design of the TRNG in a different FPGA resulted in an output of higher quality, there are many chances that the 3-bit corrector would be efficient.

In case all these failed, another possibility to ensure if the Xilinx Spartan-3E device is a good one to implement TRNGs would be to use a completely different type of design. In section 2.5 another common type of TRNG for FPGAs was introduced: Self-Timed Rings. Compared to ROs, STRs are much more resistant to variabilities in voltage and are capable to produce jitter with a higher quality because the deterministic noise generated in the process is largely diminished.

Chapter 5

Legal Aspects

IT products are bound to certificates and standards released by official organizations such as the International Organization for Standardization. In the case of security in IT there are more specific standards such as the Common Criteria for Information Technology Security Evaluation (also known as just Common Criteria). The CC is in fact the first truly international security standard as it originated by combining three previous standards: the European Information Technology Security Evaluation Criteria which was compiled in 1990, the Trusted Computer System Evaluation Criteria which was issued by the Department of Defense of the United States in 1983 and last but not least the Canadian Trusted Computer Product Evaluation Criteria which was itself a combination of the ITSEC and the TCSEC published in 1993.

However, despite the importance of RNGs in numerous cryptographic applications, the CC does not provide in its evaluation methodology (the Common Evaluation Methodology) uniform criteria to assess RNGs, nor do the previous standards on which it is based.

In spite of this, as was stated in chaper 2, it is acknowledged that for a RNG to be considered secure it has to pass several statistical tests. These are not stablished by the CC itself but by national official testing laboratories that follow the standard ISO 17025 for testing and calibration. Some of these laboratories are: the National Institute of Standards and Technology (NIST) in the United States, the Bundesamt für Sicherheit in der Informationstechnik (BSI) in Germany, United Kingdom Accreditation Service (UKAS), the Standards Council of Canada, the Comité français d'accréditation (COFRAC) in France and the National Cryptologic

Center (CCN) in Spain.

Out of the batteries of tests for TRNGs presented in section 2.3.1 the official standards presented by the NIST and the BSI are:

- **FIPS 140-2:** a standard by the NIST that acknowledges 4 levels of security although it does not specify what level a particular cryptographic application requires.
  - **Level 1:** the lowest level of security. The requirements it imposes are limited and are loosely defined. all components must be "production-grade" and various egregious kinds of insecurity must be absent
  - **Level 2:** on top of the level 1 security, it adds requirements for role-based authentication as well as tampering evidence physically.
  - **Level 3:** on top of level 2, it adds requirements for identity-based authentication, more precise measures against physical tampering of evidence and a need to separate interfaces of the module.
  - **Level 4:** the top level insists on the need of robustness against environmental attacks.
- **NIST SP 800-90:** This has been the standard that has been followed in the project by using the NIST statistical test suite, however it deals more with PRNGs than TRNGs. It is divided in three:
  - **SP 800-90A:** recommendations for the creation of pseudo-random values by proposing deterministic algorithms with an entropy input.
  - **SP 800-90B:** recommendations for designing and implementing entropy sources.
  - **SP 800-90C:** recommendations to link the entropy source with the RNG algorithm.
- **AIS31:** a standard by the BSI that expands upon the previous AIS20. It defines a methodology to test TRNGs separately from PRNGs. It stablishes three classes of TRNGs:
  - **PTG.1:** TRNG with internal tests which detect a total failure of the entropy source as well as non-tolerable statistical defects of the internal generated numbers.

- **PTG.2:** TRNG that fulfills PTG.1 and adds a stochastic model of the entropy source and statistical tests of the random raw numbers (instead of the internal random numbers).
- **PTG.3:** TRNG that fulfills PTG.2 with cryptographic postprocessing.

While the NIST has not yet approved any TRNGs, the BSI has approved several such as Schindler's generator based on noise diodes [50] and the so called *Quantis AIS 31 Validated RNG* [51].

Chapter 6

Project Management

Any type of project should be monitored from the very beginning, not just during the span of the activities performed to reach the desired goals but also even before the work has started to set its objectives, decide the materials that will be needed and calculate its costs and time it will take to complete it. This chapter focuses on this process by first showing the management of tasks in time and afterwards the costs of materials and personnel.

6.1 Planning

In this section an oversight of the distribution of time to finish the different tasks of the project is presented.The time expected to be spent developing a task is difficult to calculate before the project starts, therefore the initial naive estimation is provided followed by the actual final time spent in each task that was performed.

6.1.1 Initial Planning

Since the student and person doing the research was also employed part-time during the duration of the project, a working ship for the project of 5 hours a day was established. Another time constraint that was taken into account from the very beginning was the time required to take a measurement of the output of the TRNG. In order for us to obtain reliable confirmation of its randomness with the statistical tests it was mandatory to capture at least 10MB of data per measurement which it was calculated that could take around 3 hours. Thus, initially, only interdevice tests were planned to decide where within the same hardware the TRNG solution would be the

Table 6.1: Initial Planning

Task	Duration (hours)	Personnel
Problem Analysis	20	Elena Martínez López
Working environment set-up	25	Elena Martínez López
Implement TRNG code	30	Elena Martínez López
Taking meassurements	48	Elena Martínez López
Postprocessing	174	Elena Martínez López
Result Analysis	45	Elena Martínez López
Supervision	25	Honorio Martín
Documentation	100	Elena Martínez López
TOTAL	**467**	

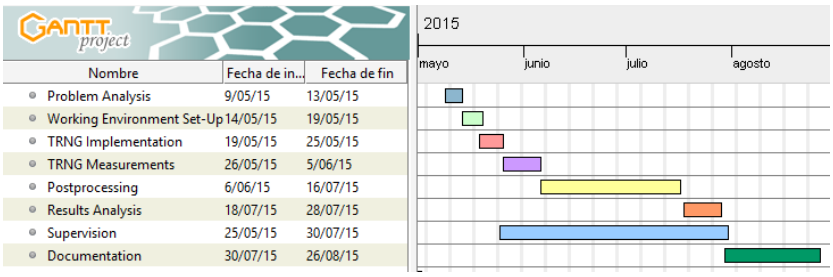

Figure 6.1: Initial Gantt

most robust. Lastly, Böhl already mentioned in his documentation the need of a post processing to improve the results of the TRNG therefore it was decided to XOR the bits in groups of 3, 5, 7 and 9. This, however would increase the time spent taking measurements greatly as it would require to repeat the captures of data 4 times.

Aside from those constraints the rest of the technical tasks, such as understanding Böhl's proposal, implementing it and treating its output once it was captured, were calculated to take a moderately short period of time of at most a week each. Notice that the time estimated for the project supervision, unlike that of the rest of tasks, is approximated and is actually consisting in tutoring sessions of about an hour or an hour and a half each scheduled throughout the whole duration of the project which is why in 6.1 it is shown to take such a long period of time.

6.1.2 Final Planning

Some considerable amount of time was wasted because of problems with the installation of the IDE Design Suite for Windows 8 and implementing a Hard Macro so that the process of moving the TRNG around the hardware was easier as well as some measures requiring

Table 6.2: Final Planning

Task	Duration (hours)	Personnel
Problem Analysis	20	Elena Martínez López
Working environment set-up	30	Elena Martínez López
Implement TRNG code	40	Elena Martínez López
Taking meassurements	45	Elena Martínez López
Postprocessing	20	Elena Martínez López
Restart Experiment	15	Elena Martínez López
Result Analysis	75	Elena Martínez López
Supervision	25	Honorio Martín
Documentation	100	Elena Martínez López
TOTAL	**370**	

to be captured again since they were faulty. However, due to the computer that was acquired for the project, the time estimated for capturing the output data was reduced from 3 hours to around an hour and a half for each measurement. What's more, the Python script that was created for post processing the numbers generated made repeating the measurements unnecessary, therefore ensuring that a lot of time was gained. These conditions allowed to introduce intradevice tests and the restart experiment. All of this is reflected in figure 6.2.

6.2 Budget

Just as it was necessary to initially approximate the time required to finish the project and as this is merely an investigation project and no monetary gain is derived from it it is all the more important to calculate its budget.

6.2.1 Initial Costs

6.2.1.1 Initial Estimated Material Costs

Table 6.3: Initial Estimated Costs

Resource	Quantity	Cost (€/item)
Spartan 3E Starter Board	1	199
HP 15-r213ns laptop	1	699
Matlab Student Suite	1	97
VMWare Workstation	1	228.63
TOTAL		**1223.23**

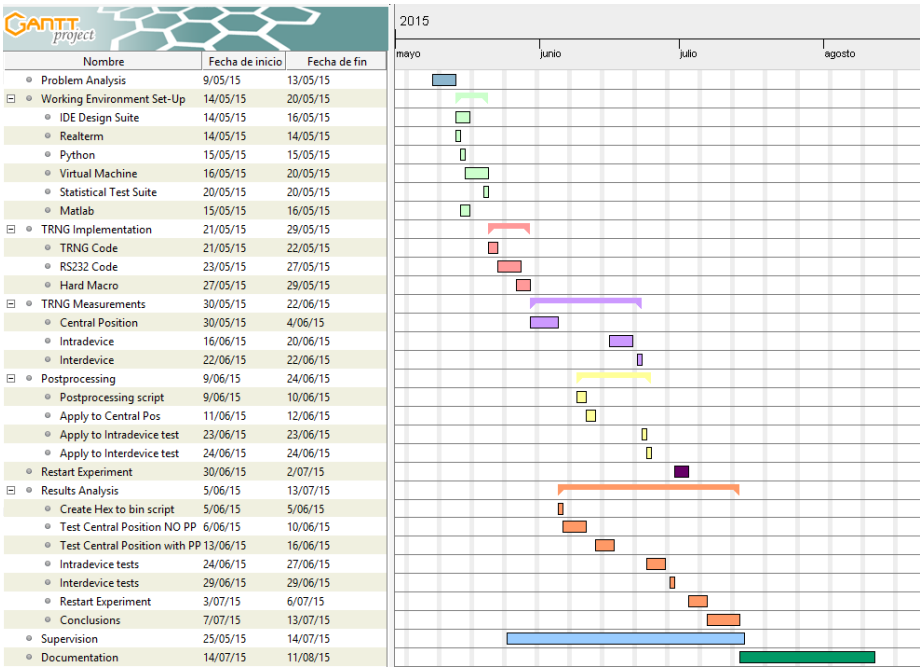

Figure 6.2: Final Gantt

Some of the software programs used, such as Python, Realterm and the Statistical Test Suite from the NIST, are either open source or free to download. Additionally, Xillinx not only allows to download their IDE Design Suite from their webpage but also provides free software licenses for students. However, Matlab and VMWare Workstation only have paid versions.

As for the hardware, investing in a new computer is key to the project, not only to set up the working environment but it also provides better resources to run a virtual machine than an older computer.

6.2.1.2 Initial Estimated Personnel Costs

Table 6.4: Initial Estimated Personnel Costs

Position	Income (€/hour)	Number of hours
Junior Engineering Investigator	13	467
TOTAL (€)		**6071**

An approximation to the salary of a junior engineering investigator for the duration of the project has been calculated from that a university department pays their part-time interns.

6.2.2 Final Costs

By 'final' this section actually refers to the real costs of the project which vary from those originally calculated in both resources used and working hours as seen in 6.1.

6.2.2.1 Final Material Costs

Table 6.5: Final Material Costs

Resource	Quantity	Cost (€/item)
Spartan 3E Starter Board	4	199
HP 15-r213ns laptop	1	600
Matlab Student Suite	1	63
TOTAL		**1459**

There have been several ways to save money:

- Student discount in HP products
- Acquiring only the basic Matlab package as well as its statistical related functions
- Instead of building a virtual machine from scratch with VMWare Workstation, a free version of VMWare Player and a prebuilt virtual machine with OS Fedora linux has been used.

In spite of this, by increasing the number of FPGAs to be tested, the budget has slightly increased by 200€.

6.2.2.2 Final Personnel Costs

Table 6.6: Final Personnel Costs

Position	Income (€/hour)	Number of hours
Junior Engineering Investigator	13	370
TOTAL (€)		**4810**

By reducing the number of working hours from 467 to 370 the costs to maintain a junior engineering investigator have plummeted. This reflects on the whole budget very favorably as the whole cost of the project has decreased.

Appendices

Appendix A

TRNG Code

The True Random Number Generator consists of a ring oscillator that is formed by a set of very simple components

Basic Components

Inverter

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith.all;
   use ieee.std_logic_unsigned.all;
entity INVERTER is
   port(
      A: in std_logic;
      B: out std_logic
   );
end INVERTER;
architecture INVARCH of INVERTER is
   begin
      B <= NOT A;
end INVARCH;
```

NAND Gate

```
library ieee;
   use ieee.std_logic_1164.all;
entity NANDGATE is
   port(
      A1: in std_logic;
      A2: in std_logic;
      B: out std_logic
```

```
   );
end NANDGATE;
architecture NANDARCH of NANDGATE is
   begin
      B <= A1 nand A2;
end NANDARCH;
```

D-FlipFlop

```
library ieee;
   use ieee.std_logic_1164.all;
entity DFLIPFLOP is
   port(
      D: in std_logic;
      Reset: in std_logic;
      Clock: in std_logic;
      Q: out std_logic
   );
end DFLIPFLOP;
architecture Sampling of DFLIPFLOP is
   begin
   process(Reset, Clock)
      begin
      if Reset='0' then
         Q <= '0';
      elsif Clock'EVENT and Clock='1' then
         Q <= D;
      end if;
   end process;
end Sampling;
```

Ring Oscillator

With the three previous components we can create our Ring Oscillator:

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith.all;
   use ieee.std_logic_unsigned.all;

entity RING is
   port(
      Clk: in std_logic;
      Rset: in std_logic;
      enable:in std_logic;
      digit0_0: out std_logic;
      digit1_0: out std_logic;
```

```
      digit2_0: out std_logic
   );
end RING;
architecture oscillator of RING is
   -- signals
   signal wire1: std_logic;
   signal wire2: std_logic;
   signal wire3: std_logic; -- first flipflop
   signal wire4: std_logic;
   signal wire5: std_logic;
   signal wire6: std_logic; -- second flipflop
   signal wire7: std_logic;
   signal wire8: std_logic;
   signal wire9: std_logic; -- third flipflop
   signal S0: std_logic;
   signal S1: std_logic;
   signal S2: std_logic;
   -- components
   component NANDGATE is
      port(
         A1: in std_logic;
         A2: in std_logic;
         B: out std_logic
      );
   end component;
   component INVERTER is
      port(
         A: in std_logic;
         B: out std_logic
      );
   end component;
   component DFLIPFLOP is
      port(
         D: in std_logic;
         Reset: in std_logic;
         Clock: in std_logic;
         Q: out std_logic
      );
   end component;
   -- to force the process
   attribute keep : string;
attribute keep of wire1,wire2,wire3,wire4,wire5,wire6,wire7,wire8,wire9:
    signal is "TRUE";
begin
   digit0_0 <= S0;
   digit1_0 <= S1;
   digit2_0 <= S2;
ngate: NANDGATE port map (A1 => enable, A2 => wire9, B => wire1);
   inv1: INVERTER port map (A => wire1, B => wire2);
   inv2: INVERTER port map (A => wire2, B => wire3); -- first dff
   inv3: INVERTER port map (A => wire3, B => wire4);
   inv4: INVERTER port map (A => wire4, B => wire5);
   inv5: INVERTER port map (A => wire5, B => wire6); -- second dff
   inv6: INVERTER port map (A => wire6, B => wire7);
```

```
   inv7: INVERTER port map (A => wire7, B => wire8);
   inv8: INVERTER port map (A => wire8, B => wire9); -- third dff
dff1: DFLIPFLOP port map (D => wire3, Reset => Rset, Clock => Clk, Q =>
    S0);
dff2: DFLIPFLOP port map (D => wire6, Reset => Rset, Clock => Clk, Q =>
    S1);
dff3: DFLIPFLOP port map (D => wire9, Reset => Rset, Clock => Clk, Q =>
    S2);

end oscillator;
```

Final Design

Turning our ring oscillator into a hard macro to easily move it around the FPGA we create a new project in which it will work alongside with the logic needed to use the serial port RS232:

```
library ieee;
   use ieee.std_logic_1164.all;
   use ieee.std_logic_arith.all;
   use ieee.std_logic_unsigned.all;
   use std.textio.all;

entity TRNG is
   port(
      CLK: in std_logic;
      RST: in std_logic;
      EN: in std_logic;
      WR: in std_logic;
      TXD: out std_logic  := '1'
   );
end TRNG;

architecture behaviour of TRNG is

   -- components
      -- HARD MACRO
   component ring
      port(
         Clk: in std_logic;
         Rset: in std_logic;
         enable:in std_logic;
         digit0_0: out std_logic;
         digit1_0: out std_logic;
         digit2_0: out std_logic
      );
   end component;
------------------------------------------------------------------------
--  Local Type Declarations
------------------------------------------------------------------------
```

```
   type tstate is (
      sttIdle,      --Idle state
      sttTransfer,     --Move data into shift register
      sttShift      --Shift out data
      );

-------------------------------------------------------------------------
-- Signal Declarations 01010010
-------------------------------------------------------------------------
constant baudDivide: std_logic_vector(7 downto 0) := "01010010"; --Baud
    Rate dividor, set now for a rate of 19200.
   signal tfReg: std_logic_vector(7 downto 0);      --Transfer holding
       register
signal tfSReg: std_logic_vector(10 downto 0)  := "11111111111";
    --Transfer shift register
signal clkDiv: std_logic_vector(8 downto 0) := "000000000"; --used for
    rClk
   signal rClkDiv: std_logic_vector(3 downto 0) := "0000";  --used for
       tClk
   signal ctr: std_logic_vector(3 downto 0) := "0000";   --used for
       delay times
   signal tfCtr: std_logic_vector(3 downto 0) := "0000";  --used to
       delay in transfer
   signal rClk : std_logic := '0';            --Receiving Clock
   signal tClk : std_logic;                 --Transfering Clock

   signal ctRst: std_logic := '0';
   signal load: std_logic := '0';
   signal shift: std_logic := '0';
   signal par: std_logic;
     signal tClkRST: std_logic := '0';
   signal rShift: std_logic := '0';
   signal dataRST: std_logic := '0';
   signal dataIncr: std_logic := '0';
   signal sttCur: tstate := sttIdle;          --Current state in the
       Transfer state machine
   signal sttNext: tstate;                    --Next state in the
       Transfer staet machine
   signal reg: std_logic_vector(7 downto 0);
   signal digit: std_logic_vector(2 downto 0);
   signal CLK_counter: std_logic;
   signal count: integer range 0 to 251 := 0;

begin

-- RING WITH COUNTER
ringhm: truerng port map (Clk => CLK_counter, Rset => RST, enable => EN,
    digit0_0 => digit(2), digit1_0 => digit(1), digit2_0 => digit(0));

--------------------------------
--------------------------------
-----------RS232---------------
-----------START----------------
--------------------------------
```

```
--------------------------------

par <= not ( ((tfReg(0) xor tfReg(1)) xor (tfReg(2) xor tfReg(3))) xor
    ((tfReg(4) xor tfReg(5)) xor (tfReg(6) xor tfReg(7))) );

--Clock Dividing Functions--
   --set up clock divide for rClk
   process (CLK, clkDiv)                          begin
         if (Clk = '1' and Clk'event) then
            if (clkDiv = baudDivide) then
               clkDiv <= "000000000";
            else
               clkDiv <= clkDiv +1;
            end if;
         end if;
      end process;

   --Define rClk
process (clkDiv, rClk, CLK)
   begin
      if CLK = '1' and CLK'Event then
         if clkDiv = baudDivide then
            rClk <= not rClk;
         else
            rClk <= rClk;
         end if;
      end if;
   end process;

   --set up clock divide for tClk
   process (rClk)
      begin
         if (rClk = '1' and rClk'event) then
            rClkDiv <= rClkDiv +1;
         end if;
      end process;

--define tClk
tClk <= rClkDiv(3);

   --set up a counter based on rClk
   process (rClk, ctRst)                          begin
         if rClk = '1' and rClk'Event then
            if ctRst = '1' then
               ctr <= "0000";
            else
               ctr <= ctr +1;
            end if;
         end if;
      end process;

   --set up a counter based on tClk
process (tClk, tClkRST)
      begin
```

```
        if (tClk = '1' and tClk'event) then
           if tClkRST = '1' then
              tfCtr <= "0000";
           else
              tfCtr <= tfCtr +1;
           end if;
        end if;
     end process;

--This process loads and shifts out the transfer shift
--register
   process (load, shift, tClk, tfSReg)
      begin
         TXD <= tfsReg(0);
         if tClk = '1' and tClk'Event then
            if load = '1' then
               tfSReg (10 downto 0) <= ('1' & par &                tfReg(7
                   downto 0) &'0');
            end if;
            if shift = '1' then
tfSReg (10 downto 0) <= ('1' & tfSReg(10 downto 1));
            end if;
         end if;
      end process;

   -- Transfer State Machine--
   process (tClk, RST)
      begin
         if (tClk = '1' and tClk'Event) then

            if RST = '1' then
               sttCur <= sttIdle;
            else
               sttCur <= sttNext;
            end if;
         end if;
      end process;

-- This process generates the sequence of steps needed
-- transfer the data
   process (sttCur, tfCtr, tfReg, tclk,WR)
      begin

         case sttCur is

            when sttIdle =>
               tClkRST <= '0';
               shift <= '0';
               load <= '0';
               if WR = '0' then
                  sttNext <= sttIdle;
               else
                  sttNext <= sttTransfer;
               end if;
```

```
            when sttTransfer =>
               shift <= '0';
               load <= '1';
               tClkRST <= '1';
               sttNext <= sttShift;


            when sttShift =>
               shift <= '1';
               load <= '0';
               tClkRST <= '0';
               if tfCtr = "1100" then
                  sttNext <= sttIdle;
               else
                  sttNext <= sttShift;
               end if;
         end case;
      end process;
--------------------------------
--------------------------------
------------RS232---------------
------------END-----------------
--------------------------------
--------------------------------

--------------------------------
--------------------------------
------------TRNG----------------
--------------------------------
--------------------------------

-- To control the sampling frequency of the ring
process (CLK, RST)
   begin
   if RST='1' then
      CLK_counter <= '0';
   elsif Clk'EVENT and Clk='1' then
      if count < 25 then
          -- The maximum value of count is modified for each frequency
         count <= count + 1;
      else
         count <= 0;
         CLK_counter <= not CLK_counter;
         reg <= digit & reg(7 downto 3);
      end if;
   end if;
end process;

tfReg<=reg;

end behaviour;
```

In order to make the different tests we have to place the Hard Macro in different places in the FPGA which is done through the ucf file:

```
#POSITION 01
#INST "ring" LOC=SLICE_X1Y91;

#POSITION 02
#INST "ring" LOC=SLICE_X65Y91;

#POSITION 03
#INST "ring" LOC=SLICE_X65Y5;

#POSITION 04
#INST "ring" LOC=SLICE_X1Y5;

#POSITION 05
#INST "ring" LOC=SLICE_X15Y87;

#POSITION 06
#INST "ring" LOC=SLICE_X107Y87;

#POSITION 07
#INST "ring" LOC=SLICE_X107Y19;

#POSITION 08
#INST "ring" LOC=SLICE_X15Y19;

#POSITION 09
#INST "ring" LOC=SLICE_X31Y45;


#PACE: End of Constraints generated by PACE
NET "CLK" TNM_NET = "CLK";
TIMESPEC TS_Clk = PERIOD "CLK" 20 ns HIGH 50 %;
#PACE: Start of Constraints generated by PACE
#PACE: Start of PACE I/O Pin Assignments
NET "CLK" LOC = C9;
NET "RST" PULLDOWN;
NET "RST" LOC = L13;

NET "enable" PULLDOWN;
NET "enable" LOC = N17;

NET "WR" PULLDOWN;
NET "WR" LOC = L14;
#NET "DBIN<0>" LOC = "N17" | PULLDOWN ;
#NET "DBIN<1>" LOC = "H18" | PULLDOWN ;
NET "TXD" IOSTANDARD = LVTTL;
NET "TXD" DRIVE = 8;
NET "TXD" SLEW = SLOW;
NET "TXD" LOC = M14;
```

Appendix B

Creating the Hard Macro

In order to place the ring oscillator throughout the FPGA with ease, it has been turned into a hard macro: a fixed circuit or component. To do so, we go to the FPGA Editor (View/Edit Routed Design) which can be found under the Place & Route options in the Processes menu.

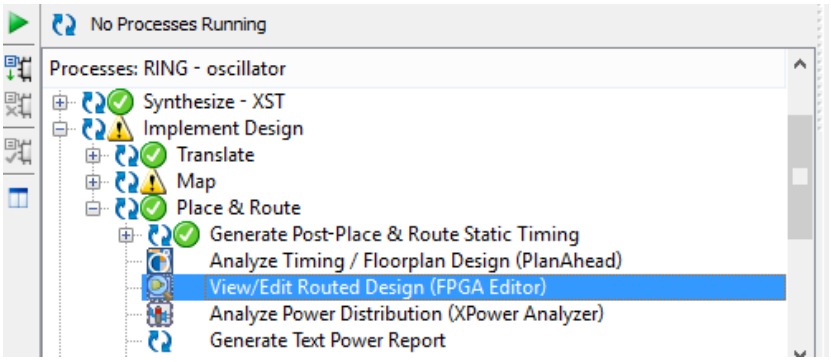

Figure B.1: FPGA Editor

In the FPGA Editor, the first thing to do is to save the design as a Hard Macro, simply go to **File** → **Save As** and in the pop-up choose the Hard Macro option and with the **Browse** option you can save it as a file with the extension **.nmc** in your project folder.

Before any changes can be done to the design its properties have to be edited from **File** → **Main Properties** where the Read Write mode can be enabled.

Now, it is necessary to unplace all components that are not of type SLICE, such as input and output signals in a way that all we have remaining are the inverters, the NAND gate and the D flip

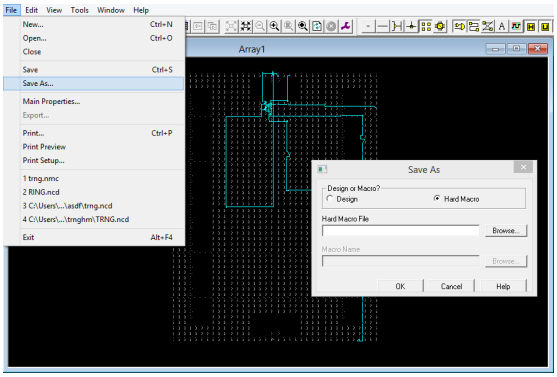

Figure B.2: FPGA Editor Save as Hard Macro

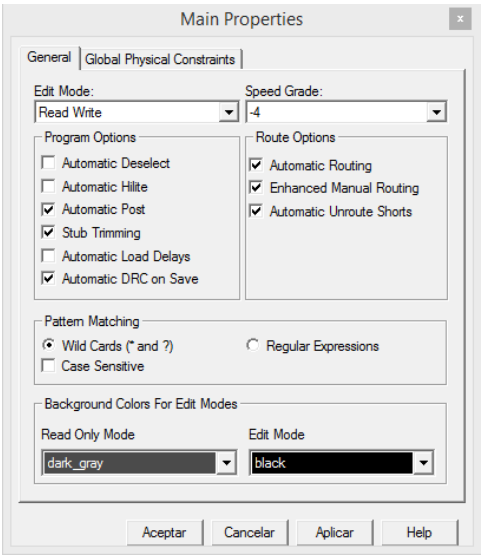

Figure B.3: FPGA Editor Read Write mode

flops. Note that the NAND and the inverters are contained in the **wire** components. To unplace components, select the component in the list by double clicking it and on the schematic right click it and select **Unplace Component**:

The components that have just been unplaced can be seen in the **Unplaced Components** list and can be removed from the design by selecting and deleting them.

To be able to use the Hard Macro, its external pins Need to be explicitly declared. Select the pin to which the input (enable, reset and clock) and output (digit0_0, digit1_0, digit2_0) signals belong and go to **Edit** → **Add Hard Macro External Pin**, where it is very important that the pins is named exactly like the name of the

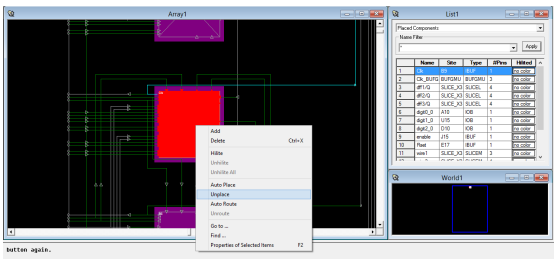

Figure B.4: FPGA Editor Unplace component

i/o defined in the vhdl file.

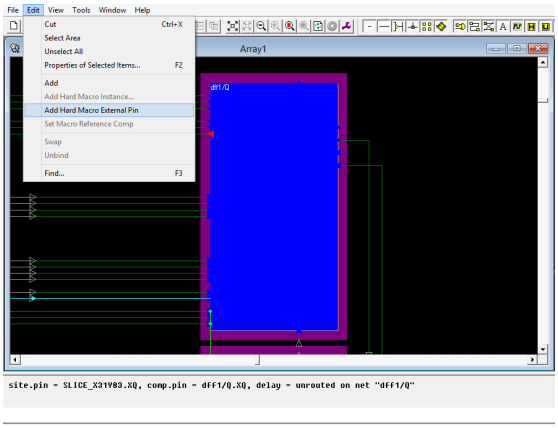

Figure B.5: FPGA Editor Add Hard Macro External Pin

The last step is to set a component of the Hard Macro as its reference component, which later on is the component that is moved around the FPGA. Select the desired component and go to **Edit** → **Set Macro Reference Component**.

In the project that will instantiate the hard macro simply add as a resource this nmc file we have just edited and instantiate it as any other component and has to be placed in the ucf file manually.

Appendix C

Python Scripts

Hex To Bin

The output of the TRNG captured with Realterm is in hexadecimal, in order to apply the statistical tests to check how robust our generator is we first need to convert it to binary.

This script requests an input file to be specified with the command **-i** as well as an output file with the command **-o** to save the converted binary string.

```
import os
from optparse import OptionParser

def getOptions():
   parser = OptionParser()
   parser.add_option("-i", "--input", help="Hexadecimal file",
       metavar="FILE")
   parser.add_option("-o", "--output", help="Output Binary File")
   return parser.parse_args()

def hexToBin(hexchar):
   binstr = bin(int(hexchar, 16))[2:]
   while((len(binstr)) < 4):
      binstr = '0' + binstr
   return binstr

if __name__ == '__main__':
   (options, args) = getOptions()
   print 'Retrieving input...'
   fin = open(options.input, 'r')
   print 'Hexadecimal file retrieved!'
   print 'Transforming to binary...'
   fout = open(options.output, 'w')
   while True:
      charhex = fin.read(1)
```

```
      if not charhex: break
      charsbin = hexToBin(charhex)
      fout.write(charsbin)
   print 'Transformation finished'
   fin.close()
   fout.close()
```

Post-Processor

Post-processing the output data of the TRNG makes the number generated all the more secure to attacks as explained in 3.2.2.

The script requires the output binary file of the previous **Hex to Bin** script as input and performs an XOR operation of as many bits as we wish through the argument **-b**. Finally, it returns the new binary file with the name we provide in argument **-o**.

```
import os
from optparse import OptionParser

def getOptions():
   parser = OptionParser()
   parser.add_option("-i", "--input", help="Binary file", metavar="FILE")
   parser.add_option("-o", "--output", help="Output Postprocessed")
   parser.add_option("-b", "--buffer", help="Bits to XOR")
   return parser.parse_args()

if __name__ == '__main__':
   (options, args) = getOptions()
   print 'Retrieving input...'
   fin = open(options.input, 'r')
   print 'Binary file retrieved!'
   buf = options.buffer
   print 'Bits to process: ' + buf
   buff = int(buf)
   print 'Creating output...'
   fout = open(options.output, 'w')
   xor = fin.read(1)
   chunk = fin.read(buff - 1)
   while chunk:
      for char in chunk:
         xor = int(xor) ^ int(char)
      if xor == True:
         fout.write('1')
      else:
         fout.write('0')
      xor = fin.read(1)
      chunk = fin.read(buff - 1)
   print 'Process finished!! Output file created'
   fin.close()
```

```
   fout.close()
```

Editing STS Result File

For every measure of the TRNG output, the NIST Statistical Test Suite results are returned in a simple text file. As an example of how the content of said files are presented, here are the results of the tests for the central position at 1MHz and a post-processing of 5 bits:

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <p09freq1Mpp5>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 33  11  12  11   3   5   7   3   7   8  0.000000 *   88/100  *  Frequency
 12  14  13  10  17   7   8   7   8   4  0.122325     96/100     BlockFrequency
 30  13  10   4  11   6   9   6   4   7  0.000000 *   92/100  *  CumulativeSums
 29  18   9   8   6   5   8   5   8   4  0.000000 *   89/100  *  CumulativeSums
 77   7   6   5   3   0   1   0   1   0  0.000000 *   60/100  *  Runs
 13  13  11  12   9  11  10   5  12   4  0.437274     99/100     LongestRun
 13  15  12   7  13   9   9   8  11   3  0.262249    100/100     Rank
  6   9  16   7  12   9  12   9   6  14  0.319084     99/100     FFT
  9  19   7   6   7  12   5   8  14  13  0.042808     99/100     NonOverlappingTemplate
  9  10  10   8  14  10   9  12   9   9  0.971699     98/100     NonOverlappingTemplate
  9   8  14  10  12   4  13  11  11   8  0.574903     99/100     NonOverlappingTemplate
 16   9  12  10   5   8   9  12  13   6  0.350485     97/100     NonOverlappingTemplate
 13   7   8   8  15   6  18  10  11   4  0.051942    100/100     NonOverlappingTemplate
 10  15   4  12  10  10  12   9   9   9  0.616305     98/100     NonOverlappingTemplate
  6  11   9  11  12   6  15  13   6  11  0.437274     99/100     NonOverlappingTemplate
  9  16   9   7  13  11   8  10   7  10  0.637119     99/100     NonOverlappingTemplate
  7   9  13  10  11  10  15   9   9   7  0.779188     99/100     NonOverlappingTemplate
 10  10  17  10  15  11   8   6   5   8  0.191687     99/100     NonOverlappingTemplate
  7  10  10   9  11   9  12  14   9   9  0.946308    100/100     NonOverlappingTemplate
 12   6  13  19   8  13   7  11   7   4  0.037566     98/100     NonOverlappingTemplate
 11  12   3  11  12   4   8  12  12  15  0.153763     99/100     NonOverlappingTemplate
  5  11  10   7  19  13  11   8   7   9  0.122325     98/100     NonOverlappingTemplate
  8  15   5   6  10  11  10   9  12  14  0.419021    100/100     NonOverlappingTemplate
 14   8   9   9  10  10   6  11  11  12  0.883171     99/100     NonOverlappingTemplate
  7  16  12   9   8  11  13  10   3  11  0.249284    100/100     NonOverlappingTemplate
  9  13   8  16   7  11   8   7  11  10  0.595549     99/100     NonOverlappingTemplate
 16  13   9  10   9  11   8  12   7   5  0.437274     97/100     NonOverlappingTemplate
 11   5   9   9  13   8  13   9  13  10  0.739918    100/100     NonOverlappingTemplate
 10  13  12   6  10  13  13   9   6   8  0.657933     98/100     NonOverlappingTemplate
  7   2  17  10  15   9   7  17   8   8  0.010988    100/100     NonOverlappingTemplate
 13   6  16  12   8  10   9   6  10  10  0.474986    100/100     NonOverlappingTemplate
 12   8   6   6  15   8  10  13  11  11  0.534146     99/100     NonOverlappingTemplate
  9   8  13  14  11   9   6  10   7  13  0.678686    100/100     NonOverlappingTemplate
 10  10  12  11   9  11   8  14   9   6  0.883171    100/100     NonOverlappingTemplate
 11  10  11   9   9   9  10   7   9  15  0.911413     99/100     NonOverlappingTemplate
  8  17  11  10   8   5  11  10   8  12  0.419021     99/100     NonOverlappingTemplate
 15   6  10  11   9  11   8  12   9   9  0.798139     99/100     NonOverlappingTemplate
 15   8  14  13  11   5  12   7  10   5  0.224821     98/100     NonOverlappingTemplate
 12  12   5   8  11  12  11  13   6  10  0.657933     99/100     NonOverlappingTemplate
```

9	9	8	10	12	11	9	12	7	13	0.946308	100/100		NonOverlappingTemplate
8	8	13	7	11	12	10	14	10	7	0.779188	97/100		NonOverlappingTemplate
6	7	14	8	11	8	10	14	15	7	0.350485	99/100		NonOverlappingTemplate
7	8	9	10	13	6	14	10	12	11	0.739918	99/100		NonOverlappingTemplate
18	6	12	9	11	11	9	9	10	5	0.249284	98/100		NonOverlappingTemplate
10	11	7	11	13	8	13	6	12	9	0.798139	99/100		NonOverlappingTemplate
12	8	14	11	9	12	11	9	9	5	0.759756	99/100		NonOverlappingTemplate
10	10	11	15	12	6	7	7	13	9	0.595549	98/100		NonOverlappingTemplate
8	11	12	11	6	13	9	15	5	10	0.474986	100/100		NonOverlappingTemplate
7	13	10	9	14	10	8	8	9	12	0.851383	100/100		NonOverlappingTemplate
17	13	9	12	7	8	6	14	6	8	0.171867	99/100		NonOverlappingTemplate
11	10	7	11	10	9	13	7	10	12	0.946308	98/100		NonOverlappingTemplate
16	12	7	8	5	7	12	15	10	8	0.213309	98/100		NonOverlappingTemplate
11	8	6	11	9	13	9	11	8	14	0.798139	99/100		NonOverlappingTemplate
7	3	8	11	9	10	14	18	12	8	0.085587	100/100		NonOverlappingTemplate
20	12	13	5	15	10	6	4	8	7	0.006661	97/100		NonOverlappingTemplate
6	8	5	8	15	12	12	12	14	8	0.304126	100/100		NonOverlappingTemplate
10	12	13	7	11	13	13	9	6	6	0.595549	99/100		NonOverlappingTemplate
16	9	13	11	6	6	15	10	7	7	0.202268	99/100		NonOverlappingTemplate
10	12	11	9	8	14	13	4	10	9	0.616305	99/100		NonOverlappingTemplate
15	10	12	14	7	10	7	8	6	11	0.494392	96/100		NonOverlappingTemplate
9	9	16	8	7	13	14	9	8	7	0.437274	97/100		NonOverlappingTemplate
10	6	11	6	8	9	12	14	11	13	0.657933	98/100		NonOverlappingTemplate
5	10	14	7	10	10	11	10	10	13	0.739918	99/100		NonOverlappingTemplate
10	10	11	11	10	9	10	7	13	9	0.987896	99/100		NonOverlappingTemplate
9	12	9	10	13	11	6	14	7	9	0.759756	99/100		NonOverlappingTemplate
10	13	11	6	10	11	8	13	7	11	0.834308	98/100		NonOverlappingTemplate
9	9	10	8	10	4	12	16	10	12	0.474986	100/100		NonOverlappingTemplate
15	9	8	14	12	6	5	13	10	8	0.319084	97/100		NonOverlappingTemplate
7	6	9	11	10	11	14	13	9	10	0.798139	99/100		NonOverlappingTemplate
6	8	11	12	9	10	9	9	18	8	0.383827	100/100		NonOverlappingTemplate
10	15	13	8	7	11	11	11	7	7	0.657933	99/100		NonOverlappingTemplate
15	7	11	15	9	9	4	13	6	11	0.191687	98/100		NonOverlappingTemplate
10	8	8	7	9	9	13	10	13	13	0.867692	99/100		NonOverlappingTemplate
8	13	14	9	13	5	6	9	13	10	0.437274	100/100		NonOverlappingTemplate
16	4	5	6	14	11	15	9	13	7	0.042808	98/100		NonOverlappingTemplate
17	12	16	6	7	14	7	10	7	4	0.030806	93/100	*	NonOverlappingTemplate
16	11	15	7	5	8	13	13	5	7	0.085587	97/100		NonOverlappingTemplate
11	15	10	10	12	5	12	14	8	3	0.171867	99/100		NonOverlappingTemplate
14	12	9	12	8	6	14	12	10	3	0.249284	97/100		NonOverlappingTemplate
15	10	13	17	7	7	12	6	9	4	0.071177	99/100		NonOverlappingTemplate
11	13	6	11	13	9	9	14	7	7	0.616305	100/100		NonOverlappingTemplate
17	10	11	7	11	12	10	11	5	6	0.304126	96/100		NonOverlappingTemplate
9	19	7	6	7	12	5	8	14	13	0.042808	99/100		NonOverlappingTemplate
10	13	9	10	11	7	11	7	8	14	0.834308	98/100		NonOverlappingTemplate
13	8	8	9	11	11	9	8	10	13	0.946308	97/100		NonOverlappingTemplate
12	10	7	9	13	9	11	8	12	9	0.946308	99/100		NonOverlappingTemplate
9	4	4	8	13	8	16	14	11	13	0.085587	99/100		NonOverlappingTemplate
11	8	10	11	6	14	12	13	5	10	0.574903	100/100		NonOverlappingTemplate
11	6	12	6	9	11	10	9	14	12	0.739918	100/100		NonOverlappingTemplate
8	11	9	15	13	10	9	10	10	5	0.678686	98/100		NonOverlappingTemplate
9	11	10	13	10	5	7	12	11	12	0.798139	98/100		NonOverlappingTemplate
4	11	17	5	9	11	9	14	8	12	0.129620	100/100		NonOverlappingTemplate
8	8	7	13	13	11	12	9	10	9	0.897763	99/100		NonOverlappingTemplate
5	7	14	9	14	7	11	18	8	7	0.080519	100/100		NonOverlappingTemplate
5	13	9	15	8	10	4	14	14	8	0.137282	100/100		NonOverlappingTemplate
11	9	10	14	8	14	8	7	10	9	0.816537	99/100		NonOverlappingTemplate
8	8	14	11	10	9	12	4	21	3	0.004981	99/100		NonOverlappingTemplate
11	5	9	13	15	11	7	11	9	9	0.595549	98/100		NonOverlappingTemplate
6	10	7	10	11	14	10	13	8	11	0.779188	98/100		NonOverlappingTemplate
7	16	8	9	12	13	8	8	7	12	0.494392	99/100		NonOverlappingTemplate
12	13	10	7	6	8	10	8	15	11	0.616305	99/100		NonOverlappingTemplate

10	11	9	13	14	9	5	9	10	10	0.798139	100/100	NonOverlappingTemplate
16	12	10	11	5	4	8	12	12	10	0.249284	99/100	NonOverlappingTemplate
19	11	4	8	6	9	13	8	15	7	0.028817	100/100	NonOverlappingTemplate
18	5	6	9	13	9	13	10	9	8	0.162606	97/100	NonOverlappingTemplate
6	6	10	10	10	12	7	16	12	11	0.474986	99/100	NonOverlappingTemplate
9	11	11	10	12	10	5	9	12	11	0.924076	99/100	NonOverlappingTemplate
11	9	10	6	15	15	4	8	14	8	0.171867	99/100	NonOverlappingTemplate
9	5	10	12	13	12	7	9	10	13	0.719747	99/100	NonOverlappingTemplate
13	8	11	8	9	11	14	4	10	12	0.574903	99/100	NonOverlappingTemplate
12	11	11	9	5	10	10	14	7	11	0.759756	99/100	NonOverlappingTemplate
10	12	9	6	5	13	10	12	16	7	0.319084	99/100	NonOverlappingTemplate
7	12	11	9	9	11	11	7	8	15	0.779188	100/100	NonOverlappingTemplate
6	7	9	15	7	10	15	12	12	7	0.334538	99/100	NonOverlappingTemplate
11	10	14	3	9	11	7	11	14	10	0.401199	99/100	NonOverlappingTemplate
7	14	9	15	7	11	7	7	12	11	0.494392	99/100	NonOverlappingTemplate
9	10	17	8	9	9	9	10	5	14	0.366918	100/100	NonOverlappingTemplate
7	13	6	15	10	6	13	7	7	16	0.129620	100/100	NonOverlappingTemplate
6	6	9	14	13	12	9	10	12	9	0.657933	100/100	NonOverlappingTemplate
5	8	10	13	10	12	6	9	18	9	0.191687	99/100	NonOverlappingTemplate
7	10	9	13	10	4	11	12	14	10	0.574903	97/100	NonOverlappingTemplate
3	9	15	12	5	16	13	8	8	11	0.071177	99/100	NonOverlappingTemplate
6	6	9	10	11	9	8	19	11	11	0.202268	100/100	NonOverlappingTemplate
10	12	14	5	12	8	5	11	11	12	0.494392	99/100	NonOverlappingTemplate
11	7	14	7	6	11	11	11	14	8	0.595549	100/100	NonOverlappingTemplate
9	7	20	8	8	10	12	9	5	12	0.085587	97/100	NonOverlappingTemplate
13	9	6	10	6	10	10	9	8	19	0.171867	98/100	NonOverlappingTemplate
13	10	12	12	7	7	11	10	9	9	0.924076	100/100	NonOverlappingTemplate
16	7	10	9	8	10	10	13	10	7	0.657933	93/100 *	NonOverlappingTemplate
16	13	6	12	6	13	11	6	8	9	0.262249	98/100	NonOverlappingTemplate
10	12	7	7	8	11	10	12	11	12	0.935716	99/100	NonOverlappingTemplate
7	8	11	12	12	10	10	11	10	9	0.983453	100/100	NonOverlappingTemplate
5	14	14	13	9	11	8	8	9	9	0.554420	100/100	NonOverlappingTemplate
17	7	7	19	8	8	9	7	8	10	0.048716	97/100	NonOverlappingTemplate
10	10	15	10	9	8	13	9	11	5	0.678686	98/100	NonOverlappingTemplate
7	14	15	6	11	8	5	14	8	12	0.213309	100/100	NonOverlappingTemplate
6	8	8	9	12	11	13	12	12	9	0.851383	99/100	NonOverlappingTemplate
4	14	11	6	12	12	9	7	10	15	0.262249	98/100	NonOverlappingTemplate
12	7	10	10	11	10	11	12	11	6	0.935716	100/100	NonOverlappingTemplate
17	12	11	5	7	11	9	12	9	7	0.319084	97/100	NonOverlappingTemplate
10	9	13	11	7	8	9	4	12	17	0.249284	99/100	NonOverlappingTemplate
16	10	9	5	16	12	12	9	7	4	0.085587	100/100	NonOverlappingTemplate
11	3	9	12	7	15	15	11	7	10	0.191687	100/100	NonOverlappingTemplate
11	7	13	8	7	16	9	11	12	6	0.437274	99/100	NonOverlappingTemplate
9	9	4	20	11	10	13	8	6	10	0.051942	100/100	NonOverlappingTemplate
12	7	7	12	13	6	8	13	11	11	0.678686	99/100	NonOverlappingTemplate
5	7	9	17	5	8	9	13	15	12	0.085587	98/100	NonOverlappingTemplate
8	7	14	12	16	11	11	8	5	8	0.319084	98/100	NonOverlappingTemplate
15	10	10	11	9	11	8	7	10	9	0.897763	99/100	NonOverlappingTemplate
15	13	14	9	8	14	4	12	5	6	0.085587	97/100	NonOverlappingTemplate
8	9	12	10	12	12	11	10	7	9	0.971699	100/100	NonOverlappingTemplate
14	12	8	7	16	10	3	4	12	14	0.042808	97/100	NonOverlappingTemplate
19	13	6	12	9	7	8	6	13	7	0.071177	98/100	NonOverlappingTemplate
11	10	9	18	4	8	8	10	10	12	0.249284	99/100	NonOverlappingTemplate
14	18	11	14	5	11	8	8	8	3	0.030806	97/100	NonOverlappingTemplate
17	10	11	7	11	12	10	10	6	6	0.383827	96/100	NonOverlappingTemplate
12	10	14	7	8	7	9	10	16	7	0.455937	95/100 *	OverlappingTemplate
0	0	0	0	0	0	0	0	0	100	0.000000 *	100/100	Universal
23	11	10	16	11	7	5	8	4	5	0.000347	98/100	ApproximateEntropy
0	0	0	0	0	0	0	0	0	0	----	------	RandomExcursions
0	0	0	0	0	0	0	0	0	0	----	------	RandomExcursions
0	0	0	0	0	0	0	0	0	0	----	------	RandomExcursions
0	0	0	0	0	0	0	0	0	0	----	------	RandomExcursions

```
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursions
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursions
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursions
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursions
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   0   0   0   0   0   0   0   0   0   0     ----      ------     RandomExcursionsVariant
   9   9   8  10  14   7  10  13   6  14  0.616305    100/100      Serial
  12  11   6   9  10   7  16   9  11   9  0.637119    100/100      Serial
  13   8  12   9  11   6  17   4  13   7  0.129620     97/100      LinearComplexity


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 96 for a
sample size = 100 binary sequences.

The minimum pass rate for the random excursion (variant) test is undefined.

For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

An asterisk next to the name of a test or proportion means the TRNG did not pass said test.

However, in order to compare the results of each measure in a box-plot simply having these text files is not enough. Since we would have to manually check each file, to make this process automatic, this python script removes all unnecessary information from the files and creates a CSV file which organizes the data in columns. This is done by removing spaces, asterisks, the opening and closing text and the *RandomExcursions* and *RandomExcursionsVariant* tests which are not helpful in testing our TRNG. Moreover, as was mentioned in section 2.3.1, the Non-Overlapping Template Matching Test produces as many p-values as blocks in which the tested sequenced was divided by its algorithm, the script computes the average proportion and p-value of these and writes it into the output CSV file.

```
import os
```

```
import fileinput

def getOptions():
   parser = OptionParser()
   parser.add_option("-o", "--output", help="Output File for
       Representation")
   return parser.parse_args()

def getColumns(inFile, delim=";", header=True):
    cols = {}
    indexToName = {}
    for lineNum, line in enumerate(inFile):
        if lineNum == 0:
            headings = line.split(delim)
            i = 0
            for heading in headings:
                heading = heading.strip()
                if header:
                    cols[heading] = []
                    indexToName[i] = heading
                else:
                    # in this case the heading is actually just a cell
                    cols[i] = [heading]
                    indexToName[i] = i
                i += 1
        else:
            cells = line.split(delim)
            i = 0
            for cell in cells:
                cell = cell.strip()
                cols[indexToName[i]] += [cell]
                i += 1

    return cols, indexToName

def find_element_in_list(element,list_element):
    try:
        index_element=list_element.index(element)
        return index_element
    except ValueError:
        return -1

if __name__ == '__main__':
   (options, args) = getOptions()
   stats = file("finalAnalysisReport.txt", 'r')
   fin = stats.readlines()
   stats.close()
   fout = file("output.txt", "w")
   words = ['*', 'TEST', '/100']
   commas = [';;;;;', ';;;;', ';;;', ';;']
   # FIRST: Edit the input file to be able to work with it
   # 14 columns separated by commas
   # We are interested in the last 3 columns
   for line in fin:
```

```
    if any(line.startswith(word) for word in ('-', 'RESULTS', '
        generator', 'The', 'random', 'sample', 'is', 'For',
        'provided')):
      continue
    if 'RandomExcursions' in line:
      continue
    lineedited = line.replace(' ', ';')
    for word in words:
      lineedited = lineedited.replace(word, '')
    if line.startswith('1'):
      lineedited = lineedited.replace('100', ';10')
    for comma in commas:
      lineedited = lineedited.replace(comma, ';')
    if lineedited.startswith(';C1;'):
      lineedited = lineedited.replace(';C1;', 'EMP;C1;')
    if ";\n" in lineedited:
      lineedited = lineedited.replace(";\n", '\n')
    fout.write(lineedited)
#SECOND: AVERAGE OF NonOverlappingTemplate
fout.close()
fout = file("output.txt.tmp", 'r')
cols, indexToName = getColumns(fout)
NonOver = ''
sumpv = 0
sump = 0
avpv = 0
avp = 0
indices = list()
offset = 0
for cell in
    range(cols['STATISTICAL'].count('NonOverlappingTemplate')):
  indices.append(cols['STATISTICAL'].index('NonOverlappingTemplate',offset))
  offset = indices[-1] + 1
for index in indices:
  sumpv = float(cols['P-VALUE'][index]) + sumpv
  sump = int(cols['PROPORTION'][index]) + sump
avpv = float("{0:.6f}".format(sumpv/len(indices)))
avp = int(sump/len(indices))
print avpv
print avp
fout.close()
fout = file("output.txt.tmp", 'r')
foutlines = fout.readlines()
fout.close()
fout2 = file(options.output, 'w')
for line in foutlines:
  if 'NonOverlappingTemplate' in line:
    continue
  if ';OverlappingTemplate' in line:
    fout2.write(line)
    fout2.write(";0;0;0;0;0;0;0;0;0;0;" + str(avpv) + ";" +
        str(avp) + ";NonOverlappingTemplate\n")
  else:
    fout2.write(line)
```

```
fout2.close()
```

Appendix D

Glossary

ASIC: Application Specific Integrated Circuits

BRAM: Block Random Access Memory

BSI: Bundesamt für Sicherheit in der Informationstechnik

CCN: Centro Criptológico Nacional

CLB: Configurable Logic Block

COFRAC: Comité français d'accréditation

CSV: Comma-separated Values

DCM: Digital Clock Manager

DCM: Digital Clock Source

DFF: D flip-flop

DLL: Delay-locked Loop

DSP: Digital Signal Processor

DSP: Digital Signal Processor

FFT: Fast Fourier Transform

FIFO: First in, First out

FIPS: Federal Information Processing Standards

FPGA: Field Programmable Gate Array

HDL: Hardware Description Language

IC: Integrated Circuit

IQR: Interquartile Range

LFSR: Linear Feedback Shift Register

NIST: National Institute of Science and Technology

PLL: Phase-locked Loop

PRNG: Pseudo-random Number Generator

RAM: Random Access Memory

RFID: Radio Frequency Identification

RNG: Random Number Generator

RO: Ring Oscillator

SRAM: Static Random Access Memory

STR: Self-Timed Ring

STS: Statistical Test Suite

TRNG: True Random Number Generator

UKAS: United Kingdom Accreditation Service

VHDL: VHSIC Hardware Description language

VHSIC: Very High Speed Integrated Circuit

XOR: Exclusive OR

Bibliography

[1] E. Böhl and M. Ihle, "A Fault Attack Robust TRNG," *18th IEEE International On-Line Testing Symposium*, 2012.

[2] E. Böhl, M. Lewis, and S. Galkin, "A True Random Number Generator with On-Line Testability," *19th IEEE European Test Symposium (ETS)*, 2014.

[3] (2014) Statistical Test Suite. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

[4] The History of Security. [Online]. Available: http://perspecsys.com/history/

[5] D. Mukhopadhyay and R. S. Chakraborty, *Hardware Security: Design, Threats, and Safeguards*. CRC Press, 2014.

[6] M. Tehranipoor and C. Wang, *Introduction to Hardware Security and Trust*. Springer, 2011.

[7] M. Haahr. Introduction to randomness and random numbers. [Online]. Available: https://www.random.org/randomness/

[8] Random number generation. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/random_number.html

[9] M. G. Segura, "Implementación en VHDL de generadores de números pseudo-aleatorios (PRNGS) criptográficamente seguros," Master's thesis, Universidad Carlos III de Madrid, 2009.

[10] H. M. González, "Hardware design of cryptographic algorithms for low-cost rfid tags," Ph.D. dissertation, Universidad Carlos III de Madrid, 2015.

[11] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1998.

[12] R. P. Brent, "Some long-period random number generators using shifts and xors," *ANZIAM*, 2007.

[13] M. Stipčević and Çetin Kaya Koç, "True Random Number Generators."

[14] K. Brar and S. P. Karanam, "True Random Number Generators."

[15] V. Fischer, A. Aubert, F. Bernard, B. Valtchanov, J. Danger, and N. Bochard, "True Random Number Generators in Configurable Logic Devices," 2009.

[16] R. B. Davies, "Exclusive OR (XOR) and hardware random number generators," 2002.

[17] J. V. Neumann, "Various Techniques Used in Connection With Random Digits," *Journal of Research of the National Bureau of Standards*, 1951.

[18] S. Halevi, "Cryptographic Hash Functions and their many applications," 2009.

[19] B. Jun and P. Kocher, "The Intel Random Number Generator," 1999.

[20] C. Fontaine, "Contribution à la recherche de fonctions booléennes hautement non linéaires, et au marquage d'images en vue de la protection des droits d'auteur," Ph.D. dissertation, Université Paris, 1998.

[21] S.-H. Kwok, Y.-L. Ee, G. Chew, K. Zheng, K. Khoo, and C.-H. Tan, "A Comparison of Post-Processing Techniques for Biased Random Number Generators," *Lecture Notes in Computer Science*, 2011.

[22] C. Colbourn, J. Dinitz, and D. Stinson, "Applications of combinatorial designs to communications, cryptography, and networking," 1999.

[23] K. Gopalakrishnan and D. Stinson, "Applications of designs to cryptography," 1999.

[24] G. Marsaglia. (1996) The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. [Online]. Available: http://www.stat.fsu.edu/pub/diehard/

[25] G. Marsaglia and W. W. Tsang, "Some difficult-to-pass tests of randomness," *Journal of Statistical Software*, 2002.

[26] J. Walker. (2008) ENT: A Pseudorandom Number Sequence Test Program. [Online]. Available: http://www.fourmilab.ch/random/

[27] Federal Information Processing Standards Publications (FIPS PUBS).

[28] W. Schindler and W. Killmann, "Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications," *Cryptographic Hardware and Embedded Systems (CHES)*, 2003.

[29] W. Schindler, "Efficient Online Tests for True Random Number Generators," *Cryptographic Hardware and Embedded Systems (CHES)*, 2001.

[30] Guide to the Statistical Tests. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/stats_tests.html

[31] "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications," National Institute of Standards and Technology, Tech. Rep., 2010.

[32] Berlekamp-massey algorithm. [Online]. Available: http://www.encyclopediaofmath.org/index.php/Berlekamp%E2%80%93Massey_algorithm

[33] What is a FPGA? Xillinx. [Online]. Available: http://www.xilinx.com/fpga/index.htm

[34] Field Programmable Gate Array (FPGA). Xillinx. [Online]. Available: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm

[35] "Spartan-3E FPGA Family Data Sheet," Xillinx, Tech. Rep. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

[36] "Spartan-3E Starter Kit Board User Guide," Xilinx, Tech. Rep., January 2011. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf

[37] I. Vasyltsov, E. Hambardzumyan, Y.-S. Kim, and B. Karpinskyy, "Fast Digital TRNG based on Metastable Ring Oscillator," *Cryptographic Hardware and Embedded Systems (CHES)*, 2008.

[38] D. E. H. nd Wayne P. Burleson and K. Fu, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers," *Computers, IEEE Transactions*, 2009.

[39] T. Güneysu and C. Paar, "Transforming Write Collisions in Block RAMs into Security Applications," *Field Programmable Technology*, 2009.

[40] V. Fischer and M. Drutarovsky, "True random number generator embedded in reconfigurable hardware," *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems*.

[41] P. Kohlbrenner and K. Gaj, "An embedded true random number generator for FPGAs," *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*.

[42] B. Sunar, W. J. Martin, and D. R. Stinson, "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks," *IEEE Transactions on Computers*.

[43] K. Wold and C. H. Tan, "Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings," *International Journal of Reconfigurable Computing*.

[44] A. Cherkaoui, V. Fischer, L. Fesquet, and A. Aubert, "A very high speed true random number generator with entropy assessment, journal = Cryptographic Hardware and Embedded Systems (CHES) year = 2013."

[45] M. K. Mandal and B. C. Sarkar, "Ring Oscillators: Characteristics and Applications," 2010.

[46] A. J. Winstanley, "Temporal Properties of Self-Timed Rings," Master's thesis, University of British Columbia, 1999.

[47] A. Cherkaoui, V. Fischer, L. Fesquet, and A. Aubert, "Comparison of Self-Timed Ring and Inverter Ring Oscillators as Entropy Sources in FPGAs," 2012.

[48] D. Massart, J. Smeyers-Verbeke, X. Caprona, and K. Schlesier, "Visual Presentation of Data by Means of Box Plots," *Practical Data Handling*, 2005.

[49] M. Dichtl and J. D. Golić, "High-Speed True Random Number Generation with Logic Gates Only," *Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[50] W. Schindler and W. Killmann, "A proposal for: Functionality classes for random number generators," BSI, Tech. Rep., 2011.

[51] Quantis AIS 31 Validated RNG. [Online]. Available: http://www.idquantique.com/random-number-generation/quantis-ais-31/

[52] E. Barker and J. Kelsey, "Recommendation for Random Number Generation using Deterministic Random Bit Generators," Tech. Rep.

[53] B. G. Munilla, "Implementación Hardware de un test estadístico para aplicaciones criptográficas con un circuito automático de seguridad," Master's thesis, Universidad Carlos III de Madrid, 2011.

[54] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin, *Using hard macros to reduce FPGA compilation time*, BYU Department of Electrical & Computer Engineering. [Online]. Available: http://rapidsmith.sourceforge.net/papers/LavinFPL10-Poster.pdf

[55] Creating hard macros. [Online]. Available: http://wiki.eng.iastate.edu/reconfigurable-computing-wiki/index.php/Creating_Hard_Macros

[56] Interfacing rs-232 with spartan-3e fpga. [Online]. Available: https://www.pantechsolutions.net/media/k2/attachments/Interfacing_RS-232_with_Spartan-3E_FPGA.pdf

[57] B. Cuzeau. (2001) VHDL - Practical Example - Designing an UART. ALSE. [Online]. Available: http://www.inventus.org/posterous/file/2009/03/192538-alse_uart_us.pdf

[58] P. P.Chu, *FPGA prototyping by VHDL examples. Xilinx Spartan-3 version.* John Wiley & Sons, Inc., 2008.

[59] "Spartan-3E Starter Kit Schematics," Xilinx, Tech. Rep., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/s3e_starter_gerbers.pdf

[60] L. Buttyan and J.-P. Hubaux, *Security and Cooperation in Wireless Networks: Thwarting Malicious and Selfish Behavior in the Age of Ubiquitous Computing.* Cambridge University Press, 2007.

[61] S. Marsland, *Machine Learning: An Algorithmic Perspective.* CRC Press, 2009.

[62] P. S. Sundaram, "Development of a FPGA-based True Random Number Generator for Space Applications," Master's thesis, Linköping Institute of Technology, 2010.

[63] W. Schindler, "Random Number Generators for Cryptographic Applications (Part 1)." Federal Office for Information Security (BSI), 2008.

[64] V. Fischer, "Random Number Generators for Cryptography, Design and Evaluation." Laboratoire Hubert Curien, 2014.

[65] B. Bradignans, J. L. Danger, V. Fischer, G. Gogniat, and L. Torres, *Security Trends for FPGAS: From Secured to Secure Reconfigurable Systems*. Springer, 2011.

[66] B. J. Abcunas and D. C. Reisberg, "Evaluation of random number generators on fpgas," 2004.

[67] M. Epstein, L. Hars, R. Krasinski, M. Rosner, and H. Zheng, "Design and Implementation of a True Random Number Generator Based on Digital Circuit Artifacts," 2003.

[68] M. Majzoobi, F. Koushanfar, and M. Potkonjak, "Testing Techniques for Hardware Security," *IEEE International Test Conference*, 2008.