FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	True random number generator on FPGA
Student:	Kryštof Šádek
Supervisor:	Ing. Jiří Buček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2020/21

Instructions

Explore the principles of random number generation (TRNG) on FPGAs. Implement a TRNG based on ring oscillators [1] on an FPGA development board. The TRNG circuit will be an internal peripheral to a soft-core CPU such as Xilinx Microblaze. The FPGA board will communicate with a PC using a serial interface (USB-UART).
Create a program for the CPU that will send a random string generated by the TRNG to the PC.
Implement a simple test of the TRNG (frequency test and runs test) that will run interleaved with TRNG generation and whose results will be sent to the PC.
Test the generator output under various physical environmental conditions (voltage, temperature).

References

[1] Buchovecká, S., Lórencz, R., Kodýtek, F. and Buček, J., 2017. True random number generator based on ring oscillator PUF circuit. Microprocessors and Microsystems, 53, pp.33-41.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 10, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

True Random Number Generator on FPGA

Kryštof Šádek

Department of Information Security
Supervisor: Ing. Jiří Buček, Ph.D.

June 3, 2020

Acknowledgements

I would like to thank Jiří Buček for his help, Filip Kodýtek for providing source codes of ROPUF and my faculty allowing me to carry out this project. Also, I want to thank my family, which made it possible for me to study at CTU, my wife for her support during those stressful times, and last but not least, God who gave us intellect and all the means to pursue progress.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 3, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Šádek, Kryštof. *True Random Number Generator on FPGA*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

This thesis deals with the implementation of a true random number generator on FPGA development board building on pair of ring oscillators and explores the influence of temperature and power supply changes on generated output, evaluated by NIST-inspired tests. The results show that environmental changes does not impact the ouput in any significant way.

Keywords TRNG, ROPUF, FPGA, randomness, testing randomness, enviromental tests, true random number generator

Abstrakt

Tato práce se zabývá implementací hardwarového generátoru náhodných čísel na FPGA vývojové desce, který staví na páru kruhových oscilátorů a zahrnuje testování vlivu změn teploty a napájecího napětí na generovaný výstup. Vyhodnocení staví na NIST testech. Výsledky ukazují, že změny prostředí neovlivňují výstup generátoru žádným významným způsobem.

Klíčová slova TRNG, ROPUF, FPGA, náhodnost, testování náhodnosti, testování vlivu prostředí, hardwarový generátor náhodných čísel

Contents

List of Figures

List of Tables

Introduction

As of ever-increasing emphasis on security, almost every device is required to be capable of cryptographical utility to some extent. Many cryptographic protocols require random numbers to provide secure encrypted data. However, the challenge lies in the fact that computers are deterministic and usually cannot give genuinely random numbers. There are a couple of ways how to obtain a so-called source of entropy for True Random Number Generator (TRNG), and their underlying principles are covered in this thesis.

At the same time, devices could be required to generate not random but unique numbers. Either for their identification as a fingerprint of a particular device or to generate a private key for asymmetric ciphers. This could be accomplished by Physically Unclonable Function (PUF), which uses imperfections and mismatches arising during the manufacturing process, causing every device to be unique. PUF is being implemented similarly to TRNG. Therefore the same circuit could be used for generating both random and unique numbers. This thesis focuses on specific implementation [1] of PUF circuit based on Ring Oscillators (RO) with an emphasis on it used as TRNG.

The aim of this is twofold. The first part consists of implementing above mentioned PUF based on ROs (ROPUF) on Field-programmable gate array (FPGA) board Digilent Cmod S7 in a way that it significantly simplifies future research and development of given circuit. How to achieve it is to simplify the ROPUF written in Very High-Speed Integrated Circuit Hardware Description Language (VHDL) and handle as much as possible in a soft microprocessor core MicroBlaze programmed in C language. MicroBlaze would then process data acquired from ROPUF and provide basic analysis. For example, frequency and runs test. Part of the proposed solution consists of a simple communication protocol over UART between FPGA and PC. The second part deals with the behavioral analysis of ROPUF under various external conditions. Due to the nature of a discussed solution, the goal is to determine whenever changes in, for example, temperature lead to changes in quality of output. Given the example of temperature, the preposition is

that with increasing temperature, imperfections in electrical circuits become more prominent, variation between different parts accumulate, and the level of entropy should rise.

The thesis is laid out in the following manner. The opening chapter explores a general TRNG composition. Then it discusses available randomness sources on FPGA and, lastly, currently existing prominent ways of TRNG implementation built on mentioned randomness sources. The next chapter deals with the specific design of ROPUF and gradually unravels the idea of PUF implemented as TRNG. The closing part of this chapter deals with randomness evaluation in general and how it is handled in this work. Those two chapters account for the theoretical part of the thesis. The practical part is laid out in chapter *Design and Implementation* and *Measurements*. The former briefly describes used tools and then separately goes through circuit design, software design for MicroBlaze processor, and software design of Python script running on a computer. The latter explores obtained data and interprets them in the context of carried out research. The last chapter concludes the whole thesis with an evaluation of set goals and draws some implications concerning future work.

Chapter 1

State-of-the-art

This chapter first describes the composition of a truly random number generator. Then it briefly describes randomness sources available to use on FPGAs, and finally, it goes through various existing approaches.

1.1 True Random Number Generator Composition

TRNG can be broken down into four distinct parts: a source of entropy, entropy extraction, post-processing, and built-in tests [2].

The entropy source is a physical process, too complicated or volatile. Such a process would be nearly impossible to describe in a mathematical model. Hence, it provides defense against predicting the following output sequence and reconstructing previously generated numbers. Some examples of such processes could be time between emissions during radioactive decay, quantum random process, shot noise from Zener diodes, or user interaction.

The entropy extraction is rather straight-forward, although unnecessary part. Its function lies in a regular sampling of analog signals generated by entropy source and conversion into a digital one. Together with the entropy source creates *a source of digital noise*.

Paradoxically, the randomness of digital noise introduces a problem in itself. The output could be unbalanced, thus vulnerable to statistical analysis. The post-processing is there to solve precisely this. Usually, the post-processing consists of deterministic function which aims to improve statistical attributes of the output.

The built-in tests observe TRNG health. Even though some prototype of TRNG is thoroughly tested before distribution, the quality of every single unit is *not* guaranteed. Reasons are differences in the manufacturing process, aging of components, the influence of the environment, or even an attacker's active effort to degrade the level of randomness. According to W. Schindler and

W. Killmann [3], "...to detect such defects TRNGs should perform start-up tests, online test and so-called tot[1] tests while they are in operation".

1.2 Randomness Source on FPGA

Without utilizing an external source, the only available physical processes on FPGA are jitter and meta-stable states in logic circuits. The latter exploits the fact that logic circuits such as D-type flip-flop require some time to settle on either of logical levels. If this time window is violated, the circuit may enter a meta-stable state. The meta-stability describes the fact output value is not deterministic, thus suitable as a randomness source. However, this method is not widely used considering high precession timing requirements and uncertainty of meta-stable state even occurring. Instead of relying on meta-stability, jitter is utilized.

Jitter is a term describing deviation in a periodic signal. If a digital clock signal is considered, the edge does not occur at exact intervals. Rather, Gauss distribution [4] around "ideal" timing is observed. This behavior is usually undesired but can be used as a randomness source. TRNG implementations drawing upon the usage of jitter as their source of randomness are doing so usually through ring oscillators. The ring oscillator is a chain consisting of logic gates in an odd number. A possible solution is to use an even number of NOT gates and single NAND gate between, allowing for enabling/disabling RO.

1.3 Existing Implementations of TRNG on FPGA

This section describes some of the existing and recently explored implementations of the true random number generator on the FPGA, focusing on various approaches to previously mentioned principles of obtaining randomness.

1.3.1 TRNG Based on Meta-stability

Contrary to section 1.2, some designs using meta-stability exists. One of those is described by Jean-Luc Danger and his colleagues [5]. They propose to design TRNG capable of higher bit-rate then usual RO-based. Proposed structure (see Figure 1.1) uses series of latches made by looped Look Up Table (LUT) blocks. Both data and enable input are wired to a global clock with a purposed delay for data input. This results in LUT entering a meta-stable state and each at different times. Their output is then XORed and synchronized by pair of D-type flip flops.

[1] total failure of the noise source

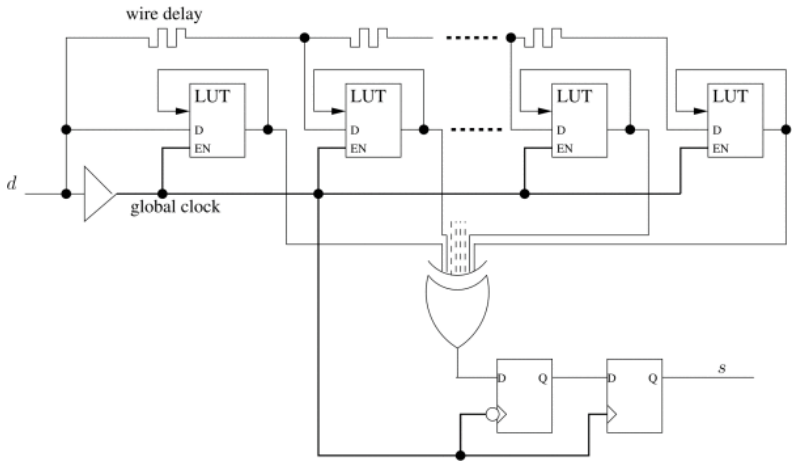

Figure 1.1: Very High Speed TRNG based on an open loop structure [5].

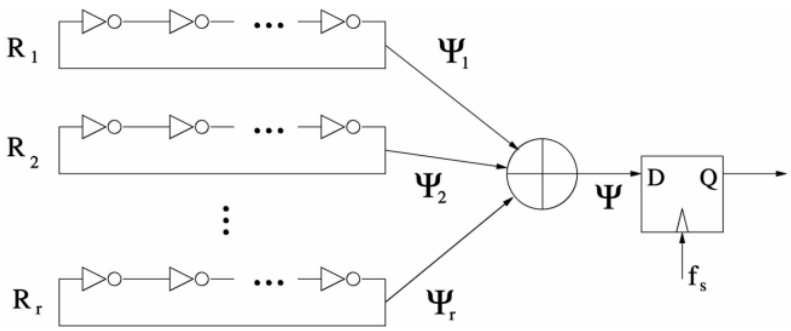

Figure 1.2: TRNG composed of multiple ROs [6].

1.3.2 TRNG Using Multiple ROs

Multiple implementations were proposed during recent years. However, the trend started with a paper by Sunar, Martin, and Stinson [6]. Their contribution simplified the following designs significantly while solving many issues introduced by their predecessors. The key idea (see Figure 1.2) is to have several ROs, and their output then hashed into a single bit by XOR gate, supposedly realized as a binary tree structure. Because the desired jitter occurs only during the rising edge of the RO output signal, the amount of entropy is reasonably limited. Naturally, increasing the number of ROs saturates the percentage of jitter compared to the deterministic part of the signal. The authors found out the best results are achieved when the length of RO is constant across all of them, which was then successfully demonstrated [7]. Nevertheless, potential performance shortcomings surfaced concerning the XOR gate. The following subsection 1.3.3 describes improved design addressing this issue.

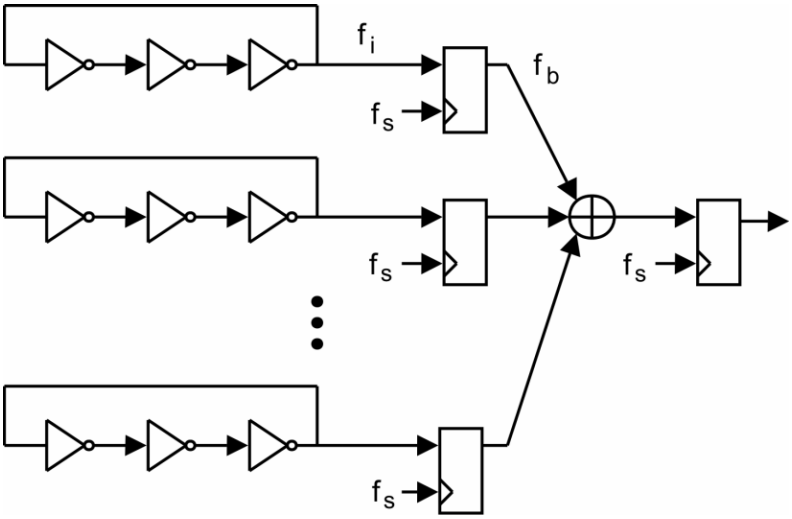

Figure 1.3: Improved design of TRNG composed of multiple ROs with additional D-type flip flop circuits [8].

1.3.3 Improved TRNG Using Multiple ROs

The problem with TRNG put forward by Sunar et al., is that the DFF circuit cannot handle transitions from all the ROs properly, consequently causing it to enter a meta-stable state. Moreover, even though a random number generation is in a picture, this behavior is undesired in this case, because it reduces the amount of entropy significantly. Proposed solution [8] considers using additional DFF for every RO before its signal runs through XOR (Figure 1.3). This prevents the propagation of unwanted state through XORs binary tree structure.

1.3.4 TRNG Utilizing Phase-locked Loop

Another approach to utilize jitter as a randomness source builds on a Phase-locked loop (PLL). The difference lies in that the PLL is part of the built-in circuitry and not synthesized inside the FPGA. Rather it needs to be available. For example, the Spartan-7 chip has three PLL available to use. The PPL generates a clock signal, which is *phase-locked* onto the input clock signal. The output frequency can differ. The general principle of PLL is shown in Figure 1.4. "[V]arious noise sources cause the internal voltage controlled oscillator (VCO) to fluctuate in frequency. The internal control circuitry adjusts the VCO back to the specified frequency, and this change is seen as jitter." [9]

The same paper then proposes a method to extract this jitter by using a D flip flop circuit followed by a decimator that filters the actual random bits. The PLL output frequency is determined as $f_{out} = f_{in} * K_M/K_D$, where $GCD(K_M, K_D) = 1$. The decimator applies K_D value, thus defining the

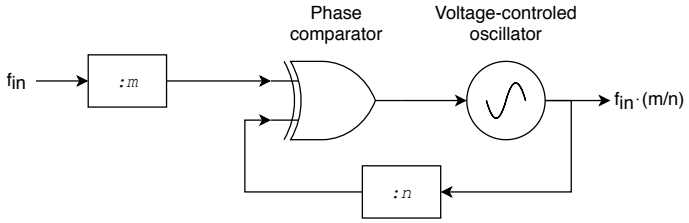

Figure 1.4: The phase-locked loop.

output rate. The conclusion shows the best results are obtained when K_D is an odd number with $f_{in} < f_{out}$. The following work [10] focuses on further optimization and uses the genetic algorithm to get the most efficient parameter configuration.

1.3.5 Chaos-based TRNG

Even though digital noise is usually obtained from a physical process, some authors believe an oscillator composed of a chaotic system can be entertained. Chaos theory proposes some functions deterministic in their nature are considerably sensitive to their input, making the output practically unpredictable. Such a system is then described in the differential equation set. An example of this approach is set forth by the implementation of the Lorenz chaotic model on FPGA in purely digital fashion [11]. While the random generator passed standardized tests based on a chaotic oscillator [12], it is unclear whenever true randomness is obtained. To conclude, the chaos-based design is mentioned because of the increasing interest but would fall into the category of pseudo-random number generator, unless further research proves otherwise.

Chapter 2

Analysis

This chapter continues the theoretical part of this thesis. It goes through an explanation of the physically unclonable function and one concrete design that uses a ring oscillator pair to obtain the output. Then it explores the idea of using this concrete PUF as a truly random number generator. The last part of this chapter describes the evaluation methods utilized in this thesis.

2.1 Ring Oscillator-based Physically Unclonable Function

This section deals with explaining the principle behind the physically unclonable function and the concrete design using ring oscillator pairs.

2.1.1 Physically Unclonable Function

Previous chapter 1 discussed popular approaches to implementing TRNG on FPGA. However, this thesis carries forward a different method proposed in a paper by S. Buchovecká et al. [1]. Their design uses PUF circuit [13] and employs it in the form of TRNG.

PUF is a function exposing the inherent properties of an electronic circuit, determined by the manufacturing process. Moreover, because a function requires input value, the same holds for PUF. The value is called a challenge. PUF then can be described as a function that returns a unique response to the same challenge for each device. This behavior is useful for generating private keys in a zero-trust environment because the key is not stored in memory, which limits the possibility of compromise significantly. Due to the nature of PUF, not only manufacturing aspects cause a difference in output, but also properties of the current environment. For that reason, some extra steps need to be taken in order to ensure a stable response for a specific challenge across time.

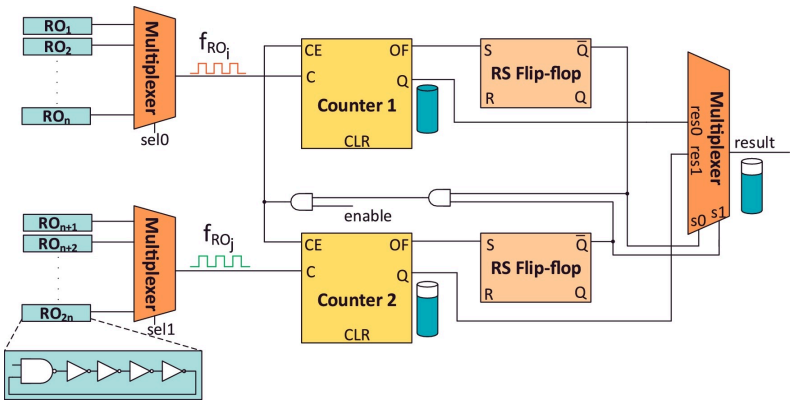

Figure 2.1: Structure of RO-based PUF [14].

2.1.2 The Main Principle of Proposed ROPUF

Proposed design [13] uses a pair of ROs, each connected to a counter. Both counters start simultaneously counting up until one of them overflows. The overflow of one causes both to stop, resulting in a specific value present in the other counter. This value is ultimately subject to the physical properties of the RO pair. A circuit layout supposes some delay between overflow detection and disabling counters. However, this delay is constant for both counters, meaning constant positive offset for the considered value. Finally, the design uses multiple RO pairs connected to multiplexers instead of counters directly. The consequence is RO pair selection through multiplexer serves as the challenge for a PUF. The schema shows Figure 2.1.

2.1.3 The Attributes of ROPUF Output

Now each acquired bit position for a selected RO set has an attribute called *stability*. A definition for the stability shows Equation 2.1 [13].

$$s_i(RO) = \begin{cases} P(b_i = 1) & \text{if } P(b_i = 1) \geq 0.5 \\ 1 - P(b_i = 1) & \text{if } P(b_i = 1) < 0.5 \end{cases} \tag{2.1}$$

$P(b_i = 1)$ stands for a probability of i position being set. The probability is an average of k times measured value. Whenever a given position is suitable for PUF shows the average stability across every RO pair.

The second attribute crucial for consideration is entropy. For ROPUF, entropy renders how much differs the output of the same challenge for different devices. The paper [13] puts forward two parameters, H_{intra} and H_{inter}, see

Equation 2.2 and 2.3.

$$H_{intra}(i) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{k=0}^{1} p_j(k)log_2(p_j(k)) \tag{2.2}$$

$$H_{inter}(i) = -\frac{1}{n}\sum_{j=1}^{n}\sum_{k=0}^{1} p_t(k)log_2(p_t(k)) \tag{2.3}$$

What H_{intra} describes is "[t]he average entropy of bit position i within each FPGA separately", where p_j is probability of bit being either set or clear. p_j averages major measured output for each RO pair. H_{inter} differs in that p_t averages across each FPGA for same RO pair. Hence H_{inter} considers the average entropy across FPGAs for n pairs.

2.2 ROPUF as TRNG

This section lays out how the ring oscillator-based physically unclonable function can be utilized as a truly random generator. It also highlights the difference between those two and how to account for them when implementing the TRNG.

2.2.1 Evaluation of ROPUF Output Behavior

The previous section 2.1.1 talks about a different topic then TRNG. But the described principle is strikingly similar to other TRNG designs discussed in chapter 1 before. Further evaluation of bit stability and bit entropy attributes bridges the gap. ROPUF output is an internal value of a counter or, in other words, a binary representation of a number. That means every position further from LSB changes twice less than the previous one. Therefore it trivially implies stability increases towards MSB, and by contrast, the near-LSB area displays high volatility. Both, H_{intra} and H_{inter}, behave in the same way – MSB position displays almost zero entropy and LSB just the opposite.

2.2.2 Difference Between PUF and TRNG

Now, PUF requires stable bits with good entropy. Ideal value would be 1, although compromise must be met considering the conflicting nature of those attributes. Thorough measurements show position 7-8 [13] are the most suitable[2]. At the same time, stability is an unwanted behavior when considering TRNG. That means PUF and TRNG differ only in a single requirement, which

[2]Improved version [14] deploys gray code as a form of post-processing, which results in the twice as long usable window of position 7 to 10.

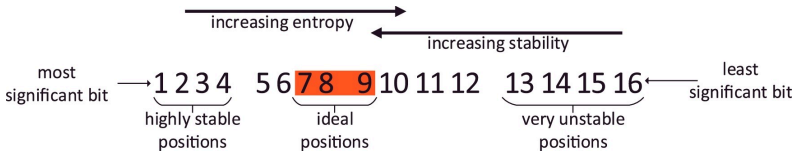

Figure 2.2: Ideal bit position for PUF functionality [14].

makes it that TRNG output selection is considerably easier to determine. Another case could be made for PUF having input while the only random number generator which has input is PRNG.

2.2.3 Adjusting ROPUF to Work as TRNG

Finally, everything is laid out for the utilization of ROPUF as TRNG. The first correction lies in changing the stability requirement. Ideal value is then $s_i(RO) = 0$. Close to zero stability has been proven [1] to exist in bits 1 to 3, with the least significant position zero performing worse.

Considering the excess of input, some other method of RO pair selection other than from the outside needs to take place. This presents no challenge to accomplish. The general approach, though, should aim for the simplest algorithms only. Additional complexity would not influence the entropy level as it is determined by the entropy of each RO pair. It would only introduce another difficulty when analyzing the final design.

2.3 Methods of Evaluation

This section goes into detail of the tests used to evaluate the true random number generator output.

2.3.1 NIST Test Suite

The method of evaluation lies in determining whenever generated output performs as random or not. Conversely, true randomness cannot be determined similarly. For establishing true randomness, a mathematical model of the physical process must exist, which is out of the scope of this thesis. Instead, "trueness" is assumed, and only statistical tests run for the output. This paper relies upon tests specified in NIST SP 800-22 [15]. Different tests are applied to the obtained values, but two are discussed in detail due to the nature of implementation, frequency, and runs tests. Furthermore, this should provide some idea of how NIST tests are framed.

2.3.2 Frequency Test

The frequency test supposes randomness of a bit sequence whenever the number of ones and zeros is roughly the same. The function requires mapping the bit values to $(-1, 1) \in \mathbb{Z}$ range. The procedure is:

1. Get the sum of n-sequence S_n
2. Calculate $s_{abs} = \dfrac{|S_n|}{\sqrt{n}}$
3. Compute $P\text{-}value = \texttt{erfc}\left(\dfrac{s_{abs}}{\sqrt{2}}\right)$, where `erfc` is a function from the standard C math library for the complementary error function

The *decision rule* states that "[i]f the computed *P-value* is < 0.01, then conclude that the sequence is non-random" [15]. *P-value* is `erfc` function for every test, only the parameter differs. Same applies to the decision rule.

2.3.3 Runs Test

A run is defined as a bit sequence of the same value without interruption bounded by the opposite value on both ends. Thus, the runs test checks if the number of runs is adequate for a random string. The prerequisite for this test is passing the frequency test first. *Test statistic* in this case is V_n and is defined as:

$$V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1 \tag{2.4}$$

Where $r(k) = 0$ for k-bit value being equal to the of following and 1 otherwise. The way the *P-value* is stated, the test is sensitive for changes between one and zero occurring too often or too scarcely.

2.3.4 Test Results Interpretation

The statistical tests in general aim to conclude if a *null hypothesis* H_0 is correct in the contrast with a *alternative hypothesis* H_A. For that it is necessary to define a *critical level* which would encompass test results with low chance of occurring. The already mentioned *P-value* is the probability of exceeding the *critical level* under the condition of H_0 being true (Type 1 error). By definition, if random variable is tested multiple times, the *P-values* must be uniformly distributed over the $[0, 1]$ interval.

Given the fact that the sequence consists of a finite number of discrete values, the *P-value* is also discrete and thus not continuous. To properly evaluate the observed set of *P-values*, the NIST proposes to either evaluate histogram

of the *P-values* with ten subintervals or by applying the χ^2 test and determining if the *P-value* correspond to the Goodness-of-Fit Distributional Test. "If *P-value* ≥ 0.0001, then the sequences can be considered to be uniformly distributed" [15].

Complementary approach is to define confidence interval and observe the proportion of passed tests (again, the generator output is split into multiple sequences tested separately). "The confidence interval [is] calculated using a normal distribution as an approximation to the binomial distribution" [15]. If the pass ratio is lower *or* higher, the output is presumably not random.

2.4 External Conditions

Due to the nature of the randomness source, it is worth exploring the impact environmental changes have on the quality of output. Not only that resulting the TRNG would run in different conditions, but also an attacker could try to rig generated noise. This kind of active attack aims to diminish the level of entropy, thus making the output predictable. A number of tests exist. For example, placing the TRNG into the oscillating electromagnetic field and observing the impact of various periodical frequencies. However, this thesis focuses on the influence of temperature changes and supply voltage variances.

The tests utilize a dynamic climate chamber, BINDER MK 56 [16]. Similar work is described in a paper [17] focusing on thermal changes influencing the PUF functionality. The paper concludes how rising temperature deepens the frequency difference in the RO pair. The determining aspect is the length of the signal path for given RO. Their respective output expresses the following equation:

$$CNT = \frac{f_1}{f_2} \times 2^{16} \tag{2.5}$$

Considering both of those facts, it is clear that temperature changes are more prevalent for ROs with significantly different signal paths as output depends on frequency *ratio*. Naturally, PUF recognizes this as problematic, so some countermeasures must take place. The authors of the paper propose symmetric RO design, which diminishes unwanted output variations greatly. TRNG has no such requirement, aiming actually for zero stability. To conclude, an area worth exploring is around lower temperatures, which might stabilize output and degrade TRNG into PUF. This fact would significantly compromise the security of the whole system. For analysis sake, the symmetric design is preferred as variable stability would be limited, and only potentially unwanted changes remain.

Next, supply voltage needs to maintain a proper level within defined bounds. Because the development board contains a switched power supply, a slight modification would allow for a change of this value. The switched power

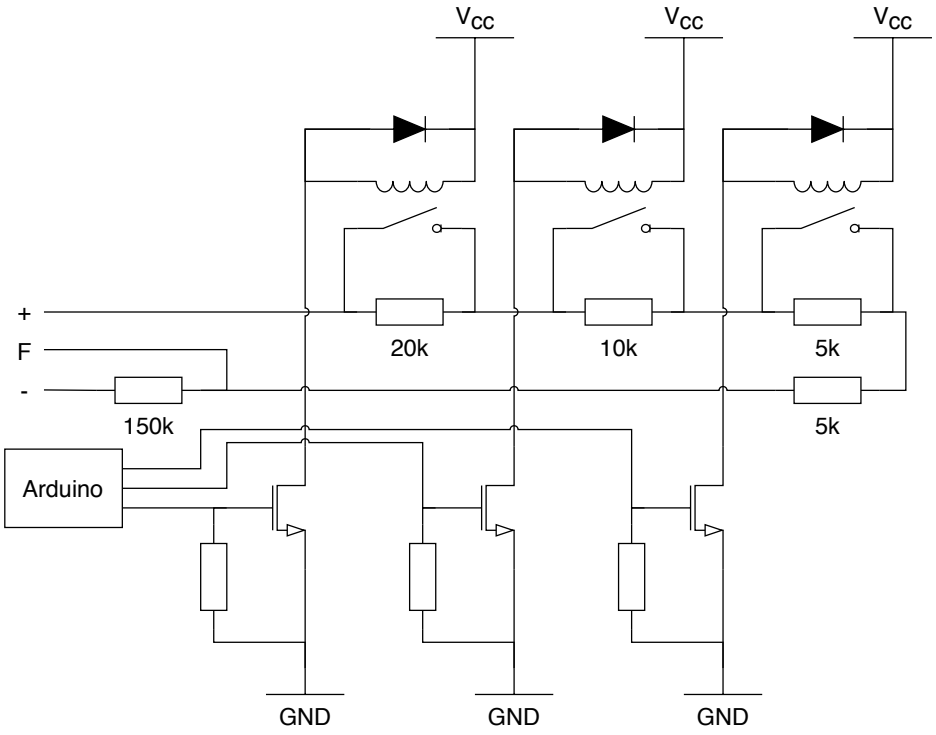

Figure 2.3: The schematic of a configurable voltage divider.

Choice	0	1	2	3	4	5	6	7
Voltage [V]	1.118	1.078	1.038	0.998	0.958	0.918	0.879	0.839

Table 2.1: Voltage levels for the respective choices.

supply is a circuit with a feedback loop. This feedback builds on wanted voltage level and is realized as a simple divider. The power supply can be forced into adjusting the voltage on different than the designed level by modifying the ratios of the divider. Even better would be to use linear power supply allowing for continuous change, but the current version does not allow for that. So by fixing external temperature and changing the supply voltage within prescribed bounds, its feasible to observe a change of stability. The expectation is to see decreasing stability when the voltage differs from the proper level.

The measurements are conducted in the environment portrayed in Figure 2.4. The Arduino board controls the modified power supply circuitry (see Figure 2.3), and together with the FPGA board is connected to PC via a USB hub. The FPGA board is placed inside the climate chamber, as well as the voltage divider. The temperature sensor is placed directly onto the FPGA

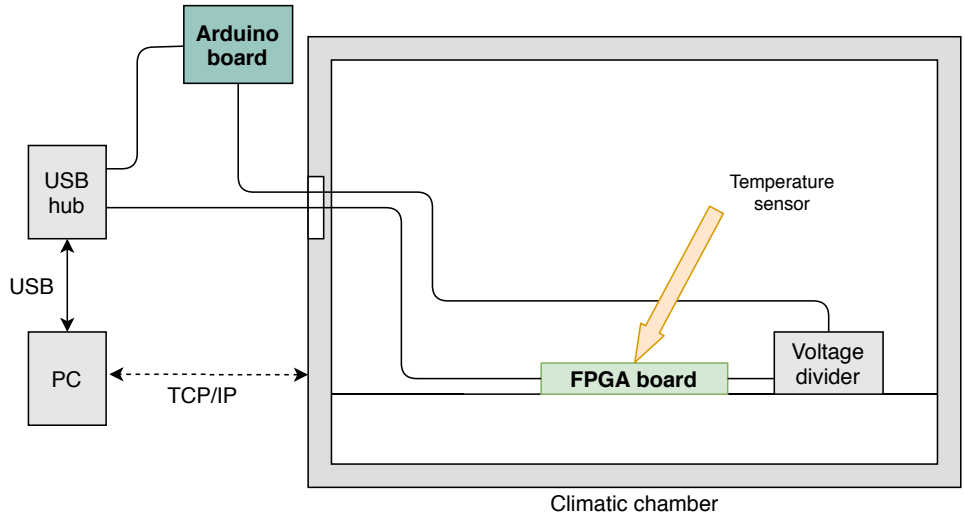

Figure 2.4: Diagram which shows the experiment configuration.

chip and is linked with the climatic chamber. The chamber then communicates with the PC over the TCP/IP. Table 2.4 shows measured voltage values at a fixed temperature according to various voltage divider setup as controlled by the Arduino board.

Chapter 3

Design and Implementation

The implementation uses the FPGA development board Digilent Cmod S7 with the Spartan-7 FPGA chip from the Xilinx family. The board contains a serial to USB converter, which significantly simplifies the usage of said board. The board is fully supported by Xilinx Vivado Design Suite 2019.2, which encompasses tools needed for developing both circuitry and software.

3.1 Hardware

The general approach for functional implementation requires not only the TRNG circuitry itself connected to the MicroBlaze processor, but also some other peripherals. Those ensure complete functionality of the final product. The whole block design is shown in Figure 3.1.

3.1.1 Minimal MicroBlaze Processor Generation

The environment simplifies the designing process considerably, as every necessary part which the processor needs for proper function generates and interconnects automatically. For that reason, it is not necessary to discuss every detail, and only a brief explanation suffices. The created blocks are as follows:

- Clock Wizard
- BRAM Module
- AXI Interrupt Controller
- AXI Interconnect
- Microblaze Debug Module (MDM)

The Clock Wizard [18] provides a configurable wrapper around external clocking primitive (e.g., a crystal oscillator). Although not utilized in this

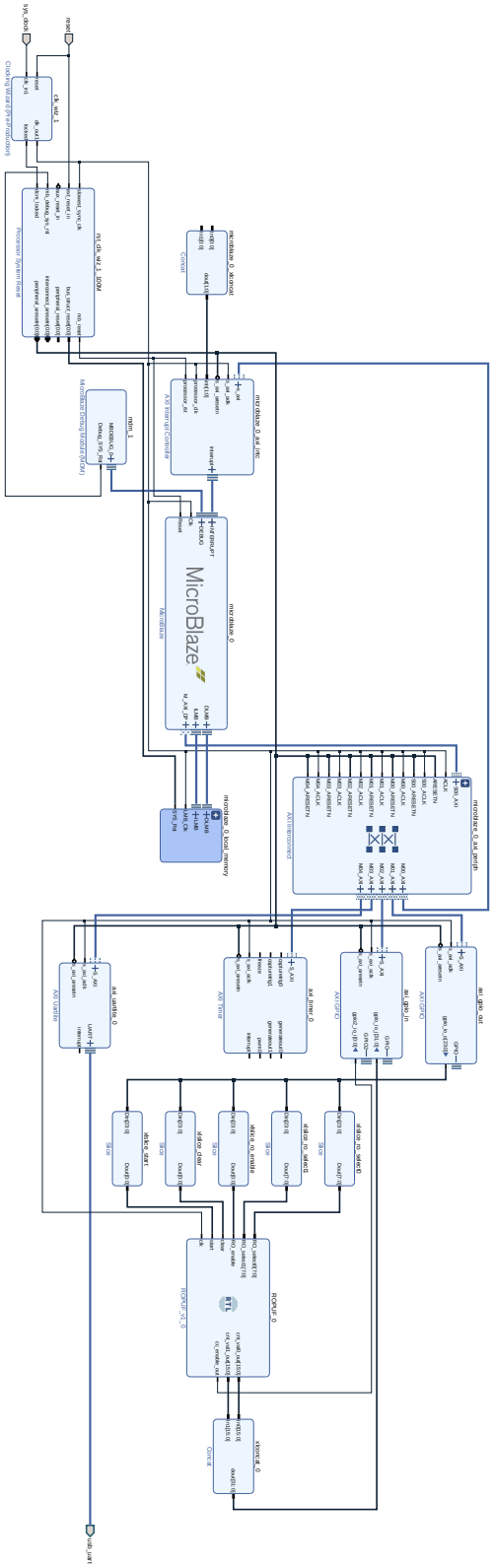

Figure 3.1: The block design of a whole project.

design, the circuit allows for dynamic reconfiguration through the AXI bus. Another advantage is the possibility of having multiple (up to seven) output clocks with a defined phase relationship. Relevant is a feature focused on minimizing output jitter and increase tolerance to input jitter as well.

The Processor System Reset [19] is another wrapper similar to the previous one. The difference is trivially in a type of signal handled. This module ensures proper timing and pulse length for individual IPs. Its function can be summarized as synchronization of an external reset signal to the slowest clock. For this concrete implementation, such a clock is the only output of the Clock Wizard. It is worth mentioning that external signals `sys_clock` and `reset` are not connected even when using the auto-connect function.

The BRAM (Block RAM) Module is not always needed. But because the Cmod S7 board does not have any physical RAM, it is necessary to synthesize one directly inside the FPGA. The module consists of five other IPs. Those are two Local Memory Buses (LMB), two LMB BRAM Controllers [20], and a single Block Memory Generator. From the practical point of view, this block is likely *not* to be configured at all. The reason is that the amount of available memory handles the Address Editor window. Available memory is specified by the address range, which is also done during the configuration of MicroBlaze Processor Wizard.

The AXI Interrupt Controller [21] provides an essential capacity for multiple interrupt handling. Although current software implementation does not utilize interrupts in any way, the hardware is capable of doing so. The MicroBlaze processor is equipped with only single interrupt input, which can be extremely limiting for even simple designs. The role of the AXI Interrupt Controller lies in the concentration of multiple (up to 32) interrupts and communicates them through a single AXI interface. In the case of even more demanding designs, the IP supports cascading and thus makes it possible to exceed the basic number of inputs.

The AXI Interconnect [22] is the most used module in this design. Its function is rather simplistic, though, as it can be described as a multiplexor for the AXI bus. The processor has only a single AXI interface, so it is unavoidable to use this IP. From the designer's perspective, it is very convenient the support for auto-connect, which makes adding AXI modules seamless. Peripherals are then accessed through memory operations. The Address Editor allows for review and reconfiguration of those addresses.

3.1.2 Mandatory Peripherals

The following modules are not part of the ROPUF circuitry per se but are indispensable nevertheless. Those two modules are:

- AXI Uartlite
- AXI Timer

Although the board itself provides an interface to facilitate UART, it does not provide anything on top of that. That means only the raw RX and TX lines are available, and the communication protocol must handle FPGA completely. Different ready-to-use IPs exist, but the current implementation goes with AXI Uartlite [23]. The main advantage is a fairly straight-forward connection to the processor through the AXI bus. Of the same significance is a fact that the Uartlite has a comparatively small physical footprint. The current design does not run into such limitations, but this kind of precaution can prove useful in future work. There are some drawbacks, but nothing of critical importance. The most visible is the impossibility to configure a transfer speed dynamically. The transfer speed is "hard-coded" into the hardware, and at the same time, the choice is somewhat limited. The last potential problem might be a limited size of input and output buffers. Both of them are defined as a 16 bit register without the possibility to adjust that size. But as the buffers exist to allow for asynchronous communication, which the current implementation does not utilize, it proves insignificant.

Some techniques to implement timers only in software exists, but those are generally somewhat inaccurate. Moreover, as it might suffice for different scenarios, this one by its nature requires a high level of accuracy. The AXI Timer [24] allows for a wide range of applications. The IP is timer *and* counter, although such thing goes for nearly all hardware timers. The interesting part lies in two 32 bit timers/counters included in a single IP with the possibility to run in a cascade mode, allowing for the total width of 64 bits. When it goes to the actual usage, it would be possible to make it run in *Capture mode* with the *Capture Trigger Input* connected directly to the *Ready Ouput* of the ROPUF. However, the timer is used not only for measuring the ROPUF run time. Therefore the utilization limits itself to a simple stoppable count-up timer available from the program.

3.1.3 ROPUF Inteface Circuitry

Before the ROPUF Module is discussed in detail, the way it is connected to the MicroBlaze deserves more attention. The building blocks go as follows:

- AXI GPIO
- Slice
- Concat

First, it is worth pointing out that different, somewhat more sophisticated methods of connection could have been used. That is, ROPUF module can be reworked in such a way to provide the AXI interface directly. Although this would simplify the final design greatly, it goes beyond the scope of the thesis. Especially as the current implementation does not diminish re-usability, only

the block design is slightly more complicated. Therefore, the current approach goes with utilizing a simple general-purpose input/output hook-up. The advantage lies in the ability to work with the ROPUF module signals separately, which gives the possibility to connect other circuitry in the future.

The central role holds the AXI GPIO module, which allows accessing GPIO through the AXI interface in the way of memory operations. The IP can operate in either single or dual-channel mode, with both channels having configurable width up to 32 bits. Now, the ROPUF module has twenty inputs and thirty-three outputs, meaning a total of 53 signals[3]. Those could easily fit into single two-channel AXI GPIO, but the problem lies in the splitting and connecting of separate wires. So it is necessary to use two of these IPs. Slice and Concat do what their names imply, simply splitting selected part of a bus or joining it together.

The first GPIO module handles only output signals and the other inputs to keep the final design intuitive. Their naming considers the perspective of the processor, as the generator act as a slave device. So `axi_gpio_out` handles signals coming from the processor into the generator and it goes the opposite with `axi_gpio_in`. Also, the signals are concatenated in the same order they enter the module optically.

3.1.4 ROPUF Module

Finally, the ROPUF module in itself is written in VHDL and split into multiple files. The original source code was provided by Ing. Kodýtek and modified by Ph.D. Buček to match needs of this thesis. The main file `ROPUF.vhd` puts all the other together into a single functioning module. The others are simple building blocks and go as follows:

- `COUNTER.vhd` is a 16 bit count-up counter which stops at maximum value indicates such event.
- `RO.vhd` is a ring oscillator consisting of four negations and single NAND gate serving as an enable (see Figure 3.2).
- `RO_SET.vhd` generates 150 ring oscillators and takes 8 bit input for their selection.
- `SR_LATCH.vhd` is an actual SR latch.
- `SYNCHRONIZER.vhd` synchronizes `async_input` to a clock.
- `TFF.vhd` is a T flip-flop.

With all the mentioned parts, the ROPUF consists of two symmetrical lines, which is expected due to its nature. The general function of the ROPUF

[3]Not including the clock signal.

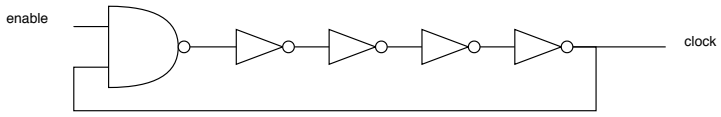

Figure 3.2: The ring oscillator defined by `RO.vhd`.

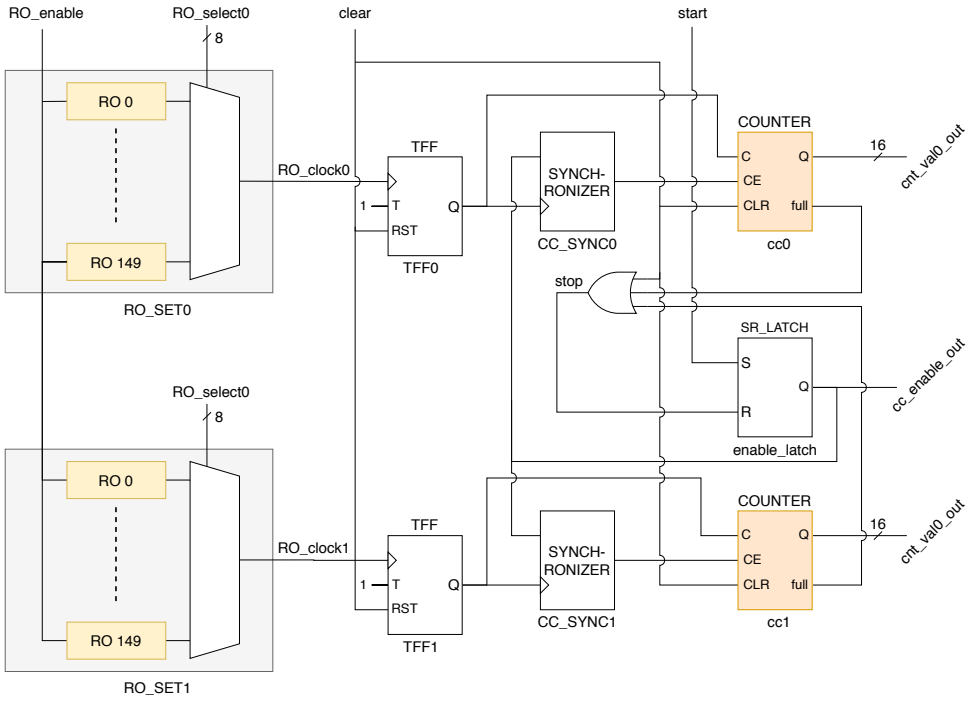

Figure 3.3: The ROPUF implementation defined by `ROPUF.vhd`.

was already discussed in Chapter 2, so it is worth going through pinout only and how does it connect to specific functionality. Both `RO_select0` and `RO_select1` are 8 bit input signals serving for specific RO pair selection. As maximum index value tops at 149, any value exceeding that one is binary stripped at MSB places (i.e., 150 would map to 22). The `RO_enable` controls whenever the ring oscillator are active (set) or not (clear). The `clear` signal empties both counters when it is set and should only be used when counters are not active. The rising edge needs to go into `start` input to start the generation process, but ROs need to be enabled first. The `clk` input is obviously for a clock signal, and that is all for input pins. The output accounts for `cnt_val0_out` and `cnt_val1_out` which are raw counter values and `cc_enable_out` – this one signals whenever the ROPUF is still running

(set) or not (clear). The diagram of described implementation is shown in Figure 3.3. All the the blocks hold their respective names with a description of their instance name.

Now for the recapitulation of how to use the ROPUF module in its current form. The first step is to select the ring oscillator pair and enable ring oscillators. Next is to send pulse into `clear` input and then `start` input. Now, the module is running which is indicated by set value on `cc_enable_out` pin. When `cc_enable_out` clears, the output is available at `cnt_val0_out` and `cnt_val1_out`. It is important to point out that the the ROPUF output is the value of counter which is not filled. That means the ROPUF ouput can be obtained with bitwise multiplication of both counter values. Also, input signals need to stay set in the required state if not specified otherwise.

3.2 MicroBlaze Software

The modular approach is taken to assure the re-usability of the running program. Thanks to such a decision, the currently running solution is ready to use as it is, but when extensive changes take place, it is not necessary to scrap everything, and individual libraries can be used as building blocks. This section explores all the libraries and then the final product as a separate thing, which uses all of the previously described parts.

When it comes to consistency, all the libraries are named by their respective names. They are split into a header (`.h`) and source (`.c`) file. Every function starts with its library name and follows the "CamelCase" naming convention. The constants use all-caps separated by underscores and variables low-caps with underscores. Every library revolves around an object of a sort. Every object is C structure defined as a named datatype with a `_t` suffix.

3.2.1 Analysis Library

The analysis library implements statistical tests for the randomness evaluation following the NIST test suite. Currently, the library is equipped to perform frequency and runs tests. The library object is `stats_t`, and compared to the others is the only one expected to be created multiple times. Its role is to hold statistical data about a specific bit sequence. It is worth pointing out all the variables are unsigned 32-bit integers. The actual library consists of a following functions:

- `void AnalysisClearStats(stats_t *stats)`
- `void AnalysisLoadValue(stats_t *stats, u16 value)`
- `u8 AnalysisFrequencyTest(stats_t *stats)`

- `u8 AnalysisIsFrequencyDirty(stats_t *stats)`
- `u8 AnalysisIsFrequencyPassed(stats_t *stats)`
- `u8 AnalysisRunsTest(stats_t *stats)`
- `u8 AnalysisIsRunsDirty(stats_t *stats)`
- `u8 AnalysisIsRunsPassed(stats_t *stats)`

The first two provide general usage. The `AnalysisClearStats` function clears attributes of respective objects and sets them to a default state. That would account for setting test result flags into a `TEST_FAILED` state and dirty flags to a `DIRTY` state. The `DIRTY` state signifies test result is no longer valid and needs to be calculated again. The `AnalysisLoadValue` function works as a setter. Its sole role is to update the `state_t` object properly with the latest data from the observed sequence. Every call of this function makes test results dirty.

The rest of the functions can be split into two parallel groups. Each triplet deals with a separate test defined in the NIST test suite. `AnalysisIsXPassed` functions serve as simple one-to-one getter of previously calculated `X` test result. And similarily `AnalysisIsXDirty` functions are getters for the dirty flag value. The argument expects the value of either one or zero, and anything else does not guarantee proper function. Lastly, `AnalysisXTest` implements the tests directly as described in NIST 800-22 [15]. Notation follows their one as closely as possible without breaking naming conventions set for this project. The return value does not reflect the test result directly. Rather, it returns `XST_FAILURE` whenever a test cannot be performed (e.g., not enough data are available) and `XST_SUCCESS` otherwise. The direct result is available through a respective getter and can be either `TEST_PASSED` or `TEST_FAILED`. The dirty flag is cleared to a `~DIRTY` value when the function returns success. All calculations run with float precision of four bytes.

3.2.2 Message Library

The message library encompasses communication protocol, which is directly interlinked with the XUartLite library. That means some changes would be required to replace the AXI Uartlite with anything else. At the same time, this library heavily relies upon the timer library. The reason being the protocol has built-in timeouts to ensure dead-lock avoidance and communication behavior predictability. The protocol uses a form of packet communication, as shown in Figure 3.4. The protocol overhead is three bytes per packet with a maximum data size defined by `MESSAGE_MAX_DATA_SIZE` constant, which is calculated from `MESSAGE_BUFFER_SIZE` with a default value of sixteen bytes. The ID specifies what type of message is coming through and telling the receiver what it should be parsing. The message length is part of the message itself,

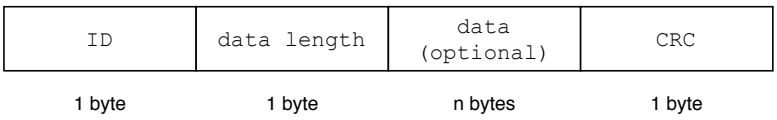

Figure 3.4: Message structure and respective data width.

providing flexibility in adjusting or expanding the protocol itself. Data length has the maximum limit to prevent flooding. Every message ends with CRC byte calculated as an XOR of all the previous bytes. The data part is optional, with zero-length as a valid option.

The library object is `msg_t` and practically serves as a buffer sending and receiving. The only other function is to hold a pointer to a timer library object and XUartLite handler. The library provides the following functions:

- `int MessageInit(msg_t *message, u32 device_id, tmr_t *timer)`
- `void MessagePointerReset(msg_t *message)`
- `void MessageBufferPush(msg_t *message, u32 data, u8 len)`
- `int MessageSend(msg_t *message, u8 msg_id)`
- `int MessageRecieve(msg_t *message)`
- `u8 MessageGetId(msg_t *message)`
- `u8 MessageGetSize(msg_t *message)`
- `u8 *MessageGetData(msg_t *message)`

Again, those functions can be split into multiple groups. Starting with the last one, the last three functions in a list are getters for respective values. They provide access to values already held inside the object buffer and does not communicate with the UART in any way. The message must be received first to get any valid data through those functions. And that is role of the `MessageRecieve` function. When called, it checks whenever data are available through the UART bus and immediately returns `XST_NO_DATA` if not. Otherwise, it performs whitelist check comparing obtained byte with a valid input message codes. This is the only place requiring code change when expanding the protocol of new messages. Failing to pass the check means flushing the content of UART FIFOs and returning `XST_INVALID_PARAM`[4]. The next step

[4]All functions are typically returning set of codes from Xilinx header file `xstatus.h`. That means sometimes they might sound not precisely appropriate.

is to obtain the expected message length. This operation runs with a timeout and can fail by either taking too long (`XST_TIMEOUT`) or its value being too large (`XST_ERROR_COUNT_MAX`) in which case FIFOs flushing takes place. When the value is valid, the function proceeds with receiving the indicated amount of data plus CRC. This operation is again running under timeout and returns the same error code if it fails. The last part is to compare incoming CRC with a calculated one. Error is signaled as a `XST_DATA_LOST`.

The next group of functions deals with sending a message. The process is split to stay universal without the need for excessive code changes when modifying the protocol. The first step is to call `MessagePointerReset`, which moves the internal buffer pointer to the beginning of the message data block within the message buffer. Next, `MessageBufferPush` can be used to push data inside the message buffer up to four bytes at a time. The drawback lies in need for manual data splitting but, on the other hand, gives full control when it comes to the endianness of the data and simplifies protocol modification, as already mentioned. The last part is to transmit the message, which is done by calling the `MessageSend` function. The argument is the message ID, which is placed to the front of the message buffer. The data length is also calculated here, and the same goes for the CRC. Finally, the function tries to transmit the message under the active timeout. It can either fail with the error code `XST_TIMEOUT` or succeed (`XST_SUCCESS`).

The only remaining function to discuss is `MessageInit`. Its purpose is to make sure UART circuitry works as expected. The timer is not checked with the presumption it was done beforehand.

3.2.3 ROPUF Library

The ROPUF library is the centerpiece of the whole project. The `ropuf_t` library object ensures connection to the AXI GPIO modules and maintains input values. Also, the library provides functionality only to read a configured range of bits. Constants `ROPUF_OUTPUT_LSB` and `ROPUF_OUTPUT_MSB` define the range. None of the following functions operates in a safe mode to minimalize processing overhead. It means no checks of input validity are performed. The programmer is responsible for the proper usage of this set of functions. The functions[5]:

- `int RopufInit(ropuf_t *ropuf, u32 gpio_in, u32 gpio_out)`
- `u8 RopufGetWord(ropuf_t *ropuf)`
- `u16 RopufReadOutput(ropuf_t *ropuf)`
- `u32 RopufReadCounters(ropuf_t *ropuf)`

[5]The libraries may contain some other functions with an underscore prefix. Those are meant for internal purposes only, but might be accessed too. go as follows

- `int RopufIsReady(ropuf_t *ropuf)`
- `void RopufStart(ropuf_t *ropuf)`
- `void RopufClear(ropuf_t *ropuf)`
- `void RopufDisableRO(ropuf_t *ropuf)`
- `void RopufEnableRO(ropuf_t *ropuf)`
- `void RopufSelectRO(ropuf_t *ropuf, u8 ro0, u8 ro1)`

Again, the `RopufInit` function ensures both AXI GPIO modules are functioning and return error code if not. It sets data direction on pins and it clears input register.

The next three functions all provide an interface to the ROPUF output. Both `RopufReadOutput` and `RopufReadCounters` functions simply read values available on the ROPUF module output pins. The former returns XORed content of both counters with the previously mentioned range applied as a mask and shifted down to the least significant position. The other simply returns the content of both counters joined together as a single 32 bit unsigned integer. The `RopufGetWord` function is the only active one in the sense that it runs the ROPUF module. It does *not* check whenever the ring oscillators are enabled, so mishandling it may lead to dead-lock. It uses the `RopufReadOutput` function to extract value from the ROPUF module, which is run in a loop to extract eight bits of data. It may run extra time, so it is not advisable to use during analysis for not missing any data. The `RopufIsReady` function technically accesses the output too. It is just the single bit signalizing whenever the module is running, but the value is negated, so it returns true when the generating process is over.

The `RopufStart` and `RopufStop` functions are an interface to respective inputs and send in just the pulse (this goes back to the discussion of how the ROPUF module works).

The last three functions all interact with the ring oscillators. The ROs start in a disabled state and need to be enabled with `RopufEnableRO` first. Otherwise, the program may fall into dead-lock when waiting for `RopufIsReady` returning true. The countermeasure would mean using a timeout, but since controlling the ROPUF module is reliable, it is sufficient to merely enabling ROs first. The `RopufSelectRO` function handles ring oscillator selection of the pair with a default choice of first (index zero) oscillators. The input values are not checked, but the previous section explored how this situation is handled by the module and is not causing any undefined behavior, let alone damaging the circuitry.

3.2.4 Timer Library

The last library stands out in that the message library directly uses it. That means it can be considered part of that library, but its usefulness goes beyond the communication area. It is separated for that reason, and it also neatly revolves around a new library object. The object for the timer library is `tmr_t`, and it is just a handler for the AXI Timer module with a single extra variable holding the previously saved timer content. The way how to use the timer is to turn it on and save its content. Then run the metered operation, and after it is done, reread the timer and calculate the difference. Those functions are available to simplify such process:

- `int TimerInit(tmr_t *timer, u32 device_id)`
- `u32 TimerStart(tmr_t *timer)`
- `void TimerStop(tmr_t *timer)`
- `u32 TimerGetDifference(tmr_t *timer)`
- `u32 TimerGetValue(tmr_t *timer)`

The `TimerInit` function makes sure the respective module works and sets it to the proper mode with the `XTC_AUTO_RELOAD_OPTION` flag. The flag means the timer continues running after overflowing. The `TimerStart` and `TimerStop` functions realize what their names imply. The start function updates the last value to the current state after starting the timer and returns it.

The `TimerGetValue` function returns the current content of the timer. That makes it useful during a multistep measuring procedure or a cycle. With that in mind, the programmer must ensure that running time is properly calculated when the timer overflows. There might prove helpful the `TIMER_MAX_VALUE` constant. The `TimerGetDifference` is somewhat more complex, which means it has longer execution time, and therefore the function is more fitting for single event measurement or the run time of the observed event is comparatively longer than of the function. It returns the amount of time passed since the last call of either `TimerStart` or itself with the overflow adjustment. That means it rewrites the last measured timer value (in opposite to the `TimerGetValue` behavior).

3.2.5 Final Program

Now, with the explanation for all the libraries done, it is proper to go through the final program. Technically, the program consists of a single `main` function, which is split into two parts. The first one is meant to run only once, and then the program enters an infinite `while` loop. All the library objects exist

as a global variable, except for the `stats_t` object. The reason is that they represent part of the hardware and are relevant throughout the whole program. Contrary to that, the `stats_t` object is just a variable without any hardware base. And that goes with the decision to utilize local variables only.

Conceptually, those two sequences perform as initialization and main program loop, respectively. The initialization is responsible for making sure all the circuitry is ready to use. If any of the `Init` functions return anything else then `XST_SUCCESS` the program is terminated with the `XST_FAILURE` error code. Such behavior detects a debugger only, but the program targets for a laboratory environment and, as such, does not require any form of signalization to the user directly. In case of successful hardware initialization it moves onto `stat_t` objects. The way the current implementation works counts with sixteen instances – one for every bit of the ROPUF output. This reflects the decision of how the output randomness is evaluated, that is, every bit separately. The initialization ends there, and after instancing a couple of variables, the program enters the main program loop.

The main program loop bears a relatively simple structure. There are only two condition structures one following the other. The first one (Figure 3.5) handles the ROPUF module and perform statistical tests, and the second implements the inward part of the communication protocol. The `runs` variable serves as a counter which holds the information of how many times the ROPUF should run. In the case of non-zero value and the ROPUF being done with generating the output, runs the following code. First, it reads the ROPUF output, and pushes it into a message buffer together with the `runs` variable. Then, it decreases the value of the `runs` variable and starts the ROPUF over if the value is larger then zero. If not, it disables the ring oscillators. The next step is to send the result of the current run, and if that fails, it sets the `runs` variable to zero. Last step consists of loading the output bit values into their respective `stats_t` structures. The values are obtained through `RopufReadOutput` function call (see Section 3.2.3 for the implications).

The alternative branch only runs if no other output is being generated, and bit statistics are not updated yet. The branch iteratively goes through the statistical tests for one bit per the main program loop cycle. It does so only if enough data are available, and the test results have the dirty flag set.

The second condition structure deals exclusively with the communication protocol. For that reason, only some technical details follow, and the protocol is described in the following section. There are three branches. The first one is executed when a message comes in. The second one is active when no data comes through the serial port and currently performs no function. Every other scenario falls withing the last branch, and it means some kind of error arisen when receiving the message (see Section 3.2.2). The individual message handlers are part of a switch-case structure with the message IDs as separate cases.

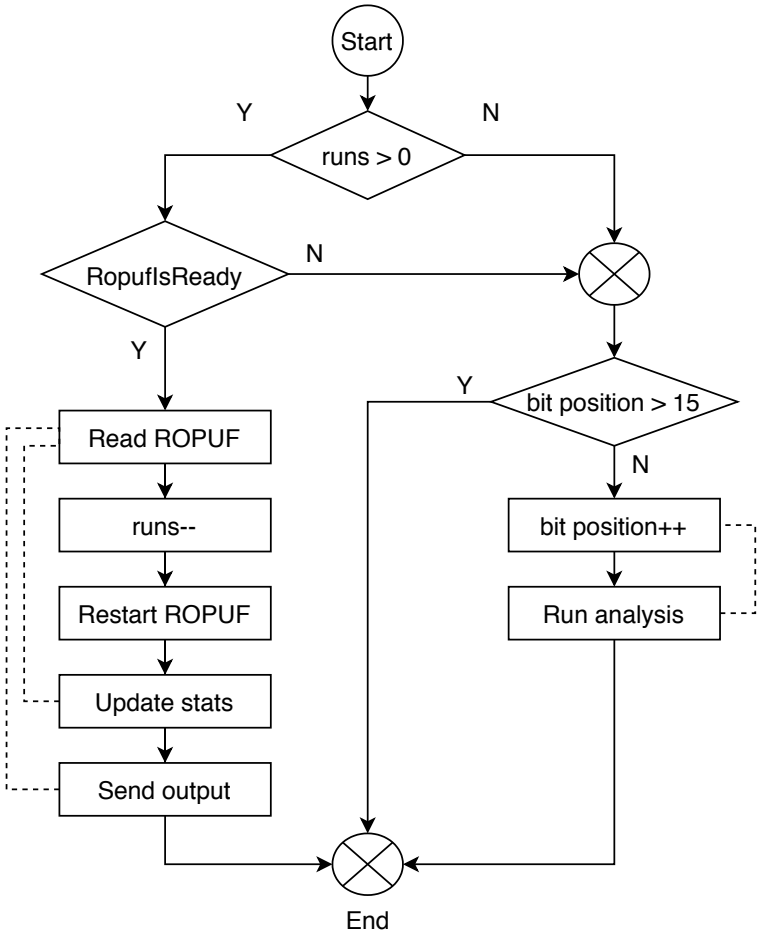

Figure 3.5: The first part of the main program loop. The dash lines show data dependency across multiple blocks.

3.2.6 Communication Protocol

The communication protocol defines six message types toward the FPGA board and seven response message types. Every message has a unique ID except for the `HELLO` message, which has the same ID for both directions (or it can be seen as the only message valid in both directions). Every message operation has a five-second timeout, which is reset by every received byte. That means it may take up to ten seconds for the `HELLO` message to come through in the case of deplorable conditions. Data longer than single-byte are transmitted with the little endianness. The FPGA board behaves as a slave device towards the computer and only response to received messages. The following part describes those challenges and how the board reacts to

them. The *Challange ID* describes the ID of the message, the *length* does not account for mandatory parts (ID, length, and CRC) and the *Response ID* lists possible responses and does not account for an error, which always yields `MESSAGE_ID_OUT_NOK` (0x99) response.

HELLO

- **Challenge ID:** `MESSAGE_ID_IN_HELLO` (0xAA)
- **Length:** 0 bytes
- **Response ID:** `MESSAGE_ID_OUT_HELLO` (0xAA)

This message type performs the function of a ping and simply respond when asked. The usage is to validate whenever the board is responding.

SET RO

- **Challenge ID:** `MESSAGE_ID_IN_SET_RO` (0xF3)
- **Length:** 2 bytes
- **Response ID:** `MESSAGE_ID_OUT_OK` (0x66)

The message carries the RO pair selection as two unsigned byte variables. The first byte makes the selection from the `RO_SET0` and the second form the other. It always responds with the `OK` message even if the selection overflows the 150 value.

GET COUNTERS

- **Challenge ID:** `MESSAGE_ID_IN_GET_COUNTERS` (0x05)
- **Length:** 2 bytes
- **Response ID:** `MESSAGE_ID_OUT_OK` (0x66), `MESSAGE_ID_OUT_COUNTERS` (0xA0)

The behavior is slightly more complex than the previous cases. The carried data is an unsigned 16-bit integer, which defines how many times the ROPUF output should be generated. If the value is non-zero, it enables the RO pair, clears the current content of the ROPUF, starts it, and responds with the `OK` message. After that, it keeps sending the `COUNTERS` responses requested amount of times. It stops doing so if the response delivery is unsuccessful. The `COUNTERS` response has a length of six bytes and contains a run ID and both ROPUF counters value obtained by the `RopufReadCounters` (see Section 3.2.3). All of those values are unsigned 16-bit integer variables. The run ID is counted downwards (e.g., if the challenge requests 1000 outputs, the first response has ID 1000, the second 999, ...).

RESET STATS

- **Challenge ID:** `MESSAGE_ID_RESET_STATS` (0xFC)
- **Length:** 0 bytes
- **Response ID:** `MESSAGE_ID_OUT_OK` (0x66)

The internal statistics do not clear their value between `GET COUNTERS` runs. It has to be done manually through this challenge.

GET STATS

- **Challenge ID:** `MESSAGE_ID_RESET_STATS` (0x0C)
- **Length:** 0 bytes
- **Response ID:** `MESSAGE_ID_OUT_STATS` (0x30), `MESSAGE_ID_OUT_TESTING` (0x31), `MESSAGE_ID_OUT_NO_STATS` (0x32)

Because the statistics have some requirements, the response is a bit more complicated. If the statistics are ready, they are sent with the `STATS` code as two unsigned 16 bit integers. The first one contains the frequency test results, and the second the runs test results. The results are single-bit values with one indicating passed test and zero failed test. The bit position aligns with the observed ROPUF counter bit position. The second possibility is that the test results are not yet ready. In that case, the response is `TESTING` without any data. The last case scenario means not enough data are available, so the test cannot run, and the `NO_STATS` is sent.

TIMEOUT

- **Challenge ID:** `MESSAGE_ID_GET_TIMEOUT` (0x09)
- **Length:** 2 bytes
- **Response ID:** `MESSAGE_ID_OUT_OK` (0x66), `MESSAGE_ID_OUT_TIMEOUT` (0x60)

The name may be somewhat misleading, but the sole purpose is to measure how long it takes to generate ROPUF output. Passed data consists of a single unsigned 16-bit integer value, which indicates how many times the ROPUF should run. Zero is a valid choice and simply means the `DEFAULT_SAMPLE` constant defines the number of runs. The handler for this message is unique in that it clogs the main program loop until it is done. For that reason, it first replies with the `OK` message, then starts the algorithm, and then it replies with the result, which is an unsigned 32-bit integer value consisting of additive run time.

3.3 Computer Software

The computer software is not necessarily part of the thesis per se. However, it plays a vital role as a master device for the FPGA slave operation mode. It loosely follows the example of the previous section and puts together three libraries. The difference represents the choice of a programming language and the paradigm as a whole. The part running on the computer is articulated as a Python script. The libraries are three separate classes, and the final product is a script tailored to the current requirements.

3.3.1 Arduino Class

The Arduino class controls the power supply. It establishes a serial link with the Arduino board, which controls the modified power supply through a series of resistors as described before.

3.3.2 Chamber Class

The chamber class controls and observes the temperature levels. The communication runs over the IP protocol. The socket is created for each exchange separately for a simple reason that the chamber is unable to maintain the communication open properly. The chamber is in an idle state by default. The idle state must be disabled first to maintain the temperature at a specific level. The chamber reports two values. The first one is the temperature inside the chamber, and it is the value set. A sensor reports the second one. The sensor is placed on the FPGA chip.

3.3.3 FPGA Class

The FPGA class contains the communication protocol. It implements a general mechanism for transmitting and receiving messages and also the concrete message types. Because the protocol is thoroughly described in Section 3.2.6, it remains to mention the method receiving messages returns a tuple of the message ID and data as a list.

3.3.4 Main Script

The main script is divided into three parts, not counting header with a shebang and includes. The first part contains the script parameters such as working directory, used serial ports, and IP address of the chamber. The second part represents the current experiment parameters. Those are the RO pairs, observed voltage levels, and temperature levels. The last part is the code itself. It is essentially a multilayered loop that first cycles through temperature settings. Inside it changes the voltage levels, and finally, then it goes through specified ring oscillator pairs. Every pair generates `SAMPLE` amount of values.

The script generates a log file that contains timestamps of every step and also temperature levels for every RO pair set of runs. The reason the temperature is not saved for every generated value springs from the relatively long running time of the request coupled with the speed (or lack thereof) of temperature changes.

Chapter 4

Measurements

This chapter discusses NIST 800-22 statistical test suite [15] application to data generated by the ROPUF. The test results provided by the FPGA directly are not analyzed here. The reason is that the test suit provides wider range of tests already implemented and allows for multiple evaluation of single set of data. The chapter is split into sections where each deals with a different approach.

4.1 Single RO Pair

The experiment configuration accounts for a single RO pair under fixed environmental conditions with a large number of samples. The goal is to determine general behavior in the context of which bits are worth exploring further. This test uses one million ROPUF cycles, which took 13 minutes to obtain. The experiment ran under the temperature of 25 °C and standard voltage (0.998 V). Every bit position is evaluated separately, although only seven least significant bits are taken into account.

The reason is the failure of tests at the fifth bit with dramatically degrading results even further (See Table 4.1). The table shows how many bitstreams passed the respective test. Each bitstream has a length of 10000 bits, which accounts for a total of 100 results per test type. According to the NIST 800-22, the number of passed tests for a random sequence under current conditions should stay above the value 95. The first test failing that condition is *runs* test at bit position three. The next bit of position fails in every test. That means only position 0 to 3 is worth exploring further.

The second part of the experiment consider the same data set, but now with multiple bit positions concatenated into a single bitstream. The Table 4.2 shows three considered ranges. All three ranges pass the same battery of tests with nearly identical results. Marked numbers show which tests failed at the P-value uniformity test, even though they succeeded at the proportion test.

Bit position	0	1	2	3	4	5
Frequency	98	99	98	99	38	5
BlockFrequency	99	100	98	98	0	0
CumulativeSums	99	99	98	98	27	1
CumulativeSums II	99	99	98	98	30	0
Runs	99	100	100	81	0	0
LongestRun	99	98	100	97	1	0
FFT	99	97	100	99	75	45
NonOverlappingTemplate	94-100	93-100	94-100	91-100	0-100	0-100
ApproximateEntropy	98	98	100	98	0	0
Serial	98	98	99	99	0	0
Serial II	99	98	100	100	81	0

Table 4.1: Test results of six lowest bit positions under fixed enviromental conditions. Failure to pass the proportion and uniformity test is indicated by the red color.

Bit range	0-2	1-3	0-3
Frequency	98	100	100
BlockFrequency	99	98	98
CumulativeSums	99	100	100
CumulativeSums II	99	99	99
Runs	98	97	97
LongestRun	97	100	100
FFT	99	95	95
NonOverlappingTemplate	95-100	94-100	94-100
ApproximateEntropy	99	98*	98*
Serial	98	100	100
Serial II	97	100	100

Table 4.2: Test results of three concatenated bit ranges under fixed enviromental conditions. The failure to pass the uniformity test is signified by the yellow color.

The results imply that the three lowest bits indicate random behavior independently, which allows for three times higher output rate when combined into a single bitstream.

4.2 All RO Pairs

This experiment follows the previous one with a difference in that it analyses every possible permutation of RO pairs (150^2 in total). That fact puts constrain on the amount of data generated. For practical reasons, each pair provides only 10000 values. And even with those limitations, the run time accounts for 50 hours. That limits significantly which types of tests can be

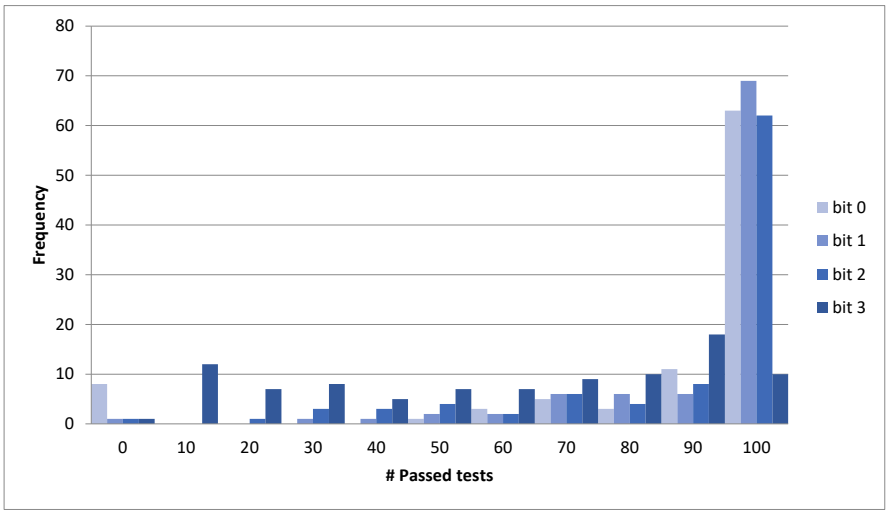

Figure 4.1: Histogram showing number of passed test per bit position of failed RO pairs.

run, although the main idea behind this experiment is to detect whenever some pair shows ultimately non-random behavior. That is deducted by two basic tests that require 100 values as a minimum. Those are frequency and runs tests. The evaluation tests bits on position 0 through 3 separately and looks for a failure of proportion test of either frequency or runs test results. The experiment fixed the environment at 24.99 °C with the chip at 38.22 °C and voltage level of 0.998 V.

Figure 4.1 shows histogram of passed test for each bit. It considers only RO pairs with majority failure. Majority failure is a failure to pass the proportion test for either frequency or runs at three or more bit positions. The highest bit shows the evenest distribution across values. The other bits display a higher proportion of passed tests, pointing in the direction of random behavior. Only the combination of lower bit position and low number (less than 60) of passed is interesting.

Five RO pairs with abysmal results are shown in Table 4.3 with the highlighted extreme of pair 141-40. It implies ring oscillator locking on each other, thus yielding non-random data. Further examination did not reveal any especially interesting anomalies.

4.3 Environmental Impact

Some constraints are applied to test the environmental impact. The first constrain is limit on the number of RO pairs to 150 randomly selected. The second is the total number of samples per pair capped at 10000. That allows for 100 bitstreams of 100-bit length. The analysis performs only frequency and runs

Bit position		0		1		2		3	
RO pair		F	R	F	R	F	R	F	R
4	59	53	53	35	35	13	13	6	5
28	51	58	58	42	42	31	30	16	15
135	101	48	48	29	29	23	22	7	4
141	40	0	0	0	0	0	0	0	0
141	50	53	53	43	42	35	35	24	23

Table 4.3: The test results of frequency (F) and runs (R) tests of failed RO pairs.

Temperature		Bit position	0		1		2		3	
Chamber	FPGA	Test type	F	R	F	R	F	R	F	R
-0.03 °C	13.37 °C	0.958 V	150	138	148	135	150	144	63	38
		0.998 V	147	139	149	138	147	141	70	39
		1.038 V	150	141	149	147	148	138	72	44
20.05 °C	32.43 °C	0.958 V	150	138	147	134	149	134	62	38
		0.998 V	149	137	147	135	147	130	65	42
		1.038 V	150	141	149	137	148	137	76	45
40.01 °C	52.26 °C	0.958 V	147	139	150	146	147	141	68	36
		0.998 V	149	139	150	133	149	140	71	44
		1.038 V	149	141	147	145	148	140	77	56

Table 4.4: The test results of frequency (F) and runs (R) tests under various environmental conditions in the selected 150 pairs.

tests for the first 4 bits separately. Each environmental condition is judged by the number of pairs failing the proportion test for each bit separately.

Table 4.4 shows all the combinations of tested environmental conditions. The table shows the number of RO pairs that passed the proportion test. The displayed temperature account for the chamber inside temperature and the temperature reported by the sensor on the FPGA chip. The generally higher proportion of failed tests is due to small sample sizes. The proper interpretation is to observe whenever significant changes take place between the condition combinations. The results do not indicate such a thing, which implies that neither the temperature nor the power supply level has any significant impact.

That is further illustrated by Figures 4.2 and 4.3. Both show variance of passed frequency tests per RO pair for three bit positions. The first graph displays the variance across voltage levels at a fixed temperature, the second just the opposite. Both graphs have highlighted the median variance values across all pairs. The most interesting is the 27-61 pair with a high variance for both evaluations, although it is not among the pairs showing a high failure rate.

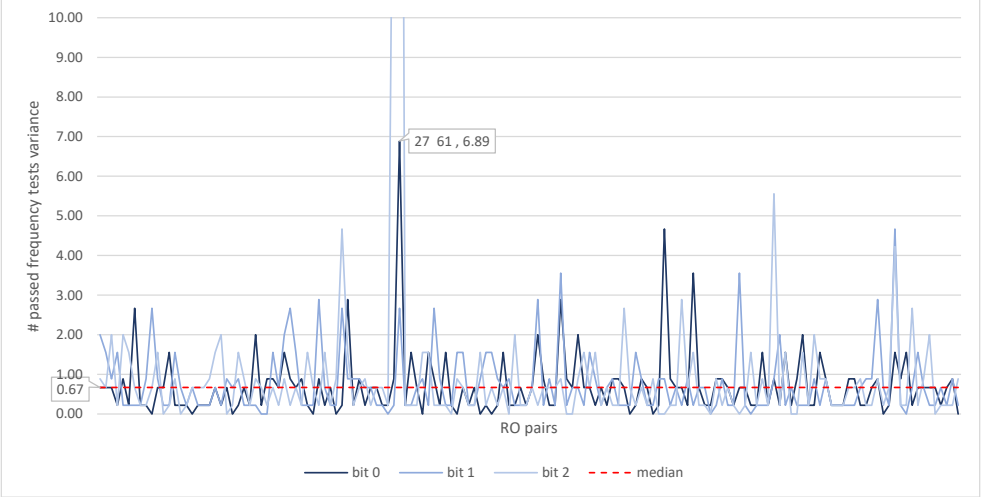

Figure 4.2: Graph showing number of passed frequency tests per RO pair variance across power supply levels.

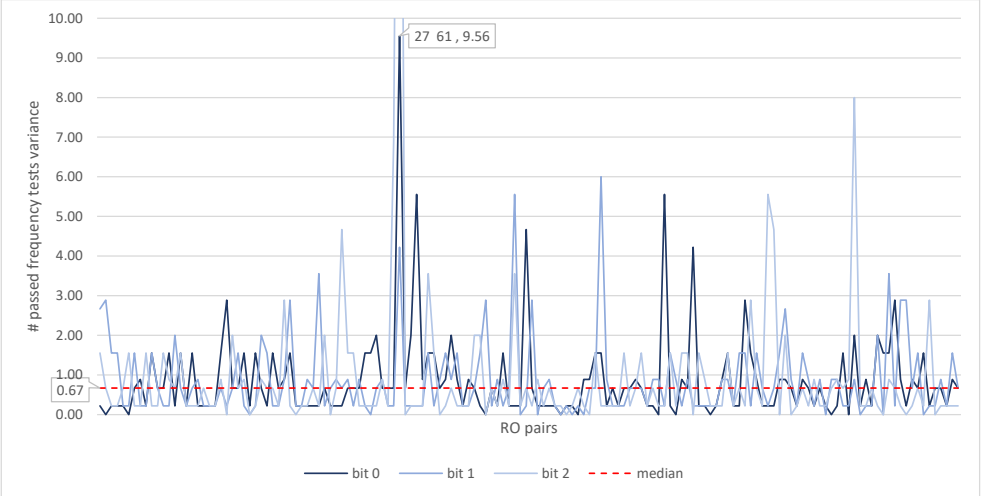

Figure 4.3: Graph showing number of passed frequency tests per RO pair variance across temperatures.

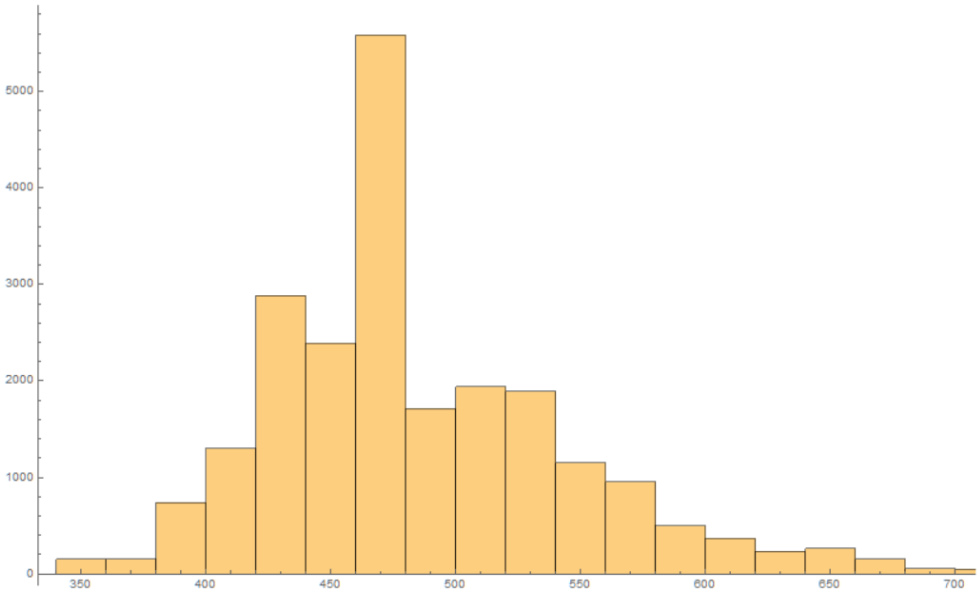

Figure 4.4: Generator cycle run time distribution across RO pairs (in µs).

4.4 Run Time

One of the metrics when it comes to random generators is the output rate. This experiment focuses on the run time of a single TRNG run cycle, which directly influences the output rate. The approach is to measure the run time of a constant number of cycles for a specific RO pair. The run time of a single cycle is deducted as the mean of those values. This procedure is repeated for multiple RO pairs. Figure 4.4 shows standard distribution with extremes at 350 and 750 µs with the most prevalent value around 470 µs.

Conclusion

The goal of this thesis is to provide reusable implementation of TRNG on FPGA board connected to the MicroBlaze processor and observe the environmental impact on the quality of the generated output.

Research showed various possibilities when it comes to obtaining randomness on FPGA. The chosen variant based on PUF circuitry proves to be a viable choice not only for its dual functionality but also for some key advantages such as generating multiple bits per single run or increased resistance to environmental changes due to ratio-based design.

The resulting implementation provides a reasonable capacity for expansion. The circuit design is simple yet reliable enough to be built on top of it. Some decisions laid the foundation for encompassing interrupts, thus handle the TRNG more effectively. Another possibility would be to provide the TRNG with AXI capability directly, which would lead to even more consistent design, but it would require some changes to the software as well. Unfortunately, those changes were outside the scope of this thesis and are left as a direction of possible improvement.

Now, considering the software running on a softcore processor only. There are some decisions worth pointing out. The first is the modulative nature of the whole program, making it (re)usable as a whole or just parts of it. Another goes to the communication protocol. It is designed in a way to provide capacity for a simple expansion without actually modifying the existing code. The general idea revolves around packets with a length parameter and CRC to ensure the safe transfer of data between the FPGA board and the computer without unnecessary overhead.

Next, only two types of tests are implemented (frequency and runs), yet it proves such capacity of the current design and reasonably straight-forward extension path. The requirement of running tests interleaved with output generation is limited to tests running when nothing else is. The reason behind this decision lies in the nature of implemented tests. Those require a complete output sequence to evaluate. Simultaneously, the tests are reasonably fast,

accounting for just a few milliseconds, compared to hundreds of milliseconds per TRNG cycle. In other words, such optimization might be unnecessary.

The analytical part primarily utilized the NIST test suit. The results suggest that neither temperature nor power supply level has any significant impact on the generated values. The main reason is probably the relative design of the TRNG, which does not rely on some constant reference and utilizes two clock sources, both under the same influence. The broader tests imply that each generator cycle can provide up to four bits from position 0 to 3, although 0 to 2 shows overall better properties. With that said, some form of post-processing might be applied to improve the statistical properties of the output, which would lead to a decrease in output rate, and thus using a wider range is preferable in such a scenario. The length of a single generator cycle varies between 350 and 750 µs.

Finally, this thesis shows the potential of ROPUF as TRNG and its reasonable stability despite varying environmental conditions. The implementation provides a simple interface to ROPUF, meaning to both PUF and TRNG functionality. The possible direction is to add start-up and tot tests and probably some form of determining which RO pairs are suitable for the TRNG functionality the most. Another viable path would be to improve the output generation in a way it would utilize the mentioned range instead of single bits separately.

Bibliography

[1] Buchovecká, S.; Lórencz, R.; et al. True Random Number Generator Based on ROPUF Circuit. In *2016 Euromicro Conference on Digital System Design (DSD)*, 2016, pp. 519–523.

[2] Hlaváč, J.; Kodýtek, F. Generátory náhodných čísel. 2017. Available from: `https://moodle-vyuka.cvut.cz/mod/resource/view.php?id=62732`

[3] Schindler, W.; Killmann, W. Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, edited by B. S. Kaliski; ç. K. Koç; C. Paar, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ISBN 978-3-540-36400-9, pp. 431–449.

[4] Brian, A.; Coughlin, S. P.; et al. Evaluation of RNGs Using FPGAs, May 2004. *Worcester Polytechnic Institute, Major Qualifying Project Report.*

[5] Danger, J.; Guilley, S.; et al. Fast True Random Generator in FPGAs. In *2007 IEEE Northeast Workshop on Circuits and Systems*, 2007, pp. 506–509.

[6] Sunar, B.; Martin, W. J.; et al. A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks. *IEEE Transactions on Computers*, volume 56, no. 1, 2007: pp. 109–119.

[7] Schellekens, D.; Preneel, B.; et al. FPGA Vendor Agnostic True Random Number Generator. In *2006 International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.

[8] Wold, K.; Tan, C. H. Analysis and Enhancement of Random Number Generator in FPGA Based on Oscillator Rings. In *2008 International Conference on Reconfigurable Computing and FPGAs*, 2008, pp. 385–390.

[9] Fischer, V.; Drutarovskỳ, M. True random number generator embedded in reconfigurable hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2002, pp. 415–430.

[10] Petura, O.; Mureddu, U.; et al. Optimization of the PLL based TRNG design using the genetic algorithm. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[11] Aseeri, M. A.; Sobhy, M. I.; et al. Lorenz chaotic model using Filed Programmable Gate Array (FPGA). In *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, volume 1, 2002, pp. I–527.

[12] Koyuncu, I.; Özcerit, A. T. The design and realization of a new high speed FPGA-based chaotic true random number generator. *Computers & Electrical Engineering*, volume 58, 2017: pp. 203–214.

[13] Kodỳtek, F.; Lórencz, R. A design of ring oscillator based PUF on FPGA. In *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, IEEE, 2015, pp. 37–42.

[14] Kodỳtek, F.; Lórencz, R.; et al. Improved ring oscillator PUF on FPGA and its properties. *Microprocessors and Microsystems*, volume 47, 2016: pp. 55–63.

[15] Bassham III, L. E.; Rukhin, A. L.; et al. *Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications*. National Institute of Standards & Technology, 2010.

[16] BINDER. Data Sheet Model MK 56. 2020. Available from: `https://www.binder-world.com/en/content/download/122303/4169888/file/Data%20Sheet%20Model%20MK%20056%20en.pdf`

[17] Kodýtek, F.; Lórencz, R.; et al. Temperature Dependence of ROPUF on FPGA. In *2016 Euromicro Conference on Digital System Design (DSD)*, 2016, pp. 698–702.

[18] Xilinx. Clocking Wizard v6.0, LogiCORE IP Product Guide. 2020. Available from: `https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf`

[19] Xilinx. Processor System Reset Module v5.0, LogiCORE IP Product Guide. 2015. Available from: `https://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf`

[20] Xilinx. LMB BRAM Interface Controller v4.0, LogiCORE IP Product Guide. 2018. Available from: https://www.xilinx.com/support/documentation/ip_documentation/lmb_bram_if_cntlr/v4_0/pg112-lmb-bram-if-cntlr.pdf

[21] Xilinx. AXI Interrupt Controller (INTC) v4.1, LogiCORE IP Product Guide. 2019. Available from: https://www.xilinx.com/support/documentation/ip_documentation/axi_intc/v4_1/pg099-axi-intc.pdf

[22] Xilinx. AXI Interconnect v2.1, LogiCORE IP Product Guide. 2017. Available from: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[23] Xilinx. AXI UART Lite v2.0, LogiCORE IP Product Guide. 2017. Available from: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf

[24] Xilinx. AXI Timer v2.0, LogiCORE IP Product Guide. 2016. Available from: https://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf

Appendix A

Acronyms

CRC Cyclic redundancy check

DFF D-type flip flop

FIFO First in first out

FPGA Field-programmable gate array

GPIO general-purpose input/output

IP Intellectual Property

LMB Local Memory Bus

LSB Least Significant Bit

MDM Microblaze Debug Module

MSB Most Significant Bit

RNG Random number generator

PLL Phase-locked Loop

PRNG Pseudo-random number generator

PUF Physically Unclonable Function

RAM Random Access Memory

RO Ring oscillator

ROPUF PUF based on ROs (ROPUF)

TRNG True random number generator

UART Universal asynchronous receiver-transmitter

VCO Voltage controlled oscillator

VHDL Very High Speed Integrated Circuit Hardware Description Language

Appendix B

Contents of Enclosed Card

```
thesis.pdf ..... the thesis text in PDF format
src ..... the directory of source codes
    hardware ..... the directory with Vivado project
    harvester ..... the directory with the computer script
    software ..... the directory with Vitis project
```