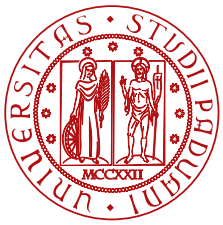

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

Analysis and FPGA-design of the Zhou-Bruck streaming algorithm for Quantum Random Number Generation

Relatore
Dr. Andrea Stanco

Laureanda
Sophia Montini

Correlatore
Dr. Davide Giacomo Marangon,
Dr. Matías Rubén Bolaños Wagner

ANNO ACCADEMICO 2023-2024

Data di laurea 19/03/2024

A te che hai avuto il coraggio di arrivare fino a questo punto.
A modo mio, ti considero importante.
Se deciderai di proseguire,
ti auguro una buona lettura.

Abstract

The central topic of this thesis is the design and implementation of a Quantum Random Number Generator. The device is composed by an entropy source, provided by a distribution in time of consecutive single-photon detections, and a real-time elaboration stage performed on a programmable development board which includes a Field-Programmable Gate Array (FPGA) chip. The elaboration process is based on a specific un-biasing algorithm and this thesis also focuses on a detailed analysis of its behaviour reporting tests' results and related observations. Moreover, a design of the algorithm suitable for FPGA applications is developed and tested. A description of the VHDL code built for this Quantum Random Number Generator is provided. Eventually, the validity of the scheme is confirmed by an actual measurement. The accuracy of the experimental data is verified by a comparison with the algorithm's software simulations.

Contents

List of Figures

Introduction

This thesis presents the analysis, design, implementation and verification of a specific quantum random number generation technique. The generator exploits the temporal degree of freedom of single-photon detection, combined with a hardware elaboration process. As regards the first element, a commercial detector was used. Instead, the second part was realized by designing a specific real-time data processing stage to be executed on a Field-Programmable Gate Array (FPGA), an integrated circuit that can be programmed through dedicated Hardware Description Languages (HDLs). The elaboration stage implements an algorithm presented in 2012 by Hongchao Zhou and Jehoshua Bruck, here referred to as the "Zhou-Bruck algorithm". It aims at converting the detector's random data flow into a stream that meets the property of an ideal uniform distribution of bits. This bit-stream is the random output of the Quantum Random Number Generator (QRNG).

A VHDL (Very high speed integrated circuit HDL) version of the algorithm suitable for FPGA applications was designed, simulated and tested. The fundamental blocks of VHDL projects are the so-called "design entities". Each entity can outline a basic element of a larger project or can be used to connect other entities that are treated as independent components. Inside each entity, an architecture describes the actual implementation of the function that is performed. The Zhou-Bruck algorithm was divided into three different basic modules that were implemented as independent entities. An external envelope was used to properly connect these components. The software tool used to develop the code and program the board is the Vivado IDE. Furthermore, a real measurement system was implemented employing the Avnet Zedboard development board as FPGA device. The development process was supported at all stages by multiple Matlab simulations conducted to analyse the algorithm's performances and accuracy. First, simulations were used to understand and test important features of the algorithm to ensure its consistency with the application requirements. Also, Matlab was exploited to verify the accuracy of the VHDL design, both through Vivado simulations and comparison with experimental data. Results of all these tests are presented. Moreover, observations and suggestions useful to optimize the proposed design are reported.

The thesis is organized as follows. Chapter 1 provides an overview of different techniques

commonly used to generate random numbers and shows how quantum mechanics is affecting this research branch. Chapter 2 presents the random number generation techniques that exploit single-photon detection and its temporal degree of freedom. Chapter 3 outlines some properties of the Zhou-Bruck algorithm that have been analysed through to the Matlab simulations. In Chapter 4, the FPGA-design is explained and commented. Chapter 5 shows the outcomes of software simulations of the VHDL code and mentions important considerations derived from Vivado implementation. Also, it describes the real experiment and discusses the obtained results. Eventually, the Conclusion summarises achievements and drawbacks of the implemented design and proposes possible improvements.

Chapter 1

Quantum Random Number Generation

Nowadays, exchange of information or data happens on a daily basis and each interaction is potentially vulnerable to external attacks, if not properly protected. Building an increased level of security is getting more and more important every year.

Current cryptography schemes rely widely on Random Number Generators (RNGs) to provide us with tools we use every day, such as One-Time Passwords (OTPs) we receive on our phone to allow, for example, electronic payments or CAPTCHAs strings used to log in at some website. All these sequences of symbols are expected to be randomly generated.

Various strategies exists to produce random numbers and they can reach different levels of robustness or generation rate. These elements influence the research for new technologies in this field. This chapter presents a brief description of some techniques of generating random numbers to help fully comprehend the purpose of this thesis.

1.1 A survey on software and hardware strategies

The task of an RNG is to generate a sequence of numbers in a predefined range of possible values. One of the main requirements for a RNG is uniformity in a given generated sequence. This means that each possible outcome should be equally probable. In case this is verified, the RNG is considered a perfectly uniform source. Moreover, we would also like all outcomes to be independent of previous outcomes, meaning that finding a given output should not give information on any future outcome after that one. This must hold for each couple of values and thus corresponds to a complete absence of correlation between symbols. Plus, it is often required for the system to be private, mainly for applications in cryptography. These results are pursued by mean of many techniques that allow the development of a large quantity of different architectures.

1.1.1 Pseudo-Random Number Generators

Sometimes, it's convenient to build a tool that approximates the behaviour of an ideal RNG. This concept is the foundation of Pseudo-Random Number Generators (PRNGs). The idea behind PRNGs is that a deterministic algorithm can take as input a certain string and produce a longer series of values that can emulate the behaviour of an ideal RNG. Obviously, knowing the input string and the algorithm makes the random string completely predictable and thus the system potentially vulnerable. Actually, this is not a problem for applications where privacy is not an essential element, such as for scientific simulations or weather forecasting. However, it is good practice for the initial string, referred to as "seed", to be randomly determined. An example of possible seed can be a sequences of uniform distribution that is used to generate other distributions. A more advanced approach can exploit some user input signal from en external interface, such as mouse clicks or key strokes [1].

For what concerns the choice of the algorithm, number theory is generally exploited. The simplest example are Linear Congruential Generators (LCGs) [2]. Their easy form is based on three parameters: a multiplier (a), an increment (b) and the modulus (m). Random numbers are provided in sequence starting from the seed, denoted with X_0, which is the first term. At every step the next number of the sequence is evaluated from the previous one according to the relation

$$X_{n+1} = aX_n + b \quad mod \;\; m \, . \tag{1.1}$$

It can be proved that the quality of a LCG depends on the choice of its parameters.

In general, since PRNG algorithms are basically sequences of software instructions, these types of RNGs are extremely fast and easy to implement. This is sufficient for many applications. However, it is not difficult to see that the sequence is not truly random and an eavesdropper could be able to recover the entire chain by only observing a small amount of subsequent values. This is the reason why many PRNGs, including LCGs, are not considered secure enough for cryptographic applications.

It's still possible to build PRNGs that assure what is called forward and backward security. This requirement is satisfied if there is no way that the knowledge of part of the sequence can lead to guessing future or past values with a probability greater than the one achieved by random guessing. This statement means that, equivalently, if sequence numbers are in the range $[0, n-1]$, it must not be possible to predict an output value not belonging to the known part with probability better than $1/n$. In case this property is verified, the structure belongs to the class of Cryptographically Secure Pseudo Random Number Generators (CSPRNGs). A simple and widely known CSPRNG is the Blum-Blum-Shub generator [3]. It is described by the following equation:

$$X_{i+1} = X_i^2 \quad mod \;\; n \, , \tag{1.2}$$

where $n = pq$ is product of two big prime numbers and X_0 is, again, the random seed. At every step i, the random sequence is given by the parity function of X_i, namely that the output value is '1' if X_i is odd or '0' if it's even.

Although PRNGs are fast and sometimes suitable for cryptographic protocols, the output of these algorithms still remains deterministic and signs of correlation might be found by randomness tests. Furthermore, an attacker that is able to obtain the seed and the algorithm can predict the whole sequence compromising the security of the system. For these reasons, other approaches exist and are able to reach higher quality results since they are based on intrinsically chaotic or unpredictable physical processes.

1.1.2 True Random Number Generators

The foundation of True Random Number Generators (TRNGs), or Hardware Random Number Generators (HRNGs), lies in the chaotic behaviour of certain natural events. Using this phenomena as entropy sources, it is possible to measure certain types of events and extract randomness from them.

What distinguishes TRNGs is the physical system used as source. Common choices are cosmic background radiation or various types of noise, such as thermal, electric or atmospheric (that is a form of radio noise caused by natural atmospheric processes). Not only natural phenomena are used, but also, for example, user's disks access times in an operating system. Of course, the structure of a TRNG must include a specific device to measure and capture this events. Usually, this is a dedicated detector. Then, raw data are often post processed to generate the desired random sequence. A scheme representing the typical structure of a TRNG is shown in Figure 1.1. The difference between this method and PRNGs is that randomness lies in the entropy source, instead of being artificially generated.

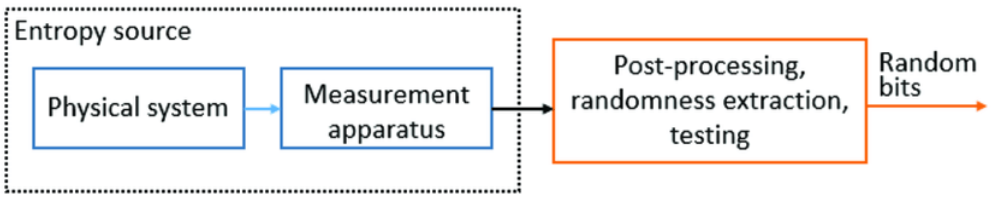

Figure 1.1: *Common structure of a TRNG*

Generally, TRNGs are more complex to implement and have limited generation rates compared to PRNGs. This second drawback is due to the physical process itself, because every time a "random event" is captured, only a limited amount of output numbers can be generated. Furthermore, dealing with physical sources of randomness brings in other issues.

For starters, in complex systems, measurable events are generally provoked by different, sometimes independent causes and it is not always easy to discern one cause from the others. For example, let's consider a case in which electronic noise is used as source. A simple implementation could be the following: the target signal is amplified and compared against a reference threshold to generate random bits. Electric noise is broadly due to two major components. The first one is shot-noise and is strictly related the quantum nature of electrons. The other one is thermal noise and is due to the ambient temperature that is able to shake electrons. In this case, the effects of one of the types of noise is indistinguishable from the other.

Another issue regarding TRNGs is the fact that a physical process might appear to be an adequate randomness source only because nowadays our models are not mature enough to fully describe it. Therefore, such sources are related to non-intrinsically random processes and thus cannot be considered truly random. A long-lasting solution should be to exploit random sources that are intrinsically random. Quantum mechanics is indeed inherently non-deterministic. Random Numbers Generators that use quantum phenomena as entropy sources are referred to as Quantum Random Number Generators (QRNGs). Instead, if the system of interest is described involving only classical theories, we are dealing with a so-called Classical True Random Number Generator (CTRNG). Anyway, note that even if a process can be described by classical laws, it might suffer from the influence of quantum effects, exactly like in the previous example about thermal noise in electronic circuits, and vice versa the same holds. Here, QRNGs are considered as devices that can be clearly modelled by quantum theory.

1.2 A possible classification for QRNGs

CTRNGs' randomness is based on our ignorance of a particular system. This makes them harder to test and also detecting signs of failure gets to be more laborious. Not identifying a malfunctioning can cause a secure system to become vulnerable to external attacks without the user knowing. Quantum Random Number Generators can overcome this problem since their behaviour can be comprehended through quantum mechanics. Keeping the generator under control becomes possible though a constant monitoring of both the system and the random string produced in output.

However, this strategy is not always adopted. Some devices simply produce an output without implementing a supervision strategy, these are referred to as trusted QRNGs. Distinctive feature of these schemes is the fact that the user cannot know whether the output is genuinely random or in control of an adversary. Sometimes this is not enough and a system to assure proper functioning and security is required. In these cases, the device must be upgraded to include a checking system that can notify when failures occur or if the output is biased. For

example, monitoring systems implemented by self-testing QRNGs could be based on a bound on the amount of correlation found between the random string and the environment or by comparison of the output distribution against the theoretical ones. Major issue regarding this type of random number generators is that the generation speed is particularly reduced with respect to trusted QRNGs. An intermediate solution is found if only part of the device is well characterized and the other part is trusted. Semi-self testing QRNGs are a compromise between the high performance of trusted generators and the reliability of self-testing devices.

A possible classification for random number generators is shown in Figure 1.2. Here, an overview will be done only for trusted QNRGs of interest for this thesis. A more complete survey on the topic, including also self-testing and semi-self testing devices, can be found in [4].

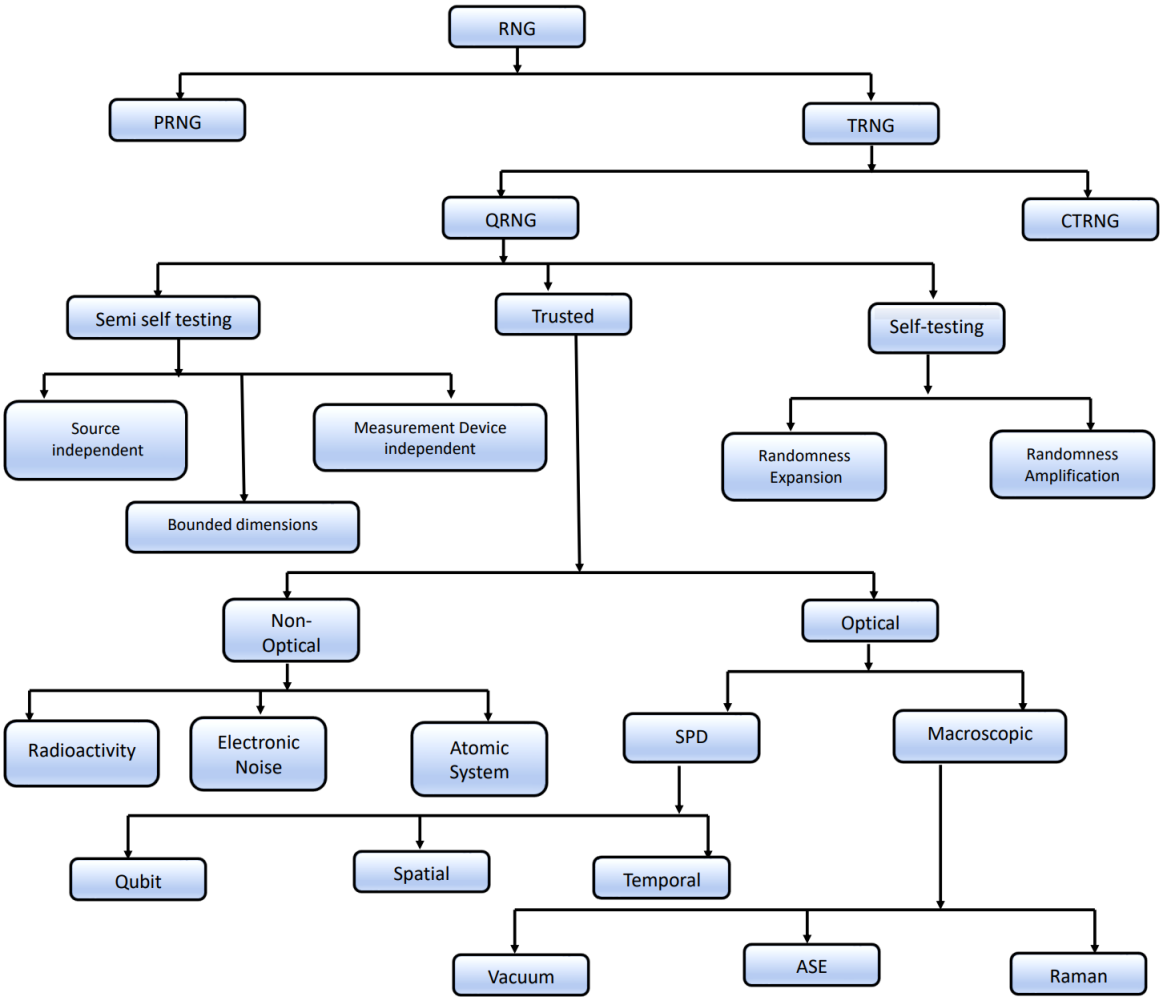

Figure 1.2: *Non-exhaustive classification of random number generators. Figure from [4]*

1.3 Trusted Quantum Random Number Generators

Trusted QRNGs are widely exploited because of the high generation rate they can achieve. Also, the theory behind their mode of operation is generally not extremely difficult since it is not required a complete theoretical model to entirely characterize the system. Plus, the implementation process is less complex than the one for self-testing generators. These properties make trusted devices suitable for many applications.

We will divide this category in two classes depending on whether optical elements are used to describe the entropy source.

1.3.1 Non-optical trusted devices

Various examples of non-optical trusted QRNGs can be taken into consideration. A common strategy is to extract randomness from radioactive decay of particles. Usually, β radiations are used, because dedicated detectors are simpler than α or γ ones. Historically, Geiger-Müller tubes serves as measurement devices as they produce voltage pulses when a particle is detected [4]. In modern applications, this system is being replaced by semiconductor devices because lower voltages are required and, even though the signal is weaker, similar results can be achieved through a simple amplification. In these systems, the probability of decay can be modelled as a Poisson distribution and the post-processing stage aims at converting it to an ideally fully uniform distribution. Depending on the relative rate of spotted particles with respect to the clock, randomness can be extracted from the number of clock pulses that pass between to subsequent detections or, vice versa, the number of spotted particles during a clock cycle. Better results in terms of output uniformity are guaranteed if the parity of each generated number is taken.

Another type of non-optical trusted quantum device is based on the measurement of electronic noise due to the shot effect. As discussed in Section 1.1.2, evaluating this process is not easy since thermal noise comes into play. However, transistors and Zener diodes in certain conditions are known to show effects predominantly due to shot noise, thus this solution is also adopted by some commercial QRNGs [4].

Other examples exploiting atomic systems have been proposed. Trapped ions and spin noise have been used to generate random numbers. However, these solutions typically have very complex setups and lower generation rates.

1.3.2 Optical trusted Quantum Generators

Thanks to the exhaustive research already carried out in this field, ease in implementation is a typical highlight of optic-based QRNGs. These systems normally exploit the quantum nature of photons to generate random numbers. Typical sources are lasers or LEDs. Randomness can be

extracted both from the microscopic properties of photons or from macroscopic quantities, such as intensity and amplitude of a light beam. Modeling these latter properties is more complicated and we will only focus on the first kind from now on. On the other hand, Single-Photon Detectors (SPDs) are the foundation of different, popular strategies for random number generation.

Qubit state detection

A photon exists as superposition of multiple possible elements, that can be, for example, polarization states or possible paths. Let's consider a light ray passing though a beam splitter. Each photon can be reflected or transmitted. Its quantum state is described as a superposition of the state in the reflected side ($|R\rangle$) and the state in the transmitted side ($|T\rangle$). $|R\rangle$ and $|T\rangle$ are basis that correctly combined represent the quantum state of the photon. Same happens in case of linearly polarized light beams. Let's consider, for example, the case of a photon polarized at a particular angle with respect to the horizontal and passed through a polarizing beam splitter. This device is build to transmit horizontally polarized light (first possible path) and reflect vertically polarized light (second possible path). If we call $|0\rangle$ the condition of absence of the photon in one path and $|1\rangle$ its presence, they represent basis vectors and the following situations would verify:

a) an horizontally polarized photon (which corresponds to a polarization angle of 0°) would have quantum state $|1\rangle_{path_1}|0\rangle_{path_2}$, because it would be transmitted to the first path;

b) a vertically polarized photon (which corresponds to a polarization angle of 90°) would have quantum state $|0\rangle_{path_1}|1\rangle_{path_2}$ for the opposite reason;

c) a photon polarized at an angle of 45° would be the superposition of the two possible states described by this expression:

$$\frac{|1\rangle_{path_1}|0\rangle_{path_2} + |0\rangle_{path_1}|1\rangle_{path_2}}{\sqrt{2}} \tag{1.3}$$

Since the outcome of the measurement at the end of one path can take two possible values and also determines the outcome at the end of the other path, each photon can produce at most one random bit. A drawback of this approach is that common detectors suffers from a dead-time, namely that after a photon is detected a non-negligible amount of time must pass before the device is ready to produce another pulse and usually bit generation rates are limited to tens of Mbps.

Temporal detection mode

As in radioactive decay-based strategies previously discussed, it is possible to produce randomness starting from arrival times of photons. The device must once again transform a Poisson distribution to a uniform one. In this case, an improvement with respect to particles decay is the larger quantity of photons available for detection that leads to higher generation rates. This strategy can be implemented as follow: being t_1 and t_2 the arrival time of two consecutive pulses, a '1' is generated if $t_2 > t_1$, otherwise the outcome is '0'. This approach can partially reduce the impact of detection dead-time. Of course, here the main limitation is given by the precision of the measured time intervals.

Spatial detection mode

The last strategy we want to mention is based on space-resolving detectors. These devices are composed by an array of sensors that can evaluate the spatial coordinates of photons. The randomness is extracted from the spatial distribution of the light's intensity. This technique can generate multiple random bits from a single detection, but it is also more likely to show correlation traces. More often, systems combining time and space information are used to generate more robust random sequences.

Chapter 2

The QRNG scheme and technology

The scope of this thesis is the design of an algorithm that is part of a single-photon based trusted QRNG. Different strategies can be exploited to implement this device. For further details, please refre to [5]. The first, simpler strategy is the direct sampling of the detector's output signal. A more complex implementation exploits an array of 256x1 single-photon detectors and a time-to-digital converter (TDC) to generate randomness with additional temporal strategies. The first solution uses a single-photon avalanche diode (SPAD) as detector, while the latter exploits a more recent device, LinoSPAD, that upgrades this technology by merging the information provided by different SPADs and also includes a time tagging feature. Both devices are connected to an FPGA so that raw data extracted by the detectors can be easily post-processed to to obtained the final random stream.

2.1 Concept behind the design

We could say that the output of a SPAD behaves similarly to a digital signal. It is low during normal working conditions and produces a rectangular pulse every time a photon is detected. The discrete sequence of bits $X = (x_1,x_2,...,x_n,...)$ obtained by sampling this signal should take the following values:

- $x_k =$'1' if a rising edge of the signal is discovered (namely if at the sampling time k the signal value equals '1' while at time $k-1$ it was equal to '0'),
- $x_k =$'0' otherwise.

The maximum content of randomness of the sequence, supposed to be hypothetically infinite, is given by the Shannon entropy as follows:

$$\frac{H(X)}{n} = -[p_0 log_2(p_0) + p_1 log_2(p_1)] \,, \tag{2.1}$$

where p_0 is the probability of x_k being '0' and p_1 is of being '1'. Higher entropy values corresponds to probabilities of the two possible outcomes closer to 0.5.

Now, note that the maximum bit-rate r_{max} is achieved when a random bit is generated at every sampling time k. Being τ_s the period of the sampling signal, this is equivalent to the relation $r_{max} = \tau_s^{-1}$. As previously stated, the length of the time interval measured between two consecutive spotted photons can be modelled as a Poisson distribution and thus the probability of having no detections during a sampling period equals

$$P[0] = e^{-r_{phot}\tau_s} , \tag{2.2}$$

where r_{phot} is the mean photon number detected per second. Since the desired output should have maximum entropy and this is realized when $P[0] = P[1] = \frac{1}{2}$, the following equation relating r_{phot} and τ_s can be obtained:

$$r_{phot} = ln\left(\frac{2}{\tau_s}\right) . \tag{2.3}$$

This clearly shows that as the sampling rate increases, the detection rate has to become larger to keep the random stream unbiased. At the same time, the detector's dead-time puts a high bound on r_{phot}. On the contrary, if the sampling rate is deliberately increased over this limit, we expect the random stream to be biased toward the zeros. In fact, if the same setup is kept, the maximum quantity of '1s' in the stream is fixed by the number of pulses, while the quantity of '0' values that separate two consecutive '1s' depends on the sampling period. The proposed idea is to considerably raise the sampling rate and deal with a stream biased in zero with percentages over 99%. Then, the sequence can be re-balanced through un-biasing algorithms.

2.2 Single detector scheme

The design presented in [5] uses the FPGA 100-MHz system clock to sample the SPAD signal. The FPGA simply saves a logical '0' for every clock cycle whenever the signal is low and one logical '1' when a rising edge is detected. This implementation is referred to as "Randy" in [5]. A schematic representation is shown in Figure 2.1. The sequence obtained in this way is strongly biased in zero and is post-processed to obtain balance by mean of the Zhou-Bruck algorithm, as stated before.

While evaluating the SPAD behaviour, dissimilarities in the photons arrival distribution were discovered with respect to the typical exponential form of a Poisson process. Two elements contribute to this incongruity: dead-time and afterpulses. We have already mentioned the first component, while the second ones are false random pulses that arise after a real one for a limited

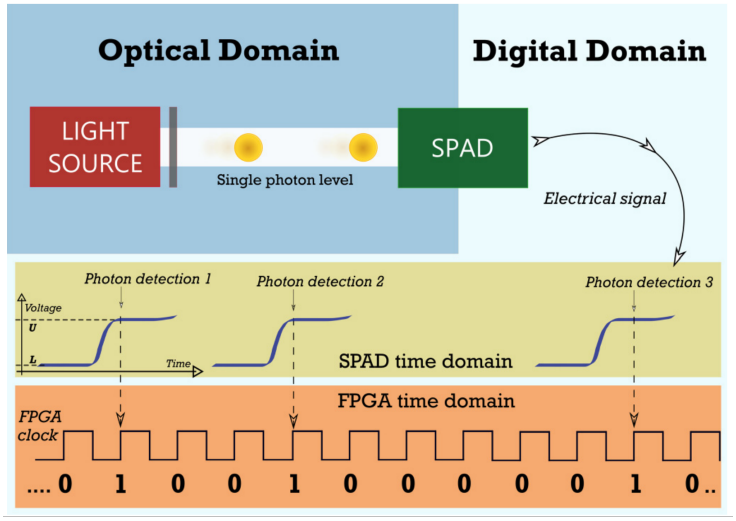

Figure 2.1: *Schematic description of Randy's working principle. Figure from [5]*

amount of time and they generate a peak of detections during this time interval. The criticality of these events is due to the correlation they can introduce. Since the post-processing algorithm only compensates the bias and cannot remove correlation, it is better to exclude pulses within this time interval from the detections used to generate the random stream. The time required to be sure enough a new detection is going to be genuine was estimated to be around 180 ns, thus, when needed, up to 18 bits after an event can be removed from the valid data.

Results expected in terms of generation rates are as high as 1.8 Mbit/s, with light intensity fixed so that the photon count rate is as close as possible to 200 kcount/s. A comparison with results generated by a traditional temporal strategy can be found in [5].

2.3 LinoSPAD variant

The previous approach is valid, but generation rates can be pushed further by introducing LinoSPAD in the system. This complex sensor uses a matrix of multiple CMOS SPADs that produce independent signals. Each pixel of the matrix is routed to the FPGA, where it is periodically sampled every 2.5 ns (which corresponds to a 400-MHz sampling frequency) and processed to produce a random stream. Also, in the FPGA every detected pulse is associated with a temporal coordinate with increased resolution by the TDC system providing an additional entropy source. This feature leads to generation rates up to the order of Gbit/s.

The LinoSPAD detectors are connected to a device that performs the un-biasing process. The algorithm is similar to the one that is used in Randy, but it is generalized to deal with non-binary digits, even though, as a consequence, it cannot run in real-time applications.

In what follows, we will focus on how randomness is extracted exploiting the time tagging

functionality because it is the main contribution to the output stream. Also, the pure sampling process is treated very similarly to the single detector case.

2.3.1 The LinoSPAD sensor

LinoSPAD has been recently introduced by the AQUA laboratory at Delft University and EPFL. It comprises an array of 256x1 SPADs, but only 64 TDCs are implemented in the FPGA so the number of pixels to be concurrently exploited is limited. However, each TDC is able to increase the resolution of each detection by a factor 140 with respect to the 400-MHz system clock. This result is achieved by emitting a code $b \in [0,139]$ to precisely locate the pulse's instant of arrival inside one of the 140 subdivisions of the 2.5-ns clock period. The resolution achieved is 17.86 ps.

The time-to-digital converter's working principle

A common time-to-digital converter's fundamental idea exploits a delay chain of blocks with known propagation time to collocate events inside a defined time interval. Each LinoSPAD's TDC is made of the series of 35 4-bit full adders (FAs), thus the 140 partition of the clock period. Every time an event reaches the input of one block, the single adder toggles its output bit. One time per clock cycle, the complete binary array is sampled to obtain the code b.

Issue related to the LinoSPAD's TDC can come from the fact that FAs are not optimized for this application, so their propagation time is not linear. This leads to a non-uniform distribution of probabilities associated with the different codes. Actually, in a typical output some codes are completely missing. However, the string correlation was proven to fall within confidence limits [6]. This assures that random numbers can be produced provided that a strategy to correct the bias is implemented.

2.4 The FPGA stage

The real-time elaboration process is performed on an FPGA. These devices were introduced in the late '80s and became widespread in the following years. They exploit the advantages of digital electronics and are very well suited for synchronous applications. In the early days of digital devices development, the interest in digital systems was widely supported by the widespread use of Boolean formalism to face logic problems and the extensive knowledge of binary arithmetic. In the 1970s, Transistor-Transistor Logic (TTL) family get ahead as promising technology[7]. These devices could discern between the binary logical values by mean of thresholds chosen to allow the recognition of the correct state even in presence of a certain amount of electronic noise.

Furthermore, this technology led to the development of gate arrays. These chips comprised a large number of NAND gates that the designer could interconnect to generate logic functions.

The major problem of TTL chips and gate arrays was that the programming phase was to be done before production. The first really programmable devices were presented in the '80s and they were Programmable Logic Arrays (PLAs). They allowed to interconnect signals though a two-level AND-OR structure whose links were user-programmable. Today, their technology is the foundation of Programmable Logic Devices (PLDs), that basically interconnect multiple PLAs. On the contrary, FPGAs generally comprise a larger quantity of simpler blocks that can be arranged as needed thanks to a considerable amount of interconnect logic.

2.4.1 The FPGA architecture

The architecture of an FPGA typically consists of programmable interconnections, clock circuitry, configurable logic and I/O blocks [8]. Also, ALUs and a RAM may be available. Examples of configurable logic blocks are multiplexers, encoders and decoders. These are basic operations that the FPGA can compute. Different blocks are interconnected by one or several programmable switch matrices. Sometimes, many logic blocks drive and read the same bus by mean of three-state buffers. Furthermore, these blocks can be connected to I/O interfaces. Usually, Flip-Flops (FFs) are placed on outputs so that clocked signals are routed to the output pins without significant delay. The same applies for inputs, since this allows to contain the device hold time requirement [8]. Also, an FPGA generally includes clock dedicated lines with low impedance so that a clock fast propagation time can be assured. Typically, Delay Locked Loops (DLLs) and Phase Locked Loops (PLLs) are also provided to support the clock. Thanks to these features, the FPGA turns out to be very suitable for synchronized architectures, like the QRNG elaboration process.

2.4.2 The Zhou-Bruck algorithm

The purpose of the algorithm developed by Hongchao Zhou and Jehoshua Bruck in 2012 [9] is to rethink von Neumann's approach [10] to the problem of generating random bits from a biased source and upgrade Peres' [11] solution to provide a system suitable for streaming applications. Their proposal aims at meeting the following features:

- random numbers are generated whenever possible, thus it can be used in real-time devices;
- algorithm complexity is expected to be linear with the input length;
- hardware resources required to store data are bounded;

- it reaches asymptotically optimal efficiency, namely that as the number of random bit increases, the generation rate gets closer to the entropy of the source.

These properties make the algorithm appropriate for practical, real-time implementations, such as the application described in this chapter.

The algorithm's workflow

The Zhou-Bruck algorithm takes in input values generated by tossing an hypothetically biased coin, modifies its internal state accordingly and, whenever a specific sequence is detected, produces a random bit that is transmitted to the output stream. There are two possible values as input: head and tail ($\{H,T\}$), the result of the coin toss. In this way, they can be easily distinguished from the binary output digits $\{0,1\}$.

The data structure used to store the current internal state is a binary tree. At the beginning, the tree is empty so that it comprises only one node, the root, whose state is "void" $\{\phi\}$. The root is fed with new inputs and data are propagated through the nodes until an empty one is found, a random value is resolved or the last layer of the tree is reached. The tree's depth determines the amount of hardware resources that need to be allocated.

Each node, including the root, behaves in the same way and toggles between five status: $x \in \{\phi,H,T,0,1\}$. Let's consider a node u having status x, left child u_l and right child u_r. Being $y \in \{H,T\}$ the input to u, the following rule set holds [9]:

1. if $x = \phi$, $x = y$ is set;
2. if $x = 1$ or 0, x is sent to the output and $x = y$ is set;
3. if $x = H$ or T, then four cases show up:
    - when $xy = HH$, $x = \phi$ is set, then a symbol T is passed to u_l and a symbol H to u_r,
    - when $xy = TT$, $x = \phi$ is set, then a symbol T is passed to u_l and a symbol T to u_r,
    - when $xy = HT$, $x = 1$ is set and a symbol H is passed to u_l,
    - when $xy = TH$, $x = 0$ is set and a symbol H is passed to u_l.

The binary version of the algorithm is enough for the scope of this thesis, but it is possible to generalize the procedure to an m-sided dice (instead of a coin, where m = 2), as pointed out in [9]. The process works as follow. Each possible dice roll outcome is coded into its binary representation in terms of H,T, assuming H as '1' and T as '0'. Then, a binary tree, called binarization tree, is built. The first digit of the binary code is assigned to the root. If this digit is

T (or, equivalently, '0'), the procedure is repeated picking the second digit and the left subtree. If it is H (or '1'), the right subtree is taken. Same process applies to the root of the subtree and the second digit. The procedure goes on, choosing the left or the right subtree at every step depending on the digit considered until the last digit is assigned. It is clear that the binarization tree is going to have the same depth as the number of bits necessary to represent m outcomes, that is $log_2(m)$. Now, each node of the binarization tree is a sequence of biased coin tosses and can be fed as input to a binary tree of the first, simpler type. At the end, randoms streams generated by each independent tree needs to be re-combined to form the final random output.

Chapter 3

Matlab simulation and analysis of the algorithm

Before actually implementing the algorithm on the FPGA, a Matlab simulation to verify some of its properties was developed. As stated in Section 2.4.2, the binary version was analysed and implemented, so that both the Matlab simulation and the FPGA design only comprise one biased, input source, one tree and one output random stream, that is expected to be balanced.

The simulation exploits the Matlab `rand()` function to generate uncorrelated numbers within the interval [0,1]. An increment b is added to the raw data before rounding to the closest integer to give us direct control of the input stream bias. Strictly speaking, b is the desired percentage of '1s' in the input string. Also, a variable N allows to set the string length, namely the number of input data processed by the algorithm.

Another parameter that can be controlled is the tree height. The maximum depth that the tree can reach depends on the input string length, but it also determines the space that needs to be allocated for its implementation. It's possible to simulate an ideal tree that grows dynamically without bound[1] as the input comes along, or set a limit to the maximum depth of the tree. In this second case, every value that should be passed to the first canceled layer is simply discarded.

This single-tree simulation was then improved to have more statistically relevant data. A program that iterates the input string generation process and the tree formation was developed. Thus, each tree is independent and each random outcome is stored individually to evaluate average properties. The number of iterations can be set and, if wanted, this process can be repeated automatically changing the value of one parameter, for example the input string length or its balance.

[1] The ideal tree actually has a bound, that is given by the computer's memory space.

3.1 Node's generation capability

First, the nodes' generation capability was evaluated. The test aimed at giving an insight into the contribution of each node to the final output. The more random bits it produces, the more one node is relevant for the generation of the output stream. The graph in Figure 3.1 shows the typical distribution of the amount of output values generated by nodes within a tree. The specific simulation used to plot the graph was run considering an ideal tree, with $N = 5 * 10^5$ input values and bias $b = 0.002$ (which corresponds to a percentage of 99.8% of '0s' in the input string, as in the real application according to [5]). Note that node numbered 1024 is the first node of the 10th layer, while the actual depth of the tree is 18 layers.

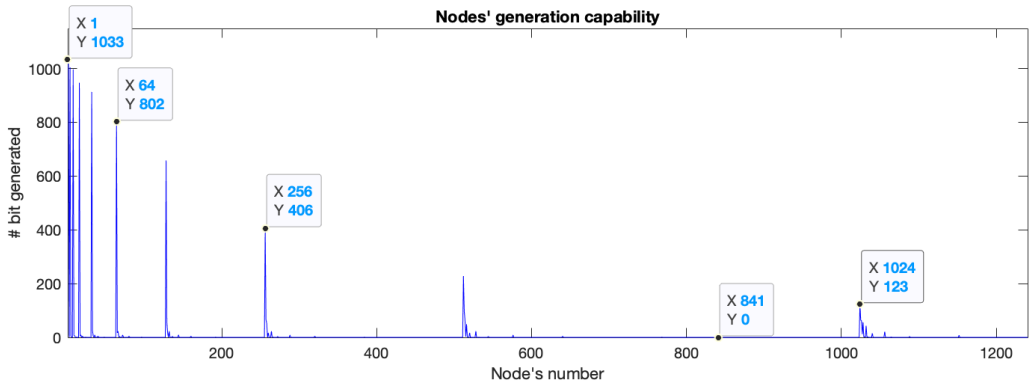

Figure 3.1: *Nodes' generation capability. Graph is truncated after the first layers of the tree*

To properly understand Figure 3.1, consider that an array structure was used to store the tree current state. The array was built in such a way that index 1 corresponds to the root, index 2 to its left child, index 3 to the right one and so on. In general, the node at index k has left child in position $2k$ and right child at index $2k + 1$.

The total number of random bits produced was 10 181. As it's possible to see from Figure 3.1, the root alone generates around 10% of the available values. The generation capability quickly decreases as lower layers are considered. Moreover, many nodes does not produce any bit, such as node 841 shown in the graph. This is the reason why cutting the tree by imposing a maximum depth does not remarkably affect the output.

3.2 Impact on the sequence's autocorrelation

Moreover, a check to verify that the algorithm does not introduce any correlation was done. First, the autocorrelation of the input string produced by the `rand()` function was evaluated to be sure that, in case correlation was found in the output, it was not due to the sequence itself. Figure 3.2 shows the result of a 100-order autocorrelation obtained considering an input stream

with the same parameters as the string used in Section 3.1 except for the length of the input sequence, that was set to N = 5 000 000. This change was done to obtain correlation coefficients more similar to the real case, while short and highly biased sequences were observe to produce results that could not be judged accurate enough. Consider that the input string has a very large bias $b_{in} = 0.0020$, while the output is successfully rebalanced by the Zhou-Bruck algorithm and $b_{out} = 0.5013$ is achieved. The red dotted lines represent the 99% correlation confidence interval. The test is considered passed, since almost every correlation coefficient of the output string lays within the confidential limits.

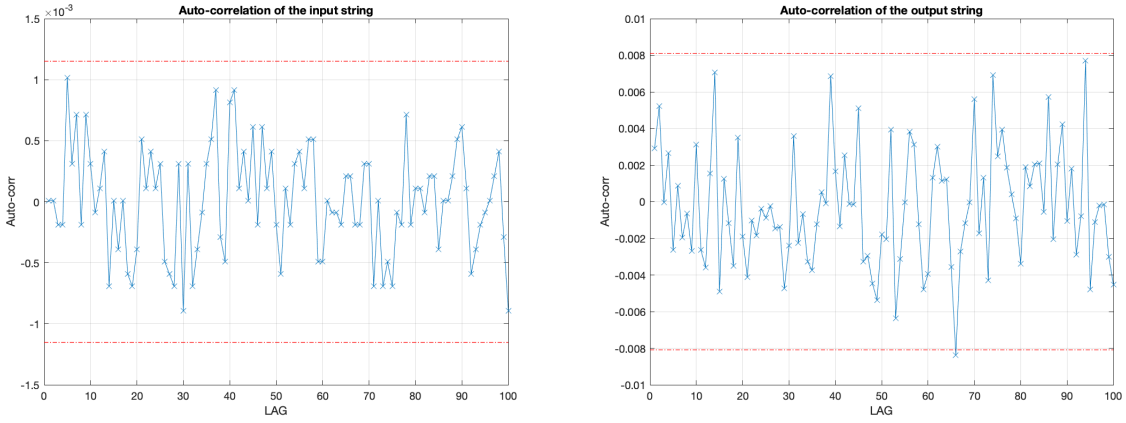

Figure 3.2: *String autocorrelation evaluated before (left) and after (right) being processed by the Zhou-Bruck algorithm*

3.3 Robustness of bias correction capability

The algorithm's performances in terms of balance achieved were analysed with respect to different input parameters. This simulation exploits the iterative code that creates independent input strings and builds the related trees. The outcome is expressed as the absolute difference between the output sequence balance and the 0.5 desired result. Outcomes that have the same input specifications are averaged to obtain more statistically relevant data. Plus, one parameter per simulation varies so that a more general behaviour of the tree is pointed out. In Figures 3.3, 3.4, 3.5 and 3.6, each blue cross was obtained by averaging 100 independent trees, while red dotted lines plot boundaries of the confidence interval at 99%.

3.3.1 Balance with respect of the input string length

First, the algorithm performances with respect to the input string length were analysed. As expected, the algorithm was noted to improve the balance of the random stream as the length of the

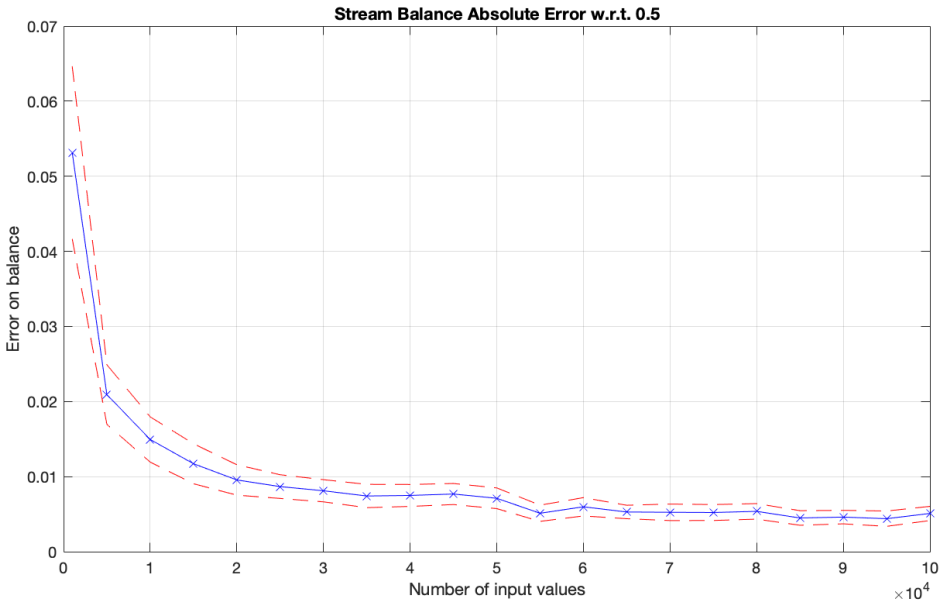

Figure 3.3: *Average absolute error of the output string bias with respect to the 0.5 desired, determined for different values of the parameter N, the length of the input sequence. The input bias is fixed at* $b = 0.01$

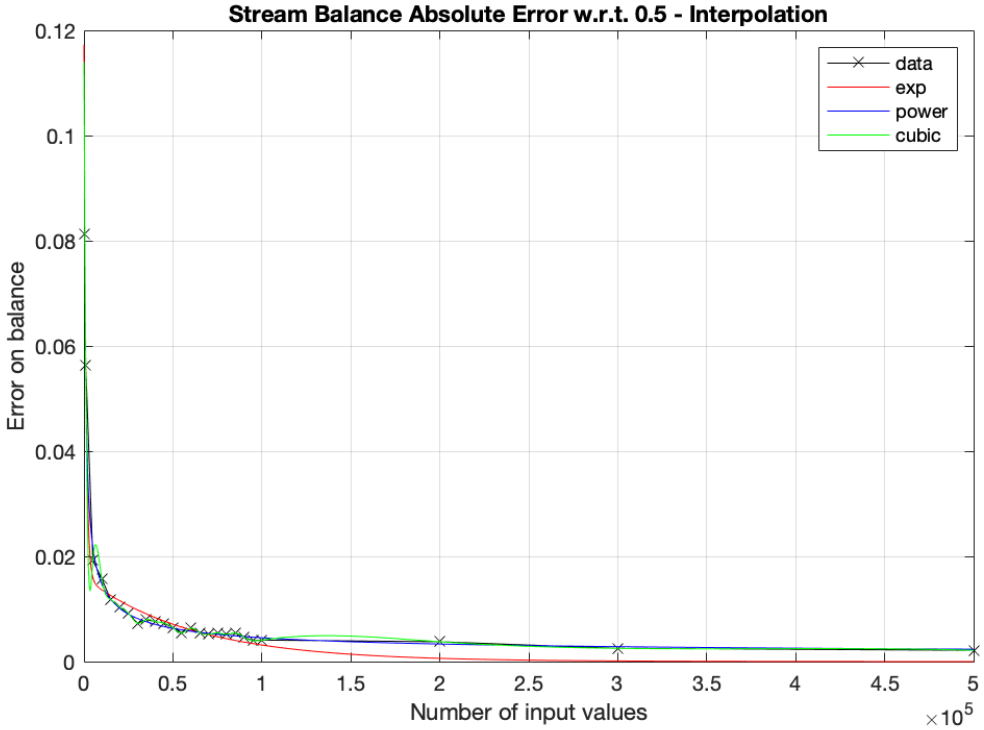

Figure 3.4: *Interpolation of experimental values in figure 3.3 evaluated on a larger set of data. The input bias is fixed at* $b = 0.01$

input sequence increased. The results are reported in Figure 3.3. For the whole simulation, the starting bias was fixed at $b = 0.01$ to make the outcome more relevant also for short input sequences and the trees' maximum height was not set to a fixed value to not alter results generated by very long sequences.

Observe that generally, with the given input bias, approximately 20 000 input values are needed to be sure enough that the percentage of '1s' in the output string is going to be between 49% and 51%, namely, that the absolute error will be below 1%. If precision below 0.5% is desired, at least roughly 60 000 input values are needed.

A longer simulation was used to choose the best interpolation method. The curve that best fits the experimental data is a power type $y = ax^b + c$, which is shown colored in blue in Figure 3.4. By changing the input bias and exploiting the new interpolated curve, the minimum number of input values required to have high probability to fulfill a requirement on the maximum balance error can be easily found.

3.3.2 Outcome obtained varying the input string bias

The robustness of the algorithm was analyzed also with respect to the bias of the input sequence. Again, blue crosses shown in Figure 3.5 are an average between 100 values, while the length of input sequence is fixed at 100 000.

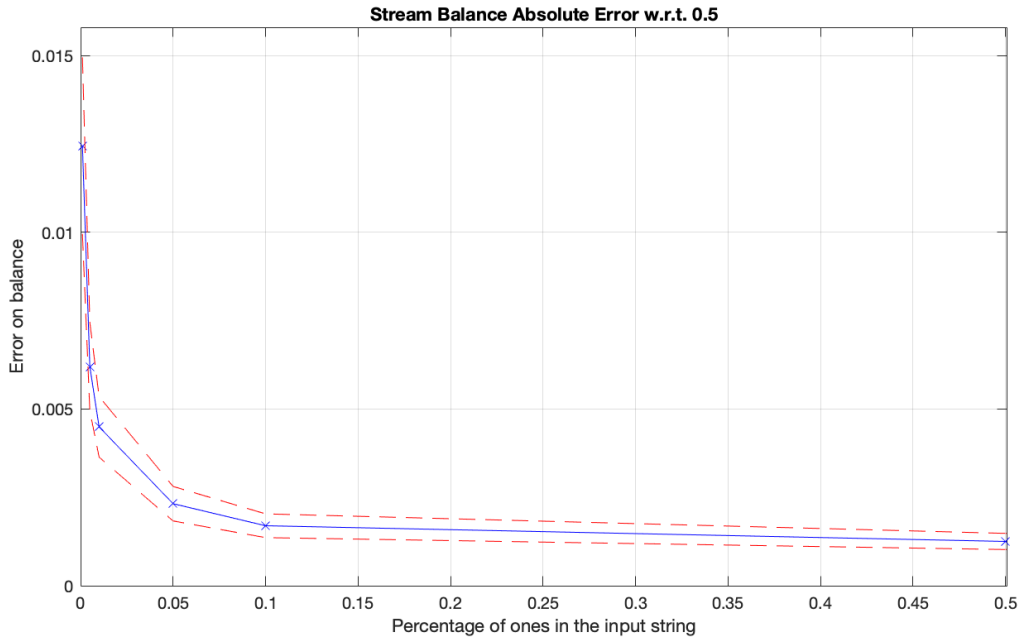

Figure 3.5: *Average absolute error of the output string bias with respect to the 0.5 desired, determined for different values of the parameter b, the bias of the input sequence. The input length is fixed at $N = 100\,000$*

Figure 3.5 serves to provide an hint of the expected outcome when parameters that determine the input bias are set, such as the light beam intensity, the detector working frequency or the FPGA sampling rate. Also, it can provide a lower bound on the starting bias if a certain precision in output is required.

3.3.3 Outcome with respect to the tree height

Lastly, a possible correlation between the tree height and the output balance was studied and results are reported in Figure 3.6. The curve shows a tendency to get closer to the ideal target, but the improvement is not regular as the tree maximum height increases. Also, note that the vertical axis has a 10^{-3} scale factor, so that visible fluctuations are actually quite limited. We concluded that the tree height has not a particularly high impact on the output stream balance.

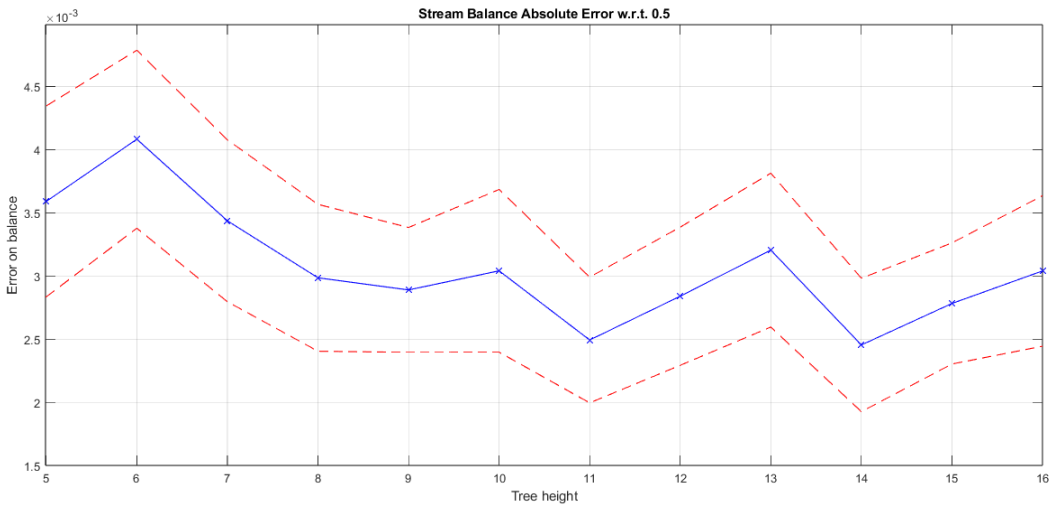

Figure 3.6: *Average absolute error of the output string bias with respect to the 0.5 desired, determined for different values of the maximum tree height. The input length is fixed at $N = 1\,000\,000$ and the input bias at $b = 0.002$*

3.4 Balance in different sub-strings of the output

An important requirement for the Zhou-Bruck algorithm is that the output stream must not show "memory effects". This can be verified by excluding the presence of patterns or, on the opposite, a constant improvement on the distribution of '1s' and '0s'. The random sequence generated by the simulation used in Section 3.1 and Section 3.2 was divided into different segments to analyze the bias of different portions of the string.

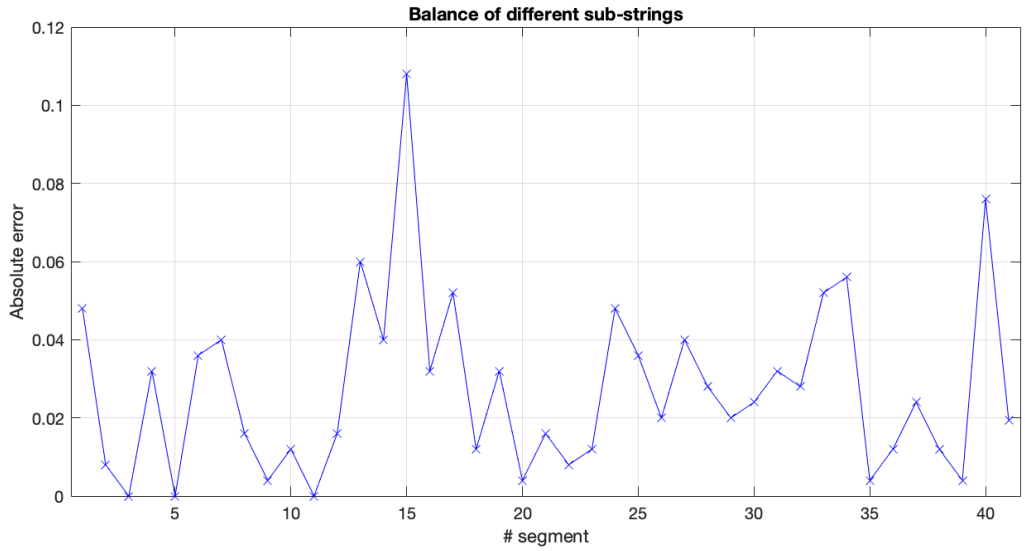

Figure 3.7: *Balance absolute error of different segments of the output stream w.r.t. the 0.5 balance desired. Simulation data are the same as in Figure 3.1*

Being the total output length 10 181, each sub-string in Figure 3.7 comprises 250 consecutive values. From direct inspection of the figure, it's clear that no memory property is present.

3.5 Nodes' generation rates within each layer

While analyzing other properties, an interesting observation for practical implementation was made. The algorithm description explained in Section 2.4.2 implies that, whenever point 3 is verified, node u always passes a value to its left child, while the right child receives a new value only if $x = y$. Moreover, u_r produces a value for the output only if it is given two consecutive different inputs, but an H is passed to it only if both x and y equals H. This condition is equivalent to receiving two consecutive '1s' and is very unlikely to happen in a string extremely biased toward the '0s'. This phenomenon results in a predictable behaviour of the nodes' generation capabilities distribution within each layer of the tree. Results for some representative layers of the single-tree simulation are shown in Figure 3.8.

Of course, layer n. 1 includes only the root, thus it is not very informative. Instead, all other layers show huge generation rates of the leftmost nodes, while most of the others are completely unproductive. The huge difference is clear even from the second layers. Layer n. 2 contains the root's children. The root's left child generates 1 032 random bits, while the right one only 1. The deeper the layer is, the more emphasized this behaviour is. Let's consider for example layer n. 4 (composed by 8 nodes). The leftmost one produces 997 useful values, while even its closest sibling generates only 9 random numbers. This characteristic is due to the fact that more

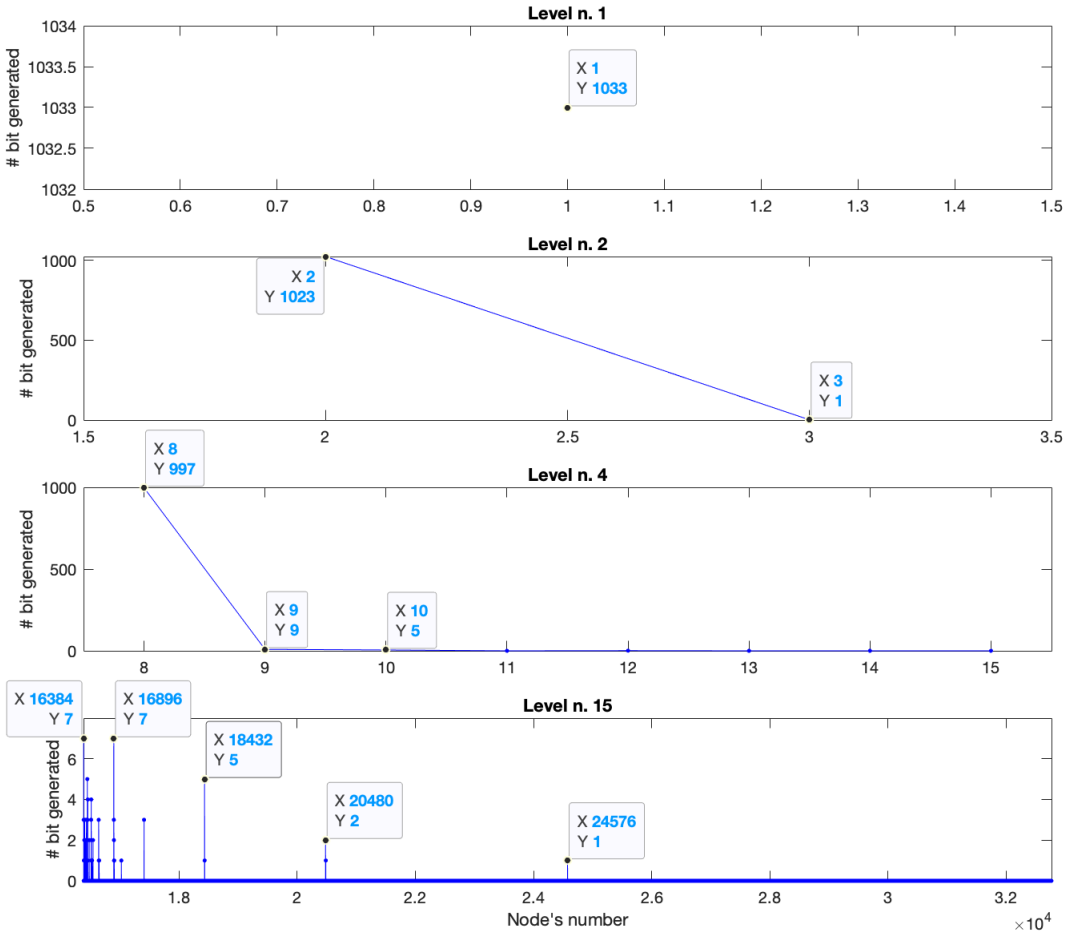

Figure 3.8: *Single node's generation capability shown represented divided per tree layers. Only four representative layers are shown. Simulation data are the same as in Figure 3.1*

useful values are generally transmitted to the left child.

Actually, a more precise pattern can be found analyzing deeper layers, where the difference in terms of bits generation is not particularly marked but a repetitive scheme among the different input sequences was observed. Let's refer to level n. 15. It comprises 16 384 nodes, labeled from 16 384 to 32 767. One random bit is generated by the node numbered 24 576, which is positioned exactly at the half of the layer. Let's denote it as u_H. Note that u_H is the leftmost node of the subtree originated by the right child of the root (node labeled as n. 3). All and only the nodes belonging to the same subtree at level n. 15 are positioned to the right of u_H. This implies that u_H is the only node at this level originated by the root's right child that produces values for the random output.

As regards the root's left subtree (that is tree originated by node n. 2, the left child of the

root), more nodes are able to produce random bits but the same behaviour is repeated. As a matter of fact, if subtree originated by different nodes are considered, the most efficient nodes inside each layer are the leftmost descendants. For example, node labeled as 20 480 produces 2 random bits and it is the leftmost node of the subtree originated by node numbered as 5, thus the right child of node n. 2. Likewise, node n. 18 432 generates 5 bits and is the leftmost child of node n. 9. The sequence can continue considering node n. 16 896, leftmost child of node n. 17, and so on.

It's clear that some paths are favoured over others. This observation can be exploited to better manage hardware resources in devices with limited equipment, such as an FPGA. However, this thesis focuses on the design of a working FPGA-based program at the first stage, without carrying out any kind of optimization.

Chapter 4

Zhou-Bruck algorithm FPGA-design

The FPGA-design aims at creating a VHDL implementation of the Zhou-Bruck algorithm in its single-tree version. VHDL is an HDL widely used to program digital electronic systems thanks to its similarities with common software programming languages, such as "if" or "switch" statements. The software interface used to develop the project is Vivado IDE. The target device is the Avnet Zedboard development board, based on the AMD Xilinx Zynq 7000 SoC. The FPGA should be connected to the output of the SPAD that provides the biased random stream. It routes the signal into different blocks that process the sequence to extract the desired output.

4.1 Architecture of the VHDL project

A schematic representation of the implemented structure is shown in Figure 4.1. Each block carries out a well defined task and is implemented as an independent entity. The "Tree" structure is the external envelope where the different components are declared and proper connections are defined.

The single input to the FPGA system is the biased sequence. On this line, a '1' is found every time a photon is detected by the SPAD. The bias of the sequence can be controlled by setting the light beam intensity, the detector's output bit-rate and the FPGA sampling frequency. At the output, the FPGA drives two signals. One line carries the actual bits of the unbiased random stream, while the other one alerts whenever a new bit is generated. The second line works as an enabler to the system that receives the two signals: when the second line is high, one random bit can be read from the first one. This arrangement is necessary since there is no way to predict when a new number will be produced. Internally, communication between different nodes is achieved using the same strategy.

Let's start discussing the central element of the structure: the node.

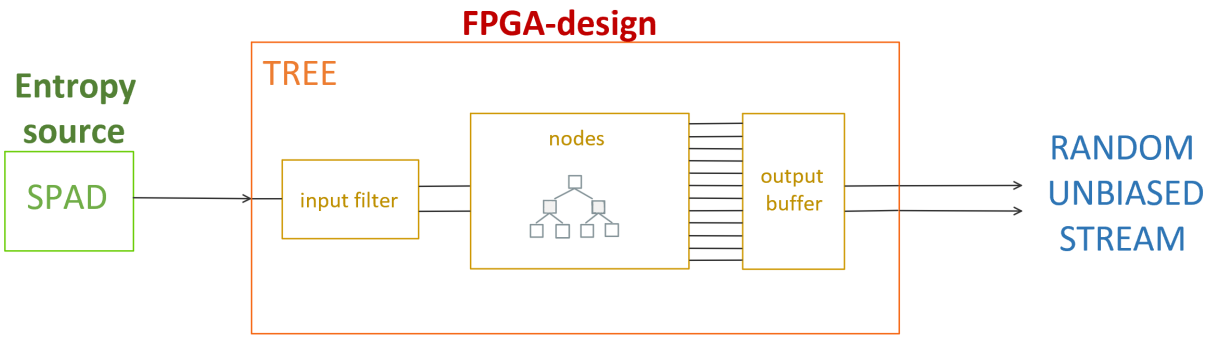

Figure 4.1: *FPGA-design structure*

4.1.1 Node's implementation

The node is the most basic component. It is entirely defined by its internal state, which is always one among $\{\phi, H, T, 1, 0\}$. A node receives in input one bit and its enabler and drives the output signals to the general buffer and to its two children nodes. Its behaviour is described by Zhou-Bruck algorithm's rules. The set of instructions described in Section 2.4.2 can be converted into the Mealy Finite State Machine (FSM) shown in Figure 4.2.

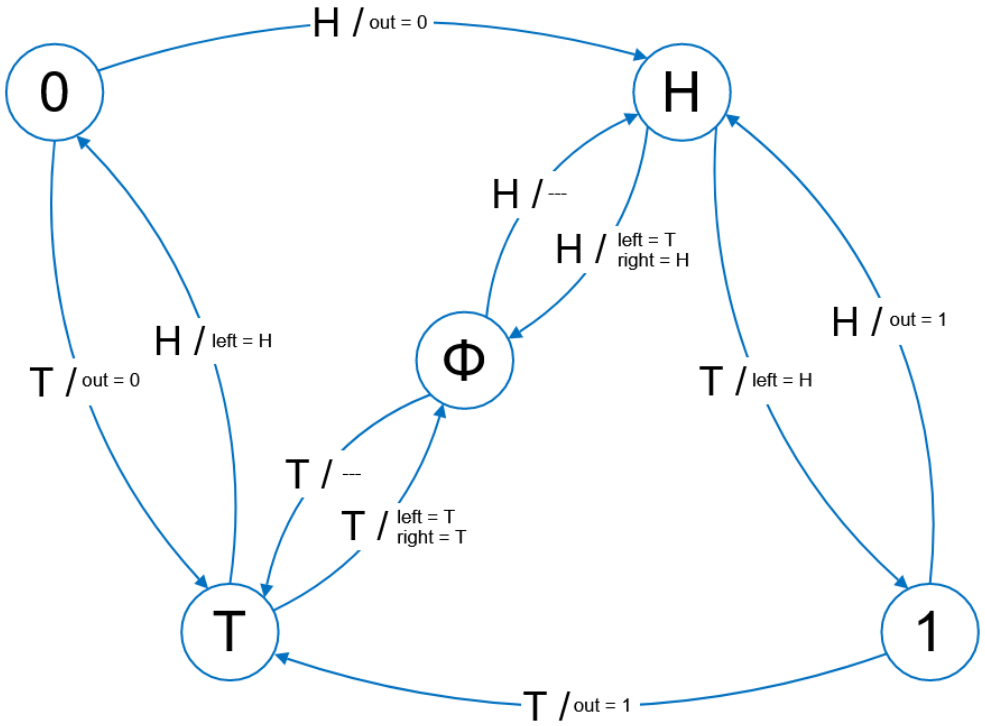

Figure 4.2: *Finite state machine (FSM) diagram that defines the behaviour of the entity "Node". The output to u_l is referred to as "left", to u_r as "right" and to the output buffer as "out"*

The correspondent VHDL structure is a simple switch case that properly assigns the new state and the outputs depending on the current state. The statement is inside a process and is activated if a rising edge of the clock is detected. Vivado implements this code using Look Up Tables (LUTs) that take one clock cycle to be updated. This implies that each node has a fixed propagation time that equals one clock period. Thus, for example, if the input at time $t = 0$ of the root (layer n. 1) is propagated through the tree, it reaches the tenth layer (that corresponds to nodes labeled from 512 to 1023 in the array notation) at time $t = 9T_{clk}$.

All nodes are directly connected to the output buffer and, as a consequence, a change in the sampling frequency with respect to the clock period can alter the order of the output bits. For example, let's consider the case of the input being propagated to the tenth layer. As previously explained, the time required for the input to reach this layer is $\Delta t = 9T_{clk}$. Let's suppose that a bit is generated by a node of the tenth layer, let's say node n. 512, and it derives from the input (to the root) at time $t = 0$. Let's assume that the following input (at $t = T_s$) will generate an output from the root itself. Then, if the sampling rate is at least ten times smaller than the clock frequency, the tree has time to completely update before the new input arrives and the bit generated by node n. 512 will precede the one produced by the root in the output sequence ($9T_{clk} < T_s$). Instead, if the system is working at maximum speed, then $f_{clk} = f_s$ and the root will send its output to the buffer before node n. 512 processes the bit from the previous step.

The propagation time must be considered while analysing the output string to certify the expected behaviour. However, in a real application the precise order of the bits is not important. We expect the properties of unpredictability and balance of the stream to be still guaranteed, even with bits in different order, otherwise this would be a sign of unwanted memory effects.

4.1.2 The output buffer

We have already mentioned the output buffer and the fact that it is directly connected to every node. Its task is to collect the output bits that are generated in parallel by all nodes and convert them into a series stream. This component is not expected to change the order of the received bits, so it can be thought of as a FIFO queue. At the output, it drives the two signals that form the unbiased random stream in Figure 4.1. In this FPGA-design, the buffer is implemented as a shift register, a widely known component in digital electronics. In this way, the only signals required are the array that stores the buffer state and an integer to know how many bits have been added to the buffer.

At every clock cycle, the presence of random bits in the buffer is checked. If bits are present, the oldest, that is the one at index 0, is sent to the output. The remaining bits are shifted toward the lowest index. The same is repeated at the following clock cycle. This behaviour satisfies the requirement for bits to be added to the random stream as soon as possible after they became

available. The shifting operation is performed only on the portion of buffer occupied by useful bits. This is necessary to avoid double assignments due to the arrival of new random bits from the nodes. Also, variables are used to make the buffer capable of accepting multiple random bits during the same clock cycle. Indeed, the signal that stores the number of bits ready for the output is updated only at the end of the loop.

Moreover, the buffer size is defined as a generic that can be set by the user. Simulations showed that if the tree has time to completely process an input before the following one becomes available, then, generally, it's very rare that more than four bits are generated at the same time. Obviously, increasing the sampling frequency compared to the clock leads to more bits being produced simultaneously. The buffer size must be chosen to minimize the risk of loosing bits and still using a reasonable quantity of FFs.

4.1.3 The input module

The input filter modifies the SPAD signal to meet the tree needs. First, it uses a rising-edge detector to isolate '1s' in the input stream so that for every detection only one *H* value is fed to the tree. Also, since we want to be able to set the sampling rate at a lower frequency than the clock, a decimator is implemented by mean of a counter. A dedicated prescale register is used the impose the ratio between the clock and the sampling frequency. If it is set to 1, the sampling rate equals the clock frequency. Once again, it is convenient to use the strategy of the two synchronized lines that carry the bit value and the enabler. It simplifies the tree structure, since the root gets to work exactly like all other nodes. Plus, this makes it easier to deal with the problem of afterpulses, as explained in Section 2.2.

As a matter of fact, the input stage also has the task of filtering afterpulses. We have already stated that the estimated required interval for the SPAD to be in a "safe-zone" is 180 ns, irrespectively of the sampling rate. After a detection, if one or more '1s' are present within the following 180-ns interval, all values included between the detection and the last '1' shall be discarded. Since we aim at working with a 100-MHz clock, 18 clock cycles must be analysed before sending the correct input to the root. The number of clock cycles is set as a constant to be easily modified in case a different clock frequency is used. With these settings, this module introduces a 180-ns delay in the input string to have time to process the SPAD signal within the required interval and to introduce "blank spaces" in the input sequence if needed. A shift register is used to store the last 18 values of the SPAD output and it is updated at every clock cycle. Whenever the sampling signal produces a pulse, the bit in the highest location of the shift register is sent to the root of the tree by rising the enabler line. This correctly introduces an 18-clock cycles delay between the moment a bit is detected by the FPGA and when the tree receives the same bit as a valid input.

In fact, two independent enable signals are used to control the root. One is the signal driven from the sampling decimator and it enables all bits that should be fed to the root according to the sampling frequency. The other one is the signal that disables the input stream to produce "blank spaces" to filter afterpulses. This last enabler can introduce pauses of variable length. To avoid using variables, every time a photon is found (which corresponds to having a '1' in the first location of the shift register), a first counter verifies if '1s' are detected within the 180-ns interval and, if so, it stores the number of bits to be discarded. After 18 clock pulses, the 1-bit due to the photon is fed to the tree and at the following clock cycle the pause is activated, if needed. While the pause is active, another counter starts from the quantity of discarded bits and is decreased to know when it is possible to reactivate the tree.

4.1.4 The "Tree" entity

The external structure is the only non-behavioural architecture of the project. Here, the input filter, the output buffer and the nodes are instantiated. Nodes are organized in an array as in the Matlab simulation. A simple "generate" statement with a "for loop" is used to realize proper connections within the tree. Wire-type signals pass bits among each node and its children or the output buffer. Obviously, signals routed towards the children are left open for nodes of the last layer of the tree.

Chapter 5

Design simulations and implementation

The design explained in Chapter 4 was tested using both Vivado behavioural simulations and the FPGA to carry out an actual measurement. The software simulations serve to verify the code behaves as expected before generating the bitstream, that is the list of instruction used to program the board. By mean of a dedicated input created by hand or exploiting a different software, simulations are able to show the evolution of the different signals in time to check for unexpected events. Moreover, another important intermediate step is the design implementation. During this phase, Vivado creates the corresponding hardware connections optimized for the target board. Eventually, the FPGA was programmed to actually test the QRNG. The stream generated during the measurement was then analysed to verify the robustness to hardware non-idealities.

5.1 Behavioural simulations

A test bench was developed to run Vivado simulations and verify the accuracy of the design. It exploits the `std.textio` and the `ieee.std_logic_textio` package to read the input stream from a `txt` file. This system allows to generate an input sequence by easily setting parameters through Matlab and then feed the stream to the simulation. The test bench also creates two output files, one reporting the input sequence after being processed by the input filter and the other one containing the output stream. In this way, it is possible to check the correspondence between bits extracted with the FPGA implementation and the output of a Matlab simulation run using the exact same input stream, that is the one filtered by the input stage. This VHDL simulations have a prescale value for the sampling rate at least equal to the tree height to give the system time to update all nodes as it happens in the Matlab code.

Vivado simulations showed that every part of the code behaves as expected. For example, in Figure 5.1 the sequence processed by the input stage is shown. The input string was generated

using Matlab and the sampling frequency is set to half of the clock to allow the simulation of multiple afterpulses.

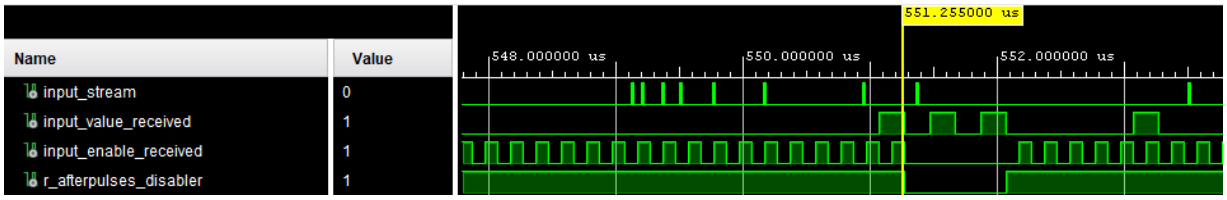

Figure 5.1: *Behavioural simulation of the input filter*

Every time `input_enable_received` has a rising edge, a new input whose value is given by `input_value_received` is sent to the root. After a series of zeros, the `input_stream` that simulates the SPAD signal produces some '1s'. Four pulses are sampled, but two of them are within the 180-ns interval and they shall be considered as unwanted afterpulses. The `r_afterpulses_disabler` prevents the input enabler from going high until the first '0' after the last afterpulse arrives.

Moreover, the tree structure was tested with input sequences of lengths up to hundreds of thousands of bits. For example, a 100 000 bits input stream with balance 0.01 produced 7 207 balanced random bits at the output. The match between the output sequences of the Vivado behavioural simulation and the Matlab code was verified by subtracting them bit-to-bit and extracting nonzero elements. As a matter of fact, a '1' in the difference sequence corresponds to a '1' produced by the VHDL simulation while Matlab generates a '0'. Vice versa, a '-1' stands for a '0' generated by the VHDL simulation and a '1' by Matlab. If the two random bits coincide, a '0' value is produced. As expected, none nonzero elements were found. The array containing the difference between the two sequences is plotted in Figure 5.2.

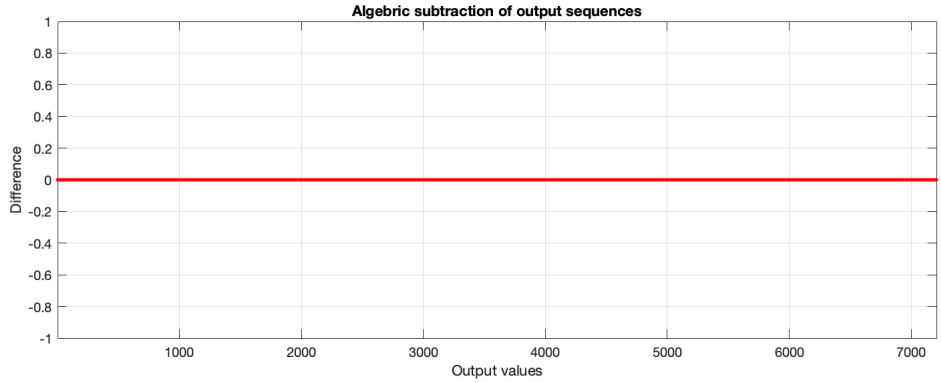

Figure 5.2: *Bitwise difference between the random streams produced by the behavioural simulation and Matlab*

5.2 Design implementation

Running Vivado design implementation showed drawbacks of this architecture. As we could expect, the software converts the node's specifications into LUTs whose components depend on the architecture described in VHDL. At the same time, the board resources set a limit on the maximum number of nodes that can be realized. Using this design, the Zynq7020 was verified to be able to support a maximum of 7 layers of the tree. Note that an optimization to enlarge the tree size would not significantly affect the balance of the output sequence, while it would increase the generation rate.

Anyway, the most critical component is the output buffer. It comprises a quite complex "for loop" that describes a large hardware process to be executed within a clock cycle. The expensive part is looking for the locations of the '1s' inside the array of random bit enablers produced by the nodes. Due to the architecture's complexity, a 10-MHz and 5-layer system was implemented. This setup was used to run a real FPGA measurement, so other adjustments were needed. For example, with this settings the 180-ns interval of the SPAD lasts only 2 clock cycles and also the sampling prescaler can be reduced.

It is still possible to reach higher performances by changing the design of the output module. This can be done, for example, by choosing an optimized research method to perform the '1s' locations inspection. Here, some simple ideas derived from the analysis of the Zhou-Bruck algorithm are proposed.

First, recall the observation pointed out in Section 3.5. Inside each layer, some nodes give a consistent contribution to the output stream, while others are even expected to be completely unproductive. Implementing the overall structure is likely to become a quite large waste of resources. A better solution can be achieved by choosing the most relevant paths of the tree and implementing only the relative nodes. This can speed up the execution of the algorithm without heavily affecting the properties of the unbiased random stream.

Plus, observe that whenever a node generates a number, none of its children is going to receive a new input during the same sampling period. This implies that none of them will be able to generate random bits. If an adequate frequency decimator is used to generate the sampling signal, then it is possible to reduce the size of the enabler array by simply using OR gates.

The paths reduction strategy and the OR gates can also be combined to obtain a more efficient design. Anyway, these proposals aim at reducing the length of the random bit enabler array, but a serious improvement can be obtained only completely re-designing the way the tree communicates with the output stage. However, as already stated, the system was proven to behave correctly, even with the timing violations present.

5.3 Measurement setup and procedure

Finally, the Zedboard was programmed to actually implement and test the QRNG. For this measurement, the height of the tree was set to 5, as well as the sampling prescaler. The output register was configured to include 20 cells. The experimental setup is shown in Figure 5.3. The measurement was performed directly connecting the Zedboard to the SPAD signal that provides the biased random sequence. A 780-nm laser combined with a series of attenuators was used as photon source. Signals driven by the FPGA were measured by mean of a quTAG. This device is a commercial TDC that analyzes waveforms and stores each time a rising edge occurs with a precision of 1 ps. It has four input channels, so that multiple signals can be logged at the same time. In our case, two were used to register the input sequence actually fed to the tree, that is the one filtered by the input module, while the other two channels served to store the output, unbiased random stream. Each pair of signals is composed by the streaming value and its enabler. This allows to reconstruct the two sequences and compare experimental data with the Matlab outcome.

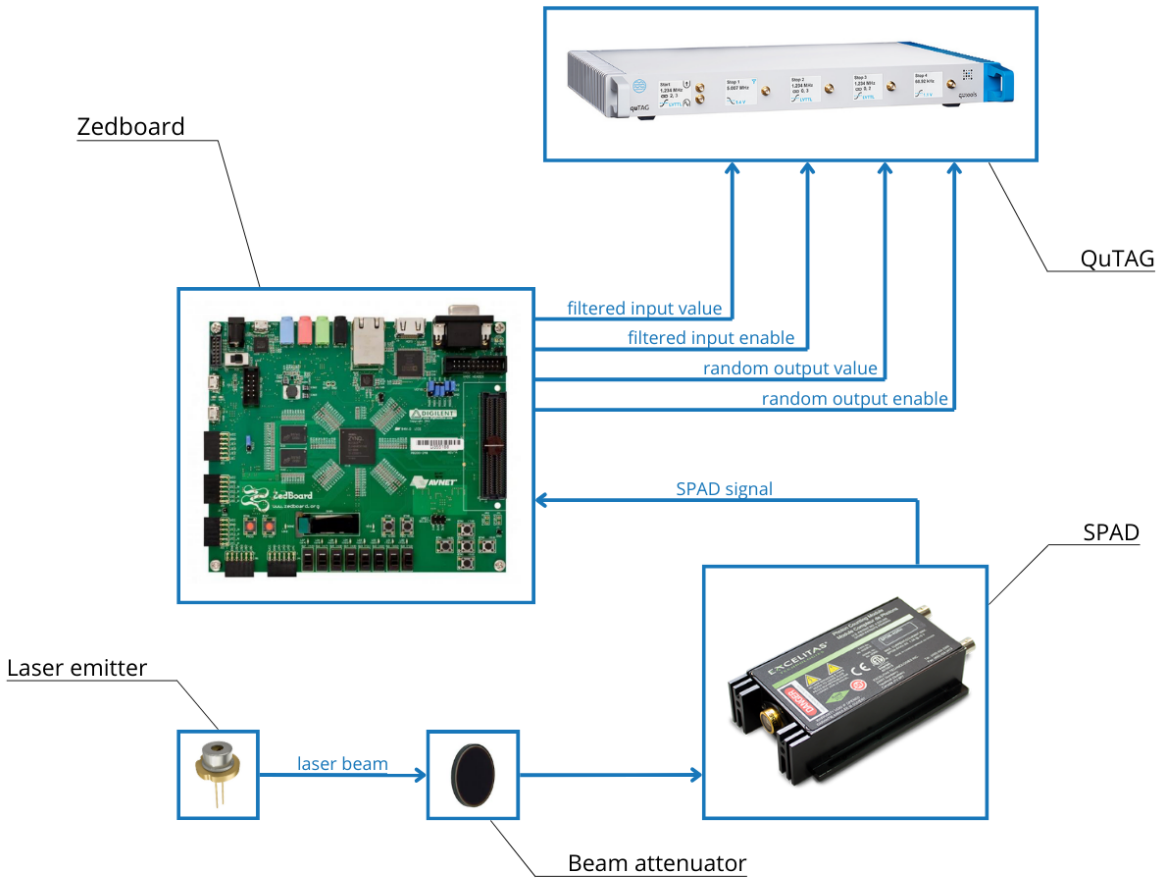

Figure 5.3: *Schematic representation of the measurement setup*

While the experiment was running, the quTAG showed the repetition rate of the signals (evaluated within a specific time window, usually set to 1 second). The channel reporting the input enabler was correctly set to 2 Mcount/s, which is the result of using a 10-MHz clock combined with a decimator of factor 5. The line carrying the input values was the slowest one and was in the order of 10 kcount/s, which is perfectly in line with the expectations, as the input stream is highly biased and this corresponds, in terms of digital signals, to having the SPAD output that is, most of the time, low. The output lines also seemed to have reasonable repetition rates. The channel connected to the output enabler line showed a slightly variable real-time rate comprised between 47 kcount/s and 49 kcount/s. Note that this generation rate is close to the frequency of the '1s' received in input multiplied by the tree height. This is not a coincidence, since random bits are produced whenever the node status differs from the received input value. Most of the time, nodes receives '0s', thus we can expect all of them manly being in the T state. When a pulse is detected, the root received an H and, as a consequence, changes state and passes another H value to a child node. The successful input-state combination is recreated for the child and the same behaviour is repeated. Thus, every detected pulse generally causes 5 random bits to be produced, each one generated by a node of a different layer. Moreover, the repetition rate of the signal carrying the random stream values was observed to vary in a small interval centered in 24 kcount/s, which is the half of its enabler rate. This is the behaviour expected for a signal that have been correctly rebalanced: around half of the random bits produced is '1', the other half is '0'.

5.4 Experimental data analysis

Proper analysis was conducted on the quTAG experimental data. N = 26 699 239 is the number of values sampled from the SPAD signal and fed to the tree. This parameter has been previously referred to as "input string length". The input bias resulted in $b_{in} = 0.0049$ and it corresponds to a sequence biased in zero over 99.5%, that is close to the target input bias according to original setup [5]. The Zedboard successfully processed the input data and produced 636 268 random bits.

5.4.1 Accuracy of the acquired data

The data extracted from the quTAG allowed the reconstruction of the input sequence and it was used to run a Matlab simulation with the same tree properties as the real experiment. Correspondence between the outcomes of the two systems was evaluated as explained in Section 5.1. Also in this case, a perfect match for every generated bit was verified. Again, Figure 5.4 shows the bitwise difference array of the output strings' values. The importance of this result is the fact

that it verifies the reliability of the system. The entropy source is described by laws of quantum mechanics and the elaboration stage is deterministic. This implies that a precise model of the overall apparatus can be developed to characterize the device, test it and assure its correct functioning. It would also be possible to implement a checking system if a semi-self testing QRNG is preferred.

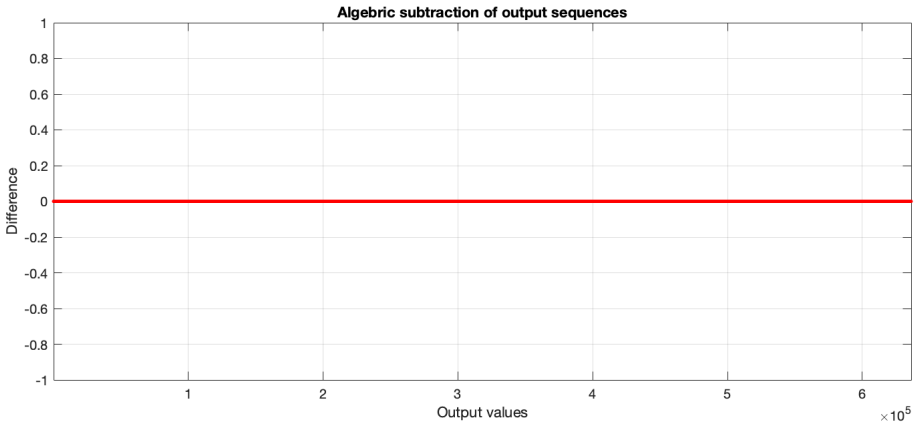

Figure 5.4: *Bitwise difference between the random streams produced by the Zedboard measurement and Matlab*

From the same result we can also derive that timing constraints refer to worst case scenarios that are extremely unlikely to happen. This could be, for example, the case of all nodes producing random bits during the same clock cycle. We know that this situation is not even realizable for many reasons stated before. For starters, not all nodes receive new bits in the same clock cycle and thus they cannot produce any number. Furthermore, a node generating a bit excludes the possibility of a bit generation from its descendants. Thus, some nodes may not produce any bit at all. Of course, the usage of a faster clock would be interesting even if it would increase the risk of timing constraints violations.

5.4.2 Random stream examination

It was already stated that the random bit generation rate achieved is around 48 kHz and that the total number of bits produced is 636 268. The compensation of the input bias is quite satisfying since the output balance determined is $b_{out} = 0.5002$. The actual result is better than what we could expect by observing the simulations shown in Section 3.3 thanks to the high number of input values received.

Moreover, the autocorrelation of the re-balanced random stream was verified because the experiment outcome is a larger set of useful data with respect to the ones generated by the Matlab simulations. The autocorrelation is plotted in Figure 5.5. It confirms that the Zhou-

Bruck algorithm does not affect the string's values correlation. Whitin this framework, this is enough for us to consider the output as truly random.

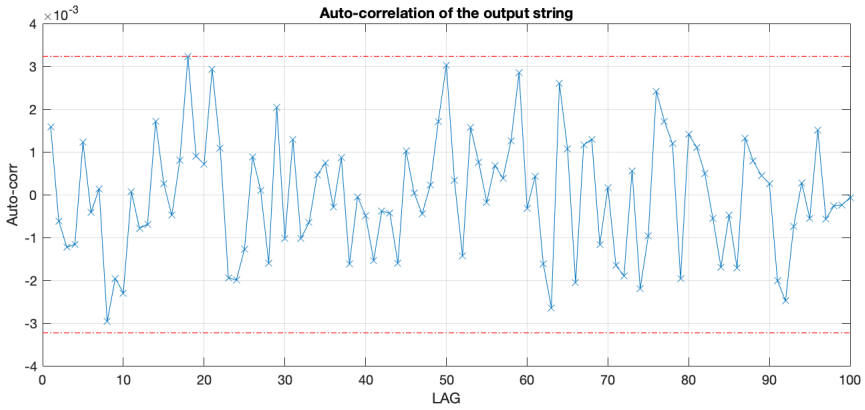

Figure 5.5: *Autocorrelation of the output string processed by the Zhou-Bruck algorithm recreated from the quTAG data*

Conclusion

This thesis has presented a QRNG that exploits a single-photon detector as entropy source. In particular, the information is extracted from the arrival time of photons that are originated from an attenuated laser beam. The FPGA samples the detector's signal with a rate high enough to produce a random stream highly biased toward the '0s'. The validity of the Zhou-Bruck algorithm as real-time re-balancing technique for this application was verified and widely analyzed. It was proved that the algorithm does not introduce any correlation among bits of the random stream and that it does not show memory effects. The VHDL code was built to resemble as much as possible the theoretical procedure of the algorithm, even if non-ideal elements could not be compensated, such as the propagation time of each node or the finite length of the output buffer. Anyway, the precision of the results testified the reliability of the FPGA-design. Thanks to the real implementation, it was also possible to demonstrate that these non-idealities do not heavily affect the random stream produced. Thus, results in terms of design accuracy and bias-compensation properties can be considered as widely satisfying. On the other hand, the major drawback of the device is the speed limitation. The system was tested with a 10-MHz clock obtaining a final random bit generation rate equal to 48 kbps. Since the device proved to be reliable, the generation rate can be improved by simply reducing the prescale register to increase the sampling rate. Future steps will investigate the possibility to increase the clock frequency, improving, in turn, the overall generation bitrate.

As a matter of fact, the proposed design gave positive results with the specified settings. This outcome is very important since for the first time an independent analysis on the algorithm presented by Zhou and Bruck was conducted and the validity of their solution was verified. Indeed, it was successfully implemented as part of the QRNG described in this thesis and the overall device can be considered suitable for low-speed applications. For example, it could be used to generate encryption keys that are able to guarantee the confidentiality of a particular communication channel. This application is known as Quantum Key Distribution (QKD). Also, it could fit devices intended for increasing the security of the transmission and storage of sensitive information, an application widely exploited by data centers or banks. Plus, online lotteries, satellite communications, the Internet of Things (IoT) and, more in general, cryptography techniques

could benefit from using quantum-safe technologies based on this QRNG. In addition, the proposed device can be upgraded to higher working frequencies to expand its field of application and some alternative solutions to perfom this optimization have been described.

Bibliography

[1] P. Kietzmann, T. C. Schmidt, and M. Wählisch, *A Guidline on Pseudorandom Number Generation (PRNG) in the IoT*. ACM Comput. Surv., Association for Computing Machinery, 2021.

[2] C.-C. Li and B. Sun, *Using Linear Congruential Generators for Cryptographic Purposes*. International Conference on Computers and Their Applications, 2005.

[3] P. Junod, *Chryptographic Secure Pseudo-Random Bits Generation: The Blum-Blum-Shub Generator*. Academia.edu Publishing, 1999.

[4] V. Mannalath, S. Mishra, and A. Pathak, *A Comprehensive Review of Quantum Random Number Generators: Concepts, Classification and the Origin of Randomness*. Quantum Infomation Processing, Springer, 2023.

[5] A. Stanco, D. G. Marangon, G. Vallone, S. Burri, E. Charbon, and P. Villoresi, *Efficient random number generation techniques for CMOS single-photon avalanche diode array exploiting fast time tagging units*. American Physical Society, 2020.

[6] A. Stanco, D. G. Marangon, G. Vallone, S. Burri, E. Charbon, and P. Villoresi, *Certification of the efficient random number generation technique based on single-photon detector arrays and time-to-digital converters*. IET Quantum Communication, 2021.

[7] J. Serrano, *Introduction to FPGA design*. CAS - CERN Accelerator School: Digital Signal Processing, 2008.

[8] B. Zeidman, *Introduction to CPLD and FPGA design*. Embedded System Conference, San Francisco, 2004.

[9] H. Zhou and J. Bruck, *Streaming Algorithms for Optimal Generation of Random Bits*. arXiv:1209.0730, 2012.

[10] J. von Neumann, *Various Techniques Used in Connection With Random Digits*. Natural Bureau of Standard Applied Mathematics Series 12, 1951.

[11] Y. Peres, *Iterating von Neumann's procedure for extracting random bits*. Annals of Statistics, vol. 20, 1992.