NTNU – Trondheim
Norwegian University of
Science and Technology

Energy Efficient True Random Number Generator

Conrad Georg Foik

Master of Science in Electronics
Submission date: June 2015
Supervisor: Per Gunnar Kjeldsberg, IET
Co-supervisor: Alf Petter Syvertsen, Silicon Labs Norway AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Project Description

Student: Conrad Georg Foik

Project Title: Energy Efficient True Random Number Generator

Project Definition: Silicon Labs is a world leader in analog-intensive, mixed signal semiconductors, and the design offices in Oslo are the hub for developing the next generation of energy-efficient microcontrollers. The microcontrollers are based on ARM cores with best-in-class energy friendly peripherals.

In a world of connected hardware, true random numbers are needed for proper cryptography. Getting entropy from on-chip sources with some random variations, such as thermal noise in the ADC, true random numbers can be generated using software algorithms. This software implementation is however costly in terms of energy and CPU-time and too expensive for a node with limited energy budget.

During an autumn project a literature study has been performed with focus on the theoretical background for true random number generators. Based on this different hardware approaches for true random number generation have been studied and the hardware complexity of a selected set of techniques has been evaluated bearing energy efficiency in mind.

This work shall be continued towards a master thesis. The two most promising true random number generator hardware solutions shall be studied in more detail and an energy efficient design shall be implementation for at least one of them. As part of the considerations, entropy sources will be studied in more detail to enable also a trade-off between energy consumption in the source and for post-processing.

Supervisor 1: Professor Per Gunnar Kjeldsberg (NTNU)
Supervisor 2: M.Sc. Alf Petter Syvertsen (Silicon Labs)

Abstract

For modern cryptography, the availability of true random numbers is indispensable. While recent technology trends require secure communication, they combine this requirement with the need for energy efficient solutions. As a result, true random number generators (TRNG) which satisfy both aspects have to be developed.

Based on this motivation, the here presented project has been focused on the realization of a TRNG for a mixed-signal microcontroller unit (MCU) environment. These kind of MCUs generally contain an analog-to-digital converter (ADC), which is well known to be influenced by random noise processes, as for example thermal noise. To avoid unnecessary design and prototype costs, it is therefore reasonable to try to implement the entropy source of a TRNG based on the existing ADC design. Possible non-random imperfections in the output of the source can be mask by deterministic post-processor algorithms. Two possible post-processors are the von Neumann corrector (VNC) and an extractor based on pairwise independent hash functions (IHF).

To evaluate the proposed concept, during this project the ADC of a typical MCU has been set up to function as an entropy source. The so generated data has been used as a basis for further simulations and analyses of the statistical characteristics of different TRNG designs. To ease these analyses, a novel test has been developed, which can be used to test a bit stream for the existence of special statistical characteristics, required by the VNC.

In addition, both the VNC and IHF have been analyzed with regard to their complexity and implemented in *SystemVerilog*. In order to find an energy efficient implementation of the IHF, two different algorithmic solutions have been considered and the chosen design has been kept generic to be tunable. For the VNC different approaches of clock gating have been explored to reduce unnecessary dynamic power consumption. After the verification of the proposed designs, both post-processors have been synthesized in a standard 65 nm technology, in order to estimate their power performance.

Finally, in connection with the ADC based source, both post-processor designs have been evaluated with regard to both randomness and energy performance. While the output of the approach using the VNC is classified as not random, the IHF based design passes the so called NIST test suite for random numbers, and can therefore be considered to be random. Hence the connection of the ADC based entropy source and the IHF depicts a functional TRNG solution. By tuning the IHF, it has been able to reach an approximated minimum energy consumption of 5.9 nJ for this approach.

Preface

The in the following presented master thesis summarizes a project performed in the spring of 2015 and marks the end of a five year master program in electronics at the Norwegian University of Science and Technology (NTNU). During the project, the energy efficient generation of true random numbers in a microcontroller has been explored. The project has been supervised by Professor Per Gunnar Kjeldsberg (NTNU) and M.Sc. Alf Petter Syvertsen from Silicon Laboratories (Silicon Labs).

As the project specification combines two extremely complex subjects, the project has had to cover a wide range of different topics, which has made the project both exciting and challenging. Examples are the mathematical and information theoretical background to understand the concept of randomness and possible post-processing approaches; statistical hypothesis testing to be able to evaluate testing results; a basic understanding of analog circuitry to consider entropy sources; digital design techniques to develop energy efficient solutions; and practical aspects as the setup of simulation and synthesis tools.

Many of this aspects are in this report presented in rather great detail. The motivation behind this is twofold. First, the presented background is considered necessary to understand the during the project developed designs and methodologies. Second, it has been my aim during the writing process to not only enable the reader to understand the here presented implementations but to ease the access to other related projects. As a result, while the theoretical background is kept as general as possible, a number of illustrating examples have been included. Being aware that this might exhaust reader familiar with the topics, it is nevertheless believed to be helpful for others.

To prepare the the performance of this project, a literature study has been performed during the fall of 2014. For convenience, it has been chosen to include some of the findings of this study in the here given report. Readers familiar with the concluding report of the literature study might therefore recognize some of the sections in this thesis. This applies mainly to Chapter 1, which has only been slightly adapted to this project. The in Section 2.1 presented concepts have also been considered during the literature study. However, the section has been revised to give a more intuitive access to the discussed topic. Section 2.3 has with minor variations been introduced during the fall of 2014. It should be noted that, while the two post-processors have been discussed during the literature study, additional aspects - including a new algorithmic solution for the extractor based on pairwise independent hash functions - have been incorporated into Section 2.4. This is also true for the corresponding complexity analysis in Section 4.1.1 and Section 4.1.2. The rest of the report is novel.

Generally, great care has been taken to reference external sources, in order to clearly distinguish the from my own work. However, two cases should be considered in more detail.

The in Section 2.4.2 presented Equation 2.43 and Equation 2.44 are in this form *not* stated in the original article, "True Random Number Generators Secure in a Changing Environment" by Barak, Shaltiel and Tromer. However, due to discrep-

ancies between the notation of the in the article presented corresponding equations and additional inconsistencies between these equations and their related mathematical explanations it is suspected that minor printing errors exists in the equations of the article. As an attempt to contact the authors has been unsuccessful and proving the validity of either one of the equations is out of the scope of this project, this suspicion has had to be accepted, out of a lack of alternatives. Both equation have therefore been modified by me, based on my understanding of the mathematical background presented in the article. Nevertheless, the reader is encouraged to evaluate this decision for him- or herself.

Further, I want to stress at this point, that the basic idea of the in Section 3.2 introduced test, has been proposed by Professor Øyvind Bakke (NTNU).

Finally, on a more personal note, I would like to thank my supervisors Professor Per Gunnar Kjeldsberg (NTNU) and M.Sc. Alf Petter Syvertsen (Silicon Labs) for consistent support and motivation throughout both this project and the preceding literature study; Professor Ingelin Steinsland, Professor Håkon Tjelmeland and the already mentioned Professor Øyvind Bakke (all NTNU) for increasing my understanding of statistical hypothesis testing; and finally, but in no way less important, my friends and family for utterly support, motivation, exhortation and distractions, not only throughout this project but the entire past five years.

Trondheim, June 12, 2015

Conrad G. Foik

Contents

List of Figures

List of Tables

List of Mathematical Symbols

Symbol	Explanation	First reference
a_i	An arbitrary element of $\mathcal{A}_X$.	Section 2.1
$\mathcal{A}_X$	$\mathcal{A}_X = \{a_0, ..., a_{k-1}\}$, the *set* of X. It contains k different discrete data values, which all are possible to be taken by x. In the binary case, $\mathcal{A}_X = \{0, 1\}^n$ denotes the set containing all 2^n possible combinations of n bits.	Section 2.1
α	The *level of significance* for a statistical test	Equation 2.19
β	The *tuning parameter* of the IHF	Equation 4.5
$c_{-/+}$	The *critical value(s)* for a statistical test	Equation 2.19
γ	The *switching activity* of a digital gate	Equation 2.47
$\mathrm{dist}(X, Y)$	The *statistical distance* between the ensembles X and Y	Equation 2.6
e	The *expected frequency of occurrence* for a given value. In Section 2.3, $e[n]$ is incoherently used to illustrate the influence of thermal noise.	Section 2.2.1
$\mathrm{E}[x]$	The expected value for the outcome x	Section 2.2.1
ϵ	A measure of the statistical distance of two ensembles. In this report also often referred to as the *quality parameter* of the IHF	Section 2.1, Section 2.4.2
$g(z)$	A polynomial in $GF(2^n)$	Equation 2.35
$G(z)$	A *irreducible* polynomial that defines $GF(2^n)$	Section 2.4.2
$GF((2^n))$	A *Galois field* with 2^n elements	Section 2.4.2
H	A pairwise independent family of hash functions	Section 2.4.2
$\mathrm{h}(x = a_i)$	The *Shannon information content* associated with the event of x taking on the value a_i	Equation 2.1
$\mathrm{H}(X)$	The *Shannon entropy* associated with the ensemble X	Equation 2.2
$\mathcal{H}_0$	The *null-hypothesis* of a statistical test	Section 2.2.1
k	Number of elements in $\mathcal{A}_X$	Section 2.1

Symbol	Explanation	First reference
κ	The associated amount of min-entropy in bits	Equation 2.3
l	The level of independence between multiple variables	Equation 2.14
m	The number of elements in $\overrightarrow{y}$. Also more loosely used as the number of output bits of a test or an algorithm	Section 2.1
min-Ent(X)	The *min-entropy* associated with the ensemble X	Equation 2.3
N	The number of performed *first-level* tests	Section 2.2.2
n	The number of elements in $\overrightarrow{x}$. Also more loosely used as the number of input bits to a test or an algorithm	Section 2.1
norm$(s;\mu,\sigma)$	The *normal distribution* of s with mean μ and standard deviation σ	Section 2.2.2
o	The *observed frequency of occurrence* for a given value	Section 2.2.1
O	An abstract measure of the number of operations required by an algorithm	Section 4.1
$p_{(i)}$	p_i is the probability that x takes on the value a_i, $p_i = \Pr(x = a_i)$. If $\mathcal{A}_X = \{0, 1\}$, p is used to denote the probability that x is equal to 0, $p = \Pr(x = 0)$.	Section 2.1
P	The *power dissipation* of a digital gate	Equation 2.45
$\mathcal{P}_X$	$\mathcal{P}_X = \{p_0, ..., p_{k-1}\}$, a set that defines the probabilities of x taking a particular value of $\mathcal{A}_X$	Section 2.1
pdf$(s;\mathcal{H}_0)$	The pdf of the *reference distribution* of s for a given $\mathcal{H}_0$	Section 2.2.1
π	The *public parameter* of the IHF	Section 2.4.2
Π	A set from which π is randomly selected	Section 2.4.2
s	The *test statistic* of a statistical test. s is a unknown variable. The *observed test statistic* is denoted s_o	Equation 2.18
t	The *security parameter* of the IHF	Equation 2.34
x	The *outcome* of X. A discrete variable taking a value out of $\mathcal{A}_X$ with a probability defined in $\mathcal{P}_X$	Section 2.1
X	The *ensemble* X contains the triple $(x, \mathcal{A}_X, \mathcal{P}_X)$. In this report, X is usually used to model a source of discrete data.	Section 2.1

Symbol	Explanation	First reference
$\overrightarrow{x}$	$\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$, the concatenation of n data values. Frequently used vectors are: $\overrightarrow{x}$ used to model the output stream of a data source / input to a post-processor or statistical test; $\overrightarrow{y} = \langle y_0, ..., y_{m-1} \rangle$ denoting the output of a post-processor.	Section 2.1, Section 2.4
$\chi^2(s; k-1)$	The *chi-squared* distribution of s with $k-1$ degrees of freedom	Section 2.2.1

List of Acronyms

Acronym	Description
ADC	*Analog-to-Digital Converter*
AES	*Advanced Encryption Standard*
CMOS	*Complementary Metal-Oxide Semiconductor*
DUT	*Device under Test*
FPGA	*Field-Programmable Gate Array*
IHF	*Extractor based on pairwise Independent Hash Functions*
IoT	*Internet of Things*
LSB	*Least Significant Bit*
MCU	*Microcontroller Unit*
NIST	*The National Institute of Standards and Technology*
NTNU	*Norwegian University of Science and Technology*
pdf	*Probability Density Function*
PRNG	*Pseudorandom Number Generator*
RNG	*Random Number Generator*
RSA	*Rivest-Shamir-Adleman Public Key Cryptosystem*
RTL	*Register-Transfer Level*
SAR	*Successive Approximation Register*
TRNG	*True Random Number Generator*
VNC	*von Neumann Corrector*

Chapter 1

Introduction

Random numbers are a fundamental part of modern cryptography. They are mainly used to determine keys in cryptographic algorithms, as the *Rivest-Shamir-Adleman Public Key Cryptosystem* (RSA), the Diffie-Hellman algorithm or the *Advanced Encryption Standard* (AES) [1][2]. However, they find also use in the generation of, for example, padding bits, which are used to extend short messages to a fixed number of bits without weakening the used cipher [3]. Due to the importance of random numbers for the security of cryptographic systems, *Random Number Generators* have to satisfy a number of strict demands. Most importantly, they have to deliver random data that is suitable to be used in cryptographic applications. For instance, if the generated data is not truly random, but rather biased or in other form predictable, an adversary might easily guess the key of a cipher from a reduced number of possibilities. It is also possible that an adversary tries to actively attack the RNG in order to determine or even manipulate the generated data. Hence, RNGs should provide an ability to resist these kinds of attacks. A further requirement that has to be taken into consideration is the speed at which random data can be produced, as some applications are time-critical.

The above mentioned requirements have been known for a long time. However, a rather new demand is introduced by current technology trends: During the last decades more and more devices have been connected to the internet. Whereas, in the past, this mostly affected stationary computer systems and later on also applied to mobile devices (e.g. smartphones, laptops), today an increasing number of gadgets is connected to the internet. This trend is generally referred to as the *Internet of Things* (IoT) [4]. The resulting advanced connectivity is accompanied by the same need for security as more traditional systems. One might, for instance, picture a personal medical device, which measures and transmits sensitive data to an external computer system. This data obviously should be shielded from any possible third party. Another example are home security systems. They need to secure that an adversary neither observes the communication nor actively manipulates the system. While the need for security, and thus reliable RNGs, in these systems is the same as for desktop computers, they generally have stricter requirements with regard to power consumption. Most IoT-devices are wireless embedded systems with limited sources of energy as, for example, batteries. Thus, when designing RNGs that are to be used in IoT related gadgets or other kinds of embedded systems, energy

efficiency has become a more and more important factor that has to be taken into consideration.

An illustrative example of typical components used in IoT gadgets is the *EFM32 Gecko family*, which is a series of *Microcontroller Units* developed by *Silicon Laboratories* [5]. On one hand, these MCUs are designed for ultra low energy consumption and thus easily integrated in battery driven systems. On the other hand, the EFM32 Geckos support hardware execution of the widely used AES-algorithm, meeting the increasing costumer requirements for security.

An example of an on jitter based RNG that meets the energy requirements of IoT applications is presented in [3]. The proposed design has a throughput of 1.74 Mbits/s and a power consumption of roughly 240 μW, using a 90 nm *Complementary Metal-Oxide Semiconductor* (CMOS) technology. As a result, the RNG requires approximately 138 pJ per single output bit. However, modern cryptography standards require normally a large number of random bits. For example the AES uses a key length of at least 128 bits [6]. Thus, creating for instance an AES key would require a total of roughly 18 nJ.

1.1 Random data

The concept of randomness is the subject of complex discussions in both mathematics and philosophy. However, for a cryptographic approach it is sufficient to think of randomness as uncertainty [7, p. 155]. Using random data to construct a cipher key should leave any adversary as uncertain about the key value as possible. Uncertainty is as such related to the amount of information on the key the adversary is missing, or in other words, the amount of information she would earn, if the key was revealed to her. In information theory, this kind of uncertainty is referred to as *entropy*. At this point it is sufficient to mention that the higher the entropy value associated with a process is, the more uncertain an observer is about the outcome. In Section 2.1, a more detailed definition of entropy is given.

Random data is accompanied by certain statistical characteristics. Most importantly, random data should ideally be independent and uniformly distributed (see Section 2.1). However, even though this characteristics are important, they should not be confused with randomness [3]. For example, deterministic algorithms exist to model uniform outcomes. This does not necessarily mean that an observer is uncertain about the next output value.

1.2 Pseudorandom and true random number generators

Modern RNGs are realized as electronic systems, either by means of software, hardware or a hybrid solution. However, electronic devices are in general deterministic systems. By definition, generating any kind of truly random data using a pure deterministic system is impossible [8][9, p. 44]. This simple but important fact describes the key challenge of designing modern RNGs and gives reason to divide them into two main classes: *Pseudorandom Number Generators* and *True Random Number*

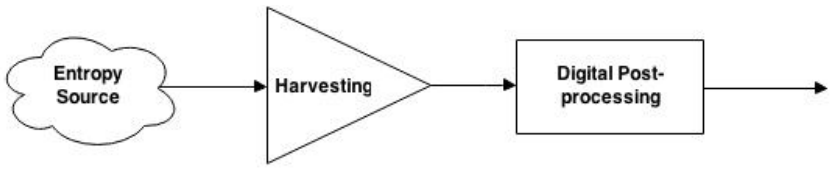


Figure 1.1: Block schematic of a TRNG

Generators.

PRNGs are deterministic systems. They use an input, called *seed*, to generate data that for an external observer appears to be random. However, if the seed becomes known, the output data is exactly predictable. Moreover, if the same seed is used for a sufficient long time, patterns start to arise in the output data, which destroys the illusion of randomness. It should nonetheless be noted that PRNGs in general play an important role in cryptography. So called "cryptographically strong" approaches exist, which prevent an adversary from gaining knowledge about the seed. They also provide satisfying long streams of data without patterns. However, they do not solve the original problem of generating random data as they rely on the existence of a random seed.

TRNGs generate random data by observing random processes inside or outside the system. In general, TRNGs consists of three parts [10][11], as depicted in Figure 1.1:

Entropy-source: The entropy-source is the process to be observed. Examples of such processes range from user interactions over thermal noise in electronic resistors to phenomena in quantum physics. It is important to note that the entropy-source is the only true random part of a TRNG. This means that no other part of the system can increase the amount of entropy associated with the data.

Harvesting: In this step the observed entropy is transformed into a digital data representation. In other words, it can be illustrated as the observation process. A common example is an *Analog-to-Digital Converter* (ADC) which converts an analog noise signal into a digital data stream. As the entropy-source and the harvesting process are closely related, in the literature harvesting is often included in the term entropy-source. This approach is adopted for the purpose of this report.

Post-processing: Most TRNGs use some kind of digital post-processing, even though this is not strictly necessary. The purpose of post-processing varies. Most commonly, it is used to reshape the statistical characteristics of the data, but may also be implemented to increase, for example, the resistance against attacks on the TRNG.

1.3 This project

Motivated by the above discussed, this project focuses on an energy efficient implementation of a TRNG in a mixed-signal MCU environment. Generally, this kind of

MCUs are equipped with an ADC. It is therefore reasonable to try to use the existing ADC as an entropy source for the TRNG, due to two reasons. First, additional design cost for the construction of an alternative source can be omitted. Second, with regard to Section 1.2 it is clear that the truly random characteristics of an entropy source cannot be simulated by means of deterministic computer-aided tools, but must be directly observed from prototypes. MCUs are typically realized by means of advanced CMOS technologies which make the production of single prototypes extremely expensive. Thus, by using the existing ADC as a source of entropy, the functionality of the TRNG can be tested and verified before production, without the need of expensive prototyping.

However, ADCs are generally designed to behave as deterministic as possible. Therefore, they must be considered to be rather weak entropy sources. As a result, post-processing is necessary in order to not compromise the quality of the TRNG. During a preceding literature study [12], the most common post-processing algorithms have been considered in a similar context as given for this project. Based on this study, two algorithms appear to be most suited for the given scenario: the *von Neumann Corrector* (VNC) and the *Extractor based on pairwise Independent Hash Functions* (IHF). Both algorithms are based on firm mathematical proofs, and can thus guarantee randomness for the right circumstances. The VNC is a very simple algorithm with a minimum of complexity. However, its functionality is dependent on very specific statistical characteristics of the used source. In contrast, the IHF is a more advanced algorithm, accompanied by more complexity, but, at the same time, it works for a wider range of source characteristics.

With regard to this project specification, the following has been achieved and is discussed in this report:

- A weak entropy source has been implemented using the ADC of an *EFM32 Wonder Gecko*
- A test for the source characteristics required by the VNC has been proposed and implemented in MATLAB
- A complexity analysis of both the VNC and the IHF has been performed
- A work flow for logical verification and synthesis for the purpose of power estimation has been composed
- Functional version of both the VNC and the IHF have been implemented in SystemVerilog
- Different combinations of the entropy source and post-processing have been evaluated both with respect to randomness and energy performance
- A functional TRNG has been designed, which passes the NIST test suite and uses approximately 5.9 nJ per output bit

1.4 Outline

The here presented report is structured in the following manner. Chapter 2 covers the background of some of the most fundamental aspects of this report. This includes the introduction of the concept of entropy as a measure of randomness, presenting the procedure of statistical hypothesis testing, a short discussion of the possible use of an ADC as an entropy source, the presentation of the two in the project considered post-processors and a brief background on power dissipation in digital systems. In Chapter 3, the setup of an entropy source based on an ADC is presented. In addition, a test to analyze the output data of the source is proposed and the corresponding analysis is performed. Chapter 4 focuses on the post-processors by first performing an complexity analysis and then presenting the implementation of both post-processors in SystemVerilog. After the implementation, the output of the post-processors in combination with the ADC based entropy source is tested for whether it can be considered to be random. This is presented in Chapter 5. Chapter 6 focuses on the power and energy performance of the different post-processors and their combination with the entropy source. Finally, Chapter 7 presents a comparison of the most important aspects of the VNC and the IHF, a conclusion and a proposal of topics that could be explored in a subsequent project.

Chapter 2

Background

This chapter covers the theoretical background of some of the most fundamental aspects of this project. Section 2.1 explores the concept of randomness and Section 2.2 introduces statistical testing, which can be used to test if a data stream can be considered random. The basic concept that enable the use of an ADC as an entropy source are presented in Section 2.3 and Section 2.4 introduces the two in this project considered post-processors. Finally, Section 2.5 focuses on dynamic power dissipation in digital systems.

2.1 Random Data

In Section 1.1, a first informal definition of the concept of randomness has been given. It has been pointed out that randomness depends on the amount of available information about an event, which can be shown to be directly connected to the probability of the event. As such, randomness is closely related to both information theory and probability theory. The following section presents therefore some basic concepts of these fields which are used repeatedly throughout this report. Most importantly, the concept of *entropy* is introduced and established as a measure of randomness.

To be able to discuss the concept of random data in more detail, it is necessary to introduce a mathematical model for a source of data. A first approach is given by the definition of an *ensemble* X as stated in [13, p. 22]: It defines X to be the triple $(x, \mathcal{A}_X, \mathcal{P}_X)$, where x is the *outcome* of the data source, which takes on one of the k values included in the *set* $\mathcal{A}_X = \{a_0, ..., a_{k-1}\}$. The probabilities of x taking on any particular value in $\mathcal{A}_X$ are defined in $\mathcal{P}_X = \{p_0, ..., p_{k-1}\}$, where p_i is the probability that x takes on the value of a_i, $p_i = \Pr(x = a_i)$. Generally, $\Pr(x = a_i)$ is referred to as the *marginal probability* of $x = a_i$. Note that $p_i \leq 0$ for all i in $[0, k-1]$ and $\sum_{i=0}^{k-1} p_i = 1$.

Based on this model of a data source, it is possible to discuss the concept of randomness in a more formal manner. As mentioned in Section 1.1, during this project, randomness is considered to be the amount of uncertainty an observer has about some data. Using the ensemble X to model a source of data, randomness is thus the level of uncertainty an observer associates with the outcome x. Uncertainty can be depicted as the lack of information an observer has about x, or, seen from a

different perspective, the amount of information she earns about x when the concrete value of x is revealed to her.

It is common, to measure the amount of information associated with x taking on a particular value a_i by means of the *Shannon information content*, which is defined as [13, p. 32],

$$\mathrm{h}(x = a_i) \equiv \log_2(\frac{1}{p_i}). \tag{2.1}$$

It is out of the scope of this report to give a detailed discussion about Equation 2.1. However, it is worth noticing the following three properties:

- The value of $\mathrm{h}(x = a_i)$ increases with decreasing p_i. This means that the more unlikely a particular event is the more information is earned if it actually occurs. This fits the human understanding of information.
- If $p_i = 1$, $\mathrm{h}(x = a_i) = 0$. In other words, if an event is certain to happen, no information is gained by observing it. Also this fact is intuitively true.
- Any base might be chosen for the logarithmic function in Equation 2.1. Choosing the base to be 2 allows to use *bits* as the unit of measurement, which is convenient for this project.

Using Equation 2.1, it is possible to measure how uncertain an observer is about one particular outcome. However, this is not sufficient to analyze the source characterized by X. Even though one event might occur with low probability, another value might occur frequently (i.e., with high probability). In such a situation an observer would be rather certain about the outcome - at least most of the time.

As an example, one might consider the ensemble X with $\mathcal{A}_X = \{0, 1\}$ and $\mathcal{P}_X = \{0.0625, 0.9375\}$. Using Equation 2.1, the Shannon information content associated with $x = 0$ is with 4 bits quiet high. However, it is expected that $x = 1$ most of the time and an observer can be close to certain that also the next outcome of X is 1.

It follows from the example, that, in order to find a meaningful measure for the source, all possible outcomes have to be taken into consideration. This can be achieved by using Shannons definition of entropy[1] [13, p. 32] as the average information content of an ensemble X,

$$\mathrm{H}(X) \equiv \sum_{i=0}^{k-1} p_i \cdot \log_2(\frac{1}{p_i}). \tag{2.2}$$

Analyzing a source by means of Equation 2.2 allows to determine how uncertain an observer is about the outcome on the average. For the purpose of this project, a source with a high entropy value is considered to be closer to a perfectly random outcome than a source with a lower associated entropy.

For the sake of completeness, it is necessary to mention that an alternative to entropy as the measure of randomness exists, which is frequently used in related

[1]For readability, Shannons definition of entropy as given in Equation 2.2 is simply referred to as "entropy", throughout the rest of this report.

literature [14]. It is referred to as *min-entropy* and defined for any ensemble X by [14],

$$\text{min-Ent}(X) \equiv \max(\kappa : \Pr(x = a_i) \leq 2^{-\kappa}, \quad \forall i \in [0, k-1]). \tag{2.3}$$

In other words, the min-entropy of an ensemble X is the maximum value of κ such that the possibility for any possible outcome is at most $2^{-\kappa}$. This means that, with regards to Equation 2.1, min-Ent(X) is equal to the smallest information content associated with the ensemble. It is worth noticing that the min-entropy is smaller or equal to the Shannon entropy, and it is therefore a more conservative method to evaluate randomness.

To give a simple example, assuming an ensemble X with $\mathcal{P}_X = \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\}$, the maximal probability is $p_{\max} = \frac{1}{2}$. Using $p_{\max}$ and solving Equation 2.3 for κ results in,

$$\kappa = \log_2(\frac{1}{p_{\max}}), \tag{2.4}$$

which yields to $\kappa = 1$ for $p_{\max} = \frac{1}{2}$.

Even though Equation 2.2 and Equation 2.3 are not equivalent, it should be noted that concept of entropy and min-entropy are fairly similar. Through out the reminder of this section and in large parts of this report, the focus is on entropy. However, in some special cases the min-entropy has to be considered. It is therefore essential to note that the in the following derived principles also hold for min-entropy, even though not explicitly stated [14].

Having established the concept entropy as a measure of randomness, it is natural to establish which characteristics of X yield $\mathrm{H}(X)$ to reach its maximum value. While this can be achieved by means of mathematical computations, a more intuitive approaches is chosen here, leaving a formal proof to Appendix A.

A first characteristic can be disclosed by considering once more the ensemble X with the set $\mathcal{A}_X = \{0, 1\}$. Assuming that an observer would be completely uncertain about the outcome x, she would expect to observe $x = 0$ with the same probability as $x = 1$. By the definition of an ensemble as given above, this means that $\Pr(x = 0) = \Pr(x = 1) = \frac{1}{2}$. In the same manner, this statement can be formulated for any arbitrary data source with $\mathcal{A}_X = \{a_0, ..., a_{k-1}\}$. The outcome x is associated with the maximum amount of uncertainty if the probability of x taking on a particular value is the same for all possible values. In other words, in order to maximize $\mathrm{H}(X)$, x must be uniform distributed over $\mathcal{A}_X$. This means that $p_i = \frac{1}{k}$ for all i in the range $[0, k-1]$. Combining this statement with Equation 2.2 yields,

$$\mathrm{H}(X) \leq \log_2(k). \tag{2.5}$$

Appendix A.1 shows that this is the maximum amount of entropy that can be achieved by source which produces a single outcome.

As the main emphasis of this project lies on binary data, it is worth to consider the consequences of using this kind of data. Denoting a source that produces a n-bit outcome x as X with $\mathcal{A}_X = \{0, 1\}^n$, where $\{0, 1\}^n$ is the set that contains all 2^n variants of a n-bit word, it follows that $k = 2^n$. Thus, by Equation 2.5, the

maximum amount of entropy associated with an n-bits binary source is n bits, which matches the intuitive understanding of randomness.

A source that produces uniformly distributed data is frequently described as *unbiased*. In contrast, a source that produces some outcomes with higher probabilities than others is said to be *biased* towards those outcomes. It is obvious that, for the purpose of this project, an unbiased outcome is desirable. However, this is rarely the case. In practice, one must often consider using a slightly biased ensemble in lack of a perfectly unbiased source. Therefore, it is of interest to find a method that can be used to relate the distribution of an ensemble X with the uniform distribution, or, more generally, with the distribution of any arbitrary ensemble Y, where $\mathcal{A}_X = \mathcal{A}_Y$. This can be achieved by means of the *statistical distance* between X and Y, defined as [14],

$$\text{dist}(X, Y) \equiv \frac{1}{2} \cdot \sum_{i=0}^{k-1} |\Pr(x = a_i) - \Pr(y = a_i)|. \tag{2.6}$$

If $\text{dist}(X, Y) = 0$ the distributions are identical. Otherwise, for $\text{dist}(X, Y) < \epsilon$, the distributions $\mathcal{P}_X$ and $\mathcal{P}_Y$ are said to be *ϵ-close*. As such, ϵ is a measure of the statistical distance.

At this point of the discussion, it follows from Equation 2.5 that it would be desirable to find a data source that produces a uniformly distributed n-bit outcome, with a large n, in order to gather a large amount of entropy. However, most entropy sources (see for instance Section 2.3) can in practice not deliver the amount of entropy required by modern cryptography by means of a single outcome. It is therefore often necessary to combine several single outcomes to one output stream. In this report, an output stream that is constructed through the concatenation of the single outcomes $x_0, ..., x_{n-1}$ is denoted as the vector $\overrightarrow{x} = \langle x_0, .., x_{n-1} \rangle$. For simplicity, it is assumed that each element x_i is the outcome of an ensemble X_i, with the set $\mathcal{A}_X$ of size k, which is the same for all elements in $\overrightarrow{x}$. (Even though this is not necessarily true, this assumption is sufficient for the purpose of this project as it covers a large number of scenarios, for example, outcomes that are produced by the same source.) To analyze the effect of the concatenation on the entropy of the output stream, the elements of $\overrightarrow{x}$ have to be considered in relation to each other, rather than separately.

It is convenient to first consider the simple case of a 2-element vector, $\overrightarrow{x} = \langle x_0, x_1 \rangle$, and establish some basic principles. Later in this section, are then generalized to a vector $\overrightarrow{x}$ with arbitrary length n.

The entropy associated with $\overrightarrow{x}$ is referred to as the *joint entropy* of X_0 and X_1. It is in this report denoted as $\text{H}(X_0, X_1)$ or simply $\text{H}(\overrightarrow{X})$ and defined similar to Equation 2.2 as [13, p. 138],

$$\text{H}(X_0, X_1) \equiv \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} \Pr(x_0 = a_i, x_1 = a_j) \cdot \log_2\left(\frac{1}{\Pr(x_0 = a_i, x_1 = a_j)}\right). \tag{2.7}$$

Here, $\Pr(x_0 = a_i, x_1 = a_j)$ is used to denote the *joint probability* of x_0 and x_1. In other words, $\Pr(x_0 = a_i, x_1 = a_j)$ is the probability of observing the combination $\overrightarrow{x} = \langle a_i, a_j \rangle$.

In order to see how different relations between x_0 and x_1 affect $\mathrm{H}(\overrightarrow{X})$, one can, for instance, think of a scenario where x_0 is collected and added to $\overrightarrow{x}$ before x_1 is collected. If there exists a relation between x_0 and x_1, the occurrences of some event $x_0 = a_i$ will affect the probability of $x_1 = a_j$.

An example could be a source which produces a single bit outcome, where each outcome has a 90% chance of being the logical inverse of the previous outcome. Collecting a first bit, x_0, from this source, it is reasonable to imply that the probabilities of receiving a 0 or a 1 are equal, $\Pr(x_0 = 0) = \Pr(x_0 = 1) = \frac{1}{2}$. However, assuming further that, without loss of generality, $x_0 = 0$, the probability of observing $x_1 = 1$ becomes 0.9.

Generally, the probability of $x_1 = a_j$ given the knowledge of x_0 is referred to as the *conditional entropy* of $x_1 = a_j$ given that $x_0 = a_i$, denoted as $\Pr(x_0 = a_i | x_1 = a_j)$. It can be shown, that the conditional entropy is related to the joint and marginal probability in the following manner [15, p. 63],

$$\begin{aligned} \Pr(x_0 = a_i, x_1 = a_j) &= \Pr(x_0 = a_i) \cdot \Pr(x_1 = a_j | x_0 = a_i) \\ &= \Pr(x_1 = a_j) \cdot \Pr(x_0 = a_i | x_1 = a_j). \end{aligned} \tag{2.8}$$

One might consider the conditional probability $\Pr(x_1 = a_j | x_0 = a_i)$ as an update of the marginal probability $\Pr(x_1 = a_j)$ based on the knowledge that the event $x_0 = a_i$ has occurred [15, p. 64]. In other words, learning one of the outcomes affects the expectation of the other one. Variables that affect each other in this manner are generally said to be *statistical dependent*[2] on each other.

However, if no relation between x_0 and x_1 exists, knowing one outcome does not affect the expectation of the other. Thus, for such variables, the conditional probability of an event equals the marginal probability of the event. In other words, $\Pr(x_1 = a_j | x_0 = a_i) = \Pr(x_1 = a_j)$,and Equation 2.8 simplifies to,

$$\Pr(x_0 = a_i, x_1 = a_j) = \Pr(x_0 = a_i) \cdot \Pr(x_1 = a_j). \tag{2.9}$$

The two variables x_0 and x_1 are said to be *statistical independent*[2], if and only if, they satisfy Equation 2.9 [15, p. 65].

Since dependencies affect the expectations associated with an outcome, it is quite natural to adopt the concept of dependencies also for the discussion of entropy. Appendix A.2.1 shows that Equation 2.7 can be reformulated as,

$$\mathrm{H}(X_0, X_1) = \mathrm{H}(X_0) + \mathrm{H}(X_1 | X_0), \tag{2.10}$$

where the *conditional entropy* $\mathrm{H}(X_1|X_0)$ is defined as the average amount of information associated with x_1 given that x_0 is known [13, p. 138], that is,

$$\mathrm{H}(X_1 | X_0) \equiv \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} \Pr(x_0 = a_i, x_1 = a_j) \cdot \log_2\left(\frac{1}{\Pr(x_1 = a_j | x_0 = a_i)}\right). \tag{2.11}$$

[2]To increase readability, the term "statistical" is dropped in the context of (in-)dependencies throughout most of the rest of this report.

There are two major characteristics of the conditional entropy that should be considered. First of all, the average amount of information associated with x_1 given that x_0 is already known can not be greater than the entropy of the same variable with x_0 unknown [16, p. 41],

$$\mathrm{H}(X_1|X_0) \leq \mathrm{H}(X_1). \tag{2.12}$$

Second, Equation 2.12 holds with equality if and only if the variables x_0 and x_1 are independent [16, p. 41]. A formal proof of this two statements is postponed to Appendix A.2.1. However, considering them with regards to this project, they are intuitively true. If there exists a relation between the two outcomes x_0 and x_1, an adversary who knows x_0 will be less uncertain about x_1. But, if no such relation exists, observing x_0 does not influence the knowledge about x_1.

Finally, combining the results of Equation 2.10 and Equation 2.12 yields,

$$\mathrm{H}(X_0, X_1) = \mathrm{H}(X_0) + \mathrm{H}(X_1|X_0) \leq \mathrm{H}(X_0) + \mathrm{H}(X_1). \tag{2.13}$$

In other words, the joint entropy of X_0 and X_1 reaches its maximum if and only if their outcomes are independent from each other [1, p. 332].

Up to this point, only two concatenated variables have been considered in order to increase readability. Appendix A.2.2 shows how the concept of joint and conditional entropy can be extended to a data stream $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$ with arbitrary length n, while this section is restricted to present the most important result.

However, it is worth to consider one special aspect regarding the use of multiple inputs. While for a 2-element vector the two variables x_0 and x_1 are either dependent or independent, a larger number of elements allows for larger number of variants. As a result, the variables $x_0, ..., x_{n-1}$ are said to be l-wise independent if all l arbitrary variables $x_{i_0}, ..., x_{i_{l-1}}$ are independent of each other [14], that is,

$$\Pr(x_{i_0} = a_{i_0}, ..., x_{i_{l-1}} = a_{i_{l-1}}) = \Pr(x_{i_0} = a_{i_0}) \cdot ... \cdot \Pr(x_{i_{l-1}} = a_{i_{l-1}}), \tag{2.14}$$

where $a_{i_0}, ..., a_{i_{l-1}}$ are arbitrary elements in $\mathcal{A}_X$. The variables in $\overrightarrow{x}$ are said to be *mutual independent* if all n variables are independent of each other, thus,

$$\Pr(x_0 = a_{i_0}, ..., x_{n-1} = a_{i_{n-1}}) = \Pr(x_0 = a_{i_0}) \cdot ... \cdot \Pr(x_{n-1} = a_{i_{n-1}}). \tag{2.15}$$

It is worth noticing, that mutual independence implies l-wise independence for every valid value of l, $l \leq n$.

Having modified the concept of independence for multiple variables, Equation 2.13 can be restated as,

$$\mathrm{H}(\overrightarrow{X}) \leq \sum_{i=0}^{n-1} \mathrm{H}(X_i), \tag{2.16}$$

where $\mathrm{H}(\overrightarrow{X})$ denotes the joint probability associated with $\overrightarrow{x}$. In other words, the entropy associated with a concatenation of n elements is upper bounded by the sum of the entropies which are associated with the single elements when they are

considered separately. It is shown in Appendix A.2.2 that Equation 2.16 holds with equality if and only if the elements of $\overrightarrow{x}$ are mutual independent. As for Equation 2.12, this is intuitively true, since any dependencies must reduce the amount of randomness associated with the outcomes.

Finally, by combining Equation 2.5 and Equation 2.16, it is possible to conclude the following for this section: The entropy associated with a concatenation of single outcomes, $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$, is maximized if and only if all elements of $\overrightarrow{x}$ are independent of each other and uniformly distributed. Hence,

$$\mathrm{H}(\overrightarrow{X}) \leq n \cdot \log_2(k), \tag{2.17}$$

where n is the number of elements in $\overrightarrow{x}$ and k the number of possible outcomes for each element. It is worth noticing that if $\overrightarrow{x}$ depicts a concatenation of single bit values, Equation 2.17 states that the maximum amount of entropy associated with $\overrightarrow{x}$ is n-bits.

It is important to understand the impact of Equation 2.17 on this project. The equation states that the entropy associated with some data is maximized if and only if this data is (mutual) independent and uniformly distributed. In other words, data that has these characteristics is considered to be perfectly random. As a result, independence and an uniform distribution are the most desirable properties of data generated by a TRNG.

2.2 Statistical Testing

Testing that a given TRNG design works as desired is a difficult task which differs vastly from the approaches used to verify the functionality of deterministic systems. In the latter case, the output of the system can be exactly predicted at any time. This makes it possible to compare the observed output values against the expected outcomes. The system then is declared functional if the observed values match the expected values. Obviously, this method cannot be used to verify the functionality of a TRNG, since the motivation behind the construction of a TRNG is that the output shall be unknown (see Chapter 1). However, as determined in Section 2.1, the output is expected to have specific statistical characteristics. It is therefore reasonable to base the verification of a TRNG on whether or not the corresponding output shows the desired statistical characteristics.

In general, testing whether or not a set of data has some statistical characteristics is referred to as testing a *statistical hypothesis*[3] [17, p. 275]. The following section (Section 2.2.1) gives a brief background on statistical hypotheses and the related testing procedure. Besides the theoretical background, the *chi-squared goodness-of-fit* test is introduced to serve both as an example of a hypothesis test and as a basis for further discussion in Section 2.2.2.

[3]As this report is mostly concerned with statistical hypotheses, for readability the term "hypothesis" is used, unless explicitly mentioned, instead of "statistical hypothesis" throughout the rest of the report.

In Section 2.2.2, the from *The National Institute of Standards and Technology* (NIST) proposed *statistical test suite for random and pseudorandom number generators for cryptographic applications* [18] is introduced. For readability, this test suite is in this report referred to as the NIST test suite for random numbers or simply the NIST test suite. It is a frequently used approach to evaluate the output of a TRNG [19][20] and includes 15 different hypothesis tests. Besides of giving a summary of all tests included in the test suite, Section 2.2.2 describes one test in detail, providing a concrete example of a random number test.

2.2.1 Testing of a statistical hypothesis

The principle of statistical hypothesis testing can be illustrated by considering the ensemble X with outcome x. By concatenating several outcomes, a series of observable data, denoted as $\overrightarrow{x} = \langle x_0, x_1, ..., x_{n-1} \rangle$, can be constructed. It is assumed that one or several of the statistical characteristics of X are unknown, but that there exists a hypothesis about their nature. Such a hypothesis can be depicted as a conjecture of the distribution of the elements in $\overrightarrow{x}$ and is as such referred to as a *statistical hypothesis* [15, p. 319]. This hypothesis is the foundation of the here presented testing procedure and traditionally referred to as the *null-hypothesis*, denoted as $\mathcal{H}_0$ [15, p. 320]. Ideally, the objective of the hypothesis test would be to produce a conclusion about $\mathcal{H}_0$, by either rejecting or accepting it, based on the observation of $\overrightarrow{x}$.

A simple example of a null-hypothesis is a conjecture regarding the expected outcome of X, $\mathrm{E}[x]$. For instance, $\mathcal{H}_0$might predict that $\mathrm{E}[x]$ is the fixed value $\mu_0 = 70$. In this report, the shorthand notation "$\mathcal{H}_0$: $\mathrm{E}[x] = \mu_0$" is used to express such a null-hypothesis. This example is used repeatedly in the following, in order to illustrate the presentation of the theoretical background.

In order to test $\mathcal{H}_0$, it has to be related to the outcomes, $\overrightarrow{x}$. This is done by the formulation of a so called *test statistic* [17, p. 276], s, as a function of $\overrightarrow{x}$,

$$s = f(\overrightarrow{x}). \tag{2.18}$$

In addition to being a function of $\overrightarrow{x}$, s must be affected by $\mathcal{H}_0$. In other words, $\mathcal{H}_0$must imply a conjecture of s. The formulation of the test statistic differs from hypothesis to hypothesis and finding a suitable s for a given $\mathcal{H}_0$is a non-trivial task. Vastly simplified, s might be depicted as a measure of $\mathcal{H}_0$.

To illustrate this, the above given example is reused. Given $\mathcal{H}_0$: $\mathrm{E}[x] = \mu_0$, the average value of the elements in $\overrightarrow{x}$, $s = \frac{1}{n} \cdot \sum_{i=0}^{n-1} x_i$, is a reasonable choice of the test statistic [15, pp. 336-337]. Here, x_i is the i-th element of $\overrightarrow{x}$. It follows from the definition of the expected value [15, p. 111], that s converges to $\mathrm{E}[x]$, as n approaches infinity. Hence, given that $\mathcal{H}_0$is true, s approaches μ_0 for large values of n. In this way, s is affected by $\mathcal{H}_0$.

In order to understand how s is related to $\mathcal{H}_0$, it is convenient to ignore any concrete solution of Equation 2.18 for the moment and rather think of s as an unknown value. However, assuming that $\mathcal{H}_0$is true, the distribution of s, which is referred to as the *reference distribution* of the test [18], is known. In other words, the probability for every possible s-value is known, given that $\mathcal{H}_0$is true. It is worth

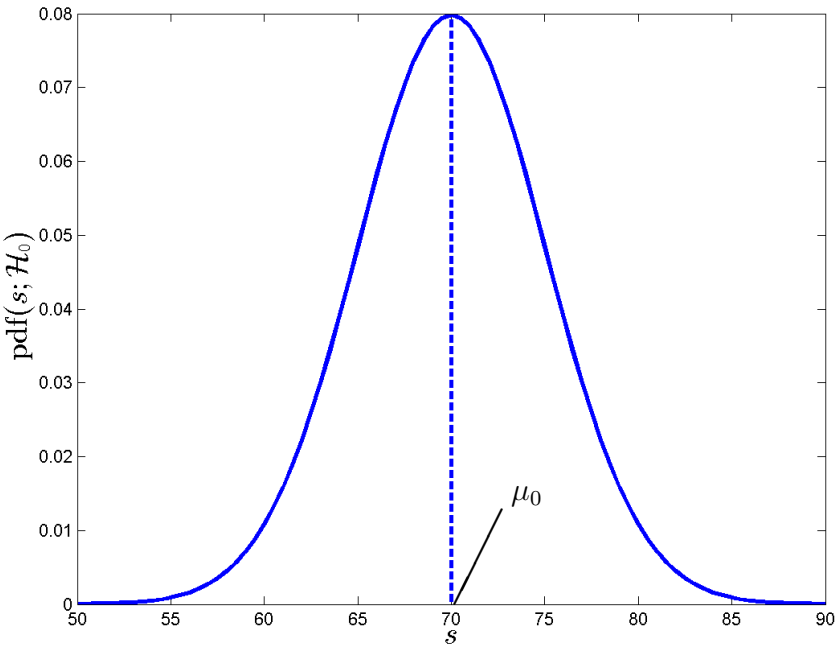


Figure 2.1: Example of a reference distribution for $\mathcal{H}_0$

noticing, that, for the purpose of this project, only continuous test statistics are of interest. This means that the reference distribution, and every other distribution of s (given that $\mathcal{H}_0$is not true), are described by their *Probability Density Functions* [15, p.88]. The pdf of the reference distribution is in this report denoted as $\text{pdf}(s;\mathcal{H}_0)$, if not specified otherwise. For the example $\mathcal{H}_0 : \text{E}[x] = \mu_0$, with $\mu_0 = 70$, it can be shown that $\text{pdf}(s;\mathcal{H}_0)$ can be approximated by the normal distribution[4] with mean 70, as depicted in Figure 2.1.

Taking now the observed outcomes $\overrightarrow{x}$ into consideration, it is possible to find a concrete value of s. To avoid confusion, such a concrete value of the test statistic s is in this report referred to as the *observed test statistic*, s_o. In other words, the test statistic s is considered to be a variable which can take different values, while the observed test statistic s_o is a fixed value.

Using the example of $\mathcal{H}_0 : \text{E}[x] = \mu_0$ and recalling that $s = \frac{1}{n} \cdot \sum_{i=0}^{n-1} x_i$, in general, s can take on the value of any real number while the observed test statistic is, for instance, fixed to $s_o = 63$.

It is possible to relate the observed test statistic to $\mathcal{H}_0$through the reference distribution of s. Thus, by considering the reference distribution, it is possible to make a statement on the probability of observing s_o given that $\mathcal{H}_0$is true, or, in mathematical terms, calculate[5] $\Pr(s = s_o|\mathcal{H}_0)$. Since s_o is a function of the

[4]It should be mentioned that the normal distribution is defined by both the mean of the data and its standard deviation, σ. However, in order to not complicate the given example unnecessarily, it has been chosen to ignore σ. The interested reader may note that $\sigma = 5$ has been used for the numerical examples.

[5]In fact, as s is a continuous distributed variable, the exact computation of $\Pr(s = s_o|\mathcal{H}_0)$ is impossible. However, throughout this section, it will become evident that an exact calculation of

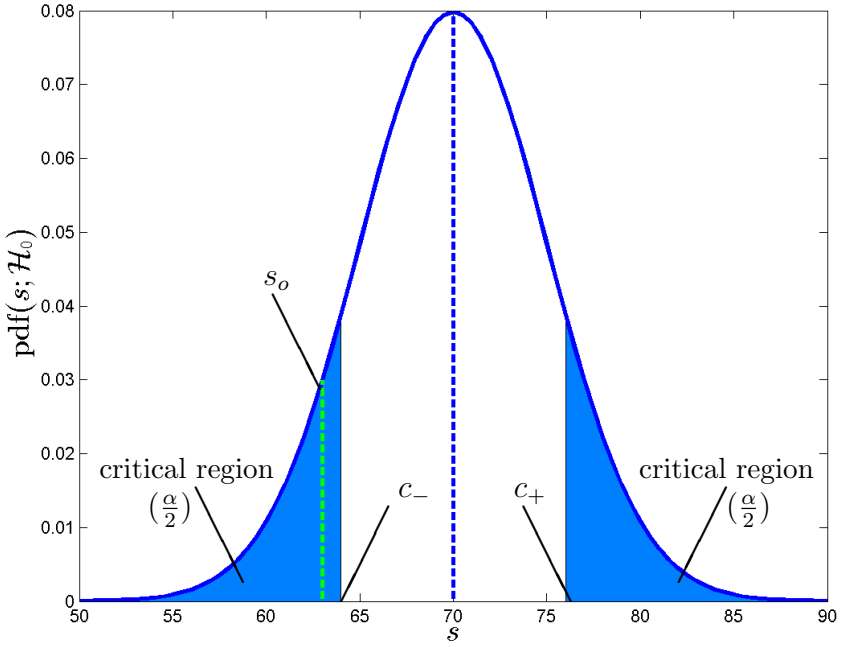


Figure 2.2: Critical regions of the reference distribution

observed outcome $\overrightarrow{x}$, as stated in Equation 2.18, this is also an indirect indicator of the probability of observing $\overrightarrow{x}$, given that $\mathcal{H}_0$is valid. It is therefore reasonable to base the conclusion of the hypothesis test on the observed test statistic, since the confidence about $\mathcal{H}_0$increases if the by $\text{pdf}(s, \mathcal{H}_0)$ defined probability of observing s_o is large. In the same manner, if $\Pr(s = s_o|\mathcal{H}_0)$ is small, the observation of s_o indicates that $\mathcal{H}_0$should be rejected.

Based on this, it is possible to evaluate the test by defining so called *critical regions* of s-values, which correspond to a small $\Pr(s = s_o|\mathcal{H}_0)$, and reject the hypothesis if s_o falls into such a region [15, p. 322]. This is illustrated in Figure 2.2. It is essential to note, that a test has either one or two critical regions, depending on the formulation of the null-hypothesis. For the used example, $\mathcal{H}_0 : \text{E}[x] = \mu_0$, two critical regions exists, as $\mathcal{H}_0$is rejected if $s_o \ll \mu_0$ and $s_o \gg \mu_0$. A test that has two critical regions is generally referred to as *two-tailed* [15, p. 330]. An example of a *one-tailed* test is the chi-squared goodness-of-fit test, which is described at the end of this section.

It can be seen from Figure 2.2, that each critical region has only one clearly defined boundary. For instance, the critical region to the right is bounded by c_+ and infinity. The fixed boundaries c_- and c_+ (or simply c in the case of a one-tailed test) are referred to as the *critical values* of a test [15, p. 322]. Using the critical values, the test can be evaluated by the following procedure: If $s_o \leq c_-$ or $s_o \geq c_+$, $\mathcal{H}_0$is rejected by the test. For the example of $\mathcal{H}_0 : \text{E}[x] = \mu_0$ with $\mu_0 = 70$, one could, for instance, chose $c_- = 64$ and $c_+ = 76$. An observed test statistic of $s_o = 63$

$\Pr(s = s_o|\mathcal{H}_0)$ is not necessary for the evaluation of a hypothesis test. Until then, the reader may note that $\Pr(s = s_o|\mathcal{H}_0)$ can be approximated by computing $\Pr(s_o \leq s \leq (s_o + \Delta)|\mathcal{H}_0)$ for an arbitrary small Δ.

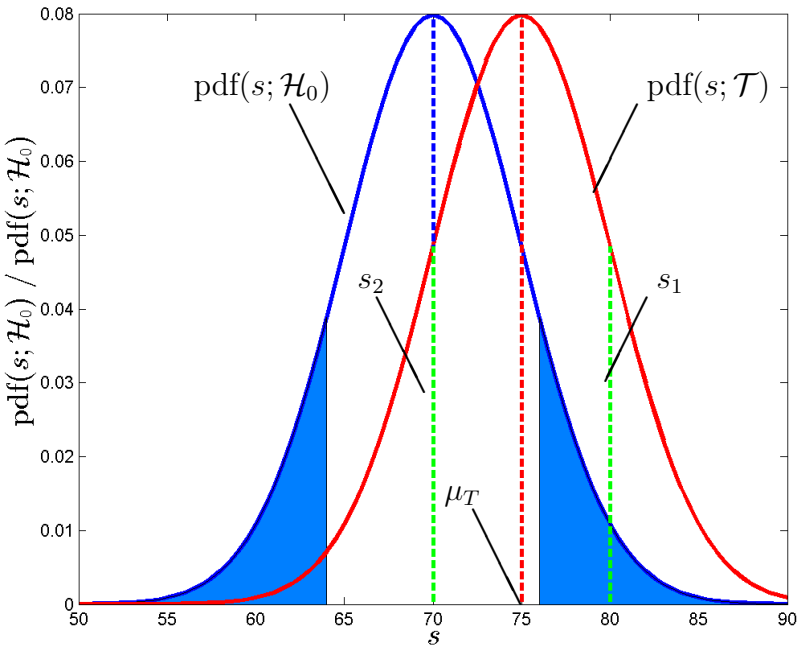


Figure 2.3: Relationship of the reference distribution and the true distribution of s

would thus yield a rejection of $\mathcal{H}_0$as $s_o < c_-$.

Two important consequences arise from the described test setup, which should be noted before the evaluation procedure is described in more detail. First, s can take on a range of values with a given probability. This makes it impossible to derive a deterministic conclusion by means of a statistical hypothesis test [15, p. 319]. For instance, considering Figure 2.2, s might take on a value that corresponds to a critical region, even if $\mathcal{H}_0$is true. Such an s-value would lead to a rejection of $\mathcal{H}_0$, while in fact $\mathcal{H}_0$is true. In the same manner, it is possible that the observed s_o yields a high value of $\Pr(s = s_o|\mathcal{H}_0)$, but $\mathcal{H}_0$is in fact false. Generally, considering a larger set of data will decrease the chances of observing an s-value that is untypical under the true circumstances. In other words, letting n grow large will increase the confidence in the result produced by the test [15, p.324]. Nevertheless, complete certainty can never be provided.

Second, a test of a hypothesis can yield to a conclusion that rejects the hypothesis, but it cannot provide a conclusion that accepts the null-hypothesis [15, pp. 320-321]. This is due to the fact that the true statistical characteristics of X are unknown. As such, even if the observation of some s_o seems to be a typical event given $\mathcal{H}_0$, it cannot rule out the possibility that s_o corresponds to data that has different statistical characteristics. In other words, the observation of a test statistic that is typical for given $\mathcal{H}_0$does not guarantee that the same observed test statistic is untypical for some different (true) circumstances. In contrast, observing an s_o that corresponds to a critical region is a valid indicator of $\mathcal{H}_0$being false, since such a value would be untypical given that $\mathcal{H}_0$is true.

This concept can be illustrated by considering once more the example $\mathcal{H}_0 : \mathrm{E}[x] = \mu_0$ and the situation depicted in Figure 2.3. It is now assumed that the real value of the expected outcome of X is not μ_0 as predicted by $\mathcal{H}_0$, but some different value μ_T. Hence, s is not distributed by the reference distribution, but by its *true distribution*, described by its probability density function, $\mathrm{pdf}(s;\mathcal{T})$. Assuming first that the observed test statistic is s_1, it is evident that, while s_1 is a typical value given the true distribution of the data, the probability of observing s_1 under the assumption that $\mathcal{H}_0$is true,$\Pr(s = s_1|\mathcal{H}_0)$, is low. As a result, $\mathcal{H}_0$is correctly rejected. However, if the observed test statistic is s_2, the probability of observing s_2 given that $\mathcal{H}_0$is true is at its maximum even though $\mathcal{H}_0$is false. Therefore, even though $\Pr(s = s_2|\mathcal{H}_0)$ is large, accepting $\mathcal{H}_0$based on the observation of s_2 would yield to an erroneous conclusion.

Finally, it is worth noticing that s_1 and s_2 correspond to the same value of $\mathrm{pdf}(s;\mathcal{T})$. This means that both observations are equally likely. It follows that the conclusions that can be drawn from a statistic hypothesis test are either the rejection of $\mathcal{H}_0$or the *failure to reject* $\mathcal{H}_0$, rather than an acceptance of $\mathcal{H}_0$.

Since $\mathcal{H}_0$is either true or false, and since the test either yields a rejection of $\mathcal{H}_0$or fails to reject the hypothesis, four possible scenarios arise. Table 2.1 summarizes these situations. Of course, if the test fails to reject $\mathcal{H}_0$and the hypothesis is true or if $\mathcal{H}_0$in fact is false and the test rejects it, the correct decision has been made. Otherwise, the decision is erroneous. Rejecting $\mathcal{H}_0$when it is in fact true is referred to as a *type I error*, while the conclusion to not reject $\mathcal{H}_0$even when it is false, is called a *type II error* [15, pp. 322-323]. Obviously, it is beneficial to design a test, that minimizes the probabilities of committing an error. Unfortunately, both error types affect each other: A small probability of committing a type I error yields a large probability of committing a type II error, and vice versa [15, p. 324]. For instance, simply rejecting every hypothesis would make it impossible to commit a type II error, while the probability of a type I error reaches its maximum. However, as stated above, the confidence that the observed test statistic is a typical value under the true circumstances increases with the size of the set of test data. This reduces the probability of both error types.

Table 2.1: Possible scenarios arising from hypotheses testing

	$\mathcal{H}_0$is true	$\mathcal{H}_0$is false
Do not reject $\mathcal{H}_0$	Correct decision	Type II error
Reject $\mathcal{H}_0$	Type I error	Correct decision

The probability of a type I error is referred to as the *level of significance* [15, p. 323] and denoted by α. It is completely defined by the choice of the test statistic and the critical values. Since a type I error implies that $\mathcal{H}_0$is true, s is truly distributed as described by the reference distribution. This means that it is possible to calculate the probability of observing a test statistic that would yield a type I error by rejecting $\mathcal{H}_0$. For the example $\mathcal{H}_0 : \mathrm{E}[x] = \mu_0$, the hypothesis is rejected if $s_o \leq c_-$ or $s_o \geq c_+$. In order to keep the test unbiased, it is convenient to define both critical values such that it is as likely to observe an s_o that corresponds to the

left critical region, as it is to observe an s_o that corresponds to the right critical region. In other words, the probability of observing an s-value that falls in a specific critical region should be exactly $\frac{\alpha}{2}$. This means, that for the given example, the level of significance can be calculated as

$$\begin{aligned} \alpha &= 2 \cdot \int_{-\infty}^{c_-} \text{pdf}(s; \mathcal{H}_0) \text{d}s \\ &= 2 \cdot \int_{c_+}^{\infty} \text{pdf}(s; \mathcal{H}_0) \text{d}s. \end{aligned} \tag{2.19}$$

It is worth noticing, that, even though Equation 2.19 has been derived from a specific example, the equation holds for all two-tailed tests which make use of a continuous distributed test statistic.

Figure 2.2 illustrates the relation between α and the critical values for $\mathcal{H}_0 : \text{E}[x] = \mu_0$ for $c_+ = 76$ and $c_- = 64$. Using Equation 2.19 to calculate the level of significance for the chosen critical values yields $\alpha \approx 0.2301$. Thus for the given example, the probability of committing a type I error is approximated 23.01%, which is quite high. To avoid this, it is in practice common to first specify the level of significance desired by a test, and then solve Equation 2.19 for the critical values. A typical value for the level of significance is, for example, $\alpha = 0.01$ [15, p. 332] [18]. Using this level of significane for the given example, more appropriated critical values would be $c_+ \approx 82.88$ and $c_- \approx 57.12$. It is worth noticing, that this would yield the test to not reject $\mathcal{H}_0$for $s_o = 64$.

In contrast to the level of significance, the probability of a type II error cannot be calculated solely based on the null-hypothesis [15, p. 323]. This is due to the fact that the true distribution of s differs from the reference distribution, if $\mathcal{H}_0$is not true. This is depicted in Figure 2.3. As a result, its not possible to calculate the probability of observing s that yields the test to not reject $\mathcal{H}_0$, given that $\mathcal{H}_0$is false[6].

Finally, instead of defining explicit critical values and basing the conclusion of the test on the comparison of the observed test statistic and c_+ or c_-, an alternative approach exists. The so called *p-value* is defined as the lowest level of significance that would yield to a rejection of $\mathcal{H}_0$based on the observed test statistic, s_o [15, p. 333]. For the example of $\mathcal{H}_0 : \text{E}[X] = \mu_0$, it follows from Equation 2.19 that,

$$\text{p-value} = \begin{cases} 2 \cdot \int_{-\infty}^{s_o} \text{pdf}(s; \mathcal{H}_0) \text{d}s & \text{if } s_o \leq \mu_0, \\ 2 \cdot \int_{s_o}^{\infty} \text{pdf}(s; \mathcal{H}_0) \text{d}s & \text{if } s_o > \mu_0. \end{cases} \tag{2.20}$$

As for Equation 2.19, it has been chosen to use a rather general notation and Equation 2.20 is therefore valid for any continuous distributed test statistic s with a symmetrical $\text{pdf}(s; \mathcal{H}_0)$ and mean μ_0.

Considering Equation 2.20, the p-value can be illustrated as the probability of observing an s-value that differs at least as much from μ_0 as s_o. Figure 2.4 depicts the p-value for the example of $\mathcal{H}_0 : \text{E}[x] = \mu_0$, with $\mu_0 = 70$ and $s_o = 63$. The resulting

[6]If $\mathcal{H}_0$is tested against a concrete *alternative hypothesis*, it is possible to calculate the probability of an erroneous failure to reject $\mathcal{H}_0$in favor of the alternative hypothesis. However, this is of no concern for the purpose of the here presented project and only mentioned for completeness.

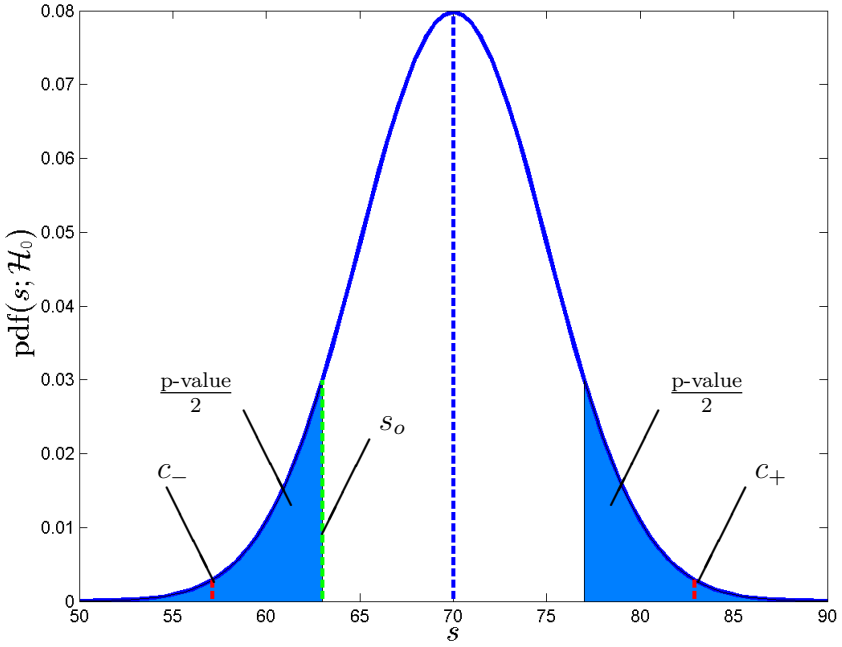


Figure 2.4: P-value for the observed test statistic s_o

p-value is approximately 0.1615. This means that the probability of observing a test statistic that is either equal or smaller than 64 or equal or larger than 76 is roughly 16.15%.

The benefits of an approach which uses a p-value is that it can give a more differentiated indication of the validity of the null-hypothesis than a simple rejection/no-rejection decision based on the critical values. However, it should be noted that, by its definition, the p-value can be used to directly evaluate a test for a given level of significance, α. Comparing Figure 2.2 with Figure 2.4, it is not difficult to see that the p-value exceeds the level of significance, if the observed test statistic, s_o, does not correspond to one of the critical regions. This can also be seen by solving Equation 2.20 with, for instance, $s_o = c_-$ and realizing that the equation becomes equivalent to Equation 2.19. Thus, if the observed test statistic equals a critical value, the corresponding p-value equals α. In the same manner, if the p-value is smaller than α, the observed test statistic lies in a critical region. Hence, for a given level of significance the test can be evaluated directly by using the p-value and rejecting $\mathcal{H}_0$ if the p-value is smaller than α.

An additional benefits of using the p-value is the fact that the p-value is a normalized parameter. This means that it can more easily be compared across different tests with different test statistics. Furthermore, even though exploring this fact in detail is out of the scope of this report, it can be shown that the p-value of a test with a continuous distributed test statistic, follows a uniform distribution, if the null-hypothesis is true [18][21]. This means that the reliability of a test can be increased by running the test on N different data sets and examine whether or not the observed p-values are distributed according to an uniform distribution. Such an approach is referred to as a *second-level test*. An example of a second-level

test is given at the end of this section and the procedure is further discussed in Section 2.2.2.

In order to execute an approach based on a second-level test, it is necessary to be able to evaluate whether or not the p-values are uniformly distributed. One possible way to test if a set of p-values or, more generally, any kind of data follows an uniform distribution is the chi-squared goodness-of-fit test, which is described next.

The chi-squared goodness-of-fit test

The chi-squared goodness-of-fit test is designed to evaluate whether an outcome of an ensemble X follows a hypothesized distribution or not. It is an example of a one-tailed statistical hypothesis test and follows the theoretical procedure, as described above. In general, the test can be executed for any hypothesized distribution. However, the main focus of the here given presentation is to check if the outcome of X is uniformly distributed. This is partly done to keep the discussion as simple as possible, and partly since the uniform distribution has a central role in the second-level test procedure discussed in Section 2.2.2.

As above, the source of the data, that is to be considered, is described by the ensemble X, which produces a set of outcomes, $\overrightarrow{x} = \langle x_0, x_1, ..., x_{n-1} \rangle$. Each element in $\overrightarrow{x}$ takes on one of the values defined in the set[7], $\mathcal{A}_X = \{a_0, a_1, ..., a_{k-1}\}$. The true distribution of X is unknown. However, a null-hypothesis exists, stating that X follows an uniform distribution. This can be expressed as,

$$\mathcal{H}_0 : \Pr(x_i = a_j) = \frac{1}{k}, \tag{2.21}$$

which holds for all i in $[0, n-1]$ and j in $[0, k-1]$.

It is convenient to base the evaluation of $\mathcal{H}_0$on the number of appearances of each element of the set in $\overrightarrow{x}$. The number of appearances of the element a_j is widely referred to as the *frequency of occurrence* [15, p. 370], or simply the *frequency*, of a_j. Under the assumption that $\mathcal{H}_0$is true, the *expected frequency* of a_j is simply the product of the number of elements in $\overrightarrow{x}$ and the in Equation 2.21 described probability of observing a_j. Thus,

$$e_j = \frac{n}{k}, \tag{2.22}$$

where e_j denotes the expected frequency of a_j.

The corresponding actually *observed frequency* of a_j is denoted as o_j. If $\mathcal{H}_0$is true, it is anticipated that the difference between the expected frequencies and the observed frequencies for all elements of $\mathcal{A}_X$ should be small [17, p. 285]. As a result, the test statistic s might be defined as [15, p. 371],

$$s = \sum_{j=0}^{k-1} \frac{(o_i - e_i)^2}{e_i}. \tag{2.23}$$

[7]Using $\mathcal{A}_X$ to describe the source of data limits the presented discussion to sets of discrete data. However, a possible way to apply the chi-squared goodness-of-fit test to a continuous data source is presented at the end of this section

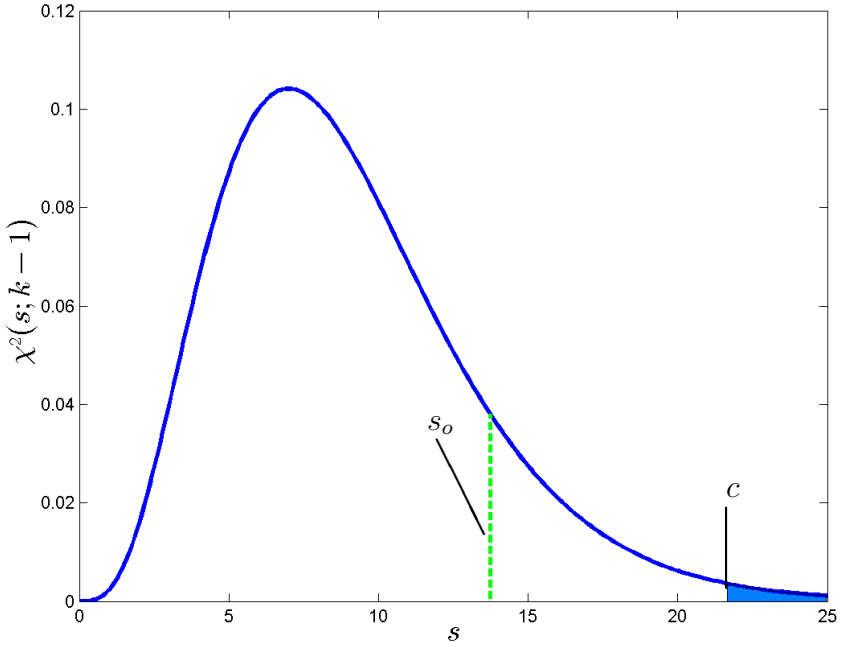


Figure 2.5: The chi-squared distribution with nine degrees of freedom

It can be shown that the reference distribution of s is approximated equal to a chi-squared distribution with $k-1$ degrees of freedom. This approximation yields to valid results, as long as each of the used expected frequencies is at least equal to 5 [15, p. 371]. An example of an chi-squared distribution is shown in Figure 2.5. The probability density function of s following a chi-squared distribution with $k-1$ degrees of freedom is in this report denoted as χ^2(s;k-1).

It is essential to note, that all s-values that are close to zero increase the confidence in $\mathcal{H}_0$. In other words, for the observed test statistic s_o, $\mathcal{H}_0$is only rejected if $s_o \gg 0$. As a result, only one critical value, c, is defined, as illustrated in Figure 2.5, and the test is one-tailed. It is necessary to slightly modify Equation 2.19, in order to be able to calculate the level of significance for a one-tailed test. Since a type I error is only committed if $\mathcal{H}_0$is true and s_o is larger or equal to c, the level of significance is equal to the probability of observing $s_o \geq c$, as defined by $\chi^2(s;k-1)$. Thus,

$$\alpha = \int_c^\infty \chi^2(s;k-1)\mathrm{d}s. \tag{2.24}$$

Modifying Equation 2.20 in the same manner, the p-value of the test for an observed test statistic can be calculated as,

$$\text{p-value} = \int_{s_o}^\infty \chi^2(s;k-1)\mathrm{d}s. \tag{2.25}$$

The presented test can thus be evaluated by either defining α and the corresponding critical value and reject $\mathcal{H}_0$if $s_o \geq c$, or by making a decision directly

based on the derived p-value. For instance, if $s_o \geq c$, it would be concluded that the data contained in $\overrightarrow{x}$ does not follow a uniform distribution.

As a final remark, it should be noted, that the chi-squared goodness-of-fit test also works for sources of data that produce a continuous outcome instead of discrete data defined by a set $\mathcal{A}_X$. An example of such a test is the above mentioned principle of a second-level test, which produces a set of p-values by executing a statistical hypothesis test multiple times. Each p-value is a continuous value in the range [0;1]. In principle, this would yield k to become infinite, and the null-hypothesis as defined in Equation 2.21 would be meaningless. However, this problem can be solved by combining similar data in a fixed number of non-overlapping groups [18]. For instance, p-values might be organized in 10 groups: The first group contains all p-values in the range [0,0.1), the second group contains all p-values in [0.1,0.2), and so on. The number of groups used is dependent on the size of the available set of data and must be chosen with regards to the requirements of the chi-squared distribution approximation, as stated in combination with Equation 2.23. Thus, for each group, the expected frequency must exceed five. The created groups can be considered as elements of the discrete ensemble-alphabet $\mathcal{A}_X = \{a_0, ..., a_{k-1}\}$, where k equals the number of groups. After this "transformation" of the continuous data source, the rest of the chi-squared goodness-of-fit test can be executed as described above.

To illustrate the concept of the chi-squared goodness-of-fit test, it can be of interest to consider an example of a second-level test. Assuming that some kind of statistical hypothesis test has been performed $N = 1000$ times, a set of 1000 p-values has been generated. As mentioned above, this p-values should be uniformly distributed, given that the underlying null-hypothesis is true. As such, it is of interest to evaluate whether or not the resulting p-values can be considered to follow the uniform distribution.

As discussed above, each p-value is a continuous variable. It is therefore necessary to group the values into equally sized, discrete ranges, in order to apply the chi-squared goodness-of-fit test. For this example, it has been chosen to divide the p-values in 10 groups. This yields the following groups: $a_0 = [0, 0.1)$, $a_1 = [0.1, 0.2)$, and so on until $a_9 = [0.9, 1]$. The notation a_j has been chosen to keep the example and the above given discussion consistent, even though a range of continuous values is not a discrete value in the classical sense. However, each group can be thought of as an element of the discrete set $\mathcal{A}_X$, which contains $k = 10$ elements. A complete overview over the groups is given in Table 2.2.

After having established the 10 groups, it is possible to formulate the null-hypothesis, $\mathcal{H}_0$. If the p-values are uniformly distributed, the probability that the p-value of an arbitrary test is placed in one of the 10 groups should be equal for all groups. Thus, by slightly reformulating Equation 2.21, the null-hypothesis for the example can be expressed as, $\mathcal{H}_0 : \Pr(x_i \in a_j) = \frac{1}{k}$, for all possible i and j, where x_i is in this case one of the 1000 p-values.

Using this null-hypothesis and recalling that for this example $N = 1000$, it follows directly from Equation 2.22, that the expected frequency, e_j, equals 100 for each group. The observed frequencies are established by considering the real p-values delivered by the original 1000 test runs. An example of the expected and

observed frequencies is presented in Table 2.2.

Table 2.2: Expected and observed frequencies of p-values for the chi-squared goodness-of-fit test

j	a_j	e_j	o_j
0	[0 , 0.1)	100	110
1	[0.1 , 0.2)	100	89
2	[0.2 , 0.3)	100	119
3	[0.3 , 0.4)	100	111
4	[0.4 , 0.5)	100	88
5	[0.5 , 0.6)	100	96
6	[0.6 , 0.7)	100	87
7	[0.7 , 0.8)	100	91
8	[0.8 , 0.9)	100	115
9	[0.9 , 1]	100	94

The appropriated test statistic, s, for $\mathcal{H}_0$has been presented in Equation 2.23. Under the assumption that $\mathcal{H}_0$is true, s is distributed by a chi-squared distribution with $k-1$ degrees of freedom. In other words, the reference distribution for this example is $\chi^2(s;9)$.

By solving Equation 2.24 for c and using $k=10$ and a level of significance of $\alpha = 0.01$, the critical value can be computed as $c \approx 21.67$. This means that the test rejects $\mathcal{H}_0$if the observed test statistic exceeds $c \approx 21.67$.

Using the in Table 2.2 presented frequencies and Equation 2.23, the observed test statistic can be computed as $s_o = 13.74$. Since $13.74 < 21.67$, this yields the test to not reject $\mathcal{H}_0$at a level of significance of 1%. By solving Equation 2.25 for $s_o = 13.74$, the p-value for the test can be approximated as 0.1319, or roughly 13.2%. It should be noted that the numerical values of this example correspond to the values used in Figure 2.5.

As a result, the chi-squared goodness-of-fit test cannot reject the hypothesis that the values in Table 2.2 follow an uniform distribution. For the second-level test, this means that neither the underlying null-hypothesis of the 1000 originally executed hypothesis tests can be rejected.

2.2.2 The NIST test suite for random number generators

Based on the in Section 2.2.1 presented background on statistical hypothesis testing, it is now possible to discuss the by NIST proposed *statistical test suite for random and pseudorandom number generators for cryptographic applications* [18]. This test suite depicts are frequently used approach to test whether a data stream can be considered to be truly random [19]. Given the complexity of the test suite, it is out of the scope of this report to present the whole approach in detail. This section presents therefore first some general aspects of the test suite and gives then a short summary of the tests involved. Finally, at the end of this section one simple but fundamental test of the test suite is introduced in detail, serving as an example.

Denoting an n-bit data stream as $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$, ideally a test for randomness could clearly evaluate whether or not the data contained in $\overrightarrow{x}$ can be considered as random. It has been stated during the introduction of this section, that such an evaluation has to be based on the statistical characteristics of the data rather than on the data directly [19]. Section 2.1 shows that the two main statistical characteristics that are associated with random data are an uniform distribution of the data and statistical independence. Thus, based on Section 2.2.1 it is reasonable to construct a test for randomness by first using one or both of expected characteristics to define a null-hypothesis and then perform a statistical hypothesis test, following the above described procedure.

However, the fundamental problem of this approach is the above stated fact, that a test of a hypothesis can only reject $\mathcal{H}_0$, but not verify the null-hypothesis. For the null-hypothesis that $\overrightarrow{x}$ contains random data, this means that, while a test can show that $\overrightarrow{x}$ is not random, it is impossible to prove that it in fact contains random bits. Thus, the described test is rather a test for *non* randomness than the desired test for randomness [19].

It should be mentioned that one approach to cope with this problematic, could be to invert the null-hypothesis, which is frequently done in other fields that make use of statistical testing [15, pp. 320-321]. For the here given scenario, $\mathcal{H}_0$ would then state that $\overrightarrow{x}$ is not random and if the test rejects the hypothesis it could be declared that $\overrightarrow{x}$ in fact is random. However, this approach is infeasible due to the fact that, while randomness is associated with well defined statistical characteristics, an infinite number of characteristics can be considered to be non random. For example, random data should be uniform distributed, whereas data that is not random can be distributed in any other possible way. It is therefore not possible to define a reasonable sized set of null-hypothesis that evaluate $\overrightarrow{x}$ for non randomness.

As a result, it is not possible to create a test or a set of tests that could clearly certify that a given data stream $\overrightarrow{x}$ is random [18][19]. Based on this, the NIST test suite has to be considered to use a slightly different approach. It consists of a total of 15 tests. Each test evaluates the null-hypothesis that $\overrightarrow{x}$ is random, by focusing on different aspects of the data for which an expectation under the assumption of randomness exists. If one of the tests reject the null-hypothesis based on the data, it is concluded that $\overrightarrow{x}$ is not random. However, if the test succeeds, that is, $\mathcal{H}_0$is not rejected, the confidence in the null-hypothesis is considered to be increased. Hence, if all 15 test succeed, this fact can be used to argue for the hypothesis that $\overrightarrow{x}$ is random. Nevertheless, it should be stressed, an absolute certain conclusion that $\overrightarrow{x}$ is random cannot be drawn from the NIST test suite.

This means in practice, that in order to increase the confidence in the hypothesis that $\overrightarrow{x}$ is random, it is desirable to increase the significance of the tests by applying a possibly large amount of data. While it is possible to achieve this by simply increasing the length n of the test sequence $\overrightarrow{x}$, NIST proposes to rather use multiple test sequences, $\overrightarrow{x}_0,...,\overrightarrow{x}_{N-1}$, all with the same fixed length n [18].

Using this approach, each of the 15 tests in the test suite is performed N times. For the purpose of this report, this N tests are referred to as *first-level* tests. Following the procedure described in Section 2.2.1, each of the first level-tests results in an observed test statistic, s_o, and a corresponding p-value. For a given (first-level)

level of significance α, the tests can then be evaluated based either s_o or the p-value.

Thus for each of the 15 tests in the suite, N conclusions exist that either indicate that the test has succeeded or that $\mathcal{H}_0$ should be rejected. If all N first-level tests reject the null-hypothesis, it can simply be stated that the test fails and the considered data is not random. However, if the results of the first-level tests for one test of the suite differ, it becomes more difficult to draw a meaningful conclusion. One approach could be to declare the test to have failed if a single first-level test fails. However, recalling from Section 2.2.1 that hypothesis testing yields not to deterministic results but must rather be interpreted in a statistical way, a number of first-level tests is actually expected to fail as N grows large, even if $\mathcal{H}_0$ is true.

Based on this aspect, it is common to unify the N results of the first-level tests by a so called *second-level* approach [18][19], which has been mentioned briefly in Section 2.2.1. It has been stated during the introduction of the p-value in the previous section, that, if a statistical test is performed multiple times, the resulting p-values are expected to be uniform distributed given that $\mathcal{H}_0$ is true. Hence, if it is not possible to evaluate one of the 15 tests of the NIST test suite directly based on the first-level tests, it is reasonable to evaluate if the N p-values can be considered to be uniformly distributed over the range [0,1].

One possible way to perform such an evaluation is by means of the above presented chi-squared goodness-of-fit test. This second-level test results in a single p-value which can be used to evaluate the test for a given second-level level of significance, α'. The second-level test fails if its p-value is smaller then α' and passes otherwise.

Using the second-level approach, the NIST test suite is considered to be passed if the second-level tests of all 15 tests in the test suite succeed. For the purpose of this project, the applied data is then assumed to be random. However, if one or more of the second-level tests is not passed, the NIST test suite fails and the data must be considered to be not random.

Having presented the general testing procedure, it is now possible to consider the 15 tests contained in the NIST test suite. Table 2.3 presents an overview over all tests, including a short description of the focus of the test and the by NIST recommended[8] length of the applied test sequences, n.

It should be noted that some of the in Table 2.3 presented tests are actually performed twice or several times for each applied test sequence, $\overrightarrow{x}$. For example, without going into detail, the Cumulative Sums test scans the test sequence $\overrightarrow{x}$ once starting at x_0 and once starting at x_{n-1}. Thus the test is performed twice for the single sequence $\overrightarrow{x}$. However, it is common to only consider one of these tests per sequence, if a test is performed multiple times [19]. For the purpose of this report, it has been chosen to present the first performed test unless one of the other executed tests fails the second-level test, in which case that test is presented.

[8]For some tests, the recommendation depends on specific test parameters, which are not discussed in this report. In that case the presented recommendation accords to the default test parameters.

Table 2.3: The NIST test suite

Test	Focus	Recommended sequence length (n)
Frequency	Proportion (*frequency*) of 1's & 0's within the test sequence	$n \geq 100$
Block Frequency	Proportion (*frequency*) of 1's & 0's within non-overlapping bit-blocks of the test sequence	$n \geq 100$
Cumulative Sums	Maximal excursion from zero of the cumulative sums of the test sequence	$n \geq 100$
Runs	Total number of uniterrupted sub-sequences (*runs*) of identical bits within the test sequence	$n \geq 100$
Longest Run	Length of the longsest uniterrupted sub-sequence (*run*) of 1's within non-overlapping bit-blocks of the test sequence	$n \geq 128$
Matrix Rank	Rank of disjoint sub-matrices of the test sequence	$n \geq 38,912$
Discrete Fourier Transform (DFT)	Repetitive patterns that yield peaks in the Discrete Fourier Transform of the sequence	$n \geq 1000$
Non-Overlapping Template (NOT)	Number of occurrences of pre-specified, non-overlapping sub-sequences (*templates*) within the test sequence	-
Overlapping Template (OT)	Number of occurrences of pre-specified, overlapping sub-sequences (*templates*) within the test sequence	$n \geq 10^6$
Universal	Number of bits between matching patterns in the test sequence	$n \geq 387,840$
Approximate Entropy	Number of occurrences of all possible overlapping bit-patterns of a specified length across the test sequence	$n \geq 32,768$
Random Excursions	Number of times the cumulative sum of the sequence takes a specific value between two zero occurrences	$n \geq 10^6$
Random Excursions Variant	Number of times the cumulative sum of the sequence takes a specific value	$n \geq 10^6$
Serial	Number of occurrences of all possible overlapping bit-patterns of a specified length across the test sequence	$n \geq 262,144$

Table2.3 – continued

Test	Focus	Recommended sequence length (n)
Linear Complexity	Length of linear feedback shift registers that can be used to characterise non-overlapping sub-sequences of the test sequence	$n \geq 10^6$

As stated at the beginning of this section, it is far beyond the scope of this project to discuss the in Table 2.3 presented tests individually. The interested reader is referred to [18], which gives a detailed introduction to each of the tests. However, in the following the Frequency test is presented in more detail, due to its importance and in order to give a general example of the build up of the tests in the NIST test suite.

The Frequency test

The *Frequency test* is a very simple but nonetheless fundamental test for randomness [18][19]. It focuses on the proportion between the number (or *frequencies*) of 0's and 1's in the data stream under test. As stated in Section 2.1, randomness is associated with an uniform distribution of the bits in the data. The frequency test analyzes therefore whether or not the number of 0's and 1's is approximately equal. It should be noted, that, since the test evaluates an important feature of a random number, while having a low computational complexity, NIST recommends to apply the Frequency test before applying other tests of the test suite. If the results of the Frequency test support the hypothesis of randomness, the test procedure can be continued. Otherwise, the data must be considered to be non-random and it is not necessary to perform the other tests of the test suite.

Denoting the data stream that is to be analyzed by the test as $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$, the null-hypothesis of the test can be formulated as,

$$\mathcal{H}_0 : \Pr(x_i = 0) = \Pr(x_i = 1), \tag{2.26}$$

for all i in $[0, n-1]$.

A number of possible test statistics can be used for an evaluation of Equation 2.26. For the Frequency test, included in the NIST test suite, the test statistic s is defined as [18][9],

[9] Actually, [18] uses the absolute value of Equation 2.27 as the test statistic. Doing so yields to slightly different equations than presented in this report. This is omitted here, in order to keep the notation consistent and to avoid extending the used mathematical background. However, the here presented equations are correct and the final result in Equation 2.30 is identical to the one presented in [18].

$$s = \sum_{i=0}^{n-1} (2 \cdot x_i - 1). \tag{2.27}$$

In other words, s is the number of 1's in $\overrightarrow{x}$ minus the amount of 0's. Obviously, if $\mathcal{H}_0$is true, s is expected to be close to zero. Further, it can be shown that for large values of n (see Table 2.3), the reference distribution of s is the normal distribution centered at $\mu = 0$ and with a standard deviation of $\sigma = \sqrt{n}$ [19]. Thus,

$$\text{pdf}(s; \mathcal{H}_0) = \text{norm}(s; 0, \sqrt{n}), \tag{2.28}$$

where $\text{norm}(s; \mu, \sigma)$ is in this report used to denote the pdf of the normal distribution with mean μ and standard deviation σ.

Combining Equation 2.19 with Equation 2.28, the level of significance for the test can be computed by,

$$\begin{aligned} \alpha &= 2 \cdot \int_{-\infty}^{c_-} \text{norm}(s; 0, \sqrt{n}) \text{d}s \\ &= 2 \cdot \int_{c_+}^{\infty} \text{norm}(s; 0, \sqrt{n}) \text{d}s. \end{aligned} \tag{2.29}$$

As stated in Section 2.2.1, Equation 2.29 can be used to determine the critical values, c_- and c_+, of the test, for a given level of significance, α. It is worth noticing that, since $\text{norm}(s; 0, \sqrt{n})$ is symmetrical around $s = 0$, it holds that $c_- = -c_+$.

For example, setting the level of significance to $\alpha = 0.01$ and analyzing a sequence of $n = 1048576$ bits, the upper critical value can be computed by Equation 2.29 to be $c_+ \approx 2638$. Thus, for the given example, $\mathcal{H}_0$of Equation 2.26 would be rejected if the observed test statistic, s_o, is larger than 2638 or smaller than -2638. In other words, the test yields the conclusion that the considered data is not random, if $|s_o| > 2638$.

Finally, in order to enable a second-level test approach for the Frequency test, it is of interest to calculate the p-value for a given observed test statistic. Combining Equation 2.20 and Equation 2.28, and using the established symmetry of the reference distribution around $s = 0$, the p-value for a given s_o can be computed by[10],

$$\text{p-value} = 2 \cdot \int_{|s_o|}^{\infty} \text{norm}(s; 0, \sqrt{n}) \text{d}s. \tag{2.30}$$

2.3 Random Noise in an ADC

It has been stated in Section 1.2, that TRNGs require a true random source of entropy. As mentioned, a range of example of processes that can be considered to be random exist. However, for the implementation of a TRNG in an MCU environment, the number of possible choices is reduced. For instance, for some random

[10]Solving Equation 2.30, it can be shown that p-value $= \frac{2}{\sqrt{\pi}} \cdot \int_{\frac{|s_o|}{\sqrt{2n}}}^{\infty} \exp(-u^2) \text{d}u$, which is identical to the result presented in [18].

processes it is unreasonable that they can be observed by means of an MCU. An example are user interactions. While technically possible to observe, a wide range of MCU based applications does not directly communicate with (human) users. In addition, it is desirable to use a random process that is embedded inside the MCU. Considering, for example, it is reasonable to assume that most modern MCUs can observe noise introduced by electromagnetic waves by using external radio antennas. However, during the external transmission the gained random data can easily be intercepted by an adversary, in which case the TRNG system must be considered to be broken [22].

In general, the in literature most frequently discussed approaches use thermal noise inside the system; either in form of variations in the voltage level of a signal or by observing so called *jitter* in clock signals [3][23][24]. The latter can be achieved by sampling a fast oscillating clock signal at a lower frequency. Due to the jitter in the fast clock signal, the result of the sampling process varies for different samples. It is worth noticing, that it is quite common to generate the high frequency clock by using ring oscillators, which enables a fully digital implementation of this approach.

However, based on Section 1.3, this project focuses on methods that use the ADC as a source of entropy. A common approach is to observe thermal noise in, for example, resistors or diodes, increase the effect of the noise by amplifying it and then sample it, in order to gain a digital representation [24]. Nevertheless, even though this approach makes use of an ADC, it is not suitable for the purpose of this project, due to two reasons. First, the described setup of an amplified thermal noise source connected to the ADC is in that form generally not embedded in an existing MCU. As such, the in Section 1.3 mentioned design and prototyping costs would occur. Second, it is reasonable to assume that using an amplifier to increase the noise signal yields an significant increase of the dissipated power of the TRNG.

To omit this undesirable consequences, the following section presents some well known noise sources that are associated directly with an ADC, and explores how they can be used to generate an entropy containing output. Based on this background, Section 3.1 shows how an ADC embedded in an MCU can be set up to work as an entropy source.

Before turning the focus on the different types of noise introduced by an ADC, some assumptions about the input signal have to be made. To simplify the following analysis, it is assumed that the input signal is noise-free. Even though this is unlikely in practice, keeping in mind that a noisy input signal is likely to yield a larger amount of entropy in the output signal, this assumption leads to a conservative result compared to the real circumstances. Since for the design of a TRNG a larger amount of entropy is more desirable than a smaller amount, this simplification is acceptable. In addition, for most of the following discussion, the input signal is considered to be constant, which yields to a simplification of the presented analysis. Also this assumption is not necessary true. However, as stated above, it is desirable to use MCU internal signals as an input for the ADC. In practice, most of the internal signals that are available to the MCU are ideally constant, as for example the supply voltage or the reference voltage for the ADC [5]. As such, also this assumption is acceptable for a first analysis of the ADC. Finally, it is assumed that it is known to

an adversary which signal is used as an input. This is a reasonable assumption with regards to *Kerckhoffs's principle*, which is introduced in Section 2.4.

An ADC transforms an analog input signal into a digital representation which uses a fixed number of bits, where the number of bits is called the *resolution* of the ADC [25, p. 614]. Noise generally refers to any unwanted effect that disturbs the transmission or processing of a signal [26, p. 3]. In other words, noise in an ADC yields the output value to differ from the original input value. It is worth noticing that even an ideal ADC would introduce noise by this definition, as the discrete nature of a digital representation makes it impossible to exactly represent a continuous analog signal. This effect is referred to as *quantization noise* [25, pp. 609-612].

However, even though quantization noise depicts a concern for signal processing, it cannot be used as a source of entropy due to two reasons. First, since for the assumed known signal, the mismatch of the analog input and the digital output can be exactly predicted, quantization noise is not random but deterministic. It is worth noticing that this implies that the effect of quantization noise is identical for every ADC device with the same resolution. Second, quantization noise does not vary with time. This means that a constant input signal yields a constant output value. As such, even if one or some of the bits of the first output value would have been unknown, none of the following output values would contain any entropy.

Besides quantization noise, practical ADCs introduce a number of other imperfections which add noise to the output signal. An overview of the most common effects is given in [25, pp. 614-618]. It is out of the scope of this report to give a detailed discussion of all of these effects. However, it is possible to state some general facts: First, any *timing uncertainty* that affects the sampling time of an ADC is irrelevant for the assumed constant input signal. Second, neither *offset errors* nor *gain errors* nor any kind of *nonlinearity* in the quantization process can be used as a source of entropy. This follows from the same argumentation that has been used to disqualify quantization noise as an entropy source. Even though, imperfections in the quantization process differ from device to device and they thus introduce an unpredictable mismatch between the input and the output of the ADC, none of these effects is time varying. As for quantization noise, this yields a constant output stream, when a constant input is applied to the ADC. An illustrative example of this fact is shown in Figure 2.6.

The above mentioned imperfections of the quantization process can be depicted as a static mismatch of the analog to digital conversation, which leave the input signal unchanged. However, the quantizer also introduces noise that can be modeled as directly affecting the input signal. For instance, [27] and [28] identify thermal noise in the comparator (which is a part of the quantizer) of ADCs which use a *Successive Approximation Register* (SAR) as the limiting factor of the achievable resolution of the ADC. In other words, SAR ADCs introduce thermal noise, which can be referred to the input of the quantizer and as such affects the input signal directly. It is reasonable to assume that this is not only true for SAR ADCs but for any type of ADC, since thermal noise is one of the dominating noise sources in electronic systems [26, p. 58].

Ignoring all irrelevant noise sources and only focusing on thermal noise, an ADC

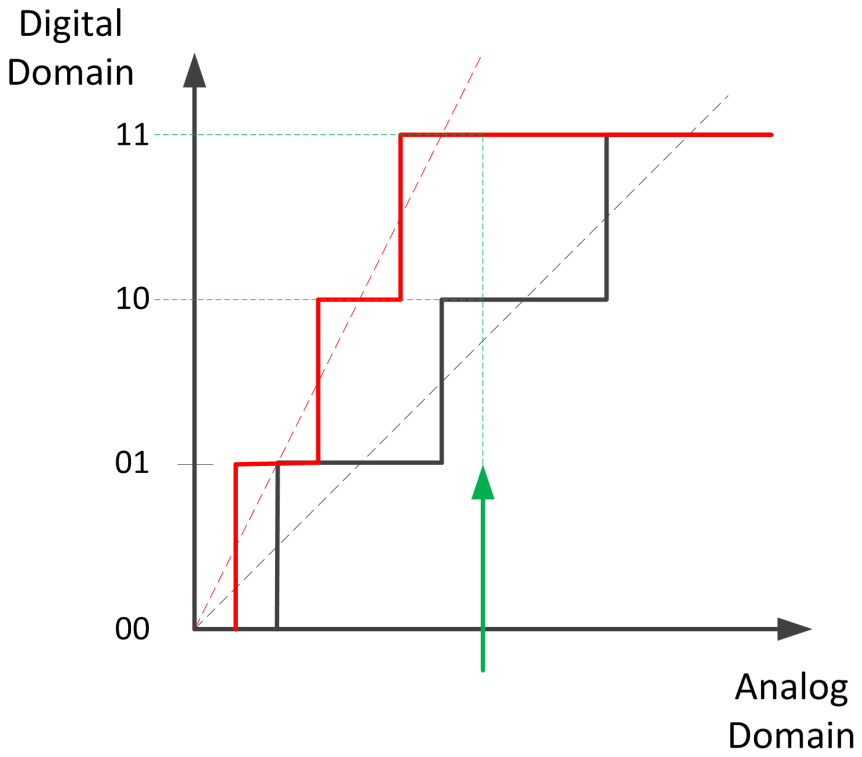


Figure 2.6: Example of a quantization imperfection: *The black "staircase" depicts the ideal quantization function. The red "staircase" illustrates a gain error. The same analog input signal (depicted by the green arrow) results in two different digital output values. However, if the analog input signal is constant, the digital output value is as well constant.*

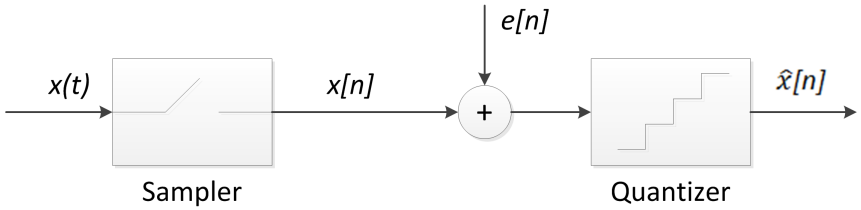


Figure 2.7: Model of an ADC with thermal noise

can for the purpose of this project be modeled as depicted in Figure 2.7. Here $x(t)$ is the analog input signal and $x[n]$ the corresponding time sampled version[11]. The time distinct thermal noise signal is marked $e[n]$ and $\hat{x}[n]$ is the quantized value of the noise contaminated version of $x[n]$, i.e., the digital representation of $x[n] + e[n]$. It is important to note that both the sampler and the quantizer are ideal in this model.

Thermal noise is a well known phenomena, which is commonly modeled as a normally distributed process with zero mean which is independent over time [26, p. 60]. The variance of the distribution depends on the power associated with the noise source, which varies with, for example, temperature and the ADC architecture. Using this model of thermal noise and assuming that $x(t)$ is constant and noise-free, it is possible to show that the ADC introduces entropy to the data. Considering first $x[n]$ without $e[n]$, the quantization process transforms $x[n]$ into a constant digital value $\hat{x}[n]$. The particular value of $\hat{x}[n]$ is not of interest, but for the purpose of illustration it can be assumed, without loss of generality, that the *Least Significant Bit* (LSB) of $\hat{x}[n]$ is 1. As a result, defining $\overrightarrow{x}_{LSB}$ as a vector containing the LSBs of a number of consecutive samples $\hat{x}[n]$, $\overrightarrow{x}_{LSB}$ would consists of only 1s. For example, 8 samples of $x[n]$ would result in $\overrightarrow{x}_{LSB} = \langle 1, 1, 1, 1, 1, 1, 1, 1 \rangle$.

However, since $e[n]$ is normally distributed around zero, the sum of $x[n] + e[n]$ is normally distributed around the constant value $x[n]$. As a result, some values of $e[n]$ yield a quantization error, as the difference between $x[n]$ and the sum $x[n]+e[n]$ becomes too large. In that case, the LSB of $\hat{x}[n]$ changes from 1 to 0. This is depicted in Figure 2.8. By reusing the example stated above, 8 samples of $x[n]$ might under the influence of $e[n]$, for instance, result in $\overrightarrow{x}_{LSB} = \langle 1, 1, 1, 0, 1, 1, 0, 1 \rangle$. Since $e[n]$ is independent and identically distributed, this is also true for the elements contained in $\overrightarrow{x}_{LSB}$. As such, the thermal noise of inside the ADC introduces random variations in the data. In other words, the output $\overrightarrow{x}_{LSB}$ contains some amount of entropy. Nevertheless, it is reasonable to assume that the contained entropy is rather low and that $\overrightarrow{x}_{LSB}$ is, for the given example, biased towards 1. It is easy to see the validity of this assumption by considering that ADCs in general are designed in order to achieve an output that is as exact as possible and not too generate data for which the LSB is close to random.

The above discussed illustration of the concept of thermal noise in an ADC is of course an ideal model. In practice, usually other aspects have to be taken into

[11]The variables t and n are in this section, as common in digital signal processing, used to distinguish between signals with continuous and discrete time characteristics. As such, n is for the reminder of this section *not* used as the fixed length of a data stream.

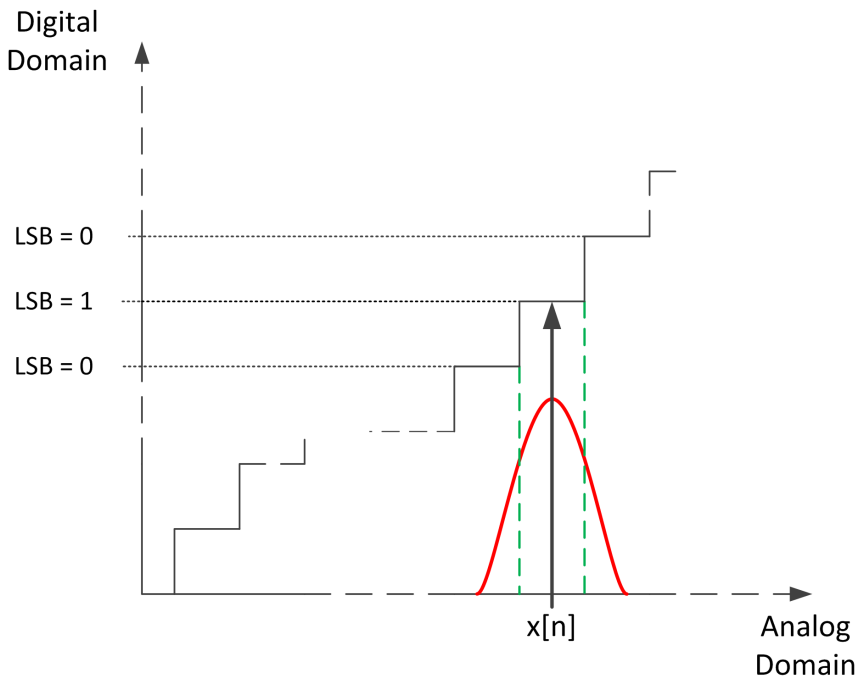


Figure 2.8: ADC transformation with thermal noise: *The constant input $x[n]$, depicted as a black arrow, yields an LSB of 1. The distribution of the thermal noise signal $e[n]$ is illustrated as a red curve. The green dotted lines indicate the boarders for a quantization error.*

consideration. For example, other noise types inside the ADC or in surrounding circuitry may affect the sampling process. In addition, it has been assumed that the input signal is strictly constant. This is not necessarily true in a realistic scenario, either because a varying signal has been chosen as an input by design, or because the chosen input signal varies due to imperfections.

However, even in the presence of these or other effects, the above presented concept is still valid and thermal noise will randomly affect the output data. Further more, the mentioned influences can actually increase the amount of entropy associated with the ADC output, if they can be considered to be random. The main drawback of this variety of possible influences is that their character must be considered to be unknown. As such, they can, for example, introduce undesired dependencies into the output data.

2.4 Post-processing algorithms

It has been mentioned in Section 1.2 that most TRNGs use some kind of digital post-processing step. While their purpose may vary, they are frequently used to mask statistical imperfections of the entropy source. For example, Section 2.3 illustrates how an ADC can generate an entropy containing but biased bit stream. Throughout this project, post-processing algorithms are mainly considered with regards to this feature, leaving other aspects, as, for example, resistance against attacks of an

adversary, to future work.

Before, presenting some known post-processing algorithms, it is essential to note a common and majorly important fact. Post-processing algorithms are in general clearly defined and exactly described processes and, as such, perfectly deterministic. It has already been stated in Section 1.2 that generating any kind of entropy by means of a deterministic system is impossible. It is fairly easy to convince oneself of this fact by considering the extreme case, in which the input stream to a post-processor, denoted as $\overrightarrow{x}$, is completely known to an adversary. In that case, the adversary could simply execute the same operations on $\overrightarrow{x}$ as performed by the post-processor and thus derive the same output stream, described by $\overrightarrow{y}$. This argumentation is easily extended to a scenario in which $\overrightarrow{x}$ contains some amount of entropy. Thus, the degree of uncertainty of the adversary can not be increased by means of post-processing.

Obviously, the above presented argumentation is only valid in the case that the adversary knows which post-processing algorithm is used. However, this assumption is highly convenient. It is generally assumed in modern cryptography that an adversary has detailed knowledge of the methods used in the system she wants to attack. This concept is referred to as *Kerckhoffs's principle* [1, p. 4] and includes TRNGs.

As a result, for any input $\overrightarrow{x}$ and the corresponding output $\overrightarrow{y}$ it must hold,

$$\mathrm{H}(\overrightarrow{x}) \geq \mathrm{H}(\overrightarrow{y}). \tag{2.31}$$

This means that the total amount of entropy contained in $\overrightarrow{y}$ cannot exceed the amount of entropy associated with $\overrightarrow{x}$. However, the amount of entropy per bit, called the *entropy rate*, can be increased. It follows directly from Equation 2.31, that an increase in the entropy rate yields $\overrightarrow{y}$ to contain less bits than $\overrightarrow{x}$. For example, for a 128-bit input $\overrightarrow{x}$ associated with 32 bits of entropy, a ideal post-processor would generate a 32-bit output with 32 bits of entropy. Thus, the entropy rate would be increased from 0.25 to the maximum value of 1. As such, TRNG post-processing algorithms can be classified as *compressive algorithms*. Another term frequently used to describe TRNG post-processing is *randomness extractor* [14].

The related literature generally refers to three basic types of post-processing algorithms: the *von Neumann Corrector* (VNC), *cryptographic hash functions* and *resilient functions* [10][14][29]. Additional approach are *block ciphers*, which are well known cryptographic algorithms and often mentioned alongside cryptographic hash functions [14][30], and an *Extractor based on pairwise Independent Hash Functions* (IHF) [14]. In a this project preceding literature study, all five approaches have been considered and discussed [12]. Based on this discussion, the VNC and the IHF have been selected for further investigation. Without going into detail here, two main reasons motivated this decision. First, in contrast to cryptographic approaches by means of hash functions or block ciphers, both the VNC and the IHF are based on firm mathematical proofs. In principle, this enables the selection of a suitable post-processing algorithm based on an analysis of the entropy source and the establishment of clearly defined boundaries in which the functionality of the TRNG is guaranteed. Second, the VNC and especially the IHF have relaxed requirements

on the statistical characteristics of the used entropy source compared to the strict requirements of resilient functions.

In the following, the VNC is presented in Section 2.4.1 and Section 2.4.2 introduces the concept of the IHF. As they are not a part of the focus of this project, a explicit presentation of both resilient functions and cryptographic approaches is omitted.

2.4.1 Von Neumann corrector

The *von Neumann Corrector* (VNC) is based on a mathematical observation formulated by John von Neumann [8] regarding a biased bit stream, in which succeeding bits are independent and *identically distributed*, which means that, for instance, the probability of observing 0 is equal for both bits. The VNC provides a simple but effective solution to transform such a stream into a perfectly unbiased bit stream.

Defining the input stream to the VNC as a binary vector with n elements, $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$, and the output stream in the same manner as the m-bit binary vector $\overrightarrow{y} = \langle y_0, ..., y_{m-1} \rangle$, the VNC can be described by Algorithm 1 [31].

Algorithm 1 The von Neumann algorithm

```
Input: x⃗ = ⟨x_0, ..., x_{n-1}⟩
Initialize: Empty output vector y⃗ = ⟨⟩
for all i in [0, ⌊n/2⌋ − 1] do
  if x_{2i} xor x_{2i+1} then
    Insert x_{2i} at the end of y⃗
  end if
end for
```

It can be seen from Algorithm 1, that the VNC operates on pairs of succeeding bits, x_{2i} and x_{2i+1}. If these bits are not identical, the XOR-operation results in a logic one. In that case, x_{2i} is inserted at the end of $\overrightarrow{y}$. However, if x_{2i} and x_{2i+1} are equal, no element is added to $\overrightarrow{y}$. As an example, the 8-bit input stream $\overrightarrow{x} = \langle 1, 1, 0, 1, 0, 0, 1, 0 \rangle$ would result in the 2-bit output $\overrightarrow{y} = \langle 0, 1 \rangle$.

Verifying that Algorithm 1 is capable of transforming a biased input stream into an unbiased output can be done by considering Table 2.4. The table presents the truth table of an XOR-operation with input bits x_{2i} and x_{2i+1}. It further shows the probabilities for the possible input combinations, $\Pr(x_{2i}, x_{2i+1})$, and the resulting output y_j. Since it is required that x_{2i} and x_{2i+1} are identically distributed, the marginal probabilities of x_{2i} and x_{2i+1} can be expressed as $p = \Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0)$. Furthermore, due to their independence, the probability of the input combinations can be expressed as a product of the marginal probabilities of the corresponding input bits (Equation 2.9). As an example, $\Pr(x_{2i} = 0, x_{2i+1} = 0) = \Pr(x_{2i} = 0) \cdot \Pr(x_{2i+1} = 0) = p^2$.

It can be seen from Table 2.4 that $\Pr(y_j = 0) = \Pr(y_j = 1) = p \cdot (1 - p)$. In other words, 0s and 1s are inserted into $\overrightarrow{y}$ with the same probability, and $\overrightarrow{y}$ is thus unbiased.

Table 2.4: Extended truth table of the XOR-operation in a VNC

x_{2i}	x_{2i+1}	$x_{2i} \oplus x_{2i+1}$	$\Pr(x_{2i}, x_{2i+1})$	y_j
0	0	0	p^2	-
0	1	1	$p \cdot (1-p)$	0
1	0	1	$p \cdot (1-p)$	1
1	1	0	$(1-p)^2$	-

However, it should be stressed that Table 2.4 is only valid, if the data in $\overrightarrow{x}$ satisfies the mentioned statistical requirements. This requirements are of great importance for the purpose of this project and are therefore repeated here:

- The successive bits x_{2i} and x_{2i+1} must be statistical independent.
- The successive bits x_{2i} and x_{2i+1} must be identically distributed. This means that $p = \Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0)$.

These characteristics of the input stream $\overrightarrow{x}$ are for the purpose of this report referred to as the *von Neumann conditions*.

Given the importance of the conditions, it is worth to consider two aspects that are closely related and easily confused with the von Neumann conditions, but *not* necessary for a functional performance of the VNC. With regards to the identical distribution of x_{2i} and x_{2i+1}, p does not have to be constant for all bits in $\overrightarrow{x}$. For example, considering four successive bits, x_0, x_1, x_2 and x_3, it is sufficient if the bit pairs $\langle x_0, x_1 \rangle$ and $\langle x_2, x_3 \rangle$ are identically distributed, that is, $p_0 = \Pr(x_0 = 0) = \Pr(x_1 = 0)$ and $p_1 = \Pr(x_2 = 0) = \Pr(x_3 = 0)$. It follows from Table 2.4 that the probabilities of inserting 0 into $\overrightarrow{y}$ are $p_0 \cdot (1 - p_0)$ and $p_1 \cdot (1 - p_1)$, respectively, which are possibly different. Nevertheless, the probability that a specific bit-pair yields an insertion of 0 into $\overrightarrow{y}$ is still equal to the probability that 1 is inserted for the given pair. Hence, the VNC still works as desired. This means that p is allowed to vary between bits in the input stream, as long as $p = \Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0)$ is satisfied.

Considering the statement about statistical independence between x_{2i} and x_{2i+1}, it has been mentioned during the discussion of Table 2.4 that this is a necessary characteristic since it allows the multiplication of the marginal probabilities of x_{2i} and x_{2i+1} to the joint probability $\Pr(x_{2i}, x_{2i+1})$. However, the table states nothing about other forms of dependencies in $\overrightarrow{x}$. Thus as long as the used multiplication is valid, the VNC functions as specified, even if dependencies in the data exist. Nevertheless, it should be stressed that the VNC is purely designed to remove the bias of an input stream but not possible dependencies. Hence, if dependencies exist in the input data, it is reasonable to assume that they as well exist in the output $\overrightarrow{y}$. With respect to Equation 2.17, which shows that the maximum amount of entropy for a bit stream is associated with both unbiased and independent data, it is therefore preferable if the data contained in the VNC-input is mutual independent, in order to result in a true random output.

A final aspect of the VNC that should be considered is the ratio of the number of input bits to the number of output bits. With respect to Algorithm 1, it is obvious that a single output bit is generated from two input bits. As such, the VNC requires at least two input bits per single output bit. However, this is a rather unlikely scenario. Assuming, for instance, that the input $\overrightarrow{x}$ is unbiased[12], it is expected that all four possible constellations of the bit pair $\langle x_{2i}, x_{2i+1} \rangle$ in Table 2.4 occur with the same probability. Thus, for an unbiased input, the expected number of input bits per single output bit is 4 bits.

In general, it can be stated, that the ratio of the number of input bits n to the number of output bits m, is undeterministic and the expected value is dependent on the bias of the input, characterized by p. A more detailed analysis of this aspect is presented in Section 4.1.1.

2.4.2 Extractor based on pairwise-independent hash functions

An approach of an *Extractor based on pairwise Independent Hash Functions* (IHF) is presented in [14][13]. The IHF can be proven to deliver a random output from any source which provides a sufficient amount of min-entropy. It is essential to note that this in principle also includes sources that produce an output with some kind of dependencies.

Compared to the simple VNC of Section 2.4.1, the concept of the IHF must be considered as more complicated. In order to give an as simple as possible introduction of the IHF, this section is structured in the following manner: First, a description of a hypothetical randomness extractor is presented, that works for an output with a given amount of min-entropy and allows an adversary to have some limited influence on the source. Second, the concept of pairwise independent hash functions is introduced and related to the described randomness extractor. Finally, two implementations of the IHF are presented; the first one is based on linear functions in a *Galois field* and the second one uses a binary matrix.

Starting describing the desired extractor, it is appropriated to first consider a scenario in which an adversary has some control over the entropy source, and as such, over the input of the extractor, denoted as x. To achieve this, x is, for the moment, viewed as the n-bit outcome of one out of a number of possible ensembles, $X_0, ..., X_{2^t-1}$. While the sets of the ensembles, $\mathcal{A}_X = \{0,1\}^n$, are identically, the distributions, $\mathcal{P}_0, ..., \mathcal{P}_{2^t-1}$, are different for each of the ensembles. However, for each $i \in \{0, ..., 2^t - 1\}$, it must hold that min-Ent$(X_i) > \kappa$. By the definition of min-entropy, given in Equation 2.3, this means that no element of $\mathcal{P}_i$ can exceed $2^{-\kappa}$.

Next, a *public parameter* π is chosen randomly and independently of $X_0, ..., X_{2^t-1}$ from a (for the moment undefined) set Π. It is important to note two major char-

[12]This is of course an example with low practical relevance, as for an unbiased $\overrightarrow{x}$, applying the VNC has no recognizable benefits.

[13]It should be noted that [14] presents a general discussion of extractors based on l-wise independent hash functions. However, the proposed algorithms use $l = 2$ and the here given presentation is therefore restricted to the discussion of pairwise independent hash functions.

acteristics of π. First, π is public and thus known by any adversary. Second, it is sufficient to chose π once and it is therefore not necessary to renew π at any given point of time.

Finally, an adversary, with the knowledge of π, is allowed to chose an arbitrary ensemble $X_i \in \{X_0, ..., X_{2^t-1}\}$. The outcome of X_i is then passed on to the extractor in form of the input x.

Based on this scenario, the desired extractor can be defined as the function,

$$f(x, \pi) = y, \tag{2.32}$$

where y is the m-bit outcome of the ensemble Y, with the set $\mathcal{A}_Y = \{0,1\}^m$ and the distribution $\mathcal{P}_Y$. It follows from the observations made with regards to Equation 2.31, that $m \leq n$, in order for Equation 2.32 to be a valid description of an entropy extractor.

Given Equation 2.32, [14] defines $f(x, \pi)$ to be "t-resilient[14] if [...] with probability $1 - \epsilon$ over the choice of the public parameter the statistical distance between [$\mathcal{P}_Y$ and the uniform distribution over the set $\mathcal{A}_Y$] is at most ϵ". The definition of statistical distance has been presented in Equation 2.6. The parameter ϵ can be considered as a measure of the quality of the extractor $f(x, \pi)$; the smaller ϵ, the more likely is the fact that $\mathcal{P}_Y$ is ϵ-close to a uniform distribution. In other words, for $f(x, \pi) = y$ with an randomly chosen π, it holds that,

$$\Pr(\text{dist}(\mathcal{P}_Y, \mathcal{U}_Y) \leq \epsilon) = 1 - \epsilon, \tag{2.33}$$

where $\mathcal{U}_Y$ denotes the uniform distribution over the set $\mathcal{A}_Y = \{0,1\}^m$. Obviously, for a good extractor it is desirable that ϵ is as small as possible. Motivated by this observation, ϵ is for the purpose of this report also frequently described as the *quality parameter* of the IHF.

The above given description of the t-resilient extractor is rather abstract. To illustrate the concept, it may be helpful to consider the following simple example. Assuming a perfectly random source with a single 3-bit outcome x, the source can be described as the ensemble $X = \{x, \mathcal{A}_X, \mathcal{U}_X\}$, where $\mathcal{A}_X = \{0,1\}^3$ and $\mathcal{U}_X$ is the uniform distribution over $\mathcal{A}_X$. This means that x can take on 8 different values with the same probability, $1/8$. This is depicted in Table 2.5, where $\Pr(x|x \in X)$ has been used to denote the probability of the corresponding x-value in the case that x is the outcome of X. It follows from Equation 2.3 that the min-entropy of the source is min-Ent(X) = 3.

To show how the extractor can cope with a limited influence of an adversary, it is now assumed that an adversary is able to fix the middle bit of x either to 0 or to 1. The resulting, manipulated versions of X are $X_0 = \{x, \mathcal{A}_X, \mathcal{P}_0\}$ and $X_1 = \{x, \mathcal{A}_X, \mathcal{P}_1\}$, respectively. The effect of the manipulation on the distributions $\mathcal{P}_0$ and $\mathcal{P}_1$ are shown in Table 2.5. Both X_0 and X_1 contain 2 bit of min-entropy.

[14]For the sake of completeness, it should be mentioned that the here given definition of resilience is not equivalent to the definition of the same term used to describe resilient functions (see the introduction of Section 2.4), even though similarities exist [29]. Since resilient functions are not part of the focus of this project, the term "(t-)resilient" is throughout this report used according to the definition of [14], which is repeated here.

Table 2.5: Output probabilities of the mainpluated ensemble X

x	$\Pr(x\|x \in X)$	$\Pr(x\|x \in X_0)$	$\Pr(x\|x \in X_1)$
000	$1/8$	$1/4$	0
001	$1/8$	$1/4$	0
010	$1/8$	0	$1/4$
011	$1/8$	0	$1/4$
100	$1/8$	$1/4$	0
101	$1/8$	$1/4$	0
110	$1/8$	0	$1/4$
111	$1/8$	0	$1/4$

Thus based on the choice of the adversary, the input to the extractor, x, arises from one of two possible ensembles: X_0 or X_1. As such, the adversary freely "chooses" an ensemble, which appears to be less random compared to original X, out of the set $\{X_0, X_1\}$. It is important to note two main aspects of this choice. First, the adversary is allowed to make her choice based on the knowledge of the used extractor $f(x, \pi)$, including the concrete value of the public parameter, π. Second, which choice the adversary makes remains unknown to the extractor. In other words, the extractor cannot adapt to the given choice by, for example, changing its public parameter.

With regards to the given example, the extractor $f(x, \pi) = y$ is said to be t-resilient with $t = 1$, if y can be considered to be the outcome of the ensemble $Y = \{y, \mathcal{A}_Y, \mathcal{P}_Y\}$, where $\mathcal{P}_Y$ satisfies the conditions described in Equation 2.33. Whether x arises from X_0 or X_1, in both cases x contains 2 bit of min-entropy. Thus, for reasonable small values of ϵ it follows from Equation 2.31 that the size of y, m, cannot exceed 2 bits, for the given example.

It is shown in the above example, that the parameter t describes the number of manipulations of the source by an adversary that the extractor can handle. It is reasonable to assume that the character of the manipulations is binary and that t manipulations thus result in 2^t different ensembles. As such, t can be considered to characterize the security performance of the extractor $f(x, \pi)$ and is therefore in this report referred to as the *security parameter* of the IHF.

Before moving on to find possible realizations of a t-resilient extractor, it must be stressed that the discussed example is extremely simple and only used for an illustrative purpose. It is essential to note, that the nature of the adversary's attack has not been specified. She may, as illustrated in the example, try to fix single bits of the source outcome. However, by the above presented formal description of the extractor, the adversary is not limited to such an attack. Again, the demand stated by the described extractor is that min-Ent$(X_i) > \kappa$ for all 2^t possible ensembles. In the following, it is shown how the required min-entropy, κ, relates to the length of the output of the extractor, m, the security parameter, t, and the quality parameter, ϵ.

It is now of interest to find a group of functions, that meet the requirements of the above described extractor, $f(x, \pi)$. As indicated during the introduction of

this section, this can be achieved by means of pairwise independent hash functions. Considering an n-bit input x ($x \in \{0,1\}^n$) and an m-bit output y ($y \in \{0,1\}^m$), the set of hash functions $H = \{h(x,\pi) : \pi \in \Pi\}$ is defined such that $h(x,\pi) = y$. H is said to be a pairwise independent hash function, if for an arbitrary but fixed x and a at random chosen π, the output y is uniformly distributed over $\{0,1\}^m$ and pairwise independent [14]. The definition of pairwise independence is given in Equation 2.14 for $l = 2$.

Given now that x is the outcome of one of the ensembles $X_0, ..., X_{2^t-1}$ and that for each ensemble min-Ent$(X_i) > \kappa$, it is shown in [14] that for at least a $1-\epsilon$ fraction of Π, the output of $h(x,\pi)$ is ϵ-close to uniform. In other words, by randomly selecting π from Π, the probability that the distribution of the m-bit output $y = h(x,\pi)$ is ϵ close to the uniform distribution over $\{0,1\}^m$ is $1-\epsilon$. From Equation 2.33, it follows therefore that a function that is a member of the pairwise independent family of hash functions H can be used as a t-resilient extractor.

Using a pairwise independent hash function, $h(x,\pi)$, to realize the t-resilient extractor $f(x,\pi)$, the different parameters of the extractor are related by [14],

$$t = \frac{\kappa - m}{2} - 2 \cdot \log_2(\frac{1}{\epsilon}) - 1, \tag{2.34}$$

where κ is the amount of min-entropy delivered by the input, m is the length of the output of the extractor, ϵ is the quality parameter of the extractor and t the security parameter. It is worth noticing that Equation 2.34 enables the extractor to be tuned in order to match different environments. For example, by increasing the amount of entropy delivered to the extractor, the security parameter can be increased. In the same manner, the security parameter can be increased by keeping κ at a fixed level and reduce the number of output bits.

Before moving on to consider concrete approaches to realize pairwise independent hash functions, it is worth to notice two more general aspects of the resilient extractor. Up to this point, it has only be assumed that a single n-bit input x is used to derive a single m-bit output. It is possible to derive a longer output by running the extractor for several different inputs and concatenate the outputs to an output with a length that is a multiple of m. However, to satisfy the principle that has been stated with regards to Equation 2.10, each input x must contain at least κ bits of *conditional min-entropy* with respect to the other inputs in order for the extractor to work as desired. Conditional min-entropy is defined analogously to the conditional entropy in Equation 2.11.

In the same manner, also the input x can be generated by the concatenation of shorter inputs as long as the associated min-entropy exceeds the minimum of κ. For example, using the notation introduced in Section 2.1, a source that produces a single bit at a time can be used to generate an input for the IHF by concatenating these bits to $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$, where each element in $\overrightarrow{x}$ is a single bit, and then passing $\overrightarrow{x}$ as an input to Equation 2.32.

To complete the discussion of the IHF, the following presents two approaches to realize such an extractor. The first one is based on linear functions in a Galois field. The second one can be considered to make use of a binary matrix.

IHF implementation using linear functions in a Galois field

One possible approach of realizing pairwise independent hash functions is by means of linear functions in a *Galois field* $GF(2^n)$ [14]. Galois fields are a rather complicated topic of number theory and far beyond the scope of this project[15]. However, for the purpose of this project, it is sufficient to think of $GF(2^n)$ as a set of size 2^n containing all the polynomials $g_i(z)$ of degree $n-1$ such that,

$$g_i(z) = c_0 + c_1 \cdot z + c_2 \cdot z^2 + ... + c_{n-1} \cdot z^{n-1}, \tag{2.35}$$

where all coefficients $c_0, ..., c_{n-1}$ are in modulo 2. Further, a polynomial $G(z)$ of degree n has to be selected for $GF(2^n)$, such that it can not be represented by the multiplication of any two polynomials $g_i(z)$ and $g_j(z)$ in $GF(2^K)$. A polynomial $G(z)$ that meets this requirement is said to be *irreducible*. It is possible to show that as long as the addition, subtraction, multiplication and division of two polynomials in $GF(2^n)$ is executed modulo $G(z)$, the result is as well a polynomial of the form specified in Equation 2.35 and, as such, an element of $GF(2^n)$ [9, p. 255].

In order to relate $GF(2^n)$ to the above presented extractor, it is essential to note that any n-bit number can be represented by means of a polynomial $g_i(z)$. This can be illustrate by the example of the extractor input x. Up to this point, the n-bit input has been denoted as x for simplicity. However, in order to increase the consistency between the notations used to describe the different post-processing algorithms, the input x may as well be described as $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$, where each bit of x is one element of $\overrightarrow{x}$. This notation appears especially appropriated if x has been created by the concatenation of n single bits, as discussed above. Using this vector notation, $\overrightarrow{x}$ can be described as an element of $GF(2^n)$ by the polynomial $g_x(z)$, defined as,

$$g_x(z) = x_0 + x_1 \cdot z + ... + x_{n-1} \cdot z^{n-1}. \tag{2.36}$$

Choosing π randomly from $\Pi = \{0,1\}^n$, the vector notation can be adopted, such that the public parameter is denoted as $\overrightarrow{\pi} = \langle \pi_0, ..., \pi_{n-1} \rangle$, where each bit of π is an element in $\overrightarrow{\pi}$. In accordance to Equation 2.36, $\overrightarrow{\pi}$ can be expressed by the polynomial $g_\pi(z)$, given as,

$$g_\pi(z) = \pi_0 + \pi_1 \cdot z + ... + \pi_{n-1} \cdot z^{n-1}. \tag{2.37}$$

Using Equation 2.36 and Equation 2.37, a set of pairwise independent hash functions can be created by using the hash functions defined as,

$$\overrightarrow{y} = h(\overrightarrow{x}, \overrightarrow{\pi}) \equiv (g_\pi(z) \cdot g_x(z))_{0,...,m-1} (\mod G(z)), \tag{2.38}$$

where the vector notation $\overrightarrow{y} = \langle y_0, ...y_{m-1} \rangle$ has been adopted for the output of the hash function, y, in the same way as for x and π, and the notation $(\cdot)_{0,...,m-1}$ is used to describe that only the m first bits of the result are used for the output $\overrightarrow{y}$.

Implementing Equation 2.38 is rather simple, since addition, subtraction and multiplication are easily executed in $GF(2^n)$. Since all coefficients are modulo 2, both

[15] For a basic introduction, the interested reader is referred to [1, pp. 93-101].

additions and subtractions are executed by a bit-wise XOR-operation. To give a simple example, the 8-bit Galois field $GF(2^8)$, defined by the well known irreducible polynomial $G(z) = 1 + z + z^3 + z^4 + z^8$, is considered. Picking the arbitrary polynomials $g_1(z) = 1 + z + z^3 + z^6 + z^7$ and $g_2(z) = 1 + z^3 + z^4$, the result of the addition is $g_1(z) + g_2(z) = z + z^4 + z^6 + z^7$. It is worth noticing that the result is as expected a member of $GF(2^8)$ and that subtracting the polynomials from each other would yield the same result.

Also the multiplication of any element of $GF(2^n)$ by z is rather simple. By considering Equation 2.35, it follows that,

$$g_i(z) \cdot z = 0 + c_0 \cdot z + ... + c_{n-2} \cdot z^{n-1} + c_{n-1} \cdot z^n. \tag{2.39}$$

This means that the multiplication with z can be performed by a simple shift operation. However, if $c_{n-1} = 1$, the result as shown in Equation 2.39 is no longer a member of $GF(2^n)$. In that case it is therefore necessary to change the result to modulo $G(z)$ by subtracting the irreducible polynomial $G(z)$ from the result of the multiplication.

To give an example, the above introduced Galois field $GF(2^n)$ is reused with the same irreducible polynomial $G(z)$. For this field, multiplying $g_1(z)$ by z would yield,

$$\begin{aligned} g_1(z) \cdot z &= (1 + z + z^3 + z^6 + z^7) \cdot z = z + z^2 + z^4 + z^7 + z^8 \\ &= (1 + z^2 + z^3 + z^7) + (1 + z + z^3 + z^4 + z^8) \\ &\equiv 1 + z^2 + z^3 + z^7 (\mod 1 + z + z^3 + z^4 + z^8). \end{aligned} \tag{2.40}$$

Based on this, it is possible to multiply any polynomial $g_i(z)$ by any arbitrary power z^j, by simply multiplying $g_i(z)$ j-times with z following the above introduced procedure. For the given example, $g_1(z) \cdot z^2$ could be computed by,

$$\begin{aligned} g_1(z) \cdot z^2 &= (1 + z + z^3 + z^6 + z^7) \cdot z^2 \equiv (1 + z^2 + z^3 + z^7) \cdot z \\ &= z + z^3 + z^4 + z^8 = 1 + (1 + z + z^3 + z^4 + z^8) \\ &\equiv 1 (\mod 1 + z + z^3 + z^4 + z^8), \end{aligned} \tag{2.41}$$

where the result of Equation 2.40 has been reused.

Finally, since the polynomials in $GF(2^n)$ have binary coefficients, any polynomial $g_1(z)$ can be multiplied by any arbitrary other polynomial in $GF(2^n)$, $g_j(z)$, by separately multiplying $g_i(z)$ with any power of z that corresponds to a non-zero coefficient of $g_j(z)$ and then adding the results. To illustrate this, the above given example is considered once more. Using the results of Equation 2.40 and Equation 2.41, multiplying $g_1(z)$ with $g_3(z) = z + z^2$ yields,

$$\begin{aligned} g_1(z) \cdot g_3(z) &= (1 + z + z^3 + z^6 + z^7) \cdot (z + z^2) \\ &= (1 + z + z^3 + z^6 + z^7) \cdot z + (1 + z + z^3 + z^6 + z^7) \cdot z^2 \\ &\equiv (1 + z^2 + z^3 + z^7) + 1 \\ &\equiv z^2 + z^3 + z^7 (\mod 1 + z + z^3 + z^4 + z^8). \end{aligned} \tag{2.42}$$

To give an example of the in Equation 2.38 defined pairwise independent hash function based on the example presented in Equation 2.42, it is possible to consider $g_1(z)$ as the polynomial representation of the input $\overrightarrow{x} = \langle 1, 1, 0, 1, 0, 0, 1, 1 \rangle$, and $g_3(z)$ to be the public parameter $\overrightarrow{\pi} = \langle 0, 1, 1, 0, 0, 0, 0, 0 \rangle$. The result of Equation 2.42 in vector form is $\langle 0, 0, 1, 1, 0, 0, 0, 1 \rangle$. Using, for instance, $m = 2$ as the output length, the 2-bit output of the has function would be $\overrightarrow{y} = \langle 0, 0 \rangle$.

Having established the methodology to multiply two polynomials of $GF(2^n)$, it is possible to derive an algorithmic solution of the IHF as described by Equation 2.38. One possibility is depicted in Algorithm 2. To keep the notation compatible that of Algorithm 1, a vector notation has been used instead of polynomials (by reversing the statement of, for instance, Equation 2.37). This means that $\overrightarrow{\pi}$ contains the coefficients of the public parameter g_π and $\overrightarrow{G}$ contains the coefficients of the irreducible polynomial $G(z)$. It is worth noticing that the coefficient corresponding to the highest degree in $G(z)$ is always one and irrelevant for the algorithm. $\overrightarrow{G}$ is therefore only of length n instead of $n + 1$, in order to avoid confusion during the bit-wise XOR-operation that involves $\overrightarrow{G}$. The notation $\overrightarrow{x'} >> 1$ in Algorithm 2 is used to denote a one bit right shift of $\overrightarrow{x'}$.

Algorithm 2 IHF solution using linear functions in $GF(2^n)$

Input: $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$
Initialize: all-zero output: $\overrightarrow{y} = \langle y_0, ..., y_{m-1} \rangle = \overrightarrow{0}$
public parameter: $\overrightarrow{\pi} = \langle \pi_0, ..., \pi_{n-1} \rangle$
irreducible polynomial: $\overrightarrow{G} = \langle G_0, ..., G_{n-1} \rangle$
all-zero interim result: $\overrightarrow{x'} = \langle x'_0, ..., x'_{n-1} \rangle = \overrightarrow{0}$

---

for all i in $[0, n-1]$ **do**
  if $\overrightarrow{\pi}(i) = 1$ **then**
    $\overrightarrow{x'} = \overrightarrow{x}$
    if $i \geq 1$ **then**
      for i times **do**
        if $\overrightarrow{x'}(n-1) = 1$ **then**
          $\overrightarrow{x'} = \overrightarrow{x'} >> 1$
          $\overrightarrow{x'} = \overrightarrow{x'}$ **xor** $\overrightarrow{G}$
        else
          $\overrightarrow{x'} = \overrightarrow{x'} >> 1$
        end if
      end for
    end if
    $\overrightarrow{y} = \overrightarrow{y}$ **xor** $(\overrightarrow{x'})_{0,...m-1}$
  end if
end for

IHF implementation using a binary matrix

Another possible solution for the IHF can be found by randomly selecting the public parameter π out of $\Pi = \{0,1\}^{m+n-1}$. Using again vector notation, the public parameter is thus denoted as $\overrightarrow{\pi} = \langle \pi_0, ..., \pi_m, ..., \pi_n, ..., \pi_{m+n-2} \rangle$. As before, the input and output are presented as $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$ and $\overrightarrow{y} = \langle y_0, ..., y_{m-1} \rangle$, respectively. Given this scenario, the function $\overrightarrow{y} = h(\overrightarrow{x}, \overrightarrow{\pi})$ can be shown to be part of a pairwise independent family of hash functions H, if it maps the bits of $\overrightarrow{x}$ and $\overrightarrow{\pi}$ to $\overrightarrow{y}$ by [14],

$$y_j = \sum_{i=0}^{n-1} x_i \cdot \pi_{i+j}, \tag{2.43}$$

for all j in the range $[0, m-1]$. It is worth noticing that a function that satisfies Equation 2.43 can be depicted as the multiplication of the vector $\overrightarrow{x}$ with an $m \times n$ matrix, where the first row of the matrix contains the elements $\pi_0, ..., \pi_{m-1}$ and all subsequent $m-1$ rows are an 1-bit shifted version of the preceding row. This observation is the origin of the name of the here presented solution. However, to keep the discussion as simple as possible, a matrix based notation is omitted here.

For the here considered case of binary number representations, Equation 2.43 can be further simplified. Since each element of $\overrightarrow{x}$ and $\overrightarrow{\pi}$ is either 0 or 1, the multiplication $x_i \cdot \pi_{i+j}$ reduces to a simple AND-operation with x_i and π_{i+j}. Further more, also the result, y_j, must either be 0 or 1. In mathematical terms, this means that y_j is a member of the 2-bit Galois field $GF(2)$. Since, as discussed above, an addition in a Galois field can be executed by means of an XOR-operation, Equation 2.43 can be simplified to [14],

$$y_j = \bigoplus_{i=0}^{n-1} (x_i \wedge \pi_{i+j}), \tag{2.44}$$

where $\wedge$ denotes a bitwise AND-operation and $\bigoplus_{i=0}^{n-1}$ denotes an XOR-operation over all i results of the subsequent term.

To give an illustration of Equation 2.44, the example used during the discussion of Algorithm 2 can be partly reused. Thus, the input is $\overrightarrow{x} = \langle 1,1,0,1,0,0,1,1 \rangle$ and the length of the output is set to $m = 2$. However, since the input length n equals 8 bit, the public parameter $\overrightarrow{\pi}$ has to be of length $n + m - 1 = 9$. This makes it necessary to use a different public parameter than for the previous example and the arbitrary vector $\overrightarrow{\pi} = \langle 0,1,0,0,0,1,0,1,1 \rangle$ is used. The computation of the 2-bit output $\overrightarrow{y} = \langle y_0, y_1 \rangle$ according to Equation 2.44 is shown in Table 2.6.

Based on Equation 2.44, an algorithmic solution of the IHF using a binary matrix might be defined as the one shown in Algorithm 3.

Table 2.6: Example for Equation 2.44

$j=0$	$\overrightarrow{x}$	1	1	0	1	0	0	1	1	
	$\overrightarrow{\pi}_{(0,7)}$	0	1	0	0	0	1	0	1	
	$\overrightarrow{x} \wedge \overrightarrow{\pi}_{(0,7)}$	0	1	0	0	0	0	0	1	$y_0 = 0$
$j=1$	$\overrightarrow{x}$	1	1	0	1	0	0	1	1	
	$\overrightarrow{\pi}_{(1,8)}$	1	0	0	0	1	0	1	1	
	$\overrightarrow{x} \wedge \overrightarrow{\pi}_{(1,8)}$	1	0	0	0	0	0	1	1	$y_1 = 1$

Algorithm 3 IHF solution using a binary matrix

Input: $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$
Initialize: all-zero output: $\overrightarrow{y} = \langle y_0, ..., y_{m-1} \rangle = \overrightarrow{0}$
public parameter: $\overrightarrow{\pi} = \langle \pi_0, ..., \pi_{m+n-2} \rangle$

for all j in $[0, m-1]$ **do**
  for all i in $[0, n-1]$ **do**
    $y_j = y_j$ **xor** (x_i **and** π_{i+j})
  end for
end for

2.5 Dynamic Power in Digital Systems

In order to derive meaningful power estimates from the post-processor implementations, which are proposed during this project, and to be able to explore power saving design techniques, it is necessary to consider the causes of power dissipation in digital systems. While a detailed discussion is far beyond the scope of this report, this section introduces the reader to some basic concepts. In addition, *clock gating*, which is an example of a simple but effective design technique to minimize power requirements in a digital circuit, is presented at the end of this section.

Generally, the power consumption of a digital system depends on a wide range of factors, as for example temperature, the used technology, the functional purpose of the system or the applied data. However, it is common to classify total power dissipation of a single digital gate into two main categories [32]. The first part describes power dissipation that can be considered to be caused by direct currents from the supply voltage of a gate to its ground level. The associated power dissipation is denoted P_{textDC} and frequently referred to as *leakage power*. The second component, P_{switch}, describes the power that is consumed due to the switching activity of a logic gate. P_{switch} is commonly called the *dynamic power* component. Thus, the total power required by a digital gate, P_{tot}, can be described by [32],

$$P_{\text{tot}} = P_{\text{DC}} + P_{\text{switch}}. \tag{2.45}$$

Examples of power consuming effects that are a part of P_{DC} are sub-threshold leakage currents in the transistors of the gate or "short-circuit" currents from the supply voltage to the ground level during switching [32, pp. 257-258][33]. In general,

these effects can be affected by choices made with respect to the global architecture of a system or by considering the process technology used to realize the CMOS transistors of the logical gates [34, p. 11.2]. As both these aspects are out of the scope of this project, it is at this point omitted to further explore P_{DC}.

Focusing on P_{switch}, this component of the total power consumption is mainly due to the fact that the switching of the output of a logical gate yields the charge and discharge of capacitances in the circuit. It can be illustrated by the example of a simple logical inverter [33]. If the output of the inverter is high, this means in practice that the output of the inverter is connected to the supply voltage, commonly denoted V_{DD}. As such, the inverter charges capacitances that exist in the circuit, typically modeled as one single capacitance C. If the inverter switches its output back to logic zero, the output of the inverter is connected to the ground level of the digital circuit. Hence the capacitance C is discharged. It can be shown, that the energy dissipated during the process of charging and discharging C is equal to $C \cdot V_{\text{DD}}^2$.

It is quite common to assume that C is charged and discharged once per period of the clock that drives the digital circuit. This would, for instance, be the case if the respective clock is used as an input to the considered inverter. Denoting the frequency of the clock as f_{clk}, it follows that [32, p. 259],

$$P_{\text{switch}} = f_{\text{clk}} \cdot C \cdot V_{\text{DD}}^2. \tag{2.46}$$

However, Equation 2.46 is a rather rough estimation of P_{switch}, since most logical gates in a digital system do not change their output value twice per clock period. As the output of a gate directly depends on the applied input values, the number of output transitions for a given gate is actually a function of the statistical character of the applied input data [33]. To cope with this fact, it is common to include the *switching activity*, γ, into Equation 2.46, which is used to describe the average number of changes in the output value. Since the capacitance C is charged during the transition from logic zero to logic one, and as each charge must at some point result in a discharge, γ is commonly computed as the average number of transitions from low to high of the gate output during a clock cycle. Based on this, Equation 2.46 can be reformulated as [33],

$$P_{\text{switch}} = \gamma \cdot f_{\text{clk}} \cdot C \cdot V_{\text{DD}}^2. \tag{2.47}$$

In general, decreasing any of the components of Equation 2.47 yields to a reduction of the consumed switching power. While this can be achieved at many different stages during the design of digital systems, it is for the purpose of this project reasonable to concentrate on techniques that can be applied during the *Register-Transfer Level* (RTL) phase of an implementation. Such techniques normally focus on the reduction of the switching activity, γ.

An example of an *Register-Transfer Level* (RTL)-technique for power reduction is the so called *guarded evaluation* [34, p. 11.17]. The concept of this technique is to reduce γ by preventing non-valid or irrelevant data to propagate into combinational blocks of a system, where they would lead to unnecessary computations, and as such, to an unnecessary high switching activity. One possible way to implement guarded

evaluation into a system is by means of latches that are inserted at the input of the considered combinational block.

However, it should be noted that the insertion of latches increases the number of gates in the system, which, with respect to Equation 2.45 yields to an increase of the power. This trade-off between a potential increase in the power consumption due to additional logic against a decrease of the power dissipation through a reduction of γ is a typical aspect of RTL-techniques for power reduction.

Another example of such a technique is the so called *clock gating*, which is especially useful if to interconnected sub-systems operate at different clock frequencies.

Clock gating

One well known design approach to reduce dynamic power consumption is the so called *clock gating* [34, pp. 11.5-11.13]. Simplified, a *clock gate* is a digital component, which can be used to suppress an applied clock signal and thus prevents it from propagating into connected synchronous blocks.

A major part of modern digital systems are synchronous registers, used to store data. The most basic example of such a storage element is the *D-flip-flop*, which loads the applied input to its output on each positive clock edge of the connected clock signal [32, pp. 431-436]. However, it is often undesirable to load a new value to the register each time a positive clock edge occurs. Therefore, many storage elements use an approach which makes it possible to chose to either load a new input or to keep the current data by reloading it.

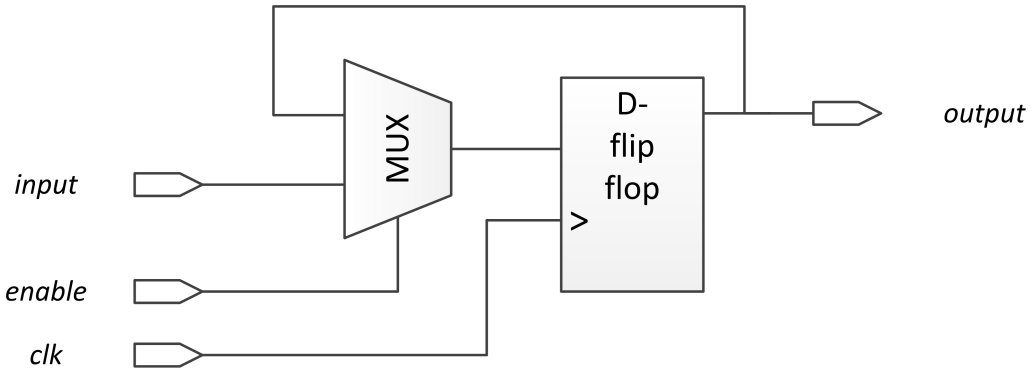
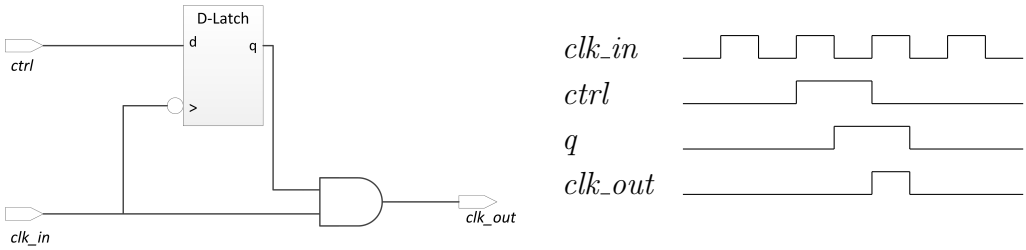


Figure 2.9: D-flip-flop with enable signal [34]

Figure 2.9 shows a common realization of such an *enabled* register [34, p. 11.8]. If the enable signal is set, the applied multiplexer selects the data input as an input to the D-flip-flop. In the case that the enable signal is low, the current data is reloaded. It should be noted, that this approach introduces an overhead in form of the introduced multiplexer.

However, even though the in Figure 2.9 depicted design works as desired, it is not a very power efficient solution. For instance, even if the D-flip-flop does not load a new value, dynamic power is dissipated on each positive clock edge. A possible way



(a) Schematic of a clock gate [34] (b) Waveform of a clock gate

Figure 2.10: A simple clock gate

to omit this is to suppress the applied clock signal, whenever no new value is loaded into the register. In this way, redundant switching activity can be reduced.

This can be achieved by means of clock gates. As stated above, clock gates are digital components that can be controlled to suppress a clock signal. In other words, by deactivating the clock gate, the applied clock is prevented from transferring to the output of the gate. If the clock gate is activated, the clock signal passes unobstructed. An example of a simple clock gate is presented in Figure 2.10.

Chapter 3

ADC as an Entropy Source

It has been mentioned during the motivation of this project (see Section 1.3), that its is desirable to explore the possibility of using an ADC as an entropy source of a TRNG.

From Section 2.3, it seems possible to generate entropy containing data based on the internal thermal noise of an ADC, even though the amount of entropy per single output bit is likely to be low. The following chapter presents therefore the implementation and analysis of such a weak source of entropy, using an ADC embedded on an *EFM32 Wonder Gecko (EFM32)* from *Silicon Laboratories* [5]. This source implementation is used to generate entropy containing data streams necessary to evaluate the TRNG designs presented during this report (see Chapter 5). In addition, the power and timing performance of the used ADC is adopted as a reference, in order to estimate the energy requirements of the introduced TRNGs.

During this project, an entropy source has been implemented by programming an *EFM32* using the C programming language. This is presented in Section 3.1. After the implementation process, it is of interest to perform a short analysis of the output data, in order to be able to choose a appropriated post-processing algorithm. It has been mentioned in Section 2.4, that, while the IHF post-processor can operate on data streams as long as they contain a sufficient amount of min-entropy, the VNC requires its input to meet rather strict requirements on its statistical characteristic. To test the output of the source for these *von Neumann conditions*, Section 3.2 proposes a novel statistical hypothesis test, which can be used to evaluate whether the VNC is an appropriate post-processing algorithm for the tested entropy source. Finally, Section 3.3 presents a short analysis of the output of the in Section 3.1 introduced entropy source.

3.1 Implementation of an Entropy Source Using an ADC

It has been shown in Section 2.3, how thermal noise in an ADC yields to random variations and thus to an entropy containing output. Even though it has been concluded that this is generally true, an illustrative example has been considered, which uses a constant input signal. It has been stated during the discussion around Fig-

ure 2.8 that, for this simple model, the expected output bits are both identically distributed and independent. Recalling from Section 2.4.1 that these are the statistical conditions that enable the use of the VNC as a post-processor, it is desirable to try to realize an entropy source, which is as closely related to the in Section 2.3 presented model as possible. Based on this, the following section introduces an entropy source which uses the ADC of an *EFM32 Wonder Gecko* with a constant input signal.

The *EFM32* includes a SAR ADC with a resolution of up to 12 bits [5]. Since the in Figure 2.8 described variations due to noise affect mostly the LSB of the output data, the full 12-resolution is needed in order to use the ADC as an entropy source. With regards to the input signal, the ADC can be connected to 6 internal signals, including the supply voltage and the reference voltage of the ADC divided by a factor of 2. Both these signals should ideally be constant, and it is therefore chosen to use the halved reference voltage as an input. The expected LSB for this input signal is 1 [5].

Considering the timing and the choice of the frequencies of the applied clocks, it is reasonable to assume that a large number of sampling operations has to be performed by the ADC. For instance, to generate a key for the AES, 128 bits are required. Even if the ADC could produce one bit of entropy per sample, 128 sampling operations would be required. However, as stated in Section 2.3, it is expected that the generated entropy is far below one per single output bit. Hence, some kind of post-processing is needed, which in general is of compressing nature (see Section 2.4) and thus require an even larger number of input bits and, as such, sampling operations. As a result, in order to keep the time required by the TRNG in a practical range, it is tried to keep the time spend to generate one ADC sample as short as possible.

The *EFM32* supports a main clock frequency of up to 48 Mhz. However, this clock signal is too fast to drive the ADC and the clock applied to the ADC must thus be downscaled. As a result, using 48 MHz for the main clock of the MCU, the maximum clock frequency for the ADC is 12 MHz. Further more, the ADC requires several clock cycles to generate a single sample. For the required 12-bit resolution, a minimum of 13 clock cycles is needed. Hence, for the given parameters, the ADC can generate approximately one sample per 1.08 μs. In other words, using the ADC as an entropy source, it produces one output bit per 1.08 μs.

Appendix B presents a C-code routine which sets the *EFM32* up to use the discussed parameters and then produces and exports an arbitrary large amount of output data.

Having established a a setup for an entropy source, it is of interest to derive the estimates of its power and energy performance, to be able to compare it to the estimates gathered for the implementations of the post-processors (see Chapter 6). Ideally, precise results could be found by measuring the MCU required voltage and current. However, the technology used to realize the *EFM32* is kept secrete by the vendor. As such, comparing the energy required by the ADC directly to the corresponding results for the post-processors is not possible.

For the purpose of this project, the power and energy performance of the ADC

in the *EFM32* is therefore rather considered to be an indicator of in practice typical values and not as concrete measurements. It is therefore suitable to base the estimation of the dissipated power and energy on the data provided in the documentation of the device [5][35]. With respect to these, the *EFM32* is supplied with 3.3 V and the ADC has an active current of roughly 351 μA. Hence, the power dissipated in the ADC can be approximated at 1.16 mW. Recalling that the sampling period is estimated to be 1.08 μs, this means that the here presented entropy source requires roughly 1.25 nJ per output bit.

Table 3.1 summarizes the most important parameters of the presented setup.

Table 3.1: Summary of the used ADC parameters

Main clock frequency	48 MHz
ADC clock frequency	12 MHz
Samplings frequency	926 kHz
Sampling periode	1.08 μs
Supply voltage	3.3 V
Active current	351 μA
Dissipated power	1.16 mW
Energy per bit (LSB)	1.25 nJ

Before moving on to further analyze the proposed entropy source, it can be stated that the in Table 3.1 presented data indicate that using the ADC as a source is quite costly with regards to both time and energy per bit. A possible alternative approach that would hide this costs could be to store the LSBs of previous sampling processes in a buffer. In other words, assuming that the ADC is first used to sample a signal independently of the process of generating random data, the LSBs of this conversions would be stored in a buffer. Due to the thermal noise in the ADC, the stored LSBs would contain some amount of entropy. When the TRNG then is activated it could use this buffer as an "entropy source". In this way, the ADC does not have to be run again in order to generate bits for the TRNG.

However, it is reasonable to assume that the so gained LSBs do not correspond to a constant input signal to the ADC. As such, their statistical characteristics are rather unknown and it is harder to find an appropriated post-processor and select reasonable parameters for it.

Even though this approach promises a drastically reduction of the time and energy used to generate the source bits of a TRNG, exploring it in more detail is left for further work, due to the limited time frame of this project.

3.2 A Test for von Neumann Conditions

As stated during its presentation in Section 2.4.1, the VNC requires its input to meet two statistical characteristics, in order to function properly. For convenience, this *von Neumann conditions* are restated here. First, denoting the input stream as $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$, where each element in $\overrightarrow{x}$ is a single bit, the elements of each successive bit-pair, $\langle x_{2i}, x_{2i+1} \rangle$, must be statistical independent. Second, the

probability of x_{2i} taking a concrete value must be equal to the probability that x_{2i+1} takes the same value. For instance, $\Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0) = p$.

It is also worth to recall that, while the von Neumann conditions must be satisfied in order for the VNC to work as described in Table 2.4, two related aspects are not required. First, p is allowed to vary between bit-pairs. Second, the $\overrightarrow{x}$ can contain forms of dependencies, as long as it is given that x_{2i} and x_{2i+1} are independent.

Since only an input with von Neumann conditions guarantees that the VNC works properly, it is of interest to be able to test an entropy source for these conditions in the same manner as the NIST test suite tests a source output for randomness. If such a test concludes that a source provides von Neumann conditions, the VNC can be used as a post-processing algorithm for the given source. However, if the test result is negative, a different source or post-processing algorithm has to be used. The following section proposes a possible test for von Neumann conditions, which is used to evaluate the entropy source presented in Section 3.1

As independence between successive bits is one of the von Neumann conditions, a first approach for a von Neumann condition test could be based on tests that are designed to test for independence. Examples of such tests are the well known *Fisher-Irwin* test [17, pp. 297-302] or an alternative test, presented in [15, pp. 373-376], which is closely related to the chi-squared goodness-of-fit test, as discussed in Section 2.2.1. While a detailed presentation of these tests is omitted here, the following gives a brief background on the basic concept of these test, before their suitability for a von Neumann condition test is discussed.

Both tests have a similar approach. They are designed to test the null-hypothesis, that two events are independent. For a von Neumann condition test, it would therefore be appropriated to formulate the null-hypothesis, that x_{2i} and x_{2i+1} are independent. The tests base the evaluation of this hypothesis on the observed frequencies of the four possible outcomes for the bit-pair $\langle x_{2i}, x_{2i+1}\rangle$. This is depicted in Table 3.2, where the observed frequency of the pair $\langle x_{2i} = 0, x_{2i+1} = 0\rangle$ is referred to as o_{00}, $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ corresponds to o_{01}, and so on. It is also shown in Table 3.2 that the addition of appropriated observed frequencies yields the number of pairs for which one bit takes a specific value. For instance, the number of pairs for which $x_{2i} = 1$ is $o_{1\mathrm{x}} = o_{10} + o_{11}$. In other words, $o_{1\mathrm{x}}$ is the number of times that $x_{2i} = 1$ without consideration of x_{2i+1}. With regards to the marginal probability, introduced in Section 2.1, it is therefore common to refer to $o_{1\mathrm{x}}$ as the *marginal frequency* of $x_{2i} = 1$ [15, p. 374].

Table 3.2: Contingency table

		x_{2i+1}		Total
		0	1	
x_{2i}	0	o_{00}	o_{01}	$o_{0\mathrm{x}} = o_{00} + o_{01}$
	1	o_{10}	o_{11}	$o_{1\mathrm{x}} = o_{10} + o_{11}$
Total		$o_{\mathrm{x}0} = o_{00} + o_{10}$	$o_{\mathrm{x}1} = o_{01} + o_{11}$	$\frac{n}{2}$

Both above mentioned tests for independence use the marginal frequencies to express expectations for the observed frequencies given that x_{2i} and x_{2i+1} are inde-

pendent. Without going into detail, it is fairly easy to understand the motivation of this approach: It follows directly from the definition of statistical independence, as presented in Equation 2.9, that the probability of observing two statistical independent bits is given by the product of their marginal probabilities. Since the marginal frequencies consider only one isolated bit, they are a reasonable choice for an estimate of the corresponding marginal probability. For instance, the marginal probability of $x_{2i} = 0$ can be estimated as,

$$\Pr(x_{2i} = 0) = \frac{o_{0\mathrm{X}}}{n/2}. \tag{3.1}$$

Accepting this estimation and letting e_{00} be the expected frequency of the pair $\langle x_{2i} = 0, x_{2i+1} = 0\rangle$ given that the bits are independent, it follows that,

$$e_{00} = \frac{n}{2} \cdot \Pr(x_{2i} = 0) \cdot \Pr(x_{2i+1} = 0) = \frac{o_{0\mathrm{X}} \cdot o_{\mathrm{X}0}}{n/2}. \tag{3.2}$$

It is worth noticing that Equation 3.1 and Equation 3.2 depict specific examples, but can be used for any arbitrary bit pair $\langle x_{2i}, x_{2i+1}\rangle$ through slight modification. Even though the Fisher-Irwin test and the in [15, pp. 373-376] presented test use different test statistics, both test can be considered to relate the observed frequencies to the expected frequencies. The evaluation of the test is then based on the match or mismatch of expectation and observation.

The above presented tests seem to be a valid approach to test whether to successive bits are independent. However, the main drawback of using one of the tests is that they are designed to test solely for independence. While independence between x_{2i} and x_{2i+1} is an important aspect of von Neumann conditions, it is as important that both bits are identically distributed, that is, $\Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0) = p$. One possible approach to modify the tests could be to use once more Equation 3.1 and estimate $\Pr(x_{2i} = 0)$ and $\Pr(x_{2i+1} = 0)$ by means of their observed frequencies, $o_{0\mathrm{X}}$ and $o_{\mathrm{X}0}$, respectively. Thus, if the source has von Neumann conditions and $\Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0)$, it would be expected that $o_{0\mathrm{X}} = o_{\mathrm{X}0}$.

However, this approach is unsatisfying for a von Neumann condition test. This is due to the fact that, while the von Neumann algorithm requires that $\Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0) = p$, it allows for p to vary between different bit pairs. But using Equation 3.1 to estimate the marginal probabilities yields rather an average value of, for instance, $\Pr(x_{2i} = 0)$ over $\frac{n}{2}$ bit pairs. An approach that compares $o_{0\mathrm{X}}$ and $o_{\mathrm{X}0}$ can therefore not cope with variations of p.

This can be illustrated by considering a bit stream of 100 bits for which p varies fast and differs from bit to bit. However, it is not impossible that for the observed 100 bits $o_{0\mathrm{X}}$ and $o_{\mathrm{X}0}$ are equal or at least in the same range. Thus by using Equation 3.1, one would conclude that $\Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0)$, even though this is in fact not the case.

Having established that tests of independence, as those discussed above, are rather unfit to be used as a von Neumann condition test, an alternative approach has to be found. In the following, such a novel alternative test procedure is proposed. It is directly based on the concept of the von Neumann corrector, as presented in Section 2.4.1. It is worth to recall that the foundation of the VNC post-processor is a simple observation regarding Table 2.4. The table states that the probability

of observing a *von Neumann pair*, that is, either $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ or $\langle x_{2i} = 1, x_{2i+1} = 0\rangle$, is equal for both constellations, if the von Neumann conditions are satisfied. As a result, it is expected that, in such a situation, the observed frequency of one von Neumann pair type, say o_{01}, is close to $\frac{o_{01}+o_{10}}{2}$. As this expectation is a direct result of the assumption of von Neumann conditions in the source X, it makes sense to use it as the basis of a von Neumann condition test.

At the beginning of the development of a test stands, as described in Section 2.2.1, the formulation of the null-hypothesis, $\mathcal{H}_0$. To be able to test both aspects of the von Neumann conditions, the null-hypothesis is formulated as,

$$\mathcal{H}_0 : \begin{cases} \Pr(x_{2i}, x_{2i+1}) = \Pr(x_{2i}) \cdot \Pr(x_{2i+1}) & \text{(independence)}, \\ \Pr(x_{2i} = 0) = \Pr(x_{2i+1} = 0) = p & \text{(identical distribution)}, \end{cases} \tag{3.3}$$

where the independence statement holds for all possible combinations of x_{2i} and x_{2i+1}. It should be noted, that in contrast to the above discussed tests for independence, the in Equation 3.3 specified null-hypothesis of the here presented alternative test approach covers both von Neumann conditions.

Based on $\mathcal{H}_0$, a appropriated test statistic s has to be found. As the focus of the test lies on the frequencies of the outcomes $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ and $\langle x_{2i} = 1, x_{2i+1} = 0\rangle$, the test statistic, s, is chosen to be the frequency of $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$, that is,

$$s = \text{Number of } \langle x_{2i} = 0, x_{2i+1}\rangle \text{ occurrences.} \tag{3.4}$$

There are two important aspects of this choice. First, s is neither the observed frequency, o_{01}, nor the expected frequency, e_{01}. As discussed in Section 2.2.1, the test statistic s is required to be a variable that can take different values with a given probability. However, both o_{01} and e_{01} are fixed values. Using the notation established in Section 2.2.1, the expected frequency can be considered to be the mean of the reference distribution of s, $\mathrm{E}[s] = e_{01}$, while the observed frequency corresponds to the observed test statistic, $s_o = o_{01}$. Second, it should be stressed, that this choice of s is arbitrary and the frequency of $\langle x_{2i} = 1, x_{2i+1} = 0\rangle$ could have been chosen in the same manner. This is due to the fact that both frequencies are supposed to be approximately equal.

Having defined s, it is necessary to find the corresponding reference distribution. In order to do so, it is reasonable to first find the mean of the distribution, $\mathrm{E}[s]$, which is equal to the expected frequency e_{01}, and then establish how s is distributed around $\mathrm{E}[s]$. A first approach could be based on the probability of observing $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ as stated in Table 2.4. The table shows that $\Pr(x_{2i} = 0, x_{2i+1} = 1) = p \cdot (1 - p)$. Assuming that the test is conducted over an input stream $\overrightarrow{x}$ with n elements, the number of bit pairs is $\frac{n}{2}$. As a result, the expected frequency for $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ is,

$$e_{01} = \mathrm{E}[s] = \frac{n}{2} \cdot p \cdot (1 - p). \tag{3.5}$$

The obvious problem of using Equation 3.5 is that p is unknown. Further more, during the discussion of the test for independence it has been pointed out that it is rather difficult to find an appropriated estimate of p that allows for variations between bit pairs.

However, it is possible to avoid the use of p. It follows from Table 2.4 that the probability of observing $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ is equal to the probability of observing $\langle x_{2i} = 1, x_{2i+1} = 0\rangle$. Thus, the expected frequency of observing $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ is equal to half the number of observed von Neumann pairs. Denoting the number of observed von Neumann pairs as $o_{\text{vN}} = o_{01} + o_{10}$, this yields,

$$e_{01} = \text{E}[s] = \frac{o_{\text{vN}}}{2}. \tag{3.6}$$

Even though p is affecting the expected frequency of von Neumann pairs, Equation 3.6 is completely independent of p. Seen from a different perspective, Equation 3.6 formulates the simple expectation that $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ is observed as often as $\langle x_{2i} = 1, x_{2i+1} = 0\rangle$, regardless of how many von Neumann pairs are expected to be observed.

To establish how s is distributed around the in Equation 3.6 stated mean E$[s]$, it may be helpful to consider the following model: The input stream $\overrightarrow{x}$ contains of $\frac{n}{2}$ bit-pairs of which o_{vN} are von Neumann pairs. Focusing merely on the von Neumann pairs, X simplifies to a source that produces either $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ or $\langle x_{2i} = 1, x_{2i+1} = 0\rangle$, both with equal probability, given that $\mathcal{H}_0$is true. Assuming that the bit pairs are generated independently from each other, such a model is in the literature classified as a *Bernoulli process* [15, p. 144].

It is important to note that the assumption that the bit pairs are independent is not part of the von Neumann conditions, but rather implies some stricter form of independence in the data. As such, this assumption yields the proposed test to evaluate the source X for stricter characteristics than required by the VNC. However, it has been discussed in Section 2.4.1 that mutual independence is a desirable feature of $\overrightarrow{x}$ in order to achieve an output with maximum entropy rate. Thus, designing the test to evaluate the data for a stricter form of independence by using the mention assumption is acceptable at this point.

It can be shown that the frequency of one specific outcome of a Bernoulli process is binomially distributed [15, p. 145]. Since, given that $\mathcal{H}_0$is true, the generation of von Neumann pairs can be modeled as a Bernoulli process, this means that the frequency of the outcome $\langle x_{2i} = 0, x_{2i+1} = 1\rangle$ is binomially distributed over the range $[0, o_{\text{vN}}]$ with probability $\frac{1}{2}$. Thus, the reference distribution of s is the binomial distribution over the same range and with the same probability. The mean and standard deviation of the distribution can be shown to be $\frac{o_{\text{vN}}}{2}$ and $\frac{\sqrt{o_{\text{vN}}}}{2}$, respectively [15, p. 147].

Without going into detail here, it should be mentioned that the binomial distribution with mean $\frac{o_{\text{vN}}}{2}$ and standard deviation $\frac{\sqrt{o_{\text{vN}}}}{2}$ can be approximated for large values of o_{vN} by the normal distribution with the same mean and standard deviation [15, pp. 172-192]. This is illustrated in Figure 3.1, which shows the validity of the approximation for $o_{\text{vN}} = 128$. An explicit choice of the number of by the test considered bits, n, which directly affects o_{vN} is postponed to the execution of the test in Section 3.3. In the meanwhile however, it is simply assumed that n is selected in a way that guarantees the approximation to be valid. In other words, for the rest of this discussion, the pdf of the reference distribution of s is considered to be the normal distribution,

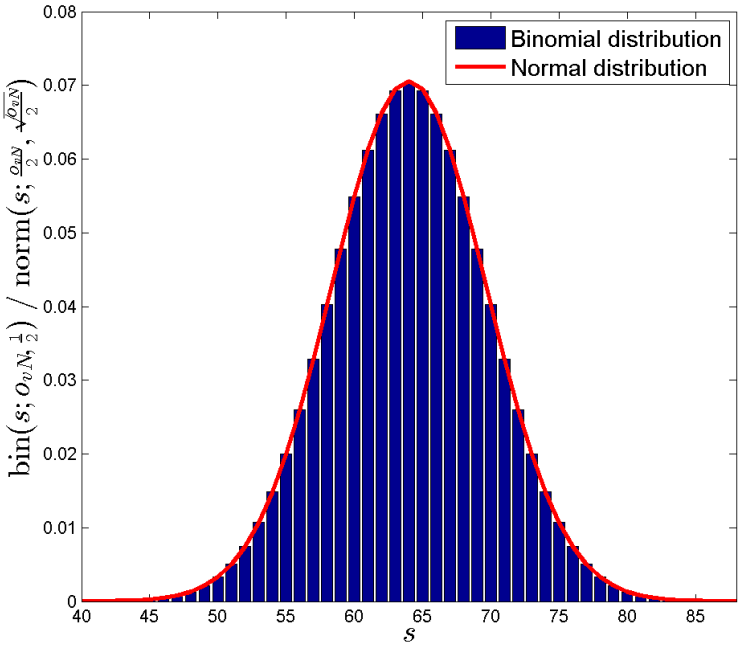


Figure 3.1: Approximation of the binomial distribution by the normal distribution for $o_{\mathrm{vN}} = 128$

$$\mathrm{pdf}(s; \mathcal{H}_0) = \mathrm{norm}(s; \frac{o_{\mathrm{vN}}}{2}, \frac{\sqrt{o_{\mathrm{vN}}}}{2}), \tag{3.7}$$

Based on the established reference distribution of s, it is now possible to conduct the test by relating the observed test statistic to the reference distribution. As stated above, for this test, the observed test statistic is the observed frequency of $\langle x_{2i} = 0, x_{2i+1} = 1 \rangle$, $s_o = o_{01}$. It is essential to note that both an observed frequency that is much larger than the expected frequency and an observed frequency that is much less than the expected value are unlikely given that $\mathcal{H}_0$is true. Thus, the test should reject $\mathcal{H}_0$either if $o_{01} << e_{01}$ or $o_{01} >> e_{01}$. This means that the given test is two-sided.

As a result, one possible way to evaluate the test based on o_{01} is to select a level of significance, α, and use it to define two critical values c_- and c_+. Since the normal distribution is a constant distribution, it is possible to combine Equation 2.19 and Equation 3.7 and derive the following relation between α and the critical values,

$$\begin{aligned} \alpha &= 2 \cdot \int_{-\infty}^{c_-} \mathrm{norm}(s; \frac{o_{\mathrm{vN}}}{2}, \frac{\sqrt{o_{\mathrm{vN}}}}{2}) \mathrm{d}s \\ &= 2 \cdot \int_{c_+}^{\infty} \mathrm{norm}(s; \frac{o_{\mathrm{vN}}}{2}, \frac{\sqrt{o_{\mathrm{vN}}}}{2}) \mathrm{d}s. \end{aligned} \tag{3.8}$$

As discussed in Section 2.2.1, the test would then yield a rejection of $\mathcal{H}_0$, if $o_{01} \leq c_-$ or $o_{01} \geq c_+$.

For example, observing $o_{\text{vN}} = 225000$ von Neumann pairs, the expected test statistic, $\text{E}[s] = e_{01}$, equals 112500 pairs. Setting the level of significance of the test, for instance, to $\alpha = 0.01$, the critical values can be computed from Equation 3.8 to be $c_- \approx 1118990$ and $c_+ \approx 113110$. In other words, for the given example, the test would fail if the observed test statistic o_{01} differs with more than approximately 610 pairs from the expected value of 112500.

However, in order to increase the significance of the test, it is reasonable to perform the test for a number of N sequences instead of a single sequence, in the same manner as done for the NIST test suite presented in Section 2.2.2. Since this approach opens for the possibility of contradicting test results of the single (first-level) tests, it is necessary to be able to derive a single test result. As for the NIST test suite, it is chosen to use a second-level approach based on the p-values of the first-level tests, using the chi-squared goodness-of-fit test, as described in Section 2.2. Since the presented test is two-sided and it can be shown that the normal distribution is a symmetrical continuous distribution, it follows from combining Equation 2.20, Equation 3.6 and Equation 3.7 that the p-value of the first-level tests can be computed by,

$$\text{p-value} = \begin{cases} 2 \cdot \int_{-\infty}^{o_{01}} \text{norm}(s; \frac{o_{\text{vN}}}{2}, \frac{\sqrt{o_{\text{vN}}}}{2}) \text{d}s & \text{if } o_{01} \leq \frac{o_{\text{vN}}}{2}, \\ 2 \cdot \int_{o_{01}}^{\infty} \text{norm}(s; \frac{o_{\text{vN}}}{2}, \frac{\sqrt{o_{\text{vN}}}}{2}) \text{d}s & \text{if } o_{01} > \frac{o_{\text{vN}}}{2}. \end{cases} \tag{3.9}$$

An implementation of the here introduced von Neumann condition test, using MATLAB, is presented in Appendix C.1. The script performs the test by first scanning through the applied data sequences and counting the number of von Neumann pairs, o_{vN}, and the number of observations of the bit-pair $\langle x_{2i} = 0, x_{2i+1} = 1 \rangle$, o_{01}, in the data. Based on these counts, the test is evaluated for each applied sequences. It is worth noticing, that the script does not explicitly compute the critical values, by means of Equation 3.8. Instead, the p-value for each sequence is calculated by solving Equation 3.9. However, recalling from Section 2.2.1, that a observed test statistic that corresponds to a critical region results in a p-value below the level of significance α, the test can directly be evaluated from the p-value for a given α. After the p-value has been found, the test script exports the results of each applied sequence to an output file. If the different sequences yield to different conclusions, the script can be enabled to run a second-level test. In that case, the chi-squared goodness-of-fit test, as described in Section 2.2.1, is used to evaluate whether the first-level p-values are uniformly distributed. This results in a single second-level p-value, computed as described in Equation 2.25.

3.3 Output Analysis of the Entropy Source

Having established a basic entropy source in Section 3.1, it is necessary to perform an analysis of the generated output, in order to be able to choose an appropriated post-processing algorithm, later during this project.

It is reasonable to start the analysis of the entropy source, by running the output through the NIST test suite, as presented in Section 2.2.2. Even if the introduced

source implementation does not pass the test suite without post-processing, executing these series of tests can yield valuable information about the statistical characteristic of the source data, which can be useful for the choice of a post-processing algorithm.

Before performing the NIST test suite, some basic test parameters have to be determined. Most importantly, the number of input bits has to be chosen. It has been stated in Section 2.2.2, that each selected test is performed N times and for each run a different n-bit input is considered. Thus, a total of $N \cdot n$ bits is used as an input. With regards to the in Table 2.3 stated requirements on the length of the input sequences, it is reasonable to choose $n = 1\,\text{Mbit}$ (1Mbit = 1048576 bits). This choice is further encouraged, as a sequence length in the range of 10^6 is a frequently used value [19].

The decision of the number of sequences, N, is more complex. As discussed in Section 2.2.2, the significance of the test procedure is strengthened with an increasing number of input bits. Thus, it is beneficial to choose a large value of N. For example, [19] suggests N to be in the range of 20000 sequences. However, given the circumstances of this project, such large value of N is not suitable. Using, for instance, $N = 20000$ would make it necessary to generate a total of 20 Gbit. In addition, for the sake of consistency, it is appropriated to determine N in a way that makes it possible to use the same value during the evaluation of the post-processors (Chapter 5). Keeping the compressing nature of the post-processing algorithms in mind (see Section 2.4), it is reasonable to assume that using $n = 1\,\text{Mbit}$ and $N = 20000$ would yield the necessity of generating more than 100 Gbit of source data. Considering both the limited time frame and the hardware resources available during this project, using such a large value of N is therefore clearly unrealistic.

Based on this argumentation, it has been chosen to use $N = 64$ for the purpose of this project. While this clearly reduces the significance of the executed tests, the amount of data that has to be generated is more suitable for the given circumstances. In addition, in the case that a second-level test must be performed to evaluate the test results of the first-level tests, the NIST test suite groups the p-values of the first-level test by default into $k = 10$ groups. Thus, with regards to Section 2.2.1, $N = 64$ yields the chi-squared goodness-of-fit test to lead to valid results, for $k = 10$ (see Equation 2.23).

Having established the number of input bits, it remains to determine the level of significance both for the first-level tests (α) and the second-level tests (α'). For the purpose of this project, it has been chosen to set $\alpha = 0.01$ and $\alpha' = 0.0001$, motivated by the fact that these are frequently used values for the NIST test suite [18][19]. Table 3.3 summarizes the parameter choices for the NIST test suite.
Based on the chosen parameters, the NIST test suite is performed on a 64 Mbit input stream, generated by the in Section 3.1 presented source implementation. It is worth noticing, that all bits have been generated at room temperature. Following the recommendation of NIST, the Frequency test, as presented in Section 2.2.2, is performed first, in order to evaluate if other tests of the test suite have to be performed.

The main results of the Frequency test for the 64 source sequences are presented in Table C.1 in Appendix C.2.1. The table shows, that the observed test statistic s_o, as defined in Equation 2.27, is in the range of approximately 400,000 for each of the 64

Table 3.3: Used parameters for the NIST test suite

Parameter	Value
Bits per input sequence (n)	1,048,576
Number of input sequences (N)	64
Level of significance for first-level tests (α)	0.01
Level of significance for second-level tests (α')	0.0001
Number of p-value groups (k)	10

performed tests. This means, that the number of 1's in the data exceeds the number of 0's by roughly 400,000, in each sequence. Recalling from the example given during the presentation of the Frequency test in Section 2.2.2, that, for a sequence of length $n = 1,048,576$, the test fails at a significance level of $\alpha = 0.01$ if $|s_o| > 2,638$, it is obvious that each of the 64 tests fails. Due to this clear result of the first-level tests, a second-level test is not necessary. However, it is worth noticing that the observed test statistics differ so extremely from the expected value $\mathrm{E}[s] = 0$, that the p-value of each test equals zero.

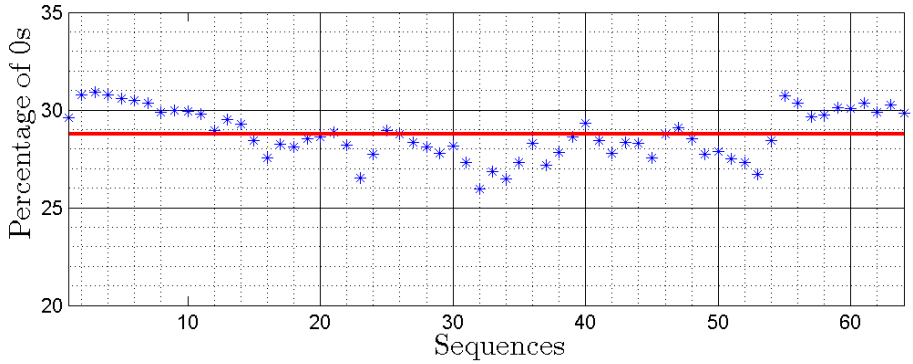
Based on the observed test statistics, it is possible to derive the percentage of 0's in the corresponding sequence. They are presented in Table C.1 and further illustrated in Figure 3.2a. The minimal percentage of 0's is approximately 25.94%, and the maximum is at roughly 30.93%. The average percentage over all 64 sequences is at approximately 28.75%. Hence, the data is clearly biased towards 1, as expected based on the used input signal for the implementation (see Section 3.1).

Figure 3.2a shows that the bias of the data varies between the sequences. With regards to Figure 2.8, a possible reason for this variations of the bias could be changes in the variance of the thermal noise process, due to variations in the temperature. However, this variations can be considered to be minor. The difference between the maximal and minimal percentage of 0's in the data is approximately 5%. In addition, the bias varies rather slowly between sequences. This is depicted in Figure 3.2b, which presents the absolute value of the difference in the percentage of 0 occurrences in the data between to subsequent sequences. The graph shows that the bias of the data rarely differs with more than 1% between two sequences.

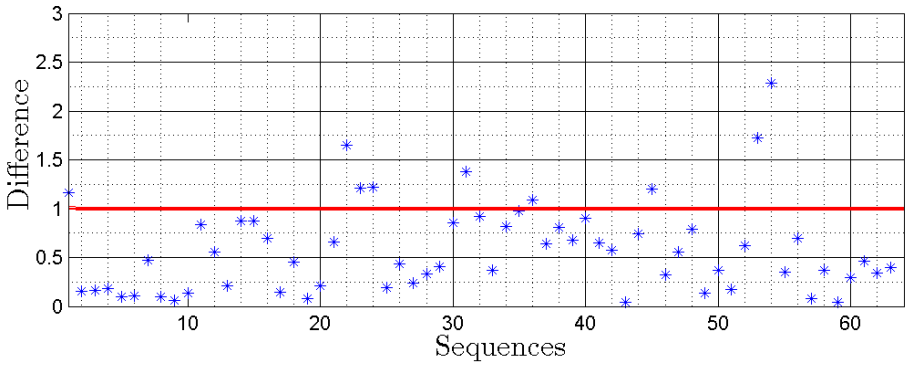
Following the recommendation of NIST, other tests of the test suite are not performed for the data delivered by the source, as the Frequency test fails.

Given the bias of the analyzed data, it is of interest to evaluate whether the VNC of Section 2.4.1 can be used to correct this statistical flaw. It is worth to recall from Section 3.1 that the used input signal for ADC has been chosen for the purpose of deriving von Neumann conditions in the output stream. While Chapter 5 analyses the data produced by a VNC implementation using the here discussed data as a source input, a first expectation of the success of this approach is at this point gained by performing the test for von Neumann conditions, which has been presented in Section 3.2.

In order to perform the test for von Neumann conditions, it is necessary to determine the test parameters, in the same manner as done above for the NIST test suite. It is worth noticing that the von Neumann condition test is designed in a



(a) Percentage of 0's in the sequences of the source data: The red line depicts the average of approximately 28.75%.



(b) Absolute value of the difference in percentage of 0's in subsequent sequences: The red line depicts 1%.

Figure 3.2: Results of the Frequency test for the source data

manner fairly similar as the tests contained in the NIST test suite for randomness, as for example, the Frequency test presented in Section 2.2.2. As a result, the same type of test parameters have to be defined for both tests.

It is reasonable to use the test parameters determined for the NIST test suite, as stated in Table 3.3, as well for the von Neumann condition test, due to two reasons. First, using the same test parameters increases consistency between the different test procedures. Second, using $n = 1\,\text{Mbit}$ yields to a total of 524,288 bit-pairs per sequence. Even though it cannot be stated with absolute certainty, it is reasonable to assume that this yields the number of observed von Neumann pairs, o_{vN}, to be in the range of several ten thousands or even hundred thousands. Taking into consideration that the for the von Neumann condition test used approximation of the binomial distribution by means of the normal distribution yields valid results for $o_{\text{vN}} = 128$ (see Figure 3.1), using an o_{vN} in the range of the thousands pairs guarantees the validity of the approximation.

The results of running the von Neumann condition test on the source data are presented in Table C.2, in Appendix C.2.1. The number of observed von Neumann pairs, o_{vN}, in the sequences ranges from 214,591 to 243,063. The average value of o_{vN} is approximately 228,960. The average number of occurrences of the bit-pair $\langle 0, 1 \rangle$,

o_{01}, is 61,143 and o_{01} ranges from 51,884 to 69,603. With regards to the example given at the end of Section 3.2, it is known that the von Neumann condition test fails for $o_{\text{vN}} = 225,000$ and $\alpha = 0.01$ if o_{01} is smaller than 111,890 or larger than 113,110. Based on this, it is obvious from the data presented in Table C.2 that each of the 64 performed first-level tests fails. This can also be seen from the fact that the p-value of each test is equal to zero, which clearly would yield a second-level test to reject the underlying null-hypothesis.

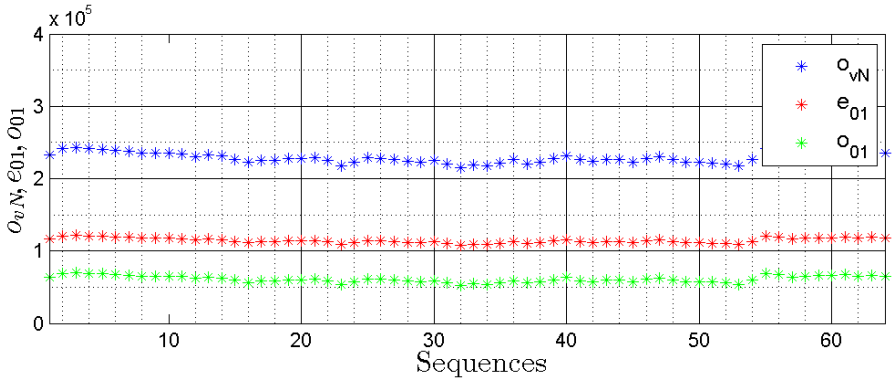
Considering the null-hypothesis of Equation 3.3 and the assumptions that have been made during the definition of the test, two factors can cause the von Neumann condition test to fail. First, the bias of the data can vary to quickly, such that the probability p of observing 0 is not equal for two successive bits. Second, the bits in the data are in some kind dependent on each other.

Evaluating which of the mentioned reasons yield the test to fail is a nontrivial task. However, some first indicators exist. With respect to the in Figure 3.2 presented results of the Frequency test, it has been stated that the bias of the tested sequences varies slowly and in a minor manner. Keeping in mind that this is a major simplification, it is possible to assume that the bias in the sequences vary in a similar manner. Based on this assumption, it is unlikely that the von Neumann condition test fails because of too fast variations of the source bias.

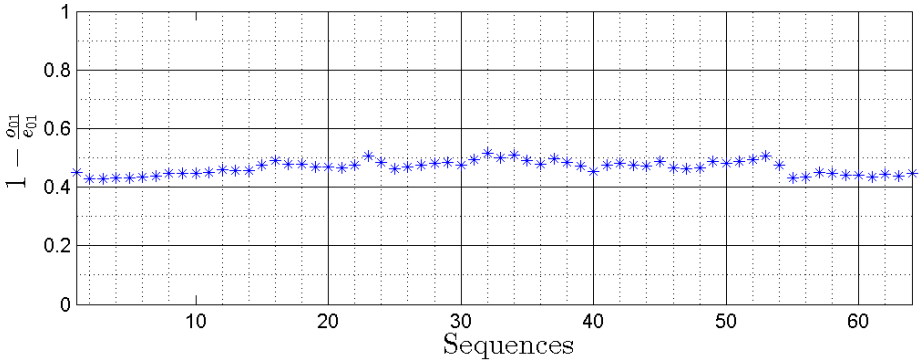
In addition, it is striking that all observed test statistics are in the same range. This is illustrated in Figure 3.3a, which shows the number of von Neumann pairs, o_{vN}, the observed test statistics, o_{01}, and the expected test statistics, $e_{01} = \frac{o_{\text{vN}}}{2}$ (see Section 3.2) for all 64 tested sequences. All three entities can be considered to be rather constant over all tests. For o_{vN} and e_{01}, this behavior is expected due to the minor variations of the bias. For a close to constantly towards 1 biased bit stream, the number of bit pairs containing a 0 is supposed to be as well unvarying. Since e_{01} is a linear function of o_{vN}, this obviously holds also for the expectation of the test statistic.

However, also o_{01} is close to constant for all 64 sequences. Further more, the small variations of the observed test statistics are in accordance with similar variations of e_{01}. This is illustrated in Figure 3.3b, which depicts the normalized difference between e_{01} and o_{01}, that is, $1 - \frac{o_{01}}{e_{01}}$. Thus, the observed test statistics are continuously smaller than the corresponding expected value, with a close to constant percentage.

As a result, observing the bit-pair $\langle 1, 0 \rangle$ is much more likely than observing $\langle 0, 1 \rangle$. Due to the limit time frame of this project, investigating the cause of this fact is left for further work. Nevertheless, it is a strong indicator of dependencies in the data. This is easiest seen by assuming for a moment that the data is independent. In that case, a fast, random changes of the source bias would have to be the cause of the failure of the von Neumann condition test. However, while this is a valid reason for o_{01} differing from the expected value e_{01}, it does not give any explanation for o_{01} being continuously smaller than e_{01}, especially not at a constant rate. Thus, it is reasonable to assume, that the high occurrence of the bit-pair $\langle 1, 0 \rangle$ is due to some system related behavior which causes this bit-pair to be more likely than $\langle 0, 1 \rangle$, and not to random variations of the bias. As such systematical behavior causes the expectation for the second bit in a pair to be affected by the first bit, it causes, from a statistical point of view, the data to be dependent, as discussed in Section 2.1.



(a) Frequencies of von Neumann pairs and the expected and the observed test statistics for the tested sequences



(b) Normalized difference between the observed and expected test statistics for the tested sequences

Figure 3.3: Results of the von Neumann condition test for the source data

In conclusion, based on the Frequency test it is obvious that the data delivered by the entropy source implementation of Section 3.1 cannot be considered to be random, due to its strong bias towards 1. The bias varies marginally around a zero-ratio of approximately 29%.

Also the von Neumann condition test for the data fails clearly. As such, the in Figure 2.7 depicted model is not fit to describe the used source implementation. Analyzing the results of the test in more detail yields to the conclusion that dependencies between bits exist in the data stream. Based on this, the successful use of the VNC post-processor for the given entropy source seems unlikely.

It exists a variety of reasons that could cause the observed dependencies. One possible explanation could be systematic noise from digital circuitry surrounding the ADC. A detailed exploration of this aspect is left for further work.

Chapter 4

Post-Processing Algorithms

In Section 2.4, two different approaches for a TRNG post-processing algorithm have been presented: Section 2.4.1 introduces the *von Neumann Corrector* (VNC) in form of Algorithm 1. The *Extractor based on pairwise Independent Hash Functions* (IHF) is discussed in Section 2.4.2, yielding two possible algorithms, Algorithm 2 and Algorithm 3.

In the following section, this algorithms are considered in more detail, with the objective to create one functional implementation of both the VNC and the IHF. With regards to the motivation of this project (see Chapter 1), the main focus during the implementation of the post-processing algorithms lies on energy efficiency. Since the implementation of an algorithm can be time consuming, it is therefore of interest to find a first indicator on the energy performance of the different algorithms, before the implementation step. In that way, it is possible to select only the most energy efficient solutions and it is avoided to spend time on rather inefficient solutions. In addition, it is important to be able to compare the complexity of the VNC and the IHF during the discussion of different solutions of complete TRNG-systems, presented in Section 7.1.

The in this section derived post-processor implementations can then be used to both test whether the post-processors can be used for the ADC based source (Chapter 5) and to estimate the energy performance of the different TRNG solutions (Chapter 6).

As a result, the following section is divided in two main parts. First, the complexities of the in this report discussed post-processing algorithms is analyzed and compared. This is presented in Section 4.1. Second, Section 4.2 introduces the implementations of the VNC and the IHF.

4.1 Complexity of the TRNG Post-Processing Algorithms

The following section presents a discussion of the complexity of the different, in Section 2.4 introduced, solutions for the VNC (Algorithm 1) and the IHF (Algorithm 2 and Algorithm 3) post-processing algorithms. The main objective of this section is to find an indication of the energy performance of the different approaches, to be

used during further discussions.

Denoting the input of a TRNG post-processing algorithm as $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$ and the output as $\overrightarrow{y} = \langle y_0, ..., y_{m-1} \rangle$, a classical approach of a complexity analysis is to relate the input size n to the number of operations, O, required by the considered algorithm. However, for a TRNG post-processing algorithm the significance of such an approach is reduced. This is due to the reason that, most practical systems require a TRNG to generate a fixed output size m, rather than operating on a fixed n-value. As an example, the AES cryptography algorithm requires a key of at least 128 random bits [6]. With regards to this observation, two aspects are of main interest when analyzing the complexity of a TRNG post-processing algorithm: First, the number of input bits that are required by the algorithm to generate m output bits should be considered. In other words, the input size n should be expressed as a function of the output size m, $n(m)$. This aspect is especially interesting as an indicator of the energy consumption of an entropy source that is used in combination with the considered post-processing algorithm, since, in general, the energy consumption of the source increases with n. Second, instead of relating the number of operations, O, of an algorithm to n, it is more suited to formulate O as a function of m, $O(m)$. In that way, using O as a energy performance indicator, it can directly be related to practical systems settings.

As a result, the complexity of the different algorithms is analyzed both with respect to $n(m)$ and $O(m)$. Section 4.1.1 separately presents the analyses for the three considered algorithms. In Section 4.1.2, these results are related to each other and discussed. Finally, based on the discussion and considering concrete parameter values, it is decided whether to use Algorithm 2 or Algorithm 3 for an IHF implementation. This decission is presented in Section 4.1.3.

4.1.1 Complexity analysis of TRNG post-processing algorithms

A solution for a VNC has been presented in Algorithm 1. The algorithm basically conducts an XOR-operation on two input bits in order to decide whether or not to add a single output bit. Hence, in an ideal situation, the algorithm converts two input bits into one input bit. However, as stated in Section 2.4.1, this is unlikely. A more realistic measure can be found by considering Table 2.4. It states that the probability for adding a bit to the output is $2 \cdot p \cdot (1-p)$. This means that the average amount of output bits generated by two input bits is $2 \cdot p \cdot (1-p)$. It follows, that the average amount of output bits generated per single input bit is $p \cdot (1-p)$. For n input bits, the expected number of output bits is therefore $m = n \cdot p \cdot (1-p)$. Reformulating this statement, the expected number of input bits required by Algorithm 1 in order to generate m output bits,n_{VNC}, can expressed as,

$$n_{\text{VNC}} = \frac{m}{p \cdot (1-p)}. \tag{4.1}$$

This means that n_{VNC} grows linearly with m and approximately inversely proportional with p.

It must be stressed, that Equation 4.1 depicts an expectation. However, to simplify both the notation and the comparison with the IHF algorithms, it has been chosen to consider n_{VNC} as a deterministic value of the input size for most of the rest of this report.

From Equation 4.1, it is fairly simple to derive the number of operations executed by Algorithm 1, O_{VNC}, as a function of m. Since one XOR-operation is executed for two input bits, the number of operations executed for n_{VNC} input bits is $O_{\text{VNC}} = \frac{n_{\text{VNC}}}{2}$. Combining this statement with Equation 4.1 yields,

$$O_{\text{VNC}} = \frac{m}{2 \cdot p \cdot (1-p)}. \tag{4.2}$$

As such, O_{VNC} grows for m and p approximately in the same manner as n_{VNC}, presented in Equation 4.1.

To complete the analysis of the complexity of the VNC, it should be noted that both Equation 4.1 and Equation 4.2 are symmetric around $p = 0.5$. This makes sense, since whether the data is biased towards 0 or 1 is irrelevant from a information theoretical point of view. Thus, throughout the rest of this section, p is only considered over the range [0 0.5].

Further, to simplify the interpretation of Equation 4.1 and Equation 4.2, the probability p is assumed to be constant at least over the n_{VNC} input bits required to generate m output bits. In contrast, the VNC allows for variations of p between succeeding bit-pairs. However, allowing for this feature here would require Equation 4.1 and Equation 4.2 to include some kind of measure of the variation of p, which exceeds the scope of this analysis. It is worth noticing that both Equation 4.1 and Equation 4.2 can still be used as valid indicators, even if p varies, by approximating a constant p value. For instance, the minimum or average probability $\Pr(x_i = 0)$ over the input bits might be chosen as an appropriated approximation of p.

Relating the number of output bits to the number of input bits for the IHF algorithms,n_{IHF} is a more difficult task than for the VNC. By its definition (see Section 2.4.2), the IHF is more concerned about the amount of min-entropy delivered by the entropy source than about the concrete number of input bits. In other words, n_{IHF} must be chosen such that the input $\overrightarrow{x}$ contains a certain amount of min-entropy, κ. It is known from Equation 2.17 that the amount of min-entropy contained in a n_{IHF} bit input cannot exceed n_{IHF}. Thus, the number of input bits for the IHF can be related to κ by,

$$n_{\text{IHF}} \geq \kappa. \tag{4.3}$$

Equation 4.3 can hold with equality if and only if the input bits are mutual independent and each single input bit is uniformly distributed.

Obviously, this equation is a rather general statement. This is due to the fact, that the IHF works for a variety of entropy sources with different statistical characteristics. To be able to formulate a more concrete version of Equation 4.3, it would therefore be necessary to have detailed knowledge of the source characteristics. In order to keep the analysis as general as possible, it has been chosen to, for the moment, use the general relation between n_{IHF} and κ as stated in Equation 4.3, leaving a more specific analysis for von Neumann conditions to the end of this section.

Equation 2.34 in Section 2.4.2 relates the security parameter of the IHF, t, to the number of output bits, m, the security parameter, ϵ, and κ. Solving Equation 2.34 for κ and combining it with Equation 4.3 yields,

$$n_{\text{IHF}} \geq \kappa = m + \beta, \tag{4.4}$$

where the *tuning parameter* of the IHF has been defined as,

$$\beta = 2 \cdot \left(2 \cdot \log_2 \frac{1}{\epsilon} + t + 1\right), \tag{4.5}$$

in order to keep the notation simple.

It follows from Equation 4.4 that κ increases linearly with both m and β. An increase in κ must in its turn result in an increase of n_{IHF} in a generally unknown manner. However,as a concrete example, if the entropy source produces mutual independent output bits, it follows from Equation 2.16 that the min-entropy of a concatenation of several outputs is the sum of the min-entropy of the single outputs. If in addition the min-entropy of all outcomes is equal, the min-entropy of the concatenation can thus be doubled by concatenate twice as many bits. Hence, if m increases by a factor of two, n_{IHF} must be increased by the same factor, for the given example.

Whereas Equation 2.34 and therefore Equation 4.4 are valid for any IHF approach, the number of operations required by Algorithm 2 and Algorithm 3 differs. Considering first the approach depicted in Algorithm 2, the algorithm needs to execute a number of XOR and shift operations. However, assuming for a moment that the above discussed parameters of the IHF (m, n_{IHF}, β and κ) are fixed, the number of executed operations of Algorithm 2, $O_{\text{IHF-1}}$, varies. This is due to the fact, that whether or not an operation should be executed depends on the concrete values of the input, $\overrightarrow{x}$, and the public parameter, $\overrightarrow{\pi}$. For simplicity, a worst case scenario has been considered for the here presented analysis. In other words, it is assumed that every possible operation has to be executed. Even though its conservative nature should be kept in mind, this assumption is a valid foundation for finding $O_{\text{IHF-1}}$ as a function of m and n_{IHF}.

Given that each possible operation must be performed, Algorithm 2 reduces mainly to two nested for-loops. While the outer loop performs a static number of n_{IHF} iterations, the number of iterations of the inner loop is dynamic. It increases by one for each iteration of the outer loop. An example for $n_{\text{IHF}} = 6$ is given in Table 4.1.

For the here conducted analysis of the complexity of Algorithm 2, the number of inner loop iterations is a appropriated measure of $O_{\text{IHF-1}}$ due to two reasons. First, the main amount of work of Algorithm 2 is executed inside the inner loop, in form of XOR and shift operations. Second, as can be seen from Table 4.1, the number of inner loop iterations increases rapidly with n_{IHF}. As a result, the number of operations performed in the inner loop becomes fast dominating. Based on this, an indicator of the number of operations required by Algorithm 2, $O_{\text{IHF-1}}$ can be expressed as,

Table 4.1: Number of inner loop iterations for IHF-1, for the example of $n = 6$

State of the algorithm	**Number of inner loop iterations**
1st outer loop iteration	-
2nd outer loop iteration	1
3rd outer loop iteration	2
4th outer loop iteration	3
5th outer loop iteration	4
6th outer loop iteration	5
Total	**15**

$$O_{\text{IHF-1}} = \sum_{i=0}^{n_{IHF}-1} i = \frac{n_{IHF} \cdot (n_{IHF} - 1)}{2}, \tag{4.6}$$

where the right hand side follows from [36]. This means that $O_{\text{IHF-1}}$ increases approximately quadratic with the number of input bits, n_{IHF}. Combining Equation 4.4 and Equation 4.6, the lower bound of $O_{\text{IHF-1}}$ can be expressed as,

$$O_{\text{IHF-1}} \geq \frac{(m + \beta) \cdot (m + \beta - 1)}{2}. \tag{4.7}$$

Equation 4.7 depicts as such the minimum amount of operations that have to be performed by Algorithm 2. It should be noted that, with regards to Equation 4.3, the presented lower bound can only be reached if the input to the IHF already is perfectly random. As such, the practical interest in Equation 4.7 is reduced. However, for the here presented analysis, considering this lower bound gives some valuable insight of the general development of $O_{\text{IHF-1}}$.

In contrast to Algorithm 2, the number of operations for Algorithm 3, $O_{\text{IHF-2}}$, is explicitly defined by the number of input and output bits. Algorithm 3 states that for each of the m output bits, n_{IHF} simple XOR and AND operations have to be performed. Hence, it is reasonable to formulate,

$$O_{\text{IHF-2}} = m \cdot n_{IHF}. \tag{4.8}$$

Combining Equation 4.4 and Equation 4.8 in the same manner as for $O_{\text{IHF-1}}$, it follows that $O_{\text{IHF-2}}$ is lower bounded by,

$$O_{\text{IHF-2}} \geq m \cdot (m + \beta). \tag{4.9}$$

Until this point, the complexity of the IHF algorithms has been considered in a general manner. However, it is of interest to be able to directly compare the complexity of the IHF approaches to the VNC solution. Since the VNC is only a valid solution if the used entropy source has von Neumann conditions, as defined in Section 3.2,

it is, for the sake of comparison, reasonable to adopt this source characteristic also for the IHF algorithms.

In addition, mutual independence between the output bits is assumed, since, as stated before, this is a desirable characteristic for the use of the VNC (see Section 2.4.1) and it simplifies the analysis of the IHF-algorithms. In the following, the complexity analysis of the IHF is therefore repeated for an entropy source that generates an output bit-stream of mutual independent bits, with a bias, specified by $p = \Pr(x_i = 0)$.

Under the assumption that the input bits are mutual independent and that p is a constant over n_{IHF} input bits, each input bit contains the same amount of min-entropy per bit, κ_b (see Equation 2.3). Using Equation 2.16, the relation of n_{IHF} and κ depicted in Equation 4.3 can thus be restated more specific as,

$$\kappa = n_{IHF} \cdot \kappa_b. \tag{4.10}$$

Assuming, as above and without loss of generality, that p is in the range [0 0.5] and recalling that the min-entropy is equal to the smallest information content of the considered data source X (see Equation 2.1 and Equation 2.3), the min-entropy per bit can be computed by,

$$\kappa_b = h(x_i = 1) = \log_2(\frac{1}{1-p}), \tag{4.11}$$

where $h(x_i = 1)$ is the information content associated with observing that an arbitrary bit x_i takes the value 1. Combining Equation 4.4, Equation 4.10 and Equation 4.11, it is possible to express the required numbers of input bits for the IHF, n_{IHF}, as a function of the requested number of output bits, m, as

$$n_{\text{IHF}} = \frac{\kappa}{\kappa_b} = -\frac{m+\beta}{\log_2(1-p)}. \tag{4.12}$$

Thus, for a source with mutual independent and identically distributed output bits, n_{IHF} increases linearly with m and β. As p increases from 0 to 0.5, $\log_2(1-p)$ decreases from 0 to -1. Over this short range, it is reasonable to approximate the logarithmic character of $\log_2(1-p)$ as a linear decrease. Accepting this simplification, n_{IHF} increases in an inversely proportional manner with decreasing p.

By combining Equation 4.6 and Equation 4.12, the number of operations for Algorithm 2, $O_{\text{IHF-1}}$, can be restated as a function of m, p and β,

$$O_{\text{IHF-1}} = \frac{1}{2} \cdot \Big(\big(\frac{m+\beta}{\log_2(1-p)}\big)^2 + \frac{m+\beta}{\log_2(1-p)} \Big). \tag{4.13}$$

Since the quadratic part of the right hand side of Equation 4.13 becomes fast dominating, it is reasonable to describe the increase of $O_{\text{IHF-1}}$ as quadratically with both m and β. In addition, accepting the simplification of $\log_2(1-p)$ as it has been stated for Equation 4.12, $O_{\text{IHF-1}}$ grows approximately with $\frac{1}{p^2}$, for an increasing bias.

Finally, by inserting Equation 4.12 into Equation 4.8, $O_{\text{IHF-2}}$ can be expressed as,

$$O_{\text{IHF-2}} = -\frac{m^2 + m \cdot \beta}{\log_2(1-p)}. \tag{4.14}$$

It follows that $O_{\text{IHF-2}}$ increases approximately quadratically with m and linearly with β. Using the same assumptions about the growth of $\log_2(1-p)$ as for Equation 4.12 and Equation 4.13, $O_{\text{IHF-2}}$ increases inversely proportional with a decreasing p.

4.1.2 Discussion on the growth of complexity

Based on the in Section 4.1.1 presented analysis, it is now possible to relate the complexity of the different post-processing algorithms to each other. Since the IHF can be realized both by Algorithm 2 and Algorithm 3, it is natural to begin a discussion on the complexity by comparing the complexity of Algorithm 2 and Algorithm 3 in order to find out which algorithm yields a more suitable implementation.

Measures of the number of operations required by Algorithm 2 and Algorithm 3 are presented in Equation 4.6 and Equation 4.8, respectively. Both equations are functions of the number of input bits. While it has been pointed out at the beginning of this section that $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ as functions of n_{IHF} are of rather reduced interest for the purpose of this report, it is nevertheless possible to gain some useful insight by considering both equations together. By equalizing Equation 4.6 and Equation 4.8, a situation in which both algorithms perform the same amount of operations can be expressed as,

$$\frac{n_{IHF}^2 - n_{IHF}}{2} = n_{IHF} \cdot m, \tag{4.15}$$

By solving Equation 4.15 for n_{IHF} it follows,

$$n_{IHF} = 2 \cdot m + 1. \tag{4.16}$$

In other words, both algorithms have the same complexity if the number of input bits is roughly twice as large as the number of output bits. However, if n_{IHF} exceeds $2 \cdot m + 1$, $O_{\text{IHF-1}}$ is larger than $O_{\text{IHF-2}}$. Thus, in a situation in which the IHF operates on more than twice as many input bits than requested output bits, using Algorithm 3 yields a solution with reduced complexity compared to Algorithm 2. In the same manner, Algorithm 2 depicts the more appropriated approach if $n_{IHF} < 2 \cdot m + 1$. Figure 4.1 shows the results of plotting $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ as functions of n_{IHF}, for the example of $m = 32$. It is worth noticing, that the difference between $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ increases the more n_{IHF} differs from $2 \cdot m + 1$.

Having established a relation of Algorithm 2 and Algorithm 3 in terms of n_{IHF} and m, it remains to investigate the impact of the tuning parameter β. Even though it is not possible to find a general expression of $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ in terms of β, Equation 4.7 and Equation 4.9 define the lower limits of the number of operations for both algorithms with respect to m and β. Repeating the above presented procedure and equalizing Equation 4.7 and Equation 4.9 yields,

$$\frac{(m+\beta) \cdot (m+\beta-1)}{2} = m \cdot (m+\beta). \tag{4.17}$$

By solving the equation for m, it can be stated that the lower bounds of $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ are equal if,

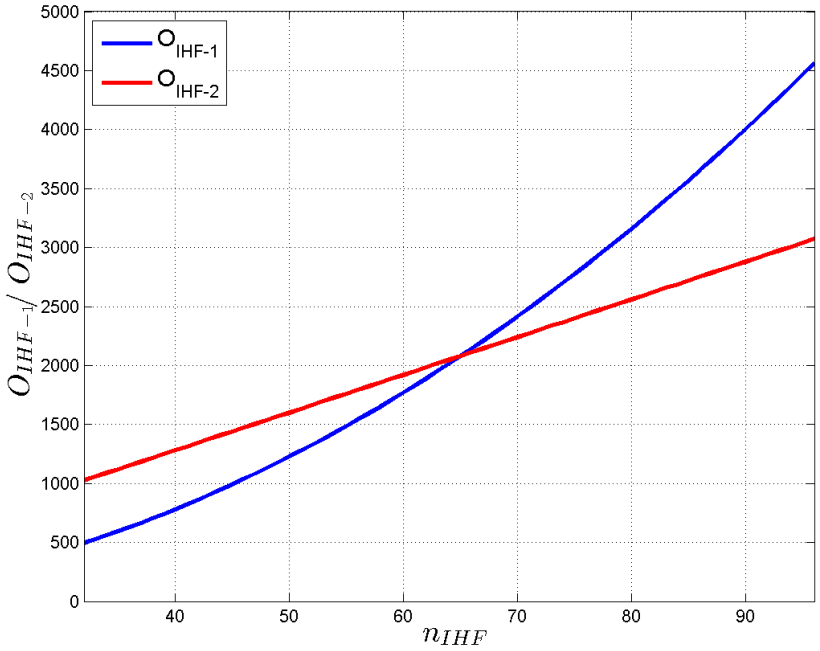


Figure 4.1: Number of operations of the IHF-algorithms as functions of n_{IHF}

$$m = \beta - 1. \tag{4.18}$$

Thus, if Equation 4.18 is satisfied, the complexity of Algorithm 2 and Algorithm 3 have the same lower bound. However, in the case that β exceeds the number of output bits by more than one, Algorithm 3 has a smaller lower bound than Algorithm 2 and can therefore theoretically yield a less complex implementation. On the other hand, if $m > \beta - 1$, Algorithm 2 is bounded by a lower number of operations than Algorithm 3. Figure 4.2 illustrates this fact for $\beta = 20$, which could, for example, corresponds to $\epsilon = 0.0625$ and $t = 1$.

It is worth noticing that, while $O_{\text{IHF-2}}$ is lower bounded by zero for $m = 0$, the bound of $O_{\text{IHF-1}}$ has an β related offset. By considering Equation 4.7 the offset can be evaluated to be $\frac{\beta^2-\beta}{2}$, which equals 190 for the in Figure 4.2 depicted situation.

After having established some general relations between the complexity of Algorithm 2 and Algorithm 3, the focus of the discussion turns to considering the efficiency of the VNC in comparison to the IHF. It is convenient to start by comparing the number of required input bits. For an entropy source that produces mutual independent bits with a bias defined by p, the numbers of input bits, n_{VNC} and n_{IHF}, are presented in Equation 4.1 and Equation 4.12, respectively. It follows from these equations that both n_{VNC} and n_{IHF} increase linearly with m. Equation 4.12 shows further that n_{IHF} additionally increases linearly with β. However, since the VNC does not have a comparable tuning parameter, it is reasonable to depict β as a constant throughout the rest of this discussion.

Figure 4.3 plots n_{VNC} and n_{IHF} as functions of m for $\beta = 20$ and different values

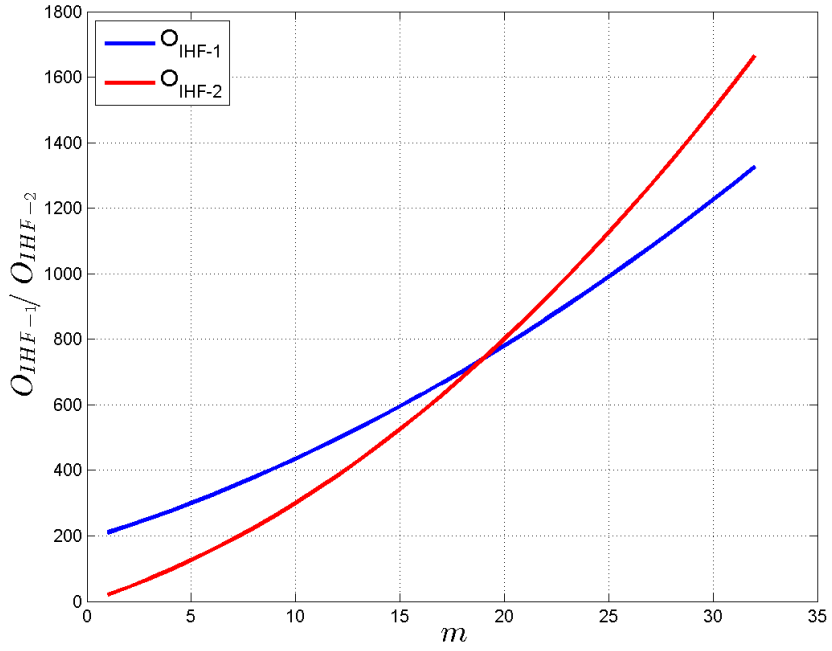


Figure 4.2: Lower bounds of the number of operations of the IHF-algorithms as functions m

of p. It is essential to note that the existence of β yields n_{IHF} to have a constant offset, which increases with decreasing p. Thus for small values of m, n_{VNC} is smaller than n_{IHF}. However, it can also be seen from Figure 4.3 that n_{VNC} increases with a steeper gradient than n_{IHF} for the same bias p. As a result, for large values of m, the IHF requires less input bits than the VNC. For example, for the in Figure 4.3 depicted situation, the VNC is requiring a lower number of input bits than the IHF for $p = 0.1$ as long as $m < 29$. If $m > 29$, n_{VNC} exceeds n_{IHF} by an with m increasing difference.

As depicted in Figure 4.3, the gradients for both n_{VNC} and n_{IHF} increase as p approaches zero. From Equation 4.1 and Equation 4.12, it follows that the gradients of n_{VNC} and n_{IHF} are $\frac{1}{p \cdot (1-p)}$ and $\frac{-1}{\log_2(1-p)}$, respectively. It is worth noticing that the gradients of n_{VNC} and n_{IHF} can be interpreted as the number of input bits that has to be added to the input in order to generate a single extra output bit. Figure 4.4 shows the gradients of n_{VNC} and n_{IHF} as functions of p. As stated during the introduction of Equation 4.1 and Equation 4.12, both gradients grow approximately inversely proportional with p. In addition, the gradient of n_{VNC} is constantly larger than the corresponding gradient of n_{IHF}, which is in accordance to the observations made for Figure 4.3. To give a theoretical example, for $p = 0.5$ four additional input bits would have to be passed to the VNC in order to generate a single extra output bit. In contrast, the IHF requires only a single additional input bit per extra output bit, given the same input bias.

Besides the gradients themselves, Figure 4.4 does also depict the ratio of the gradients, $\frac{1}{p \cdot (1-p)} \Big/ \frac{-1}{\log_2(1-p)}$, which can be interpreted as a measure of the efficiency of the IHF in terms of input bit requirements compared to the IHF. Reconsidering the

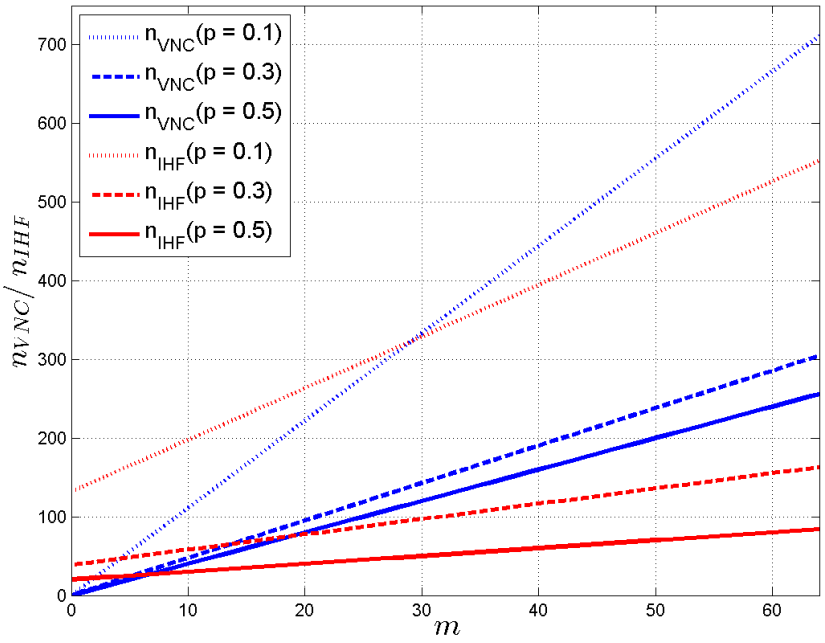


Figure 4.3: Number of input bits required by the VNC and the IHF as functions of m

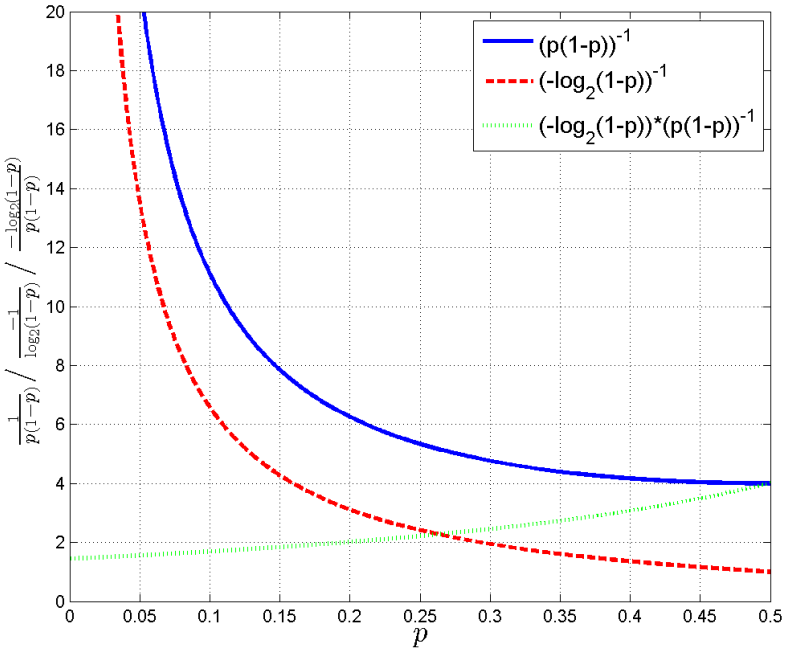


Figure 4.4: Slopes of n_{VNC} and n_{IHF} as functions of p

above presented example with $p = 0.5$, the IHF requires only a single extra input bit to create an additional output bit, whereas the VNC requires four bits in the same situation. Thus, the ratio between the gradients is four. One can therefore say that the IHF is four times more efficient in converting input bits to output bits than the VNC. Figure 4.4 shows that the ratio of the gradients reaches its maximum for the discussed example of $p = 0.5$. The ratio decreases with p and approaches a minimum of approximately 1.445 for a close-to-constant input.

Finally, it is worth noticing that, by Equation 4.12, the gradient of n_{IHF}, shown in Figure 4.4, also describes the growth of the β-related offset of n_{IHF}. Thus, the offset of n_{IHF} increases fast with an increasing bias of the input (decreasing p). This observation is in accordance to what is depicted in Figure 4.3.

To complete the comparison of n_{VNC} and n_{IHF}, the following can be concluded: Even though n_{IHF} has a β-related offset, due to the fact that its gradient is lower than the gradient of n_{VNC} the IHF is a more suitable approach in terms of input bit efficiency, at least for a more practical amount of output bits. However, if β increases, the number of output bits required in order for the IHF to be the more efficient solution increases by the same amount. Further, both the offset and the gradient of n_{IHF} increase vastly as p decreases. As a result, the VNC might be the preferable approach for a strongly biased source and a large β.

The number of input bits that are required by the VNC and the IHF are important measures of the efficiency of TRNG solutions that use one of these algorithms. However, since the input bits are generated by the entropy source of the TRNG, both n_{VNC} and n_{IHF} are rather indicators of the energy requirements of the used source. To get a impression of the energy performance of the post-processing algorithms themselves, it is therefor of interest to compare the number of operations required by the three different algorithms. Assuming the same source conditions as for the above presented comparison of n_{VNC} and n_{IHF}, Equation 4.2, Equation 4.13 and Equation 4.14 can be used to express O_{VNC}, $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$, respectively. Figure 4.5 shows O_{VNC}, $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ as functions of m for different source biases (defined by p) and the tuning parameter of the IHF fixed to $\beta = 20$.

As expected by considering the corresponding equations, the number of operations for the two IHF solutions, $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$, increases approximately quadratically with m, while O_{VNC} increases linearly. Since O_{VNC} does not have an offset and m is an integer, it follows that O_{VNC} is smaller than both $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ for all valid values of β, and that the difference between the number of operations executed by the VNC and the corresponding measures of the IHF solutions increases vastly for an increasing m.

For example, given an input bias described by $p = 0.3$, the VNC has to perform approximately 38 operations in order to create 16 output bits, while Algorithm 2 and Algorithm 3 have to execute 2412 and 1119 operations, respectively. Increasing the required output to $m = 32$ bits yields O_{VNC} to increase to roughly 76 operations. However, for the same output $O_{\text{IHF-1}}$ increases to 5055 and $O_{\text{IHF-2}}$ becomes 3234.

It can also be seen from Figure 4.5 that, as for the number of input bits in Figure 4.3, the rates at which the numbers of operations grow increase with decreasing p.

In contrast to identifying the VNC as the most appropriated approach in terms

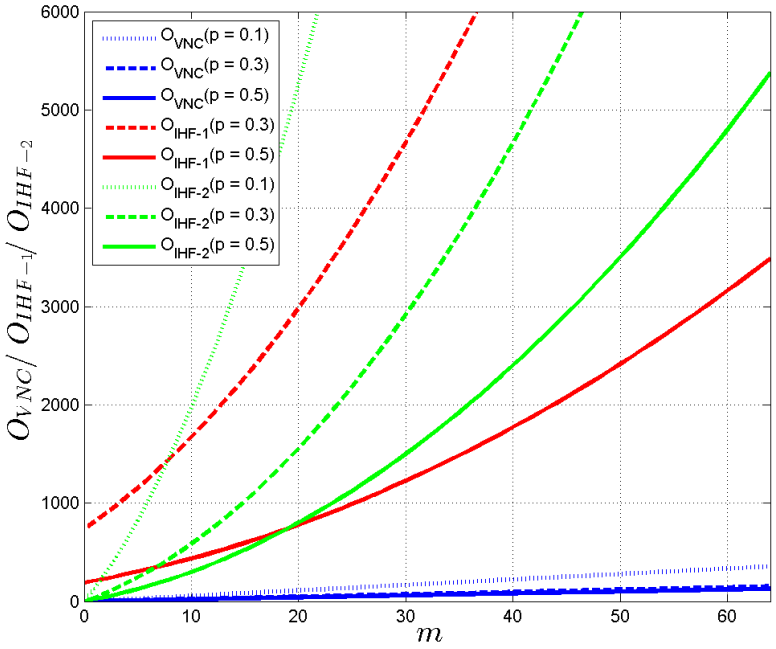


Figure 4.5: Number of operations of the VNC and the IHF-algorithms as functions of m

of the amount of required operations, it cannot generally be determined whether Algorithm 2 or Algorithm 3 is more efficient for the given input characteristics. Starting by considering $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ for $p = 0.5$, it should be noted that $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ are in this situation equal to their lower bounds as described in Equation 4.7 and Equation 4.9, since Equation 4.3 reduces to an equality. Thus, by recalling the observations made with regards to Equation 4.18, it is expected that $O_{\text{IHF-2}}$ exceeds $O_{\text{IHF-1}}$ for $m > \beta - 1$. This is in accordance with the situation depicted in Figure 4.5. Hence, if the input characteristics allow the IHF to operate closely to the in Equation 4.4 defined bound of n_{IHF}, it follows that $O_{\text{IHF-1}} < O_{\text{IHF-2}}$ if m exceeds β.

However, it has been stated with regards to Equation 4.16 that Algorithm 3 requires less operations than Algorithm 2 if the number of input bits exceeds approximately twice the size of the number of output bits. As can be seen from Figure 4.4, the IHF requires two additional input bits in order to generate one extra output bit for $p \approx 0.3$. This means that for $p = 0.3$, n_{IHF} must be approximately twice the size of m, even in the unrealistic case of $\beta = 0$. In other words, $O_{\text{IHF-2}}$ must be smaller than $O_{\text{IHF-1}}$ if p decreases below roughly 0.3, independently of β. Even though Figure 4.5 uses $\beta = 20$, it illustrates this point, as $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ grow in a fairly similar manner for $p = 0.3$.

An additional factor that works in favor of Algorithm 3 is the β-related offset of $O_{\text{IHF-1}}$. While the offset is considerably small for values of p close to 0.5, it follows from Equation 4.13 that it increases roughly with $\frac{1}{p^2}$ as p decreases. Thus, the offset increases fast as the input becomes more biased, as shown in Figure 4.5.

Finally, the influence of the input bias on the number of required operations

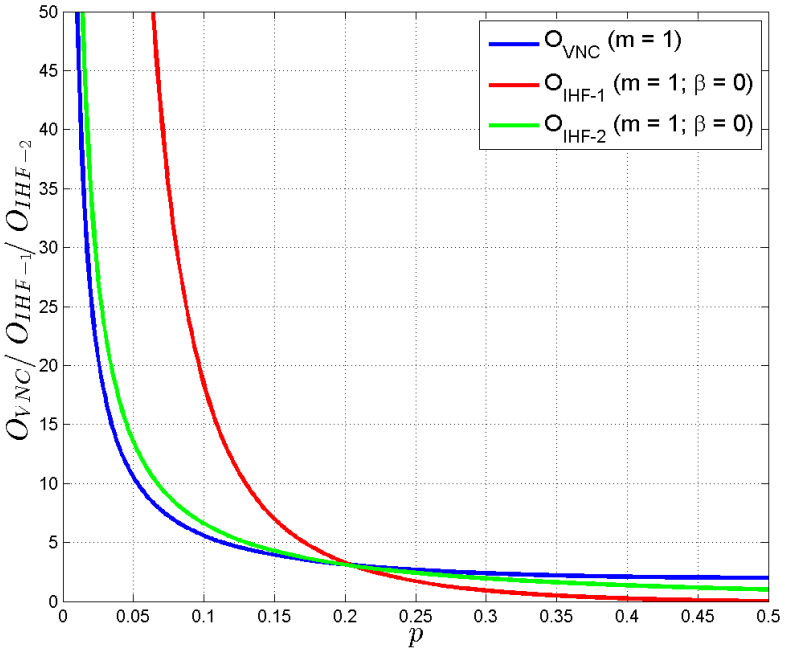


Figure 4.6: Indication of the number of operations for the VNC and the IHF-algorithms as functions of p

should be considered. To do so, Figure 4.6 shows the plots of O_{VNC}, $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ as functions of p, for fixed values of m and β. From the discussion of Figure 4.5, it is known that especially $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$ grow fast with m and β. Thus, in order to keep the focus of Figure 4.6 on the development of the functions in terms of p, it has been chosen to use $m = 1$ and $\beta = 0$. The reader should nevertheless keep in mind, that, while valid for this illustration, $m = 1$ and $\beta = 0$ are in practice unrealistic values.

It can be seen from Figure 4.6 that all three functions grow in similar manner and increase vastly as p approaches zero. This is in accordance to the discussions of Equation 4.2, Equation 4.13 and Equation 4.14, where it has been stated that O_{VNC} and $O_{\text{IHF-2}}$ increase approximately with $\frac{1}{p}$, while $O_{\text{IHF-1}}$ increases with roughly $\frac{1}{p^2}$. As a result, the growth of O_{VNC} and $O_{\text{IHF-2}}$ is nearly identical. $O_{\text{IHF-1}}$, on the other side, increases slightly faster for small p-values. For instance, while O_{VNC} and $O_{\text{IHF-2}}$ equal 20 operations for $p \approx 0.025$ and $p \approx 0.034$ respectively, $O_{\text{IHF-1}}$ reaches the same amount of operations already for $p \approx 0.096$.

In conclusion, the VNC is the preferable solution with regards to the number of operations that have to be executed. This is due to the fact that O_{VNC} has a linearly increase over m instead of the quadratic increase of $O_{\text{IHF-1}}$ and $O_{\text{IHF-2}}$. An increase in the input bias has approximately the same influence on the VNC as it has on the IHF realized by means of Algorithm 3. Algorithm 2 performs slightly worse for small p-values. A general statement on whether a IHF solution for von Neumann conditions should be realized by using Algorithm 2 or Algorithm 3 cannot be made. Algorithm 2 is, dependent on m, likely to achieve a lower complexity for p close to 0.5 and small values of β. However, for an increasing β and decreasing

p-values, Algorithm 3 becomes more efficient. If p decreases below approximately 0.3, Algorithm 3 uses less operations than Algorithm 2, independently of the chosen tuning parameter, β.

4.1.3 Choice of an IHF algorithm suited for implementation

In contrast to the single solution for the VNC, two possible solutions of the IHF have been presented in form of Algorithm 2 and Algorithm 3. As an implementation of both algorithms would be too time consuming for the frame of this project, it is therefore first necessary to identify which of the two algorithms is better suited for a hardware implementation. Besides comparing the algorithms directly, it is reasonable to based this decision on the findings on the complexity of the two solutions, as discussed in Section 4.1.1 and Section 4.1.2. However, it is worth to recall that the number of input bits required by the solutions, n_{IHF}, is equal for Algorithm 2 and Algorithm 3. This is due to the fact that n_{IHF} is dependent to the amount of min-entropy requested by the extractor (see Equation 4.3). The min-entropy is related to the different parameters of the IHF by means of Equation 2.34, which is generally true for any IHF solution. Thus, to decide which of the two solutions is more suited for a implementation, it is sufficient to only focus on aspects that directly influence the size of the implementation of the IHF.

A first indication follows directly from comparing Algorithm 2 and Algorithm 3, as stated in Section 2.4.2. While Algorithm 3 is a rather simple algorithm, which executes the same operations for every output bit, Algorithm 2 is more complex and executes different operations, depending on the concrete input $\overrightarrow{x}$ and the chosen public parameter $\overrightarrow{\pi}$. As a result, Algorithm 2 would require a larger control overhead than the one needed to execute Algorithm 3. It is reasonable to assume that this overhead is close to constant for varying lengths of the input and the output. Hence, it becomes negligible as the algorithms operate on an increasing amount of data and has therefore not be considered during the analysis of the complexity in Section 4.1.1. However, for small inputs and outputs, the difference in the amount of control logic should be taken into consideration.

Another aspect that votes in favor of Algorithm 3 is the fact that the algorithm generates the output $\overrightarrow{y}$ bit-wise. As a result, only one single bit, y_j, has to be kept track of between single operations. In contrast, Algorithm 2 computes all elements of $\overrightarrow{y}$ in parallel. It can be seen from the definition of Algorithm 2, that this makes the use of an n_{IHF}-bit interim variable, $\overrightarrow{x'}$, necessary. For a hardware implementation of the algorithm, such an interim variable is likely to cause an additional n_{IHF}-bit memory element in comparison to Algorithm 3.

Turning the attention to the results presented in Section 4.1.2, it has been stated with respect to Equation 4.16 that Algorithm 2 requires more operations than Algorithm 3 if the number of input bits, n_{IHF}, is roughly twice as large as the number of output bits, m. Obviously, making a decision based on this property with complete certainty would require the knowledge of the concrete parameters used for the implementation of the IHF. For the purpose of this project, it is desirable to avoid such a commitment to a certain tuning of the IHF at this stage. However, it is worth to recall, that, by Equation 4.4, the n_{IHF} input bits must contain $\kappa = m + \beta$ bits of min-entropy. Thus, even by using the unrealistic tuning parameter $\beta = 0$, each

input bit would have to contain an average of 0.5 bits of min-entropy

While such an amount of entropy can be delivered by high-entropy sources, the source that is in the focus of this project is rather unlikely to achieve this amount of entropy per bit. For instance, considering the results of Section 3.3, p could be approximated as 0.29. Hence, even under the assumption that the generated bits would be independent, by Equation 2.3, the min-entropy of the source would be roughly 0.49 per bit. Recalling that based on the performed analysis, dependencies have been observed in the source data, the actual amount of min-entropy is further reduced. Based on this, for the context of this report, Algorithm 3 seems the appropriate choice for an implementation based on the number of operations that have to be performed.

This statement is further strengthened by briefly considering the effect of the tuning parameter of the IHF, β, on the complexity of Algorithm 2 and Algorithm 3. Generally, β is not considered in detail in this report, as the VNC lacks such a tuning parameter and focusing on other aspects seems therefore more appropriated for a comparison of the two solutions. However, in order to achieve a working implementation of the IHF, it is necessary to find an approximated order for a minimum value of β for which the IHF is functional. In Section 4.1.2, $\beta = 20$ has been chosen as an example, which, by Equation 4.5 corresponds to $\epsilon = 0.0625$ and $t = 1$. A security parameter of $t = 1$ correlates to a low level of resilience of the IHF, which is reasonable for this project, as, again, the VNC cannot be tuned for security.

Considering the quality parameter, $\epsilon = 0.0625$ is rather high value. Equation 2.33 states that for $\epsilon = 0.0625$, the statistical distance (see Equation 2.6) between the distribution of the output of the extractor, $\mathcal{P}_Y$, and the corresponding uniform distribution, $\mathcal{U}_Y$, is upper bounded by 0.0625. With respect to the fact that the output of a TRNG should be as uniform distributed as possible, a statistical distance of 0.0625 seems large, especially for the strict requirements of TRNGs in a cryptographic context. Equation 2.33 states further that, with a probability of ϵ, selecting the public parameter π at random does not yield a functional extractor design. In other words, for $\epsilon = 0.0625$, roughly 6 out of 100 public parameters will not satisfy $\text{dist}(\mathcal{P}_Y, \mathcal{U}_Y \leq \epsilon)$. It is therefore common to use smaller values for ϵ. An example is given in [14], which sets $\epsilon = 2^{-35}$. However, keeping the energy performance of the design in mind, it is at this point chosen to accept $\epsilon = 0.0625$, and rather decrease the value if latter performed test with focus on randomness (see Chapter 5) indicate the necessity. Thus, for the moment it is assumed that $\beta = 20$. Nevertheless, it should be stressed that this is considered to be the minimum value of β.

Using this assumption, it has been stated with regards to Figure 4.2 and Equation 4.18 that, even though the lower bound of the number of operations required by Algorithm 2, $O_{\text{IHF-1}}$, increases slower with β for a fixed m, the lower bound of Algorithm 3 is smaller as long as $m < \beta$ (see Equation 4.18). As mentioned before, it is omitted to determine the number of output bits, m, at this stage of the implementation process. However, since for both algorithms the number of operations that have to be performed increases vastly with m, it is reasonable to assume that m is chosen to be less than 20 bits. Thus, also the lower bound of the number of operations indicates that Algorithm 3 is the preferable algorithm for the purpose of this project.

In conclusion, Algorithm 3 appears to be more suitable for an energy efficient hardware implementation than Algorithm 2, based on the number of operations, the required control logic and additional memory requirements. The rest of the report focuses therefore on a possible implementation of the IHF by means of Algorithm 3.

4.2 Implementation of Post-Processing Algorithms

After Section 4.1 has established a basic understanding of the complexity of the different discussed algorithms, the focus of this section is on the implementation of both a VNC and a IHF solution in SystemVerilog. The purpose of these implementations is twofold. First, implementing the algorithms allows to use them in combination with data gathered from the entropy source (Section 3.1) to simulate the behavior of a TRNG design. The output that results from such simulations can then be used to verify the true random character of the design by applying it to the NIST test bench, presented in Section 2.2.2. Second, concrete power estimates can be derived from the implementations.

On the first sight, implementing the algorithms, verifying their random behavior in connection with an entropy source and estimating their power performance appear like three clearly separated processes. However, in practice, the three processes are interactive, as illustrated in Figure 4.7. During the design phase of the implementation, the RTL description of a given post-processing algorithm is generated, using SystemVerilog. To test if the RTL implementation matches the algorithmic description of the post-processor (as given in Section 2.4), the RTL design has to be *logical verified*. This is done, by running RTL simulations with a known input and evaluating whether the outcome matches the expectations. The for the simulations necessary setup (*simulation routines* and *testbenches*; see Section 4.2.1) are in this project created in SystemVerilog. As such, their generation can simplified be considered to be a part of the design phase. All RTL simulations are in this project performed using *Incisive* (commonly referred to as *NCSim*) from *Cadence*.

When the logical behavior of the created design is verified, the RTL implementation can be used to evaluate the post-processor in connection with the entropy source for randomness. For this purpose, it is necessary to first generate the output of the post-processor, which then can be passed on to, for instance, the NIST test suite. This can be easily achieved by once more running RTL simulations similar to those used for the logical verification of the design. It is therefore possible to reuse most of the above mentioned simulation setup. However, instead of using a known input, data generated by the entropy source of Section 3.1 is passed to the simulation as an input.

The RTL description of a design does not contain any information of the technology used for the implementation. Since, as mentioned in Section 2.5, the power performance of a digital design is directly dependent on the underlying technology, the technology has to be taken into consideration in order to derive power estimations of the implementation. This is achieved by *synthesizing* the design, which

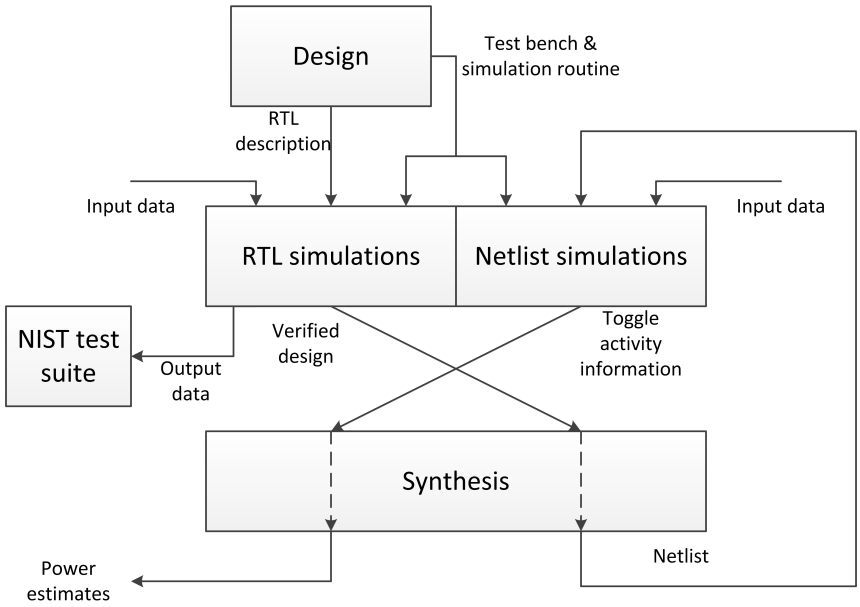


Figure 4.7: Work flow of the post-processor implementation process

transforms the RTL description into a technology dependent *netlist*. The synthesis tool used during this project is the *Encounter RTL Compiler* from *Cadence*.

While it is possible to derive power estimates based on the created netlist, it is stated in Equation 2.47 that the switching activity, γ, of the signals in the implementation affect the dynamic power performance of the design. Since γ depends on the statistical characteristics of the applied input data, to include the switching activity in the power estimation a third round of simulations has to be run. The setup of the used simulations is basically equivalent to the one used for the simulations described above. The main difference is that the third round simulations are run using the netlist description of the design instead of the RTL implementation. As the focus of these simulations is on the switching activity of the signals and not on the generated output values, it is not of importance whether the used input data is known or derived by using an entropy source. However, in order to get significant power estimates the used input data should yield to a switching activity that is typical for the design.

In order to execute the in Figure 4.7 described work flow, a system setup has been created, using a server environment at *Norwegian University of Science and Technology* (NTNU). Even though the setup has originally been implemented for the here presented project, many aspects are of a rather general nature and the setup can therefore serve as a starting point for other projects using a similar work flow. An introduction to the system setup is presented in Appendix E.

Based on the in Figure 4.7 presented work flow, the rest of this section is structured in the following manner: Section 4.2.1 introduces some general aspects of the design and the simulation setup that are alike for both post-processors. A detailed description of the implementation of the VNC and the IHF including the logical verification of their behavior is presented in Section 4.2.2 and Section 4.2.3, respectively.

The presentation of the simulations used to evaluate the ability of the post-

processors to generate a random output is postponed to Chapter 5. In the same manner, Chapter 6 covers the simulations concerned with the energy performance of the designs.

A detailed discussion of the synthesis process is omitted in this report, due to several reasons. First, synthesis is generally considered to be a rather complicated design process, which makes use of complex computer-aided design tools. As such, a detailed discussion is out of the scope of this report. Second, the synthesis process is highly dependent on the underlying technology. Thus, focusing on this design step would be inconsistent with the rather general nature of this project. Third, most of the technology related information are confidential and can therefore not be presented in the context of this report. For instance, revealing the used technology library would prohibit the possibility to present concrete values of the power estimates. The latter, however, is considered to be of more interest for the purpose of this report. As a result, this report restricts itself to present only the basic setup and results of the synthesis process. For all implementations in this project, the synthesis process has been executed using a standard 65 nm technology.

4.2.1 General aspects of the design and simulation process

To be able to design meaningful implementations of a TRNG post-processing algorithm, it is necessary to define a system environment. In other words, it has to be established how the TRNG is build up and how it communicates with other surrounding systems.

In general, the system architecture of an MCU must be considered to be rather complicated. For example, resources are often shared. Using, for instance, the ADC as an entropy source, the ADC is likely to be usable by other modules as well and to have a wide range off possible settings. Thus, in order to be used by a TRNG, it would have to be ensured that the ADC is not used by another module and the ADC would have to be set up to work as an entropy source (see Section 3.1). Another example of the complexity of MCU systems is the fact that modules in an MCU in common not communicate directly but through data buses. This would make it necessary to adapt the here presented implementations to the communication protocol which accompanies the bus system. Several other examples exists and in general it can be stated that integrating an implementation into an MCU environment requires a substantial overhead in form of additional control logic.

However, while being aware of this fact, it has been chosen to design the in this section presented implementations for a much simpler system environment. This is mainly motivated by two arguments. First, different MCUs use different system architectures. Thus, by not integrating the presented implementations into a specific MCU architecture, the results are kept rather general. Second, it is reasonable to assumed that the VNC and the IHF would require roughly the same amount of system architecture related overhead. This means that for a comparison of the power consumption of the two approaches, this kind of overhead is not of interest.

Figure 4.8 shows the simple system model which is used during the implementation process. It is assumed that some kind of *superior system* exists, which, for instance, is controlled by some end-user. This superior system supplies a main clock signal to the TRNG, *sys_clk*, and is able to reset the device by pulling an asyn-

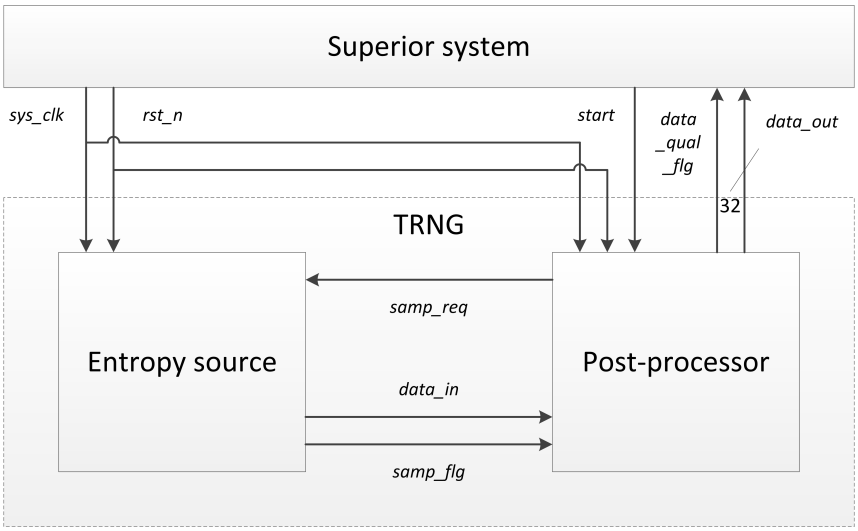


Figure 4.8: System environment for post-processor implementations

chronous reset signal, *rst_n*, to logic zero. The TRNG itself persists of two parts: the entropy source and the post-processing module which is to be implemented.

The communication between the superior system and the TRNG is executed through the post-processing module of the TRNG. As such, the post-processing can be considered to not only execute the post-processing of the data but to serve as the control unit of the TRNG as well. This is motivated by the fact that using the ADC as an entropy source, the ADC depicts a rather memoryless module which continuously produces a single bit output. The post-processing algorithms on the other side need at least a minimum of state control logic and it makes therefore sense to combine the control logic of the post-processor with the control unit of the TRNG. This will become clearer during the implementation processes in Section 4.2.2 and Section 4.2.3.

The superior system can start the TRNG by setting the synchronous signal *start* to high. This triggers the TRNG to generate 32 random bits. Even though the amount of output bits is in principle arbitrary, setting it to 32 bits is reasonable, since 32 bits is a commonly used word width for modern MCU architectures [5]. It has been mentioned in Chapter 1 that a larger amount of random bits is needed for the purpose of modern cryptography. The obvious solution is to run the TRNG several times and concatenate several 32-bit outputs. From Equation 2.16 it is known that as long as each 32-bit output can be considered as random and the outputs are dependent of each other, this concatenation is a random data string. Chapter 5 evaluates this aspect further.

Once the TRNG has been activated by the superior system, any activity of the *start* signal is ignored until the 32-bit output is generated. During the generation process, the post-processing module can request single bit data values from the entropy source by setting the *samp_req* signal to logic one. The entropy source reacts on this request by transmitting a single bit output over the *data_in* channel to the post-processor.

However, since the ADC based entropy source generates an output bit at a slower

rate than the clock cycles of the fast system clock (Table 3.3), it must be guaranteed that the post-processor does not read invalid data from *data_in*. In order to do so, the entropy source signals the arrival of new data on *data_in* by setting the flag-signal *samp_flg* high for the duration of one clock cycle of *sys_clk*. Finally, if the TRNG has generate the required 32 output bits, the post-processing module indicates this to the superior system by setting *data_valid_flg* to a logic one. The superior system can then simultaneously read the requested output from the 32-bit output channel *data_out*, before it eventually restarts the system by triggering *start*.

Table 4.2: List of input and output for the post-processing modules

	Name	Type	Bit width	Description
Superior system	*sys_clk*	Input	1	Main clock signal of the system
	rst_n	Input	1	Asynchronous negative reset signal
	start	Input	1	Synchronous flag, triggering generation of random data
	data_out	Output	32	32-bit word of random data
	data_valid_flg	Output	1	Synchronous flag, signaling to the system that *data_out* is valid and can be read
Entropy source	*data_in*	Input	1	Data from entropy source
	samp_flg	Input	1	Synchronous flag, signaling new data on *data_in*
	samp_req	Output	1	Signal, requesting data from source

Table 4.2 summarizes the different inputs and outputs of the post-processing modules. It is worth to recall that these signals are the same for both the VNC and the IHF implementation.

After the design of the post-processing modules is completed, it is necessary to be able to run simulations on the implementations. While the purpose of the simulations varies, it has been stated during the introduction of this section, that the setup of the different simulations are similar. Further, since both the VNC and the IHF have been designed with regards to the same system environment, it is reasonable to use a similar simulation setup for both post-processors. As a result, one main simulation setup is created for the purpose of this project, which can be adjusted to the given circumstances by means of minor modifications.

Figure 4.9 shows a schematic overview of the used setup. The implementation that is the subject of the test, frequently referred to as the *Device under Test* (DUT),

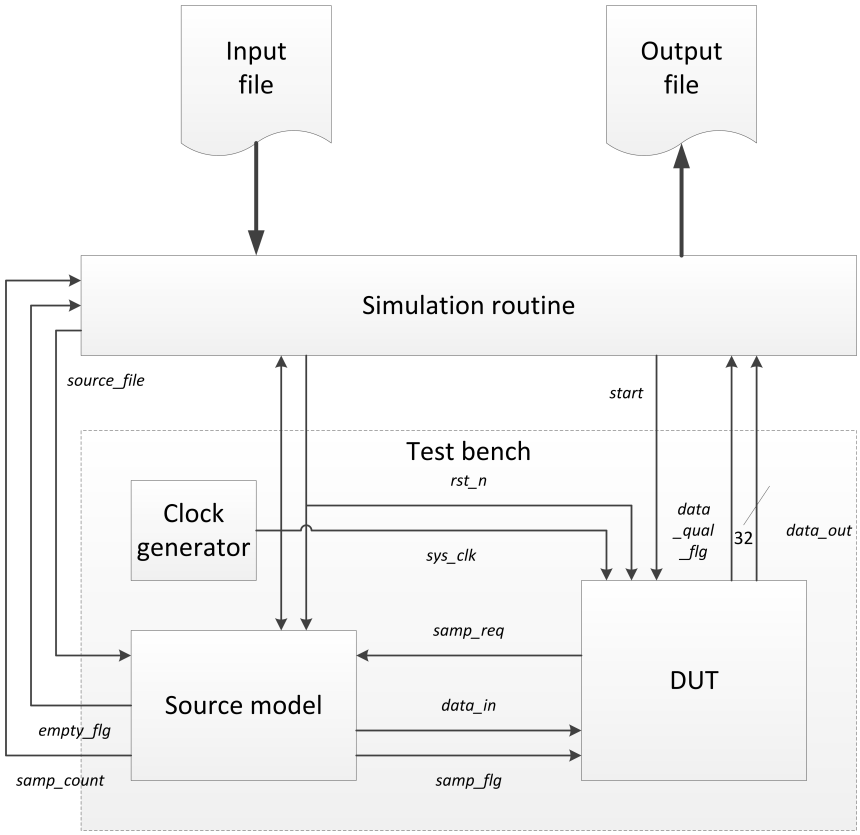


Figure 4.9: Simulation setup for post-processing implementations

is placed on a *test bench* together with the models of a clock generator and of an entropy source. The SystemVerilog codes for the two models and an example of a test bench are presented in Appendix D.4.2. The test bench is controlled by a *simulation routine*, which in many ways can be interpreted as the superior system of Figure 4.8. Even though, the exact behavior of the used simulation routine is different for each simulation, their principle build-up is identical. Their main objective is to read some input data from an external file and pass it to the source model. Then it activates the DUT and runs it once or several times in order to generate a specified number of output words. Finally, the simulation writes the gathered words to an external output file, which can be used to evaluate the DUT. An example of a simulation routine is presented in Appendix D.4.3.

Even though the simulation setup of Figure 4.9 is quite similar to the used system environment depicted in Figure 4.8, some differences exists. For instance, while the simulation routine controls the test bench in a similar manner as the superior system of Figure 4.8 controls the TRNG, the simulation routine does not provide the clock signal *sys_clk*. On the contrary, *sys_clk* is generated by the clock generator model of the test bench and passed to the simulation routine. The reason that the generation of the clock is left to the test bench is that it is practical to develop the simulation routine in a software-like high-level approach (see the example in Appendix D.4.3). Such an approach is in general less concerned about timing constraints and it is therefore reasonable to leave the generation of a clock signal to the test bench models which are more closely related to a hardware description. However, a minimum of

synchronization between the test bench and the simulation routine is necessary and *sys_clk* must therefore be passed to the simulation routine.

An additional difference between the superior system of Figure 4.8 and the simulation routine of Figure 4.9 is the communication with the entropy source. In the system model of Figure 4.8, the superior system does not communicate with the entropy source besides of resetting the module. In contrast, the simulation routine passes some information about the input file, *source_file*, to the source model, which is used by the model to read the data from the input file. This approach makes it possible to reuse the test bench for different simulation routines with different input files, without having to adapt the source model. The source, in turn, reports the number of bits, *samp_count*, that have been read from the input file and passed on the the DUT and signals if the whole file has been read by setting the *empty_flg* flag signal. This signals can be used by the simulation routine to, for example, abort the simulation if the number of requested output bits exceed the amount that can be generated from a given input file.

As mentioned above, the presentation of the simulations for the purpose of randomness and power evaluations are postponed to Chapter 5 and Chapter 6, respectively. Hence, the focus of this section is on simulations for logical verification. For the purpose of this project, the main focus of the logical verification step is to verify that the implementations work as defined by their respective algorithms. This is partly motivated by the simplicity of the in Figure 4.8 presented system environment, and partly by the fact that the designs have been implemented for the purpose of evaluating the respective algorithm. The later aspect makes it highly important that a given design behaves in accordance to its algorithmic specification. In contrast, it is not of particular interest for this project to test if the system behaves well defined if, for example, control signals are applied in an unexpected manner. This would normally be verified in an industrial implementation context.

Thus, focusing mainly on the functionality of the implemented modules, it has been chosen to execute two verification simulations per implementation. First, considering only the execution of the corresponding algorithm, a short known input is applied to the DUT, generating only a few output bits. With regards to the examples presented for during the introduction of the VNC and the IHF (see Section 2.4.1 and Section 2.4.2), it is suitable to use an 8-bit input resulting in 2 output bits. Based on the short input length, the complexity of the executed computations and control sequences is reasonably low. This makes it possible to analyze the behavior of the DUT in detail, for instance, by considering the waveforms that result from the simulation.

If this first simulation yields the conclusion that the DUT behaves as desired, a second simulation is executed. During this simulation, a larger known input is applied to the DUT resulting in four output words. It is worth noticing that the number of input bits required to generate 4 32-bit words varies between the VNC and the IHF. The motivation behind this simulation is twofold. On one hand, successfully generating several words shows that the control unit of the DUT can handle the typical flow of generating the 32-bit output word, informing the superior system and being restarted. On the other hand, if all the 128 output bits match the expectation based on the input file, the confidence that the DUT executes the

computation of the output data as described by the corresponding algorithm.

Table 4.3 summarizes the two simulations used for logical verification.

Table 4.3: Summary of simulations for logical verification

	Input	Output	Purpose
Short simulation	8 bits	2 bits	Detailed evaluation of algorithmic behavior using waveforms
Long simulation	Varies	128 bits	Verification of control flow; evaluation of algorithmic behavior based on large data set

4.2.2 Implementation of the von Neumann corrector

In order to establish an implementation of a VNC, it is meaningful to first derive a abstract description of its functionality. The VNC can be described as shown in Algorithm 1 in Section 2.4.1. However, Algorithm 1 is based on an input stream, denoted as the vector $\overrightarrow{x}$, with a fixed length, n. The length, m, of the output, $\overrightarrow{y}$, is as a result undetermined and varies with the applied input stream. This is rather unpractical, as most real applications use a determined number of requested bits, and m is as such fixed. It makes therefore sense to reformulate Algorithm 1 in order to work with a fixed m and varying n. This is presented in Algorithm 4.

Algorithm 4 The von Neumann algorithm with fixed output length

Input: $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$
Initialize: Empty output vector $\overrightarrow{y} = \langle\rangle$
bit counter $i = 0$
repeat
  if x_{2i} **xor** x_{2i+1} **then**
    Insert x_{2i} at the end of $\overrightarrow{y}$
  end if
  Increase i by one
until Length of $\overrightarrow{y}$ equals requested m

Instead of executing an XOR-operation on all successive bit pairs in $\overrightarrow{x}$, Algorithm 4 executes as many XOR-operations as needed to create an output of length m. As a result, the input stream must be of sufficient length, n. Equation 4.1 describes an expectation of n for a fixed m. However, for the purpose of implementation, this aspect is negligible and it is simply assumed that $\overrightarrow{x}$ is of infinite length. This assumption is justifiable, since the in Section 3.1 discussed entropy source can create an unlimited amount of data.

Based on Algorithm 4 and with regards to the assumed system environment, depicted in Figure 4.8, a simple flow diagram for a VNC module can be defined as described in Figure 4.10. During the beginning of the operation the module is in its

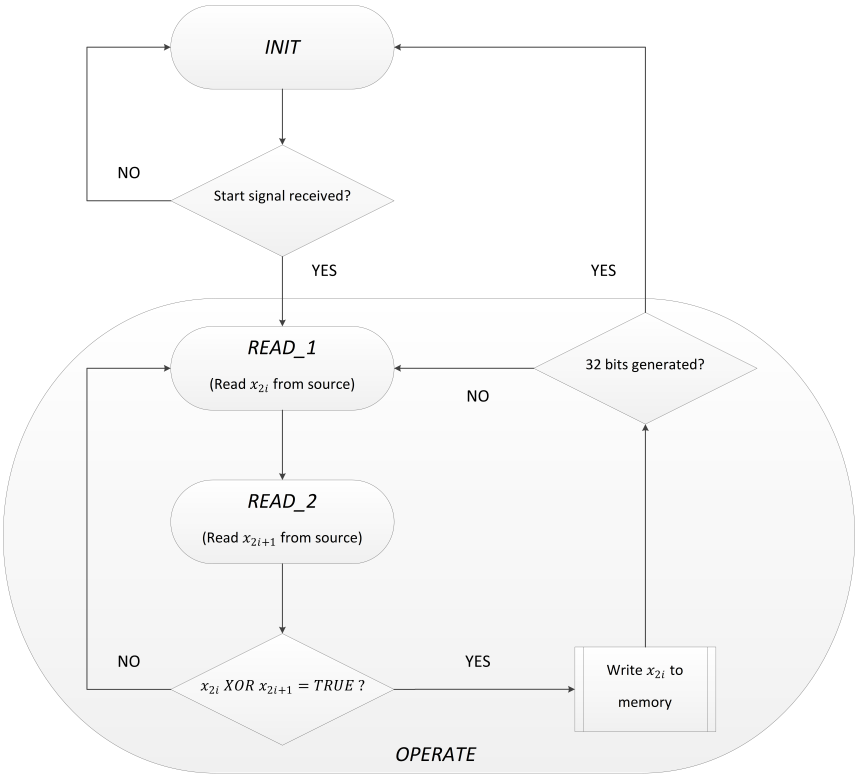


Figure 4.10: Flow diagram of a VNC module

initial state, called *IDLE*. The module is kept in this state, until the superior system activates the module, by setting the start signal. If the start signal is received, the module changes to a operational state, *OPERATE*, which executes Algorithm 4. The module retains in this state, until an output of 32 bits has been produced.

In *OPERATE*, the VNC-module constantly reads bits from the entropy source. Following the notation of Algorithm 4, the first bit of a successive bit pair is referred to as x_{2i}, and x_{2i+1} denotes the second bit of the pair. To keep track of which bit has been read from the source, *OPERATE* has two sub-states, called *READ_1* and *READ_2*. After x_{2i+1} has been read during *READ_2*, Algorithm 4 is executed by performing an XOR-operation with x_{2i} and x_{2i+1}. If the result of the XOR-operation is logic one, x_{2i} is written to an output memory block of the VNC module. It is important to note, that this writing process is executed on a transition between two sub-states and is not considered to be an autonomous sub-state. If the XOR-operation results in a logic zero, x_{2i} is discarded. In both cases, the module returns to sub-state *READ_1*. However, if writing x_{2i} yields the module to reach the requested amount of 32 bits, it returns from *OPERATE* to *IDLE*.

Considering the in Figure 4.10 presented flow-diagram, it makes sense to base the implementation of the VNC-module on three sub-modules. This approach is depicted in Figure 4.11, which uses the input and output signals as defined in Table 4.2. The control sub-module, *vnc_ctrl*, determines in which of the main states, *IDLE* or *OPERATE*, the VNC is. While in *OPERATE*, the sub-module activates the other sub-modules and requests data from the entropy source. In addition, it evaluates if

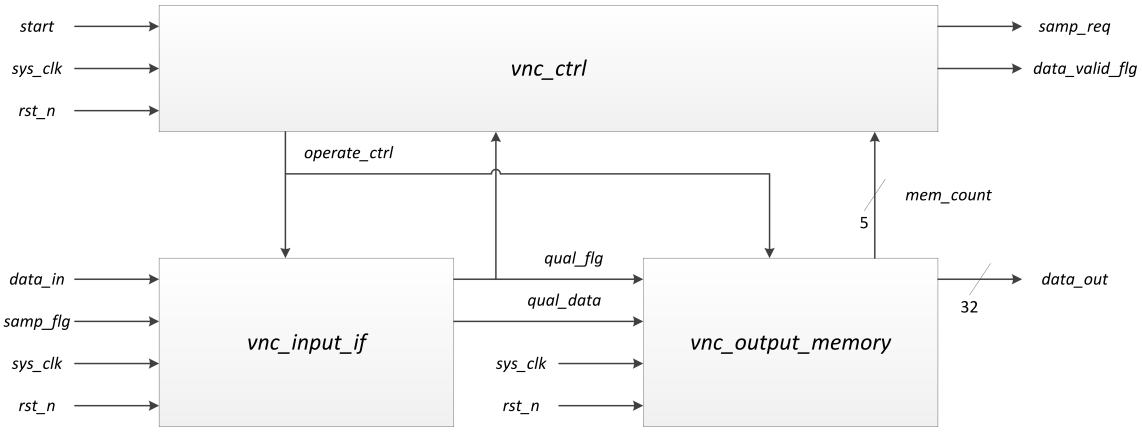


Figure 4.11: Block schematic of a VNC module

enough data has been gathered and communicates with the superior system.

While *vnc_ctrl* handles the main state control of the VNC module, most of the in *OPERATE* required operations are performed in the input interface of the module, *vnc_input_if*. This sub-module has a single state register and is either in *READ_1* and *READ_2*. In addition, it contains a single data register to store the first bit of a bit-pair, x_{2i}. The purpose of the sub-module is to execute the in Algorithm 4 described procedure. In the case that the in the algorithm included XOR-operation results in a logic 1, *vnc_input_if* qualifies the current bit (x_{2i}) and informs the other sub-modules about this decision.

If the bit x_{2i} is qualified by *vnc_input_if*, it is stored in the output memory of the VNC module, *vnc_output_memory*. This sub-module reads the bit from *vnc_input_if* and stores it in a 32-bit output register bank. The register bank is directly connected to the 32-bit output of the VNC, *data_out*. Besides of the output register bank, *vnc_output_memory* contains a 5-bit counter, which is increased each time a new bit is stored. The purpose of this counter is twofold. First, as the count equals the number of qualified input bits, it is used internal in *vnc_output_memory* to derive the address of the register in the output register bank to which the next qualified bit should be written to. Second, the counter is passed on to *vnc_ctrl*, which uses it to keep track of the amount of bits collected. This information can then be used to determine if *vnc_ctrl* should stay in *OPERATE* or if it should return to *IDLE* and signal to the superior system that the generated data can be read from *data_out*.

Before presenting the detailed implementations of the sub-modules, it is of interest to note that, in contrast to the IHF (see Section 2.4.2 and Section 4.2.3), the VNC as presented is not tunable by, for instance, adjusting the ratio of the number of required input bits per output bit or by the IHF tuning parameter β. Hence, the influence of changing a tunable parameter on the final implementation does not have to be analyzed. This enables the exploration of the effect of different design approaches on the energy performance of the implementation.

Considering the above given approach of a VNC implementation, it is reasonable to assume that the principle of clock gating, as presented in Section 2.5, should have a positive effect on the power consumption of the module. For instance, most registers in the module are not updated unless a new bit arrives from the entropy

source. Thus, considering Table 3.1, which states that the ADC based entropy source provides a new bit roughly every $50th$ clock cycle of the main clock (*sys_clk* of Figure 4.11), during most clock cycles the registers of the implementation are updated unnecessarily. It is worth noticing that this effect is further increased for some registers. For instance, each single register of the output register bank in *vnc_output_memory* is only updated once during the generation of 32 bits. Hence, it is reasonable to assume that the introduction of clock gates in the design can decrease the dissipated dynamic power, as defined by Equation 2.47, drastically.

However, each clock gate comes at the cost of additional hardware that potentially increases the power consumption of the module. It is therefore of interest to explore whether adding clock gates to the design improves the power efficiency of the module or if it influences the performance negatively. In order to do so, four different approaches to implement the sub-modules of the VNC are introduced. The first one makes exhaustive use of the principle of clock gating, by applying a clock gate to every register that can have an individual clock signal. This should reduce the switching activity, γ, of the registers to a level close to the minimum. The second approach uses clock gating only if at least four registers can be connected to one clock gate[1]. In the third approach, no clock gate is explicitly included during the design phase, but the synthesis tool is enabled to add clock gates as it sees fit. Finally, the fourth approach avoids the use of clock gates altogether. It should be noted that, for the following presentation of the design of the sub-modules, the third and the fourth approach are equivalent, since the clock gates of the third approach are not included before the synthesis process. An overview over all four approaches is presented in Table 4.4. The SystemVerilog realization of the in *Approach 1* and *Approach 2* used clock gates is presented in Appendix D.1

Table 4.4: Overview over the different VNC design approaches

Approach 1	Exhaustive use of clock gates
Approach 2	Clock gates used four or more registers
Approach 3	Clock gates inserted by synthesis tool
Approach 4	No use of clock gates

Figure 4.12 shows the implementation of *vnc_ctrl* for an approach that clock gates each single register and an approach that uses no explicit defined clock gates. Since the sub-module contains only two registers, this are the only two approaches considered for this module. The two registers are the status register, which is either in *IDLE* or *OPERATE*, and a register that contains the current value of *data_valid_flg*.

The main difference between the two in Figure 4.12 depicted approaches is the existence of a clock gate that controls the clocks applied to the two registers. During *IDLE*, *sys_clk* passes the gate unopposed. This is necessary, because the status register must be able to handle the *start* signal, which is unpredictably applied by the superior system. However, during *OPERATE*, the clock gate blocks *sys_clk*,

[1]This approach is based on a design "rule of thumb" of *Silicon Laboratories*, which depicts a trade-off between the reduced switching activity and the increased overhead which accompany the introduction of clock gates.

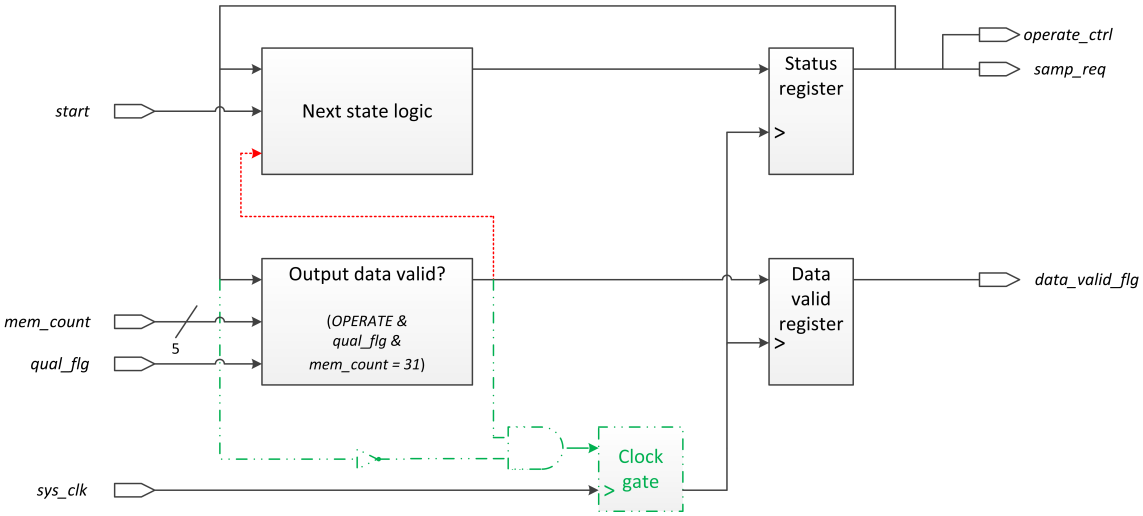


Figure 4.12: Block schematic of *vnc_ctrl*: The green illustrated version depicts a clock gated solution, used in *Approach 1*. Red depicts an approach without clock gates used in *Approach 2*, *Approach 3* and *Approach 4*

unless all 32 requested bits have been generated. This means that neither register is updated during the entire duration of *OPERATE* aside from the last clock cycle of the state. It is worth noticing that this makes it possible to simplify the next state logic of the state register. Since the only positive clock edge that is applied to the register in *OPERATE* indicates that *OPERATE* is completed, it is sufficient to design the next state logic such that the status register returns from *OPERATE* immediately to *IDLE*. The gated clock, however, prevents that this state transition is executed before all 32 output bits are generated. In contrast, when clock gating is not applied, more information has to be used to in the next state logic in order to evaluate if *OPERATE* should be left. This is depicted in Figure 4.12 by an additional input to the next state logic.

The implementation of *vnc_input_if* is presented in Figure 4.13. As for *vnc_ctrl*, only two registers are used in the sub-module and therefore only a totally clock gated approach and a approach with no explicit clock gates is discussed. The registers are respectively used to store the state of the sub-module (*READ_1* and *READ_2*) and the first bit of a bit pair (x_{2i}).

Compared to Figure 4.12, the main difference of the clock gated approach of *vnc_input_if* is that two nested clock gates are used. The first one supplies the status register which changes its state each time a new source bit is received. The second one is applied to the register that contains x_{2i}, which only has to be updated for every second input bit. If an approach without clock gates is used, registers have to be used for which the load operation can be controlled by an enable signal. As illustrated in Section 2.5, the ability to control the register in this way comes at the cost of some minor logical overhead, which is not explicitly depicted in Figure 4.13. Appendix D.1 presents SystemVerilog codes of memory elements both with and without an enable signal. The control signals used to enable the registers are the same as used to control the corresponding clock gate in the alternative approach.

In contrast to *vnc_ctrl* and *vnc_input_if*, the output memory sub-module *vnc-*

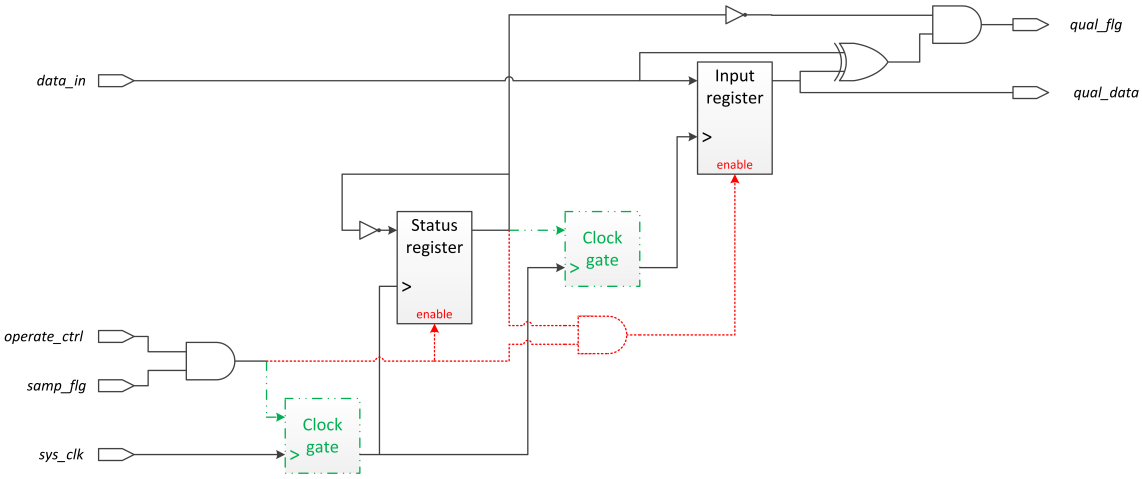
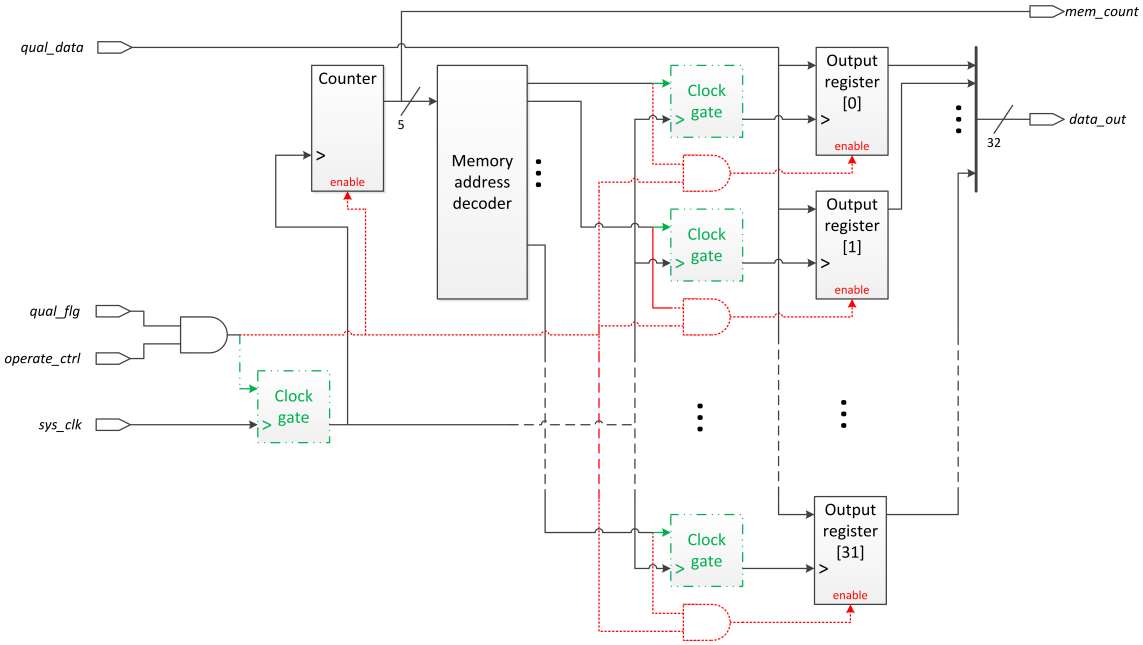


Figure 4.13: Block schematic of *vnc_input_if*: The green illustrated version depicts a clock gated solution, used in *Approach 1*. Red depicts an approach without clock gates used in *Approach 2*, *Approach 3* and *Approach 4*

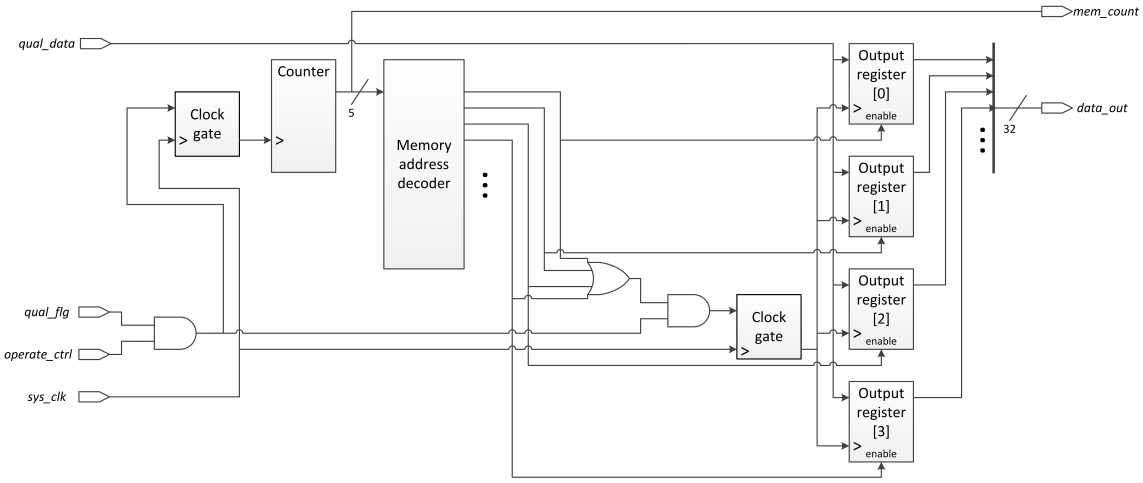
_output_memory contains more than two registers. These registers are used to realize the output bit counter and the output memory bank, which is connected to *data_out*. This means that a total of 37 registers is embedded in *vnc_output_memory*. As a result, three different approaches are presented. Figure 4.14a shows both an approach which uses exhaustive clock gating and an approach without any explicit defined clock gates. In Figure 4.14b, a third approach is depicted which uses clock gates to supply at least four registers per clock gate.

Before focusing on the difference between the three approaches, it should be noted that they all use the same basic concept. The data received from *vnc_input_if* is directly connected to each register of the output register bank. Thus, in order to only load the input bit to the appropriated register, some kind of control signal is required. This is provided by the counter of the sub-module, which keeps track of how many bits have been stored already. The resulting 5-bit count is then passed through a combinatorial address decoder, which converts it into 32 control signals, one for each register. As only one register is activated at the time, the 32-bit output of the address decoder can be thought of as the *one-hot* representation of the output counter. Since the counter, as such, controls the sub-module, it can be, from an analytic point of view, considered to be the status register of *vnc_output_memory*.

Based on this, the approaches depicted in Figure 4.14a are quite similar to the approaches used to implement *vnc_input_if*, shown in Figure 4.13. For the clock gate version, a main clock gate is used to suppress *sys_clk* unless the sub-module is activated and a new bit has to be stored. The gate clock is applied to the counter, which thus only increases its count on the arrival of a new bit. In addition, each register of the memory bank has its own clock gate, which blocks the applied clock, unless it is activated by the respective control signal from the address decoder. In this way, each register of the memory bank receives only a single positive clock edge during a normal generation procedure of 32 output bits. If no clock gates are used, enable signals must be added to the both the registers of the memory bank and the counter. This is depicted in Figure 4.14a and, again, quite similar to Figure 4.13.



(a) Block schematic of *vnc_output_memory*: Green depicts a exhaustive clock gated solution used in *Approach 1*. The red version makes no use of explicit defined clock gates and is used in *Approach 3* and *Approach 4*



(b) Block schematic of *vnc_output_memory* using one clock gate for at least for registers. This solution is used in *Approach2*. Note that only the first four registers of the output register bank are depicted.

Figure 4.14: Block schematics of *vnc_output_memory*

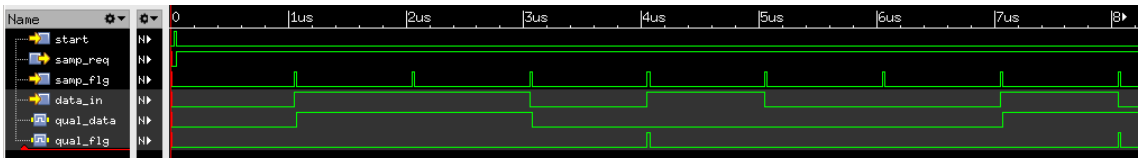


Figure 4.15: Waveform of the first VNC verification simulation

Figure 4.14b shows an implementation of *vnc_output_memory*, which uses clock gates to supply groups of at least four registers. As the counter contains five registers, one clock gate is applied to it. This clock gate is essentially equivalent to the main clock gate, of the clock gated approach of Figure 4.14a. However, for the output memory bank, the approach differs from the above presented solution. Figure 4.14b shows how four registers are grouped and applied to the same clock gate. As a result, the clock gate is activated, which means that the clock is not suppressed, whenever one of the four corresponding control signals delivered by the address decoder is high. It is important to note, that this makes it necessary to enable the registers explicitly, as otherwise all four registers would load the delivered data simultaneously.

At this point, multiple approaches of an implementation of a VNC have been presented. Before turning the focus on the logical verification of these implementations, it is worth to note a tendency that is similar for the three presented sub-modules. Considering Figure 4.12, Figure 4.13 and Figure 4.14a, it is depicted, that slightly less control logic, in form of additional gates and registers with enable signals, is needed if clock gating is applied. Thus, while the clock gates themselves depict an overhead in hardware, the design itself can be slightly simplified. As such, the relative overhead between a solution using exhaustive clock gating and an approach without clock gates, is reduced. However, by comparing Figure 4.14a and Figure 4.14b, it is evident that this not the case if clock gating is only applied partly.

As presentd in Table 4.3, the verification of the presented VNC implementations is first based on a short simulation which results in only a two output bits. In this way, it can be checked whether or not the VNC-algorithm is executed as described in Algorithm 1, by analyzing the executed process in detail. Since such a simulation thus mainly considers the execution of the algorithm, it focuses mainly on the *vnc_input_if* sub-module, which performs Algorithm 1. The other sub-modules are considered in more detail in a succeeding simulation.

A simple example of a 8-bit input of the VNC and the corresponding output has been given in Section 2.4.1. It is restated here for convenience: Applying Algorithm 1 on the 8-bit input $\overrightarrow{x} = \langle 1, 1, 0, 1, 0, 0, 1, 0 \rangle$ yields the output $\overrightarrow{y} = \langle 0, 1 \rangle$. It should be noted that, for the given example, $\overrightarrow{x}$ contains all possible combinations of the bit pair $\langle x_{2i}, x_{2i+1} \rangle$. It is therefore suitable to use the given example as a first, short simulation to verify the behavior of the VNC implementation. Figure 4.15 shows the resulting, correct waveform for an implementation that uses no clock gating. The corresponding waveforms for the other implementations are essentially equivalent to the here presented results, and thus verify the functionality of the proposed designs.

In order to further increase the confidence in the functionality of the implementation and to widen the scope of the simulation to include also *vnc_ctrl* and *vnc_output_memory* in a direct manner, a second, larger simulation is executed.

During this simulation a longer sequence of known input bits is applied to the VNC and the simulation routine is running the VNC 4 times in order to produce a total of 128 bits.

The results of this simulation are shown in Figure 4.16, for the VNC implementation without clock gates. Corresponding results have been found for the other implementations. As before, all results are correct and essentially equivalent.

After the functionality of the proposed VNC implementations has been verified, each proposal has to be synthesized. As stated in Section 4.2.1, a presentation of this step is omitted here. The basic results are presented in Appendix F, including the power estimations of the implementations, which are further discussed in Chapter 6.

However, since the difference between *Approach 3* and *Approach 4* (see Table 4.4) is highly dependent on the synthesis process, it should be mentioned that the synthesis tool inserts one single clock gate into *Approach 3*. The clock gate controls the clock connected to the counter of *vnc_output_memory* and is as such similar to the clock gate on the left hand side of the block schematics presented in Figure 4.14.

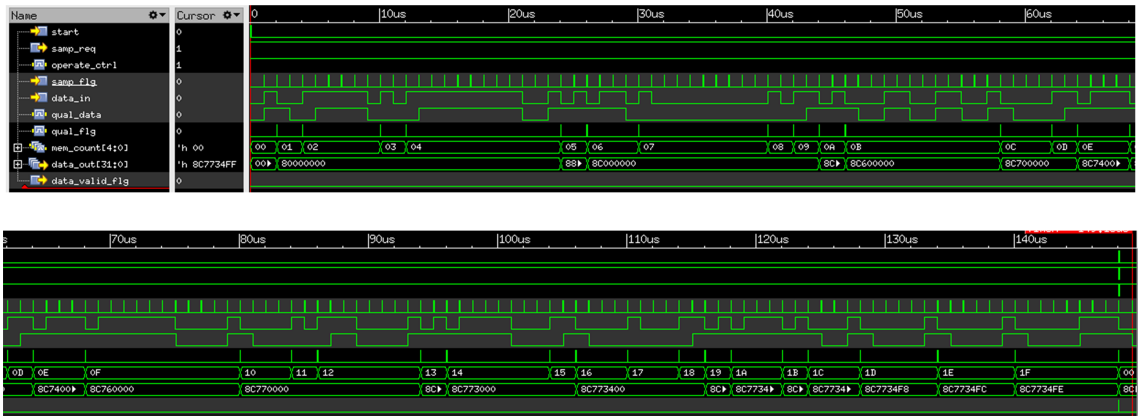
4.2.3 Implementation of an extractor based on pairwise independent hash functions

Before discussing a possible implementation of the IHF, it is worth to consider an aspect that differs from the implementation procedure of the VNC. While the VNC operates on single bit pairs and adds one and one bit to the output until the output reaches the desired length of 32 bits, the IHF can be considered to operate on n-bit input blocks in order to generate an m-bit output. It should be stressed that these are parameters determined by the IHF and not by the chosen system architecture (Figure 4.8). To make this explicit, the number of input and output bits for the IHF are for the rest of this project denoted as n_{IHF} and m_{IHF}, respectively.

For the implementation of the IHF, it is desirable to leave the parameters m_{IHF} and n_{IHF} undetermined. In the case of n_{IHF} this is mainly due to the reason, that the amount of input bits required depends on the amount of entropy per bit, which is determined by the source. Thus, in order to keep the implementation as general as possible, n_{IHF} is considered a variable that can be tuned with respect to the used source.

For the output length m_{IHF} it would be possible to simply fix $m_{\text{IHF}} = 32$. However, the complexity of Algorithm 3 increases drastically with increasing m_{IHF}, as, for instance, can be seen from the discussion presented with regards to Figure 4.5 in Section 4.1.2. It has been stated in Section 2.4.2, that the outputs of multiple IHF-operations can be concatenated to one output, as long as the (conditional) min-entropy requirements of the extractor are satisfied by the input of each operation. Thus, it might be possible to reduce the total number of operations by executing an IHF-algorithm multiple times, for example 8 times for $m_{\text{IHF}} = 4$, and then concatenate the results to one 32-bit output. In order to be able to test this method and analyze its effect on the energy performance of the design, m_{IHF} is kept undetermined in the same manner as n_{IHF} for the rest of this implementation.

A flow diagram of a possible IHF module based on Algorithm 3 is presented in



(a) Waveform for the first generated output word

(b) Waveform for all four generated output words

Figure 4.16: Waveform of the second VNC verification simulation

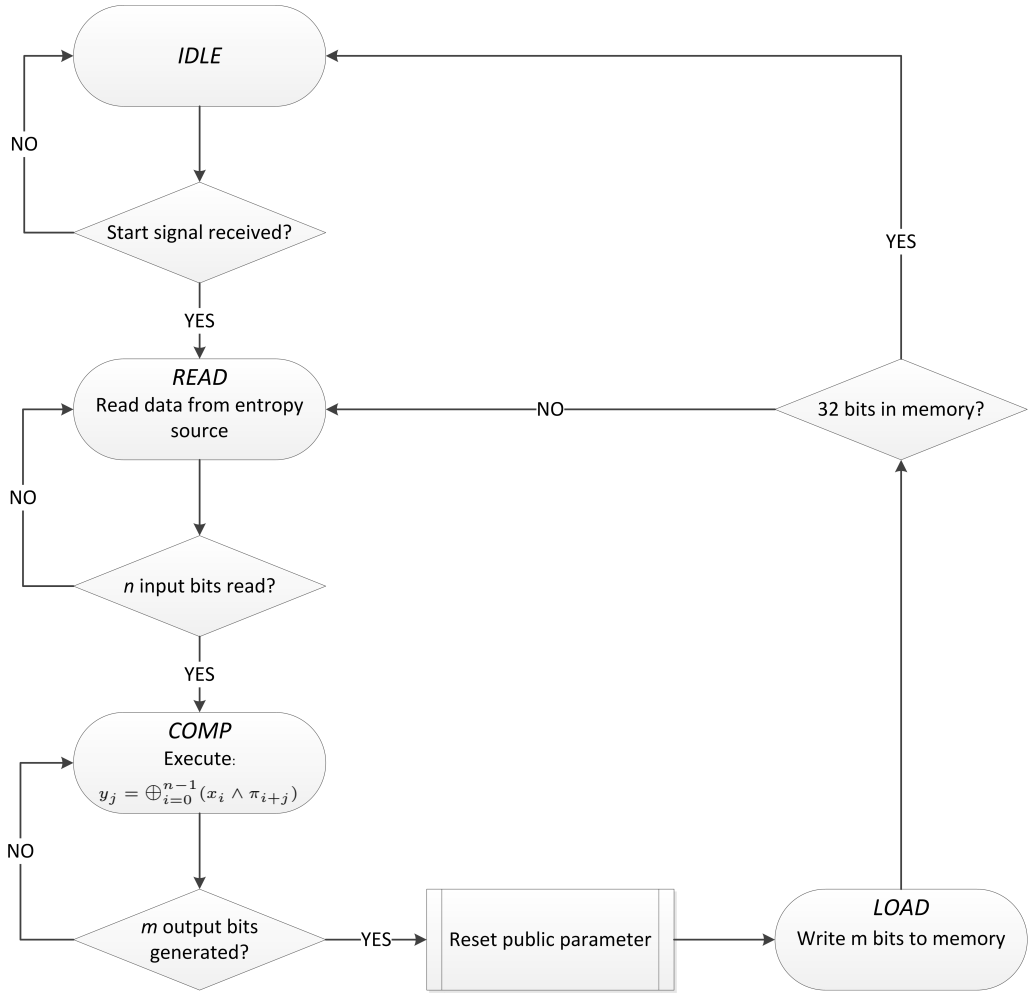


Figure 4.17: Flow diagram of a IHF module

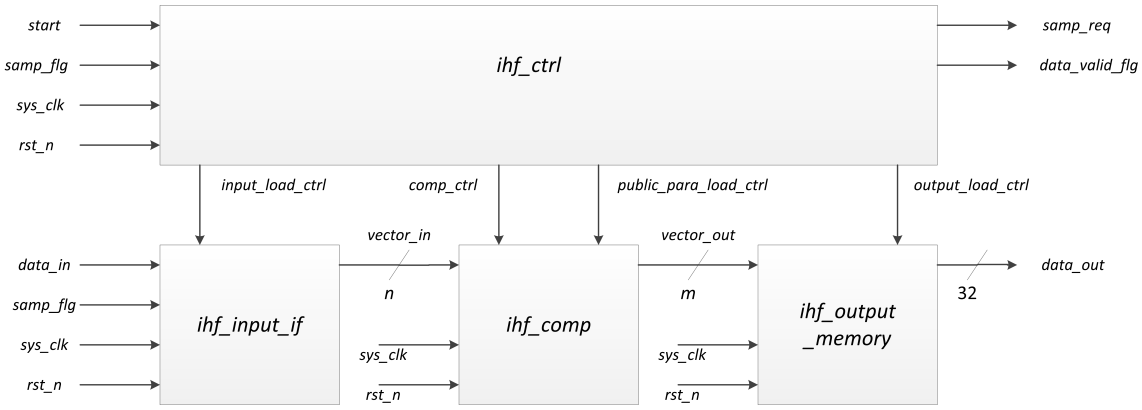


Figure 4.18: Block schematic of a IHF module

Figure 4.17. As the VNC module (Figure 4.10), the IHF module is initial in the *IDLE* state, where it remains until it is activated by a start signal. Once the module is started it switches to the state *READ*. In this state, the module requests input bits from the entropy source and stores them into a input memory. The module remains in this state until the entire n_{IHF}-bit input required by Algorithm 3 is received. When this is the case, the module begins to compute the outcome of the IHF, by changing to the *COMP* state. While in *COMP*, the module executes Algorithm 3 by repeatedly performing Equation 2.44. It is worth noticing, that this means that the module generates one output bit per time interval which it spends in *COMP*. After having generated the m_{IHF} output bits of the algorithm in this manner, the module changes to a state referred to as *LOAD*. During this state transition, the public parameter, $\overrightarrow{\pi}$, of Algorithm 3 is reset. Why this is necessary is related to the method *COMP* uses to execute Equation 2.44 and will become clear when the implementation is presented in more detail, below in this section. In *LOAD*, the module stores the m_{IHF} output bits in an output memory block and evaluates how many bits already have been gathered in this way. If the number of bits stored in the output memory does not equal the requested 32 output bits, the module returns to *READ*, in order to generate the remaining bits. However, once the 32 output bits have been generated, the module returns from *LOAD* to *IDLE*, where it remains until it is reactivated.

Based on the flow diagram of Figure 4.17, it is appropriated to divide an IHF implementation into four sub-modules, quite similar to the approach used to implement the VNC (see Figure 4.10). This is depicted[2] in Figure 4.18. It might be helpful to recall that the input and output signals are defined in Table 4.2. As for the VNC, a control sub-module, *ihf_ctrl*, is used as a communication interface to the superior system and to control the other sub-modules of the implementation. The sub-module can therefore be considered to be a state machine used to realize the in Figure 4.17 depicted flow diagram.

Once activated by the *start* signal, *ihf_ctrl* activates *ihf_input_if* and starts requesting bits from the entropy source. The *ihf_input_if* sub-module is basically a simple input register, used to store the n_{IHF} required input bits. When all n_{IHF} input bits have been gathered, *ihf_ctrl* activates *ihf_comp*, which runs Algorithm 3 on the

[2]For convenience, the subscript "IHF" is not used in the in this section presented figures.

n_{IHF} bits stored in *ihf_input_if*. This computation results in the m_{IHF}-bit output *vector_out*, which is stored in *ihf_output_memory*. As illustrated in Figure 4.17, *ihf_ctrl* repeats this process until a total of 32 bits has been loaded into *ihf_output_memory*.

It has been discussed during the beginning of this section, that the parameters n_{IHF} and m_{IHF} of the presented IHF implementation are left generic, in order to be able to analyze the effect of different parameter choices on the design. To keep the focus of the analysis on the effect of the parameter choices, exploring the impact of design techniques on the power performance of the design, as it has been done with clock gating for the VNC implementation (see Section 4.2.2), is omitted here. As a consequence, only a single implementation approach is presented here. Furthermore, since no low level design techniques are applied, it is not necessary to consider the sub-modules of the IHF solution in the same amount of detail as it has been done in Section 4.2.2. This is due to the reason, that changing the parameters m_{IHF} and n_{IHF} in most cases just affects the number of registers used in the sub-modules, but not the logical structure.

The SystemVerilog implementation of *ihf_ctrl* is shown in Appendix D.3.3. The sub-module is basically a direct realization of a state machine with a behavior as specified in Figure 4.17. As such it contains some next-state logic, output logic and state-registers. In addition it makes use of two counter. The first counter counts, depending on the state, the number of bits that have been stored into *ihf_input_if* or how many of the m_{IHF} required bits have been generated by *ihf_comp*. Reusing the same counter for both purposes reduces the amount of hardware resources necessary to implement the sub-module. The second counter is used to count the number of bits already stored in *ihf_output_memory*. It is worth noticing that the only component in *ihf_ctrl* that is dependent on the parameter choice is the first counter, which has to count up to n_{IHF} and m_{IHF}, respectively. Further, as n_{IHF} always must be larger than m_{IHF} (see, for instance, Equation 4.4), the number of registers necessary to realize this counter is only dependent on n_{IHF}.

As mentioned above, the *ihf_input_if* sub-module is a simple input memory block, used to store the n_{IHF} bits that are required to execute Algorithm 3. To keep the implementation simple, a basic n_{IHF}-bit shift-register is used to realize the sub-module. This is depicted in Figure 4.19 and the related SystemVerilog implementation can be found in Appendix D.3.4. Obviously, the number of register elements in *ihf_input_if* is directly affected by the choice of the parameter n_{IHF}.

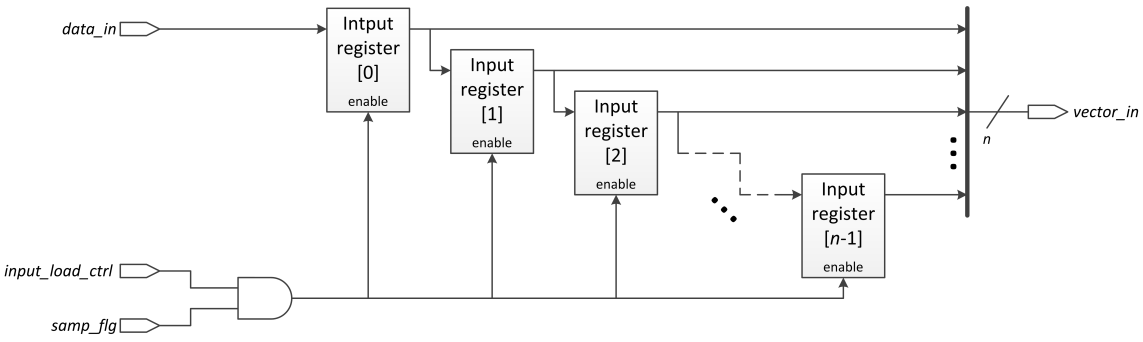


Figure 4.19: Block schematic of *ihf_input_if*

A block schematic of the used realization of *ihf_comp* is presented in Figure 4.20 (see Appendix D.3.5 for the SystemVerilog RTL-description.). As specified during the presentation of Figure 4.17, the sub-module generates one output bit per clock cycle, by executing Equation 2.44. This is done by first performing a bitwise AND-operation with the n_{IHF} input bits $(x_0, ..., x_{n_{\text{IHF}}-1})$ and the n_{IHF} first bits of the public parameter $(\pi_0, ..., \pi_{n_{\text{IHF}}-1})$ and then combining the results by applying them to an XOR network. The resulting output bit, y_j, can then be loaded into a shift output register. On the next positive clock edge, the content of the public parameter register is shifted towards the LSB and the procedure is repeated. It should be noted that using a shift register for the public parameter makes it necessary to reset the register before *ihf_comp* can be reused, as illustrated in Figure 4.17. It is easy to see from Figure 4.20 that both the size of the register and the complexity of the combinatorial logic depends on the choice of the parameters m_{IHF} and n_{IHF}.

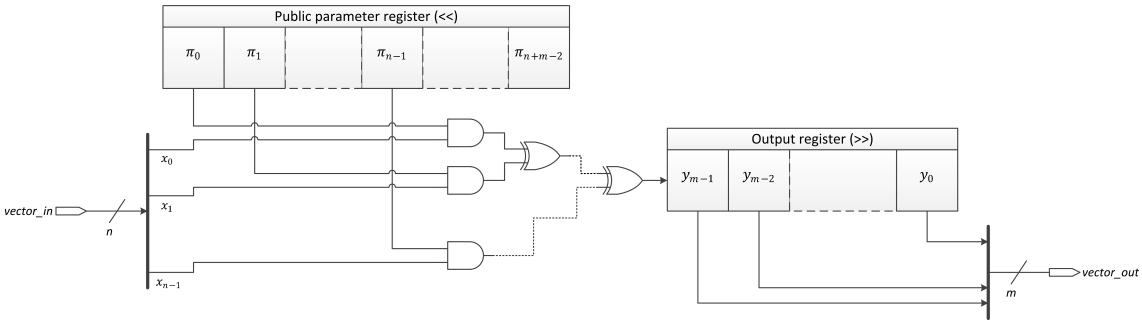


Figure 4.20: Block schematic of *ihf_comp*

The realization of *ihf_output_memory* is illustrated in Figure 4.21 and the corresponding SystemVerilog implementation can be found in Appendix D.3.6. The sub-module contains 32 register elements to store the generated output. The single register elements are arranged in m_{IHF} shift registers. This makes it possible to simultaneously load the entire m_{IHF}-bit input, *vector_out*, while using a shift register approach, similar to *ihf_input_if*. The number of shift registers and the number of register elements per shift register is of course dependent on the parameter m_{IHF}. However, it is important to note, that m_{IHF} only affects the structure of *ihf_output_memory*, but not the number of totally used register elements. As the effect of the different structures on the power performance of the sub-module should be reasonable small, it is reasonable to consider the power performance of *ihf_output_memory* to be independent of the choice of the parameter m_{IHF}.

Finally, before moving on to the verification of the design, some important aspects should be noted. First, while the number of registers and other logical gates in *ihf_comp* is directly dependent on m_{IHF}, the other sub-modules are only negligible or not at all affected by the choice of this parameter. Second, even though implemented in a different manner, the functionality of *ihf_input_if* and *ihf_output_memory* are quite similar to the output memory of the VNC, *vnc_output_memory*. As such, the memory sub-modules of the IHF could be realized in the same manner as for the VNC (see Figure 4.14). It should be noted that this would enable for all four clock gating approaches, which are discussed in Section 4.2.2. Third, the combinatorial elements of Figure 4.21 are directly connected to the input registers of the IHF.

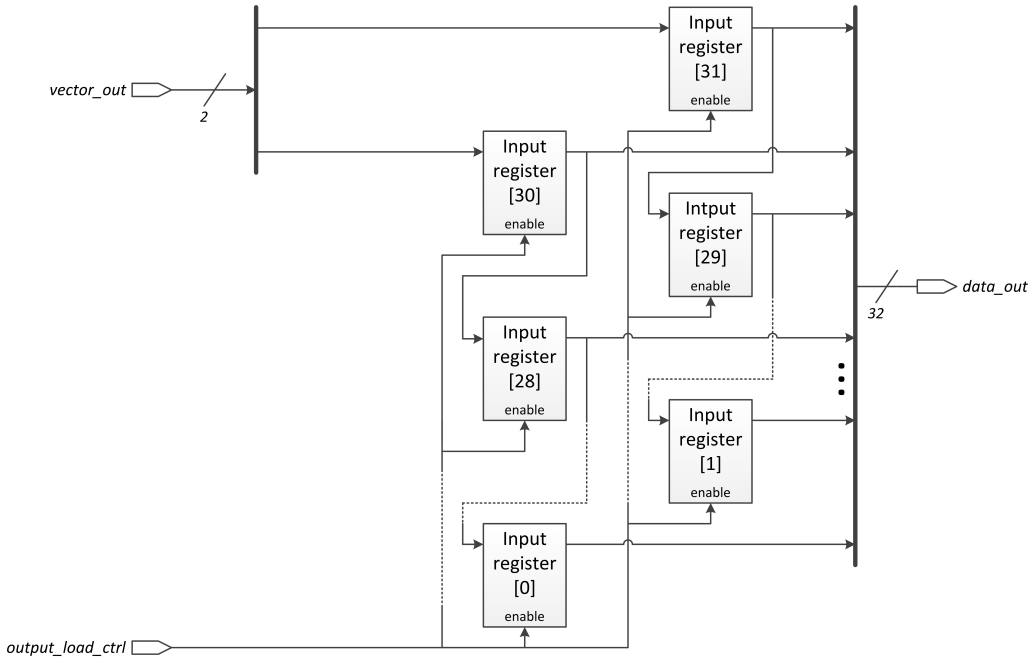


Figure 4.21: Block schematic of *ihf_output_memory*: *Example for* $m_{\mathrm{IHF}} = 2$

This yields to unnecessary switching in *ihf_comp* during *READ*. A simple method to avoid this is the in Section 2.5 mentioned guarded evaluation. However, as mentioned, such design techniques are not considered in this project and left for further work.

Having presented the implementation of the IHF module, the focus turns to verifying its correct logical behavior. Since the verification simulation has to be executed on a non-generic version of the IHF module, it is necessary to choose some explicit values for the parameters n_{IHF} and m_{IHF}. However, it is reasonable to assume that the choice of the parameters does not affect the logical behavior of the implementation, due to two reasons. First, for the sub-modules *ihf_ctrl*, *ihf_input_ihf* and *ihf_output_memory*, the parameters influence only the number or the structure of the used register elements. Second, the combinatorial logic in *ihf_comp* which is used to execute Algorithm 3 is extremely regular. As a result, the presented IHF implementation can easily be scaled according to the used parameters. It is therefore sufficient to verify the behavior of the implementation using a single set of parameter values.

Further, since the at this point chosen parameters are for verification purposes only, and not used for, for instance, the evaluation of the random character of the output, the parameters can be chosen freely as long as $n_{\mathrm{IHF}} > m_{\mathrm{IHF}}$. In order to simplify the simulation procedure, it is therefore suitable to choose rather small values. Table 2.6 in Section 2.4.2 presents a simple example of Algorithm 3, using an input length of $n_{\mathrm{IHF}} = 8$ and an output length of $m_{\mathrm{IHF}} = 2$. To keep the example and the simulation process consistent, it has been chosen to adopt these parameter values for the verification process.

Using the same argumentation as for the parameters m_{IHF} and n_{IHF}, the public

parameter $\overrightarrow{\pi} = \langle 0,1,0,0,0,1,0,1,1 \rangle$ has been chosen according to the example presented in Table 2.6.

For a first short simulation (see Table 4.3), that aims to verify the correct execution of Algorithm 3 in the *ihf_comp* sub-module, it is suitable to use the 8-bit input which has been used in the example of Table 2.6. Thus, the input $\overrightarrow{x} = \langle 1,1,0,1,0,0,1,1 \rangle$ is applied to the module. The expected output is $\overrightarrow{y} = \langle 0,1 \rangle$. Figure 4.22 shows the resulting correct waveform for the relevant signals in *ihf_comp* for the given stimuli. The generated output is $\langle 0,1 \rangle$ and thus equals the expected outcome. Note that only the time interval of the actual computation state, *COMP* is shown, since *ihf_comp* is deactivated during all other states.

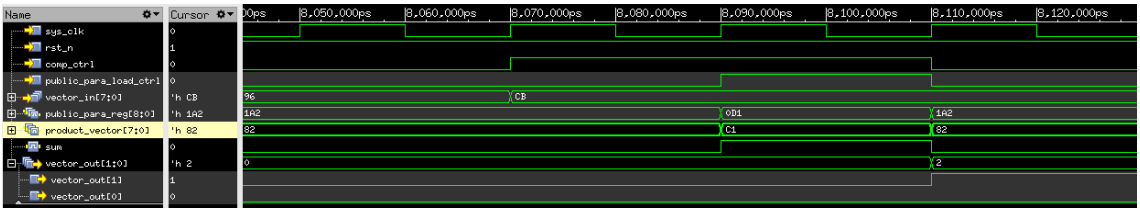


Figure 4.22: Waveform of the first IHF verification simulation

In order to include the other sub-modules into the scope of the simulation and to further increase the confidence in the correct execution of Algorithm 3 by the module, a second simulation with a larger input is performed.

With regards to Table 4.3 the used simulation triggers the IHF implementation to generate four 32-bit words. Thus, a total of 128 bits is created during this second verification simulation. Based on the used parameters $m_{\text{IHF}} = 2$ and $n_{\text{IHF}} = 8$, this means that the simulation requires an input of 512 bits.

The result of the simulation in form of a waveform for the most relevant signals of the IHF module is presented in Figure 4.23. It shows that the module behaves according to its specification.

Having verified that the presented IHF implementation behave in accordance to its specification, the module is synthesized. As for the VNC this is not presented in detail. In order to explore their effect on the implementation, the synthesis process has been performed with a number of different combinations of the parameters m_{IHF} and n_{IHF}. The basic results of the process are presented in Appendix F, including the power estimates which are discussed in more detail in Chapter 6.

(a) Waveform of the first generated output word

(b) Work flow of the post-processor implementation process

Figure 4.23: Waveform of all four generated output words

Chapter 5

Evaluation of Randomness

Having established functional implementations of both the VNC and the IHF in Section 4.2, it is of interest to evaluate if the implementations can be used to produce a random output when applied to the ADC based entropy source of Section 3.1. In order to do so, it is first necessary to generate a sufficient amount of output data for the two post-processors. As discussed with regards to Figure 4.7, this can easily be achieved by running RTL simulations of the post-processor designs, using input data that is generated by the ADC. The resulting output data can then be evaluated by using the NIST test suite (Section 2.2.2) and, in case of the VNC, the von Neumann condition test, presented in Section 3.2.

As for the analysis of the entropy source output in Section 3.3, the parameters for the NIST test suite and the von Neumann condition test have to be determined before the tests can be performed. The test parameters for the source analysis are presented in Table 3.3. In order to keep the different tests consistent, and recalling that the small number of test sequences, $N = 64$, has been chosen to cope with the compressing nature of the post-processing algorithms, it is reasonable to reuse these test parameters for the in this chapter presented analysis.

Section 5.1 evaluates the possibility of using the VNC as a post-processor for the ADC based entropy source. The results of using the IHF is presented in Section 5.2.

5.1 Analysis of the VNC Output

Before performing an analysis of the output of the VNC using data provided by the source implementation of Section 3.1, it is worth to recall the main results of the corresponding analysis of the source data. Section 3.3 shows that the source output is biased towards 1 and has a percentage of 0's of roughly 29%. The performed von Neumann condition test fails, in all likelihood due to dependencies in the data. Thus, it is rather unlikely that applying the ADC data to the VNC results in the desired random output. However, the here performed analysis is of interest with regards to two aspects. First, proving that the generated data in fact is not random increases the confidence in the proposed von Neumann condition test. Second, exploring how the VNC affects the input data potentially increases the understanding of the statistical characteristics of the entropy source.

Using the test parameters as stated in Table 3.3, a total of 64 Mbits has to be generated, by using the simulation setup presented in Section 4.2.1. It is worth to recall from Section 2.4.1, that the exact number of the input bits requested by the VNC depends on the statistical characteristics of the input. However, if the von Neumann conditions are satisfied, the expected number of input bits, n_{VNC}, is described by Equation 4.1. It states that n_{VNC} is a function of p, which is the probability of a bit being equal to 0. Even though, p is unknown, it has been stated that the average percentage of 0's in the source data is roughly 29%. It is reasonable to use this observation to derive an approximated expectation of n_{VNC}. Thus by setting $p = 0.29$, Equation 4.1 predicts that the VNC requires an average of 4.86 input bits from the ADC, in order to generate a single output bit.

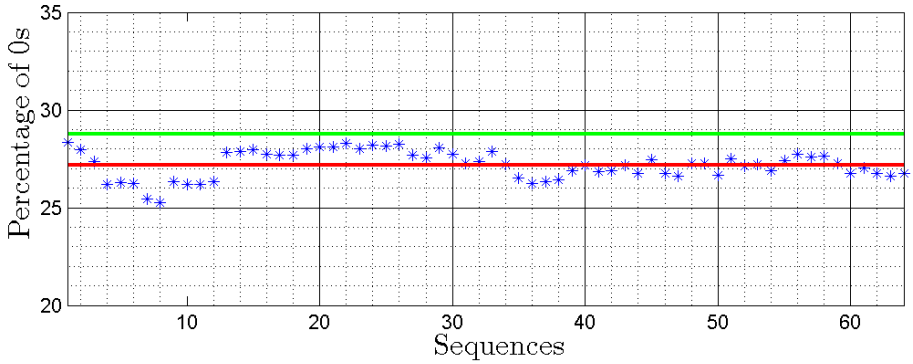
Running a simulation routine that triggers the VNC to generate a 64 Mbits output from the source data yields the VNC to request a total of 303,121,314 bits from the ADC. In other words, for the given circumstances, the VNC requires approximately 4.52 input bits per single output bit, which accords to roughly 93% of the expected 4.86 input bits.

Based on the generated output of the VNC, it is possible to run the data through the NIST test suite. Following the recommendation of NIST (see Section 2.2.2), the data is first only applied to the Frequency test. If the test is successful, the other test of the test suite are executed.

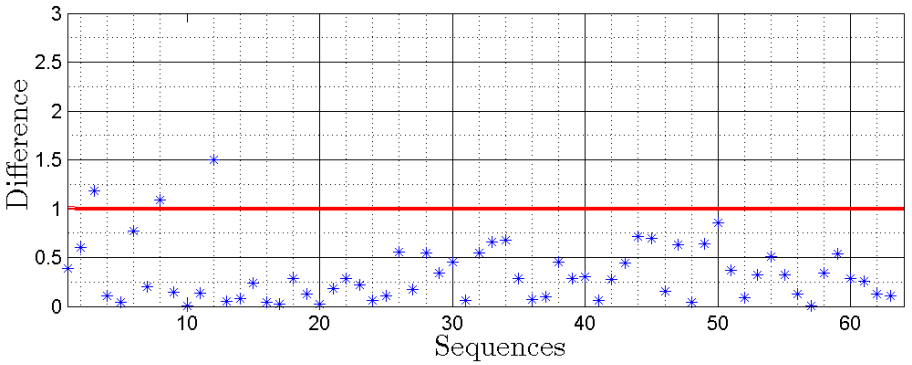
The results of the Frequency test for the 64 generated output sequences are presented in Table C.3 in Appendix C.2.2. It shows that the observed test statistic, s_o, which is defined in Equation 2.27, is larger than 450,000 for each of the performed tests. Recalling from the example given during the presentation of the Frequency test in Section 2.2.2, that the test fails for a sequence of $n = 1048576$ at a significance level of $\alpha = 0.01$ if the observed test statistic, s_o, either exceeds 2638 or is smaller than -2638, it is evident that the test fails for all 64 applied sequences. This is also implied by the fact, that the p-value of each test equals zero. As the result of the first-level tests is a clear rejection of the null-hypothesis, a second-level test is not performed.

In the same manner as done in Section 3.3, s_o can be used to derive the percentage of 0's in the used sequences. The results are presented in Table C.3 and in Figure 5.1a. The percentage varies between approximately 25.26% and 28.34% and has an average value of roughly 27.2%. Compared to the corresponding results for the source data (Figure 3.2a), which has an average percentage of 0's around 28.75%, this depicts a slight increase in the bias of the data. However, with regards to its input data, the output of the VNC has a reduced variation of the bias. This is further illustrated in Figure 5.1b, which shows that the percentage of 0's between two successive sequences only in three cases is larger than 1% and never exceeds the 2% mark. As a result, the bias of the generated output can be assumed to be rather constant, based on the same argumentation as of Section 3.3.

Even though the observed increase in the data is undesirable, it is worth noticing, that the observed average percentage of 27.2% is consistent with the analysis of the source data. During the performance of the von Neumann condition test for the output of the entropy source, it has been stated that on average 228,960 von Neumann pairs are observed (o_{vN}) per test sequence. The average number of obsered bit-pars



(a) Percentage of 0's in the sequences of the VNC output data: The red line depicts the average of approximately 27%. The green line depicts the average for the source data (29%).



(b) Absolute value of the difference in percentage of 0's in subsequent sequences: The red line depicts 1%.

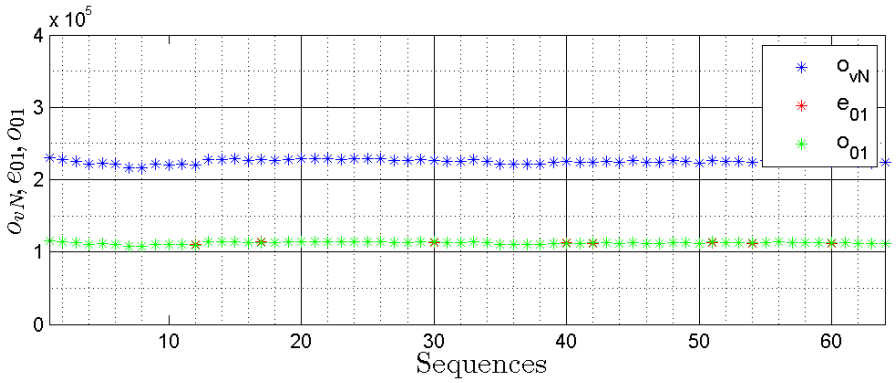
Figure 5.1: Results of the Frequency test for the VNC

of the type $\langle 0, 1\rangle$, o_{01}, is 64,143. In other words, the VNC produces on the average 64,143 bits that are equal to 0 per 228,960 generated output bits. This corresponds to an expected percentage of 0's of roughly 26.7%, which is approximately equal to the observed 27.2%.

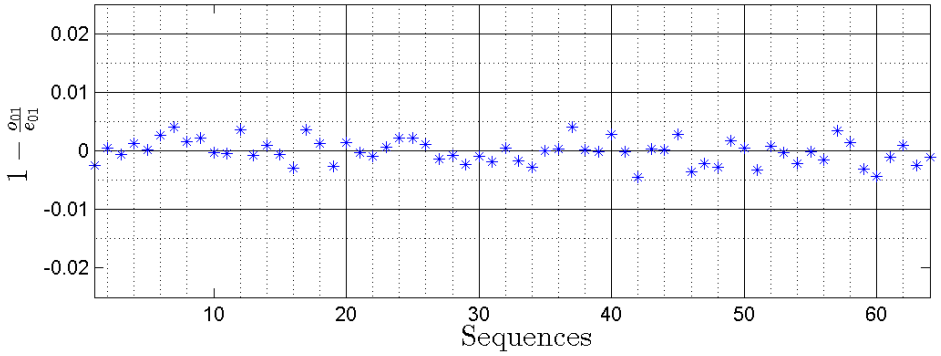
At this point of the analysis, it can be clearly stated that the Frequency test rejects the hypothesis that the generated output data of the VNC is random. It is therefore not necessary to perform any of the other tests of the NIST test suite. Again, it should be stressed, that this result is expected based on the in Section 3.3 performed von Neumann condition test. However, during the corresponding discussion of the results of the von Neumann condition test, it has been stated, that this incompatibility most likely is due to dependencies in the source data, which cause the observation of $\langle 0, 1\rangle$ to be less likely than the occurrence of $\langle 1, 0\rangle$. As the VNC compresses the input data by only selecting certain bits, it affects the structure and, as such, potentially the dependencies between bits. To explore this effect, it is therefore of interest to perform the von Neumann condition test also for the output of the VNC and compare the results to those of Section 3.3.

The results of the 64 performed first-level von Neumann condition tests are presented in Table C.4 in Appendix C.2.2. The table shows that the observed number

of von Neumann pairs varies between 215,855 and 229,770 and has an average of approximately 225,050. The corresponding observed test statistics, $s_o = o_{01}$ (see Equation 3.4), which is the frequency of the bit-pair $\langle 0, 1 \rangle$ in the respective sequence, ranges from 107,764 to 115,117, with an average of roughly 112,540. It is essential to note, that the observed test statistics of the single first-level tests are close to their expected values of $e_{01} = \frac{o_{vN}}{2}$, introduced in Equation 3.6. Figure 5.2a illustrates this and shows that the observed test statistics, o_{01}, are close to identical to their expected values. This can also be seen from Figure 5.2b, which depicts the normalized difference between o_{01} and e_{01}. In contrast to the corresponding results for the source data (see Figure 3.3b), which is in the range of 0.5, the normalized difference for the VNC output is close to zero for all tested sequences. This points towards the conclusion that the VNC output satisfies the von Neumann conditions, as defined in Equation 3.3.



(a) Frequencies of von Neumann pairs and the expected and the observed test statistics for the tested sequences



(b) Normalized difference between the observed and expected test statistics for the tested sequences

Figure 5.2: Results of the von Neumann condition test for the VNC

The hypothesis that the VNC output satisfies the von Neumann conditions is further strengthened by considering the p-values of the performed tests. Eventhough, the values vary vastly for different sequences, none of them is equal to 0. To unify the results of the 64 first-level tests to one concrete conclusion, it is reasonable to perform a second-level test, as discussed, for instance, in Section 2.2.2. With respect to Table 3.3, the p-values are therefore grouped into 10 equally size groups over the

interval [0,1]. This is illustrated in Figure 5.3. Given that the null-hypothesis of the test is true, which means that the von Neumann conditions are satisfied by the data, the depicted p-values are expected to follow an uniform distribution. To evaluate if this is the case of the observed data, the chi-squared goodness-of-fit test of Section 2.2.1 is performed. The resulting second-level p-value is approximately 0.99. Recalling from Table 3.3 that for the purpose of this report a second-level test is considered to be successful if the p-value exceeds $\alpha' = 0.0001$, it is obvious that the VNC output data passes the von Neumann condition test.

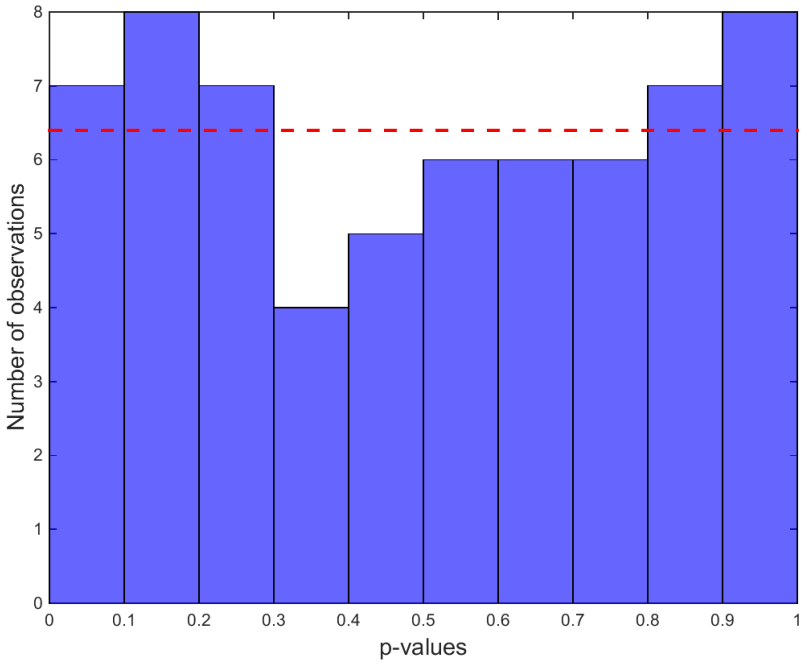


Figure 5.3: Observed p-values of the first-level von Neumann condition test for the VNC: *The red line marks a uniform distribution.*

Before moving on to discuss a suitable reaction to the positive result of the von Neumann condition test, it is worth to consider one final aspect of the observed data. With regards to Figure 5.2b, it has been stated that the magnitude of the difference between o_{01} and e_{01} is significantly reduced with respect to Figure 3.3b. In addition, it should be noted, that the normalized difference for the VNC output varies around 0. That means that, while for some sequences the observed test statistic exceeds its expected value, for other sequences o_{01} is smaller than e_{01}. In contrast, for the data generated by the ADC, the considered difference is continuously positive. It has been argued in Section 3.3 that this is an indicator for dependencies in the source data. As Figure 5.2b does not show the same systematic tendencies, it is reasonable to conclude that this specific kind of dependencies has been removed from the data.

Based on the result of the von Neumann condition test, it seems likely that the output data of the VNC satisfies the von Neumann conditions. It is therefore reasonable to investigate the possibility of generating a set of random data, by running the generated output data once more through the VNC, as illustrated in Figure 5.4.

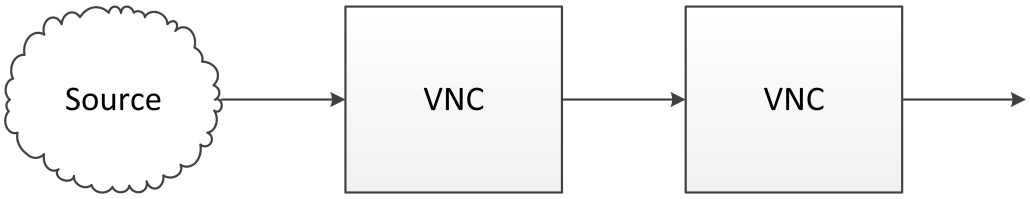


Figure 5.4: Illustration of the second VNC iteration approach

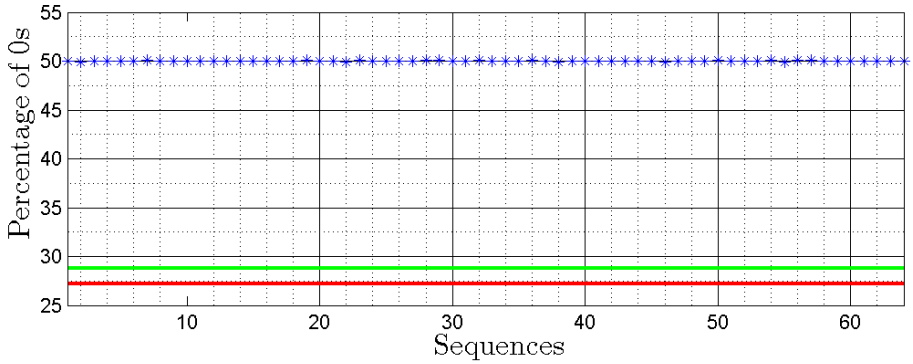
In order to investigate this *second iteration* approach, it is first necessary to create a sufficient amount of data during the *first iteration*, which then can be used by the second iteration to generate a 64 Mbits output. This output can then be applied to the NIST test suite, in the same way as done above for the first iteration.

Running a simulation routine that triggers the VNC to produce a 64 Mbits output, it is observed that the VNC uses 308,584,632 bits of the data, which has been produced by the first iteration. This corresponds to an average of roughly 4.6 input bits per single output bit. Recalling that the expected number of input bits, n_{VNC}, can be calculated by means of Equation 4.1, and using the average percentage of 0's in the input as an estimate of the bias p, n_{VNC} can be computed in the same manner as during the beginning of this section. The average percentage of 0's in the output of the first VNC iteration is approximately 27.2%. Hence by setting $p \approx 0.27$, the ratio between the number of input and output bits for the second iteration is roughly 5.07, which differs with slightly more than 9% from the observed value. Thus, the observation of the input-output-ratio of the second VNC-iteration matches the expected value decently. Since Equation 4.1 is only valid if von Neumann conditions are satisfied, this can be considered to strengthen the hypothesis that these conditions are satisfied for the input of the second iteration. However, it should be noted, that the expectation and the observation of the corresponding ratio for the input of the first VNC-iteration differ with only 7%, even though it has been shown that von Neumann conditions are not satisfied for this set of data.

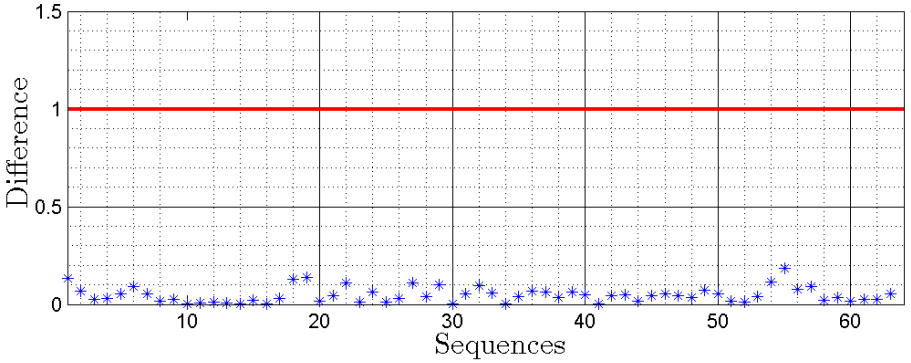
As before, the evaluation of the output of the second iteration by the NIST test suite is initiated by running the Frequency test. The results of the test are summarized in Table C.5 in Appendix C.2.2. The table shows that the observed test statistic, s_o (see Equation 2.27), varies between -2,872 and 2,250. It has been stated above and during the example given in Section 2.2.2, that the Frequency test does not fail for the given circumstances and at a level of significance of $\alpha = 0.01$, if $|s_o| \leq 2,638$. Thus, considering Table C.5 in detail, it can be seen that the Frequency test is successful for all 64 sequences, with the exception of sequence 56.

Figure 5.5a shows the percentage of 0's, which has been computed based on the corresponding s_o. The figure shows, that the bias of the data has been clearly reduced during the second iteration of the VNC. The average portion of 0's in the data is roughly 50.01% and differs thus only with 0.01% from a perfectly unbiased bit stream. In addition, the bias of the data can be considered to be nearly absolutely constant, varying between approximately 49.89% and 50.14%. This is further illustrated in Figure 5.5b, which shows that the bias difference between two successive sequences is below 0.5%, for all tested sequences.

The in Figure 5.5 presented data clearly supports the hypothesis that the fre-



(a) Percentage of 0's in the sequences of the output of the second VNC iteration. The red and the green line mark the average for the source data and the first VNC iteration, respectively.



(b) Absolute value of the difference in percentage of 0's in subsequent sequences: The red line depicts 1%.

Figure 5.5: Results of the Frequency test for the second VNC iteration

quencies of 0's and 1's in the output of the second VNC iteration are equal, which would mean that the Frequency test is successful. However, as stated, sequence 56 fails the test. In order to derive a clear result for the test, it is therefore reasonable to perform a second-level test, as previously performed during the von Neumann condition test for the first VNC-iteration.

Figure 5.6 shows the distribution of the 64 observed p-values of the first-level tests. To evaluate if the depicted situation could correspond to an uniform distribution, the chi-squared goodness-of-fit test is performed, resulting in a second-level p-value of approximately 0.83. As this value clearly exceeds the for this project determined second-level level of significance, $\alpha' = 0.0001$ (Table 3.3), the Frequency test is considered to be successful for the output of the second iteration of the VNC.

Based on this result, the other tests of the NIST test suite are performed, in order to evaluate whether the output of the second iteration can be considered to be random. The most important results of the performed tests are summarized in Table 5.1. It is worth to recall from Section 2.2.2, that it is common to only consider one result for tests that are performed several times for the 64 sequences. Therefore, only one result per test is presented in Table 5.1. Further, it should be noticed that some of the NIST tests require the specification of individual test

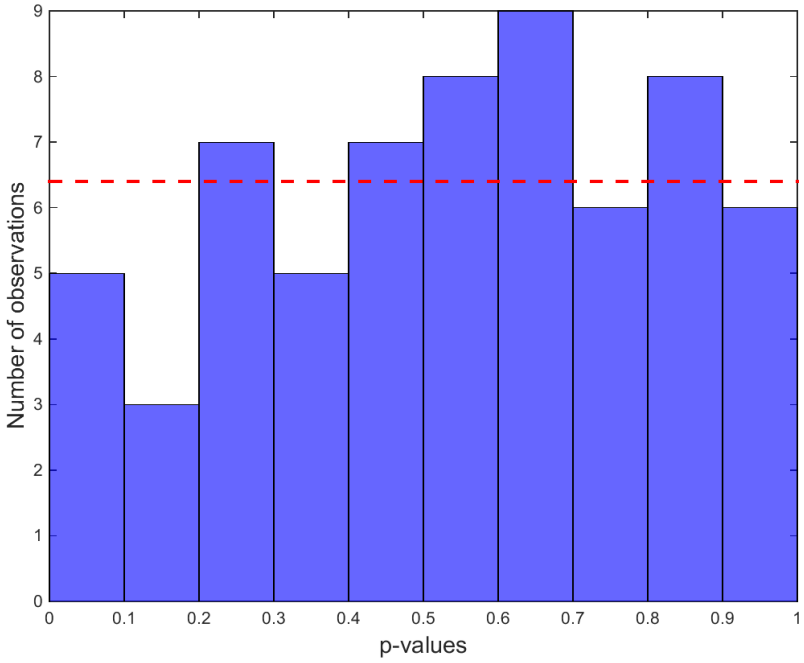


Figure 5.6: Observed p-values of the first-level Frequency tests for the second VNC iteration: *The red line marks a uniform distribution.*

parameters, which have not been discussed in this report, due to the time frame of the project. However, the here presented tests use mainly the in the NIST test suite set default values[1].

With respect to the in Table 5.1 presented results, 8 of the 15 performed tests fail. Hence, it is concluded that also the data generated by the second iteration of the VNC cannot be considered to be truly random. It is out of the scope to analyze the results of each of the performed tests. However, the results of the Block Frequency test and the Runs test should be explored in slightly more detail.

The Block Frequency test passes as clearly as the Frequency test. With regards to Table 2.3, both tests are quite similar. While the Frequency test compares the expected number of 0's in the sequence to the observed frequency, the Block Frequency test evaluates the same measure but over smaller bit-blocks of the sequence. Thus, the fact that the Block Frequency test is successful shows that the numbers of 0's and 1's are in the same range, even if shorter intervals are considered. This increases the confidence in the conclusion that the tested data is constantly unbiased, as stated during the discussion of the in Figure 5.5 presented results of the Frequency test.

In contrast to the two frequency related tests, the Runs test rejects the null-hypothesis for all 64 performed first-level tests, and thus clearly fails. This test focuses on the number of uninterrupted sub-sequences of identical bits (*runs*) in the data (see Table 2.3). In other words, the test determines whether the oscillation

[1]The interested reader may note that the block length of the Frequency Block test has been set to 16,384, following the NIST recommendation for the used n value

Table 5.1: Results of the NIST test suite for the output of the second VNC-iteration

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	63	0.834308	Passed
BlockFrequency	64	63	0.468595	Passed
CumulativeSums	64	63	0.407091	Passed
Runs	64	0	0.000000	Failed
LongestRun	64	32	0.000000	Failed
Rank	64	61	0.178278	Passed
DFT	64	58	0.000000	Failed
NOT	64	0	0.000000	Failed
OT	64	0	0.000000	Failed
Universal	64	25	0.000000	Failed
Approx.Entropy	64	0	0.000000	Failed
Rand.Excur.	37	35	0.009998	Passed
Rand.Excur.Var.	37	36	0.001084	Passed
Serial	64	0	0.000000	Failed
LinearComplexity	64	62	0.637119	Passed

between 0's and 1's in the data is either too slow or too fast compared to the expectations for a random bit stream. While a detailed presentation of the mathematical background and the explicit results of the test is omitted in this report, it is nevertheless possible to draw a simple but important conclusion. Since the test fails, the number of runs of identical bits must either be too small or too large.

One possible explanation of this observation could be that the data is biased. For example, the towards 1 biased 10-bit stream $\langle 1, 1, 1, 1, 1, 0, 1, 1, 1, 1 \rangle$ contains, due to its bias, only 3 runs, which is too low for a sequence of 10 bits. However, the here considered output of the second VNC-iteration passes both the Frequnecy test and the Block Frequency test. Thus, as stated above, the data is unbiased. As a result, it is reasonable to conclude that the for random data untypical number of runs is due to some kind of dependencies. For instance, if observing a bit that is equal to 1 increases the probability that the next bit is also equal to 1, it is likely that the data contains only a few long runs. In the same manner, a dependency that makes it likely that a bit is the logical inverse of the preceding bit yields a too high number of runs.

Based on this first, short analysis of the results of the NIST test suite, as presented in Table 5.1, the following can be concluded: Both the Frequency and the Block Frequency test show that by performing a second iteration of the VNC the bias of the data can successfully be removed. However, as the data still contains dependencies, it fails the NIST test suite and cannot be considered to be truly random.

To complete the analysis of the output of the second VNC-iteration, the von Neu-

mann condition test is performed for the data. The results are presented in Table C.6. Even though the average number of observed von Neumann pairs is slightly increases compared to the outcome of the first VNC iteration, the in Table C.6 presented data is fairly alike to the corresponding data in Table C.4. In order to keep the here presented discussion focused, a detailed analysis of the test results is therefore omitted at this point. However, a graphical illustration of the most relevant data is presented in Figure C.1 in Appendix C.2.2. It is worth noticing, that the figures show the same main characteristics as the corresponding graphs for the first VNC iteration, presented in Figure 5.2a.

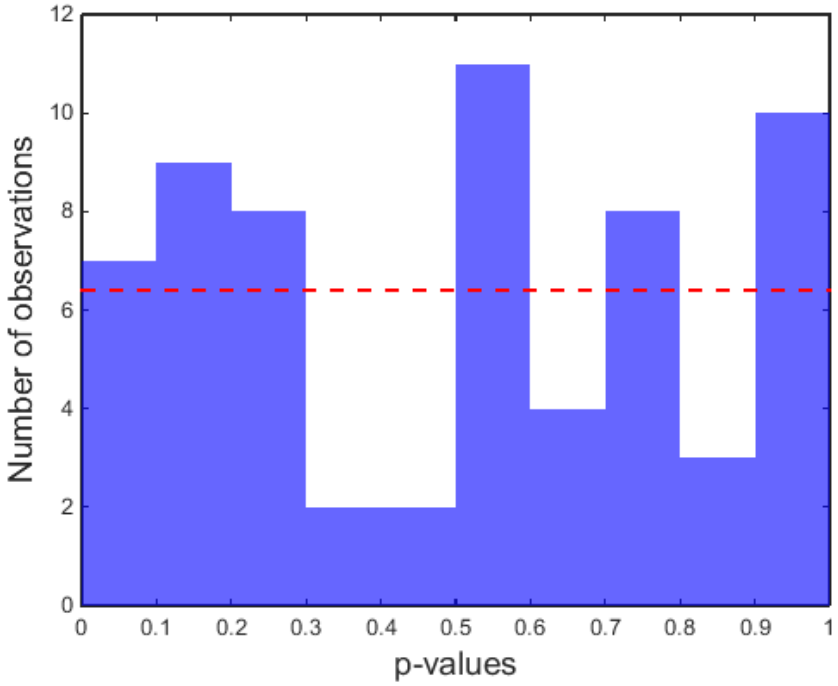


Figure 5.7: Observed p-values of the first-level von Neumann condition tests for the second VNC iteration: *The red line marks a uniform distribution.*

As for the first iteration, the p-values of all 64 performed first-level von Neumann condition tests for the second iteration vary for each sequence. To derive a final result of the test, once more, a second-level test is performed. Figure 5.7 shows the distribution of the p-values over the range [0,1]. The resulting second-level p-value is approximately 0.07. Even though this value is much smaller than the second-level p-value of the von Neumann condition test for the first iteration of 0.99, it is well beyond the defined second-level level of significance, $\alpha' = 0.0001$. Thus, the output of the output of the second VNC iteration passes the test and it is therefore concluded that the data satisfies the von Neumann conditions.

In order to derive a general conclusion of the in this section presented analysis, it is helpful to consider all the performed test in relation to each other. Table 5.2 summarizes the the main results of the executed tests for the output of the entropy source (Section 3.3) and the outcomes of the two VNC-iterations. The by the entropy

source delivered data is strongly biased (the percentage of 0's is roughly 29%). It does therefore not pass the Frequency test and fails, as such, the NIST test suite. Further more, do to dependencies in the data stream, the von Neumann condition test fails, which indicates that the VNC is not a suitable post-processor for the given entropy source.

Table 5.2: Summary of the statistical tests of VNC

	von Neumann condition test	Frequency test	NIST test suite
Source data	Failed	Failed	Failed
First VNC-iteration	Succeeded	Failed	Failed
Second VNC-iteration	Succeeded	Succeeded	Failed

Running the source data through a first VNC-iteration confirms the result of the von Neumann condition test. Instead of eliminating the bias of the applied data, the output of the first iteration has a slightly increased bias compared to the input, which yields the Frequency test to fail. As such, the proposed von Neumann condition test successfully identifies the absence of von Neumann conditions in the source data.

However, performing the von Neumann condition test on the output of the first iteration yields an acceptance of the null-hypothesis. In other words, the test concludes that the data generated by the first iteration satisfies the von Neumann conditions. Based on the related analysis, it seems likely that this positive test result is due to the compressing property of the VNC. Since the VNC only selects some of its input bits as an output, it changes the structure of the data. This seems to eliminate the dependencies that yields the von Neumann condition test to fail during the analysis of the source data. It should be stressed at this point, that this feature of the VNC is rather a "side effect" of Algorithm 1 that works for the here presented entropy source. A general, firm mathematical proof does not exist.

As expected from the result of the von Neumann condition test, performing a second VNC-iteration generates a close to perfectly unbiased output that passes the Frequency test. This means that the introduced von Neumann condition test has correctly detected the desired conditions in the data that yield to a successful performance of the VNC. However, executing the other test of the NIST test suite leads to the conclusion that the output of the second VNC-iteration can still not be considered to be random. The performed analysis of the test results indicates this is due to remaining dependencies in the bit stream.

In conclusion, by running two iterations of the VNC it is possible to transform the biased data of the in this project used entropy source to a nearly perfectly unbiased bit stream. However, even though it has been observed that the VNC removes some dependencies in the data, the final output still contains too many dependencies to be considered truly random. As such, the VNC must be considered to be an unsuited post-processor for the purpose of this project.

The in Section 3.2 proposed von Neumann condition test has been shown to successfully identify both the presence and the absence of von Neumann conditions. In other words, for data that passes the test, the VNC is able to generate an unbiased

output. As such, the test works in accordance to its definition. However, it does not take dependencies in the data into consideration. Hence, data that passes the test does not necessary result in a true random output, when applied to the VNC. As a result, it is questionable how useful it is in practice to execute the von Neumann condition test in order to evaluate if the VNC can be used for a given source.

As a final remark, it is at this point naturally to consider the performance of a third VNC-iteration. The von Neumann condition test for the output of the second iteration succeeds and it can therefore be concluded that the output of a third iteration would pass the frequency related tests of the VNC test suite. In addition, it is possible, that a further compression of the data by means of the VNC would further reduce the remaining dependencies in the data. However, exploring the possibility of a third iteration is omitted in this report, due to two reasons. First, as mentioned, no mathematical background exists that indicates that the VNC is suited to remove statistical dependencies. Hence, running a third iteration with the purpose of removing dependencies lacks a mathematical motivation. Second, to be able to evaluate the output of the third iteration, it would be necessary to generate 64 Mbits of data. With regards to the compressive character of the VNC-algorithm, this would require an enormous amount of source data, which cannot be provided by means of this project. As a result, considering a third VNC-iteration is left for further work.

5.2 Analysis of the IHF Output

The evaluation of the output of the IHF for randomness differs vastly from the corresponding analysis of the VNC presented in Section 5.1. The main reason for this is the fact that the used IHF-algorithm (Algorithm 3) can be tuned by choosing different values for the number of input bits, n_{IHF}, the number of bits generated per execution, m_{IHF}, and the tuning parameter β, defined in Equation 4.5. In contrast, the VNC-algorithm (Algorithm 1) is fixed to operate on two input bits, which possibly result in one output bit, and does not provide the possibility of any kind of tuning.

Hence, due to the variety of different possible parameter combinations for the IHF, it is out of the scope of this report to perform an as detailed analysis of the IHF as the one of the VNC, presented in Section 5.1. This section focuses therefore rather on finding combination of IHF-parameters, that yield the output data to pass the NIST test suite, and less on the detailed analysis of the statistical characteristics of the data.

In order to find a first indication of suitable parameters, it is worth to recall the functionality of the in Section 4.2.3 presented IHF-implementation. The designed implementation uses input blocks of n_{IHF} bits to generate output blocks of m_{IHF} bits, which then are concatenated to derive an output of fixed output length of 32 bits. Finally, in order to generate the by the NIST test suite required number of bits (see Table 3.3), the simulation routine of Figure 4.9 concatenates the 32-bit output of the IHF to create 64 test sequences of length $n = 1\,\text{Mbits}$.

Two important consequences derive from this concept. First, considering only the operation of the IHF on a single input block, it is important that the parameters of the post-processor satisfy Equation 4.4. The equation states that the n_{IHF}-bit input to the IHF must contain an amount of min-entropy, κ, that is equal to the sum of the output length m_{IHF} and the tuning parameter β. Second, in order to be able to concatenate the m_{IHF}-bit outputs to a truly random bit stream, each input block must actually contain κ bits of conditional min-entropy, which follows from the concept introduced during the discussion of Equation 2.10. This means that dependencies between the blocks must be taken into consideration during the choice of the parameter. For instance, as strong dependencies between the blocks result in a lower amount of conditional entropy in the input blocks, the block length n_{IHF} must be increase to gather the same amount of conditional min-entropy.

Thus, the relation of the IHF-parameter can in theory be computed based on the conditional min-entropy in the source data. However, in practice, estimating κ for the input blocks is a non-trivial task. For example, ignoring for a moment the aspect of possible dependencies between blocks, a value of the min-entropy of the individual input blocks could be estimated, by considering a large number of n_{IHF}-bit blocks of the source data. By counting the number of occurrences of each of the $2^{n_{\text{IHF}}}$ possible values of a block, the probability of the most frequently block value can be approximated. This probability could then be related to the min-entropy of the blocks, by means of Equation 2.3. However, this approach becomes fast unpractical. A block size of, for instance, $n_{\text{IHF}} = 32$ would result in 2^{32} possible outcomes. As a result, the amount of memory necessary to keep even the count of each outcome clearly exceeds the hardware resources of this project. In addition, taking the dependencies between the input blocks into consideration, it has been shown during discussion presented in Section 5.1 that the specific nature of statistical dependencies is hard to identify, even if their existence can be detected by means of statistical tests. This further impedes the estimation of the conditional min-entropy of the input data.

In order to cope with this problematic, it is therefore necessary to find a simpler approach to derive an useful estimate of the conditional min-entropy. Starting by eliminating the necessity of performing a complicated analysis of the dependencies in the source data, it is for the here presented analysis suitable to consider the conditional min-entropy to be a constant value of κ bits for all blocks. With regards to the concept described in Equation 2.11, this is of course a simplification, as the conditional (min-)entropy of a block is dependent of possible dependencies between the considered block and previous blocks. However, for long data stream, it is not unreasonable to assume that dependencies between blocks are similar for all blocks in the data. This simplification is therefore accepted at this point in order to ease the presented analysis. It should be noted, that by the concept of Equation 2.10, this means that if a n_{IHF} block of data has an estimated conditional min-entropy of κ, a block of the double size is considered to contain 2κ bit of conditional min-entropy.

Further more, an indication of the amount of conditional min-entropy in the source data can be derived by assuming that the data consists of biased but mutual independent bits. Recalling the results of the von Neumann condition test for the source data, which indicates the existence of dependencies in the data (see for instance Figure 3.3 in Section 3.3), this assumption is clearly not valid. Nevertheless,

keeping its invalidity in mind, using this assumption at this point makes it possible to find at least an upper bound for κ. This bound can then be used as an indicator for suitable parameter combinations.

As stated in Section 3.3, the percentage of 0's in the source data varies insignificantly around roughly 29%. Using this value to approximate the bias of the data, the probability that any arbitrary bit equals zero becomes $p = 0.29$. Hence, the most likely value taken by an n_{IHF}-bit input block is the sequence which contains n_{IHF} 1's. Denoting the probability of observing this outcome as $p_{\max}$ and considering Equation 2.9, it follows that $p_{\max} = 0.71^{n_{\text{IHF}}}$. By inserting this into Equation 2.4, the upper bound of the conditional min-entropy, denoted $\kappa_{\max}$, for an n_{IHF}-bit input block can be found. Table 5.3 presents this value for some suitable choices of n_{IHF}, where $\kappa_{\max}$ has been rounded to the next lowest integer.

Table 5.3: Upper bound of the min-entropy

n_{IHF}	8	16	32	64	128
$\kappa_{\max}$	4	7	15	31	63

Starting by considering an input block length of $n_{\text{IHF}} = 32$, the upper bound of the conditional min-entropy per block is $\kappa_{\max} \approx 15$. Based on the argumentation presented in Section 4.1.3, which states that reasonable minimum value of β should at least be in the range of $\beta \approx 20$, and taking Equation 4.4 into consideration, it seems unlikely that an input block length of n_{IHF} can result in even a single true random output bit. However, to verify this statement, the IHF is set up to generate a 64 Mbit output, with $n_{\text{IHF}} = 32$ and $m_{\text{IHF}} = 1$. The for the IHF required public parameter π has been chosen randomly[2]. The created output is then applied to the NIST test suite. Table 5.4 presents the main results of the tests.

Based on the presented results, the Cumulative Sums test fails the second-level approach at a level of significance of $\alpha' = 0.0001$. As such, the data does not pass the NIST test suite and cannot be considered to be truly random. It should further be noted that the Frequency test for the given data fails for more than 10% of the performed first-level tests. Given the significance of this test (see Section 2.2.2), this is an additional implication of the non-random character of the data, even though the Frequency test passes the second-level test. Hence, considering the results as presented in Table 5.4, it is concluded, that for the given entropy source an input block of $n_{\text{IHF}} = 32$ bits does not contain enough conditional min-entropy to enable the IHF to extract a single truly random output bit.

As a result, n_{IHF} must be increased. Considering $n_{\text{IHF}} = 64$, it follows from Table 5.3 that the conditional min-entropy per input block is bounded by $\kappa_{\max} = 31$. Assuming once more that $\beta \approx 20$ and using $\kappa_{\max}$ as an estimate of the available min-entropy, Equation 4.4 can be solved for $m_{\text{IHF}} = 11$. Thus, the number of output bits that can be generated by a 64-bit input block is upper bounded by 11 bits, even though a successful generation of 11 bits seems unlikely, since $\kappa_{\max}$ does not take dependencies in the source data into consideration. However, based on the stated,

[2]In this project, all concrete values of π used for simulations have been randomly selected by means of either coin tossing or rolling dices.

Table 5.4: Results of the NIST test suite for the output of the IHF with $n_{\text{IHF}} = 32$ and $m_{\text{IHF}} = 1$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	57	0.002971	Passed
Block Frequency	64	63	0.706149	Passed
Cumulative Sums	64	58	0.000068	Failed
Runs	64	64	0.706149	Passed
Longest Run	64	63	0.299251	Passed
Rank	64	64	0.602458	Passed
DFT	64	62	0.706149	Passed
NOT	64	64	0.468595	Passed
OT	64	61	0.074177	Passed
Universal	64	64	0.500934	Passed
Approx.Entropy	64	62	0.964295	Passed
Rand.Excur	34	33	0.804337	Passed
Rand.Excur.Var.	34	34	0.299251	Passed
Serial	64	64	0.949602	Passed
Linear Complexity	64	63	0.834308	Passed

it is reasonable to explore the possibility of running the IHF with $n_{\text{IHF}} = 64$ and $m_{\text{IHF}} < 11$. To evaluate this and in order to find an as large as possible value of m_{IHF}, the IHF is therefore simulated 4 times with $n_{\text{IHF}} = 64$, while m_{IHF} equals respectively 1, 2, 4 and 8 bits. The in that way generated data is then applied to the NIST test suite and a summary of the results is depicted in Table 5.5. The detailed results of each simulation are presented in Appendix C.2.3.

Table 5.5 shows that all tested output block sizes smaller or equal to $m_{\text{IHF}} = 4$ passes the NIST test suite for the in Table 3.3 defined test parameters. This means, that, for the entropy source of Section 3.1 and the parameters $n_{\text{IHF}} = 64$ and $m_{\text{IHF}} = 4$, the in Section 4.2.3 presented implementation of the IHF generates data that can be considered to be truly random, based on the evaluation of a 64 Mbit bit stream. Even though it is desirable to further verify this result by applying an increased number of test sequences, N, to the NIST test suite, this is left for further work, due to the large amount of source data necessary for such an investigation. Thus, for the purpose of this project, the in Table 5.5 presented conclusions are accepted.

However, for $n_{\text{IHF}} = 64$ and $m_{\text{IHF}} = 8$ the IHF output data clearly fails the second-level test of the Random Excursions Variant test and, as a result, does not pass the NIST test suite. This means that, while a 64-bit input block of the source data contains enough conditional min-entropy to generate 4 random bits, the actual value of κ is not large enough to extract 8 true random bits. Using the assumption that $\beta \approx 20$, it follows from Equation 4.4 that the average amount of conditional min-entropy per bit is $0.375 \leq \kappa_b < 0.4375$.

Table 5.5: Summary of the results of the NIST test suite for the IHF with $m_{\text{IHF}} = 64$

Test	Second-level p-values			
	$m_{\text{IHF}} = 1$	$m_{\text{IHF}} = 2$	$m_{\text{IHF}} = 4$	$m_{\text{IHF}} = 8$
Frequency	0.017912	0.862344	0.017912	0.931952
Block Frequency	0.213309	0.253551	0.162606	0.054199
Cumulative Sums	0.602458	0.911413	0.275709	0.602458
Runs	0.964295	0.568055	0.637119	0.178278
Longest Run	0.122325	0.500934	0.706149	0.028181
Rank	0.671779	0.671779	0.949602	0.834308
DFT	0.468595	0.407091	0.407091	0.568055
NOT	0.407091	0.407091	0.025193	0.500934
OT	0.350485	0.671779	0.407091	0.134686
Universal	0.637119	0.324180	0.637119	0.706149
Approx.Entropy	0.637119	0.976060	0.834308	0.671779
Rand.Excur.	0.275709	0.003804	0.195163	0.275709
Rand.Excur.Var.	0.739918	0.000500	0.195163	0.000001
Serial	0.534146	0.772760	0.060239	0.350485
Linear Complexity	0.178278	0.350485	0.090936	0.253551
Conclusion	Passed	Passed	Passed	Failed

Having established a combination of the parameters n_{IHF} and m_{IHF} that yields the IHF to pass the NIST test suite, it is of interest to find on this basis other combinations that yield a positive test result. This is motivated by two facts. First, further exploring possible parameter combinations might yield a better estimate of the in the source data contained conditional min-entropy.

Second, while Section 4.1 shows that the number of operations performed by the IHF increases with n_{IHF} and m_{IHF}, accepting the for this section used simplification that κ is constant for all blocks, the number of required input bits per output bit can be reduced by increasing the parameters. To see this, the combination $n_{\text{IHF}} = 64$ and $m_{\text{IHF}} = 4$ is considered. Clearly, 16 input bits are required per output bit. Assuming once more that a minimum value of β should be in the range of $\beta \approx 20$, the conditional min-entropy per block can be estimated as approximately $\kappa \approx 24$, by using Equation 4.4. Recalling that by the assumption that the min-entropy is constant, a block of double length has a doubled amount of conditional min-entropy, the conditional min-entropy per block for $n_{\text{IHF}} = 128$ is roughly $\kappa \approx 48$. Using this value and solving Equation 4.4 for the output block length yields $m_{\text{IHF}} = 28$ for $\beta = 20$. Hence, even by picking a for a digital implementation more suited output length of $m_{\text{IHF}} = 16$, using an input block length of $n_{\text{IHF}} = 128$ reduces the number of required input bits per output bit to 8.

In order to evaluate this statement, the IHF is simulated again, this time with $n_{\text{IHF}} = 128$. Based on the above discussed, it seems suitable to use output block lengths of $m_{\text{IHF}} \leq 16$. However, it has been shown during the discussion of Table 5.5 that the average amount of conditional min-entropy per bit is upper bounded by

0.4375. This means that the upper bound of κ for $n_{\text{IHF}} = 128$ is $\kappa < 56$. Using once more $\beta \approx 20$ and Equation 4.4 yields $m_{\text{IHF}} < 36$ for 128 bits per input block. It seems therefore reasonable to also evaluate $n_{\text{IHF}} = 128$ with $m_{\text{IHF}} = 32$. As a result, the IHF simulation for $n_{\text{IHF}} = 128$ is performed with output block sizes of 2, 4, 8, 16 and 32 bits[3]. Each simulation results in a total of 64 Mbit output bits, which are as before applied to the NIST test suite. While a detailed presentation of the test results is postponed to Appendix C.2.3, Table 5.6 summarizes the most important results.

Table 5.6: Summary of the results of the NIST test suite for the IHF with $m_{\text{IHF}} = 128$

Test	Second-level p-values				
	$m_{\text{IHF}} = 2$	$m_{\text{IHF}} = 4$	$m_{\text{IHF}} = 8$	$m_{\text{IHF}} = 16$	$m_{\text{IHF}} = 32$
Frequency	0.568055	0.060239	0.253551	0.500934	0.031497
Block Frequency	0.195163	0.299251	0.637119	0.275709	0.100508
Cumulative Sums	0.602458	0.739918	0.015963	0.407091	0.195163
Runs	0.500934	0.671779	0.602458	0.534146	0.437274
Longest Run	0.772760	0.324180	0.025193	0.437274	0.350485
Rank	0.195163	0.324180	0.253551	0.772760	0.437274
DFT	0.534146	0.637119	0.350485	0.706149	0.534146
NOT	0.213309	0.706149	0.500934	0.378138	0.834308
OT	0.437274	0.275709	0.043745	0.082177	0.772760
Universal	0.407091	0.178278	0.911413	0.671779	0.407091
Approx.Entropy	0.671779	0.082177	0.437274	0.500934	0.862344
Rand.Excur.	0.330628	0.004715	0.739918	0.611108	0.378138
Rand.Excur.Var.	0.330628	0.414525	0.392456	0.953553	0.888137
Serial	0.299251	0.568055	0.299251	0.378138	0.035174
Linear Complex.	0.007880	0.931952	0.739918	0.739918	0.739918
Conclusion	Passed	Passed	Passed	Passed	Passed

The results of Table 5.6 show that the five applied parameter combinations for $n_{\text{IHF}} = 128$ pass the second-level approaches of all tests in the NIST test suite. As a result, for the used entropy source, all five combinations yield the IHF to extract a data stream that can be considered to be truly random, based on the performed tests.

It should be noted that the parameter combinations of $n_{\text{IHF}} = 128$ with $m_{\text{IHF}} = 2$ and $m_{\text{IHF}} = 4$ are of minor practical relevance, as the same amount of output bits also can be generated by a smaller input block size (see Table 5.5). Nevertheless, the positive test results for $m_{\text{IHF}} = 2$ and $m_{\text{IHF}} = 4$ further strengthen the confidence in the other test results. For example, if the NIST test suite would fail for $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 4$, this would indicate that the source actually does not provide enough min-entropy to the IHF in order to extract 4 true random bits. That would also mean, that the tests for $m_{\text{IHF}} = 8$, $m_{\text{IHF}} = 16$ and $m_{\text{IHF}} = 32$ pass the test suite by

[3]It would be reasonable to also evaluate the combination $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 1$. However, this is omitted in this project, due to the large amount of necessary source data.

chance and would eventually fail, if a larger number of test sequences, N, would be applied. However, since no such indications exist, the in Table 5.6 presented results are at this point accepted for this project, leaving a more detailed verification to further work.

Based on this, since the combination $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 32$ passes the NIST test suite, it follows from Equation 4.4 that a 128-bit input block contains at least 52 bits of conditional min-entropy, assuming again that $\beta \approx 20$. Thus, the average amount of conditional min-entropy per bit is bounded by $0.40625 \leq \kappa_b$, which is in accordance to the results presented with regards to an input block length of $n_{\text{IHF}} = 64$.

To summarize the in this section presented analysis, it can be concluded that, based on the performed tests, the IHF can be used to extract true random data from the in Section 3.1 presented entropy source. For a input block length of $n_{\text{IHF}} = 64$, the IHF passes the NIST test suite for output block sizes up to $m_{\text{IHF}} = 4$. Increasing the input block size to $n_{\text{IHF}} = 128$ makes it possible to increase the output block length m_{IHF} up to 32 bits. The latter corresponds to a ratio of 4 input bits per generated output bit.

Under the in this project used assumption that the tuning parameter of the IHF is in the range of $\beta \approx 20$, the average conditional min-entropy per bit is approximated to be bounded by $0.40625 \leq \kappa_b < 0.4375$

At this point, it would be reasonable to explore the possibility of using, for instance, $n_{\text{IHF}} = 256$ in order to further increase n_{IHF}. However, this is omitted here, due to two reasons. First, the digital implementation of the IHF (Section 4.2.3) is embedded in a system that uses a fixed amount of 32 output bits. Thus increasing m_{IHF} above 32 would require an adjustment of both the post-processor module and the system model. Second, the limited time frame of this project makes it necessary to postpone an investigation of $n_{\text{IHF}} = 256$ to further work.

Chapter 6

Estimation of Power and Energy Performance

After having analyzed the post-processor implementations with regards to the statistical characteristics (Chapter 5), it remains to derive some concrete estimates of their power and energy performance. The following chapter focuses on this aspect, by first deriving power estimates from the post-processor implementations, following the procedure presented in Figure 4.7. Then, these power estimates are related to the timing and energy performance of the ADC setup of Section 3.1. This results in estimations of the energy performance for the in this report considered combinations of an ADC and the different post-processors. However, it should be recalled from Section 3.1 that the ADC cannot directly be compared to the in this report presented implementations, since the used technology is unknown. The resulting performance estimates are nevertheless considered to be sufficient for the purpose of this project.

As stated in Equation 2.47, the switching activity γ has to be taken into consideration, in order to derive meaningful power estimates from the post-processor implementations. Figure 4.7 illustrates how this can be achieved through netlist simulations. However, it has been stated during the introduction of the switching activity in Section 2.5, that γ depends on the statistical behavior of the applied input values. Thus, to be able to estimate a valid value for γ, it is important to choose the simulation parameters to match the realistic circumstances in the best possible way.

For the main clock of the system (*sys_clk* in Figure 4.8) a frequency of 50 MHz has been used during the simulations. The sampling frequency of the source module that is used to model the ADC entropy source has been 1 MHz. These values are a better fit for a computer-aided simulation than the in Table 3.1 listed values of 48 MHz and 926 kHz, respectively, that have been used during the setup of the entropy source. However, recalling once more that the ADC cannot directly be compared to the post-processor implementations and given the small difference between the simulated and the in practice used values, this simplification is considered to be acceptable.

Further more, to capture a realistic picture of the switching activity due to the statistical behavior of the source data, it has been chosen to use data that has been generated by the in Section 3.1 presented entropy source. This is especially impor-

tant for the evaluation of the VNC implementation, since the number of operations performed by this post-processor is undeterministic and directly dependent on the statistics of the input data (see Section 2.4.1).

In order to assure, that γ depicts a statistical significant average, the applied post-processor has been triggered to generate 1024 words during the simulation. This corresponds to a total of 32 kBits (32,768 bits), which is considered to be a sufficient amount of data to approximate γ, while keeping the computation requirements of the necessary simulations in a reasonable bound. It should be noted that the performed simulations restart the considered post-processor immediately after the last word has been generated. Obviously this scenario is not realistic, as most TRNGs are typically used to generate one or several words and then stay in the idle state for long time intervals. However, for modern MCUs it is not unreasonable to assume that modules as the TRNG are disabled when not needed, by disconnecting them from the supply voltage [5]. As a result, power and thus energy consumption of the post-processors can be assumed to be negligible when ever the TRNG is not active.

In the following, the power and energy performance of the VNC is analyzed in Section 6.1. Section 6.2 presents similar considerations for the IHF-implementation. A direct comparison of both implementations is postponed to Section 7.1.

6.1 Power and Energy Performance of the VNC

It has been shown in Section 5.1 that the VNC clearly does not pass the NIST test suite for the given source data. As such, implementing the VNC for an ADC based entropy source is pointless, as the desired functionality cannot be achieved. However, it is nevertheless of interest to analyze the VNC for its energy performance, due to three reasons.

First, four different approaches of the VNC-implementation (see Table 4.4) have been realized, in order to explore the benefits of clock gating in a digital system. Analyzing and comparing the related results is likely to increase the understanding of the effect of clock gating. Based on this, appropriated design choices can be made during future implementation processes of other digital systems.

Second, with respect to the in Section 4.1.2 presented comparison of the VNC and the IHF, the VNC is the more effective implementation in terms of performed operations. Hence, it is expected, that the VNC consumes less power than the IHF. Evaluating this assumption and finding a concrete measure of the difference might motivate the implementation of an alternative entropy source, which fits the VNC.

Third, Section 5.1 indicates that multiple iterations of the VNC yield improved statistical characteristics of the final output data, even though it has not been shown that in this way generated data can pass the NIST test suite. However, if this approach has a beneficial energy performance, it might be considerable to explore this possibility further, during a future project.

The most important results of the synthesis process are presented in Table F.1 and Table F.2 in Appendix F. This includes the results of the power estimations.

Figure 6.1 presents the estimated power of the VNC module for the four considered approaches, divided into leakage and dynamic power (see Equation 2.45).

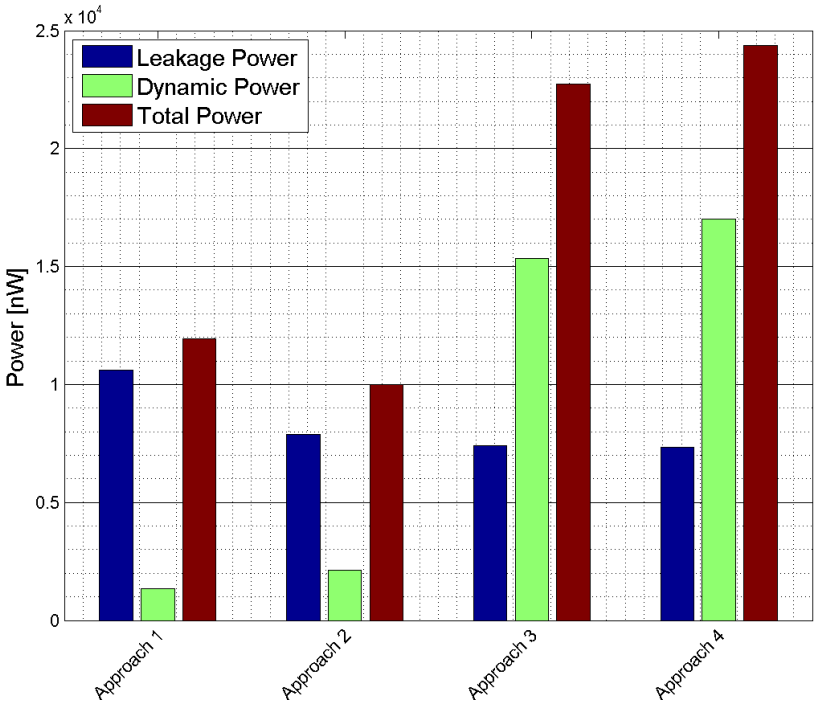


Figure 6.1: Power estimates for the VNC

Approach 4, which uses no clock gating, dissipates with approximately 24,357 nW in total the larges amount of power. The in total smallest power consumption is achieved by *Approach 2*, which uses clock gates for four or more register elements. This approach uses round about 10,002 nW, which corresponds to roughly 41% of the power consumed by *Approach 4*.

Recalling that *Approach 3* uses only one single clock gate, which has been inserted by the synthesis tool into the output memory sub-module, the power component that is due to leakage increases with the number of clock gates. This matches the expectations, as clock gates depict additional logic gates which yield an increase in the leakage currents of the module. However, the difference is relatively small, ranging approximately from 7,356 nW to 10,595 nW. Further more, it is worth noticing, that the difference between the leakage power of *Approach 2* and *Approach 4* is less than 530 nW.

In contrast, the dynamic power consumption decreases approximately from 17,001 nW to 1331 nW as the number of clock gates increases. In other words, *Approach 1* dissipates only around 8% of the dynamic power consumed by *Approach 4*. As such the clock gates achieve their purpose (see Section 2.5). It should be noted that, the difference of the dynamic power dissipation between *Approach 1* and *Apprach 2* and the same difference between *Approach 3* and *Approach 4* is with 786 nW and 1,656 nW, respectively, rather low. However, the dynamic component of the consumed power in *Approach 2* and *Approach 3* differ vastly with approximately 13,228 nW.

At this point, it can be stated, that from the four considered approaches, *Approach 2* depicts the best trade-off between increased leakage power due to additional logic used in the clock gates and reduced dynamic power. Considering that *Approach 2* only inserts clock gates in the output memory of the VNC (see Section 4.2.2), and is therefore equivalent to *Approach 3* and *Approach 4* with regards to the other sub-modules, it is of interest to explore *vnc_output_memory* in more detail. Figure 6.2 presents the power estimates of the *vnc_output_mem* sub-module for the four approaches.

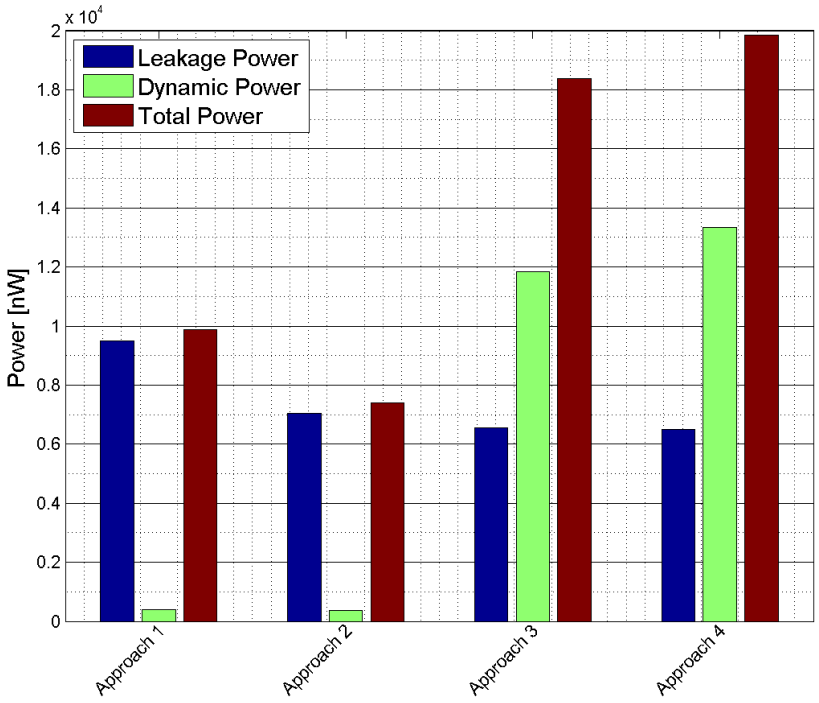


Figure 6.2: Power estimates for *vnc_output_memory*

The in Figure 6.2 presented data depicts a similar situation as shown in Figure 6.1. It is essential to note that the total power of the output memory sub-module stands in the case of *Approach 1*, *Approach 3* and *Approach 4* for roughly 80% of the total power dissipated in the VNC. Even though this ratio is slightly reduced to 74% for *Approach 2*, *vnc_output_memory* has thus a major impact on the total power performance of the VNC module.

As for Figure 6.1, the leakage power of the output memory increases with the number of used clock gates. However, the dynamic power consumption behave slightly different. As before, *Approach 3* consumes less dynamic power than *Approach 4*, and from *Approach 3* to *Approach 2* the dynamic component is majorly reduced. Nevertheless, in contrast to Figure 6.1, *Approach 1* dissipates slightly more dynamic power than *Approach 2*, even though the difference is negligible at approximately 17 nW. There are several reasons that could cause this observation. One possible explanation is the fact that clock gates, as presented in Figure 2.10a, contain logical gates, and as such consume some dynamical power themselves. Given the insignificant difference of only 17 nW, it is also possible that this observation is due to inaccuracies during the estimation process. However, with regards to the

limited time frame of this project, exploring this issue in detail is left for further work.

Disregarding the difference in dynamical power between *Approach 1* and *Approach 2*, it is worth to recall that both approaches use clock gates to control the output registers of *vnc_output_memory*, as depicted in Figure 4.14. Thus, with respect to Figure 6.2, using clock gating for the output register bank seems to account for the main reduction of the dynamical component of the power consumption of the sub-module. For *Approach 2*, the dynamical power reaches a minimum of approximately 365 nW, which corresponds to roughly 3% of the dynamical power of *Approach 4*. In contrast, the by the synthesis tool inserted clock gate of *Approach 3*, which controls the counter of the output memory sub-module (Section 4.2.2), reduces the dynamical power only with roughly 1496 nW, to about 89% of the corresponding power component in *Approach 4*.

As a result, it can be clearly stated, that for the presented VNC-implementation, *Approach 2* describes the most beneficial solution with regards to power consumption. Therefore, only a realization based on *Approach 2* is considered during the following estimation of the energy performance of the VNC.

However, before focusing on the energy requirements of the VNC, it is reasonable to summarize the main conclusions drawn from the comparison of the four approaches. Considering Figure 6.1 and Figure 6.2, it can be clearly stated that the output memory dissipates the largest amount of the by the VNC consumed power. From Section 4.2.2, it is known that this sub-module contains 32 output registers that are accessed at an undeterministic rate far below of the applied clock frequency. The above considered data shows that by clock gating this registers and the in the sub-module embedded counter, the dynamic power of the output memory can be reduced to roughly 3% compared to a solution that uses no clock gates. However, the minimum total power dissipation for *vnc_output_memory* is achieved when groups of four registers are applied to one common clock gate (*Approach 2*). Using one clock gate per output register (*Approach 1*) yields a significant increase in leakage power. Further more, the data indicates the possibility that exhaustive clock gating actually increases the dynamical power consumption when compared to *Approach 2*.

Taking the whole VNC module into consideration, *Approach 1* further reduces the dynamical power. Nevertheless, the introduced leakage power exceeds the reduction in the dynamic component. It is worth noticing that this indicates that the in Section 4.2.2 observed slight reduction of necessary control logic for *Approach 1* compared to the other approaches has no noteworthy effect on the overall power consumption.

As such, *Approach 2* is the optimal tested solution, reducing the total power consumption to approximately 10,002 nW, which corresponds to about 41% of the power dissipated by a VNC solution that does not use clock gates.

As a final remark, it should be stressed that, while *Approach 2* is the most suitable solution of the here tested approaches, it is possible that even more efficient solutions exist. For instance, approaches that uses a common clock gate for register groups of two or eight registers or even an approach that uses one clock gate for the entire 32-bit register bank could possibly derive better results. Even though this is

a highly interesting aspect, exploring this issue is left for further work, due to the limited time frame of the project.

To estimate the energy performance of the VNC, it is necessary to find the time interval during which the VNC consumes the above discussed power. As mentioned during the introduction of this chapter, it is assumed that the post-processors only consume power during the generation of a word. Even though a small number of control steps are performed before the entropy source is activated and after it is deactivated, the time interval needed to generate an output word is clearly dominated by the sampling periods of the ADC.

Using the in Table 3.1 summarized data for the ADC based entropy source, the time the ADC uses to generate a single input bit for the VNC is roughly 1.08 μs. However, considering the in Section 5.1 presented observations, an average of 4.52 input sample are required to produce a single output bit during the first iteration of the VNC. Thus, the average time required by the VNC to produce a single bit is approximately 4.88 μs. Considering the above discussed power consumption, it follows that the average energy required by the VNC to generate a single output bit is roughly 48.8 pJ.

Widening the focus to the energy performance of the entire TRNG design, the entropy source has to be taken into consideration. Table 3.1 states that the ADC uses approximately 1.25 nJ to create a single VNC input. Hence, the required energy to create the 4.52 bits that are on average needed by the VNC is around 5.65 nJ. As a result, the total energy consumed by a TRNG which is realized by the combination of the ADC and the VNC amounts to approximately 5.7 nJ per single output bit. This means that the energy required by the VNC corresponds to under 1% of the total energy consumption of the TRNG. Table 6.1 summarizes the main results of this estimation.

Table 6.1: Energy estimates for the VNC

	First iteration	Second iteration	Third iteration
Input bits per output bit	4.52	20.8	83,2
Time per output bit	4.88 μs	22.46 μs	89.86 μs
VNC energy per output bit	48.8 pJ	224.6 pJ	898.6 pJ
ADC energy per output bit	5.65 nJ	26 nJ	104 nJ
Total energy per output bit	5.7 nJ	26.2 nJ	104.9 nJ

As mentioned during the introduction of this section, it is reasonable to also consider the energy performance of an approach that uses several iterations of the VNC. However, to do so, some simplifying assumptions with regards to the power and timing characteristics are required.

Considering first the power performance of such an approach, it is clearly dependent on the method used to realize the iterations. One possibility is, for instance, to first use the VNC implementation to perform the first iteration and store the output in some form of external memory. For further iterations the VNC can then operate on the data stored in the memory. While this does not effect the VNC implemen-

tation directly, it is reasonable to assume that the external memory has a shorter response time per requested bit than the ADC, which yields a different switching activity γ. Another possibility is to change the VNC implementation to directly perform two or several implementations on the data, by inserting some additional logic and registers into the system, which of course effects the power requirements. However, due to the limited time frame of this project, these different approaches and the corresponding effect of the power performance cannot be explored in this report. For simplicity it is therefore assumed that the above presented power estimate is also valid in the case of several iterations.

With respect to the timing performance, it is reasonable to assume that the timing is still dominated by the ADC. Thus the time necessary to generate one bit is the product of the ADC sampling period and the average number of input bits per output bit. It has been presented in Section 5.1 that approximately 4.6 input bits per output bit are requested during the second iteration. With respect to the amount of bits required by the first iteration, this yields a total of roughly 20.8 source bits per output bit of the second iteration. A simulation of a possible third iteration has not been performed and thus no observed input-output-ratio exists. However, recalling from Section 5.1 that the output of the second iteration satisfies the von Neumann conditions and is unbiased, it can be assumed with regards to Equation 4.1 that a third iteration requires roughly 4 input bits per output. Hence, the total amount of ADC requests per third iteration output bit can be approximated to 83.2.

Table 6.1 summarizes the energy performance of the second and a possible third iteration. It can be concluded that the energy consumption of an iteration based TRNG increases by a factor of roughly 4 for each additional iteration. It is reasonable to assume that this statement also holds for cases in which more than three iterations are used. It should be noted, that the energy performances of the ADC and the VNC increase with the same factor. Thus, the ratio of the two components is unaffected, and the VNC in general accounts for less than 1% of the energy consumption of the discussed TRNG solution.

6.2 Power and Energy Performance of the IHF

It has been shown in Section 4.1 that the number of operations performed by the IHF, and as such the power and energy requirements of the post-processor, depends on the choice of parameters used to realize the algorithm. During this project, the main focus has been on the size of the input blocks, n_{IHF}, and the corresponding number of output bits, m_{IHF}. This yields in principle to a huge number of possible combinations that could be considered. However, for convenience, the following section focuses on those combinations which have successfully passed the in Section 5.2 presented NIST test suite. In addition, the combination $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 1$ is considered. Even though this combination has not been tested explicitly, with regards to the in Table 5.6 presented results, it is reasonable to assume that it would pass the NIST test suite.

The power estimates of the IHF for different parameters can be found in Table F.3 and Table F.4 and are summarized in Figure 6.3. The figure shows that the dissi-

pated power of the IHF module increases with both n_{IHF} and m_{IHF}. This observation is in accordance with both the complexity analysis and the description of the implementation, presented in Chapter 4, which imply that an increase of the parameter leads to additional hardware in form of combinatorial logic and registers. With regards to Equation 2.45, the result is an increase in the total power consumption of the IHF module. Thus, the smallest power dissipation of approximately 107,308 nW is achieved by the solution that uses $n_{\text{IHF}} = 64$ and $m_{\text{IHF}} = 1$. The combination of $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 32$ consumes with roughly 219,880 nW the largest amount of power.

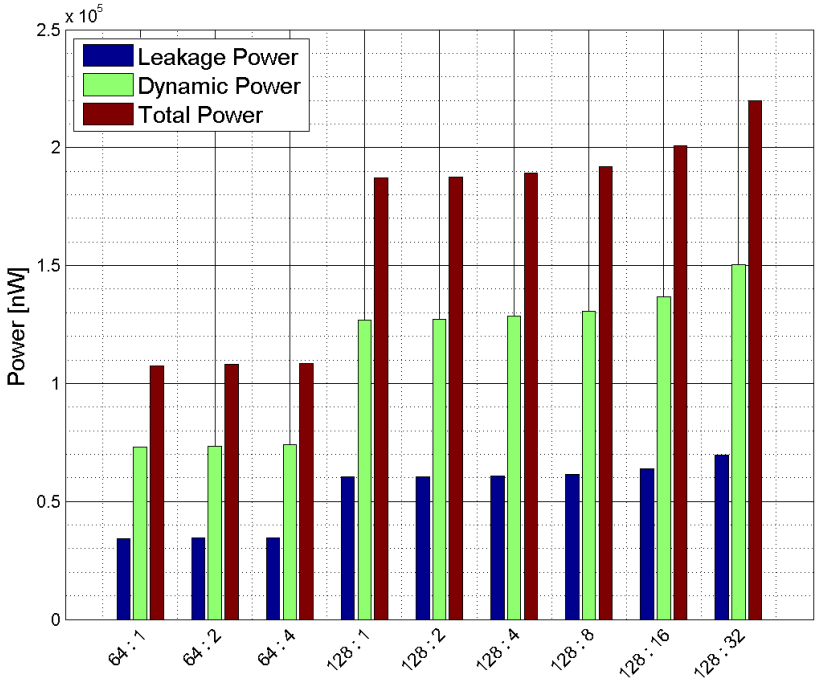


Figure 6.3: Power estimates for the IHF

In general, the in Appendix F presented data indicates that both the dynamical and the leakage power increase with the parameter choice. Nevertheless, it should be noted that for some modules with similar parameter choices the data does not support this statement. For instance, for $n_{\text{IHF}} = 64$ the estimated leakage power with $m_{\text{IHF}} = 4$ is slightly smaller than for $m_{\text{IHF}} = 2$. However, it is likely that this observation is caused by differences in the length of the the simulations used to estimate the switching activity γ. Recalling that both modules have been simulated to generate a total of 32 kBits, due to the different parameter choices, the simulation of the modules differ both in length and in the number of used source bits. For example, while the combination $n_{\text{IHF}} = 64$ and $m_{\text{IHF}} = 2$ requires 1,048,576 bits from the entropy source, $m_{\text{IHF}} = 4$ uses only 524,288 source bits. This difference in the simulations is likely to yield variations in the power estimates. As a result, it is in this report assumed that the discussed unexpected decreases in power are due to inconsistencies during the power estimation. Given the negligibleness of the differences, this aspect is therefore ignored for the purpose of this project.

Another general factor that should be noted from Figure 6.3 is the fact that the

dynamic power component is more than twice as large for all considered combinations. With regards to Equation 2.47, one reasonable explanation for this observation is a high switching activity in the module.

Before focusing on the time needed by the different parameter combinations to generate a single bit, it is of interest to consider the rate at which the power consumption for the IHF increases with m_{IHF}. Figure 6.4 illustrates[1] the growth of the dissipated power as a function of m_{IHF} for $n_{\text{IHF}} = 128$. The figure shows that the power consumption growths approximately linear from 187,286 nW to 219,881 nW. This rather minor growth can be explained by the fact that, as stated in Section 4.2.3, only one sub-module, *vnc_comp*, is directly affected by m_{IHF}.

At first sight, a linear growth of the consumed power seems to contradict the corresponding statements of Section 4.1, which indicate a quadratic growth of the required number of operations (see for instance Figure 4.2). However, this can be explained by comparing the formulation of Algorithm 3 in Section 2.4.2 with its in Figure 4.20 depicted implementation, *ihf_comp*. Only the inner loop of Algorithm 3 is explicit realized in hardware. simplified, this means that only this loop affects the power consumption of the module. The number of iterations for the inner loop is dependent of n_{IHF}. Even though the relation between n_{IHF} and m_{IHF} is rather loosely defined by means of Equation 4.4, it is reasonable to assume a close to linear relation. Hence, the linear growth of the power dissipation matches the expectations for the used implementation.

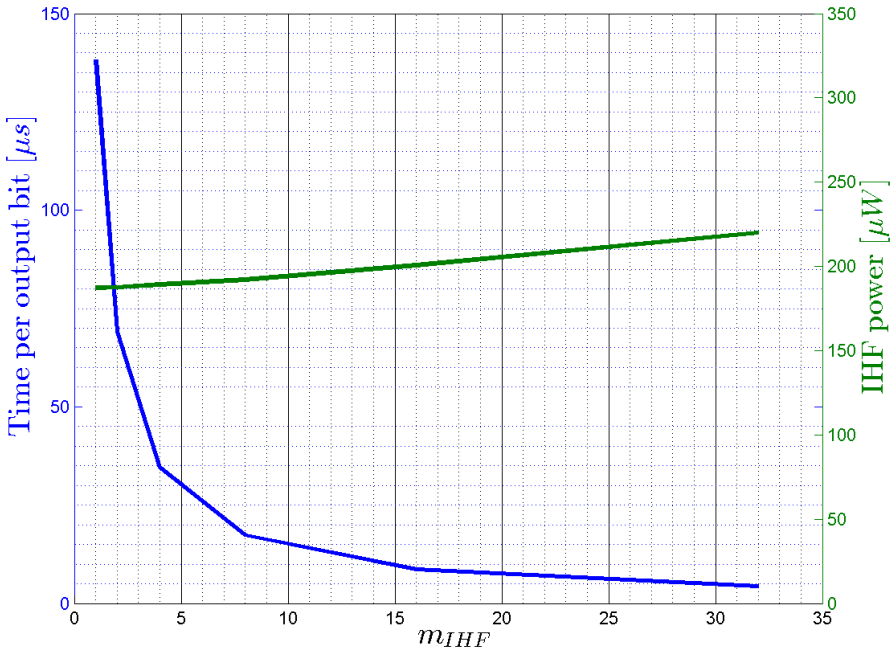


Figure 6.4: Power and time per sample for the IHF

The outer loop of Algorithm 3 is realized by running *ihf_comp* m_{IHF} times in a

[1] A similar plot could have been derived from $n_{\text{IHF}} = 64$. However, this is omitted here as only three estimates are available.

row, while shifting the public parameter register in accordance to the description of the algorithm. Since this computation cycle is not performed in parallel with the process of gathering data from the entropy source (see Figure 4.17), the time the IHF uses to generate the output data, and hence the time during which it consumes power, increases with m_{IHF}. However, it is essential to note that this does not affect the time used to generate an output bit. For example, for $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 1$, the outer loop of Algorithm 3 has to be performed only once, which corresponds to one clock cycle. Nevertheless, for this parameter combination, the algorithm has to be performed 32 times. Thus the total time spend performing the algorithm equals 32 clock periods. It is evident that this equals the time used by the combination $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 32$, which performs the algorithm once but with 32 outer loop iterations.

Thus, since the time required to perform Algorithm 3 corresponds to one clock cycle per output bit, and considering the relation between the clock frequency and the sampling frequency presented in Table 3.1, it is suitable to adopt the approach of Section 6.1, which approximates the time during which the post-processor consumes power with the time the ADC requires to produce the necessary input bits. As stated in Table 3.1, the ADC spends roughly $1.08\,\mu$s for the generation of one output bit. The number of required input bits per output bit is equal to the ratio of the input and output parameters, $\frac{n_{\text{IHF}}}{m_{\text{IHF}}}$. Hence, the time used to generate one output bit can be approximated by $1.08 \cdot \frac{n_{\text{IHF}}}{m_{\text{IHF}}}\,\mu$s. The results of this approximation for the tested parameter combinations are presented in Table 6.2 and Figure 6.4 illustrates the required time as a function of m_{IHF}. The figure shows that the time required per output bit decreases inversely proportional with m_{IHF}, from approximately $138\,\mu$s to $4\,\mu$s.

Comparing the two functions in Figure 6.4 it is evident that the time factor dominates the growth of the energy performance of the IHF. While the consumed power increases with a factor less than 1.2, the required time can be decreased by a factor of roughly 34.5 by increasing m_{IHF} from 1 to 32. Hence, the energy of the IHF is more strongly affected by its time factor than by its power component, and decreases with an increasing m_{IHF}. This is illustrated in Figure 6.5, and the corresponding results for each considered combination can be found in Table 6.2. Note that the minimum energy requirements of the IHF correspond to the parameter combination $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 32$, which requires approximately 946 pJ per output bit.

In addition to the IHF, the energy performance of the ADC has to be taken into consideration in order to derive a full picture of the requirements of the whole TRNG. The energy required by the ADC per single generated bit is roughly 1.25 nJ as stated in Table 3.1, and the energy dissipated to produce the required input bits for a single IHF output bit can thus be found as $1.25 \cdot \frac{n_{\text{IHF}}}{m_{\text{IHF}}}$ nJ. It is evident that this means that also the energy used to supply the ADC decreases inversely proportional with m_{IHF}, as depicted in Figure 6.5. Hence, the total energy of the considered TRNG design, which is the sum of the energy required by the IHF and the ADC, decreases with m_{IHF}. For the in this project considered cases the minimum energy per bit is reached for $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 32$. This combination requires approximately 5.9 nJ per bit.

Finally, it should be noted that, while the power dissipated by the IHF increases

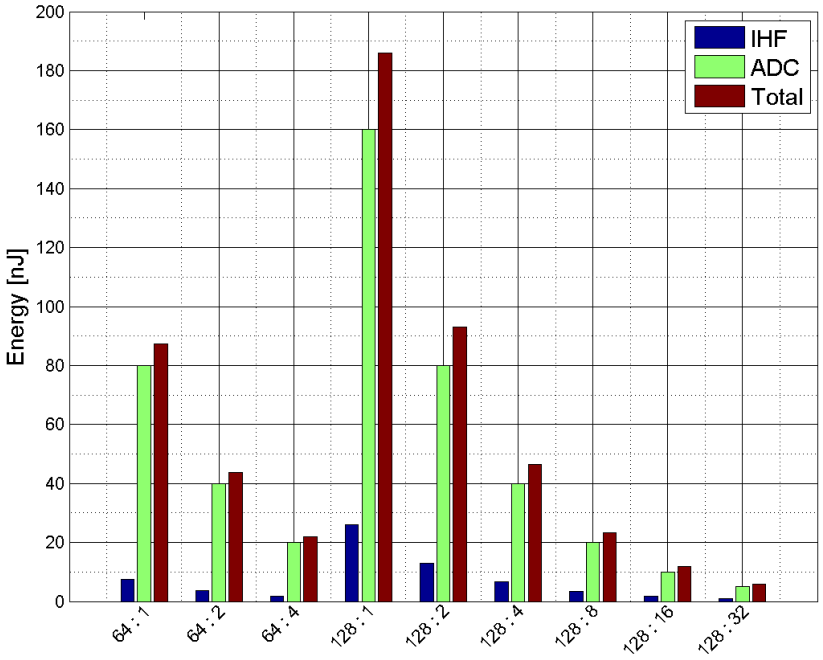


Figure 6.5: Energy estimates for an IHF based TRNG solution

slowly with m_{IHF} (Figure 6.4), the power component of the energy consumed by the ADC is independent of m_{IHF}. Hence, the required energy decreases faster for the ADC than for the IHF, for the same increase of m_{IHF}, as can be seen in Figure 6.4. As a result, the percentage of the total energy used to supply the TRNG that corresponds to the IHF increases with m_{IHF}. However, the contribution of the IHF to the total energy consumption is in general rather low. For $n_{\text{IHF}} = 128$, it decreases from roughly 14% for $m_{\text{IHF}} = 1$ to 16% for $m_{\text{IHF}} = 32$, and for $n_{\text{IHF}} = 64$ it accounts for around 8.5%.

Table 6.2 depicts a summary of the for the energy estimation relevant data, for all considered parameter combinations of the IHF.

In conclusion, the obtained power estimates for the IHF show that the dissipated power increases with both n_{IHF} and m_{IHF}. Generally, this growths affects both the leakage power and the dynamic power components. Further, it can be stated that the dynamic power consumed by the IHF is more than twice as large as the leakage power, for the same parameter choice.

Even though the power requirements slightly grow, increasing m_{IHF} while keeping n_{IHF} fixed decreases the time used to generate a single bit drastically. This has a major impact on the energy performance of both the IHF and the ADC. As a result, the minimum consumption of the in this project considered parameter combinations is achieved by $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 32$. This combination yields a total energy dissipation of approximately 5.9 nJ per output bit. It is worth to recall from Section 5.2, that this combination passes the NIST test suite for the given test parameters.

Table 6.2: Energy estimates for the VNC

		Power	Time per output bit	Energy per bit		
				IHF	ADC	Total
$n_{\mathrm{IHF}} = 64$	$m_{\mathrm{IHF}} = 1$	107 µW	69.1 µs	7.4 nJ	80 nJ	87.4 nJ
	$m_{\mathrm{IHF}} = 2$	108 µW	34.5 µs	3.7 nJ	40 nJ	43.7 nJ
	$m_{\mathrm{IHF}} = 4$	108 µW	17.2 µs	1.9 nJ	20 nJ	21.9 nJ
$n_{\mathrm{IHF}} = 128$	$m_{\mathrm{IHF}} = 1$	187 µW	138.2 µs	25.8 nJ	160 nJ	185.8 nJ
	$m_{\mathrm{IHF}} = 2$	188 µW	69.1 µs	13 nJ	80 nJ	93 nJ
	$m_{\mathrm{IHF}} = 4$	189 µW	34.6 µs	6.5 nJ	40 nJ	46.5 nJ
	$m_{\mathrm{IHF}} = 8$	192 µW	17.3 µs	3.3 nJ	20 nJ	23.3 nJ
	$m_{\mathrm{IHF}} = 16$	201 µW	8.6 µs	1.7 nJ	10 nJ	11.7 nJ
	$m_{\mathrm{IHF}} = 32$	220 µW	4.3 µs	946 pJ	5 nJ	5.9 nJ

Chapter 7

Discussion, Conclusion and Further Work

In this chapter, the most important aspects of the VNC and IHF for this project are compared and discussed (Section 7.1). In Section 7.2, a final conclusion of the presented project is given, including a list of topics suitable for further investigation.

7.1 Comparison of the VNC and the IHF

Comparing the VNC to the IHF is a non-trivial task, due to the reasons that both post-processors require different statistical characteristics in the source data (Section 2.4). While the IHF can operate on data as long as it contains a sufficient amount of min-entropy, the VNC requires explicit statistical conditions, in form of independent and identically distributed bits. However, with regards to the performed analyses in Section 3.3 and Chapter 5, it can be stated that the requirements of the VNC are not satisfied by the entropy source used in this project. Further more, considering the results of a performed second VNC iteration (see Section 5.1), it has been observed that while the VNC generates an unbiased output for data that satisfies the von Neumann conditions, the output cannot necessarily be considered to be random, due to dependencies. In contrast, as shown in Section 5.2, the IHF can easily be tuned to pass the NIST test suite, by increasing the number of input bits per output bit.

Due to the fact that the IHF passes the performed test for randomness, while the output of the VNC is rejected by the NIST test suite, it is difficult to directly compare their performance with regards to power and energy consumption. A first indication is given in Section 4.1.2, which compares the theoretical number of operations and the number of required input bits for both algorithms, under the assumption of von Neumann conditions. With regards to Figure 4.5, this analysis shows that the VNC is expected to perform a much smaller number of operations compared to the IHF. This is an indication of a lower energy consumption in the VNC post-processor. However, Figure 4.3 and Figure 4.4 show that the number of input bits per output bit is lower for the IHF, which reduces the number of required source bit requests, and as such, the energy dissipated in the entropy source. It should be recalled from Chapter 6 that the used entropy source consumes more

than 99% and 84% for a VNC and an IHF based design, respectively. Hence, reducing the dominating energy requirements of the entropy source by improving the ratio between input and output bits seems more suitable for the purpose of this project.

Considering the estimated power consumption of the two implemented post-processors, the VNC consumes with 10 μW below 5% of the approximately 220 μW dissipated by the most energy efficient variant of the IHF. This observation can be considered to be in accordance with the situation as depicted in Figure 4.5. However, these numbers can be used only as a fist indicator and not for direct comparison, due to two reasons. First, the IHF has been tuned to pass the NIST test suite, which the VNC fails. Assuming a different entropy source which fits the VNC, it is reasonable to assume that the input data would contain a larger amount of min-entropy due to reduced dependencies. As such, in a scenario where both post-processors could be used, the IHF might be tuned to a more power efficient solution. Second, power saving clock gates have been included into the design of the VNC, which reduce the dissipated power with roughly 59% (see Section 6.1). Recalling that this is mainly due to the reduction of the power consumption in the memory element of the VNC and with regard to the fact that the IHF makes use of two similar storage sub-modules (Figure 4.18), it is reasonable to assume that introducing clock gates into the IHF module would lead to a reduction in the power consumption. Thus for a fair comparison of the two post-processor implementations, clock gates should be inserted into the IHF in a similar manner as for *Approach 2* for the VNC implementation (Section 4.2.2). Nevertheless, given the fact that the IHF performs more complex computations than the VNC and in addition requires both input and output memory, it is reasonable to assume that its power dissipation exceeds that of the VNC.

Finally, with regard to the energy consumption, the most efficient solution of the tested IHF approaches, uses approximately 5.9 nJ per output bit, where the energy consumed by the ADC is included. This is roughly equivalent to the VNC approach which requires around 5.7 nJ. However, it should be once more stressed, that the latter result is irrelevant as the output of the VNC cannot be considered to be random. Section 5.1 and Section 6.1 briefly consider the possibility of deriving a random output by iterating the VNC post-processing several times. Even though this approach has no mathematical proof and as such it must be considered to be rather a pseudorandom solution, Table 5.1 shows some first improvements in the data, with respect to randomness. Nevertheless, while the second iteration still not passes the NIST test suite and it has not been possible to evaluate the test suite for a third iteration, the energy consumption of a third VNC iteration has been approximated to be 105 nJ per output bit. As such, while it has not been shown that the output of a third iteration can be considered as random, performing this iteration requires more than 20 times more energy per output bit than the IHF based approach. Hence, lacking both a formal mathematical proof and a motivation based on an improved energy consumption, a further investigation of the possibility of using multiple VNC iterations is not of interest for this project.

7.2 Conclusion and Further Work

During the here presented project, it has been shown that an ADC embedded in a typical MCU can be used as an entropy source for a TRNG, without modifying the existing implementation of the ADC. Nevertheless, the low entropy rate of the data generated by the ADC makes post-processing necessary. While the *von Neumann Corrector* (VNC) cannot be used to mask the statistical imperfections of the source data, applying the *Extractor based on pairwise Independent Hash Functions* (IHF) results in an output which can be considered as random, based on the evaluation of a 64 MBit output, using the NIST test suite. The energy consumption of the resulting combination of the ADC and the IHF is approximated to be 5.9 nJ per output bit. This value must be considered to be quite high compared with existing solutions. For example, the design proposed in [3] requires approximately 128 pJ per output bit.

Considering the entropy source as proposed in this report, three main disadvantages can be identified when using an unmodified ADC as a source. First, the delivered data is biased and contains dependencies, which yields a low entropy rate. This makes the use of the VNC impossible and makes it necessary to use a large number of input bits for the IHF. Second, the ADC consumes approximately 1.25 nJ per source output bit. As such, the entropy sources dominates the energy consumption of the TRNG designs, dissipating at least 84% of the total energy, for the considered solutions. Third, the sampling period of the ADC is long compared to the clock that drives the post-processors, which increases the dissipated energy unnecessarily. Hence, even though possible, the implementation of an entropy source purely based on an ADC must be considered unsuited for an energy efficient TRNG design.

To ease the analysis of the source data and enable a choice of a suitable post-processor early on in the design phase, a novel statistical test for von Neumann conditions has been proposed. For the performed analyses, the test proved to be able to identify both the absence and the presence of von Neumann conditions in the data. In that manner, it works as specified. However, even though the results of the test are correct and useful for the analysis of the source, passing the test does not imply that the corresponding output of the VNC passes the NIST test suite. This is due to the reason that dependencies in the data can exist even in the presence of von Neumann conditions.

Comparing the VNC and the IHF, the presented project illustrates the benefits of a tunable post-processor. While the VNC simply fails the NIST test suite, the IHF can be adapted to the low level of entropy in the source data, by increasing the number of input bits per operation. Further more, Section 6.2 shows how the IHF can be tuned to find a trade-off between power efficiency of the post-processor on one hand and an improved ratio between the number of input bits and the number of output bits. This enables the optimization of the energy performance of the overall TRNG design by choosing appropriated parameters for the IHF. In contrast, the VNC has a low, but fixed power consumption and its input-output-ratio is dependent on the statistical characteristics of the source. Hence, the IHF seems in practice more suitable for the realization of a TRNG.

In order to improve the energy performance of the post-processors, two ap-

proaches have been explored separately. On one hand, for the IHF energy optimizations have been performed by first selecting an appropriated algorithm out of two choices and then finding an optimal parameter combination. This yields the above mentioned TRNG solution, consuming 5.9 nJ per output bit. One the other hand, the impact of clock gating on the power dissipation has been explored for the VNC. The results show that, while introducing clock gates for all possible registers reduces the dynamic power, it yields a noticeable increase in leakage power. However, using one clock gate for blocks of at least four registers leads to a nearly as low dynamical power component, while at the same time keeping the increase of leakage power at a minimum. For a considered 32-bit memory element, containing infrequently accessed registers, this approach yields a reduction of the dynamical power of roughly 97%. Even though, this has not been tested, it is reasonable to assume that adopting clock gating in a similar manner to the IHF will lead to a drastic decrease in the power consumption.

Based on the analysis presented in this project, the proposed TRNG using an ADC and the IHF, with an input of 128 bits and an output size of 32 bits, can be considered to produce random data. However, it should be stressed that a further investigation of this statement is desirable. During this project, evaluating data by means of the NIST test suite has been limited to blocks of data of 64 Mbit, due to limitations in time and hardware. In practice, this is a rather small amount of data, and exhaustive testing is necessary to increase the confidence in the proposed design.

In addition, it is reasonable to try to further optimize the presented design in terms of energy performance. One possible approach could be to increase the number of both the input and the output bits of the IHF. While this is expected to increase the power dissipated by the IHF, it can possibly yield a further improvement of the input-output-ratio and thus decrease the overall energy consumption of the TRNG.

It has been shown that roughly 66% of the power consumption of the IHF is due to dynamic power dissipation. Besides optimizing the design by tuning, the for the VNC tested clock gating should therefore be applied to the memory sub-modules of the IHF. As they are similar to the memory block of the VNC, it is reasonable to expect a large decrease in the dissipated power. If this design technique is used, it should be kept in mind that further investigation of the number of registers per clock gate could yield to even more optimized results. In addition to clock gating, it has been mentioned in Section 4.2.3, that guarded evaluation is likely to reduce the dynamic power component of the IHF. Also this design technique should therefor be explored.

However, even though the above presented further optimizations of the given design might yield a minor improvement of the energy performance, the main focus of further investigations should be to reduce the energy consumption of the source. An alternative solution of using the ADC by storing the LSBs of sampling processes that precede the activation of the TRNG has been briefly introduced in Section 3.1. This would hide the cost of producing the source data, does not require noteworthy modifications of the ADC and could be evaluated by first using a software based approach, as done for the source, considered in this report. However tuning the IHF to fit the data and meet strict energy requirements is a non-trivial task, due to the

uncertainty about the output data.

Generally, it should be evaluated if an alternative for the ADC should be used as an entropy source. One possibility is the use of jitter in clock signals produced by digital ring oscillators, as briefly presented in Section 2.3. This approach is frequently discussed in the literature and is likely to have an output with an higher entropy rate, a shorter sampling period and a lower power consumption. In addition, it should be noted that this approach can be implemented using digital circuitry. It is therefore possible to perform a first analysis of the source by using a *Field-Programmable Gate Array* (FPGA), which is much simpler and cheaper than to create a CMOS prototype.

Chapter 8

References

[1] W. Trappe and L. C. Washington, *Introduction to Cryptography with Coding Theory*, 2nd ed. Pearson Prentice Hall, 2006.

[2] B. Jun and P. Kocher, "The intel random number generator", Cryptography Research Inc, Tech. Rep., 1999.

[3] M. Bucci and R. Luzzi, "Fully digital random bit generators for cryptographic applications", *IEEE Transactions on Circuits and Systems - I: Regular Papers*, vol. 55, no. 3, pp. 861–875, 2008.

[4] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Editorial: Internet of things", *International Journal of Communication Systems*, vol. 25, no. 9, pp. 1101–1102, 2012.

[5] Silicon Laboratories. (2013). EFM32WG Reference Manual, [Online]. Available: `www.silabs.com/Support\%20Documents/TechnicalDocs/EFM32WG-RM.pdf`.

[6] NIST, "Advanced encryption standard", *Federal Information Processing Standards Publication*, 2001.

[7] B. Schneier and N. Ferguson, *Practical Cryptography*. John Wiley & Sons, 2003.

[8] J. von Neumann, "Various techniques used in connection with random digits", *Applied Math Series*, pp. 36–38, 1951.

[9] B. Schneier, *Applied Cryptography*, 2nd ed. John Wiley & Sons, 1996.

[10] B. Sunar, W. J. Martin, and D. R. Stinson, "A provably secure true random number generator with built-in tolerance to active attacks", *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 109–119, 2007.

[11] F. Pareschi, G. Setti, and R. Rovatti, "Implementation and testing of high-speed CMOS true random number generators based on chaotic systems", *IEEE Transactions on Circuits and Systems - I: Regular Papers*, vol. 57, no. 12, pp. 3124–3136, 2010.

[12] C. G. Foik, *Analysis of Hardware Solutions for True Random Number Processing*, NTNU, 2014.

[13] D. J. C. MacKay, *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[14] B. Barak, R. Shaltiel, and E. Tromer, "True random number generators secure in a changing environment", *Workshop on Cryptographic Hardware and Embedded Systems*, pp. 166–180, 2003.

[15] R. E. Walpole, R. H. Myers, S. L. Meyers, and K. Ye, *Probability & Statistics for Engineers and Scientits*, 9th ed. Pearson.

[16] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. John Wiley & Sons, 2006.

[17] J. L. Hodges, Jr. and E. L. Lehmann, *Basic Concepts of Probability and Statistics*. Holden-Day, 1964.

[18] NIST, "A statistical test suite for random and pseudorandom number generators for cryptographic applications", *NIST Special Publication 800-22*, 2010.

[19] F. Pareschi, R. Rovatti, and G. Setti, "Second-level NIST randomness test for improving test reliability", *IEEE International Symposium on Circuits and Systems*, pp. 1437–1440, 2007.

[20] K. Wold and C. H. Tan, "Analysis and enhancement of random number generator in FPGA based on oscillator rings", *Internationl Journal of Reconfigurable Computing*, 2009.

[21] D. J. Murdoch, Y.-L. Tsai, and J. Adcock, "P-values are random variables", *The American Statistican*, vol. 62, no. 3, pp. 242–245, 2008.

[22] D. Naccache and D. M'Raïhi, "Cryptographic smart cards", *IEEE Mirco*, vol. 16, no. 3, pp. 14–24, 1996.

[23] N. C. Göv, M. K. Mıhçak, and S. Ergün, "True random number generation via sampling from flat band-limited gaussian processes", *IEEE Transactions on Circuits and Systems - I: Regular Papers*, vol. 58, no. 5, pp. 1044–1051, 2011.

[24] W. T. Holman, J. A. Connelly, and A. B. Dowlatabadi, "An integrated analog/digital random noise source", *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications*, vol. 44, no. 6, pp. 521–528, 1997.

[25] T. C. Carusone, D. Johns, and K. Martin, *Analog Integrated Circuit Design*, 2nd ed. John Wiley & Sons, 2012.

[26] S. Haykin, *Communication Systems*, 4th ed. John Wiley & Sons, 2001.

[27] P. Harpe, E. Cantatore, and A. van Roermund, "A 10b/12b 40 kS/s SAR ADC with data-driven noise reduction achieving up to 10.1b ENOB at 2.2 fJ/conversation-step", *IEEE Journal of Solid-State Circuits*, vol. 48, no. 12, pp. 3011–3018, 2013.

[28] D. Zhang, C. Svensson, and A. Alvandpour, "Power consumption bounds for SAR ADCs", in *20th European Conference on Circuit Theory and Design*, 2011.

[29] B. Chor, O. Goldreich, J. Hastad, J. Freidmann, S. Rudich, and R. Smolensky, "The bit extraction problem or t-resilient functions", *26th IEEE Symposium on Foundations of Computer Science*, pp. 396–407, 1985.

[30] NIST, "Recommendation for random number generation using deterministic random bit generators", *NIST Special Publication*, 2012.

[31] M. Blum, "Independent unbiased coin flips from a correlated biased source - a finite state markov chain", *Combinatorica*, vol. 6, no. 2, pp. 97–108, 1986.

[32] J. P. Uyemura, *Introduction to VLSI Circuits and Systems*. John Wiley & Sons, 2002.

[33] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital CMOS circuits", *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[34] C. Piguet, *Low-Power CMOS Circuits*. CRC Press, 2005.

[35] Silicon Laboratories. (2012). EFM32WG990 Datasheet, [Online]. Available: `www.silabs.com/Support\%20Documents/TechnicalDocs/EFM32WG990.pdf`.

[36] K. Rottmann, *Matematisk Formelsamling*, 11th ed. Spektrum forlag, 2003.

[37] Cadence, *Cadence NC-Verilog simulator tutorial*, 2003.

[38] A. Hasanbegovic, *Tutorial: Digital logic design and verification flow using a standard cell library*, 2013.

Appendix A

Extended Background on Information Theory

In the following an extended background of the in Section 2.1 introduced concepts is presented. Appendix A.1 proves the statements with regards to entropy and the uniform distribution, and Appendix A.2 demonstrates the validity of the statements with respect to statistical (in-)dependencies. Note that the here presented proofs are mainly identical to those presented in [13] and [16].

A.1 Entropy and the Uniform Distribution

In order to prove the validity of Equation 2.5, it is suitable to first show that the upper bound of $\mathrm{H}(X)$ is $\log_2(k)$, and then prove that this bound is reached if and only if the ensemble X is associated with an uniform distribution.

In order to find an upper boundary, *Jensen's inequality* proves to be useful [13, p. 35]. It states that, for some concave function $f(z)$, it holds that,

$$\mathrm{E}[f(z)] \leq f(\mathrm{E}[z]). \tag{A.1}$$

In other words, the expectation of the result of $f(z)$ for some variable z is always less or equal than the result of f using the expectation of z as an input. Further, it holds that, if f is strictly concave and Equation A.1 solves to an equality, then z is a constant. It is worth noticing that Jensen's inequality also holds for convex functions. However, in that case the inequality sign in Equation A.1 is reversed.

By choosing,

$$f(z) = \log_2(z), \tag{A.2}$$

which is a strictly concave function for all z greater than zero, Equation 2.2 can be rewritten as,

$$\mathrm{H}(X) = \sum_{i=0}^{k-1} p_i \cdot f(\frac{1}{p_i}) = \mathrm{E}[f(\frac{1}{p_i})]. \tag{A.3}$$

Inserting Equation A.3 into Equation A.1 and solving the right hand side with regards to Equation A.2 yields,

$$\mathrm{H}(X) \leq f(\mathrm{E}[\frac{1}{p_i}]) = f(\sum_{i=0}^{k-1} p_i \cdot \frac{1}{p_i}) = \log_2(k), \tag{A.4}$$

where k is the number of elements in $\mathcal{A}_X$.

Having found the desired upper bound of the entropy, it remains to show that Equation A.4 solves with equality if and only if X is associated with an uniform distribution. To do so, Jensen's inequaltiy should be considered once more. It states, that since f is strictly concave, p_i must be a constant if Equation A.4 solves to an equation. By the definition of $\mathcal{P}_X$ given in Section 2.1, it follows that if p_i is constant for all i, then p_i must be equal to $\frac{1}{k}$, which means that $\mathcal{P}_X$ describes an uniform distribution. Thus, if Equation A.4 solves to an equality, X follows an uniform distribution. Showing that this implication also holds the other way around can be easily done by solving Equation 2.2 with $p_i = \frac{1}{k}$, as done during the discussion of Equation 2.5 in Section 2.1. As a result, $\mathrm{H}(X)$ reaches it maximum $\mathrm{H}(X) = \log_2(k)$ if and only if X os associoated with an uniform distribution.

A.2 Entropy and Statistical Dependencies

In order to prove the in Section 2.1 presented statements regarding the joint and conditional entropy, it is suitable to adapt the in that section used approach and first focus on the 2-bit vector $vector x_0, x_1$ (Appendix A.2.1) and then expend the concept to a data stream $\overrightarrow{x} = \langle x_0, ..., x_{n-1} \rangle$ of arbitrary length n (Appendix A.2.2). Note that this section is mainly concerned with the sums of the probabilities of all possible outcomes of the ensemble X, and less concerned with a specific outcome a_i. Thus in order to increase the readability of the equations, the term a_i is dropped whenever not explicitly necessary and the short hand notation $\sum_{x_0}$ is used to denote a summation over all possible values of x_0. For example, $\sum_{x_0} \Pr(x_0|x_1 = a_j) = \sum_{i=0}^{k-1} \Pr(x_0 = a_i|x_1 = a_j)$.

A.2.1 Conditional entropy of a 2-bit vector

To show that the joint entropy, $\mathrm{H}(X_0, X_1)$ (Equation 2.7), can be expressed as the sum of $\mathrm{H}(X_0)$ and the conditional entropy $\mathrm{H}(X_1|X_0)$ (Equation 2.11), Equation 2.7 can be rewritten as,

$$
\begin{aligned}
\mathrm{H}(X_0, X_1) &\equiv \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} \Pr(x_0 = a_i, x_1 = a_j) \cdot \log_2(\frac{1}{\Pr(x_0 = a_i, x_1 = a_j)}) \\
&= \sum_{x_0}\sum_{x_1} \Pr(x_1|x_0) \cdot \Pr(x_0) \cdot \log_2(\frac{1}{\Pr(x_0)}) \\
&\quad + \sum_{x_0}\sum_{x_1} \Pr(x_1|x_0) \cdot \Pr(x_0) \cdot \log_2(\frac{1}{\Pr(x_1|x_0)}) \\
&= \sum_{x_0} \Big[\Pr(x_0) \cdot \log_2(\frac{1}{\Pr(x_0)}) \cdot \sum_{x_1} \Pr(x_1|x_0)\Big] \\
&\quad + \sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \log_2(\frac{1}{\Pr(x_1|x_0)}) \\
&= \mathrm{H}(X_0) + \mathrm{H}(X_1|X_0),
\end{aligned}
\tag{A.5}
$$

where Equation 2.8 has been used alongside with the fact that,

$$
\sum_{x_1} \Pr(x_1|x_0 = a_i) = 1, \tag{A.6}
$$

which follows from the definition of probability [15, p. 53].

To show that $\mathrm{H}(X_1|X_0)$ is upper bounded by $\mathrm{H}(X_1)$ the relation between $\mathrm{H}(X_1|X_0)$ and $\mathrm{H}(X_1)$ has to be considered. By means of Equation A.6, $\mathrm{H}(X_1)$ can be rewritten as,

$$
\begin{aligned}
\mathrm{H}(X_1) &= \sum_{x_1} \Pr(x_1) \cdot \log_2(\frac{1}{\Pr(x_1)}) \\
&= \sum_{x_1} \Big[\Pr(x_1) \cdot \log_2(\frac{1}{\Pr(x_1)}) \cdot \sum_{x_0} \Pr(x_0|x_1)\Big] \\
&= \sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \log_2(\frac{1}{\Pr(x_1)}).
\end{aligned}
\tag{A.7}
$$

Using Equation A.7 the difference between $\mathrm{H}(X_1)$ and $\mathrm{H}(X_1|X_0)$ can be expressed as,

$$\begin{aligned}
\mathrm{H}(X_1) - \mathrm{H}(X_1|X_0) &= \sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \log_2(\frac{1}{\Pr(x_1)}) \\
&\quad - \sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \log_2(\frac{1}{\Pr(x_1|x_0)}) \\
&= \sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \log_2\Big(\frac{\Pr(x_1|x_0)}{\Pr(x_1)}\Big) \\
&= \sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \log_2\Big(\frac{\Pr(x_0, x_1)}{\Pr(x_0) \cdot \Pr(x_1)}\Big) \\
&= -\sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \log_2\Big(\frac{\Pr(x_0) \cdot \Pr(x_1)}{\Pr(x_0, x_1)}\Big).
\end{aligned} \tag{A.8}$$

Since the negative logarithmic function in Equation A.8 is a strictly convex function, Jensen's inequality as stated in Equation A.1 (note that the inequality is reversed in this situation since a convex functions is considered), can be used to find a lower bound for Equation A.8,

$$\begin{aligned}
\mathrm{H}(X_1) - \mathrm{H}(X_1|X_0) &\geq -\log_2\Big(\sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \frac{\Pr(x_0) \cdot \Pr(x_1)}{\Pr(x_0, x_1)}\Big) \\
&= -\log_2\Big(\sum_{x_0}\Big[\Pr(x_0) \cdot \sum_{x_1} \Pr(x_1)\Big]\Big) \\
&= -\log_2(1) = 0.
\end{aligned} \tag{A.9}$$

It is easy to see that Equation 2.10 follows directly from Equation A.9.

The next goal is to prove that Equation 2.10 holdes with equality if and only if x_0 and x_1 are independent. Since the negative logarithmic function is strictly convex, one can once more make use of Jensen's inequality as presented in Equation A.1, which states that if Equation 2.10 solves to an equality then,

$$\zeta = \frac{\Pr(x_0 = a_i) \cdot \Pr(x_1 = a_j)}{\Pr(x_0 = a_i, x_1 = a_j)}, \tag{A.10}$$

where ζ is a constant for all possible combinations of a_i and a_j. Using Equation 2.8, Equation A.10 can be rewritten to,

$$\begin{aligned}
\zeta &= \frac{\Pr(x_0 = a_i) \cdot \Pr(x_1 = a_j)}{\Pr(x_1 = a_j|x_0 = a_i) \cdot \Pr(x_0 = a_i)} = \frac{\Pr(x_1 = a_j)}{\Pr(x_1 = a_j|x_0 = a_i)}, \\
&\Rightarrow \zeta \cdot \Pr(x_1 = a_j|x_0 = a_i) = \Pr(x_1 = a_j),
\end{aligned} \tag{A.11}$$

which means that $\Pr(x_1 = a_j|x_0 = a_i)$ is proportional to $\Pr(x_1 = a_j)$. However, this cannot be true for any $\alpha \neq 1$, since both probabilities must add up to one, that is,

$$\begin{aligned}\sum_{x_1} \Pr(x_1) \quad &= \sum_{x_1} \zeta \cdot \Pr(x_1|x_0 = a_i) = \zeta \cdot \sum_{x_1} \Pr(x_1|x_0 = a_i) = \zeta, \\ &\Rightarrow \zeta = 1, \quad \Rightarrow \Pr(x_1 = a_j) = \Pr(x_1 = a_j|x_0 = a_i).\end{aligned} \tag{A.12}$$

It follows from Equation A.12 that if Equation 2.10 solves with equality then x_0 and x_1 are independent. This implication holds also the other way around, which is easily shown by solving Equation 2.11 for the two independent outcomes x_0 and x_1,

$$\begin{aligned}\mathrm{H}(X_1|X_0) \quad &= \sum_{x_0}\sum_{x_1} \Pr(x_0, x_1) \cdot \log_2(\frac{1}{\Pr(x_1|x_0)}) \\ &= \sum_{x_0}\sum_{x_1} \Pr(x_0) \cdot \Pr(x_1) \cdot \log_2(\frac{1}{\Pr(x_1)}) \\ &= \sum_{x_1} \Big[\Pr(x_1) \cdot \log_2(\frac{1}{\Pr(x_1)}) \cdot \sum_{x_0} \Pr(x_0)\Big] \\ &= \mathrm{H}(X_1),\end{aligned} \tag{A.13}$$

where Equation 2.9 has been used.

A.2.2 Conditional entropy of an n-bit vector

To extend the in Appendix A.2.1 presented concept to an n-bit vector $\overrightarrow{x} = \langle x_0, ..., x_{n-1}\rangle$, it is reasonable to first define the joint probability of the n ensembles $X_0, X_1, ..., X_{n-1}$ as,

$$\mathrm{H}(X_0, X_1, ...X_{n-1}) \equiv \sum_{x_0}\sum_{x_1}...\sum_{x_{n-1}} \Pr(x_0, x_1, ...x_{n-1}) \cdot \log_2(\frac{1}{\Pr(x_0, x_1, ..., x_{n-1})}). \tag{A.14}$$

Rearranging the right hand side of Equation A.14 in the same manner as it has been done in Equation A.5 yields [16, p. 22],

$$\mathrm{H}(X_0, X_1, ..., X_{n-1}) = \mathrm{H}(X_0) + \sum_{i=1}^{n-1} \mathrm{H}(X_i|X_0, ..., X_{i-1}), \tag{A.15}$$

where $\mathrm{H}(X_i|X_0, ..., X_{i-1})$ is the extended version of the conditional entropy [16, p. 23],

$$\mathrm{H}(X_i|X_0, ..., X_{i-1}) = \sum_{x_0}...\sum_{x_i} \Pr(x_0, ..., x_i) \cdot \log_2(\frac{1}{\Pr(x_i|x_0, ..., x_{i-1})}). \tag{A.16}$$

The notation $\Pr(x_i|x_0, ..., x_{i-1})$ denotes the conditional probability of x_i given $x_0, x_1, ..., x_{i-1}$.

Now, using the same approach as for Equation A.8 and Equation A.9 yields,

$$
\begin{aligned}
\mathrm{H}(X_i) - \mathrm{H}(X_i|X_0,...,X_{i-1}) &= \sum_{x_0} ... \sum_{x_i} \Pr(x_0,...,x_i) \cdot \log_2(\frac{1}{\Pr(x_i)}) \\
&\quad - \sum_{x_0} ... \sum_{x_i} \Pr(x_0,...,x_i) \cdot \log_2(\frac{1}{\Pr(x_i|x_0,...,x_{i-1})}) \\
&= \sum_{x_0} ... \sum_{x_i} \Pr(x_0,...,x_i) \cdot \log_2\Big(\frac{\Pr(x_i|x_0,...,x_{i-1})}{\Pr(x_i)}\Big) \\
&= \sum_{x_0} ... \sum_{x_i} \Pr(x_0,...,x_i) \cdot \log_2\Big(\frac{\Pr(x_0,...,x_i)}{\Pr(x_i)\cdot\Pr(x_0,...,x_{i-1})}\Big) \\
&\geq -\log_2\Big(\sum_{x_0} .. \sum_{x_i} \Pr(x_0,..,x_i) \cdot \frac{\Pr(x_i)\cdot\Pr(x_0,..,x_{i-1})}{\Pr(x_0,..,x_i)}\Big) \\
&= -\log_2\Big(\sum_{x_i} \Pr(x_i) \cdot \sum_{x_0} ... \sum_{x_{i-1}} \Pr(x_0,...,x_{i-1})\Big) \\
&= -\log_2(1) \\
&= 0,
\end{aligned}
\tag{A.17}
$$

where it has been used that,

$$
\begin{aligned}
\Pr(x_i|x_0,...,x_{i-1}) &= \frac{\Pr(x_0,...,x_i)}{\Pr(x_0)\cdot\Pr(x_1|x_0)\cdot ... \cdot \Pr(x_{i-1}|x_0,...,x_{i-2})} \\
&= \frac{\Pr(x_0,...,x_i)}{\Pr(x_0,...,x_{i-1})}.
\end{aligned}
\tag{A.18}
$$

From Equation A.17 it follows that,

$$
\mathrm{H}(X_i) \geq \mathrm{H}(X_i|X_0,...,X_{i-1}), \tag{A.19}
$$

which is the equivalent to Equation 2.12 for an arbitrary number of variables. By repeating the procedure presented in Equation A.10 till Equation A.13 with $\frac{\Pr(x_i)\cdot\Pr(x_0,...,x_{i-1})}{\Pr(x_0,...,x_i)}$, it is possible to show that Equation A.19 holds with equality if and only if,

$$
\Pr(x_i = a_{j_i}) = \Pr(x_i = a_{j_i}|x_0 = a_{j_0},...,x_{i-1} = a_{j_{i-1}}), \tag{A.20}
$$

which means that x_i is independent of all previous values $x_0,...,x_{i-1}$. Equation 2.16 follows from combining Equation A.15 and Equation A.19.

Appendix B

Setup of an ADC as an Entropy Source

The following presents a C-code routine, used to set up the ADC of an *EFM32 Wonder Gecko* to function as an entropy source [5]. The *EFM32* exports the data via an serial communication protocol. A script that can be used to receive the data via MATLAB is included in the zip-file which accompanies this report.

Even though the presented code has not been directly created by *Silicon Laboratories*, it makes exhaustive use of design examples and includes libraries published by the company.

Main routine

```
//------------------------------------------------------------------------
// Include librarys
//------------------------------------------------------------------------

#include "VirtualSerial.h"
#include "em_cmu.h"

//------------------------------------------------------------------------
// Definitions and declarations
//------------------------------------------------------------------------

// CMU status-reg: HFXO is enabled and ready
#define HFXO_EN_AND_RDY             (CMU_STATUS_HFXORDY |
    CMU_STATUS_HFXOENS)

// Clock frequencies
#define ADC_CLK_FREQ     12000000  // in Hz
#define HFPER_CLK_FREQ   48000000  // in Hz

// ADC acquisition time
#define ADC_ACQUISITION_TIME  _ADC_SINGLECTRL_AT_1CYCLE

// ADC states for interrupt handling
#define ADC_SAMPLE_WAIT 0
#define ADC_SAMPLE_RDY  1

// Size of sample buffer in RAM
```

```
#define SAMPLE_BUFFER_SIZE 8192 // in byte

// Number of requested buffer
#define NUM_REQ_BUFFER 65536

// ADC status variable for interrupt handling
uint8_t adcSampleStatus;

// Function declarations
void switch2HFXO(void);
void initADC(void);
void exportSampleBuffer(uint8_t [], int);
void waitForGo(void);

//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// switch2HFXO:
// + Enables HFXO
// + Switches HFCLK-source from HFRCO to HFXO
// + Disables HFXO
//---------------------------------------------------------------------------

void switch2HFXO(void){

        // Variable for register access
        uint32_t regWord;

        // Set HFXO timeout to 16k cycles
        regWord = CMU->CTRL;
        regWord &= ~_CMU_CTRL_HFXOTIMEOUT_MASK;
        regWord |= CMU_CTRL_HFXOTIMEOUT_16KCYCLES;
        CMU->CTRL = regWord;

        // Enable HFXO
        CMU->OSCENCMD = CMU_OSCENCMD_HFXOEN;

        // Wait for HFXO to be ready
        regWord = CMU->STATUS;
        while(!(regWord & HFXO_EN_AND_RDY)){regWord = CMU->STATUS;}

        // Switch HFCLK to HFXO
        CMU->CMD = CMU_CMD_HFCLKSEL_HFXO;

        // Disable HFRCO
        CMU->OSCENCMD = CMU_OSCENCMD_HFRCODIS;
}

//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// initADC:
// + Enables ADC and sets mode:
//              - WarmUpMode: KeepWarm
//              - ADC clock freq: ADC_CLK_FREQ (see "Definitions")
//              - Input Signal: Vref / 2
```

```
//              - Acquisition time: ADC_ACQUISITION_TIME (see "
    Definitions")
// + Enables interrupt for single conversion
//-----------------------------------------------------------------

void initADC(void){

        // Variable for register access
        uint32_t regWord;

        // Activate HFPER and ADC clock
        CMU_ClockEnable(cmuClock_HFPER, true);
        CMU_ClockEnable(cmuClock_ADC0, true);

        // Prescale the ADC clock & set keep-warm-mode
        regWord = ADC0->CTRL;
        regWord &= ~(_ADC_CTRL_WARMUPMODE_MASK | _ADC_CTRL_PRESC_MASK);
        regWord |= ((HFPER_CLK_FREQ / ADC_CLK_FREQ) - 1) <<
            _ADC_CTRL_PRESC_SHIFT;
        regWord |= ADC_CTRL_WARMUPMODE_KEEPADCWARM;
        ADC0->CTRL = regWord;

        // Set up Single Sample options
        regWord = ADC0->SINGLECTRL;
        regWord &= ~(_ADC_SINGLECTRL_INPUTSEL_MASK |
            _ADC_SINGLECTRL_AT_MASK);
        regWord |= ADC_SINGLECTRL_INPUTSEL_VREFDIV2;
        regWord |= (ADC_ACQUISITION_TIME << _ADC_SINGLECTRL_AT_SHIFT);
        ADC0->SINGLECTRL = regWord;

        // Enable interrupt for single conversion complete
        ADC0->IFC = ADC_IFC_SINGLE;
        ADC0->IEN = ADC_IEN_SINGLE;
        NVIC_EnableIRQ(ADC0_IRQn);
}

//-----------------------------------------------------------------

//-----------------------------------------------------------------
// ADC0_IRQHandler (Interrupt routine for single conversion interrupt):
// + Sets adcSampleStatus to ADC_SAMPLE_RDY
//-----------------------------------------------------------------

void ADC0_IRQHandler(void){

        // Clear interrupt flag
        ADC0->IFC = ADC_IFC_SINGLE;

        // Disable interrupt
        ADC0->IEN = _ADC_IEN_RESETVALUE;

        // Set adcSampleStatus to ready
        adcSampleStatus = ADC_SAMPLE_RDY;

}
```

```
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
// main routine
//-----------------------------------------------------------------------

int main(void)
{

        // Buffer form ADC samples
        uint8_t sampleBuffer[SAMPLE_BUFFER_SIZE];

        // Counter for ADC samples
        uint32_t sampleCounter = 0;

        // Counter for exported buffers
        uint32_t bufferCount = 0;

        // Delete me
        uint8_t testCount = 0;
        uint32_t dummy = 0;
        uint8_t dummyBuffer[SAMPLE_BUFFER_SIZE];

        // Chip errata
        CHIP_Init();

        // Switch clock to HFXO
        switch2HFXO();

        // Initialize ADC
        initADC();

        // Set up hardware for PC communication
        SetupHardware();

        // Wait for start signal from receiver
        waitForGo();

        // Set adcSampleStatus to wait
        adcSampleStatus = ADC_SAMPLE_WAIT;

        // Trigger ADC single conversion
        ADC0->CMD = ADC_CMD_SINGLESTART;

        while(1){

          // Wait for sample to be ready
          while(adcSampleStatus == ADC_SAMPLE_WAIT){}

            // Read sample from ADC
            sampleBuffer[sampleCounter] = ADC0->SINGLEDATA;


            // Export buffer if full
            if(sampleCounter == SAMPLE_BUFFER_SIZE-1){
```

```
                // Send sample buffer to receiver
                exportSampleBuffer(sampleBuffer, SAMPLE_BUFFER_SIZE);


                // Clear sampleCounter
                sampleCounter = 0;

                // Increase bufferCounter
                bufferCount++;

              } // if: sampleCount == SAMPLE_BUFFER_SIZE-1

              // Otherwise increase sampleCounter
              else sampleCounter++;

              // Trigger ADC for single conversion
              if(bufferCount != NUM_REQ_BUFFER) ADC0->CMD =
                  ADC_CMD_SINGLESTART;

              // Set ADC status to WAIT
              adcSampleStatus = ADC_SAMPLE_WAIT;

              // Re-enable ADC interrupt
              ADC0->IEN = ADC_IEN_SINGLE;
          }

}

//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
// exportSampleBuffer: Exports buffer to receiver via serial
    communication
//-----------------------------------------------------------------------

void exportSampleBuffer(uint8_t sampleBuffer[], int sampleBufferSize){

          int sample_i=0;
          int sent = 0;

          uint8_t byte;

          char TX_RDY = 'X';
          char RX_RDY = 'Q';

          // Signal to receiver that data can be exported
          Endpoint_SelectEndpoint(CDC_TX_EPADDR);
          Endpoint_Write_8(TX_RDY);
          Endpoint_ClearIN();
          while (!Endpoint_IsINReady());

          while(!sent){

            // Wait for reply from receiver
            Endpoint_SelectEndpoint(CDC_RX_EPADDR);
            if(Endpoint_IsOUTReceived()){
```

```
            // Check if receiver is ready
            if(Endpoint_Read_8() == RX_RDY){

                // Switch to TX mode
                Endpoint_ClearOUT();

                // Loop through sample buffer
                for(sample_i = 0; sample_i < sampleBufferSize; sample_i
                    ++){

                    // Read sample from buffer and add LSB to byte
                    byte = byte + (sampleBuffer[sample_i] % 2);
                    byte = byte << 1;

                    // If byte is full, write it to TX terminal
                    if((sample_i % 8) == 7){
                        Endpoint_SelectEndpoint(CDC_TX_EPADDR);
                        Endpoint_Write_8(byte);
                        Endpoint_ClearIN();
                        while (!Endpoint_IsINReady());

                        // Reset byte
                        byte = 0;

                    } // end if: byte is full

                } // end for: sample_i

                // Set flag if buffer is exported
                sent = 1;

            } // end if: RX_RDY

            else Endpoint_ClearOUT();

          } // end if: package received

        } // end while: !sent

} // end void: exportSampleBuffer

//------------------------------------------------------------------

//------------------------------------------------------------------
// waitForGo: Waits for start signal from receiver
//------------------------------------------------------------------

void waitForGo(void){

        int go = 0;

        char reply = 'A';

        Endpoint_SelectEndpoint(CDC_RX_EPADDR);
```

```
        while(!go){

          if(Endpoint_IsOUTReceived()){

            Endpoint_ClearOUT();

            Endpoint_SelectEndpoint(CDC_TX_EPADDR);
            Endpoint_Write_8(reply);
            Endpoint_ClearIN();
            while (!Endpoint_IsINReady());

            go = 1;

          }
        }

}
//-------------------------------------------------------------------------
```

Appendix C

Statistical Testing

C.1 The von Neumann Condition Test

```
%% ------------------------------------------------------------------
% vNeumannTest.m
% Script to run test for v.Neumann conditions

%% ------------------------------------------------------------------
% Close previous figures
close all;

%% ------------------------------------------------------------------
% Defines

SOURCE = 'C:\Users\Conrad\Documents\MATLAB\TRNG\adc_data\data.
    adc_010415_500Mb'; % Address of the source file
TARGET = [SOURCE, '_vNeumman_test_res']; % Target address for results

NUM_OF_SEQUENCES = 64; % Number of test sequences
SEQUENCE_LENGTH  = 1048576; % Bits per test sequence
BIT_PER_LINE     = 8; % Number of bits per line in the source file
NUM_OF_LINES     = SEQUENCE_LENGTH/BIT_PER_LINE; % Number of lines per
    sequence in source file

ENABLE_2_LEVEL_TEST = false; % Enables second-level test
NUM_OF_P_GROUPS     = 10; % Number of intervals used to group p-values
    during second-level test

%% ------------------------------------------------------------------
% Counter for v.Neumann pairs

num_vN_pairs = zeros(NUM_OF_SEQUENCES, 1); % Vector for number of
    vNeumann-pairs
num_01_pairs = zeros(NUM_OF_SEQUENCES, 1); % Vector for number of 01-
    pairs
num_10_pairs = zeros(NUM_OF_SEQUENCES, 1); % Vector for number of 10-
    pairs

%% ------------------------------------------------------------------
% Fetch number of vNeumann-pairs from source file
```

```
% Print info msg to terminal
disp('Fetching data from source file');

% Open source file
source_id = fopen(SOURCE);

% Read each sequence
for seq_i = 1:NUM_OF_SEQUENCES

    % Print info msg to terminal
    disp(['Reading sequence: ',num2str(seq_i)]);

    % Read each line of the sequence & count vNeumann pairs
    for line_i = 1:(NUM_OF_LINES)

        % Read line from source
        line = fgetl(source_id);

        % Convert ASCII to binary
        line = line - 48;

        % Check each bit pair in line
        for pair_i = 1:2:BIT_PER_LINE

            % Read bit pairs from line
            bit_1 = line(pair_i);
            bit_2 = line(pair_i+1);

            % Evaluate bit pair
            if((bit_1 == 0) && (bit_2 == 1))

                % Update vNeumann bit pair counter
                num_vN_pairs(seq_i) = num_vN_pairs(seq_i) + 1;
                num_01_pairs(seq_i) = num_01_pairs(seq_i) + 1;

            elseif ((bit_1 == 1) && (bit_2 == 0))

                % Update vNeumann bit pair counter
                num_vN_pairs(seq_i) = num_vN_pairs(seq_i) + 1;
                num_10_pairs(seq_i) = num_10_pairs(seq_i) + 1;

            end % if: vN-pair

        end % for: pair_i

    end % for: line_i

end % for: seq_i

%close source file
fclose(source_id);

%% ------------------------------------------------------------------
% Run first-level vNeumann-hypothesis test

% Print info msg to terminal
```

```
disp('Running first-level v.Neumann hypothesis test');

% Init. empty p-value vector
p_values = zeros(NUM_OF_SEQUENCES,1);

% Check hypothesis for each sequence
for seq_i = 1:NUM_OF_SEQUENCES

    % Mean and stand. deviation of the norm. pdf. approximation
    mean_norm = 0.5*num_vN_pairs(seq_i);
    std_norm = sqrt((0.5^2)*num_vN_pairs(seq_i));

    % Set observed test statistic for the sequence
    obs_test_stat = num_01_pairs(seq_i);

    % Compute p-value for the sequence
    if(obs_test_stat <= mean_norm)
        p_values(seq_i) = 2*normcdf(obs_test_stat,mean_norm,std_norm);
    else
        p_values(seq_i) = 2*(1-normcdf(obs_test_stat,mean_norm,std_norm
            ));
    end

end

% Plot p-value histogram
figure(1)
title('P-values for the vNeumann-hypothesis test');
h = histogram(p_values,NUM_OF_P_GROUPS);
xlabel('p-values ranges');
ylabel('Number of p-values');

%% ------------------------------------------------------------------
% Run second-level vNeumann-hypothesis test (chi-squared
%  goodness-of-fit)
% NOTE: Test must be enabled by setting ENABLE_2_LEVEL_TEST

if(ENABLE_2_LEVEL_TEST == true)

    % Compute degrees of freedom
    degrees_of_freedom = NUM_OF_P_GROUPS - 1;

    % Set expected and observed frequencies
    exp_p_freq = NUM_OF_SEQUENCES/NUM_OF_P_GROUPS;
    obs_p_freq = h.Values;

    % Compute observed test statistic
    obs_test_stat = 0;
    for group = 1:NUM_OF_P_GROUPS
        obs_test_stat = obs_test_stat + (((obs_p_freq(group) -
            exp_p_freq)^2)/exp_p_freq);
    end

    obs_test_stat;

    % Compute p-value of the second level test
```

```
    sec_lvl_p_value = chi2cdf(obs_test_stat,degrees_of_freedom,'upper')
        ;

end % if: ENABLE_2_LEVEL_TEST

%% ---------------------------------------------------------------------
% Print test results to target file

% Open target file
target_id = fopen(TARGET,'w');

% Print table header
fprintf(target_id,'Sequence \t Num. vN pairs \t Num. 01 pairs \t p-
    value \n');
fprintf('\n')

% Print results for each sequence
for seq_i = 1:NUM_OF_SEQUENCES
    fprintf(target_id,'%-8d \t %-13d \t %-13d \t %-1.4f \n',seq_i,
        num_vN_pairs(seq_i),num_01_pairs(seq_i),p_values(seq_i));
end

% Print p-value of second-level test
if(ENABLE_2_LEVEL_TEST == true)
    fprintf(target_id,'\n\n');
    fprintf(target_id,'Second-level p-value: %1.4f\n',sec_lvl_p_value);
end

% Close target file
fclose(target_id);

% Clear line variable
clear line;
```

C.2 Results of Statistical Testing

C.2.1 Results for the Entropy Source

Table C.1: Results of the Frequency test for the source data

Sequence	s_o	Percentage of 0's	p-value	Sequence	s_o	Percentage of 0's	p-value
1	427600	29.6105%	0.00000	33	485274	26.8604%	0.00000
2	403234	30.7723%	0.00000	34	493022	26.4909%	0.00000
3	399974	30.9277%	0.00000	35	475860	27.3092%	0.00000
4	403494	30.7599%	0.00000	36	455432	28.2833%	0.00000
5	407322	30.5774%	0.00000	37	478238	27.1959%	0.00000
6	409438	30.4764%	0.00000	38	464754	27.8388%	0.00000
7	411772	30.3652%	0.00000	39	447858	28.6444%	0.00000
8	421722	29.8908%	0.00000	40	433690	29.3200%	0.00000
9	419680	29.9881%	0.00000	41	452690	28.4141%	0.00000
10	420960	29.9271%	0.00000	42	466270	27.7665%	0.00000
11	423778	29.7927%	0.00000	43	454218	28.3412%	0.00000
12	441366	28.9540%	0.00000	44	455060	28.3011%	0.00000
13	429766	29.5071%	0.00000	45	470680	27.5562%	0.00000
14	434140	29.2986%	0.00000	46	445466	28.7585%	0.00000
15	452494	28.4234%	0.00000	47	438754	29.0786%	0.00000
16	470864	27.5475%	0.00000	48	450494	28.5188%	0.00000
17	456220	28.2458%	0.00000	49	467046	27.7295%	0.00000
18	459254	28.1011%	0.00000	50	464218	27.8643%	0.00000
19	449836	28.5501%	0.00000	51	471890	27.4985%	0.00000
20	448144	28.6308%	0.00000	52	475522	27.3254%	0.00000
21	443820	28.8370%	0.00000	53	488498	26.7066%	0.00000
22	457614	28.1792%	0.00000	54	452432	28.4263%	0.00000
23	492188	26.5307%	0.00000	55	404426	30.7155%	0.00000
24	466834	27.7396%	0.00000	56	411782	30.3647%	0.00000
25	441290	28.9577%	0.00000	57	426384	29.6684%	0.00000
26	445334	28.7648%	0.00000	58	424800	29.7439%	0.00000
27	454490	28.3283%	0.00000	59	416994	30.1162%	0.00000
28	459418	28.0933%	0.00000	60	417966	30.0698%	0.00000
29	466398	27.7604%	0.00000	61	411816	30.3631%	0.00000
30	457792	28.1708%	0.00000	62	421494	29.9016%	0.00000
31	475718	27.3160%	0.00000	63	414278	30.2457%	0.00000
32	504578	25.9398%	0.00000	64	422648	29.8466%	0.00000

Table C.2: Results of the von Neumann condition test for the source data

Sequence	o_{vN}	o_{01}	p-value	Sequence	o_{vN}	o_{01}	p-value
1	233120	64051	0.0000	33	219117	54818	0.0000
2	241763	69138	0.0000	34	217755	53581	0.0000
3	243063	69603	0.0000	35	221478	56467	0.0000
4	241571	68576	0.0000	36	226100	59044	0.0000
5	240375	68246	0.0000	37	220231	55563	0.0000
6	239237	67622	0.0000	38	223097	57432	0.0000
7	238142	66815	0.0000	39	227651	60208	0.0000
8	235647	65033	0.0000	40	231865	63315	0.0000
9	235308	65230	0.0000	41	226011	59272	0.0000
10	235656	65027	0.0000	42	223461	57914	0.0000
11	234033	64492	0.0000	43	226495	59614	0.0000
12	229973	62075	0.0000	44	225902	59527	0.0000
13	232681	63351	0.0000	45	222462	57044	0.0000
14	231238	62789	0.0000	46	228317	60876	0.0000
15	226405	59565	0.0000	47	230219	61903	0.0000
16	222142	56662	0.0000	48	226403	60322	0.0000
17	225200	58697	0.0000	49	222793	57068	0.0000
18	224911	58678	0.0000	50	223301	58021	0.0000
19	227134	60196	0.0000	51	221965	56886	0.0000
20	227566	60419	0.0000	52	220609	55727	0.0000
21	228514	60957	0.0000	53	217907	53840	0.0000
22	224771	58919	0.0000	54	226698	59648	0.0000
23	217402	53816	0.0000	55	241601	68840	0.0000
24	223275	57567	0.0000	56	239239	67495	0.0000
25	228881	61469	0.0000	57	234330	64317	0.0000
26	228307	60774	0.0000	58	234792	64967	0.0000
27	226573	59572	0.0000	59	237007	66407	0.0000
28	224537	58208	0.0000	60	237157	66275	0.0000
29	222851	57334	0.0000	61	239084	67462	0.0000
30	225838	59103	0.0000	62	235651	65468	0.0000
31	220635	55989	0.0000	63	238067	66804	0.0000
32	214591	51884	0.0000	64	235466	65145	0.0000

C.2.2 Results for the VNC

Table C.3: Results of the Frequency test for the VNC

Sequence	s_o	Percentage of 0's	p-value	Sequence	s_o	Percentage of 0's	p-value
1	454272	28.3386	0.00000	33	463818	27.8834	0.00000
2	462426	27.9498	0.00000	34	477712	27.2209	0.00000
3	474978	27.3513	0.00000	35	492016	26.5389	0.00000
4	499724	26.1713	0.00000	36	498048	26.2512	0.00000
5	497392	26.2825	0.00000	37	496668	26.3170	0.00000
6	498352	26.2367	0.00000	38	494608	26.4153	0.00000
7	514520	25.4657	0.00000	39	485028	26.8721	0.00000
8	518798	25.2618	0.00000	40	478984	27.1602	0.00000
9	496046	26.3467	0.00000	41	485364	26.8560	0.00000
10	499164	26.1980	0.00000	42	484110	26.9158	0.00000
11	499126	26.1998	0.00000	43	478258	27.1949	0.00000
12	496234	26.3377	0.00000	44	487578	26.7505	0.00000
13	464746	27.8392	0.00000	45	472618	27.4638	0.00000
14	463618	27.8929	0.00000	46	487230	26.7671	0.00000
15	461906	27.9746	0.00000	47	490476	26.6123	0.00000
16	466998	27.7318	0.00000	48	477230	27.2439	0.00000
17	467952	27.6863	0.00000	49	476332	27.2867	0.00000
18	467494	27.7081	0.00000	50	489714	26.6486	0.00000
19	461558	27.9912	0.00000	51	471692	27.5080	0.00000
20	458934	28.1163	0.00000	52	479506	27.1354	0.00000
21	459330	28.0975	0.00000	53	477658	27.2235	0.00000
22	455454	28.2822	0.00000	54	484448	26.8997	0.00000
23	461406	27.9984	0.00000	55	473732	27.4107	0.00000
24	456818	28.2172	0.00000	56	466878	27.7375	0.00000
25	458136	28.1544	0.00000	57	469546	27.6103	0.00000
26	455870	28.2624	0.00000	58	469364	27.6190	0.00000
27	467514	27.7072	0.00000	59	476550	27.2764	0.00000
28	471118	27.5354	0.00000	60	487882	26.7359	0.00000
29	459702	28.0797	0.00000	61	481972	27.0178	0.00000
30	466810	27.7408	0.00000	62	487282	26.7646	0.00000
31	476406	27.2832	0.00000	63	490004	26.6348	0.00000
32	475214	27.3400	0.00000	64	487714	26.7440	0.00000

Table C.4: Results of the von Neumann condition test for the VNC

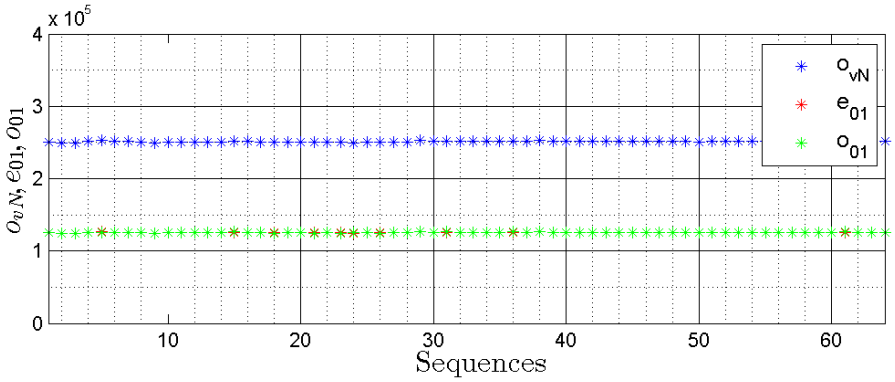
Sequence	o_{vN}	o_{01}	p-value	Sequence	o_{vN}	o_{01}	p-value
1	229770	115177	0.2231	33	227743	114072	0.4008
2	227703	113802	0.8356	34	225448	113043	0.1791
3	225617	112885	0.7474	35	221988	110988	0.9797
4	221210	110469	0.5630	36	220904	110422	0.8984
5	222190	111081	0.9526	37	221208	110156	0.0568
6	220976	110195	0.2125	38	222004	110989	0.9560
7	217002	108057	0.0566	39	223408	111727	0.9225
8	215855	107764	0.4815	40	225066	112219	0.1856
9	221439	110484	0.3169	41	223974	112003	0.9461
10	220668	110371	0.8748	42	223851	112435	0.0313
11	221027	110562	0.8365	43	225365	112656	0.9111
12	220231	109729	0.0995	44	223833	111898	0.9377
13	227473	113826	0.7074	45	226707	113028	0.1715
14	227565	113673	0.6462	46	223589	112188	0.0960
15	228973	114566	0.7397	47	223756	112119	0.3082
16	226445	113555	0.1623	48	226063	113351	0.1790
17	227154	113171	0.0884	49	225636	112630	0.4286
18	226467	113093	0.5549	50	223043	111470	0.8274
19	228073	114336	0.2097	51	226524	113638	0.1141
20	228915	114293	0.4917	52	224959	112387	0.6965
21	228595	114328	0.8985	53	225663	112862	0.8978
22	229279	114739	0.6777	54	224344	112413	0.3089
23	227577	113711	0.7452	55	225890	112961	0.9463
24	229081	114292	0.2991	56	227307	113828	0.4642
25	228732	114110	0.2844	57	226611	112919	0.1044
26	228847	114296	0.5940	58	226954	113317	0.5018
27	226731	113523	0.5083	59	225881	113286	0.1460
28	226599	113391	0.7007	60	224107	112544	0.0382
29	228045	114294	0.2555	61	224712	112484	0.5892
30	226943	113573	0.6700	62	224011	111899	0.6527
31	225771	113091	0.3870	63	222500	111532	0.2318
32	225477	112678	0.7989	64	223907	112072	0.6165

Table C.5: Results of the Frequency test for the second VNC iteration

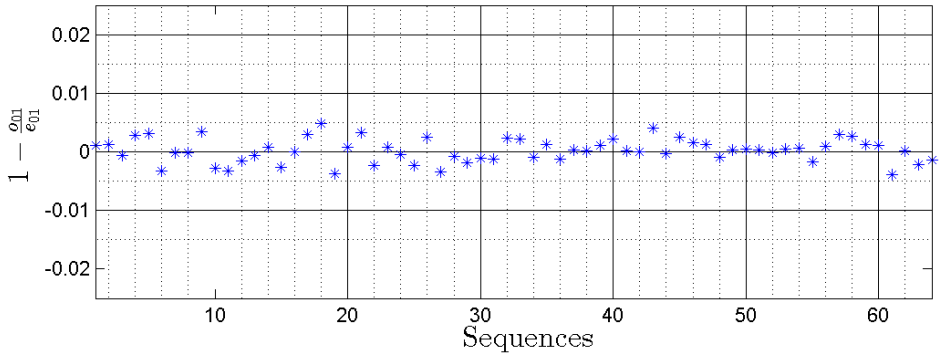
Sequence	s_o	Percentage of 0's	p-value	Sequence	s_o	Percentage of 0's	p-value
1	-572	50.0273	0.57643	33	824	49.9607	0.42100
2	2250	49.8927	0.02800	34	-410	50.0196	0.68886
3	866	49.9587	0.39771	35	-352	50.0168	0.73103
4	314	49.9851	0.75911	36	-1220	50.0581	0.23349
5	-318	50.0151	0.75614	37	240	49.9885	0.81469
6	798	49.9620	0.43580	38	1578	49.9248	0.12331
7	-1130	50.0539	0.26980	39	802	49.9618	0.43350
8	-10	50.0005	0.99220	40	-552	50.0263	0.58984
9	-320	50.0153	0.75466	41	520	49.9752	0.61158
10	-824	50.0393	0.42100	42	508	49.9758	0.61982
11	-746	50.0355	0.46629	43	-456	50.0217	0.65609
12	-876	50.0417	0.39229	44	578	49.9725	0.57244
13	-616	50.0293	0.54746	45	196	49.9906	0.84820
14	-444	50.0212	0.66458	46	1132	49.9460	0.26895
15	-498	50.0237	0.62673	47	14	49.9993	0.98909
16	-96	50.0046	0.92530	48	-942	50.0449	0.35761
17	-122	50.0058	0.90516	49	-192	50.0092	0.85126
18	530	49.9748	0.60475	50	-1752	50.0836	0.08709
19	-2162	50.1031	0.03474	51	-580	50.0277	0.57111
20	682	49.9675	0.50540	52	-248	50.0119	0.80863
21	306	49.9854	0.76507	53	-510	50.0243	0.61845
22	1236	49.9410	0.22742	54	-1338	50.0638	0.19133
23	-1114	50.0531	0.27664	55	1046	49.9501	0.30702
24	-844	50.0403	0.40981	56	-2872	50.1369	0.00503
25	438	49.9791	0.66884	57	-1272	50.0606	0.21416
26	234	49.9888	0.81924	58	656	49.9687	0.52176
27	854	49.9593	0.40429	59	200	49.9904	0.84514
28	-1424	50.0679	0.16433	60	-542	50.0259	0.59660
29	-2228	50.1062	0.02957	61	-160	50.0077	0.87583
30	-88	50.0042	0.93151	62	394	49.9812	0.70041
31	-18	50.0008	0.98597	63	-190	50.0091	0.85280
32	-1142	50.0544	0.26475	64	954	49.9545	0.35152

Table C.6: Results of the von Neumann condition test for the second VNC iteration

Sequence	o_{vN}	o_{01}	p-value	Sequence	o_{vN}	o_{01}	p-value
1	250322	125023	0.5812	33	251996	125720	0.2680
2	249029	124359	0.5331	34	251885	126069	0.6142
3	249599	124878	0.7533	35	252358	126026	0.5424
4	251395	125348	0.1633	36	252226	126270	0.5318
5	252519	125867	0.1183	37	251788	125854	0.8733
6	251129	125978	0.0989	38	252575	126274	0.9572
7	251517	125774	0.9507	39	252013	125870	0.5866
8	250387	125212	0.9411	40	251942	125700	0.2802
9	249512	124334	0.0911	41	251994	125968	0.9080
10	250082	125397	0.1545	42	251912	125963	0.9777
11	250617	125710	0.1087	43	251440	125204	0.0396
12	250562	125481	0.4242	44	251771	125916	0.9032
13	250500	125327	0.7583	45	251762	125565	0.2078
14	250274	125045	0.7130	46	251430	125520	0.4367
15	252155	126414	0.1802	47	251501	125589	0.5195
16	251536	125777	0.9714	48	251249	125745	0.6307
17	250799	125024	0.1337	49	251244	125592	0.9047
18	250831	124813	0.0161	50	250594	125232	0.7951
19	250697	125823	0.0580	51	251402	125661	0.8732
20	250545	125178	0.7057	52	251514	125776	0.9396
21	249997	124588	0.1006	53	251277	125583	0.8248
22	250296	125435	0.2512	54	251403	125626	0.7633
23	249875	124842	0.7024	55	251379	125907	0.3856
24	249798	124965	0.7917	56	250820	125292	0.6375
25	251093	125833	0.2528	57	251346	125300	0.1368
26	250125	124756	0.2203	58	250936	125143	0.1944
27	250871	125869	0.0835	59	251170	125421	0.5128
28	250644	125419	0.6984	60	251641	125677	0.5672
29	252706	126584	0.3581	61	252010	126505	0.0464
30	251762	126021	0.5768	62	250945	125458	0.9538
31	252225	126269	0.5331	63	251237	125899	0.2630
32	252331	125878	0.2523	64	251125	125730	0.5038



(a) Frequencies of von Neumann pairs and the expected and the observed test statistics for the tested sequences



(b) Normalized difference between the observed and expected test statistics for the tested sequences

Figure C.1: Results of the von Neumann condition test for the second VNC iteration

C.2.3 Results for the IHF

Table C.7: Results of the NIST test suite for the IHF with $n_{\mathrm{IHF}} = 64$ and $m_{\mathrm{IHF}} = 1$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	64	0.017912	Passed
BlockFrequency	64	62	0.213309	Passed
CumulativeSums	64	64	0.602458	Passed
Runs	64	63	0.964295	Passed
LongestRun	64	63	0.122325	Passed
Rank	64	64	0.671779	Passed
DFT	64	64	0.468595	Passed
NOT	64	64	0.407091	Passed
OT	64	63	0.350485	Passed
Universal	64	63	0.637119	Passed
Approx.Entropy	64	64	0.637119	Passed
Rand.Excur.	42	40	0.275709	Passed
Rand.Excur.Var.	42	42	0.739918	Passed
Serial	64	64	0.534146	Passed
LinearComplexity	64	64	0.178278	Passed

Table C.8: Results of the NIST test suite for the IHF with $n_{\text{IHF}} = 64$ and $m_{\text{IHF}} = 2$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	62	0.862344	Passed
BlockFrequency	64	64	0.253551	Passed
CumulativeSums	64	64	0.911413	Passed
Runs	64	64	0.568055	Passed
LongestRun	64	63	0.500934	Passed
Rank	64	64	0.671779	Passed
DFT	64	62	0.407091	Passed
NOT	64	63	0.407091	Passed
OT	64	61	0.671779	Passed
Universal	64	63	0.324180	Passed
Approx.Entropy	64	64	0.976060	Passed
Rand.Excur.	39	36	0.003804	Passed
Rand.Excur.Var.	39	39	0.000500	Passed
Serial	64	63	0.772760	Passed
LinearComplexity	64	64	0.350485	Passed

Table C.9: Results of the NIST test suite for the IHF with $n_{\text{IHF}} = 64$ and $m_{\text{IHF}} = 4$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	63	0.017912	Passed
BlockFrequency	64	64	0.162606	Passed
CumulativeSums	64	63	0.275709	Passed
Runs	64	63	0.637119	Passed
LongestRun	64	64	0.706149	Passed
Rank	64	64	0.949602	Passed
DFT	64	63	0.407091	Passed
NOT	64	63	0.025193	Passed
OT	64	63	0.407091	Passed
Universal	64	63	0.637119	Passed
Approx.Entropy	64	64	0.834308	Passed
Rand.Excur.	39	39	0.195163	Passed
Rand.Excur.Var.	39	39	0.195163	Passed
Serial	64	64	0.060239	Passed
LinearComplexity	64	62	0.090936	Passed

Table C.10: Results of the NIST test suite for the IHF with $n_{\mathrm{IHF}} = 64$ and $m_{\mathrm{IHF}} = 8$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	64	0.931952	Passed
BlockFrequency	64	63	0.054199	Passed
CumulativeSums	64	64	0.602458	Passed
Runs	64	63	0.178278	Passed
LongestRun	64	63	0.028181	Passed
Rank	64	63	0.834308	Passed
DFT	64	63	0.568055	Passed
NOT	64	63	0.500934	Passed
OT	64	64	0.134686	Passed
Universal	64	64	0.706149	Passed
Approx.Entropy	64	64	0.671779	Passed
Rand.Excur.	39	37	0.275709	Passed
Rand.Excur.Var.	39	39	0.000001	Failed
Serial	64	62	0.350485	Passed
LinearComplexity	64	64	0.253551	Passed

Table C.11: Results of the NIST test suite for the IHF with $n_{\mathrm{IHF}} = 128$ and $m_{\mathrm{IHF}} = 2$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	64	0.568055	Passed
BlockFrequency	64	63	0.195163	Passed
CumulativeSums	64	63	0.602458	Passed
Runs	64	61	0.500934	Passed
LongestRun	64	64	0.772760	Passed
Rank	64	64	0.195163	Passed
DFT	64	64	0.534146	Passed
NOT	64	63	0.213309	Passed
OT	64	64	0.437274	Passed
Universal	64	63	0.407091	Passed
Approx.Entropy	64	63	0.671779	Passed
Rand.Excur.	41	40	0.330628	Passed
Rand.Excur.Var.	41	41	0.330628	Passed
Serial	64	64	0.299251	Passed
LinearComplexity	64	62	0.007880	Passed

Table C.12: Results of the NIST test suite for the IHF with $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 4$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	64	0.060239	Passed
BlockFrequency	64	64	0.299251	Passed
CumulativeSums	64	64	0.739918	Passed
Runs	64	63	0.671779	Passed
LongestRun	64	64	0.324180	Passed
Rank	64	64	0.324180	Passed
DFT	64	63	0.637119	Passed
NOT	64	63	0.706149	Passed
OT	64	64	0.275709	Passed
Universal	64	63	0.178278	Passed
Approx.Entropy	64	64	0.082177	Passed
Rand.Excur.	45	45	0.004715	Passed
Rand.Excur.Var.	45	45	0.414525	Passed
Serial	64	63	0.568055	Passed
Linear Complexity	64	64	0.931952	Passed

Table C.13: Results of the NIST test suite for the IHF with $n_{\text{IHF}} = 128$ and $m_{\text{IHF}} = 8$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	63	0.253551	Passed
BlockFrequency	64	64	0.637119	Passed
CumulativeSums	64	64	0.015963	Passed
Runs	64	64	0.602458	Passed
LongestRun	64	64	0.025193	Passed
Rank	64	62	0.253551	Passed
DFT	64	64	0.350485	Passed
NOT	64	63	0.500934	Passed
OT	64	63	0.043745	Passed
Universal	64	64	0.911413	Passed
Approx.Entropy	64	63	0.437274	Passed
Rand.Excur.	40	40	0.739918	Passed
Rand.Excur.Var.	40	40	0.392456	Passed
Serial	64	63	0.299251	Passed
LinearComplexity	64	63	0.739918	Passed

Table C.14: Results of the NIST test suite for the IHF with $n_{\mathrm{IHF}} = 128$ and $m_{\mathrm{IHF}} = 16$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	64	0.500934	Passed
BlockFrequency	64	64	0.275709	Passed
CumulativeSums	64	64	0.407091	Passed
Runs	64	64	0.534146	Passed
LongestRun	64	64	0.437274	Passed
Rank	64	64	0.772760	Passed
DFT	64	64	0.706149	Passed
NOT	64	62	0.378138	Passed
OT	64	63	0.082177	Passed
Universal	64	64	0.671779	Passed
Approx.Entropy	64	64	0.500934	Passed
Rand.Excur.	41	41	0.611108	Passed
Rand.Excur.Var.	41	41	0.953553	Passed
Serial	64	63	0.378138	Passed
LinearComplexity	64	64	0.739918	Passed

Table C.15: Results of the NIST test suite for the IHF with $n_{\mathrm{IHF}} = 128$ and $m_{\mathrm{IHF}} = 32$

Test	Number of performed first-level tests	Number of succeeded first-level tests	Second-level p-value	Conclusion
Frequency	64	63	0.031497	Passed
BlockFrequency	64	64	0.100508	Passed
CumulativeSums	64	63	0.195163	Passed
Runs	64	61	0.437274	Passed
LongestRun	64	63	0.350485	Passed
Rank	64	63	0.437274	Passed
DFT	64	64	0.534146	Passed
NOT	64	63	0.834308	Passed
OT	64	64	0.772760	Passed
Universal	64	64	0.407091	Passed
Approx.Entropy	64	62	0.862344	Passed
Rand.Excur.	33	33	0.378138	Passed
Rand.Excur.Var.	33	32	0.888137	Passed
Serial	64	63	0.035174	Passed
LinearComplexity	64	64	0.739918	Passed

Appendix D

SystemVerilog Code of Post-Processing Implementations

In the following the generated SystemVerilog code of the post-processor implementations and the simulation setup is presented. For modules and codes that are essentially equivalent and differ only trivially, only one example is presented. This is for example the case for the different approaches of the VNC.

Note that in the following codes for the VNC, "*all_cg*" is used to describe *Approach 1* and "*some_cg*" describes *Approach 2*. "*syn_cg*" and "*no_cg*" mark *Approach 3* and *Approach 4*, respectively.

D.1 Submodules for General Purposes

General parameter

```
// Time unit factor
// (Use to compute ps/ns/us.. from sec values)
`define US_FACT 1_000_000
`define NS_FACT 1_000_000_000
`define PS_FACT 1_000_000_000_000

// Timescale parameter
`define TIME_UNIT 1ns
`define TIME_FACT `NS_FACT// NOTE: Adjust to TIME_UNIT
`define TIME_PREC 1ps
`define TIMESCALE `timescale `TIME_UNIT/`TIME_PREC

// Clocking parameter
`define SYS_CLK_FREQ 50_000_000 // System clock frequency: 50MHz
`define SYS_CLK_PER  (`TIME_FACT / `SYS_CLK_FREQ) // System clock
    period

// Source parameter
`define SOURCE_SAMP_CYCL 50  // Number of sys_clk cycles per ADC sample
    : 50 (1MHz samplings freq)

// Source input file parameter
`define MAX_BIT_PER_LINE 32
```

```
// Post-processing parameter
`define OUTPUT_MEM_SIZE 32

// I/O Handling
`define NULL 0
`define EOF -1
```

Simple clock gate

```
// Timescale
`TIMESCALE

module clk_gate(clk_in, ctrl, clk_out);

    // Inputs
    input clk_in;
    input ctrl;

    // Output
    output clk_out;

    // Latch
    reg    q_latch;

    always_latch  begin

       if(!clk_in) q_latch = ctrl;

    end // always_latch

    assign clk_out = q_latch & clk_in;

endmodule // clk_gate
```

Simple register

```
// Timescale
`TIMESCALE

module memory_reg(clk,
                  rst_n,
                  data_in,
                  data_out);

    //inputs
    input clk;
    input rst_n;
    input data_in;

    //output
    output data_out;

    //register
    reg    data_reg;
```

```
    always_ff @(posedge clk or negedge rst_n) begin

       if(!rst_n) data_reg = 0;

       else data_reg = data_in;

    end

    assign data_out = data_reg;

endmodule // data_reg
```

Simple register with enable signal

```
// Timescale
`TIMESCALE

module memory_en_reg(clk,
                     rst_n,
                     data_in,
                     en,
                     data_out);

    //inputs
    input clk;
    input rst_n;
    input data_in;
    input en;

    //output
    output data_out;

    //register
    reg    data_reg;


    always_ff @(posedge clk or negedge rst_n) begin

       if(!rst_n) data_reg = 0;

       else begin

         if (en) data_reg = data_in;

         else data_reg = data_reg;

       end

     end

    assign data_out = data_reg;

endmodule // data_reg
```

D.2 VNC Implementation

D.2.1 Definitions for the VNC

VNC parameter

```
// VNC Control States
`define VNC_CTRL_STATUS_IDLE     0
`define VNC_CTRL_STATUS_OPERATE  1

// Clock gate parameter
`define REG_PER_CG 4
`define NUM_OUTPUT_CG (`OUTPUT_MEM_SIZE / `REG_PER_CG)
```

D.2.2 *vnc.sv*

The VNC top-module

```
//------------------------------------------------------------------------------
// VNC_ALL_CG.SV
// Top module of the VNC. Exhaustive clock gate use
//------------------------------------------------------------------------------

// Timescale
`TIMESCALE

module vnc_all_cg(sys_clk,
                  rst_n,
                  start,
                  data_in,
                  samp_flg,

                  samp_req,
                  data_out,
                  data_valid_flg);

   //---------------------------------------------------------------------------
   // Define module variables

   // Inputs
   input sys_clk;
   input rst_n;
   input start;
   input data_in;
   input samp_flg;

   // Outputs
   output                          samp_req;
   output [`OUTPUT_MEM_SIZE-1:0]   data_out;
   output                          data_valid_flg;

   // Intern wires
   wire                                operate_ctrl;
   wire                                qual_flg;
```

```
    wire                                    qual_data;
    wire [$clog2(`OUTPUT_MEM_SIZE)-1:0] mem_count;

    //------------------------------------------------------------------
    // Connect sub modules

    vnc_ctrl_all_cg VNC_CTRL_ALL_CG(.sys_clk(sys_clk),
                                     .rst_n(rst_n),
                                     .start(start),
                                     .qual_flg(qual_flg),
                                     .mem_count(mem_count),
                                     .samp_req(samp_req),
                                     .data_valid_flg(data_valid_flg),
                                     .operate_ctrl(operate_ctrl));

    vnc_input_if_all_cg VNC_INPUT_IF_ALL_CG(.sys_clk(sys_clk),
                                            .rst_n(rst_n),
                                            .data_in(data_in),
                                            .samp_flg(samp_flg),
                                            .operate_ctrl(operate_ctrl),
                                            .qual_flg(qual_flg),
                                            .qual_data(qual_data));

    vnc_output_memory_all_cg VNC_OUTPUT_MEMORY_ALL_CG(.sys_clk(sys_clk),
                                                       .rst_n(rst_n),
                                                       .qual_flg(qual_flg
                                                           ),
                                                       .qual_data(
                                                           qual_data),
                                                       .operate_ctrl(
                                                           operate_ctrl),
                                                       .data_out(data_out
                                                           ),
                                                       .mem_count(
                                                           mem_count));

endmodule // vnc_all_cg
```

D.2.3 *vnc_ctrl.sv*

vnc_ctrl* for *Approach 1

```
//------------------------------------------------------------------
// VNC_CTRL_ALL_CG.SV
// VNC control sub-module. Exhaustive clock gate use
//------------------------------------------------------------------

// Timescale
`TIMESCALE

module vnc_ctrl_all_cg(sys_clk,
                       rst_n,
                       start,
                       qual_flg,
                       mem_count,
                       samp_req,
```

```
                    data_valid_flg,
                    operate_ctrl);

//------------------------------------------------------------------
// Define module variables

// Inputs
input                                    start;
input                                    qual_flg;
input [$clog2(`OUTPUT_MEM_SIZE)-1:0] mem_count;
input                                    sys_clk;
input                                    rst_n;

// Outputs
output samp_req;
output operate_ctrl;
output data_valid_flg;

// Intern wires
wire ctrl_status_reg_clk;
wire ctrl_status_reg_clk_en;
wire nx_ctrl_status;
wire operate_done;

// Intern register
reg pr_ctrl_status;
reg data_valid_flg_reg;

//------------------------------------------------------------------
// Assign input variables

assign operate_done = (pr_ctrl_status == `VNC_CTRL_STATUS_OPERATE) &
    (mem_count == `OUTPUT_MEM_SIZE-1) & qual_flg;

//------------------------------------------------------------------
// Clock gate for intern register clock

assign ctrl_status_reg_clk_en = operate_done | (pr_ctrl_status ==
   `VNC_CTRL_STATUS_IDLE);

clk_gate ctrl_status_reg_clk_gate(.clk_in(sys_clk),
                                  .ctrl(ctrl_status_reg_clk_en),
                                  .clk_out(ctrl_status_reg_clk));

//------------------------------------------------------------------
// Next state logic and register for control status

assign nx_ctrl_status = ((pr_ctrl_status == `VNC_CTRL_STATUS_IDLE) &
    start) ? `VNC_CTRL_STATUS_OPERATE : `VNC_CTRL_STATUS_IDLE;

always_ff @(negedge rst_n or posedge ctrl_status_reg_clk) begin

   if(!rst_n) pr_ctrl_status = 0;

   else pr_ctrl_status = nx_ctrl_status;
```

```
    end

    //-----------------------------------------------------------------
    // Register for data_valid_flg

    always_ff @(negedge rst_n or posedge ctrl_status_reg_clk) begin

        if(!rst_n) data_valid_flg_reg = 0;

        else data_valid_flg_reg = operate_done;

    end

    assign data_valid_flg = data_valid_flg_reg;

    //-----------------------------------------------------------------
    // Assign output variables

    assign samp_req = (pr_ctrl_status == `VNC_CTRL_STATUS_OPERATE) ? 1 :
        0;
    assign operate_ctrl = (pr_ctrl_status == `VNC_CTRL_STATUS_OPERATE) ?
        1 : 0;

endmodule // vnc_ctrl_all_cg
```

vnc_ctrl* for *Approach 2*, *Approach 3* and *Approach 4

```
//-----------------------------------------------------------------
// VNC_CTRL_NO_CG.SV
// VNC control sub-module. No clock gates
//-----------------------------------------------------------------

// Timescale
`TIMESCALE

module vnc_ctrl_no_cg(sys_clk,
                      rst_n,
                      start,
                      qual_flg,
                      mem_count,
                      samp_req,
                      data_valid_flg,
                      operate_ctrl);

    //-----------------------------------------------------------------
    // Define module variables

    // Inputs
    input                                  start;
    input                                  qual_flg;
    input  [$clog2(`OUTPUT_MEM_SIZE)-1:0]  mem_count;
    input                                  sys_clk;
    input                                  rst_n;

    // Outputs
    output samp_req;
```

```
output operate_ctrl;
output data_valid_flg;

// Intern wires
wire   operate_done;

// Intern register
reg pr_ctrl_status;
reg data_valid_flg_reg;
reg nx_ctrl_status;

//----------------------------------------------------------------------
// Assign input variables

assign operate_done = (pr_ctrl_status == `VNC_CTRL_STATUS_OPERATE) &
    (mem_count == `OUTPUT_MEM_SIZE-1) & qual_flg;

//----------------------------------------------------------------------
// Next state logic for CTRL_STATUS register

always_comb begin

   case (pr_ctrl_status)

     // Switch to OPERATE if start signal is received in IDLE
     `VNC_CTRL_STATUS_IDLE: nx_ctrl_status = (start) ?
         `VNC_CTRL_STATUS_OPERATE : `VNC_CTRL_STATUS_IDLE;

     // Switch to IDLE if operate_done is set in OPERATE
     `VNC_CTRL_STATUS_OPERATE: nx_ctrl_status = (operate_done) ?
         `VNC_CTRL_STATUS_IDLE : `VNC_CTRL_STATUS_OPERATE;

     // Default (not reachable)
     default : nx_ctrl_status = `VNC_CTRL_STATUS_IDLE;

   endcase // case (pr_ctrl_status)

end // always_comb begin

//----------------------------------------------------------------------
// CTRL_STATUS register

always_ff @(negedge rst_n or posedge sys_clk) begin

   if(!rst_n) pr_ctrl_status = 0;

   else pr_ctrl_status = nx_ctrl_status;

end

//----------------------------------------------------------------------
// data_valid_flg register

always_ff @(negedge rst_n or posedge sys_clk) begin

   if(!rst_n) data_valid_flg_reg = 0;
```

```
        else data_valid_flg_reg = operate_done;

    end

    assign data_valid_flg = data_valid_flg_reg;

    //-----------------------------------------------------------------
    // Assign output variables

    assign samp_req = (pr_ctrl_status == `VNC_CTRL_STATUS_OPERATE) ? 1 :
        0;
    assign operate_ctrl = (pr_ctrl_status == `VNC_CTRL_STATUS_OPERATE) ?
        1 : 0;

endmodule // vnc_ctrl_no_cg
```

D.2.4 *vnc_input_if.sv*

vnc_input_if* for *Approach 1

```
//-----------------------------------------------------------------
// VNC_INPUT_IF_ALL_CG.SV
// VNC input interface sub-module. Exhaustive clock gating.
//-----------------------------------------------------------------
// Timescale
`TIMESCALE

module vnc_input_if_all_cg(sys_clk,
                           rst_n,
                           data_in,
                           samp_flg,
                           operate_ctrl,

                           qual_flg,
                           qual_data);

    //-----------------------------------------------------------------
    // Define module variables

    // Inputs
    input data_in;
    input samp_flg;
    input rst_n;
    input sys_clk;
    input operate_ctrl;

    // Outputs
    output qual_data;
    output qual_flg;

    // Intern wires
    wire    intern_clk;
    wire    data_reg_clk;
    wire    qual_flg_reg_clk;
    wire    vn_xor;
```

```
wire    operate_en;

// Intern registers
reg     data_load_en_reg;
reg     data_reg;

//-----------------------------------------------------------------
// Assign input variables

assign operate_en = operate_ctrl & samp_flg;

//-----------------------------------------------------------------
// Clock gate for intern clock signal

clk_gate intern_clk_gate(.clk_in(sys_clk),
                         .ctrl(operate_en),
                         .clk_out(intern_clk));

//-----------------------------------------------------------------
// Register for data_load_en (Enable signal for input data register)

always_ff @(posedge intern_clk  or negedge rst_n) begin

   if(!rst_n) data_load_en_reg = 1;

   else data_load_en_reg = ~data_load_en_reg;

end

//-----------------------------------------------------------------
// Clock gate for data register

clk_gate data_reg_clk_gate(.clk_in(intern_clk),
                           .ctrl(data_load_en_reg),
                           .clk_out(data_reg_clk));

//-----------------------------------------------------------------
// Data register for input data from source

always_ff @(posedge data_reg_clk or negedge rst_n) begin

   if(!rst_n) data_reg = 0;

   else data_reg = data_in;

end

//-----------------------------------------------------------------
// Assign von Neumann XOR operation

assign vn_xor = data_in ^ data_reg;

//-----------------------------------------------------------------
// Assign output variables

assign qual_flg = vn_xor & (~data_load_en_reg);
```

```
   assign qual_data = data_reg;

endmodule // vnc_input_if_all_cg
```

vnc_input_if* for *Approach 2*, *Approach 3* and *Approach 4

```
//-----------------------------------------------------------------------
// VNC_INPUT_IF_NO_CG.SV
// VNC input interface sub-module. No clock gating.
//-----------------------------------------------------------------------
// Timescale
`TIMESCALE

module vnc_input_if_no_cg(sys_clk,
                          rst_n,
                          data_in,
                          samp_flg,
                          operate_ctrl,

                          qual_flg,
                          qual_data);

   //-----------------------------------------------------------------------
   // Define module variables

   // Inputs
   input data_in;
   input samp_flg;
   input rst_n;
   input sys_clk;
   input operate_ctrl;

   // Outputs
   output qual_data;
   output qual_flg;

   // Intern wires
   wire   vn_xor;
   wire   operate_en;
   wire   data_load_en;

   // Intern register
   reg    data_load_en_reg;
   reg    data_reg;

   //-----------------------------------------------------------------------
   // Assign input variables

   assign operate_en = operate_ctrl & samp_flg;

   //-----------------------------------------------------------------------
   // Register for data_load_en (Enable signal for input data register)

   always_ff @(posedge sys_clk  or negedge rst_n) begin

      if(!rst_n) data_load_en_reg = 1;
```

```
        else data_load_en_reg = (operate_en) ? ~data_load_en_reg :
            data_load_en_reg;

    end

    assign data_load_en = data_load_en_reg & operate_en;

    //-----------------------------------------------------------------
    // Data register for input data from source

    always_ff @(posedge sys_clk or negedge rst_n) begin

        if(!rst_n) data_reg = 0;

        else data_reg = (data_load_en & operate_en) ?  data_in : data_reg
            ;

    end

    //-----------------------------------------------------------------
    // Assign von Neumann XOR operation

    assign vn_xor = data_in ^ data_reg;

    //-----------------------------------------------------------------
    // Assign output variables

    assign qual_flg = vn_xor & (~data_load_en_reg);
    assign qual_data = data_reg;

endmodule // vnc_input_if_no_cg
```

D.2.5 *vnc_output_memory.sv*

vnc_output_memory* for *Approach 1

```
//-----------------------------------------------------------------
// VNC_OUTPUT_MEMORY_ALL_CG.SV
// VNC output memory sub-module. Exhaustive clock gating used.
//-----------------------------------------------------------------
// Timescale
`TIMESCALE

module vnc_output_memory_all_cg(sys_clk,
                                rst_n,
                                qual_flg,
                                qual_data,
                                operate_ctrl,
                                data_out,
                                mem_count);


    //-----------------------------------------------------------------
    // Define module variables
```

```
// Input
input sys_clk;
input rst_n;
input qual_flg;
input qual_data;
input operate_ctrl;

// Outputs
output [0:`OUTPUT_MEM_SIZE-1]            data_out;
output [$clog2(`OUTPUT_MEM_SIZE)-1:0]    mem_count;

// Intern wires
wire [`OUTPUT_MEM_SIZE-1:0] data_reg_addr;
wire [`OUTPUT_MEM_SIZE-1:0] data_reg_clk;
wire                        output_mem_clk;
wire                        operate_en;

// Intern registers
reg [$clog2(`OUTPUT_MEM_SIZE)-1:0] mem_count_reg;


//-----------------------------------------------------------------
// Assing input signals

assign operate_en = operate_ctrl & qual_flg;

//-----------------------------------------------------------------
// Include clock gate for all registers in the submodule

clk_gate output_mem_clk_gate(.clk_in(sys_clk),
                             .ctrl(operate_en),
                             .clk_out(output_mem_clk));

//-----------------------------------------------------------------
// Memory counter

always_ff @(posedge output_mem_clk or negedge rst_n) begin

   // Reset
   if(!rst_n) mem_count_reg = 0;

   // Increase counter
   else mem_count_reg = mem_count_reg + 1;

end

// Assign to output
assign mem_count = mem_count_reg;

//-----------------------------------------------------------------
// Memory address decoder

generate

   genvar addr_i;
```

```
        for(addr_i=0; addr_i < `OUTPUT_MEM_SIZE; addr_i++) begin

            // Set bit in data_reg_addr according to current memory count
            assign data_reg_addr[addr_i] = (addr_i == mem_count_reg) ? 1 :
                0;

        end // for: addr_i < `OUTPUT_MEM_SIZE

    endgenerate

    //-----------------------------------------------------------------
    // Memory register bank

    generate

        genvar reg_i;

        for (reg_i=0; reg_i < `OUTPUT_MEM_SIZE; reg_i++) begin

            clk_gate data_reg_clk_gate_i(.clk_in(output_mem_clk),
                                         .ctrl(data_reg_addr[reg_i]),
                                         .clk_out(data_reg_clk[reg_i]));

            memory_reg data_reg_i(.clk(data_reg_clk[reg_i]),
                                  .rst_n(rst_n),
                                  .data_in(qual_data),
                                  .data_out(data_out[reg_i]));

        end // for: reg_i < `OUTPUT_MEM_SIZE

    endgenerate

endmodule // vnc_output_memory_all_cg
```

vnc_output_memory* for *Approach 2

```
//-----------------------------------------------------------------
// VNC_OUTPUT_MEMORY_SOME_CG.SV
// VNC output memory sub-moudle. Clock gates for at least 4 registers
//-----------------------------------------------------------------
// Timescale
`TIMESCALE

module vnc_output_memory_some_cg(sys_clk,
                                 rst_n,
                                 qual_flg,
                                 qual_data,
                                 operate_ctrl,
                                 data_out,
                                 mem_count);


    //-----------------------------------------------------------------
    // Define module variables
```

```
// Input
input sys_clk;
input rst_n;
input qual_flg;
input qual_data;
input operate_ctrl;

// Outputs
output [0:`OUTPUT_MEM_SIZE-1]              data_out;
output [$clog2(`OUTPUT_MEM_SIZE)-1:0]      mem_count;

// Intern wires
wire [`OUTPUT_MEM_SIZE-1:0] data_reg_addr;

wire [`OUTPUT_MEM_SIZE-1:0] data_reg_clk;
wire [`NUM_OUTPUT_CG-1:0]   gated_clk;

wire                        output_mem_clk;
wire                        operate_en;

wire [`NUM_OUTPUT_CG-1:0]    data_reg_clk_gate_addr;

// Intern registers
reg [$clog2(`OUTPUT_MEM_SIZE)-1:0] mem_count_reg;

//-----------------------------------------------------------------
// Assing input signals

assign operate_en = operate_ctrl & qual_flg;

//-----------------------------------------------------------------
// Include clock gate for all registers in the submodule

clk_gate output_mem_clk_gate(.clk_in(sys_clk),
                             .ctrl(operate_en),
                             .clk_out(output_mem_clk));

//-----------------------------------------------------------------
// Memory counter

always_ff @(posedge output_mem_clk or negedge rst_n) begin

   // Reset
   if(!rst_n) mem_count_reg = 0;

   // Increase counter
   else mem_count_reg = mem_count_reg + 1;

end

// Assign to output
assign mem_count = mem_count_reg;

//-----------------------------------------------------------------
// Memory address decoder
```

```
generate

   genvar addr_i;

   for(addr_i=0; addr_i < `OUTPUT_MEM_SIZE; addr_i++) begin

      // Set bit in data_reg_addr according to current memory count
      assign data_reg_addr[addr_i] = (addr_i == mem_count_reg) ? 1 :
          0;

   end // for: addr_i < `OUTPUT_MEM_SIZE

endgenerate

//-----------------------------------------------------------------
// Output register clock gate bank

generate

   genvar cg_i;

   for (cg_i=0; cg_i < `NUM_OUTPUT_CG; cg_i++) begin

      // Select active clock gate
      assign data_reg_clk_gate_addr[cg_i] = (cg_i == mem_count_reg[
          $clog2(`OUTPUT_MEM_SIZE)-1:$clog2(`REG_PER_CG)]) ?
          operate_en : 0;

      // Clock gate
      clk_gate data_reg_clk_gate_i(.clk_in(sys_clk),
                                   .ctrl(data_reg_clk_gate_addr[cg_i
                                       ]),
                                   .clk_out(gated_clk[cg_i]));

      assign data_reg_clk[cg_i*`REG_PER_CG] = gated_clk[cg_i];
      assign data_reg_clk[cg_i*`REG_PER_CG + 1] = gated_clk[cg_i];
      assign data_reg_clk[cg_i*`REG_PER_CG + 2] = gated_clk[cg_i];
      assign data_reg_clk[cg_i*`REG_PER_CG + 3] = gated_clk[cg_i];

   end

endgenerate

//-----------------------------------------------------------------
// Memory register bank

generate

   genvar reg_i;

   for (reg_i=0; reg_i < `OUTPUT_MEM_SIZE; reg_i++) begin

      memory_en_reg data_reg_i(.clk(output_mem_clk),  //
          data_reg_clk[reg_i]
                               .rst_n(rst_n),
                               .data_in(qual_data),
```

```
                                    .en(data_reg_addr[reg_i]),
                                    .data_out(data_out[reg_i]));

        end // for: reg_i < `OUTPUT_MEM_SIZE

    endgenerate

endmodule // vnc_output_memory_some_cg
```

vnc_output_memory* for *Approach 3* and *Approach 4

```
//-----------------------------------------------------------------------
// VNC_OUTPUT_MEMORY_NO_CG.SV
// VNC output memory sub-module. No clock gating
//-----------------------------------------------------------------------
// Timescale
`TIMESCALE

module vnc_output_memory_no_cg(sys_clk,
                               rst_n,
                               qual_flg,
                               qual_data,
                               operate_ctrl,
                               data_out,
                               mem_count);


    //-------------------------------------------------------------------
    // Define module variables

    // Input
    input sys_clk;
    input rst_n;
    input qual_flg;
    input qual_data;
    input operate_ctrl;

    // Outputs
    output [0:`OUTPUT_MEM_SIZE-1]            data_out;
    output [$clog2(`OUTPUT_MEM_SIZE)-1:0]    mem_count;

    // Intern wires
    wire [`OUTPUT_MEM_SIZE-1:0] data_reg_addr;
    wire [`OUTPUT_MEM_SIZE-1:0] load_en;
    wire                        operate_en;

    // Intern registers
    reg [$clog2(`OUTPUT_MEM_SIZE)-1:0] mem_count_reg;

    //-------------------------------------------------------------------
    // Assing input signals

    assign operate_en = operate_ctrl & qual_flg;

    //-------------------------------------------------------------------
    // Memory counter
```

```
    always_ff @(posedge sys_clk or negedge rst_n) begin

        // Reset
        if(!rst_n) mem_count_reg = 0;

        // Increase counter
        else mem_count_reg = (operate_en) ?  mem_count_reg + 1 :
            mem_count_reg;

    end

    // Assign to output
    assign mem_count = mem_count_reg;

    //----------------------------------------------------------------
    // Memory address decoder

    generate

        genvar addr_i;

        for(addr_i=0; addr_i < `OUTPUT_MEM_SIZE; addr_i++) begin

            // Set bit in data_reg_addr according to current memory count
            assign data_reg_addr[addr_i] = (addr_i == mem_count_reg) ? 1 :
                0;

        end // for: addr_i < `OUTPUT_MEM_SIZE

    endgenerate

    //----------------------------------------------------------------
    // Memory register bank

    generate

        genvar reg_i;

        for (reg_i=0; reg_i < `OUTPUT_MEM_SIZE; reg_i++) begin

            assign load_en[reg_i] = data_reg_addr[reg_i] & operate_en;

            memory_en_reg data_reg_i(.clk(sys_clk),
                                     .rst_n(rst_n),
                                     .data_in(qual_data),
                                     .en(load_en[reg_i]),
                                     .data_out(data_out[reg_i]));

        end // for: reg_i < `OUTPUT_MEM_SIZE

    endgenerate

endmodule // vnc_output_memory_no_cg
```

D.3 IHF Implementation

D.3.1 Definitions for the IHF

IHF parameter

```
// IHF Control States
`define IHF_CTRL_STATUS_IDLE   2'b00
`define IHF_CTRL_STATUS_READ   2'b01
`define IHF_CTRL_STATUS_COMP   2'b11
`define IHF_CTRL_STATUS_LOAD   2'b10

// IHF parameter
`define IHF_INPUT_SIZE 128
`define IHF_OUTPUT_SIZE 32
`define IHF_PUBLIC_PARA_SIZE (`IHF_INPUT_SIZE + `IHF_OUTPUT_SIZE - 1)
`define IHF_OUTPUT_MEM_BLOCK_SIZE (`OUTPUT_MEM_SIZE / `IHF_OUTPUT_SIZE)

// Public parameter
// NOTE : Format is [MSB....LSB]
//`define IHF_PUBLIC_PARA 129'h19dd2c957f1ea49d3382e5d09c529f2cd
//`define IHF_PUBLIC_PARA 128'h9dd2c957f1ea49d3382e5d09c529f2cd
//`define IHF_PUBLIC_PARA 131'h69dd2c957f1ea49d3382e5d09c529f2cd
`define IHF_PUBLIC_PARA 159'h1aee655c9dd2c957f1ea49d3382e5d09c529f2cd
//`define IHF_PUBLIC_PARA 135'h1A9dd2c957f1ea49d3382e5d09c529f2cd
//`define IHF_PUBLIC_PARA 143'h1aee9dd2c957f1ea49d3382e5d09c529f2cd
//`define IHF_PUBLIC_PARA 64'h9dd2c957f1ea49d3
//`define IHF_PUBLIC_PARA 65'h19dd2c957f1ea49d3
//`define IHF_PUBLIC_PARA 67'h19dd2c957f1ea49d3
//`define IHF_PUBLIC_PARA 71'h9dd2c957f1ea49d33
//`define IHF_PUBLIC_PARA 33'h19dd2c957
//`define IHF_PUBLIC_PARA 32'h9dd2c957
//`define IHF_PUBLIC_PARA 9'h1A2
```

D.3.2 *ihf.sv*

The IHF top-module

```
//------------------------------------------------------------------------------
// IHF.SV
// IHF top-module
//------------------------------------------------------------------------------
// Timescale
`TIMESCALE

module ihf(sys_clk,
           rst_n,
           start,
           data_in,
           samp_flg,
           samp_req,
           data_out,
           data_valid_flg);

   //-------------------------------------------------------------
   // Define module variables
```

```
// Inputs
input sys_clk;
input rst_n;
input start;
input data_in;
input samp_flg;

// Outputs
output                              samp_req;
output [0:`OUTPUT_MEM_SIZE-1]       data_out;
output                              data_valid_flg;

// Intern wires
wire                            input_load_ctrl;
wire                            comp_ctrl;
wire                            public_para_load_ctrl;
wire                            output_load_ctrl;
wire [`IHF_INPUT_SIZE-1:0]      vector_in;
wire [`IHF_OUTPUT_SIZE-1:0]     vector_out;

//-----------------------------------------------------------------
// Connect sub-modules

// Control unit
ihf_ctrl IHF_CTRL(.sys_clk(sys_clk),
                  .rst_n(rst_n),
                  .start(start),
                  .samp_flg(samp_flg),
                  .data_valid_flg(data_valid_flg),
                  .samp_req(samp_req),
                  .input_load_ctrl(input_load_ctrl),
                  .comp_ctrl(comp_ctrl),
                  .output_load_ctrl(output_load_ctrl),
                  .public_para_load_ctrl(public_para_load_ctrl));

// Input interface
ihf_input_if IHF_INPUT_IF(.sys_clk(sys_clk),
                          .rst_n(rst_n),
                          .data_in(data_in),
                          .samp_flg(samp_flg),
                          .input_load_ctrl(input_load_ctrl),
                          .vector_in(vector_in));

// Computation Unit
ihf_comp IHF_COMP(.sys_clk(sys_clk),
                  .rst_n(rst_n),
                  .comp_ctrl(comp_ctrl),
                  .public_para_load_ctrl(public_para_load_ctrl),
                  .vector_in(vector_in),
                  .vector_out(vector_out));

// Output memory
ihf_output_memory IHF_OUTPUT_MEMORY(.sys_clk(sys_clk),
                                    .rst_n(rst_n),
                                    .vector_out(vector_out),
```

```
                                        .output_load_ctrl(
                                            output_load_ctrl),
                                        .data_out(data_out));

endmodule // ihf
```

D.3.3 *ihf_ctrl.sv*

The *ihf_ctrl* sub-module

```
//-----------------------------------------------------------------------
// IHF_CTRL.SV
// IHF control sub-module
//-----------------------------------------------------------------------
// Timescale
`TIMESCALE

module ihf_ctrl(sys_clk,
                rst_n,
                start,
                samp_flg,
                data_valid_flg,
                samp_req,
                input_load_ctrl,
                comp_ctrl,
                output_load_ctrl,
                public_para_load_ctrl);

    //-----------------------------------------------------------------------
    // Define module variables

    // Inputs
    input sys_clk;
    input rst_n;
    input start;
    input samp_flg;

    // Outputs
    output data_valid_flg;
    output samp_req;
    output input_load_ctrl;
    output comp_ctrl;
    output output_load_ctrl;
    output public_para_load_ctrl;

    // Intern wires
    wire clear_count;
    wire input_loaded;
    wire comp_count_reached;
    wire comp_done;
    wire output_count_reached;
    wire output_count_en;

    // Intern registers
    reg [$clog2(`IHF_INPUT_SIZE)-1:0]                count_reg;
    reg [$clog2(`IHF_OUTPUT_MEM_BLOCK_SIZE)-1:0] output_count_reg;
```

```
reg [1:0]                                   ihf_ctrl_status;

reg                                         data_valid_flg_reg;

reg [1:0]                                   ihf_ctrl_status_nx;


//-----------------------------------------------------------------
// Assign intern status flags

assign input_loaded = samp_flg & (count_reg == `IHF_INPUT_SIZE-1) ?
    1 : 0;
assign comp_count_reached = (count_reg == `IHF_OUTPUT_SIZE-1) ? 1 :
    0;
assign comp_done = (ihf_ctrl_status == `IHF_CTRL_STATUS_COMP) ?
    comp_count_reached : 0;
assign output_count_reached = (output_count_reg ==
    `IHF_OUTPUT_MEM_BLOCK_SIZE -1) ? 1: 0;

//-----------------------------------------------------------------
// Next state logic

always_comb begin

       case(ihf_ctrl_status)

      // IDLE
      `IHF_CTRL_STATUS_IDLE: begin
         if(start) ihf_ctrl_status_nx = `IHF_CTRL_STATUS_READ;
         else ihf_ctrl_status_nx = `IHF_CTRL_STATUS_IDLE;
      end

      // READ
      `IHF_CTRL_STATUS_READ: begin
         if(input_loaded) ihf_ctrl_status_nx = `IHF_CTRL_STATUS_COMP;
         else ihf_ctrl_status_nx = `IHF_CTRL_STATUS_READ;
      end

      // COMP
      `IHF_CTRL_STATUS_COMP: begin
         if(comp_done) ihf_ctrl_status_nx = `IHF_CTRL_STATUS_LOAD;
         else ihf_ctrl_status_nx = `IHF_CTRL_STATUS_COMP;
      end

      // LOAD
      `IHF_CTRL_STATUS_LOAD: begin
         if(output_count_reached) ihf_ctrl_status_nx =
            `IHF_CTRL_STATUS_IDLE;
         else ihf_ctrl_status_nx = `IHF_CTRL_STATUS_READ;
      end

      // Default (not reachable)
      default: ihf_ctrl_status_nx = `IHF_CTRL_STATUS_IDLE;

   endcase // case (ihf_ctrl_status)
```

```
end // always_comb

//-----------------------------------------------------------------
// State register for ihf_ctrl_status

always_ff @(posedge sys_clk or negedge rst_n) begin

   // Reset
   if(!rst_n) ihf_ctrl_status = 'IHF_CTRL_STATUS_IDLE;

   // Update status on pos. edge sys_clk
   else ihf_ctrl_status = ihf_ctrl_status_nx;

end // always_ff


//-----------------------------------------------------------------
// Register for data_valid_flg

always_ff @(posedge sys_clk or negedge rst_n) begin

   // Reset
   if(!rst_n) data_valid_flg_reg = 0;

   else data_valid_flg_reg = ((ihf_ctrl_status ==
      'IHF_CTRL_STATUS_LOAD) & output_count_reached) ? 1 : 0;

end // always_ff

//-----------------------------------------------------------------
// Assign intern control signals

assign clear_count = (ihf_ctrl_status == 'IHF_CTRL_STATUS_LOAD) ? 1
   : 0;
assign output_count_en = (ihf_ctrl_status == 'IHF_CTRL_STATUS_LOAD)
   ? 1 : 0;
assign count_clear = (ihf_ctrl_status == 'IHF_CTRL_STATUS_IDLE) ? 1
   : 0;

//-----------------------------------------------------------------
// General purpose counter
// Behaviour of counter is dependent on ihf_ctrl_status
// IDLE: Set to zero (default)
// READ: Increase count on pos.edge sys_clk if samp_flg
// COMP: Set count to 0 on pos.edge sys_clk if clear_count. Else
   increase.
//
// NOTE: Counter overflows on IHF_INPUT_SIZE

always_ff @(posedge sys_clk or negedge rst_n) begin

   // Reset
   if(!rst_n) count_reg = 0;

   // Pos edge sys_clk
   else begin
```

```
        case(ihf_ctrl_status)

          // Increase count on samp_flg if status = READ
          `IHF_CTRL_STATUS_READ: begin
             if(samp_flg) count_reg = count_reg + 1;
             else count_reg = count_reg;
          end

          // Increase count on sys_clk if status = COMP
          // Reset on clear_count
          `IHF_CTRL_STATUS_COMP: begin
             if(clear_count) count_reg = 0;
             else count_reg = count_reg + 1;
          end

          // Default (IDLE)
          default: count_reg = 0;

        endcase

    end // pos. edge sys_clk

end // always_ff

//-----------------------------------------------------------------
// Counter for output loads

always_ff @(posedge sys_clk or negedge rst_n) begin

    // Reset
    if(!rst_n) output_count_reg = 0;

    // Pos clk egde
    else begin

        if(count_clear) output_count_reg = 0;
        else output_count_reg = (output_count_en) ? output_count_reg +
            1 : output_count_reg;

    end // else

end // always_ff

//-----------------------------------------------------------------
// Assign output variables

assign input_load_ctrl = (ihf_ctrl_status == `IHF_CTRL_STATUS_READ)
    ? 1 : 0;
assign comp_ctrl = (ihf_ctrl_status == `IHF_CTRL_STATUS_COMP) ? 1 :
    0;
assign public_para_load_ctrl = start | (comp_done & !
    output_count_reached);
assign output_load_ctrl = (ihf_ctrl_status == `IHF_CTRL_STATUS_LOAD)
     ? 1 : 0;
assign data_valid_flg = data_valid_flg_reg;
```

```
    assign samp_req = (ihf_ctrl_status == `IHF_CTRL_STATUS_READ) ? 1 :
        0;

endmodule // ihf_ctrl
```

D.3.4 *ihf_input_if.sv*

The *ihf_input_if* sub-module

```
//-----------------------------------------------------------------
// IHF_INPUT_IF.SV
// IHF input interface sub-module
//-----------------------------------------------------------------
// Timescale
`TIMESCALE

module ihf_input_if(sys_clk,
                    rst_n,
                    data_in,
                    samp_flg,
                    input_load_ctrl,

                    vector_in);

    //-----------------------------------------------------------------
    // Define module variables

    // Inputs
    input sys_clk;
    input rst_n;
    input data_in;
    input samp_flg;
    input input_load_ctrl;

    // Outputs
    output [`IHF_INPUT_SIZE-1:0] vector_in;

    // Intern wires
    wire input_load_en;

    // Intern registers
    reg [`IHF_INPUT_SIZE-1:0] vector_in_reg;


    //-----------------------------------------------------------------
    // Assign input variables

    assign input_load_en = input_load_ctrl & samp_flg;

    //-----------------------------------------------------------------
    // Shift register for input data
    // On pos. edge of sys_clk data_in is loaded as MSB if input_load_en
        .
    // Other bits are shifted towards LSB.

    always_ff @ (posedge sys_clk or negedge rst_n) begin
```

```
        // Reset
        if(!rst_n) vector_in_reg = 0;

        // Pos. edge sys_clk
        else begin

            if(input_load_en) begin
                vector_in_reg = vector_in_reg >> 1;
                vector_in_reg[`IHF_INPUT_SIZE-1] = data_in;
            end
            else vector_in_reg = vector_in_reg;

        end // pos. edge sys_clk

    end // always_ff

    //-----------------------------------------------------------------
    // Assign output variables

    assign vector_in = vector_in_reg;

endmodule // ihf_input_if
```

D.3.5 *ihf_comp.sv*

The *ihf_comp* sub-module

```
//-----------------------------------------------------------------
// IHF_COMP.SV
// IHF sub-module to perform the IHF algorithm
//-----------------------------------------------------------------

// Timescale
`TIMESCALE

module ihf_comp(sys_clk,
                rst_n,

                comp_ctrl,
                public_para_load_ctrl,

                vector_in,
                vector_out);

    //-----------------------------------------------------------------
    // Define module variables

    // Input
    input sys_clk;
    input rst_n;
    input comp_ctrl;
    input public_para_load_ctrl;
    input [`IHF_INPUT_SIZE-1:0] vector_in;

    // Output
```

```
output [`IHF_OUTPUT_SIZE-1:0] vector_out;

// Intern register
reg [`IHF_OUTPUT_SIZE-1:0] vector_out_reg;
reg [`IHF_PUBLIC_PARA_SIZE-1:0] public_para_reg;

// Intern wire
wire [`IHF_INPUT_SIZE-1:0]      product_vector;
wire                            sum;

//-----------------------------------------------------------------
// Register for public parameter
// Shifts towards LSB on pos. edge of sys_clk if comp_ctrl
// Loads IHF_PUBLIC_PARAMETER on pos. edge of sys_clk if
    public_para_load_ctrl
// Load of IHF_PUBLIC_PARA has priority over shift

always_ff @(posedge sys_clk or negedge rst_n) begin

   // Reset
   if(!rst_n) public_para_reg = `IHF_PUBLIC_PARA;

   // Pos. edge sys_clk
   else begin

      if(public_para_load_ctrl) public_para_reg = `IHF_PUBLIC_PARA;
      else if(comp_ctrl) public_para_reg = public_para_reg >> 1;
      else public_para_reg = public_para_reg;

   end

end // always_ff

//-----------------------------------------------------------------
// Execute computations for the IHF

assign product_vector = public_para_reg[`IHF_INPUT_SIZE-1:0] &
    vector_in;
assign sum = ^product_vector;

//-----------------------------------------------------------------
// Register for vector_out
// On pos. edge of sys_clk it loads current sum as MSB and shifts
    towards
// LSB if comp_ctrl

always_ff @(posedge sys_clk or negedge rst_n) begin

   // Reset
   if(!rst_n) vector_out_reg = 0;

   // Pos. edge sys_clk
   else begin

      if(comp_ctrl) begin
         vector_out_reg = vector_out_reg >> 1;
```

```
            vector_out_reg['IHF_OUTPUT_SIZE-1] = sum;
        end
        else vector_out_reg = vector_out_reg;

    end
  end // always_ff

  //-----------------------------------------------------------------
  // Assign output variables

  assign vector_out = vector_out_reg;

endmodule // ihf_comp
```

D.3.6 *ihf_output_memory.sv*

The *ihf_output_memory* sub-module

```
//-----------------------------------------------------------------
// IHF_OUTPUT_MEMORY.SV
// IHF output memory sub-module
//-----------------------------------------------------------------
// Timescale
`TIMESCALE

module ihf_output_memory(sys_clk,
                         rst_n,
                         vector_out,
                         output_load_ctrl,
                         data_out);

  //-----------------------------------------------------------------
  // Define module variables

  // Inputs
  input sys_clk;
  input rst_n;
  input ['IHF_OUTPUT_SIZE-1:0] vector_out;
  input                        output_load_ctrl;

  // Output
  output [0:'OUTPUT_MEM_SIZE-1] data_out;

  // Intern wires
  wire ['IHF_OUTPUT_SIZE-1:0]['IHF_OUTPUT_MEM_BLOCK_SIZE-1:0]
      data_out_reg;


  //-----------------------------------------------------------------
  // Multi-dimensional shift register for data_out
  // The output memory is devided in IHF_OUTPUT_MEM_BLOCK_SIZE blocks
      with
  // IHF_OUTPUT_SIZE registers each. The "top" register of each block
      is loaded
  // with one bit of output vector on pos. edge sys_clk if
      load_output_ctrl.
```

```
    // The other registers are loaded in a top-to-buttom manner. If the
        whole
    // output has been loaded, the registers at the buttom of the blocks
         contain
    // the LSBs of the output, the top registers contain the MSBs.

    generate

    genvar block_i;
    genvar reg_i;

    for(block_i=0; block_i < `IHF_OUTPUT_SIZE; block_i++) begin

      for(reg_i=`IHF_OUTPUT_MEM_BLOCK_SIZE-1; reg_i >= 0; reg_i--) begin
      if(reg_i == `IHF_OUTPUT_MEM_BLOCK_SIZE-1) begin

      // Top register of each block
      memory_en_reg data_reg_block_i_reg_top(.clk(sys_clk),
      .rst_n(rst_n),
      .data_in(vector_out[block_i]),
      .en(output_load_ctrl),
      .data_out(data_out_reg[block_i][`IHF_OUTPUT_MEM_BLOCK_SIZE-1]));

      end

      else begin

      // Other registers are loaded by shift operation
      memory_en_reg data_reg_block_i_reg_i(.clk(sys_clk),
      .rst_n(rst_n),
      .data_in(data_out_reg[block_i][reg_i+1]),
      .en(output_load_ctrl),
      .data_out(data_out_reg[block_i][reg_i]));

      end

      // Assign register bank to output
      assign data_out[reg_i*`IHF_OUTPUT_SIZE + block_i] = data_out_reg[
          block_i][reg_i];

      end // for reg_i

    end // for block_i

    endgenerate

endmodule // ihf_output_memory
```

D.4 Simulation Setup

D.4.1 Definitions

Definition of *src*

```
`ifndef TYPES
```

```
`define TYPES

// Define source struct
typedef struct { string fileName;
                 int bitPerLine;
                 int fileID;
                 int scanID;
                 } src;

`endif // TYPES
```

D.4.2 Example of a test bench

Test bench for the IHF

```
//-----------------------------------------------------------------------
// IHF_TESTBENCH.SV
// Testbench for the IHF module
//-----------------------------------------------------------------------
// Timescale
`TIMESCALE

module ihf_testbench(rst_n,
                     start_ihf,

                     source_file,

                     sys_clk,
                     empty_flg,
                     data_out,
                     data_valid_flg,
                     samp_count);

   //--------------------------------------------------------------------
   // Define variables

   // Inputs / Simulation interface
   input rst_n;
   input start_ihf;
   input src source_file;

   // Outputs / Simulation interface
   output                          sys_clk;
   output                          empty_flg;
   output [0:`OUTPUT_MEM_SIZE-1]   data_out;
   output                          data_valid_flg;
   output int                      samp_count;

   // Intern wires
   wire bit_req;
   wire source_data;
   wire samp_flg;

   //--------------------------------------------------------------------
   // Connect modules
```

```
    // Connect clock generator
    clk_gen CLK_GEN(.sys_clk(sys_clk));

    // Connect the data source
    source SOURCE(.sys_clk(sys_clk),
                  .rst_n(rst_n),
                  .bit_req(bit_req),
                  .source_file(source_file),
                  .source_data(source_data),
                  .samp_flg(samp_flg),
                  .empty_flg(empty_flg),
                  .samp_count_out(samp_count));

    // Connect to device under test (IHF)
    ihf IHF(.sys_clk(sys_clk),
            .rst_n(rst_n),
            .start(start_ihf),
            .data_in(source_data),
            .samp_flg(samp_flg),
            .samp_req(bit_req),
            .data_out(data_out),
            .data_valid_flg(data_valid_flg));

endmodule // ihf_testbench
```

Model of a clock generator

```
// Timescale
`TIMESCALE

// Define the clock generator
module clk_gen( output reg sys_clk );

    initial sys_clk = 0;

    always
      begin
         #(`SYS_CLK_PER/2) sys_clk = 1;
         #(`SYS_CLK_PER/2) sys_clk = 0;
      end //

endmodule // clk_gen
```

Model of the entropy source

```
//-----------------------------------------------------------------
// SOURCE.SV
// Model of the entropy source
//-----------------------------------------------------------------
// Define timescale
`TIMESCALE

//-----------------------------------------------------------------
// Specifie design
```

```
module source(sys_clk,
              rst_n,
              bit_req,

              source_file,

              source_data,
              samp_flg,
              empty_flg,
              samp_count_out);

  //-----------------------------------------------------------------
  // Define module variables

  // Inputs
  input sys_clk;
  input rst_n;
  input bit_req;

  input src source_file;


  // Outputs
  output reg source_data;
  output reg samp_flg;
  output reg empty_flg;

  output int samp_count_out;

  // Intern variables
  int sys_clk_count = 0;
  int bit_count = 0;
  int samp_count;
  bit empty = 0;
  bit [`MAX_BIT_PER_LINE-1:0] line;

  //-----------------------------------------------------------------
  // Source clock prescaler
  // Uses sys_clk to create a downscaled periodic signal samp_flg
  // Samp_flg indicates that a sample can be read from the source

  always @(posedge sys_clk or negedge rst_n) begin

    // Reset
    if(!rst_n) begin

      samp_flg = 0;
      sys_clk_count = 0;

    end // if !rst_n

    // On posedge sys_clk
    else begin

      // If a bit is requested
      if(bit_req) begin
```

```
        // Increase sys_clk counter
        sys_clk_count++;

        // If number of sys_clk cycles for a sample is reached
        if(sys_clk_count == `SOURCE_SAMP_CYCL) begin

          // Reset sys_clk counter
          sys_clk_count = 0;

          // Set samp_flg
          samp_flg = 1;

        end // if: sys_clk_counter == SOURCE_SAMP_CYCL

        // If number of sys_clk cycles for a sample is NOT reached
        else samp_flg = 0;

      end // if: bit_req

      // If no bit is requested
      else begin

        // Keep count sys_clk_count, samp_flg low
        sys_clk_count = 0;
        samp_flg = 0;

      end // else: !bit_req

    end // if rst_n

end // always @ (posedge sys_clk or negedge rst_n)

//----------------------------------------------------------------
// Source buffer reader
// On each samp_flg one bit is read from source_buffer and written
    to
// source_data.
// If all bits are read from source_buffer empty_flg is set

always @(posedge samp_flg or negedge rst_n) begin

    // Reset
    if(!rst_n) begin

      source_data = 0;
      empty_flg = 0;
      samp_count_out = 0;

    end // if: !rst_n

    // One posedeg samp_flg
    else begin

      // If the buffer is NOT empty
      if (!empty) begin
```

```
            // If new line is reached
            if(bit_count == 0) begin

                // Read next line form source file
                source_file.scanID = $fscanf(source_file.fileID, "%b\n",
                    line);

            end // if: bit_count == 0

            // Set empty flg if end of source file has been reached
            if (source_file.scanID == `EOF) begin

                empty = 1;
                empty_flg = 1;
                source_data = 0;

            end

            // If end of source not reached
            else begin

                // Write bit from line to source_data
                source_data = line[source_file.bitPerLine - 1 -bit_count
                    ];

                // Keep empty_flg low
                empty_flg = 0;

                // Increase sample counter
                samp_count++;

                // Update bit_count
                if(bit_count == source_file.bitPerLine-1) bit_count = 0;
                else bit_count++;

            end

            // Set samp_count to output
            samp_count_out = samp_count;

        end // if (!empty)

        // If the buffer is empty
        else begin

            // Set outputs
            source_data = 0;
            empty_flg = 1;
            samp_count_out = samp_count;

        end // else: empty

    end // else: rst_n

end // always: posedge samp_flg or negedge rst_n
```

```
endmodule // source
```

D.4.3 Example of a simulation routine

Simulation routine for the IHF

```
// Simulation to gather toggle data for power estimation
// Run IHF on data from the ADC to produce 1024 words.

// Timescale
`TIMESCALE

module ihf_sim();

   // Specifie simulation parameters
   const int REQ_NUM_WORDS = 1024;
   const string OUTPUT_FILE = "simulationResults/data.
      ihf_adc_energy_sim_res";
   const string INPUT_FILE = "/home/foik/MASTER_PRJ/trng_design/sysvlog
      /models/sourceData/adc_010415_500Mb.txt";
   const int INPUT_BIT_PER_LINE = 8; // Number of bits per line in
      INPUT_FILE

   // Test bench interface (Test bench outputs)
   wire sys_clk;
   wire empty_flg;
   wire [0:`OUTPUT_MEM_SIZE-1] data_out;
   wire                        data_valid_flg;
   int                         samp_count;

   // Test bench interface (Test bench inputs)
   reg rst_n;
   reg start_ihf;
   src source_file;

   // Connect to the IHF test bench
   ihf_testbench IHF_TESTBENCH(.rst_n(rst_n),
                               .start_ihf(start_ihf),

                               .source_file(source_file),

                               .sys_clk(sys_clk),
                               .empty_flg(empty_flg),
                               .data_out(data_out),
                               .data_valid_flg(data_valid_flg),
                               .samp_count(samp_count));

   // Run simulation
   initial begin

      // Simulation variables
      bit                                      exit;
      int                                      word_count;
      int                                      bit_count;
      string                                   exit_status;
```

```
int                                             output_file_id;

// Display info-msg
$display("\n\n\nRunning ihf_sim.sv\n");

// Specifie source input file
source_file.fileName = INPUT_FILE;
source_file.bitPerLine = INPUT_BIT_PER_LINE;

// Open source input file
$display("Opening: %s",source_file.fileName);
source_file.fileID = $fopen(source_file.fileName,"r");

// Exit simulation if source file failed to open
if(source_file.fileID == 0) begin
   $display("ERROR source_id handle was NULL. Exit simulation!");
   $finish;
end

// Open output file
output_file_id = $fopen(OUTPUT_FILE, "w");

// Initial stimulation values
rst_n = 1;
start_ihf = 0;

// Initial intern values
word_count = 0;
bit_count = 0;
exit = 0;

// Reset test bench
@(posedge sys_clk);
rst_n = 0;

// Start IHF
@(posedge sys_clk);
rst_n = 1;
start_ihf = 1;

// Run main simulation unti source buffer is empty or requested
   number of bits has been gathered
while(!exit) begin

   // Wait for next pos edge of sys_clk
   @(posedge sys_clk);

   // Clear start_ihf
   if(start_ihf) start_ihf = 0;

   // Exit simulation if source buffer is empty
   if(empty_flg) begin

      // Print unqualified data from IHF to output file
      $fdisplay(output_file_id, "%32b", data_out);
```

```
            exit = 1;
            exit_status = "Source buffer has been emptied";

        end // if : empty_flg

        // If output data from IHF is ready
        if(data_valid_flg) begin

            // Write word to output file
            $fdisplay(output_file_id, "%32b", data_out);

            // Increase word count
            word_count++;

            // Display info-msg to terminal
            $display("Number of collected words: %d", word_count);

            // Exit simulation if requested number of words has been
                gathered
            if(word_count == REQ_NUM_WORDS) begin

                exit = 1;
                exit_status = "Requested number of output words has been
                    collected";

            end

            // Otherwise restart IHF
            else begin

                @(posedge sys_clk);
                start_ihf = 1;

            end // else: !if(word_count == REQ_NUM_WORDS)

        end // if: data_valid_flg

    end // while: !exit

    // Evaluate test
    $display("Test has been exit: %s\n", exit_status);
    $display("Number of used source samples: %d\n", samp_count);

    // Close output file
    $fclose(output_file_id);

    // Stop simulation
    $stop;

  end

endmodule
```

Appendix E

User Guide for Logical Verification and Synthesis for Power Estimations

This user guide is a short description on how to setup and use the *Cadence* tool chain available through the *Mercury* server environment at the *Norwegian University of Science and Technology* (NTNU) for the purpose of logical verification and synthesis for power estimations. It should be stressed that, while the here introduced methods are a functional solution, other more suitable approaches may exist, depending on the specific project. The purpose of this documentation is therefore rather to serve as an example of a working setup, which should be adapted to fit a certain project.

Most of the here presented information can be found in [37][38][1] and the reader is encouraged to consider these documentations for a more detailed background on the used tools. The tools which are considered in this documentation are *Incisive* (commonly referred to as *NCSim*) and *Encounter RTL Compiler* (or simply *RTL Compiler*), both parts of the *Cadence* tool chain. It is assumed that RTL level design has been performed in SystemVerilog. Note that the here presented setup makes partly use of the example setup that accompanies [38], and this example must therefore be available. The here presented setup and some simple example files are contained in a zip-file that accompanies this documentation.

This documentation is structured as follows: First, in Appendix E.1, the basic work flow and its motivation is presented. Second, the reader is introduced to how to set up the tool chain for the given work flow (Appendix E.2). Finally, Appendix E.3 shows how the work flow is executed.

E.1 Work flow

As mentioned, this documentation focuses on two steps of a digital implementation. First, logical verification is considered to test whether or not the considered design behaves in accordance with its specifications. Second, the design is synthesized in order to achieve an estimation of the power requirements of the design.

[1] Note that [38] is an NTNU intern documentation and, as such, not publicly available.

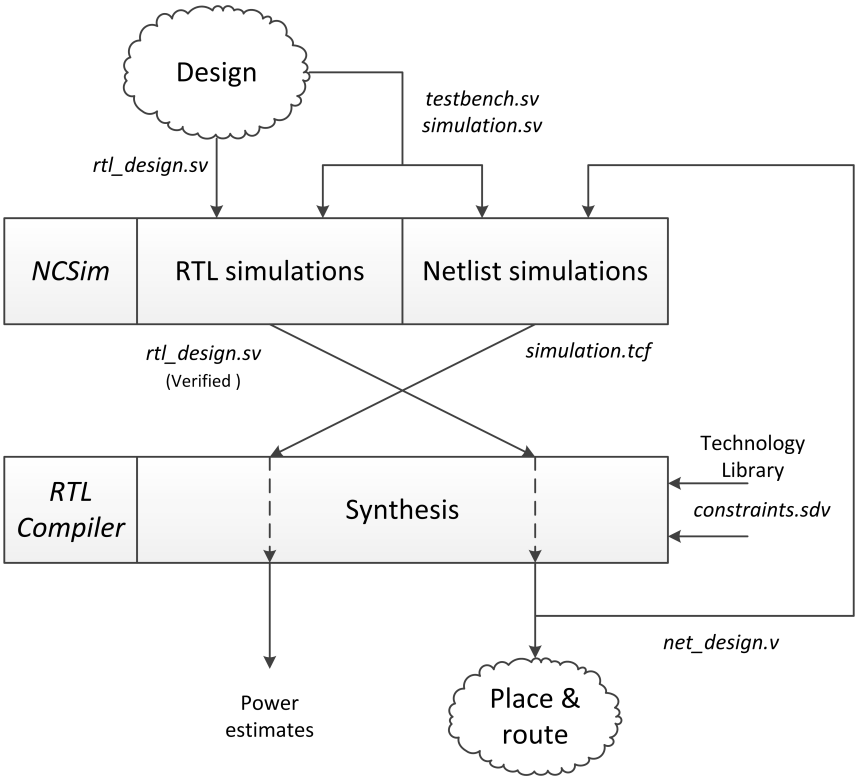


Figure E.1: Work flow of the verification and synthesis process

Figure E.1 illustrates the work flow for a minimal working example. It is assumed that during some preceding design process three SystemVerilog files have been created: *rtl_design.sv*, *testbench.sv* and *simulation.sv*. Each file contains a module with the same name. For example, *rtl_design.sv* declares the module *design*. Further more, *simulation.sv* is the top level instance which includes *testbench.sv*. The testbench in its turn includes *rtl_design.sv*, which is the target of both the verification and the synthesis steps.

To verify the logical behavior of *rtl_design.sv*, the three SystemVerilog files are passed on to *NCSim* which runs *simulation.sv* on the design file. If the verification is successful, the design can be synthesized by *RTL Compiler*. Besides of *rtl_design.sv*, the synthesize tool needs some information about the design constraints, which are specified in *constraints.sdv*, and the used technology library, which is not specified in this documentation (please refer to [38] for more information about the library). Based on this inputs, *RTL Compiler* performs the synthesis for *rtl_design.sv*, which results in a Verilog netlist of the design, *net_design.v*. The netlist can then be used for the "place and route" implementation step, which is described in [38].

In addition to the netlis, *RTL Compiler* can deliver reports on the estimated power, area and timing requirements of the design. For the purpose of this documentation, the power estimates are of main interest. However, it is a well known fact that the power that is required by a digital design depends directly on the toggling activity of the design. Since the *RTL Compiler*, at this point of the work flow, does not have any information about the toggling behavior of *rtl_design.sv*, it uses some default assumptions about the signal activities. These default assumptions are in general not good enough for useful power estimations, especially if most signals of

the design are updated at a much slower rate than the main clock signal. To cope with this problem, a second simulation of the design is performed by passing *simulation.sv*, *testbench.sv* and *net_design.v* (the nestlist *not* the RTL description!) to *NCSim*. The purpose of this second simulation is to gather information about the toggle activity of the design[2], which is exported from *NCSim* in form of a *tcf*-file. Using the *tcf*-file, the synthesis of *rtl_design.sv* can be repeated to achieve a more accurate power estimation.

E.2 Setup

For consistency, the here presented setup is based on the example directory, *tutorial*, of [38]. Thus, *tutorial* should be downloaded and unpacked on the *Mercury* server. Opening the directory, it contains five sub-directories. In addition, the two directories, *sysvlog* and *ncsim*, which can be found in the zip-file that accompanies this user guide, should be included. The directory *sysvlog* contains three SystemVerilog files, which depict a simple example of a simulation-testbench-design hierarchy. The second directory, *ncsim*, contains the setup used to access *NCSim* and run simulations on the SystemVerilog files.

At this point, *tutorial* should thus contain the following seven sub-directories:

```
/tutorial
    /ncsim
    /rtl_compiler
    /soc_encounter
    /sysvlog
    /tech_lef
    /vhdl
    /virtuoso
```

For the purpose of this documentation, *soc_encounter*, *tech_lef* and *vhdl* are not of interest and can be ignored. The directory *rtl_compiler* is used to access *RTL Compiler* in order to synthesize the design file. To avoid confusion, one might consider deleting the content of *rtl_compiler*, *rtl.tcl* and *seq_det.sdc*, as this files belong to the example presented in [38].

E.2.1 Setup of *sysvlog*

Opening *sysvlog*, the directory contains the three example files, *rtl_design.sv*, *testbench.sv* and *simulation.sv*. To simplify the further work flow, it makes sense to create a compile list of all simulation related SystemVerilog files, called *simCompList.sh*. For the used example, such a list would for instance look like the one presented below.

```
#!/bin/bash

SIM_COMP_LIST=(
"/home/YOUR_USER_NAME/tutorial/sysvlog/testbench.sv"
"/home/YOUR_USER_NAME/tutorial/sysvlog/simulation.sv"
```

[2]In order to get significant power estimates, the used simulation, *simulation.sv*, should run the design in a typical scenario, i.e. with a typical toggling activity.

```
)
```

In the case that this setup is used with a design that consists of more than one SystemVerilog file, it should be considered to create a compile list for the design files as well, for instance, *designCompList.sh*. This can be done in the same manner as for *simCompList.sh*.

E.2.2 Setup of *ncsim*

Returning to *tutorial* and entering *ncsim* the basic setup has already been created. If it for some reason should be desirable to repeat the setup from ground up, please refer to [37]. The directory contains a single file, *init.sh*, and two sub-directories, *simulations* and *netlist_simulations*. For this basic introduction, *init.sh* can be ignored. The sub-directories *simulations* and *netlist_simulations* are used to respectively run simulations on the SystemVerilog files and the netlists. Entering first *simulations*, a couple of path variables have to be adopted to the used system environment. Open for this purpose first *cds.lib* and *hdl.var* and change the defined paths so that they match the used system environment.

In the same manner, open *compile.sh* and change the paths in the upper part of the script. It is worth noticing that, in the case that a compile list for design related SystemVerilog files (*designCompList.sh*, see Appendix E.2.1) has been created in *sysvlog*, *compile.sh* can easily be adapted to use this list instead of a single design file. This is done by uncomment all lines using the variable *DESIGN_COMP_LIST* and comment each line that uses *DESIGN*.

The purpose of the *compile.sh* script is to compile all the specified SystemVerilog files. This is necessary in order to simulate the files. For the here described work flow, *compile.sh* is as such rather a sub-script that is called by another script that executes the simulation (see below). However, if it is of interest to simply check the compilablity of the created SystemVerilog files, it is possible to do so by calling the script directly from the terminal by typing:

```
bash compile.sh
```

The script then compiles the specified SystemVerilog files and returns error messages to the terminal if the compiling process is unsuccessful.

Having setup the content of the *simulations* sub-directory, *ncsim/netlist_simulations* must be setup in a similar manner. Opening the sub-directory, the paths in *cds.lib* and *hdl.var* have to be changed in the same way as described above.

Further, the script *netlist_sim.sh* has to be adapted. This requires slightly more complicated changes than for the other files. First, adapt the paths described by *CDS_LIB* and *SIM_COMP_LIST*, similar to the way it has been done for *compile.sh*. Second, the netlist that should be simulated must be defined by *DESIGN_NETLIST*. Following the earlier used example, it is assumed that the netlist, *net_design.v*, will be contained in the directory *sysvlog*, after the synthesis process has been executed for the first time. Third, the used technology library has to be specified by setting *TECH_LIB*. Finally, some information has to be specified in order to extract the desired toggling activity data in form of a *tcf*-file. This includes the name of the

instances of the design and the testbench as declared in the superior SystemVerilog file.[3] For the example of *rtl_design.sv*, it is included into *testbench.sv* by using capital letters:

```
rtl_design RTL_DESIGN(.en(en),
                      .clk_in(clk_in),
                      .clk_out(clk_out));
```

As a result, the instance name of the design file has to be specified in *netlist_sim.sh* in the same manner, using capital letters:

```
export DESIGN_MODULE=RTL_DESIGN
```

In addition, a target directory has to be specified to which the simulation tool will export the toggling information in form of a *tcf*-file. This is done, by setting *TCF_OUTPUT_FILE*. Since the *tcf*-file is latter on used by *RTL Compiler*, it is reasonable to specify the *tutorial/rtc_compiler* sub-directory.

E.2.3 Setup of *rtl_compiler*

At this point, the simulation tool is completely set up and it remains to create files that are needed by the *RTL Compiler*. For this purpose, the sub-directory *tutorial/rtl_compiler* should be entered. Two files must be created in this directory.

First, create *rtl_design.tcl*. This file is later on passed to the synthesis tool and defines which operations should be executed.

For the here presented example, a working version of *rtl_design.tcl* can be found in the zip-file which is attached to this documentation[4]. However, if, for instance, another technology library or different SystemVerilog files are used, *rtl_design.tcl* has to be modified accordingly. In any case, it is important that the in *rtl_design.tcl* specified technology library and the to be created netlist (*net_design.v*) are in accordance to the in *netlist_sim.sh* specified related instances. It is worth noticing that the last four commands of *rtl_design.tcl* are commented. The motivation behind this will become evident below, when the functionality of the file is discussed in more detail.

As discussed in Appendix E.1, the *RTL Compiler* needs some information about the design constraints. This information is specified in the *constraints.sdc* file, which should be created next. However,the nature of the information contained in *constraints.sdc* depends strongly on the specific project. It is therefore out of the scope of this documentation to discuss the general content of *sdc*-files. A version of *constraints.sdc* that works for the here presented design example, *rtl_design.sv*, can be found in the attached zip-file. For other examples, please refer to [38].

[3] If the used project setup differs from the here described simulation-testbench-design hierarchy, this part of *netlist_sim.sh* has to be modified. In addition, changes have to be performed for *togsim.tcl* which is contained in the *netlist_simulations* directory. As the nature of this changes depend on the used project setup, this is not further described in this documentation.

[4] This file is in many ways equivalent to *rtl.tcl* of [38]. Both files can, if desired, be combined.

E.3 Execution

After having set up the tool environment, the in Appendix E.1 described work flow can be executed. As depicted in Figure E.1, it is first of interest to verify the logical behavior of *rtl_design.sv*. To do so, the *simulate.sh* script in the *ncsim/simulations* sub-directory can be used. This script first compiles every available SystemVerilog file (by calling *compile.sh*) and then elaborates and starts the simulation. It is simply called from the terminal by typing

```
bash simulate.sh simulation -gui
```

Note that the input "*simulation*" refers to *simulation.sv*. In the same manner, the script can be called with possible other SystemVerilog simulations, as long as they are specified in *simCompList.sh* (see the setup of Appendix E.2.1). The second input argument, "*-gui*", is optional. Using it starts a graphical user interface for the simulation, which can be used to, for example, extract waveforms. If *simulate.sh* is called without this input, *NCSim* runs the simulation in the terminal.

After having verified the correct behavior of *rtl_design.sv*, the design can be synthesized in order to generate the netlist, *net_design.v*. In order to do this, first enter the *tutorial/virtuoso* directory and type:

```
tcsh
source .cshrc
```

These commands set path variables that are needed by the *RTL Compiler*, which now can be used to generate *rtl_design.v*, by entering *tutorial/rtl_compiler*. However, before calling the *RTL Compiler*, it should be verified, that the current *rtl_design.tcl* file specifies that the netlist should be created. This is the case, if the lower part of the file looks equivalent to:

```
...
## Writes a technology dependent netlist
write -mapped > ../sysvlog/net_design.v
## Read toggle file for power estimation
#read_tcf ./simulation.tcf
## Write out area and timing reports
#report area > rtl_design_area_report.rep
#report timing > rtl_design_timing_report.rep
#report power > rtl_design_power_report.rep
exit
```

If this is the case, the synthesis step can be executed by calling:

```
rc < rtl_design.tcl
```

This commands calls the *RTL Compiler*, which executes the synthesis, while printing the status to the terminal. When the synthesis has been successful, the *RTL Compiler* reports "Synthesis succeeded" to the terminal and creates the netlist, *net_design.v*, in *tutortial/sysvlog*.

Using the netlist, it is now possible to run simulations to gather the toggling information. To do so, enter the sub-directory *tutorial/ncsim/netlist_simulations* and type:

```
bash netlist_sim.sh simulation
```

This runs the *netlist_sim.sh* script with the *simulation.sv* simulation. As for *simulate.sh*, the script *netlist_sim.sh* can easily be run with other SystemVerilog simulation files, by including them in *simCompList.sh* in the *sysvlog* sub-directory and passing their name as an input to *netlist_sim.sh*. However, in contrast to *simulate.sh*, *netlist_sim.sh* does not support the "*-gui*" input argument and always runs the simulation in the terminal.

If the simulation has been successful, *NCSim* creates the desired *tcf*-file in the *tutorial/rtl_compiler* sub-directory. The *tcf*-file has the same name as the simulation that has been used to generate it. In the case of the here given example, the *tcf*-file is therefore called *simulation.tcf*.

This file can now be used to estimate the power requirements of the design by once more using *RTL Compiler*. Note that the simulations scripts change some path variables and it is therefore necessary to once more enter the *virtuoso* sub-directory and source the *.cshrc* file as described above. After that is accomplished, the *rtl_design.tcl* file has to be slightly modified. First, since the netlist *net_design.v* already exists, it does not have to be created again and the related command can be prevented from execution by commenting it. Second, *simulation.tcf* must be passed to the *RTL Compiler*, and the corresponding command line must therefore be uncommented. Finally, uncomment the lines that correspond to desired reports. After the modification, the lower part of *rtl_design.tcl* should look similar to this:

```
...
## Writes a technology dependent netlist
#write -mapped > ../sysvlog/net_design.v
## Read toggle file for power estimation
read_tcf ./simulation.tcf
## Write out area and timing reports
report area > rtl_design_area_report.rep
report timing > rtl_design_timing_report.rep
report power > rtl_design_power_report.rep
exit
```

At this point, the *RTL Compiler* can be called, by once more typing:

```
rc < rtl_design.tcl
```

This triggers the tool to once more run the synthesis process. However, instead of generating a new netlist, the *RTL Compiler* creates three reports in the *tutorial/rtl_compiler* directory. The desired power estimates can be found in the *rtl_design_power_report.rep* file.

Appendix F

Synthesis of Post Processing Implementations

Table F.1: Synthesis results for the VNC modules

		Approach 1	*Approach 2*	*Approach 3*	*Approach 4*
Power	Leakage	10595.030 nW	7885.180 nW	7395.500 nW	7356.075 nW
	Dynamic	1330.703 nW	2116.530 nW	15344.476 nW	17000.895 nW
	Total	11925.733 nW	10001.711 nW	22739.976 nW	24356.970 nW
Cells		180	108	112	115
Area		932 μm^2	667 μm^2	678 μm^2	684 μm^2
Time slack		15806 ps	15739 ps	15529 ps	15493 ps
Max. frequency		238.1 MHz	232.6 MHz	222.2 MHz	222.2 MHz

Table F.2: Synthesis results for the *vnc_output_memory* sub-modules

		Approach 1	*Approach 2*	*Approach 3*	*Approach 4*
Power	Leakage	9485.628 nW	7037.110 nW	6547.448 nW	6508.011 nW
	Dynamic	381.501 nW	364.913 nW	11832.848 nW	13329.273 nW
	Total	9867.129 nW	7402.023 nW	18380.296 nW	19837.284 nW
Cells		160	94	98	101
Area		837 μm^2	593 μm^2	605 μm^2	611 μm^2

Table F.3: Synthesis results for the IHF modules with $n_{\mathrm{IHF}} = 64$

		$m_{\mathrm{IHF}} = 1$	$m_{\mathrm{IHF}} = 2$	$m_{\mathrm{IHF}} = 4$
Power	Leakage	34186.729 nW	34517.347 nW	34502.028 nW
	Dynamic	73121.548 nW	73568.466 nW	74010.662 nW
	Total	107308.278 nW	108085.813 nW	108512.691 nW
Cells		403	414	409
Area		2877 μm^2	2904 μm^2	2906 μm^2
Time slack		15634 ps	15631 ps	15654 ps
Max. frequency		227.3 MHz	227.3 MHz	232.6 MHz

Table F.4: Synthesis results for the IHF modules with $n_{\mathrm{IHF}} = 128$

		$m_{\mathrm{IHF}} = 1$	$m_{\mathrm{IHF}} = 2$	$m_{\mathrm{IHF}} = 4$
Power	Leakage	60394.166 nW	60305.850 nW	60778.523 nW
	Dynamic	126892.068 nW	127300.007 nW	128469.588 nW
	Total	187286.234 nW	187605.858 nW	189248.111 nW
Cells		726	719	725
Area		5083 μm^2	5081 μm^2	5121 μm^2
Time slack		15575 ps	15488 ps	15569 ps
Max. frequency		227.3 MHz	222.2 MHz	227.3 MHz
		$m_{\mathrm{IHF}} = 8$	$m_{\mathrm{IHF}} = 16$	$m_{\mathrm{IHF}} = 32$
Power	Leakage	61496.940 nW	63843.692 nW	69488.932 nW
	Dynamic	130534.049 nW	136793.082 nW	150391.854 nW
	Total	192030.989 nW	200636.774 nW	219880.786 nW
Cells		728	751	809
Area		5179 μm^2	5387 μm^2	5859 μm^2
Time slack		15571 ps	15556 ps	15554 ps
Max. frequency		227.3 MHz	227.3 MHz	227.3 MHz