Tiny True Random Number Generator

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Shashi Prashanth Karanam
Bachelor of Science
Kamala Institute of Technology and Science, 2006

Director: Dr. Jens-Peter Kaps, Assistant Professor
Department of Electrical and Computer Engineering

Spring Semester 2009
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my parents Sarala and Jayanth, and my brother Jayachand.

Acknowledgments

I would like to thank Dr. Kaps for encouraging me to pursue this project and for helping me throughout the course of this research. I would also like to thank Dr. Gaj and Dr. Hwang for their valuable suggestions and support. Special thanks to Ahmad Salman, Marcin Rogawski and all other CERG team members, and Prakash Devarapalli.

Table of Contents

List of Tables

List of Figures

Abstract

TINY TRUE RANDOM NUMBER GENERATOR

Shashi Prashanth Karanam,

George Mason University, 2009

Thesis Director: Dr. Jens-Peter Kaps

Random Number Generators (RNGs) play a crucial role in the security of modern-day cryptographic modules. In cryptography they are used to generate initialization vectors for cryptographic primitives and protocols, keys for secret and public-key algorithms, seeds for pseudo-random number generators, challenges, nonces, padding bits, and system parameters in security protocols. A True Random Number Generator (TRNG) is a physical device that generates statistically independent and unbiased bits. A TRNG harvests randomness present in the underlying physical source and the generator will have no internal state kept. Increased research interest in the field of reconfigurable computing is making Field Programmable Gate Arrays (FPGAs) a preferred platform for cryptographic implementations. Hence, a pure digital implementation of a TRNG is highly demanded by modern-day applications. This thesis describes a simple TRNG design based on a single ring oscillator implemented using pure logic gates focusing on low power, and low area cryptographic applications. The randomization technique is based on sampling phase jitter contained in the oscillator ring. The TRNG has a very low area consumption, high throughput/area ratio and generates output bits at an acceptable bit rate. The security of the cryptographic

primitives relies on the quality of the generated random bits, hence a TRNG for cryptographic applications must meet stringent requirements and should generate bits that can not be reproduced and are unpredictable in nature. The generator can be tested for its good statistical properties using a statistical test suite, ideally adjusted to a perfect RNG. Our TRNG design have been verified against statistical test suites from Diehard, NIST and BSI.

Chapter 1: Introduction

1.1 The need for Random Bits in Cryptography

Random Number Generators (RNGs) find numerous applications in a wide range of areas from art to statistics and cryptography. The degree of randomness required is determined by the type of application. Weaker forms of randomness can be associated with simpler applications (like randomly selecting a music track from the list) whereas many cryptographic applications demand a very high degree of apparent randomness. A large set of cryptographic applications [1] depend on generation of random numbers. Few of such consumers of random bits in a cryptographic module include

1. keys (Secret keys in symmetric-key cryptosystems, user private keys in public key cryptosystems, message private keys in randomized public key schemes)
2. Initialization vectors for cryptographic primitives and protocols
3. Nonces, Challenges, padding bits, and system parameters in security protocols
4. Seed for Pseudo-Random number generators
5. Cryptographic accelerators and Smart cards.

A few non cryptographic applications include

1. Games, Gambling, Lotteries and Draws
2. Simulations
3. Software testing

4. Random sampling (e.g: drug screening) and many more .

A RNG is a computational or physical device that generates sequences which are unpredictable and are random. A RNG for cryptographic applications must meet stringent requirements since the security of cryptographic modules primarily rely on the unpredictability of the keys or initialization vectors used. Thus, an adversary having complete knowledge of the design and access to previously produced bits must not be able to predict the future bits.

Bruce Schneier, in his book Handbook of Applied Cryptography [1] states a RNG may possess one or more of the following properties

1. Output is random : All the elements of the sequence are generated independently. This means it should pass all the statistical tests of randomness.

2. Output is unpredictable : Provided complete knowledge of the algorithm or the sequence generating mechanism, and all of previously generated bits, it should be computationally infeasible to determine the next random bit.

3. Output cannot be reproduced : If the sequence generating mechanism is always started from a known state and ran for multiple times it should not reproduce any of the previous sequences or everytime it should produce completely unrelated random sequences.

1.2 RNGs using Pure Digital Elements

Due to recent advances and increased research interest in the field of reconfigurable computing using very flexible high speed computing fabrics like Field Programmable Gate Arrays (FPGAs), RNGs using pure digital elements are drawing much attention. Research in this field is of primary focus today since substantial changes can be made to data path and control logic if required once designed and it is possible to adapt a new hardware structure during runtime by loading a new circuit. Also it offers huge performance advantages over traditional software based methods.

1.3 Tiny True Random Number Generators

A Tiny True Random Number Generator is important to have because

1. Of small area for area constraint applications such as RFID tags, smart nodes.
2. Of Low power for power constraint applications such as battery powered devices,and for ultra low power cryptographic applications such as sensor nodes.
3. To have high throughput per area and energy spent.

1.4 Thesis Goals

The Primary goal of the Thesis was to:

1. Design and Build a simple True Random Number generator (TRNG) in an FPGA using logic gates only.
2. Build a compact and efficient design with a very low area, high area per throughput ratio and low power.
3. Build a design with an acceptable output bit rate suitable for cryptographic applications.
4. Validate the TRNG design using statistical test suites for randomness.

The Secondary goal of the project was to:

1. Analyze the mathematical framework of a provable secure true random number generator [2] proposed by Sunar, Martin and Stinson through implementation.

This Thesis will show that we were successful in achieving the goals.

1.5 Thesis Organization

Chapter 2 of this thesis discusses the classification and explanation of different types of RNGs. Chapter 3 describes the sources of randomness for a TRNG. Chapter 4 presents the generic architecture of a TRNG. Chapter 5 describes how a RNG can be tested and presents the details of Diehard [3], NIST [4] and BSI [5] statistical test suites. Chapter 6 covers the background of TRNG designs appeared in literature targeting FPGA platforms. Chapter 7 details our TRNG design. In Chapter 8, we present the conclusion and discuss some future work on our TRNG design.

Chapter 2: Classification

This chapter discusses different categories of RNGs and classifies their randomization generation techniques.

2.1 Types of RNGs

RNGs can be classified as:

2.1.1 Pseudo Random Number Generators (PRNGs)

A PRNG is a deterministic algorithm which generates sequence of bits with little or no discernible pattern in the bits. It is a function which once initialized by a seed (random value) generates a sequence of numbers which only approximates the properties of a TRNG. Given the same seed the PRNG will always produce the same sequence of bits. These are also called as deterministic RNGs.

2.1.2 True Random Number Generators (TRNGs)

A TRNG is one which generates statistically independent and unbiased bits. These are also called as non-deterministic RNGs.

A short comparison of characteristics of PRNGs and TRNGs is presented in table 2.1. Since TRNGs take longer time to produce the same number of bits they are less efficient

Table 2.1: Comparison between PRNGs and TRNGs

Characteristic	**PRNG**	**TRNG**
Efficiency	Excellent	Poor
Determinism	Deterministic	Nondeterministic
Periodicity	Periodic	Aperiodic

in terms of throughput when compared to PRNGs. However bits produced by TRNGs are non-deterministic making them more suitable for cryptographic applications. TRNGs have no period i.e., they produce a sequence which never repeats. The security of a PRNG depends on the computational complexity of possible attacks i.e., it should be computationally infeasible to compute the next output even if all previous outputs generated from a PRNG and algorithm used are known. Hence PRNGs suitable for cryptographic applications are also known as Cryptographically Secure Pseudo Random Number Generators (CSPRNGs) and must be resistant to known attacks. PRNGs satisfy only first of the Schneier requirements of a RNG where as TRNGs possess all the three properties that are stated in section 1.1. The major difference between a PRNG and a TRNG is how its internal state is kept, in a TRNG there is no internal state kept in the generator and the bits produced are independent of previously generated bits.

There is a third type of RNGs called hybrid RNGs which can have design elements from both PRNGs and TRNGs [6], [7].

2.2 Classification of Generation Methods of TRNGs

The focus in this section will narrow down to only TRNGs. TRNGs can be further classified as

1. Software (SW) based Generators or non-physical TRNGs and
2. Hardware (HW) based Generators or physical TRNGs

The fundamental component of a TRNG is the entropy source it relies on. Entropy is defined as "the measure of uncertainty associated with a random variable".

The entropy source for a SW-TRNG [1] are random events in a computer system that can be captured using software procedures such as the

1. mouse movements and clicks
2. keystrokes

Table 2.2: Comparison between SW-TRNGs and HW-TRNGs

Characteristic	**SW-TRNG**	**HW-TRNG**
Expensive	Less	More
SW-Integration	Easy	Tough
Statistical properties	Weak	Good
Entropy	Less	High
Subject to observation and Manipulation	Yes	Yes

3. the system clock

4. content of input/output buffers, RAM content

5. operating system values such as system load and network statistics.

HW-TRNGs exploit randomness which occurs in physical phenomenon where sources are faster, higher in quality and more protected by themselves.

A comparison between two approaches is presented in the table 2.2. When compared to HW-TRNGs, SW-TRNGs have low entropy and are less robust w.r.t. observation and manipulation. Hence, SW-TRNGs usually have more than one entropy source and a strong post processing unit which can execute a drastic compression.

2.3 Classification of HW-TRNGs

The focus in this section and throughout the thesis document are HW-TRNGs which from now on we refer to as TRNGs. True Random numbers can be generated electronically using either

1. Analog components,

2. Pure digital elements, or

3. A mix of Analog and digital elements

In modern day security applications TRNGs using pure digital elements are highly preferred over analog components because they

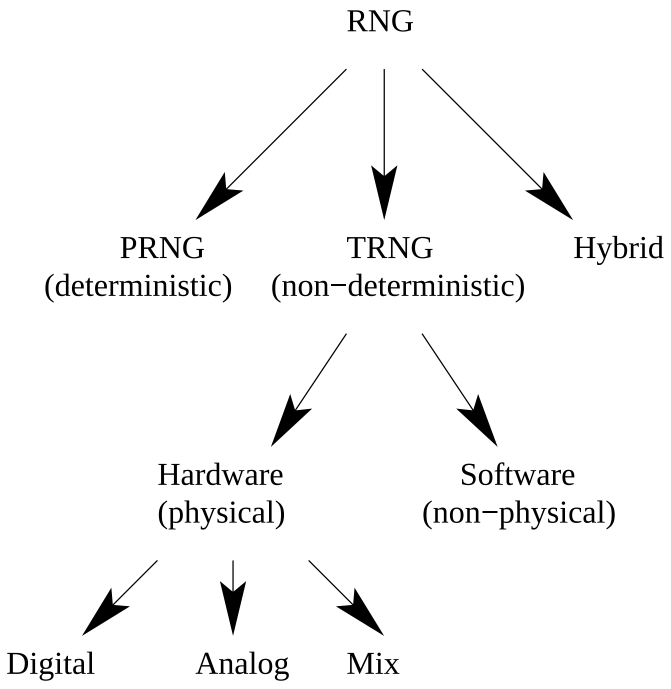

Figure 2.1: RNG Classification.

1. are cumbersome and can't be easily integrated into ASIC or FPGAs,
2. have low throughput,
3. are highly sensitive to environmental changes,
4. are less flexible and
5. operate at lesser speeds

On the contrary, digital components offer greater speeds and more robustness, high flexibility, are less subjective to environmental changes and can easily fit into ASIC and FPGA platforms. A summary of classifications discussed in this chapter is presented in figure 2.1.

Chapter 3: Sources of Randomness

In this chapter we discuss the available sources of randomness for a physical TRNG or a HW-RNG. The Entropy source for a TRNG is a physical source. Such sources of randomness include

1. **Electronic Noises:** Electronic noises can be primarily categorized into

   Thermal Noise: It is also called as Johnson noise and is generated by thermal agitation of electrons in a conductor [8]. Thermal noise is approximately white and its amplitude is random and gaussian. This kind of noise can be observed in resistors.

   Shot Noise: Shot noise normally occurs when there is a voltage differential or potential barrier. When the electrons and holes cross the barrier, shot noise is produced. Shot noise is also white and gaussian in nature. A diode, a transistor or a vacuum tube will all produce shot noise [8]. An avalanche noise is produced by an avalanche diode at a specified reverse bias voltage where an avalanche breakdown occurs, a similar effect can be observed by zener diode at zener breakdown.

   1/f Noise It is also called as Flicker noise or Pink noise [9]. It results from a variety of effects, such as generation and recombination noise in a transistor due to base current, impurities in the conductive channel. It is always related to the direct current flow. Flicker noise is more prominent in FETs and resistors.

   TRNGs using electronic noise as a source of randomness typically contains an amplifier to bring the output of the physical process into the macroscopic realm, and a sampler to convert the output into a digital signal [10].

2. Quantum mechanical properties of a photon or a nuclear decay from a radioactive substance are believed to be random in nature. They can also be used as a entropy

source for a HW-RNG.

3. **Metastability:** By violating the setup and hold conditions of a flip flop, the pair of gates internal to the flip flop which are usually cross connected will behave unpredictably or oscillate about some intermediate voltage which is neither a logical high or logical low. These oscillations die after sometime and the flip flop finally settles down into a unpredictable logical high or low.[10], [11], [12].

4. **Jitter:** Jitter is defined as the short-term variations of a digital signal's significant instants from their ideal positions in time. In general, jitter can be defined as the deviation of the timing edges from their ideal locations. It can also be seen in characteristics such as amplitude, frequency or phase of successive cycles. In frequency domain representation it is termed as phase noise which is a representation of rapid, short-term, random fluctuations in the phase of a wave caused by time domain instabilities.

   Jitter consists of two components viz., deterministic jitter and non deterministic jitter. Deterministic jitter is predictable and reproducible. The peak-to-peak value of this jitter is bounded and the bounds can easily be predicted. Few examples of this kind include data dependent jitter, sinusoidal jitter, uncorrelated jitter and duty cycle distortion. Random jitter is also called Gaussian jitter as its standard deviation grows with time and is considered to be unbounded and unpredictable. The thermal and shot noise in electrical circuits are white and have Gaussian distribution and can be considered as the source of random jitter. Edge deviations, which occur in electronic signals contain a level of randomness and are primary source of randomness for our TRNG implementation and many other TRNG implementations [13],[14],[15],[16],[17],[18],[11],[14]

   There are several ways to characterize jitter [19],[20] such as

   Phase jitter : It is the difference between the measured phase advance of the clock from an ideal clock.

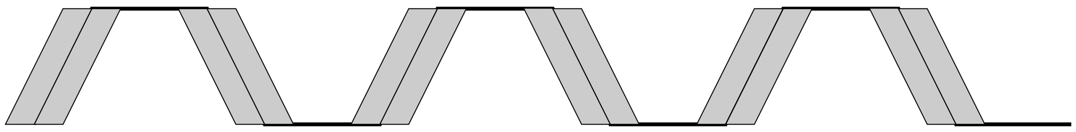

Figure 3.1: Phase Jitter

Period jitter : The measure of a deviation in a clock's period from its average period is defined as Period jitter.

Cycle-to-cycle jitter : The deviation in clock period between any two adjacent cycles is termed as cycle-to-cycle jitter.

n-cycle jitter : n-cycle is jitter is similar to cycle-to-cycle jitter but measures the deviation of clock period over n cycles.

In other words phase jitter can be defined as a measure of the relative distance at which the actual phase as shifted from the ideal clock phase and period jitter can be defined as a measure of the relative speed at which the actual period as shifted from the ideal clock period.

If a TRNG is targeted for a FPGA or is seeking a pure digital implementation then it is impossible to use first two sources of randomness.

Chapter 4: Generic Architecture

In this chapter we present the generic design of a TRNG. Quality of a good TRNG design relies on three basic components viz.,

1. a randomness source,
2. a sampler, and
3. a post processing unit. A TRNG usually follows the generic architecture [21], [13] depicted in figure 4.1.

4.1 Randomness Source

Any TRNG is necessarily based on some kind of non-determinstic physical phenomenon. The noise source is the most critical component as it determines the available entropy. The noise source generates an analog noise signal and this analog noise signal is fed into a sampler to get the digitized analog signal (das). Furthermore, some noise sources can

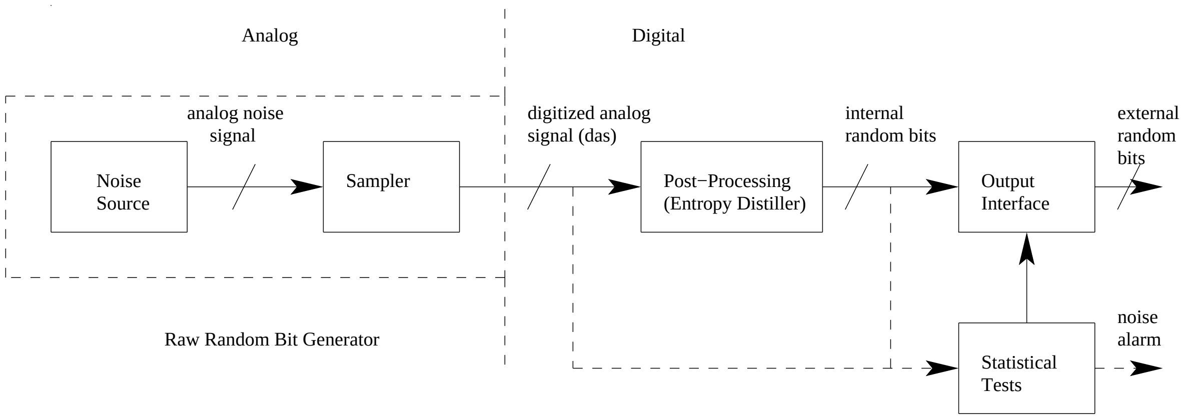

Figure 4.1: Generic Architecture.

exhibit biases in their output which should be eliminated by suitable postprocessing. The available noise sources for a TRNG are from physical process such as electronic noise, chaotic circuits, nuclear decay, metastability, jitter amongst others as discussed earlier. The most widely used technique to generate a random bit stream using digital elements is either by sampling the jittered oscillations or by harvesting the metastabilities in flip flops. Quantification of entropy, estimation of statistical properties of the available entropy, active monitoring techniques for detection of entropy source failure and long term external effects on the entropy source are the significant design tasks that are usually associated with a randomness source.

4.2 Harvesting Mechanism

The entropy source is sampled using a harvesting mechanism which does not disturb the physical process but yet collects the maximum entropy. A digitizer/sampler extracts the digitized analog signal (das) from an analog noise signal. For example a comparator or a VCO or a D flip flop can serve as the sampling unit depending on the type of the noise source used. Noise source followed by a sampler together is considered as the raw random bit generator.

4.3 Post-processor

Post processing increases the randomness of a TRNG by applying a compression function on the 'das' resulting in a lower speed output stream. For this reason it is also called as entropy distiller as it distills the entropy in the outgoing sequence. The amount of compression required depends both on the effective entropy of the source and on the efficiency of the post processing algorithm used. The probability distribution of the random bit words after post processing is much closer to a uniform distribution than that of the raw random bits words. A Post-processor also masks the imperfections in an entropy source and improves the robustness of the design. Furthermore it provides tolerance in the presence of environmental changes or tampering. A few examples of Post- processing units are Von Neumann corrector, XOR corrector, extractor function [22], Hash function (eg: SHA-1[23]), resilient functions [2], [13], etc. This component might not be needed in all designs but should be employed to strengthen the design if the sources exhibit a bias.

4.4 Statistical tests

In order to confirm the quality of the RNG it should pass a battery of statistical tests. The most common are NIST randomness tests from National Institute of Standards and Technology [4],BSI [5] from German government, and Diehard [3]. Additionally for a TRNG the question arises which random numbers should be tested? The available random numbers in a TRNG are 'das' bits and internal random numbers. External random numbers are not under the designer's control. Consider a situation where the noise source totally beaks down or a weaker noise source is used with a strong post processing unit, then testing of internal random numbers will pass any statistical test suite but these internal random numbers are deterministic, having either zero entropy or low entropy respectively. The TRNG behaves as a PRNG. Hence it is always a good practice to test the 'das' bits if the RNG permits access. Failure of the noise source the failure can then be easily detected as the output becomes deterministic.

Chapter 5: TESTING OF TRNGs

In chapter 2 we discussed the two major categories of RNGs viz., TRNGs and PRNGs. This chapter discusses the testing of those devices to validate that the source of randomness is functioning as expected. Various statistical tests can be applied to a bitstream to analyze the properties of randomness associated with the sequence and indeed the quality of a RNG that produced the bitstream can be estimated. In this Chapter we discuss hypothesis testing followed by basic statistical tests, and then tests from NIST[4], Diehard[3] and BSI[5] test suites. At the end of the chapter we also talk about Chi-square and Normal distributions.

Unbiased "fair" coin toss with sides labeled as "0" and "1" is a perfect example of a TRNG since each outcome have a 0.5 probability of being "0" or "1" . Furthermore, because the flips are independent of each other, the results of previous coin flips do not influence future coin flips. Therefore knowing the previous outcome will not help in predicting future output of the sequence. Consider an example of a 10 bit sequence generated using the above procedure which has first the 8 outcomes as "1" and the last two outcomes as "0". Even though each outcome of a RNG is equally likely there are good chances that a sequence of outputs might repeat. This sequence will fail any statistical tests we might subject to. So, if it is impossible to prove randomness, the realistic approach we can employ is to collect many bitstreams from a RNG and expose them to statistical tests. Although there are an infinite number of possible statistical tests there is no specific finite set of tests that can prove whether a bitstream is random or not. Any test/procedure that is applied determines whether the tested bitstream possess a certain attribute that a truly random sequence would likely exhibit too and detects any deviations of the tested binary sequence from randomness. The conclusion of any test is not definite but rather probabilistic.

Table 5.1: Hypothesis Testing

Situation	**Accept** H_0	**Accept** H_a
Data is random (H_0 **is true)**	No error	Type 1 error
Data is not random (H_a **is true)**	Type 2 error	No error

5.1 Hypothesis Testing

A statistical test can be formulated to test a specific

1. null hypothesis (H_0) or
2. alternative hypothesis (H_a)

Using the null hypothesis, a RNG is tested to see if it is producing random values where as under the alternative hypothesis a RNG is tested for its non randomness property [1],[4].

For each test, a relevant randomness statistic is chosen to determine the acceptance or rejection of the null hypothesis (H_0) or the alternative hypothesis (H_a). Each calculated test statistic value is a function of the data and is assumed to have distribution of all possible values. A critical value is determined by mathematical methods under theoretical reference distribution. Typically the test statistic value is compared against the critical value and a probabilistic conclusion is drawn depending on the hypothesis used.

Statistical hypothesis testing is a conclusion generation procedure that has two possible outcomes viz., the data is random (accept H_0) or the data is non-random (accept H_a).

If the data tested, exhibits the property that it is being tested for then a conclusion to reject H_0 will only occur a small percentage of time. This is called Type I error. Instead if the data tested, in truth is non random then a conclusion to reject H_a will only occur a small percentage of time. This is called Type II error. If 'S' is the test statistic value and 't' is critical value then the Type I error probability is

1. P(S>t $//H_0$ is true) = P(reject $H_0//H_0$ is true)
2. P(S$\leq t//$H$_0$ is true) = P(accept $H_0//H_0$ is true)

and the Type II error probability is

1. P(S $\leq t$//H$_0$ is false) = P(accept $H_0//H_0$ is false)

2. P(S>t//H_0 is false) = P(reject $H_0//H_0$ is false)

The probability of a Type I error is often called as level of significance of the test and is denoted by α and the probability of a Type II error is denoted by β. α denotes the probability that the test will indicate the sequence is not random when it really is random and β denotes the probability that the test will indicate the sequence is random when it really is not random.

A P value is defined as the probability that a perfect random generator would have produced a sequence less random than the sequence that was tested, given the kind of non-randomness assessed by the test. Usually test codes are written under null hypothesis, ideally if

1. P-value =1, tested sequence is perfectly random and

2. P-value =0,then the sequence appears to be completely non-random or if

3. P-value $\geq\alpha$, H_0 is accepted and the tested sequence appears to be random and

4. P-value $< \alpha$, H_0 is rejected and the tested sequence is completely non-random

5.2 Basic Statistical Tests

In [1] Menezes et.al. describes and proposes to run the five basic tests to determine whether the tested bitstream possess some specific characteristics that a truly random sequence would likely to exhibit. These tests usually require bitstreams containing more than 10,000 consecutive bits from the RNG. The tests include

1. Frequency test (Monobit test)

2. Serial test (two-bit test)

3. Poker test
4. Runs test
5. Autocorrelation test

There are several Statistical test suites commercially available for testing random outputs but only test suites from NIST [4], Diehard [3] and BSI [5] are considered . RNGmeter from the ComScire (see www.comscire.com), Crypt-X from the Queensland University of Technology in Australia (see www.isi.qut.edu.au/resources/cryptx/) are few other statistical tests amongst others. The five basic tests above are also part of either NIST or Diehard or BSI test suites, hence are explained in the respective sections when discussing the test suites.

5.3 NIST Statistical Test Suite

National Institute of Standards and Technology (NIST) founded in 1901 is a non-regulatory agency of the United States Department of Commerce. NIST Computer Security Devision specifies the security requirements that are to be satisfied by a cryptographic module. NIST released FIPS(Federal Information Processing Standard) 140-1 in January 1994 and it was revised in May 2001 as FIPS 140-2. Both FIPS 140-1 and FIPS 140-2 are certification standards containing a section covering RNGs. The current statistical test suite from NIST is 800-22 (a special publication with revisions) [4]. The Test suite is officially known as "A statistical Test Suite for Random and Psuedorandom Number Generators for Cryptographic Applications" and comprises of 15 statistical tests. NIST advises to disregard the Fast Fourier Transform test as of February 9, 2009 since they discovered a error in the test code. An overview of each test is presented below.

5.3.1 Frequency (Monobits) Test

The purpose of the test is to determine whether the number of ones and zeros in the tested bitstream are approximately the same. This would be expected for a truly random sequence.

The test estimates the closeness of the fraction of ones and zeros to 1/2. It is recommended that each tested bitstream consists a minimum of 100 bits.

5.3.2 Frequency Test within a Block

The purpose of the test is to determine whether the number of ones and zeros in the tested bitstream are approximately the same as would be expected for a truly random sequence in an M-bit block. The test estimates the closeness of the fraction of ones and zeros to $M/2$. For block size equal to 1 the test is identical to Frequency (Monobit) test. It is recommended that each tested bitstream consists of a minimum of 100 bits. The test requires an additional input which is block size M. M should be selected such that $M \geq 20$, $M > 0.01n$, and $n \geq MN$ where n is the number of bits in the bitstream and $N < 100$.

5.3.3 Runs Test

The purpose of the test is to determine the total number of runs in the tested bitstream, where a run is an uninterrupted sequence of identical bits. A run length of k means a sequence of k identical bits. Every run is bounded by a bit of the opposite value before and after. The test estimates whether that number of runs of ones and zeros of various lengths of the tested bitstream is as expected for a truly random sequence. In other words the test estimates the speed of oscillations between ones and zeros within the tested bitstream. It is recommended that each tested bitstream consists of a minimum of 100 bits.

5.3.4 Test for the Longest Run of ones in a Block

The purpose of the test is to determine the longest run of ones in the tested bitstream in an M-bit block. The test estimates whether the length of the longest run of ones within the tested bitstream is consistent with the length of the longest run of ones that would be expected for a truly random sequence. Only the test for ones is conducted since an irregularity in the expected length of the longest runs of ones implies an irregularity in the expected length of the longest runs of zeros. The size of the M is selected depending on the

Table 5.2: Minimum number of bits and block size

Minimum n	**M**
128	8
6272	128
750000	10000

number of bits (n) tested and it is recommended that each tested bitstream consists of a minimum of bits as shown in the table 5.2.

5.3.5 Binary Matrix Rank Test

The purpose of the test is to determine the rank of disjoint sub-matrices of the entire sequence. The test sequentially divides the input bit stream into n/M*Q blocks where M and N are number of rows and columns in a matrix and are equal to 32. The number of bits within the tested bitstream is n. The test discards the unused bits. Each row of the matrix is filled with successive Q-bit blocks. Ranks for 32x32 matrix are determined over the field 0,1. Other values of M and Q can be considered but new approximations need to be computed for the reference distribution. The test verifies for linear dependence among fixed length substrings of the tested bitstream. It is recommended that each tested bitstream consists of a minimum of 38 matrices i.e., n$\geq 38 * M * Q.ForM = Q = 32, n \geq 38,912$.

5.3.6 Discrete Fourier Transform (Spectral) Test

The purpose of the test is to detect repetitive patterns (periodic features) in the tested bitstream that are close to each other. Test determines if the peak heights in the Discrete Fourier Transform of the tested bitstream exceeds the 95% threshold is different than 5%. It is recommended that each tested bitstream consists of a minimum of 1000 bits.

5.3.7 Non-overlapping Template Matching Test

The purpose of the test is to determine the number of occurrences of pre-defined target substrings. The test rejects the bitstreams that exhibits too many occurrences of a given non-periodic (aperiodic) pattern. m-bit window is used to search for a specific m-bit pattern and these aperiodic patterns are pre-defined in a template library within the test code. If the pattern is not found, the window slides one bit position and if the pattern is found the window slides m-bit positions. It is recommended that each tested bitstream consists a minimum of 10^6 bits. The test requires an additional input which is the m-bit window size. The test code is written for templates of m equal to 2 through 10 and recommends to use m equal to either 9 or 10.

5.3.8 Overlapping Template Matching Test

The purpose of the test is to determine the number of occurrences of pre-defined target substrings. m-bit window is used to search for a specific m-bit pattern, The main difference between this test and the Non-Overlapping Template Matching Test the m-bit window slides only one bit position even if a match is found with the pattern. It is recommended that each tested bitstream consists a minimum of 10^6 bits. The test requires an additional input which is the m-bit window size and NIST recommends to use m=9 or m=10 although different values can be selected for m.

5.3.9 Maurer's Universal Statistical Test

The purpose of the test is to determine the number of bits between matching patterns. The test detects if the test can be significantly compressed without loss of information. A random sequence will have a characteristic number of distinct patterns, hence a significantly compressible sequence is considered to be non-random. The input bit sequence is divided into two segments viz., an initialization segment consisting of Q L-bit non-overlapping blocks, and a test segment consisting of K L-bit non-overlapping blocks and the final bits that do not form L-bit blocks are discarded. L should be chosen between 6 and 16 and Q and K should be chosen such that Q = 10*2^L and K = [n/L] - Q == 1000*2^L, n should be greater than or equal to ((Q+K)*L) table 5.3 shows the size of Q and n for different values of L.

Table 5.3: Values of L, Q and n

L	**Q**	**n**
6	640	$\geq 387,840$
7	1280	$\geq 904,960$
8	2560	$\geq 2,068,480$
9	5120	$\geq 4,654,080$
10	10240	$\geq 10,342,400$
11	20480	$\geq 22,753,280$
12	40960	$\geq 49,643,520$
13	81920	$\geq 107,560,960$
14	163840	$\geq 231,669,760$
15	327680	$\geq 496,435,200$
16	655360	$\geq 1,059,061,760$

5.3.10 Linear Complexity

The purpose of the test is to determine the length of a linear feedback shift register (LFSR) to assess the randomness of the tested bitstream. A longer feedback register implies a random sequence. The input sequence n is partitioned into M-bit N independent blocks, where n = MN. The values of M and N must be chosen in the range $500 \leq M \leq 5000 and N \geq$ 200 respectively, hence a minimum of 10^6 input bits are recommended. The test requires an additional user input for value M which should be in the above range.

5.3.11 Serial Test

The purpose of the test is to determine the frequency of all possible overlapping m-bit patterns within the tested bitstream. The test estimates the occurrences of the 2^m m-bit overlapping patterns and compares if they are approximately the same as would be expected for a random sequence. For a truly random sequence, every m-bit pattern has the same chance of appearing as every other m-bit pattern in 2^m combinations. If m is set to one the test becomes identical to Frequency (Monobits) test. The test requires input values of m and n, and they should be chosen such that $m < [log2n] - 2$ holds good. The default value of m is 16 in the test code.

5.3.12 Approximate Entropy Test

The purpose of the test is to determine the frequency of all possible overlapping m-bit and (m+1)-bit within the tested bitstream. The test estimates the occurrences of the 2^m $2^{(m+1)}$ m-bit (m+1)-bit overlapping patterns respectively and compares if they are approximately the same as would be expected for a random sequence. The test requires input values of m and n, and they should be chosen such that $m < [log2n] - 2$ holds good. The default value of m is 10 in the test code.

5.3.13 Cumulative Sum (Cumsum) Test

The purpose of the test is to determine the maximal excursion from zero of the cumulative sum of adjusted digits in the tested bitstream. Zero is adjusted to -1 and one is one. Cumulative sum is considered as the random walk. The test estimates whether the random walk of the tested bitstream is too large or too small relative to the expected behavior of the random walk of a truly random sequence. For a truly random sequence the random walk should be near zero. It is recommended that each tested bitstream consists a minimum of 100 bits.

5.3.14 5.3.14 Random Excursions Test

The purpose of the test is to determine the number of cycles having exactly K visits in a random walk. Random walk is derived from partial sums after being transferring 0 and 1 to -1 and 1. A random walk consists of a sequence of steps of unit length taken at random that begin at and return to the origin and the test estimates if the number of visits to a particular state within a cycle deviates from as what would be expected for a truly random sequence. The test is a series of eight tests for each of the states -4,-3,-2,-1 and 1,2,3,4. Hence the test outputs eight conclusions. It is recommended that each tested bitstream consists a minimum of 10^6 bits.

5.3.15 Random Excursions Variant Test

The purpose of the test is to determine the number of cycles having exactly K visits in a random walk. The test estimates if the number of visits to a particular state within a cycle deviates from as what would be expected for a truly random sequence. The test is a series of eighteen tests for each of the states -9,-8,...,-1 and 1,2,....,9. Hence the test outputs eighteen conclusions. It is recommended that each tested bitstream consists a minimum of 10^6 bits.

For all the above tests, typically level of significance α can be chosen in the range [0.001, 0.01] and the default α value in the test code is 0.01 . An α of 0.001 indicates an approximation of one sequence in 1000 sequences tested to be rejected and an α of 0.01 indicates that one would expect one sequence in 100 sequences to be rejected. For a tested sequence with a defined α equal to 0.001 or 0.01, if a P-value is greater than or equal to 0.001 then the sequence is considered to be random with a confidence of 99.9

5.4 Diehard Tests

The Diehard tests [3] were developed by Prof. Georges Marsaglia from the Florida state University and were first released on a CD-ROM in 1995. This statistical test suite is usually considered to be "the most powerful general test of randomness". The test suite consists of 15 different independent statistical tests and requires atleast 80 million bits(10-12 MegaBytes) in binary format.

5.4.1 Birthday Spacings

The name is based on Birthday Paradox [1]. Chooses random points on a large interval and lists the spacings between the points. The number of values that appear more than once in the list should be asymptotically Poisson distributed.Test is repeated for 9 times using bits 1-24, 2-25, ..., 9-32(counting from the left) from a random integer in the bitstream to set the parameter values for the test. Test returns 9 P-values, their associated mean and

Chi-Square values, and degrees of freedom [1].

5.4.2 Overlapping Permutations

This is the overlapping 5-Permutation (OPERM5) test . Analyzes sequences of five consecutive 32 bit random integers. It looks at a sequence of 1 million such 32 bit random integers. Each set of five consecutive integers can be in one of 120 states for the 5! possible orderings of five numbers.The 120 possible orderings should occur with statistically equal probability. Test is repeated twice for 1 million integers. Test returns 2 P-values and their associated chi-square values, and degrees of freedom.

5.4.3 Ranks of 31x31 and 32x32 Matrices Test

This test has two sub tests which performs Binary Rank test for 31x31 and 32x32 Matrices. The leftmost 31 bits of 31 random integers are selected from the bitstream to form a 31x31 Matrix and 32 random integers are selected to form a 32x32 Matrix and the ranks are determined over the field (0,1).Test returns a P-value, its associated chi-square value, and degrees of freedom .

5.4.4 Ranks of 6x8 Matrices Test

This is a Binary Rank Test for 6x8 Matrices. Six random 32 bit integers are chosen from the testing bitstream and a single byte from those random integers are chosen to form 6x8 Matrices. Ranks over the field (0,1) are determined for 100,000 such matrices. Test is repeated for 25 times for bits 1-8, 2-9, ..., 25-32 from integers in the bitstream. Test returns 25 P-values, their associated Chi-Square values, and degrees of freedom.

5.4.5 Monkey Tests on 20-bit words

The name is based on the infinite Monkey Theorem. Treat sequences of 20 bits as words and the test counts the number of missing 20-bit words in a string of 2^{21} overlapping 20 bit words. The number of missing words should be normally distributed. Test is repeated for

20 times and returns the number of the missing words, Z-score (standard normal variate) values and associated P-values for all the runs.

5.4.6 Monkey Tests OPSO, OQSO, DNA

This test consists of 3 sub tests.

Monkey test OPSO (Overlapping-Pairs-Sparse-Occupancy)

The OPSO test is similar to the Monkey tests on 20-bit words except it considers 2 letter words from an alphabet of 1024 letters. Each letter is determined by a specified ten bits from a 32 bit integer in the bitstream. OPSO generates 2^{21} overlapping 2-letter words from $((2^{21})+1)$ keystrokes and counts the number of missing 2-letter words.

Monkey test OQSO (Overlapping-Quadraples-Sparse-Occupancy)

The OQSO test is similar to the Monkey tests on 20-bit words except it considers 4 letter words from an alphabet of 32 letters. Each letter is determined by a specified five bits from a 32 bit integer in the bitstream. OQSO generates 2^{21} overlapping 4-letter words from $((2^{21})+3)$ keystrokes and counts the number of missing 4-letter words.

Monkey test DNA

The OQSO test is similar to the Monkey tests on 20-bit words except it considers an alphabet of 4 letters C,G,A,T determined by two designated bits in the bitstream. It considers 10 letter words. DNA generates $2^{2}1$ overlapping 10-letter words from $((2^{21})+9)$ keystrokes and counts the number of missing 10-letter words.

5.4.7 Count the 1's in a stream of bytes

Consider the bitstream under test as a stream of bytes. Hence, each byte can contain 0 to 8 1's with different probabilities. Test converts the count to five letter words Viz. A, B, C, D and E. Letters are determined by the number of 1's in a byte. There are 5^5 possible

5-letter words and the counts are made on the frequencies for each word from a string of 256,000 overlapping 5 letter words. Test returns a P-value, its associated chi-square and Z-score values, and degrees of freedom .

5.4.8 Count the 1's in specific bytes

Count the 1's in specific bytes test is similar to the count the 1's in a stream of bytes test except for a specific byte say the leftmost byte is chosen from each integer. Test is repeated for 25 times for bits 1-8, 2-9, ..., 25-32 from integers in the bitstream. Test returns 25 P-values, their associated Chi-Square, and Z-score values, and degrees of freedom .

5.4.9 Parking Lot Test

Randomly park a car of a circle radius in a square of side 100. Then try to park the 2nd, 3rd and so on, if the car overlaps an existing one then try again. After 12,000 attempts the number of successfully parked cars should follow a certain normal distribution. Test is repeated for 10 times. Test returns the number of successfully parked cars, Z-score values and their associated P-values.

5.4.10 Minimum distance Test

Place randomly chosen 8000 points in a square of side 1000. Find the minimum distance between the pairs. The square of this distance should be exponentially distributed. Test is repeated for 100 times. Test returns square distance, mean and P-values from these 100 runs but only results from test numbers = 0 mod 5 are reported.

5.4.11 Random spheres Test

Choose 4000 random points in a cube of edge 1000. Centre a sphere at each point such that large enough to reach the next closest point. The smallest sphere's volume should be exponentially distributed. Test is repeated 20 times. Test returns volume, mean and P-values.

5.4.12 The Squeeze Test

Multiply 2^{31} by random floats on [0,1) until you reach 1. The random floats are provided by floating random integers from the bitstream to get a sequence of uniform variables on [0,1). The number of iterations needed to reach 1 should follow a certain distribution. Test is repeated for 100,000 times. Test returns a P-values, its associated Chi-Square, and Z-score values, and degrees of freedom [1].

5.4.13 Overlapping Sums Test

Random integers are floated to get a sequence of uniform variables on [0,1). Add sequences of 100 consecutive floats. The sum should be normally distributed. Test is repeated for 10 times. Test returns a P-value.

5.4.14 Runs Test

Test counts runs up, and runs down in a sequence of random floats uniform on [0,1) obtained by floating 32-bit integers in the testing bitstream. Ascending(runs up) and descending(runs down) runs are counted for sequences of length 10,000. Test is repeated for 10 times. Test returns P-values for runs up and runs down.

5.4.15 The Craps Test

Test plays 200,000 games of craps, counts the number of wins and number of throws necessary to end each game. The number of wins should follow a normal distribution with mean 200000*p and variance 200000*p*(1-p) where p is 244/495. Throws necessary to complete the game can be any value between 1 and infinity but all the values above 21 are grouped under 21. Test returns chi-square value, observed and expected wins for all throws from 1 through 21 and also returns a cumulative chi-score value and a P-value. Test also returns degree of freedom [1].

Most of these tests run KS (Kolmogorov-Smirnov) test on the obtained P-values. KS test determines if two data sets differ significantly, it is a form of minimum distance estimation between two data sets. This test takes the advantage of making no assumption about the distribution of the data (non-parametric and distribution free). The KS test also return a P-value.

When the bitstream really fails big test returns P-values of 0's or 1's to six or more places. If the bitstream contains truly independent random bits then P-values uniform on [0,1) are returned.

5.5 BSI Tests

The German office for IT security (Bundesamt für Sicherheit in der Informationstechnik) (BSI) has published evaluation methodologies AIS 20 for PRNGs and AIS 31 [5] for TRNGs. One of the important features of AIS 31 is it focusses on the verification of the minimum entropy requirement of the random source under evaluation. Target applications usually have requirements on the properties of internal and external random numbers on the basis of different possible attack scenarios. In consideration with these circumstances AIS 31 defines two functionality classes viz., P1 and P2 that describes the strengths of the TRNG. A P1 class TRNG is required to produce an output that is statistically inconspicuous. A P2 class TRNG requirements guarantee that it is practically impossible to determine random numbers even if the predecessors or successors are known i.e, as stated in [5] "The prospects of success for systematic guessing of the external random numbers(realized through systematic exhaustion attacks)-even if external random number sub-sequences are known-should at best be negligibly higher than would be the case if the external random numbers had been generated by an ideal random number generator".

The standard further describes a number of sub-classes for the two functionality classes P1 and P2 that identifies the class specific requirements. One of the sub-requirement suggests a mechanism for detecting the total failure of the physical noise source and is referred to as "tot(total failure) test". To verify the specific properties of functionality classes P1 and P2 BSI includes the following statistical tests.

1. T0 : A disjointness test
2. T1 : A monobit test
3. T2 : A poker test
4. T3 : A run test
5. T4 : A long run test
6. T5 : A autocorrelation test
7. T6 : A uniform distribution test
8. T7 : A comparative test for multinomial distributions
9. T8 : A entropy test

The first six tests (T0-T5) are run on the internal random numbers and the tests(T6-T8) are run on the 'das' bits. Tests T1-T4 are directly taken from [24] along with the rejection limits. Brief mathematical description of the tests can be found in [5]. The test codes are written in Java. A GUI of test suite is available in German language.

5.6 Chi-square and Normal Distributions

Chi-Sqaure and Normal distributions are most widely used theoretical probability distributions in statistics.Most of statistical tests from Test suites [4],[5],[3] use Chi-square and normal distributions as reference distributions.

5.6.1 Chi-Sqaure Distribution

The Chi-Sqaure distribution is used to compare the goodness-of-fit of the observed distribution to the expected distribution under a hypothesized distribution. If squares of 'v' independent standard normal distributed random variables i.e., normal distributed variables with mean 0 and variance 1 are summed then the random variable X is distributed according to the Chi-Sqaure distribution with 'v' degrees of freedom.

Definition 5.6.1: Let $v \geq 1$ be an integer. A (continuous) random variable X has a χ^2 distribution with v degrees of freedom if its probability density function is defined by

$$f(x) = \begin{cases} \frac{1}{\gamma(v/2)2^{v/2}} x^{(v/2)-1} e^{-x/2} & : \quad 0 \leq x < \infty, \\ 0 & : \quad x < 0, \end{cases}$$

where Γ is the gamma function. The mean and variance of this distribution are μ = v and σ^2= 2v-01

The gamma function is defined by: $\Gamma(t) = \int_0^\infty \mathrm{x}^{t-1} e^{-x} dx, fort > 0.$

For example the set of parameter values with v =5, α (level of significance) = 0.01 and x = 11.0705 means that if random Variable X has Chi-Sqaure distribution with 5 degrees of freedom then X exceeds 11.0705 about 1% of time.

5.6.2 Normal Distribution

Normal distribution also known as Gaussian distribution is a continuous probability distribution that describes events around a mean. If large number of independent random variables having same mean and variance are summed then the random variable X attains the normal distribution.

Definition 5.6.2: A (continuous) random variable X has a normal distribution with mean μ and variance σ^2 if its probability density function is defined by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} exp \frac{-(x-\mu)^2}{2\sigma^2}, -\infty < x < \infty (5.1)$$

Notation: X is said to be $N(\mu, \sigma^2)$. If X ix N(0,1) then X is said to have a standard normal distribution.

The graph of the associated probability density function is a bell shape curve, with a peak at the mean and symmetric about the vertical axis i.e., $P(X > x) = P(X < -x)$ for any x. For example the set of parameter values with α (level of significance) = 0.01 and x = 11.0705 means that if random Variable X has standard normal distribution then X exceeds 11.0705 about 1% of time.

Chapter 6: TRNG Background

Randomization techniques find applications in many modern day applications and are especially critical for cryptographic modules. Electronically generating random bits has been attempted for many years. Thus far a large number of designs appeared in patents, industry and academia. The designs vary significantly depending on when they were invented, the type of entropy sources and harvesting techniques they employ, and the destination application. In this chapter we discuss the TRNG designs suitable for FPGA platforms and a few ASIC implementation designs which can also be targeted onto an FPGA platforms easily. Design principles are based either on jitter or metasatbility or both.

6.1 The Fischer-Drutarovsky Design

The design [14] was the first TRNG proposal targeting FPGAs and highlights the significance of TRNGs for reconfigurable platforms. The design samples the on-chip jitter in an analog Phase-Locked loop (PLL) implemented in the digital Altera Field Programmable Logice Device (FPLD) APEX EP20K200-2X.

This architecture consists of an on-chip PLL, multiple D flip flops, a xor gate and a decimator. In analog PLLs, various noise sources cause fluctuations in frequency of the Voltage controlled oscillator giving rise to intrinsic jitter. This jitter of the clock signal generated by the on-chip PLL is sampled via delay cascaded samplers. The multiple samples are obtained by delayed sampling at regular intervals. The samples are then XOR-ed and downsampled using decimator to extract the random bits. The key point behind multiple sampling is to not miss the sampling near the transition zone that is influenced by the jitter which according to [14] is of the order of only several tens of picoseconds. Hence the output obtained after the XOR will be uncertain. The reported implementation has a throughput close to 70 Kbits/sec and is verified using the NIST[4]tests for statistical behavior.

6.2 The Tkacik Design

The entropy source in this design [25] is the jitter in the ring oscillators. A ring oscillator consists of an odd number of inverters connected in a feedback loop to form a ring. This TRNG is based on a linear feedback shift register (LFSR), and a cellular automata shift register (CASR). Each shift register is clocked by a free running independent ring oscillator. The LFSR has 43 bits, and a characteristic primitive polynomial of $x^43 + x^41 + x^20 + x + 1$ which gives a cycle length of $2^{43} - 1$. The design uses 37-bit CASR which has a maximal length of $2^{37} - 1$. Since the cycle lengths of the two deterministic circuits are relatively prime, the cycle length of the combined generator is close to 2^{80}. To generate a random number, selected 32 bits of the LFSR and CASR are permuted and XORed together. The output is sampled only when a new number is requested and the TRNG outputs 32 bits at a time. The entire RNG design is written in Verilog RTL except the RO's which are netlists of odd number of inverters manufactured in certain process technology. However the whole design can be easily targeted onto on an FPGA.

The design has very poor performance and failed the Diehard [3] , NIST 140-1[4] and the Crypt-X tests miserably when it used only LFSR or CASR. The TRNG has far better statistical behavior when used combined LFSR and CASR design. Slight bias is observed in the output bits because of the missing all zero pattern, on the order of 2^{-43} and 2^{-37} when used LFSR and CASR alone respectively, but this bias drops close to 2^{-80} when used a combined generator .

This TRNG has an internal state, and hence it can not be assured that the sequences of numbers it generates are not repeatable although it has a high cycle length. The sources that contribute to the random behavior of the design along with the jitter in the ring oscillators are the frequency variations of the two ring oscillators with variations in temperature and voltage, and the uninitialized LFSR and CASR at power up. The author in [25] states that this TRNG has been used with minor variations at Motorola for a number of years.

In [26] Dichtl attacked this design based on two weaknesses

1. The design uses a combination of TRNG and PRNG elements where PRNG elements are seeded with two low entropy oscillators.

2. The design uses linear components, and assuming that the adversary having access to earlier bits one can build a linear model to attack it.

Schindler further analyzes this TRNG under a formulated stochastical model. He develops lower and upper entropy bounds on the random output bits and suggests to sample 60,000 times more slowly because the amount of state information the TRNG elements outputs at top sampling rates exceeds the amount of entropy.

6.3 The Epstein et al. Design

This design [18] focuses on digital circuits that exhibit the properties of metastability and jitter. The basic element (BE) of the design consists of two inverters and two multiplexers. Using multiplexers the inverters can be configured such that they form two independent free running ring oscillators in meta-stable mode or they can be cross connected to form a bi-stable memory device. The multiplexers serve as the switching element as well as the delay element. The randomness is derived from the condition created when switched from meta-stable mode to bi-stable mode. At this point the logical state of the circuit is determined by the relative and absolute values of the instantaneous output voltages and internal noise.

The implemented prototype chip was manufactured using a 0.18um CMOS technology and consisted of 8 zones. Each zone had 9 different styles. Each style had 15-31 different varieties with a total of 247 distinct BEs. Each BE differs from the others in terms of propagation delay of the inverters and the delay elements used. The output of the varities is fed into a style multiplexer along with the XOR-ed output of the varieties. The outputs of all the style multiplexers are further fed into a Zone multiplexer along with the XOR-ed output of the style multiplexers. Selection of various BEs is designated through multiple levels of multiplexers giving a choice of selection from

1. A specic variety of a specic style of the RNG.
2. The XOR values of all the varieties of a specic style.
3. The XOR value of specic variety combining all styles.
4. The XOR value of all varieties of all styles can be selected.

The output of the zone multiplexer is further fed into a synchronizer. A synchronizer is a series of three flip flops which is used to capture the random bit. Three flip flops in series compensate for any of the metastable behaviors from the previous flip flops. So many different varieties are laid out to conrm the randomness of the output against environmental conditions and variations in manufacturing technology. The design is verified using Diehard [3] tests after being postprocessed by the Von Neumann corrector.

6.4 The Kohlbrenner-Gaj Design

This technique uses the intrinsic jitter contained in digital circuits as the entropy source. The design [15] uses only the configurable logic blocks (CLBs) common to all FPGAs and is designed to perfectly match the CLB architecture of a Xilinx Virtex-II FPGA. The proposed design consists of two independent and identically configured ring oscillators, a sampler unit, and a controller.

Each of the ring oscillators consists of a buffer between two transparent latches, and an inverter in a feedback loop. The ring oscillator is built into a CLB. The oscillator frequency is determined by the delay elements on the path i.e., two lookup tables, four multiplexers and two memory cells. The oscillator signal passes twice through the CLB and is inverted in only one of the passes. The output of the ring oscillator is taken from the buffer and is a stream of regular pulses running at frequency of 130 MHZ.

The two ring oscillators each supply a stream of pulses to the sampler unit. The sampler unit uses one such clock signal to sample the other clock signal. The frequency of the two clock signals is chosen to be close but not identical. For this reason the authors found that it is important to place the two ring oscillators close to each other to compensate for the temperature differences on the chip.

The stream of samples consists of a run of ones and a gap of zeros. The output random bit is counted modulo 2 of the length of this run and gap. To eliminate any biases in the produced output, the TRNG output is also postprocessed with a simple XOR corrector. The design has on-chip self testing capability to halt bit output on failure of the source of randomness. Design can produce speeds of up to 0.5 Mbits/second. 1 Gbits of random data was verified using the NIST statistical test suite [4].

6.5 The Bucci-Luzzi Design

This approach [16] introduces the concept of stateless generator and focuses on the verification of a minimum entropy requirement for the noise source. The authors propose the designs with reset circuits to clear the state of the TRNG. The TRNG is restarted before the collection of each output bit in order to avoid any dependencies between the collected bits. If the TRNG generates truly independent bits then no post processing is required or if the generator is stuck in a fixed bit then any biases in the output can be eliminated by using a stateless post processor. Hence fast noise sources with a low entropy per bit can be adopted if sufficient compression factor is chosen in the post processing.

The authors propose two examples of stateless RNGs, one based on two free running oscillators and the other based on a chaotic circuit as a noise source [16]. The former is more adaptable to reconfigurable platforms, hence we present only the details of the first design. It has two free running oscillators and a D flip flop serving as the sampler. The two oscillators are chosen to oscillate at different frequencies where a slow running oscillator samples a fast running oscillator. Both oscillators are stopped after each bit generation to avoid any phase shift between the ring oscillator frequencies. Every time the oscillators are stopped the D flip flop is also reset to eliminate any dependencies from the flip flop's current state.

The relevant figure of merit in this approach becomes entropy/second instead of the entropy/bit as the output quickly diverges from the start state into an unpredictable state.

6.6 The Dichtl and Golic Design

This design [27], [28] introduces a new concept of ring oscillators based on Linear feedback shift registers (LFSRs). The authors describe two new structures of ROs viz., Fibonacci Ring oscillator (FIRO) and Galois Ring oscillator (GARO). Random delays and transition times of the logic gates in the circuit are the primary sources of randomness.

A FIRO consists of a number of inverters connected in cascade in such a way that the output of each inverter with the exception of the last inverter is used as the input to the next inverter and the output of the last inverter is used as feedback to the first inverter. The feedback path consists of XOR gates and switches. The output of the previous inverter is XOR-ed with the feedback signal before it is given as input to the next inverter. Switch is closed and XOR gate is present only if the output of a previous inverter is '1' else the feedback signal propagates. Basically, the structure is identical to an LFSR except the delay elements are replaced by the inverters.

A GARO consists of a number of inverters connected in cascade in such a way that the output of each inverter, except the last inverter is XOR-ed with the feedback signal which forms input to the next inverter. The output of the last inverter directly defines the feedback signal. The feedback path consists of switches. Switch is closed and the XOR gate is only present when the feedback signal is '1'.

The feedback coefficients can be conveniently represented by a binary polynomial for both the oscillators.

$f(x) = \sum_{i=0}^{r} f_i x^i \qquad where f_0 = f_r = 1$ and r is an odd number of inverters.

It is important that the FIRO and GARO are not stuck in a single fixed state. The necessary and sufficient conditions for the oscillators not to be stuck in a single fixed state are given in theorems 6.6.1 and 6.6.2.

Theorem 6.6.1 : A FIRO does not have a fixed point if and only if
f(x) = (1+x)h(x) and h(1)=1

Theorem 6.6.2 : A GARO does not have a fixed point if and only if
f(1) = 1 and r is odd

From above, for GARO 'r' should be an odd number and for FIRO 'r' can be even or odd, but it is necessary that $r \neq 2$.

To increase the randomness and robustness of the design the authors propose a Fibonacci-Galois Ring Oscillator (FIGARO) which simply XORs the output of FIRO and GARO and samples the XOR output. The length of the two oscillators have to be chosen mutually prime to each other in order to maximize the period of the random sequence, and to minimize the interlocking and coupling effect between two oscillators. The design also uses two level sampling to reduce any bias introduced by the sampling flip flop. The authors propose to use a self controlled LFSR [27] as postprocessing unit to eliminate biases in the output. The authors say that they are able to generate statistically independent bits at a speed of 6.25 Mbits/s with a slight bias of zeros of about 0.0056 by restart method for a FIRO implementation. In restart method, oscillators are always restarted from the same initial conditions. To extract a random bit they wait the time it takes to observe a bit change in the otherwise pseudo-random bitstream. However it is not known from [27], [28] if the design was verified using any statistical test suites.

6.7 The Vasyltsov et al. Design

The proposed design [11] uses the metastability phenomenon in digital circuits and applies it to a traditional ring oscillator. The design consists of an odd number of inverters that can either be configured as a traditional ring oscillator while in generation mode or a single inverter in a feedback loop while in metastable mode. It has the same number of multiplexers as that of inverters, and a control clock generator to switch between the generation and metastable modes. D flip flop functions as a sampling unit, and a delay component is used to synchronize the sampling process with generating random data process that has a pre-defined delay.

Initially the design is chosen to operate in metastable mode where each inverter forms an independent noise source. The output voltage converges to metastability level and stays there as long as required. when in this mode the output voltage stochastically fluctuates around the metastable level due to inherent thermal noise. After a while, the system is switched to operate in generation mode where the inverters are now connected to form a traditional RO. Now the momentary voltages inside the RO are random because the value of each inverter output was defined by random noise in the previous metastable mode. After sampling a random bit in the generation mode the TRNG is switched back to meta stable mode again.

The new entropy accumulation technique uses the metastability phenomenon in the ring oscillator, compared to the traditional methods of sampling the jitter in the ring oscillators. The design has been verified using AIS.31 [5] and FIPS 140-1/2 [4] statistical tests.

6.8 The O'Donnell-Suh-Devadas Design

This design [29] uses the metastability properties of digital elements as a source of randomness. The RNG is build around physically unclonable functions (PUFs). A PUF is a function that maps a set of challenges to a set of responses based on a complex physical system. The physical system contains many random components that are introduced during the manufacturing process and cannot be controlled. This design uses Silicon PUFs (SPUFs) which generate their responses based on the hidden timing and delay information of the integrated circuits along with the challenge value supplied.

The design consists of a PRNG, a register, a SPUF delay circuit which consists of multiplexers and a Gated D-Latch, and a Von Neumann corrector. The SPUF delay circuit accepts an n bit challenge and forms two delay paths in 2n different configurations. It computes 1-bit output by measuring the relative delay difference between two paths that are of same lay-out length. The delay paths or the switch components are implemented using a pair of 2x1 multiplexers. To generate a response bit, two delay paths are excited simultaneously to allow the transitions to race against each other. The gated latch at the end measures which rising edge arrives first and sets its output to '0' or '1' accordingly.

The PUF-RNG design searches for meta-stable challenges by repeatedly applying a challenge and checking if an unstable output is obtained. For certain challenges, the setup and hold conditions of the sampling circuit are violated causing the gated D-latch to enter a meta-stable state. Hence, the response from these challenges are used to generate the random bit. The response to the challenge is not only dependent on the physical system, but is also dependent on other characteristics such as temperature and voltage. Therefore every time a random number request is made, it is confirmed that the challenge being used is still unpredictable if not the current challenge is fed into a PRNG to create a new challenge to be tested. The local register is used to save the unpredictable input challenge.

The above design methodology was implemented using AEGIS secure processor [12] which contains a PUF. The PUF is used to generate a 32-bit random number. A 64 stage PUF delay circuit has been fabricated and tested in TSMC's 0.18um, single-poly, 6-level metal process. The design needs post-processing due to bias and uses the Von Neumann corrector. The PRNG, Von Neumann corrector, and the the meta-stable challenge searching technique is implemented in the software. The PUF RNG has been verified using the NIST test suite [4].

6.9 The Schaumont et al. Design

This design [30] implements both a delay-based PUF and a jitter-based TRNG using ring oscillators by sharing and reusing a significant amount of hardware resources. The PUF design is based on a ring-oscillaor PUF proposed by Suh et al.[31] and the TRNG design is based on jitter obtained from ring oscillators as proposed by Sunar et al [2]. It has the same implementation structure as the design implemented in [17].

The design is targeted for a Xilinx Spartan3S500E FPGA device. Each ring oscillator of the TRNG implementation has 3 inverters. Designs consisting of 32 rings, 64 rings and 128 rings were implemented. Each output from the ring oscillators is sampled by an extra flip flop and is then fed into a binary XOR tree. The output of the XOR tree is then sampled to extract a random bit. The PUF output is created by pair-wise comparison of the ring oscillator frequencies. The challenge is the choice of a ring oscillator pair and the response is the comparison result of both ROs. The choice of RO is done using an n-bit wide memory-mapped decode register. A random bit is extracted by feeding the signal from RO directly to a PUF counter. The design uses only one counter, the frequency comparison operation to evaluate response of a PUF is handled in software. The selection between TRNG mode/PUF mode is made by using 2x1 multiplexers. The design can generate a random output stream at 3.2 Mbps. It has a 31-bit unique device signature (authors in [30] define uniqueness as a measure of how clearly a PUF can distinguish one FPGA from another). The TRNG outputs have been verified using Diehard[3] and NIST[4] test suites.

6.10 The Sunar et. al Design

A simple design shown in figure 6.1 is proposed by Sunar et al.[2]. The source of randomness is the phase jitter. The design aims at sampling the jittered oscillations. A ring oscillator consists of an odd number of inverters connected in a feedback loop to form a ring. The design consists of 'r' ring oscillators and 'i' inverters in each ring oscillator. The outputs from 'r' ring oscillators are XOR-ed and sampled at an independent clock frequency using

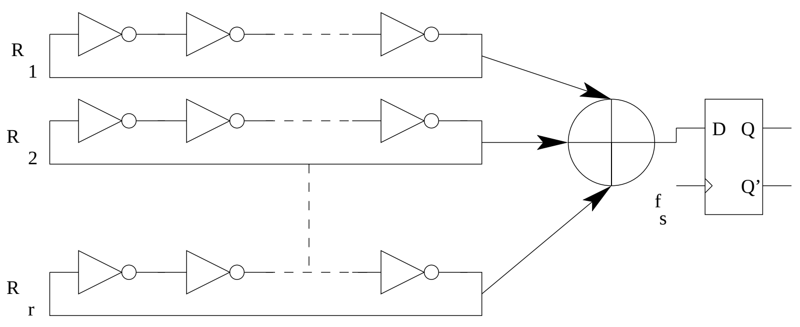

Figure 6.1: Ring Oscillator Design.

a D-type ip op to produce 'das' bits.

Ideally, the period of the wave depends on the number of inverters and the delay of a single inverter. Due to the feedback path in the oscillator ring, the output of the inverters will oscillate from a logical one to a logic zero and back. The key idea is to populate the spectrum (output waveform) with jitter events (transition zones) and then sample. Populating the whole spectrum with jitter events requires too many ring oscillators. The authors built an urn model[2], and calculate the numbers of oscillator rings (r) required to achieve a certain fill rate in the spectrum, at a certain confidence level. Fill rate is defined as the portion of the spectrum that is filled with random events. A compromise with the ll rate is to allow a fraction of the spectrum to be deterministic and a compensation for this is done through postprocessing. Occasionally transition zones from different ring oscillators overlap. As the number of oscillator rings increase, chances of having multiple oscillators with identical phase increase which in turn decreases the available entropy. Because of the compromise the output from the XOR will have transition zones and deterministic regions. To filter out the deterministic bits obtained by sampling the unfilled portions of the spectrum the authors in [2] recommend to use a resilient function of appropriate strength for post-processing of the TRNG output. They show a simple technique to obtain a resilient function from linear cyclic codes.

The authors in [2] provide the mathematical background for the generation of true random numbers in their design and analyze the quality of the output of the TRNG based on a set of assumptions at the input.

Schillekens et al. [13] implemented three TRNGs based on the above methodology on an Xilinx Virtex-II FPGA. In [13] authors chose designs consisting of 110 ring oscillators with 13 inverters in each ring oscillator, 110 ring oscillators with 3 inverters in each ring oscillator, and 210 ring oscillators with 3 inverters in each ring oscillator respectively. The reference design implementation was able to produce a throughput of 2.5 Mbits at a sampling frequency of 40 MHZ after post-processing. A Resilient function based on linear cyclic code [256,16,113] as suggested by [2] was used for the post-processing. The output sequence is verified using the Diehard and NIST tests.

The design in [2] received criticism from Dichtl and Golic [28] in terms of assumptions used. Among the criticisms are the unrealistic probabilistic modeling of the jitter, independence among the ring oscillators, unrealistic speed of the XOR gate, high sampling rate, and violation of set-up and hold times for sampling flip flop. Most of these criticisms will be addressed chapter 7.

In [17] Knut Wold and Chik How Tan proposed an improvement to the Sunar's design that does not require post-processing. Ditchl and Golic pointed out in [28] that 114 ring oscillators each running at a frequency of 40 MHZ would lead to an input frequency of 4.6 GHZ which is too fast for the XOR gate. To cope with the problem the authors in [17] introduce an extra flip flop after each ring oscillator as can be seen in the figure 6.2. The output's from these flip flops are then XOR-ed and then finally sampled using a D flip flop to extract a random bit. All the extra flip flops and the sampling flip flop are run at same sampling frequency. The implementation produced a output stream at a rate of 100Mbps with a sampling frequency of 100MHZ using 50 ring oscillators with 3 inverters in each ring oscillator and without any post-processing. The above design was implemented on an Altera Cyclone II FPGA and the output is verified using Diehard and NIST tests.

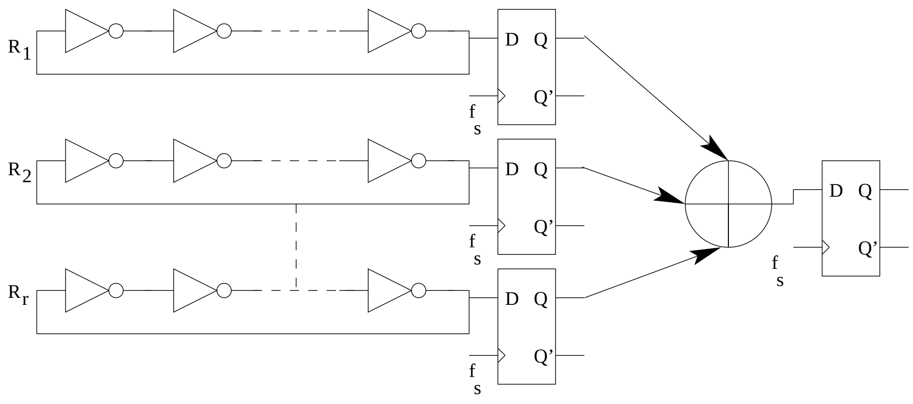

Figure 6.2: Ring Oscillator Design with a Flip Flop after each Ring Oscillator.

Chapter 7: Tiny True Random Number Generator

Our Tiny True Random Number Generator is based on the design presented by Sunar et al. in [2]. The design can be seen as an sequential approach to the design proposed in [2] with the introduction of the flip flop after the ring oscillator. We wanted to design and build a tiny and simple TRNG for area and power constraint applications such as RFIDs, sensor nodes, smart cards. Therefore we started from the hypothesis that Sunar et al. design can be serialized.

7.1 TTRNG Design

Our design shown in figure 7.1 consists of a single ring oscillator, a two input XOR gate, two D flip flops and a interface unit. The output of the ring oscillator is sampled by the D flip flop (DFF1). The advantage of this flip flop is that the signals on the input of the XOR gate are synchronous with the sampling clock and are updated only once in the every sampling period. The sampled bit from DFF1 is XORed with the over sampling bit and stored in another D flip flop (DFF2). After n-times overlapping, the resulting raw random bit also called das is stored in DFF3 of the interface unit.

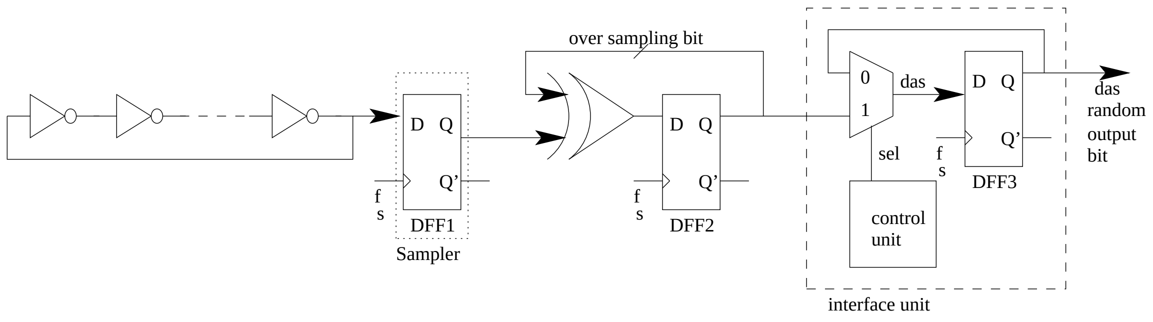

Figure 7.1: Our Design.

The interface unit consists of a control circuit, a 2x1 multiplexer and a D flip flop. Control unit is a counter that generates an active high enable signal after every 'n' iterations indicating that a random bit is ready. This enable signal drives the select line of the Multiplexer. Multiplexer helps to keep the current random bit in the register (DFF3) until the next random bit is generated. Size of the register (Dff3) depends on the output interface. collection mechanism employed and the control unit is modified accordingly to generate all the required control signals if need be. The whole design is clocked at the sampling frequency 'fs'. Choice of parameters for the implementation such as number of inverters in a ring oscillator, 'n' (loop number), and sampling frequency 'fs' is presented in section 7.4.

The Sampling frequency of our design is always chosen to be slower than to the ring oscillator frequency such that we do not over sample from the same period. Also, the sampling frequency is chosen such that ring oscillator frequency is not a multiple of sampling frequency in order to avoid sampling only at the edges.

7.2 Tools

We described TRNG designs in VHDL and used Xilinx ISE 9.1 and Xilinx EDK 9.1 for their implementation.

7.2.1 Implementation platform

The design and testing of TRNGs was done on Virtex-II system board from Memec Design [32] provided by CERG (Cryptographic Engineering Research Group). The Virtex-II system board consists of one XC2V1000-4FG456C FPGA, one 16Mx16(32MB) DDR memory, one XC18V04 ISP PROM (flash memory), two On-board oscillators, RS-232 port, JTAG port, user push button switches, LVDS transmit and receive ports, and additional user support circuits.

The Virtex-II [33] architecture is optimized for high-density and high performance logic

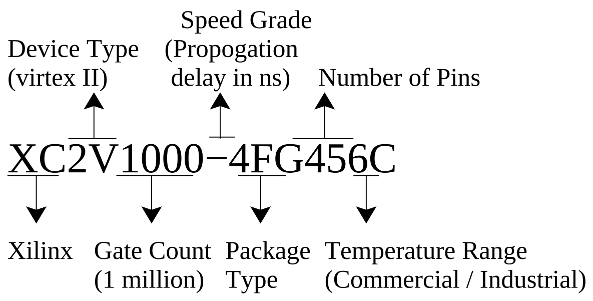

Figure 7.2: Virtex II.

designs. The programmable device consists of input/output blocks(IOBs) and internal configurable logic blocks (CLBs). CLB resources include four slices and two thistate buffers, and provide functional elements for combinatorial and sequential logic. Each slice has two function generators (FG) that can be configured as 4-input look up tables, or as 16-bit shift register, or as 16-bit distributed select RAM memory, and two storage elements which are either edge triggered D flip flops or level sensitive latches. Every CLB is connected to a switch matrix to access routing resources. The FPGA has 18-bit x 18-bit dedicated multipliers, 18 Kbit dual port RAM and Digital Clock Manager (DCM). DCM provides clock multiplication, division, phase shifting and delay compensation.

The two Oscillators of the system board run at 100 MHZ and 24 MHZ. Virtex-II FPGA has active low reset, and two user push button switch inputs that can be used to generate an active low signal. Programming the design into PROM allows for quick download of revisions of the design eliminating the requirement to download the bit file every time the FPGA is restarted. The XC18V04 PROM can store designs upto 4MB or 2^{22} bits.

7.2.2 Random Number Collection

In order to verify out TRNG design we ran several suits of statistical viz., Diehard, NIST and BSI. Some of those tests requires atleast 80 million bits. Therefore, we created a hardware system with a MicroBlaze 32-bit soft core processor to store generated random bits into 32 MB DDR external memory using the Embedded Development Kit(EDK). MicroBlaze is a virtual microprocessor that is built by combining blocks of code called cores inside a Xilinx

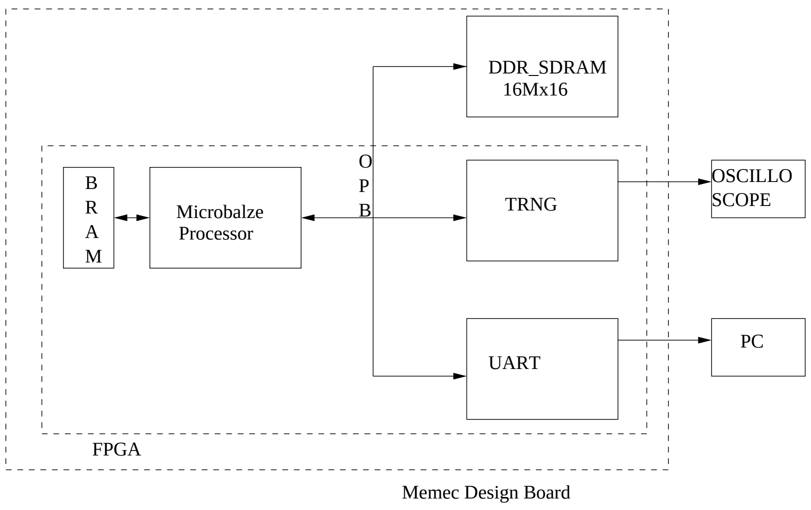

Figure 7.3: Bit Collection Mechanism.

FPGA. It has a 32-bit Harvard Reduced Instruction Set Computer (RISC) architecture with a 32x32 bit LUT RAM based register file.

Base System Builder in XPS is used to create an embedded processor project by using selected Xilinx Intellectual Property cores for the processor, UART and SDRAM interface. We added our TRNG design as a user-defined custom peripheral (TRNG design) to that processor system as can be seen in figure 7.3. The On-chip Peripheral Bus (OPB) connects on-chip and off-chip peripherals and memory, and our TRNG to the Microblaze processor.The proceesor interacts with our TRNG peripheral through software. The random bits our TRNG generates are stored in one 32-bit register. The control circuit of the TRNG generates a control signal whenever the register is full and the random bits in the register are copied into the external memory through software program.

The Xilinx EDK generates the bit file and allows for programming the FPGA. A cable adhering to the IEEE 1149.1 or Joint Test Action Group (JTAG) is used to download the bit file generated from the Hardware Description Language (HDL) onto FPGA device. We use an RS-232 interface to transmit the random bits stored in DDR memory to a Personal Computer. we forward internal signals of our TRNG to ports on the Virtex II in order to verify required intermediate signals on an oscilloscope.

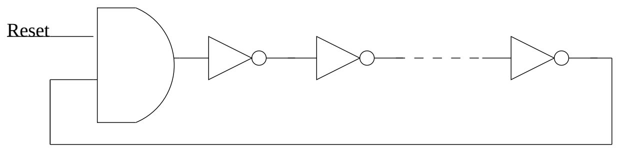

Figure 7.4: Ring Oscillator with a reset.

7.2.3 Ring oscillator implementation issue

Xilinx tools can not synthesize/implement a ring oscillator consisting of only an odd number of inverters. Hence, a 'latch' or a 'nand' gate, or a 'and' gate needs to be added in the loop. We chose to use a 'and' gate so that we can reset the TRNG externally with the same polarity as the microbalze desired. The implemented ring oscillator is shown in the figure 7.4. Introducing an 'and' gate does not alter the behavior of the ring oscillator but it adds some extra delay and thus increase the period a bit. With the addition of an 'and' gate and by using ISE advanced synthesis options it was possible to get the ring oscillator synthesized. However during mapping stage of the implementation all logic has been removed due to optimization. Therefore, we used UNISIM libraries, with 'keep' and 'INIT' attributes, 'synthesis translate-on' and 'synthesis translate-off' directives in the design to prevent logic optimization on the ring oscillator. UNISIM libraries contains descriptions for all the device primitives, or lowest-level building blocks.

7.3 TRNG implementation results

7.3.1 Sunar et al. Design

we implemented the Sunar et al. design with 114 ring oscillators and 13 inverters in each ring oscillators as presented in [2] as sample design. The authors in [2] aim at obtaining a fill rate of 0.60 at a confidence level 0.99. A resilient function based on cyclic code [256,16,113] was implemented to filter out the deterministic bits. A resilient function can be easily obtained from linear codes. The p bit internal random numbers (say r[i]) are calculated from q 'das' bits (say s[i]) using $(r[i]....r[i+p-1]) = (s[i]....s[i+q-1]) * (G^T)$ where G

Table 7.1: Sunar et al. design without post-processing

# of ROs	# of inverters in a RO	fs in MHZ	Area in Slices	Through-put in Mbps	Through-put to area ratio	Test Suite Results
114	13	24	1608	24	14925	NIST (Fail)
114	13	8	1608	8	4975	NIST (Fail)

is a generator matrix for an [p,q,d] linear code and $p > q$. For cyclic codes, the generator matrix will have the form

$$\begin{pmatrix} g_0 & 0 & \cdots & 0 \\ g_1 & g_0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \\ g_{p-q-1} & g_{p-q-2} & \cdots & g_0 \\ 0 & g_{p-q-1} & \cdots & g_0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & & g_{p-q-1} \end{pmatrix}^T \tag{7.1}$$

In hardware, the resilient function was implemented as a LFSR where the position of the taps is determined by the generator matrix. The above post-processing has a compression factor of 256/16 =16 and can filter out 112 corrupted bits i.e., for every 256 'das' bits generated we get 16 bit internal random numbers.

The experimental evidence shows that the period of the ring oscillator is roughly 15 ns (66 MHZ) for 13 inverters on a Virtex II FPGA. We used sampling frequencies of 24 MHZ and 8 MHZ. The results in terms of area in slices, throughput in Mbps, throughput to area ratio and the statistical test suite results, without and with post-processing are presented in tables 7.1 and 7.2. Bitstreams of 1.5 million bits were generated and verified using the NIST statistical test suite [4]. Unfortunately the design failed to pass the NIST tests.

Table 7.2: Sunar et al. design with post-processing

# of ROs	# of inverters in a RO	fs in MHZ	Area in Slices	Through-put in Mbps	Through-put to area ratio	Test Suite Results
114	13	24	1781	1.5	842	NIST (Fail)
114	13	8	1781	0.5	280	NIST (Fail)

Table 7.3: Sunar et al. design with a flip flop after each ring oscillator

# of ROs	# of inverters in a RO	fs in MHZ	Area in Slices	Through-put in Mbps	Through-put to area ratio	Test Suites Result
114	13	24	1608	24	14925	Diehard (Pass), NIST (Pass)
50	11	24	612	24	39215	Diehard (Pass), NIST (Pass)

7.3.2 Sunar et al. Design with a flip flop after each ring oscillator

We implemented the design with same design parameters as above (114 ring oscillators and 13 inverters in each ring oscillator) and with a flip flop after each ring oscillator as shown in figure 6.2 in section 6.10. Bitstreams of 86.4 million bits were generated. The output bits passed the Diehard [3] and NIST [4] statistical test suites without any post-processing. The same design was also implemented with 50 ring oscillators and 11 inverters in each ring oscillator which is similar to the number of ring oscillators as implemented in [17]. The experimental evidence shows that the period of a ring oscillator is around 13ns (76 MHZ) for 11 inverters on a Virtex II FPGA. The output bits passed the NIST Diehard tests without any post-processing. The results are tabulated in table 7.3.

7.3.3 Our design

Our TRNG design as shown in figure 7.1 in section 7.1 was implemented with a single ring oscillator consisting of 13 inverters or 11 inverters for different choices of 'n' (loop number). Bitstreams of roughly 10 MB (86.4 million bits) have been captured from the

Table 7.4: Our Design

# of inverters in a RO	**'n' loop number**	**fs in MHZ**	**Area in Slices**	**Through-put in Mbps**	**Through-put to area ratio**	**Test Suites Result**
13	114	24	21	210	10000	Diehard (Pass), NIST (Pass), BSI (Pass)
13	110	24	21	218	10380	Diehard (Pass), NIST (Pass), BSI (Pass)
13	105	24	21	228	10857	Diehard (Fail), NIST (Fail)
11	110	24	18	218	12111	Diehard (Pass), NIST (Pass) BSI (Pass)

TRNG. The data is tested using statistical test suites from Diehard [3], NIST [4] and BSI [5]. The TRNG passed the tests from Diehard and BSI without any post-processing for loop numbers 114 and 110, but very few NIST tests reported a p-value less than the alpha (significance level). The design also passed all the NIST tests after being post-processed by Von Neumann corrector. For every pair of bits examined, the Von Neumann corrector outputs the first bit if the bits are different and discards both bits and output nothing if they are same. Hence, a Von Neumann corrector reduces bias in the output bits. We implemented the Von-Neumann corrector in software. The design started to fail slowly for loop numbers less than or equal to 105. The results are tabulated in table 7.4. Note that the Von Neumann corrector reduces the throughput approximately by four.

The choice of design parameters "13 inverters" and "114 times oversampling" are with reference to the model in [2] such that we build an identical sequential TRNG of the one proposed in [2]. Additional practical reasons for choosing 13 inverters or 11 inverters in a ring oscillator is to have the oscillator output period slightly above 10ns i.e., less than 100 MHZ in order to ease analog measurements on the oscilloscopes. The experimental evidence shows that the period of the ring oscillator is roughly 13ns (76 MHZ) for 11 inverters and

15 ns (66 MHZ) for 13 inverters on a Virtex II FPGA. As mentioned in section 7.2 sampling frequency is always chosen less than or equal to ring oscillator frequency and 24 MHZ is chosen such that one of the on-board clock oscillators can be used to generate the sampling clock. Throughput of the TRNG is a dependent on sampling frequency and can be increased by sampling at higher frequencies.

The results from Diehard and NIST statistical test suites are presented in Appendix from one of the bitstreams of our design with 11 inverters in a ring oscillator and 110 loop number.

7.4 Analysis

7.4.1 Evidence of Random Jitter

An ideal clock or a periodic wave would be operating at a particular frequency defined by a perfect rising and falling edges with a definite period. A real clock or a periodic signal will never have definite rising and falling edges but will always be associated with short term variations from their ideal positions in time. We define this behavior as jitter. Jitter in the ring oscillator occurs because of the internal noise within the gates of the ring oscillator. For experimental evaluation of the jitter in the ring oscillators we measured the frequency of a ring oscillator over a period of 1 ms. The analog output of the ring oscillator is monitored and captured with Agilent DSO6054A 500MHZ oscilloscope. Measurements are made on a ring oscillator with 51 inverters in a Virtex 2 FPGA chip, and the histogram plot between the number of samples in a cycle and number of occurrences of samples over 4 millions. Because of the limitation of the oscilloscope we were able to collect a maximum of 4 million samples is shown in figure 7.5. The samples are gaussian distributed with mean = 218.57 and standard deviation = 2.54.

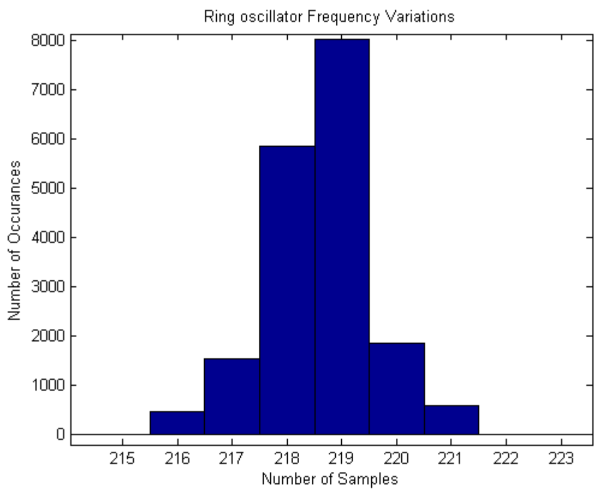

Figure 7.5: Histogram of ring oscillator frequency distribution.

7.4.2 Dependency among ring oscillators

In order to analyze if there is a dependency between ring oscillators we placed identical ring oscillators consisting of 13 inverters each manually at different locations in FPGA. we observed if the inverters of one ring are placed into LUTs of the same slice as the inverters of the other ring oscillator then both rings are interlocked as can be seen in figure 7.6. They are phase shifted by 90 degrees and operate at almost identical frequencies. Other configurations such as two ring oscillators whose inverters we placed into different slices of the same CLB in consecutive locations are not interlocked and operate at closer frequencies, and ring oscillators that are placed far to each other behave independently at different frequencies as can be seen in figures 7.7 and 7.8. respectively.

7.4.3 Reasons for failure of Sunar et al. design

Therefore the design shown in figure 6.1 with 114 ring oscillators in parallel of 13 inverters as suggested in [2] would cause the output of the XOR tree to switch at a maximum of 7.26 GHZ (0.13 ns) (114 * 66 MHZ). On a Virtex II FPGA our experiments show that a ring oscillator with 13 inverters in a oscillator ring operates at an average frequency of 66 MHZ. The XOR gate on Virtex II FPGA can not compute at that speed. Even if the XOR

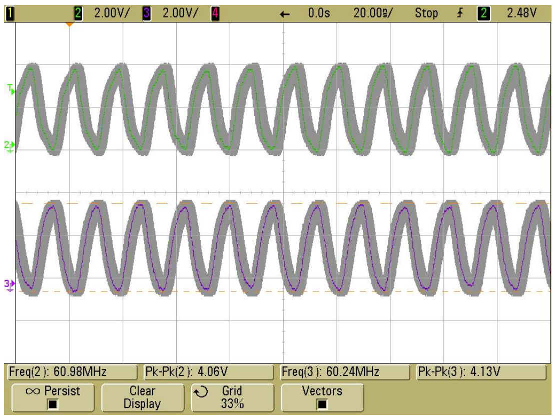

Figure 7.6: Very closely placed ring oscillators.

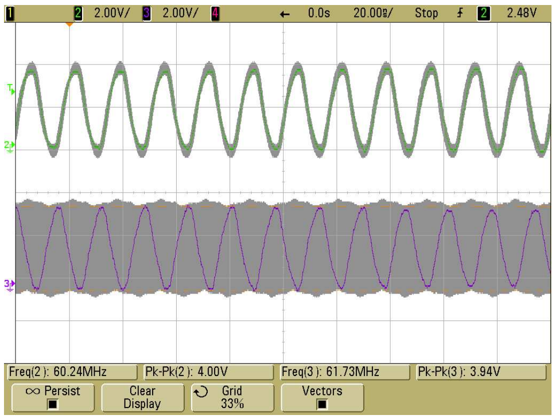

Figure 7.7: Closely placed ring oscillators.

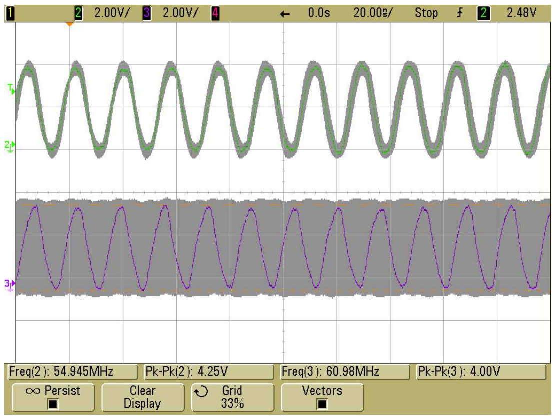

Figure 7.8: Far placed ring oscillators.

gate could be computed the sampling flip flop would not be able to latch the result due to set-up and hold times violation. Though for the Virtex II FPGA, a signal to be sampled must be available on the input of the flip flop from 0.37 ns before the clock pulse to 0.09 ns after. Indeed violating the set up and hold conditions will give rise to metastability and metastability by itself can be the source of randomness. However, whether metastability can really be achieved depends on the small manufacturing variations. Our interest is to explore the randomness caused by jitter.

Experimental evidence shows that inverters of ring oscillators placed in the same LUTs of the same slice in consecutive locations interlock. Hence for a design with large number of oscillator rings, the possibility of having multiple oscillators with identical phase increases which in turn decreases the available entropy. It becomes more difficult to verify the independence among the ring oscillators when a large number of rings are used. However, manual place and routing may isolate the rings from interacting with each other. But this approach will not be convenient for a design with larger number of rings.

The Introduction of flip flop after ring oscillator helps to cope with the problem with many transitions in the sampling period.

7.4.4 Performance of our design

In this section we compare the previous FPGA targeted TRNG designs with our design in terms of post-processing, resource usage and output bit rate. The evaluation criteria is

Post-Processing Score:

1. 0: complex post-processing is needed (e.g. resilient function)
2. 1: simple post-processing is sufficient (e.g. XOR corrector)
3. 2: post-processing is not necessary

Resource usage Score:

1. 0: huge resources

Table 7.5: Comparison of our design with others

Design	**Post Processing**	**Resource usage**	**Bit rate**	**Result**
Sunar	0	0	0	1
Sunar with flip flop	2	0	2	4
Gaj	1	1	1	3
Golic	0	1	1	2
Tkacik	2	1	1	4
Fischer	1	0	1	2
our	1	1	0	2

2. 1: negligible resources

Output bit-rate Score:

1. 0: output bit-rate up to 1 M bits

2. 1: output bit-rate 1 to 10 M bit/s

3. 2: output bit-rate more than 10 M bits

and the results are tabulated in table 7.5.

Chapter 8: Conclusion and Future Work

8.1 Conclusion

We believe that we were successful in constructing a simple and small TRNG targeted for FPGA using logic gates only. Our design is simple, straightforward, and uses a very small amount of FPGA resources generally fewer than 20 CLB slices.

It is shown that our TRNG design passes the tests from Diehard, NIST and BSI. we were successful in analyzing the reasons for the failure of the design in [2]. We also verified the design in [2] with a flip flop after each ring oscillator passes the tests from Diehard and NIST.

Our design can be seen as an sequential approach to the design proposed in [2] with the introduction of the flip flop after the ring oscillator. We showed that by over sampling technique, jitter in one ring oscillator is able to generate sufficient randomness.

Because of very low area of the TRNG, our design is suitable for area constraint and low power applications such as RFID tags, Sensor nodes and it is secure as the design is verified for randomness properties using statistical tests.

8.2 Future Work

Build a supporting mathematical justification for our design addressing the minimum over sampling required to extract enough randomness from a single ring oscillator. A closer look into the mathematical framework of the design provided in [2]. Since the Sunar et al. design passes the statistical tests suites with less number of ring oscillators, and without any post-processing when introduced a flip flop after each ring oscillator than suggested in [2].

We would like to carry out experiments using restart approach on our design as introduced in [16]. In this approach the TRNG is brought into same initial state after collection of each random bit by restarting the generator and by allowing a sufficient wait time before sampling the next random bit.

We would like to explore the scaleability of our design in terms of area vs. throughput, and energy vs. throughput.

Appendix A: An Appendix

A.1 Diehard Statistical Test Suite Results:

Number of bits tested : 86400000

Number of one's : 43167550

Number of zero's : 43232450

A.1.1 Birthday Spacings Test:

Bits used	**mean**	**chisqr**	**p-value**
1 to 24	16.05	16.6995	0.474897
2 to 25	15.86	16.5168	0.487536
3 to 26	15.68	12.2491	0.784826
4 to 27	16.00	16.6859	0.475835
5 to 28	15.84	11.4079	0.834590
6 to 29	15.83	21.9554	0.186441
7 to 30	15.67	18.2235	0.374877
8 to 31	16.04	9.7831	0.912458
9 to 32	15.73	20.6559	0.242062

no. of bdays=1024, no. of days/yr=2^{24}, lambda=16.00, sample size=500

degree of freedoms is: 17

p-value for KStest on those 9 p-values: 0.812132

A.1.2 Overlapping Permutations Test:

For samples of 1,000,000 consecutive 5-tuples

Sample 1

chisqr	p-value
76.794046	0.952153

degrees of freedom : 99

chisqr	p-value
82.131408	0.889988

degrees of freedom : 99

A.1.3 Ranks of 31x31 and 32x32 Matrices Test

Ranks of 31x31 matrices:

Rank	Observed	Expected	$(O-E)^2$/E	Sum
r¡=28	195	211.4	1.275	1.275
r=29	5228	5134.0	1.721	2.996
r=30	22977	23103.0	0.688	3.683
r=31	11600	11551.5	0.203	3.887

chi-square = 3.887 with df = 3; p-value = 0.274

Ranks of 32x32 matrices:

Rank	Observed	Expected	$(O-E)^2$/E	Sum
r¡=29	236	211.4	2.858	2.858
r=30	5130	5134.0	0.003	2.861
r=31	23065	23103.0	0.063	2.924
r=32	11569	11551.5	0.026	2.950

chi-square = 2.950 with df = 3; p-value = 0.399

A.1.4 Ranks of 6x8 Matrices Test

Bits 1 to 8

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	905	944.3	1.636	1.636
r=5	21715	21743.9	0.038	1.674
r=6	77380	77311.8	0.060	1.734

chi-square = 1.734 with df = 2; p-value = 0.420

Bits 2 to 9

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	937	944.3	0.056	0.056
r=5	21652	21743.9	0.388	0.445
r=6	77411	77311.8	0.127	0.572

chi-square = 0.572 with df = 2; p-value = 0.751

Bits 3 to 10

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	936	944.3	0.073	0.073
r=5	21508	21743.9	2.559	2.632
r=6	77556	77311.8	0.771	3.404

chi-square = 3.404 with df = 2; p-value = 0.182

Bits 4 to 11

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	955	944.3	0.121	0.121
r=5	21790	21743.9	0.098	0.219
r=6	77255	77311.8	0.042	0.261

chi-square = 0.261 with df = 2; p-value = 0.878

Bits 5 to 12

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	938	944.3	0.042	0.042
r=5	21786	21743.9	0.082	0.124
r=6	77276	77311.8	0.017	0.140

chi-square = 0.140 with df = 2; p-value = 0.932

Bits 6 to 13

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	919	944.3	0.678	0.678
r=5	21777	21743.9	0.050	0.728
r=6	77304	77311.8	0.001	0.729

chi-square = 0.729 with df = 2; p-value = 0.695

Bits 7 to 14

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	843	944.3	10.867	10.867
r=5	21716	21743.9	0.036	10.903
r=6	77441	77311.8	0.216	11.119

chi-square = 11.119 with df = 2; p-value = 0.004

Bits 8 to 15

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	907	944.3	1.473	1.473
r=5	21610	21743.9	0.825	2.298
r=6	77483	77311.8	0.379	2.677

chi-square = 2.677 with df = 2; p-value = 0.262

Bits 9 to 16

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	954	944.3	0.100	0.100
r=5	21533	21743.9	2.046	2.145
r=6	77513	77311.8	0.524	2.669

chi-square = 2.669 with df = 2; p-value = 0.263

Bits 10 to 17

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	929	944.3	0.248	0.248
r=5	21570	21743.9	1.391	1.639
r=6	77501	77311.8	0.463	2.102

chi-square = 2.102 with df = 2; p-value = 0.350

Bits 11 to 18

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	919	944.3	0.678	0.678
r=5	21679	21743.9	0.194	0.872
r=6	77402	77311.8	0.105	0.977

chi-square = 0.977 with df = 2; p-value = 0.614

Bits 12 to 19

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	857	944.3	8.071	8.071
r=5	21734	21743.9	0.005	8.075
r=6	77409	77311.8	0.122	8.198

chi-square = 8.198 with df = 2; p-value = 0.017

Bits 13 to 20

Rank	**Observed**	**Expected**	$(O - E)^2$/**E**	**Sum**
r¡=4	907	944.3	1.473	1.473
r=5	21694	21743.9	0.115	1.588
r=6	77399	77311.8	0.098	1.686

chi-square = 1.686 with df = 2; p-value = 0.430

Bits 14 to 21

Rank	**Observed**	**Expected**	$(O - E)^2$/**E**	**Sum**
r¡=4	998	944.3	3.054	3.054
r=5	21651	21743.9	0.397	3.451
r=6	77351	77311.8	0.020	3.471

chi-square = 3.471 with df = 2; p-value = 0.176

Bits 15 to 22

Rank	**Observed**	**Expected**	$(O - E)^2$/**E**	**Sum**
r¡=4	975	944.3	0.998	0.998
r=5	21507	21743.9	2.581	3.579
r=6	77518	77311.8	0.550	4.129

chi-square = 4.129 with df = 2; p-value = 0.127

Bits 16 to 23

Rank	**Observed**	**Expected**	$(O - E)^2$/**E**	**Sum**
r¡=4	937	944.3	0.056	0.056
r=5	21540	21743.9	1.912	1.968
r=6	77523	77311.8	0.577	2.545

chi-square = 2.545 with df = 2; p-value = 0.280

Bits 17 to 24

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	936	944.3	0.073	0.073
r=5	21770	21743.9	0.031	0.104
r=6	77294	77311.8	0.004	0.108

chi-square = 0.108 with df = 2; p-value = 0.947

Bits 18 to 25

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	987	944.3	1.931	1.931
r=5	21733	21743.9	0.005	1.936
r=6	77280	77311.8	0.013	1.949

chi-square = 1.949 with df = 2; p-value = 0.377

Bits 19 to 26

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	916	944.3	0.848	0.848
r=5	21596	21743.9	1.006	1.854
r=6	77488	77311.8	0.402	2.256

chi-square = 2.256 with df = 2; p-value = 0.324

Bits 20 to 27

Rank	**Observed**	**Expected**	$(O-E)^2$/**E**	**Sum**
r¡=4	942	944.3	0.006	0.006
r=5	21733	21743.9	0.005	0.011
r=6	77325	77311.8	0.002	0.013

chi-square = 0.013 with df = 2; p-value = 0.993

Bits 21 to 28

Rank	**Observed**	**Expected**	$(O - E)^2$/**E**	**Sum**
r¡=4	981	944.3	1.426	1.426
r=5	21534	21743.9	2.026	3.453
r=6	77485	77311.8	0.388	3.841

chi-square = 3.841 with df = 2; p-value = 0.147

Bits 22 to 29

Rank	**Observed**	**Expected**	$(O - E)^2$/**E**	**Sum**
r¡=4	951	944.3	0.048	0.048
r=5	21759	21743.9	0.010	0.058
r=6	77290	77311.8	0.006	0.064

chi-square = 0.064 with df = 2; p-value = 0.968

Bits 23 to 30

Rank	**Observed**	**Expected**	$(O - E)^2$/**E**	**Sum**
r¡=4	954	944.3	0.100	0.100
r=5	21718	21743.9	0.031	0.130
r=6	77328	77311.8	0.003	0.134

chi-square = 0.134 with df = 2; p-value = 0.935

Bits 24 to 31

Rank	**Observed**	**Expected**	$(O - E)^2$/**E**	**Sum**
r¡=4	934	944.3	0.112	0.112
r=5	21798	21743.9	0.135	0.247
r=6	77268	77311.8	0.025	0.272

chi-square = 0.272 with df = 2; p-value = 0.873

Bits 25 to 32

Rank	**Observed**	**Expected**	$(O-E)^2$**/E**	**Sum**
r¡=4	924	944.3	0.436	0.436
r=5	21668	21743.9	0.265	0.701
r=6	77408	77311.8	0.120	0.821

chi-square = 0.821 with df = 2; p-value = 0.663

TEST SUMMARY, 25 tests on 100,000 random 6x8 matrices

0.420175	0.751213	0.182357	0.877783	0.932337
0.694537	0.003851	0.262235	0.263312	0.349640
0.613610	0.016593	0.430369	0.176350	0.126877
0.280070	0.947252	0.377309	0.323727	0.993362
0.146565	0.968424	0.935249	0.872945	0.663305

p-value for KStest on those 9 p-values: 0.812132 = 0.452876

A.1.5 Monkey Tests on 20-bit words

20 bits/word, 2097152 words 20 bitstreams. # of missing words should average 141909.33 with sigma=428.00.

Bitstream	**# missing words**	**z-score**	**p-value**
1	141580	-0.77	0.779191
2	141768	-0.33	0.629379
3	141299	-1.43	0.923066
4	141898	-0.03	0.510560
5	142692	1.83	0.033725
6	142192	0.66	0.254484
7	142758	1.98	0.023691
8	142295	0.90	0.183768
9	142680	1.80	0.035881
10	143187	2.99	0.001417
11	142846	2.19	0.014317
12	143108	2.80	0.002550
13	142556	1.51	0.065406
14	142510	1.40	0.080244
15	141534	-0.88	0.809740
16	142783	2.04	0.020611
17	141786	-0.29	0.613386
18	142159	0.58	0.279832
19	142218	0.72	0.235396
20	141891	-0.04	0.517080

A.1.6 Monkey Tests OPSO, OQSO, DNA

Monkey test OPSO (Overlapping-Pairs-Sparse-Occupancy):

Bits used	**# missing**	**z-score**	**p-value**
43 to 32	141983	0.2540	0.399734
42 to 31	141738	-0.5908	0.722670
41 to 30	141543	-1.2632	0.896743
40 to 29	141927	0.0609	0.475707
39 to 28	142289	1.3092	0.095232
38 to 27	141992	0.2851	0.387796
37 to 26	141774	-0.4667	0.679627
36 to 25	142185	0.9506	0.170907
35 to 24	141965	0.1920	0.423885
34 to 23	141551	-1.2356	0.891700
33 to 22	141689	-0.7598	0.776301
32 to 21	142093	0.6333	0.263254
31 to 20	141511	-1.3736	0.915210
30 to 19	141736	-0.5977	0.724976
29 to 18	141832	-0.2667	0.605133
28 to 17	141709	-0.6908	0.755152
27 to 16	142024	0.3954	0.346269
26 to 15	141554	-1.2253	0.889764
25 to 14	142079	0.5851	0.279251
24 to 13	142178	0.9264	0.177106
23 to 12	141648	-0.9011	0.816242
22 to 11	141394	-1.7770	0.962216
21 to 10	142286	1.2989	0.096996

Monkey test OQSO (Overlapping-Quadraples-Sparse-Occupancy):

Bits used	**# missing**	**z-score**	**p-value**
38 to 32	142093	0.6226	0.266770
37 to 31	141936	0.0904	0.463982
36 to 30	141732	-0.6011	0.726119
35 to 29	142118	0.7074	0.239673
34 to 28	141929	0.0667	0.473419
33 to 27	142308	1.3514	0.088280
32 to 26	142124	0.7277	0.233400
31 to 25	142383	1.6057	0.054174
30 to 24	142233	1.0972	0.136280
29 to 23	141350	-1.8960	0.971022
28 to 22	142011	0.3446	0.365181
27 to 21	141795	-0.3876	0.650829
26 to 20	141852	-0.1943	0.577045
25 to 19	141418	-1.6655	0.952096
24 to 18	141905	-0.0147	0.505855
23 to 17	142116	0.7006	0.241784
22 to 16	142301	1.3277	0.092139
21 to 15	141969	0.2023	0.419852
20 to 14	141950	0.1379	0.445174
19 to 13	141675	-0.7943	0.786501
18 to 12	142248	1.1480	0.125477
17 to 11	142173	0.8938	0.185715
16 to 10	141919	0.0328	0.486925
15 to 9	142245	1.1379	0.127589

Bits used	**# missing**	**z-score**	**p-value**
14 to 8	141819	-0.3062	0.620275
13 to 7	141592	-1.0757	0.858968
12 to 6	141729	-0.6113	0.729496
11 to 5	142035	0.4260	0.335054

Monkey test DNA:

Bits used	**# missing**	**z-score**	**p-value**
35 to 32	141738	-0.5054	0.693360
34 to 31	142095	0.5477	0.291949
33 to 30	141569	-1.0039	0.842292
32 to 29	142112	0.5978	0.274971
31 to 28	141735	-0.5142	0.696461
30 to 27	141825	-0.2488	0.598227
29 to 26	141991	0.2409	0.404811
28 to 25	142041	0.3884	0.348857
27 to 24	142119	0.6185	0.268124
26 to 23	141814	-0.2812	0.610725
25 to 22	142206	0.8751	0.190751
24 to 21	142253	1.0138	0.155345
23 to 20	142397	1.4386	0.075138
22 to 19	141433	-1.4051	0.920005
21 to 18	141740	-0.4995	0.691286
20 to 17	141975	0.1937	0.423199
19 to 16	142181	0.8014	0.211454
18 to 15	142372	1.3648	0.086157
17 to 14	141436	-1.3963	0.918681

Bits used	# missing	z-score	p-value
16 to 13	141406	-1.4847	0.931195
15 to 12	141747	-0.4788	0.683977
14 to 11	141731	-0.5260	0.700572
13 to 10	141619	-0.8564	0.804120
12 to 9	141670	-0.7060	0.759902
11 to 8	141884	-0.0747	0.529781
10 to 7	142320	1.2114	0.112868
9 to 6	141700	-0.6175	0.731545
8 to 5	141673	-0.6971	0.757142
7 to 4	142151	0.7129	0.237957
6 to 3	142031	0.3589	0.359832

A.1.7 Count the 1's in a stream of bytes

Degrees of freedom: $5^4 - 5^3$=2500; sample size: 2560000

chisquare	z-score	p-value
2592.85	1.313	0.094567

A.1.8 Count the 1's in specific bytes

Degrees of freedom: $5^4 - 5^3$=2500; sample size: 256000

bits used	chisquare	z-score	p-value
1 to 8	2529.50	0.417	0.338276
2 to 9	2598.29	1.390	0.082264
3 to 10	2402.47	-1.379	0.916092
4 to 11	2430.40	-0.984	0.837531
5 to 12	2428.02	-1.018	0.845644
6 to 13	2389.24	-1.566	0.941378
7 to 14	2498.03	-0.028	0.511140
8 to 15	2471.84	-0.398	0.654770
9 to 16	2528.57	0.404	0.343102
10 to 17	2431.20	-0.973	0.834708
11 to 18	2498.70	-0.018	0.507353
12 to 19	2470.35	-0.419	0.662509
13 to 20	2481.21	-0.266	0.604795
14 to 21	2495.14	-0.069	0.527380
15 to 22	2425.35	-1.056	0.854436
16 to 23	2492.25	-0.110	0.543664
17 to 24	2649.69	2.117	0.017133
18 to 25	2538.18	0.540	0.294595
19 to 26	2426.08	-1.045	0.852093
20 to 27	2529.78	0.421	0.336838
21 to 28	2541.71	0.590	0.277657
22 to 29	2379.15	-1.709	0.956281
23 to 30	2601.59	1.437	0.075400
24 to 31	2478.24	-0.308	0.620860
25 to 32	2504.62	0.065	0.473945

A.1.9 Parking Lot Test

Of 12000 tries, the average no. of successes should be 3523.0 with sigma=21.9

succeses	z-score	p-value
3504	-0.8676	0.807188
3523	0.0000	0.500000
3509	-0.6393	0.738676
3568	2.0548	0.019949
3518	-0.2283	0.590298
3524	0.0457	0.481790
3525	0.0913	0.463617
3565	1.9178	0.027568
3498	-1.1416	0.873180
3549	1.1872	0.117571

Square side=100, avg. no. parked=3528.30 sample std.=23.23 p-value of the KSTEST for those 10 p-values: 0.571916

A.1.10 Minimum distance Test

Sample no.	d^2	**mean**	**equiv uni**
5	2.2746	0.7721	0.898332
10	0.0212	1.0565	0.021073
15	0.1247	0.8049	0.117820
20	0.6823	0.8466	0.496294
25	1.3105	0.8876	0.732090
30	1.1743	0.8427	0.692781
35	0.2208	0.8208	0.199013
40	0.3868	0.7984	0.322109
45	0.6093	0.7534	0.457921
50	0.5128	0.7339	0.402754
55	0.9941	0.7217	0.631780
60	1.0146	0.7716	0.639284
65	1.9596	0.7711	0.860471
70	0.6528	0.7524	0.481129
75	1.5258	0.7617	0.784220
80	0.8336	0.7405	0.567320
85	1.8212	0.7372	0.839642
90	0.0311	0.7213	0.030746
95	1.3012	0.7628	0.729577
100	2.1656	0.8081	0.886557

p-value of KS test on 100 transformed mindist$^{2\prime}s$: 0.131523

A.1.11 Random spheres Test

sample	r^3	**equiv. uni.**
1	33.128	0.668542
2	38.877	0.726345
3	34.480	0.683155
4	31.616	0.651411
5	2.079	0.066954
6	1.191	0.038914
7	8.909	0.256933
8	56.611	0.848482
9	0.612	0.020190
10	24.153	0.552952
11	41.298	0.747568
12	41.606	0.750144
13	0.414	0.013711
14	13.461	0.361552
15	5.454	0.166236
16	70.329	0.904086
17	10.003	0.283532
18	10.361	0.292031
19	103.360	0.968106
20	11.807	0.325359

p-value for KS test on those 20 p-values: 0.601500

A.1.12 The Squeeze Test

Chi-sqr	z-score	p-value
37.297458	-0.513089	0.677269

degrees of freedom: 37.297458

A.1.13 Overlapping Sums Test

p-value for 10 kstests on 100 kstests: 0.325282

Test no	p-value
1	0.185391
2	0.069114
3	0.005341
4	0.888086
5	0.062559
6	0.828611
7	0.266054
8	0.554125
9	0.895794
10	0.537858

A.1.14 Runs Test

Up and down runs in a sequence of 10000 numbers

Set 1

runs up; ks test for 10 p's: 0.892419

runs down; ks test for 10 p's: 0.679890

Set 2

runs up; ks test for 10 p's: 0.878139

runs down; ks test for 10 p's: 0.438401

A.1.15 The Craps Test

No. of wins: Observed = 98662 Expected = 98585.86

z-score= 0.341, pvalue=0.36672

Analysis of Throws-per-Game:

Throws	**Observed**	**Expected**	**Chisq**	**Sum of $(O-E)^2$/E**
1	66571	66666.7	0.137	0.137
2	37576	37654.3	0.163	0.300
3	27398	26954.7	7.289	7.590
4	19242	19313.5	0.264	7.854
5	13998	13851.4	1.551	9.405
6	9736	9943.5	4.332	13.737
7	7103	7145.0	0.247	13.984
8	5166	5139.1	0.141	14.125
9	3645	3699.9	0.814	14.939
10	2616	2666.3	0.949	15.888
11	1892	1923.3	0.510	16.398
12	1385	1388.7	0.010	16.408
13	1005	1003.7	0.002	16.410
14	751	726.1	0.851	17.261
15	532	525.8	0.072	17.333

Throws	**Observed**	**Expected**	**Chisq**	**Sum of $(O-E)^2$/E**
16	374	381.2	0.134	17.467
17	273	276.5	0.045	17.513
18	185	200.8	1.248	18.760
19	143	146.0	0.061	18.821
20	109	106.2	0.073	18.894
21	300	287.1	0.578	19.473

Chisq= 19.47 for 20 degrees of freedom, p= 0.49132

SUMMARY of craptest:

p-value for no. of wins: 0.366722

p-value for throws/game: 0.491317

A.2 NIST Test Suite Results:

Number of bits tested : 21562705

Number of one's : 10779206

Number of zero's : 10783499

P-value less than alpha (alpha = 0.01 in our case) is flagged with an "*". It is acceptable for a few individual tests to fail. In our case none of the the tests indicated a failure except two runs of Non-Overlapping Template matching test.

Table A.1: NIST Test Suite Results

Statistical Test	**P-value**
Frequency	0.355223
Block Frequency	0.643119
Cumulative Sums	0.573675
Cumulative Sums	0.481221
Runs	0.030844
Longest Run	0.518476
Rank	0.971692
NonOverlapping Template	0.573369
NonOverlapping Template	0.087844
NonOverlapping Template	0.626749
NonOver lapping Template	0.725479
NonOverlapping Template	0.335733
NonOverlapping Template	0.771810
NonOverlapping Template	0.212443
NonOverlapping Template	0.958261
NonOverlapping Template	0.359128
NonOverlapping Template	0.637296
NonOverlapping Template	0.917773
NonOverlapping Template	0.680270
NonOverlapping Template	0.858704
NonOverlapping Template	0.578545
NonOverlapping Template	0.392710
NonOverlapping Template	0.963932
NonOverlapping Template	0.190981
NonOverlapping Template	0.136642
NonOverlapping Template	0.779652
NonOverlapping Template	0.357220
NonOverlapping Template	0.629798
NonOverlapping Template	0.309260
NonOverlapping Template	0.497389
NonOverlapping Template	0.156441
NonOverlapping Template	0.360448
NonOverlapping Template	0.244008
NonOverlapping Template	0.193580
NonOverlapping Template	0.688256
NonOverlapping Template	0.029765
NonOverlapping Template	0.987582
NonOverlapping Template	0.191661
NonOverlapping Template	0.561238
NonOverlapping Template	0.872209
NonOverlapping Template	0.575510
NonOverlapping Template	0.953775

Table A.2: NIST Test Suite Results

Statistical Test	**P-value**
NonOverlapping Template	0.708769
NonOverlapping Template	0.160443
NonOverlapping Template	0.627757
NonOverlapping Template	0.662570
NonOverlapping Template	0.696194
NonOverlapping Template	0.486535
NonOverlapping Template	0.660200
NonOverlapping Template	0.594223
NonOverlapping Template	0.383421
NonOverlapping Template	0.008744*
NonOverlapping Template	0.248890
NonOverlapping Template	0.268825
NonOverlapping Template	0.180182
NonOverlapping Template	0.150626
NonOverlapping Template	0.861009
NonOverlapping Template	0.093526
NonOverlapping Template	0.563616
NonOverlapping Template	0.183459
NonOverlapping Template	0.014657
NonOverlapping Template	0.955495
NonOverlapping Template	0.876902
NonOverlapping Template	0.579401
NonOverlapping Template	0.423808
NonOverlapping Template	0.321044
NonOverlapping Template	0.826774
NonOverlapping Template	0.262060
NonOverlapping Template	0.735979
NonOverlapping Template	0.542398
NonOverlapping Template	0.294489
NonOverlapping Template	0.587817
NonOverlapping Template	0.284985
NonOverlapping Template	0.868695
NonOverlapping Template	0.887104
NonOverlapping Template	0.785219
NonOverlapping Template	0.626408
NonOverlapping Template	0.075814
NonOverlapping Template	0.431580
NonOverlapping Template	0.094638
NonOverlapping Template	0.961000
NonOverlapping Template	0.571690
NonOverlapping Template	0.470367
NonOverlapping Template	0.243620

Table A.3: NIST Test Suite Results

Statistical Test	**P-value**
NonOverlapping Template	0.174577
NonOverlapping Template	0.943736
NonOverlapping Template	0.963180
NonOverlapping Template	0.311020
NonOverlapping Template	0.600872
NonOverlapping Template	0.479501
NonOverlapping Template	0.851769
NonOverlapping Template	0.539260
NonOverlapping Template	0.701412
NonOverlapping Template	0.015498
NonOverlapping Template	0.772386
NonOverlapping Template	0.972908
NonOverlapping Template	0.845218
NonOverlapping Template	0.475838
NonOverlapping Template	0.804597
NonOverlapping Template	0.635523
NonOverlapping Template	0.783382
NonOverlapping Template	0.519413
NonOverlapping Template	0.466569
NonOverlapping Template	0.571032
NonOverlapping Template	0.061094
NonOverlapping Template	0.695386
NonOverlapping Template	0.553744
NonOverlapping Template	0.974115
NonOverlapping Template	0.689949
NonOverlapping Template	0.250371
NonOverlapping Template	0.323625
NonOverlapping Template	0.786354
NonOverlapping Template	0.192136
NonOverlapping Template	0.384204
NonOverlapping Template	0.942551
NonOverlapping Template	0.480066
NonOverlapping Template	0.309686
NonOverlapping Template	0.521923
NonOverlapping Template	0.199240
NonOverlapping Template	0.580081
NonOverlapping Template	0.968343
NonOverlapping Template	0.798613
NonOverlapping Template	0.543091

Table A.4: NIST Test Suite Results

Statistical Test	**P-value**
NonOverlapping Template	0.443366
NonOverlapping Template	0.017349
NonOverlapping Template	0.268378
NonOverlapping Template	0.856264
NonOverlapping Template	0.803348
NonOverlapping Template	0.698083
NonOverlapping Template	0.674379
NonOverlapping Template	0.397060
NonOverlapping Template	0.332263
NonOverlapping Template	0.201704
NonOverlapping Template	0.941130
NonOverlapping Template	0.180375
NonOverlapping Template	0.259884
NonOverlapping Template	0.205776
NonOverlapping Template	0.738439
NonOverlapping Template	0.690577
NonOverlapping Template	0.411947
NonOverlapping Template	0.983478
NonOverlapping Template	0.345341
NonOverlapping Template	0.594742
NonOverlapping Template	0.208972
NonOverlapping Template	0.007743*
NonOverlapping Template	0.197983
NonOverlapping Template	0.467691
NonOverlapping Template	0.855341
NonOverlapping Template	0.484397
NonOverlapping Template	0.749982
NonOverlapping Template	0.655636
NonOverlapping Template	0.721982
NonOverlapping Template	0.256416
NonOverlapping Template	0.732759
NonOverlapping Template	0.961000

Table A.5: NIST Test Suite Results

Statistical Test	**P-value**
OverlappingTemplate	0.075511
Universal	0.748333
ApproximateEntropy	0.984421
RandomExcursions	0.322547
RandomExcursions	0.559239
RandomExcursions	0.512914
RandomExcursions	0.934510
RandomExcursions	0.625945
RandomExcursions	0.888139
RandomExcursions	0.584231
RandomExcursions	0.130241
RandomExcursionsVariant	0.343258
RandomExcursionsVariant	0.344102
RandomExcursionsVariant	0.219082
RandomExcursionsVariant	0.155476
RandomExcursionsVariant	0.521436
RandomExcursionsVariant	0.638845
RandomExcursionsVariant	0.285951
RandomExcursionsVariant	0.387322
RandomExcursionsVariant	0.643658
RandomExcursionsVariant	0.534702
RandomExcursionsVariant	0.910464
RandomExcursionsVariant	0.510064
RandomExcursionsVariant	0.534500
RandomExcursionsVariant	0.754698
RandomExcursionsVariant	0.730080
RandomExcursionsVariant	0.761229
RandomExcursionsVariant	0.770046
RandomExcursionsVariant	0.582894
Serial	0.201195
Serial	0.126447
LinearComplexity	0.306065

Bibliography

Bibliography

[1] A. J. Menezes, P. C. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography.* CRC Press Inc., 1997.

[2] B. Sunar, W. Martin, and D. 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, Jan 2007.

[3] G. Marsaglia, *Diehard Battery of Tests of Randomness*, Florida State University, http://www.stat.fsu.edu/pub/diehard/, 1995.

[4] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, National Institute of Standards and Technology NIST, NIST Special Publication 800-22, May 2001.

[5] W. Killmann and W. Schindler, "A proposal for: Functionality classes and evaluation methodology for true (physical) random number generators," Bundesamt für Sicherheit in der Informationstechnnik (BSI), Bonn, Germany, Tech. Rep. Ver 3.1, Sep 2001.

[6] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudo random number generator," in *In the Sixth Annaul Workshop on Selected Areas in Cryptograhy.* Springer, 1999, pp. 13–33.

[7] K. H. Tsoi, K. H. Leung, and P. Leong, "Compact FPGA-based true and pseudo random number generators," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM03)*, April 2003, pp. 51–61.

[8] K. H. Lundberg, "Noise sources in bulk CMOS ," http://web.mit.edu/klund/www/papers/UNP_noise.pdf, October 2002.

[9] M. S. Keshner, "1/f noise," in *Proceedings of the IEEE*, vol. 70, March 1982, pp. 212–218.

[10] H. C. A. V. Tilborg, *Encyclopedia of Cryptography and Security.* Springer, 2005.

[11] I. Vasyltsov, E. Hambardzumyan, Y.-S. Kim, and B. Karpinskyy, "Fast digital trng based on metastable ring oscillator," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, ser. LNCS, E. Oswald and P. Rohatgi, Eds., vol. 5154. Springer, 2008, pp. 164–180.

[12] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 570–580, Nov.-Dec. 2007.

[13] D. Schellekens, B. Preneel, and I. Verbauwhede, “FPGA vendor agnostic true random number generator,” in *Field Programmable Logic and Applications – FPL 2006*, A. Koch, P. Leong, and P. Leong, Eds. IEEE, Aug 2006, pp. 139–144.

[14] V.Fischer and M.Drutarovský, “True random number generator embedded in reconfigurable hardware,” in *Workshop on Cryptographic Hardware and Embedded Systems - CHES 2002*, ser. Lecture Notes in Computer Science (LNCS), vol. 2523. Springer, Aug 2003, pp. 415–430.

[15] P. Kohlbrenner and K. Gaj, “An embedded true random number generator for FPGAs,” in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, Feb 2004, pp. 71–78.

[16] M.Bucci and R.Luzzi, “Design of testable random bit generators,” in *CHES-2005*, vol. 3659/2005, sep 2005, pp. 147–156.

[17] K. Wold and C. H. Tan, “Analysis and enhancement of random number generation in FPGA based on oscillator rings,” in *International Conference on Reconfigurable Computing and FPGAs*, Dec 2008, pp. 385–390.

[18] M. Epstein, L. Hars, R. Krasinski, M. Rosner, and H. Zheng, “Design and implementation of a true random number generator based on digital circuit artifacts,” in *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2003*, ser. Lecture Notes in Computer Science (LNCS), C. D. Walter, Çetin K. Koç, and C. Paar, Eds., vol. 2779. Berlin: Springer-Verlag, Sep 2003, pp. 152–165.

[19] B. J. Abcunas, S. P. Coughlin, G. T. Pedro, and D. C. Reisberg, “Evaluation of random number generators on FPGAs,” Worcester Polytechnic Institute, Worcester, MA, USA, MQP Report, 2004.

[20] W. R. Coppock and C. R. Philbrook, “A mathematical and physics analysis of circuit jitter with application to cryptographic random bit generation,” Worcester Polytechnic Institute, Worcester, MA, USA, MQP Report, 2005.

[21] W. Schindler, *Cryptographic Engineering.* Springer, 2009, ch. Evaluation Criteria for Physical Random Number Generators, pp. 25–54.

[22] B. Barak, R. Shaltiel, and E. Tromer, “True random number generators secure in a changing environment,” in *Cryptographic Hardware and Embedded Systems - CHES 2003.* Springer-Verlag, 2003, pp. 166–180.

[23] B. Jun and P. Kocher, “The Intel random number generator,” Intel Corporation, Tech. Rep., April 1999.

[24] *Security Requirements for Cryptographic Modules*, National Institute of Standards and Technology NIST, FIPS Publication 140-2, May 2001.

[25] T. E. Tkacik, “A hardware random number generator,” in *CHES–2002*, june 2002, pp. 450–453.

[26] M. Dichtl, “How to predict the output of a hardware random number generator,” in *Cryptographic Hardware and Embedded Systems - CHES 2003*, ser. LNCS, C. D. Walter, Çetin Kaya Koç, and C. Paar, Eds., vol. 2779. Springer, 2003, pp. 181–188.

[27] J. D. Golić, “New methods for digital generation and postprocessing of random data,” *IEEE Transaction on Computers*, vol. 55, no. 10, pp. 1217–1229, Oct 2006.

[28] M. Dichtl and J. D. Golić, “High-speed true random number generation with logic gates only,” in *Cryptographic Hardware and Embedded Systems - CHES 2007*, ser. Lecture Notes in Computer Science (LNCS), vol. 4727. Springer, Sep 2007, pp. 45–62.

[29] C. W. O’Donnell, G. Edward Suh, and S. Devadas, “Puf-based random number generation,” Massachusetts Institute of Technology, Cambridge, MA 02139, Tech. Rep., 2004.

[30] A. Maiti, N. Raghunandan, A. Reddy, and P. Schaumont, “Physical unclonable function and true random number generator: a compact and scalable implementation,” in *GLSVLSI ’07: Proceedings of the 17th great lakes symposium on Great lakes symposium on VLSI*, Boston, Massachusetts, USA, May 2009.

[31] G. Suh and S. Devadas, “Physical unclonable functions for device authentication and secret key generation,” in *Annual ACM IEEE Design Automation Conference Proceedings of the 44th annual conference on Design automation*, 2007, pp. 9–14.

[32] *Virtex-II^{TM} V2MB1000, Development Board, User’s Guide*, 1st ed., Xilinx, Inc., May 2002.

[33] *Virtex-II Platform FPGAs:, Complete Data Sheet*, Ds031 (v3.5) ed., Xilinx, Inc., Nov 2007.

Curriculum Vitae

Shashi Prashanth Karanam received his Bachelor of Technology Degree from Kamala Institute of Technology and Science, Singapur, India in 2006. He was involved with teaching various undergraduate and graduate courses at George Mason University, both as a Teaching Assistant and as a Lab Instructor. He was a Hardware Support Engineer intern with Wireless Ventures, Mclean and a Computer Engineer intern with Microwave Technology Inc. He is a research student with Cryptographic Engineering Research Group (CERG).