A TRUE RANDOM NUMBER GENERATOR IN FPGA FOR CRYPTOGRAPHIC APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SALİH YILDIRIM

IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

DECEMBER 2012

Approval of the thesis:

A TRUE RANDOM NUMBER GENERATOR IN FPGA FOR CRYPTOGRAPHIC APPLICATIONS

submitted by **SALİH YILDIRIM** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Name, Last name	: SALİH YILDIRIM
Signature	:

ABSTRACT

A TRUE RANDOM NUMBER GENERATOR IN FPGA FOR CRYPTOGRAPHIC APPLICATIONS

Yıldırım, Salih

M.Sc., Department of Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı

December 2012, 121 pages

In this thesis a True Random Number Generator (TRNG) employed for cryptographic applications is investigated, implemented and evaluated. The design of TRNG and its embedded tests are described in VHDL language and then implemented on an FPGA platform. Randomness is extracted from the jitter of ring oscillators that has self-failure detecting and sampling logic. The implementation needs only primitive resources which are common in all kinds of FPGAs. The embedded randomness tests described in Federal Information Processing Standard (FIPS) 140-1 are realized on FPGA. The statistical quality of the generated random bits is also confirmed by running the Diehard and NIST (National Institute of Standards and Technology) Test Suites seperately. The implemented TRNG has a throughput up to 0.5 Mbps and its core occupies only 25 slices of Xilinx Virtex-5 FPGA. This design demonstrates the possibility of generating and confirming true random bit sequences by using only the internal resources of FPGAs. The performance of our TRNG is also compared with a separate IC, RPG100 from FDK Corporation.

Keywords: True Random Number Generator (TRNG), Field Programmable Gate Array (FPGA), National Institute of Standard and Technology Special Publication 800-22 (NIST SP800-22), Diehard Test Suite.

ÖZ

KRİPTO UYGULAMALARI İÇİN FPGA ÜZERİNDE GERÇEK RASSAL SAYI ÜRETECİ

Yıldırım, Salih

Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Cüneyt F. Bazlamaççı

Aralık 2012, 121 sayfa

Bu tezde kriptografik uygulamalarda kullanılan bir Gerçek Rassal Sayı Üreteci (GRSÜ) araştırılmış, gerçeklenmiş ve değerlendirilmiştir. Bu GRSÜ ve gömülü testlerinin tasarımı VHDL dilinde tanımlanmış ve bir FPGA platformunda gerçeklenmiştir. Rassallık halka osilatör seğirmelerinden içsel hata algılama yetisine sahip bir örnekleme devresi ile çıkarılmıştır. Gerçekleme yalnızca tüm FPGA'lerde ortak olarak yer alan en temel kaynaklara ihtiyaç duymaktadır. FIPS 140-1 standardında açıklanan gömülü rassallık testleri FPGA üzerinde gerçeklenmiştir. GRSÜ'nin istatiksel özellikleri kriptografik uygulamalar için standart olan NIST SP800-22 ve Diehard istatiksel test kütüphaneleri ile de doğrulanmıştır. Gerçeklenen GRSÜ'nin üretim hızı 0,5 MBps'a kadar ulaşmaktadır ve çekirdeği sadece 25 adet Xilinx Virtex-5 FPGA slice kaynağı kullanmaktadır. Bu tasarım sadece FPGA'in içsel kaynaklarını kullanarak da rassal sayı üretilip bu rassallığın doğrulanabileceğini göstermektedir. Ayrıca gerçeklenen GRSÜ'nin başarımı ayrı bir tümleşik devre olan, FDK şirketinden RPG100, ile de karşılaştırılmıştır.

Anahtar Kelimeler: Gerçek Rassal Sayı Üreteci (GRSÜ), Field Programmable Gate Array (FPGA), National Institute of Standard and Technology Special Publication 800-22 (NIST SP800-22), Diehard Test Kütüphanesi.

To My Daughter

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor Assoc. Prof. Dr. Cüneyt Bazlamaççı for his guidance and support in my study. His inspiring suggestions and meticulous feedback in every step of this thesis enabled me to write it and made it an invaluable experience for me. It has been a pleasure to write this thesis under his guidance.

I would like to thank to my technical leader Mr. Hakan Solmaz in Aselsan for his support and time. Also it is a binding duty to thank to Aselsan Inc. for facilities and equipments provided for the completion of my thesis.

Last but not least, most special thanks and love go to my wife Gül Dilek Yıldırım for her continuous support and encouragements. It could have been impossible to write this thesis without her love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS

AIS	Anwendungshinweise und Interpretationen zum Schema
BSI	Bundesamt für Sicherheit in der Informationstechnik
ASCII	American Standard Code for Information Interchange
ASIC	Aplication Specific Integrated Circuit
CASR	Cellular Automata Shift register
FIPS	Federal Information Processing Standard
FIRO	Fibonacci Ring Oscillator
FPGA	Field Programmable Gate Array
GARO	Gallois Ring oscillator
IC	Integrated Circuit
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
METARO	Metastable Ring oscillator
NIST	National Institude of Standards and Technology
PLL	Phase Lock Loop
PRNG	Pseudo Random Number Generator
RAM	Random Access Memory
RNG	Random Number Generator
RO	Ring Oscillator
TERO	Transition effect ring oscillator
TFF	Toggle Flip Flop
TRNG	True Random Number Generator
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

CHAPTER 1

INTRODUCTION

Random number generators (RNGs) are one of basic cryptographic primitives that generate random quantities for cryptosystems. These random quantities are utilized for processes such as key generation, authentication, padding and password generation, etc. The implementation of counter measure processes to side channel attacks also needs random quantities. Security of various cryptographic systems directly depends on random numbers that should remain unknown to an adversary. Hence random numbers used in cryptographic systems should meet stringent requirements. These numbers should first be unpredictable (unbiased and independent) and irreproducible and also should have good statistical features.

Random numbers that have good statistical properties can be generated by deterministic processes. Such generators are called pseudo random number generators (PRNG). PRNGs are implementations of deterministic polynomial time algorithms that expand a short seed value into uniformly distributed long bit sequences having good statistical features [1]. However the output of a PRNG is not unpredictable because the input (i.e. seed) deterministically governs the output and a PRNG can generate a fixed number of bits meaning they repeat themselves in long sequences. Because of these drawbacks, PRNGs are not suitable for many cryptographic applications.

On the other hand, some generators use uncontrollable and unpredictable physical processes as the source of randomness. This type of RNGs are called true random number generators (TRNG). The statistical properties and features of the generated random numbers by TRNGs are based on the randomness of the physical process used and the extraction method employed. If the underlying physical process cannot be controlled, the generator output also becomes unpredictable and uncontrollable.

In cryptography, Kerckhoff's principle says that "A cryptosystem should be secure even if everything about the system, except the key, is public knowledge". Even for Kerckhoff's secure crypto system, protection of the confidential keys is a very important issue [2]. If an information system is used in an uncontrollable environment, cryptographic keys used in the security components should never be open to the outside. For this reason, it is generally recommended that the security system should be implemented in a single chip and the keys should be generated inside the same chip. Therefore, implementation of random number generators in logic devices (including FPGAs) is a crucial issue.

Traditionally, in spite of the long and difficult design cycle, the common choice of implementation platform in cryptography was ASIC because of high performance, low power and design security. On the other hand FPGA manufacturers have come closer to filling the performance gap between FPGAs and ASICs in recent years. Besides performance, existence of vendor specific security solutions in cryptography increases the prevalence of FPGAs on this market [3].

In this thesis we also have chosen FPGA as our implementation platform because of its increasing usage in cryptography and its flexible and faster design cycle. Implementing a TRNG on an FPGA is not a straightforward issue because FPGAs are digital devices, which are produced to implement deterministic processes. Any non-deterministic behavior at the logic level can cause failures on such devices. On the other hand unpredictable and uncontrollable events due to underlying physical processes are unavoidable in such devices although device vendors aim to minimize them. These underlying physical processes and impurities can provide a source of randomness for TRNG implementation on FPGA.

In general, one can utilize three kinds of randomness in an FPGA. These are i) unstable propagation delay of logic gates, ii) transient behavior of logic gates between two states (meta-stability), and iii) thermal noise in FPGA [2]. The most popular source of randomness in FPGAs is the unstable propagation delay in logic gates. This source appears as jitter on ring oscillators which are the main component of most TRNG designs.

In the development cycle of a TRNG, the validation of randomness is among the most important parts of the process. The validation of a TRNG can be done by using statistical test suites that are published by standard institutes such as NIST. There are several statistical tests for randomness. The confirmation of these test suites is required for a particular TRNG but still this does not means that the implemented TRNG is a perfect random number generator.

Random numbers that are generated by a TRNG can be biased and can have bad statistical properties even if they are extracted correctly. Therefore post processing is also required in most TRNGs in order to get good statistical properties. This post processing brings the risk of being unaware of TRNG failure since post processing may hide the impurity of the outputs. Because of this risk TRNG tests should be run in real time and in a continuous manner for detecting any possible failure in the random number generator [4].

In our thesis work, we have realized an already proposed TRNG [5] method on Virtex-5 FPGA with ML507 Xilinx FPGA Development Kit. Our implementation uses only standard FPGA resources and has 0.5 Mbits/sec throughput. The generated random numbers are tested and confirmed by NIST statistical test suite [6] and Diehard battery of tests [7] using 1 Gigabit as the sample size. Besides the embedded randomness tests that are published by FIPS 140-1 [8] are also realized in our TRNG in order to control the outputs concurrently in real time operation.

The rest of the thesis is organized as follows:

In Chapter 2, background information about true random number generators for cryptographic applications is given. Practical methods and randomness tests found in literature for true random number generation in FPGA are presented.

In Chapter 3, implementation details of our TRNG and its embedded test module are explained. The design is presented using a top to bottom hierarchy.

In Chapter 4, statistical evaluation of our TRNG and simulation of the design are presented. This chapter contains statistical test results and circuit simulation outputs. A comparison of our TRNG and RPG100, an existing IC TRNG [10] (from FDK Corporation) has also been performed.

Chapter 5 is the conclusion, which includes a summary of the study and possible future work.

CHAPTER 2

TRUE RANDOM NUMBER GENERATORS

2.1 General Structure of True Random Number Generators

True Random Number Generators (TRNG) utilize some physical processes as the source of randomness. If the physical process utilized is unpredictable and it cannot be controlled, then the output is also uncontrollable and unpredictable [11]. The throughput and statistical characteristics of TRNGs are closely related to the quality of the randomness of the source and the extraction method used. Generally, the statistical properties of randomness extractor output bits do not have good statistical properties even if they are extracted correctly because physical randomness sources generally have low entropy that does not fulfill the cryptographic applications requirements. For this reason, some post-processing algorithms have to be employed to enhance the statistical parameters of the output bit-stream. However, the TRNG output post-processing creates a risk that it will mask abnormalities coming from either a poorly designed generator or an external attack. This masked fault could remain undetected by standard statistical tests also.

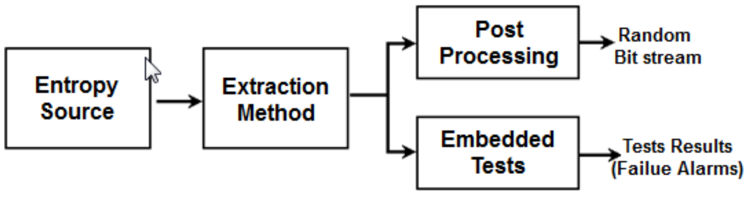


Figure 2-1 - General Structure of TRNG

The security of the generator can be increased by implementing embedded randomness tests that are running concurrently with the generator. This test equipment detects any failure or deviation from randomness.

The general structure of a TRNG is given in Figure 2-1. There are four main sub blocks that are named as Entropy Source, Extraction Method, Post Processing and Embedded Tests. These sub blocks of a True Random Number generator are explained in the following subsections of this chapter in detail.

2.1.1 Entropy Source

Entropy source of a true random number generator provides a stationary random process to the extractor for sampling truly random bits from that process. Sampled random bit must be unpredictable. Possibility of a true guess on the logical level of that sampled bit must be %50 even if the predecessors and the successors of that bit are known [12]. Furthermore, it should not be possible to control this process by any means. For our interest, we have to find this kind of an entropy source in logic devices especially in FPGAs. Logic devices are digital devices that are manufactured to implement deterministic processes. Each non-deterministic behavior in such a system (caused by a meta-stability, clock jitter, radiation errors, etc.) can have catastrophic consequences on the overall system behavior. On the other hand the underlying technology of these digital devices is still analogue circuits, which are running on physical processes. So the unpredictable events due to the physical nature of the underlying technology are unavoidable. The vendors of logic devices are working hard to minimize the impurities coming from the underlying physical process [11].

In general, there are three phenomena and their combinations, which can be used as the source of randomness in FPGAs. These are the variation of the delay in logic gates, the analog behavior of logic gates between two logic levels (e.g. meta-stability) and the thermal noise generated inside the device [2].

Logic gates of FPGAs have their own time delay, which is not constant and stable. Instability of the delays in logic gates causes variations signal propagation time. If one measures this variation of propagation delay, this can be assumed to be random. The variation of time delay of logic gates causes jitter on the ring oscillator's clock frequency. A ring oscillator is a closed loop of logic gates which includes odd number of inverter(s) which provide(s) an oscillation. There are several methods that use these ring oscillators as the source of randomness. The variation in propagation time is also used in generators with delay elements assembled in an open chain. The chain is used here to increase or adjust total delay [11].

Some generators use jitter of the clock signals that are synthesized in the embedded analog PLLs to generate random numbers. Analog PLLs are common resource for some FPGAs which can be utilized easily. Because RC filters, which is the only analog part of PLL, can easily be realized using the same technology in FPGAs [1]. We can therefore consider a PLL-based TRNG as a generator, which can be implemented in logic devices in general.

In standard logic devices a logic state is acceptable to be in one of two logic states "logical one" and "logical zero". These two states are represented with different voltage levels. In order to resolve these states, there is a forbidden area between these states. While logical state is changing, the voltage level must be passes through this forbidden area. There is a possibility that the logical state is neither logical one nor logical zero in this forbidden area. This state is named meta-stability [12]. The concept of using meta-stability as a TRNG source is not very common for logic devices in literature because manufacturers mostly solve the meta-stability problem by using dedicated hardwired flip flops. On the other hand there is still a practical way to employ the meta-stability such as soft latches implemented in LUTs.

2.1.2 Entropy Extraction Methods

Entropy extraction is a method which is applied to entropy source in order to obtain an output random string. Each kind of entropy source has its own extraction method. In logic devices entropy extraction method includes a sampling mechanism. This mechanism aims to take a sample of randomness when the entropy of the source is high enough. The entropy source and the extraction method must always be considered together because extraction methodology is directly related to the entropy source. Extraction method for each TRNG can be different and have its own pros and cons.

2.1.3 Post Processing

Generally, statistical quality of TRNGs is not high enough for cryptographic applications, because the randomness source of TRNGs is physical processes which have some weaknesses causing rarely the production of non-random numbers (long sequences of zeros or ones). The statistical quality of generated random numbers can be degraded for several reasons. Possible reasons of degradation are [11]:

- Entropy of the source may not be high enough
- Entropy may not be extracted correctly.
- The extracted samples may be correlated.

Even if the entropy of the source is properly extracted, the output of TRNG should be reconsidered before using it. For this reason post-processing is necessary to improve the statistical properties of the generated random numbers.

Post processing algorithms generally aims to reduce bias and/or correlation on generated bits in order to increase the entropy. If the post processing mechanism cannot increase the entropy, it degrades the throughput to increase the entropy per bit.

On the other hand, the post processing algorithms are sometimes used for increasing the throughput of the TRNG. Also these types of stretching algorithms are generally used for pseudo random number generators. In some papers the random number generators which have physical source of entropy and a pseudo post processing are called hybrid random number generators.

Next, we will discuss the most common post-processing techniques[11].

2.1.3.1 The Exclusive-or Post Processing

The exclusive-or (XOR) post processing is a simple linear function. This post processor applies exclusive-or operation on successive n bits in order to produce a single bit.

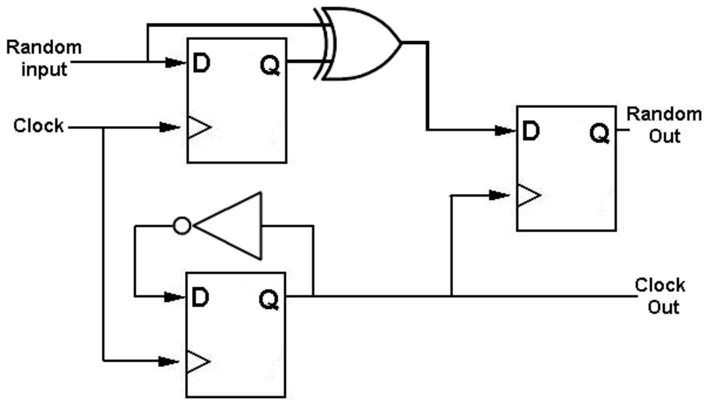


Figure 2-2 - Exclusive-OR Post Processor (n=2)

The XOR post processing dramatically reduces the throughput (1/n times). Therefore the bias on the input stream will also dramatically reduce at the output if the input bits are independent of each other. The important advantages of the XOR post processor are its simplicity and the possibility to maintain a constant output bit-rate.

2.1.3.2 The Von Neumann Post Processing

The Von Neumann post processor is a simple procedure that takes a pair of successive bits and uses the first bit of the pair if the bits are different while throwing away identical pairs as shown in Table 2-1.

Table 2-1 - Von Neumann Mapping Table

Input pair	Output bit
00	Null
01	0
10	1
11	Null

The Von Neumann post processing dramatically reduces the bias on the input sequence except the case that the input sequence has a cycle with period of 2 bits. However the output bit rate of this post process is directly dependent on the input sequence. The non-constant output bit rate is the main disadvantage of this post processing method.

2.1.3.3 Linear Feed Back Shift Registers (LFSR)

A linear feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. LFSRs are commonly used for post processing because of the reasons listed below:

- It is easy to implement an LFSR in hardware.
- Produces sequences with good statistical properties.
- It is possible to analyze it using algebraic techniques.

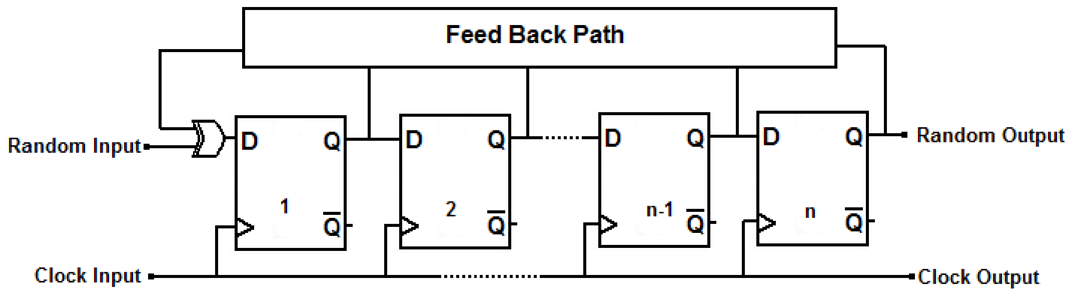


Figure 2-3 - Linear Feedback Shift Register

A LFSR of length n consists of n delay elements (Flip Flops). The clock input of all flip flops are connected to the same clock source and the movement of data through the register is controlled by rising or falling edge of this clock.

For each falling/rising edge of clock:

- First flip-flop gets its input from the feedback path and the random input. This feedback path is a combinational block that applies an exclusive-or operation on the register content according to the feedback polynomial.
- All flip-flops except the first one get their input from the previous one.
- The output is taken from the last flip flop (nth) of the register.

The LFSRs are also used as pseudo random number generators. For using LFSRs as PRNGs the random input of the register is removed. The initial value of the LFSR becomes the seed value of the pseudo random number generator and this value should be updated before the LFSR cycle ends.

2.1.3.4 Resilient Functions

Resilient functions are special functions that are commonly used in cryptography and coding theory. They are suitable for post processing because "the knowledge of any m values of the input to the function does not allow one to make any better than random guess at the output" [13]. These functions are derived as Boolean functions so their implementations are feasible for FPGAs. The main disadvantage of this kind of post processing is that they produce one bit per n input bits. So this post processing method degrades the throughput n times.

2.1.3.5 Encryption of Extracted Random Source

The output of cryptographic algorithms has good statistical characteristics because of the diffusion and confusion properties of cryptographic algorithms. These statistical features of cryptographic algorithms also can be used for post processing. Besides this if a cryptographic algorithm is used as the post processing method of a

TRNG, the variables of the algorithm can be taken both from the entropy source of generator and the system where the TRNG is being used. So the effect of the post process can be changed for each generation. This kind of post-processing block (the cipher or hash function) is relatively complex and expensive, hence the TRNG should re-use (share) the cipher that is used for data encryption.

2.2 Statistical Evaluation and Testing

There are various statistical tests that can be applied to a sequence to confirm if it is truly random. Randomness is a probabilistic property. The properties of a random sequence can be characterized and described in terms of probability. There are numerous possible statistical randomness tests. These tests search a pattern in the input sequence, which indicates that the sequence is nonrandom. While there are so many tests for determining whether a sequence is random or not, there are no specific finite set of tests that can produce an certain decision of being random or not. In addition, the results of statistical testing should be interpreted with care and caution to avoid incorrect conclusions about a specific generator.

The quality of a true random number generator’s output must be evaluated using standard normalized statistical tests. The most commonly encountered tests are the following: FIPS 140-1, FIPS 140-2, NIST statistical test suite and Diehard test suite. We will discuss these groups of tests in the following sections.

2.2.1 FIPS 140-1 and 140-2

The National Institute of Standards and Technology (NIST) is an institute that provides leadership, technical guidance, and coordination of U.S. Government efforts in the development of standards and guidelines. The Federal Information Processing Standards (FIPS) Publication Series is the official series of the NIST. This publication specifies the security requirements that are to be satisfied by a cryptographic module [8].

The FIPS publications classify the cryptographic modules into security levels, i.e., level 1 up to 4, four being the highest. Cryptographic modules that implement a random number generator should generally have the capability to perform statistical tests for randomness. For Levels 1 and 2, such tests are not required. For Level 3, the tests should be called upon demand. For level 4, the tests should be performed at power-up and should also be called upon demand. Four tests that are specified in the next sub sections are recommended by FIPS 140-1 and later in FIPS 140-2 [9]. A single bit stream of 20,000 consecutive bits of a random number generator's output is subjected to each of the following tests. If any of the tests fail, then the module should enter an error state. These tests are "monobit test", "poker test", "runs test" and "long runs test", which are explained below.

2.2.1.1 Monobit Test

The purpose of the monobit test is to check whether the number zeros and the number of ones in a sequence are approximately the same as expected for a truly random sequence. The monobit is a basic test for a random sequence because the passing ratios of the other tests are dependent on this test result.

Implementation of the test:

- Count the number of ones for each 20,000 bit stream. Denote this quantity by X.
- The test is passed if $9{,}654 < X < 10{,}346$

2.2.1.2 Poker Test

The Poker test divides the 20,000 bit stream into 5,000 contiguous 4 bit segments. Then the test counts and stores the number of occurrences of each of the 16 possible 4 bit. Then, *f(i),* which is the number of occurrences of *i*, is used to check if the equation below is satisfied or not.

$$1.03 < \frac{16}{5000} * \left(\sum_{i=0}^{15} (f(i)^2) - 5000 \right) < 57.4$$

2.2.1.3 Runs Test

Run is a terminology used in runs test that corresponds to consecutive occurrence of ones or zeros. A run of length k consists of exactly k identical bits and is bounded before and after with a bit of the opposite value.

The purpose of the runs test is to determine whether the number of runs of ones and zeros of various lengths are in between expected intervals. This test particularly determines whether the oscillation between such zeros and ones is too fast or too slow. Acceptable intervals of run numbers are given in the Table 2-2.

Table 2-2 - Accepted Intervals for Runs Test

Length of Run	**Required Interval**
1	2,267 - 2,733
2	1,079 - 1,421
3	502 - 748
4	223 - 402
5	90 - 223
6+	90 - 223

2.2.1.4 The Long Run Test

A long run is defined as a run of length 34 or more (of either zeros or ones).On a sample of 20,000 bits, the test is passed if there are no long runs.

2.2.2 NIST Statistical Test Suite

The Information Technology Laboratory (ITL) is a major research component of NIST. ITL develops tests and measurement methods, reference data and technical analysis that help advance the development and use new information technology [14]. The ITL‘s special publication NIST-SP800-22 [6] provides criteria for characterizing and selecting appropriate random number generators. It also includes some recommended statistical tests that are useful as a first step in determining whether or not a generator is suitable for a particular cryptographic application. Still, no set of statistical tests can absolutely certify a generator as appropriate for use in a particular application. NIST SP800-22 includes 15 statistical tests in order to evaluate statistical characteristics of the random number generators. These tests are listed in Table 2-3.

Table 2-3 - Tests of NIST SP800-22

	Test Name
1	The Frequency (Monobit) Test
2	Frequency Test within a Block
3	The Runs Test
4	Tests for the Longest-Run-of-Ones in a Block
5	The Binary Matrix Rank Test
6	The Discrete Fourier Transform (Spectral) Test
7	The Non-overlapping Template Matching Test
8	The Overlapping Template Matching Test
9	Maurer's "Universal Statistical" Test
10	The Linear Complexity Test
11	The Serial Test
12	The Approximate Entropy Test
13	The Cumulative Sums (Cuscus) Test
14	The Random Excursions Test

15	The Random Excursions Variant Test

The tests in the list are developed, implemented and evaluated by ITL. ITL also provides ANSI C codes for the test suite. There is no specific order of application for these tests but the frequency test has the highest priority since if it fails then it is highly probable that the others fail too.

The statistical tests are formulated to test a specific hypothesis. For NIST statistical test suite, the hypothesis is that the sequence under a test is random. For each test, a relevant randomness statistic must be chosen and used to determine the acceptance or rejection of this hypothesis. Under the assumption of randomness, such a statistic has a distribution of possible values. A critical value is then determined from the theoretical reference distribution of this statistic, which is obtained by mathematical methods. In order to determine the acceptance or rejection, a statistical value is computed on the sequence under test. Then the computed statistical value is compared to the critical value. If the statistical value exceeds the critical value, the hypothesis for randomness is rejected. Otherwise it is accepted.
There are two possible error conditions for this hypothesis. These are:

- Error Type-1: when the statistical test rejects a sequence that is, in truth, random.
- Error Type-2: when the statistical test accepts a sequence that is, in truth, not random.

Table 2-4 - NIST Error Table

TRUE SITUATION	**CONCLUSION**	
	Acceptance	Rejection
Data is random	No error	Error Type-1
Data is not random	Error Type-2	No error

A statistical randomness test cannot decide certainly whether a sequence is random or not random because of the possibility of these errors. The probability of an Error Type-1 is often called the level of significance of the test and is denoted as α. The level of significance of a test indicates that the sequence under test might have non-random properties even if it is taken from a good generator. Common values of α in cryptography are around 0.01.

Each statistical test of NIST SP800-22 calculates a P-value (probability value) that summarizes the strength of the evidence against the hypothesis. This P-value corresponds to the probability that a perfect random number generator produced a sequence that is less random than the sequence being under test. If this P-value is greater than the level of significance (α), then the sequence under test is accepted to be random. Otherwise, it is rejected.

The NIST also recommends a five step strategy for testing RNGs.

1. Selection of a generator (select a hardware or software based generator for evaluation).
2. Binary sequence generation (for a fixed sequence of length n and the pre-selected generator, construct a set of m binary sequences and save the sequences to a file).
3. Execute the Statistical Test Suite (invoke the NIST Test Suite).
4. Examine the P-value (an output file is produced with relevant values such as P-values for each statistical test).
5. Assessment: Pass/Fail assignment (for a fixed significance level, a certain percentage of P-values are expected to indicate a failure, for example, if α = 0.01, then about 1% of the sequences are expected to fail).

2.2.3 Diehard Battery of Randomness Tests

The Diehard battery of tests was developed and published in 1996 [7]. The battery of the tests is supposed to provide a better statistical evaluation of random sequences in comparison to the original FIPS statistical tests. The test battery consists of 15 different, independent statistical tests that are listed in Table 2-5.

Table 2-5 - List of Randomness Tests in Diehard Battery

	Test Name
1	Birthday Spacing Test
2	Overlapping 5-Permutations (OPERM5) Test
3	Binary Rank Tests (three kinds of rank matrices tests)
4	Bitstream (Monkey) Test
5	Overlapping-Pairs-Sparse-Occupancy (OPSO) Test
6	Overlapping-Quadruples-Sparse-Occupancy (OQSO) Test
7	DNA Test
8	Count The 1s Tests (two kinds of tests)
9	Parking Lot Test
10	Minimum Distance Test
11	Random Spheres Test
12	The Squeeze Test
13	Overlapping Sums Test
14	Runs Test
15	Craps Test

The diehard test suite also generates P-values for each test as the NIST test suite and the results of these P-values are supposed to be uniformly distributed. Unlike the NIST test suite, the test is considered to be successful when the P-value is in range $[(0+\alpha/2)\ ,\ (1-\alpha/2)]$ where α is the level of significance of the test.

More detailed description of these 15 tests and software to perform them are available on the internet link of the Diehard CDROM [7]. This suite requires at least 80 million bits (10-12 Megabytes) of random data for each run of the suite.

2.3 Practical Implementations of TRNGs built in FPGA

In this section some sample practical designs reported in literature of true random number generators built in FPGA are presented and briefly explained.

2.3.1 Basic Ring Oscillator Based Design

The basic ring oscillator based design is proposed first in [13]. The principle of this design is based on sampling phase jitter in ring oscillators. Ring oscillator is a combinational loop of delays, which includes an odd number of inverters. That provides the oscillation in the loop. The instability of the propagation delay of each logic gate in closed loop generates jitter on the ring oscillator clock. This design employs a large number of ring oscillators (114 for the selected Xilinx Virtex-2 FPGA) each composed of 13 inverters. The number of employed ring oscillators is determined according to the measured jitter. The outputs of all ring oscillators are input to a multiple input exclusive-or (XOR) operation in order to get a high-frequency random signal. The output of exclusive-or operation is sampled using a low frequency reference clock. This method is simply presented in Figure 2-4.

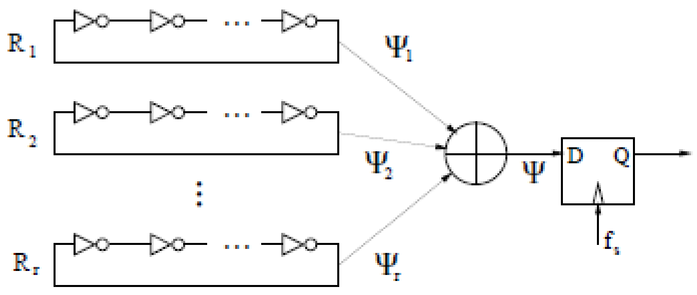


Figure 2-4 - Ring Oscillators Based Design [13]

The digital random output is then post processed using resilient functions. The relationship between the number of oscillators and the randomness of the output is shown in [13]. The weakness of this method is the assumption that the ring oscillators are independent with each other. This weakness is later explained in detail in [15]. The improvements and revisions are proposed in [16].

The advantages of basic ring oscillators based design:

- Technology independent (Suitable for all FPGA families).
- Easy design and implementation.
- Synthesis can be done using fully automated FPGA tools.
- Relatively high and constant throughput.

The disadvantages of basic ring oscillators based design:

- High power consumption (Because of large number of ring oscillators).
- The ring oscillators are probably not totally independent with each other. This causes a correlation on the output of design and degrades the quality of randomness.
- The external manipulations or attacks on the generator will not be detected because of the existence of resilient corrector.
- The power consumption of design and the high fan in of single exclusive or gate can cause excessive local heating.

2.3.2 PLL Based Design

The PLL based design of true random number generator is proposed in [1]. The basic principle behind this method is to extract the randomness from the jitter of the output clock signal of embedded analog phase-locked loop (PLL) in FPGAs. The jitter on the reference clock is sampled by using a rationally related clock signal synthesized in the same on-chip analog PLL. The most important requirement of this method is that the reference signal has to be sampled near the edges influenced by the jitter. The basic structure of the proposed generator is depicted in Figure 2-5.

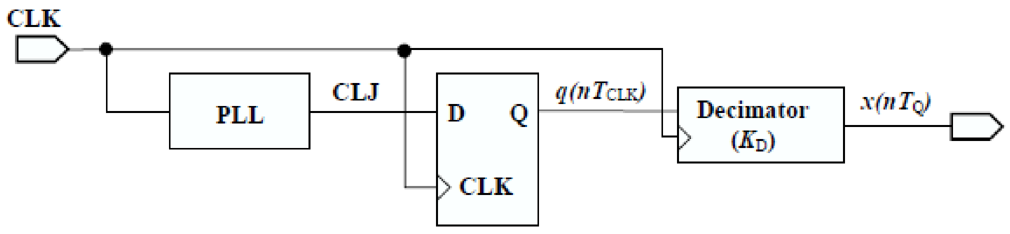


Figure 2-5 - TRNG of Fisher and Drutarovsky [1]

The design includes an on-chip PLL whose clock generation factor is m/d. m is the multiplication factor and d is the division factor of the PLL. The design has also a flip flop (DFF) for sampling the jitter on the reference clock. The last part of the generator is decimator part. The decimator is an n bit buffer where the output of this buffer is input to a multiple input exclusive-or (XOR) operation. The result of this operation generates the random output of the TRNG.

The signal named CLJ is a rectangular clock waveform with the frequency:

$$F_{CLJ} = \frac{\mathrm{m}}{\mathrm{d}} \mathrm{F}_{\mathrm{CLK}}$$

Signal CLJ is sampled by the D flip-flop using the reference clock signal with frequency F_{CLK}. There are d rising edges of CLK signal and 2m edges (rising and falling) of CKJ waveform during time period T_Q.

$$\mathrm{d}\,\frac{1}{F_{CLK}} = \mathrm{m}\,\frac{1}{F_{CLJ}}$$

$$\mathrm{T_Q} = \mathrm{d}\ T_{CLK} = \mathrm{m}T_{CLJ}$$

It has been shown in [1] that if m and d are relative primes (Greatest Common Divisor GCD (m, d) = 1), the set of samples creates an equidistant set of values. The worst-case distance between the two closest edges of *CLK* and *CLJ* during the period T_Q is given as

$$MAX(\Delta_{min}) = \frac{T_{CLK}}{4m} GCD(2m, d) = \frac{T_{CLJ}}{4d} GCD(2m, d)$$

If *m*, *d*, and F_{CLJ} are properly chosen then worst-case distance between the two closest edges of CLK and CLJ can be controlled. Therefore if the intrinsic analog PLL jitter (σ_{jitter}), which is already specified by FPGA vendors is greater than the worst-case distance between the two closest edges of CLK and CLJ, the sampling edge of *CLK* will fall at least once into the edge zone of *CLJ* during each period T_Q [1].

Hence if we sample CLJ for the period T_Q, at least one of the samples will statistically depend on the random jitter. The output is then generated by the decimator which is a bit-wise addition in modulo 2 of d samples and this output will be random.

The randomness of this design was tested by NIST statistical test suite by using 1-Gigabit of continuous TRNG output. The results of tested sequences was within the expected confidence intervals for all tests and *P*-values were uniformly distributed over the (0,1] interval.

The advantages of PLL Based Design:

- No need for post processing.
- Easy design and implementation
- Synthesis can be done using fully automated FPGA tools
- Constant throughput.
- Low power consumption (PLL can be enabled just only for generation)

The disadvantages of PLL based design:

- Low throughput.
- Use of PLLs could be restrictive in some designs (some FPGAs contain only one or two PLLs).
- The external manipulations or attacks on the generator will not be detected because of existence of resilient corrector.

2.3.3 State Machine Based Design

The state machine based design of random number generator is proposed in [17]. The basic structure of this design is presented in Figure 2-6. The design employs two state machines: one of them is a linear feedback shift register (LFSR) and the other one is a cellular automata shift register (CASR). These state machines are clocked by jittery clock signals that are generated by two independent free running ring oscillators.

The employed state machines have different lengths and only 32-bits of these machines are used for generating the output. The employed LFSR is based on a primitive 43 bit long polynomial, which provides a cycle length of (2^{43} – 1). The employed CASR is 37 bit long, which provides a cycle length of (2^{37} – 1). To generate a 32 bit random number, 32 bits of the LFSR and CASR are selected and permuted and added using bitwise modulo - 2 exclusive or operation. The lengths of the state machines are selected to be relative primes in order to get the cycle length of the combined generator close to $2^{\text{Sum of lengths}}$ – 1 (2^{80}).

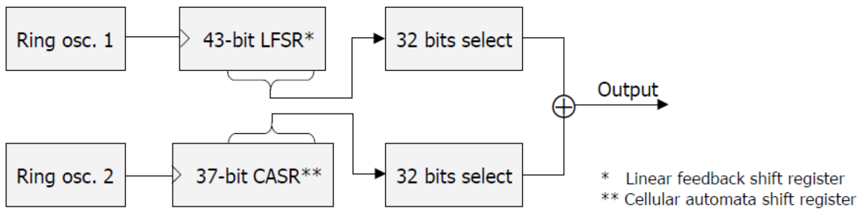


Figure 2-6 - State Machine Design [17]

The initial state of the state machines would be critical if the machines are restarted for each 32 bit output generation. But the ring oscillators that drive state machines are never stopped, even if the generator is not in use. So due to the time drift of the clocks, the LFSR and CASR can be supposed to be in an undetermined state after a sufficiently long period. Also for getting multiple successive random words, the design requires the same sufficiently long time period for each successive words.

The minimum sufficient time for wait between two successive words permits the state machines to pass at least twice their cycle.

This method has been realized for random number generation in some custom silicon chips that are produced by Freescale Semiconductor with the name Random Number Generator Accelerator (RNGA). But Freescale also does not recommend using this generator in high-level data security applications since the output of this generator can be determined from the initial states of the machines theoretically [12].

The advantages of state machine based design:

- No need for post processing.
- Easy design and implementation
- Synthesis can be done using fully automated FPGA tools
- Constant and high throughput.

The disadvantages of state machine based design:

- It is difficult or even impossible to describe the randomness with a mathematical model.
- The structure of TRNG mixes pseudo-randomness with true-randomness.
- True randomness is based on the presence of frequency variation and drift, but absence of true randomness could not be detected because of having pseudo-randomness also.

2.3.4 FIGARO Design

The FIGARO Design of true random number generator is proposed in [18] [19]. Figure 2-7 presents the basic structure of the design. This design contains two special ring oscillators, one Fibonacci and one Galois ring oscillators. These ring oscillators differ from the ordinary LFSRs containing inverters instead of flip flops. The feedback path of the ring oscillators can be expressed using Fibonacci and

Galois polynomials and the ring oscillators are named according to these polynomials such as Fibonacci Ring Oscillator (FIRO) and Galois Ring Oscillator (GARO). Because of the unstable delay time of each inverter, the internal state of these ring oscillators change chaotically and very fast. Hence the internal state of the ring oscillators cannot be predicted after a specified time. In [15], it is stated that each of the rings having linear feedback gives a random output after a period as small as 25ns.

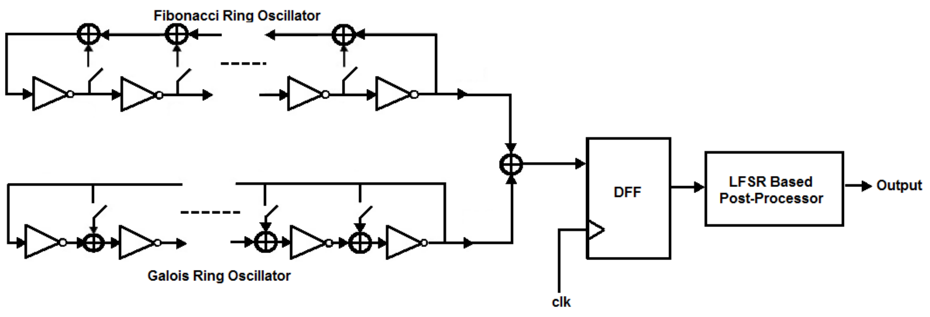


Figure 2-7 - Figaro Design

The FIGARO design combines the random output of the Fibonacci ring oscillator and the Galois ring oscillator with an exclusive or operator. The output of the exclusive or gate is sampled with a sampling frequency that is chosen relatively low to avoid output bit correlations.

Golic [18] has also proposed a method of random data post processing based on self-clock-controlled LFSR. The proposed post processing can be used for randomness extraction and for computationally secure throughput upgrade of input random data.

The advantages of FIGARO Design:

- Easy design and implementation in all FPGA families.
- Synthesis can be done using fully automated FPGA tools.
- Uses relatively few logic sources and requires only logic blocks.
- Constant and high throughput.
- By restarting for each generation, power consumption is assumed low.

The disadvantages of FIGARO Design:

- It is difficult to describe the randomness with a mathematical model.
- The robustness of the generator against attacks is difficult to estimate.
- Entropy on the output of the exclusive or operation is expected to be very high according to [15] on the other hand the bias on the output is reported to be very high [11].
- The output spectrum has some dominant frequencies.
- Some of the implementations in the literature stop running randomly [11].

2.3.5 Metastable Ring Oscillator (Meta-RO) Based Design

The Metastable Ring Oscillator based true random number generator design is proposed in [20]. This design includes a metastable ring oscillator. The metastable ring oscillators are designed to extract the randomness from the metastable condition of the digital devices instead of jitter on the output clock. This new source of randomness is expected to reduce the entropy accumulation time and increase the throughput of the generator.

The basic element of the generator is the metastable ring oscillator which has the ability to be set in metastable mode. The metastable ring oscillators are composed of inverters and each inverter has its own switch. The generator has a control for switching all of the inverters as shown in Figure 2-8.

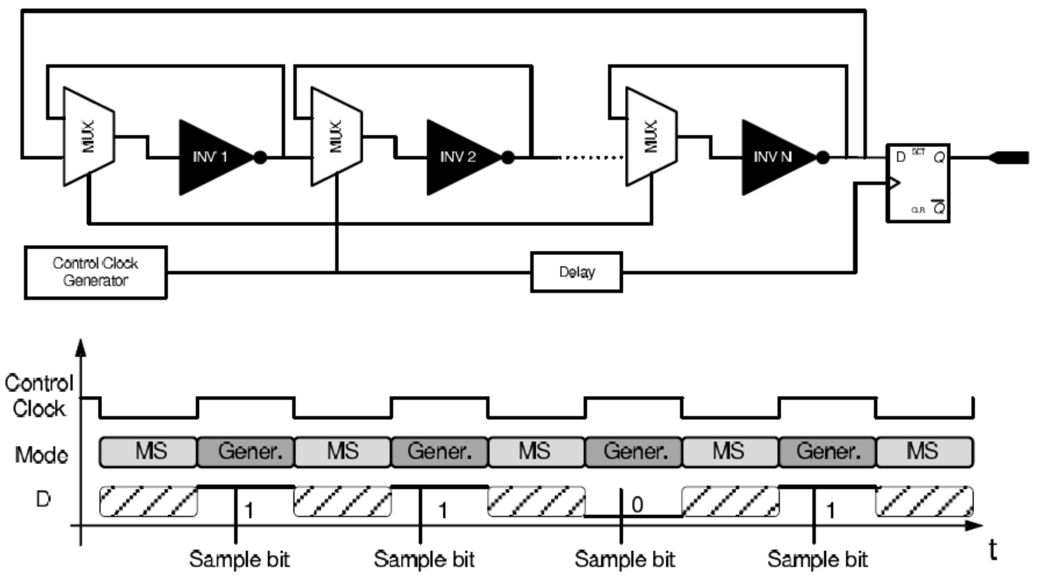


Figure 2-8 - Meta-RO based Design [20]

The generator has two modes of operation. These are metastability (entropy accumulation) mode and oscillating mode. In the metastability mode, the individual switches of all inverters are closed. The output of each inverter is connected to its input and it converges somewhere near a metastability level. This voltage level fluctuates around the metastable level because of thermal noise as long as the switches are closed. In the metastable mode each of the inverter forms an independent randomness source. Whenever the switches are opened the initial state of the ring oscillator is completely determined by from the fluctuations of each inverter.

For the validation of the proposed method, an ASIC design is implemented and simulated in Cadence [20]. The outputs are shown to have passed tests of AIS.31 Class P2 and FIPS 140-1/2. The throughput is reported to be 35-50 Mbits/sec. The estimated area for the design with 65nm semiconductor technology is reported to be 1μm^2 (for Digital TRNG core only).

On the other hand an FPGA implementation is also done using Xilinx XC2V3000-5. This design is shown to have also successfully passed FIPS 140-1/2 and AIS.31 Class P1, but problems exist with AIS.31 Class 2 tests. It is also mentioned that the

FPGA implementation of this method is not very stable compared to ASIC design [20].

The advantages of Metastable Ring Oscillator based design:

- Uses relatively few logic resources and requires only logic blocks.
- Relatively high throughput (35 Mbits / sec for the ASIC design).
- High entropy source.

The disadvantages of Metastable Ring Oscillator based design:

- It is difficult to describe the randomness with a mathematical model.
- Manually intervention to FPGA tools is required for implementation.
- Weaknesses against temperature and voltage changes on the generator, which can cause degradation on the statistical characteristics of the output bit-stream.
- Robustness against attacks is questionable.

2.3.6 Metastable Flip Flop Based Design

The metastable flip flop based true random number generator is proposed in [21]. Generating metastable condition on a standard flip flop is a challenging issue for FPGAs because vendors solve metastability problems in general on their products. In [21], the authors propose a method to generate metastable condition that utilizes programmable delay lines (PDL), which alter the propagation delay in a controlled fashion. The method uses PDLs to equalize the signal arrival times to flip flops accurately. These PDLs have the capability of adjusting the delay of signal with a resolution of pico second. The method has an adaptive feedback mechanism that tunes the PDL according to the probability of the output bits being monitored. Whenever a small bias is detected on the output, the PDLs are reconfigured to put the flip flop in metastable region again. The monitoring module compensates the effects of environment changes (temperature, voltage, etc.). The general structure of the proposed design is shown in Figure 2-9.

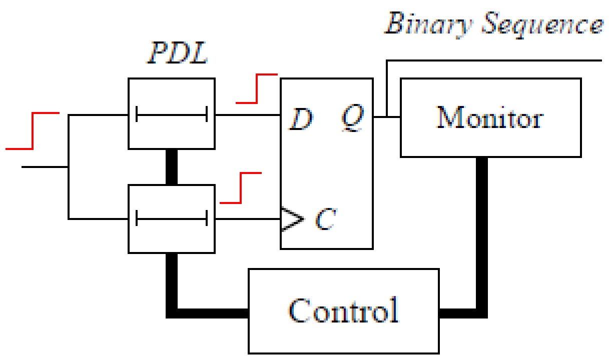


Figure 2-9 - Metastable Flip Flop Design [21]

In [21], the design is implemented on Xilinx Virtex 5 FPGAs. Throughput is reported 2 Mbits/sec with a von Neumann corrector. And NIST randomness tests are passed with high rates.

The advantages of metastable flip flop based design:

- Relatively high throughput
- Uses very few logic resources.
- Low power consumption.
- Robust for external changes and attacks

The disadvantages of metastable flip flop based design

- Manual placement is needed in order to ensure that the generator is in proper operation in all circumstances.
- FPGA family and even model specific design is required.
- Not feasible for some FPGAs.

2.3.7 Transition Effect Ring Oscillator Based Design

The transition effect ring oscillator (TERO) based true random number generator is proposed in [22]. The design includes a new high-entropy digital circuit named transition effect ring oscillator that can be realized in FPGA. TERO is a kind of bi-stable Flip-Flop (FF) with intentionally lengthened feedback paths. The generator extracts the random bit while TERO is resolving a metastable event. The structure of TERO is shown in Figure 2-10.

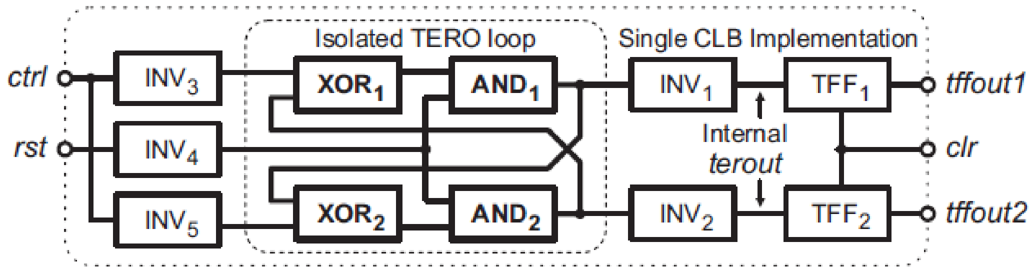


Figure 2-10 - TERO [22]

The TERO design includes two XOR gates that can act like inverters or buffers according to the ctrl signal. If ctrl = ‘1’ both XOR gates act like inverters and if ctrl = ‘0’ then the two XOR gates act like buffers. For these two conditions there should be no oscillation on the loop but at the rising edge of the ctrl signal two XOR gates both act as inverters and try to change their outputs. This action disturbs the steady state behavior of the loop and a pulse rises in the loop. This pulse disappears in a small transient period. The randomness is extracted from this transient effect of TERO loop using T-Flip Flops (TFF). TFFs resolve if the loop made even or odd number of oscillations. The “rst” and “clr” signals are used to initialize TERO to zero for each generation, which prevents correlation between two successive bits.

The implemented design, in [22] produces random bits with a throughput of 250 kbps. The generated bit streams are confirmed by the NIST test suite without any complex post processing.

The advantages of TERO based design:

- Uses resources common to all FPGA families.
- Mathematical model has been presented in the paper.
- Uses very few logic resources. (2 CLBs of Xilinx Virtex-5).
- Low power consumption.

The disadvantages of TERO based design:

- Manual placement is needed in order to ensure the generator proper operation in all circumstances.

2.3.8 Crosstalk Effect Based Design

Cret et. al. [23],[24] have presented a new method of implementing TRNGs in FPGAs. The method is based on using the logic resources of the FPGA close to its maximal capacity in a given region, either globally or locally and exploiting the interconnection network as intensely as possible. The authors has experienced that this kind of heavy load operation causes crosstalk at the dedicated carry chains which are present in each Xilinx FPGA. The carry chains, are most common type of fast dedicated lines between neighboring logic cells, which allow creating arithmetic functions efficiently. The crosstalk begins to occur after the threshold, which is a fraction of the carry chain length, is exceeded. Cret et. al. [23],[24] proposes a design that utilizes this kind of crosstalk as an entropy source, which can be exploited to obtain a high quality TRNG.

The architecture in this design consists of a chain of inverters, which is driven by the system clock as shown in Figure 2-11. The outputs of the (n + m) inverters represent currents flowing through the chain line rapidly. Data from the outputs of inverters is added to a counter value and the result is collected in an accumulator. After a threshold is exceeded by increasing the m, the final result obtained in the accumulator becomes different at the end of each run of a fixed amount of clock cycles due to crosstalk and other electrostatic or magnetic interferences that appear in the interconnection network.

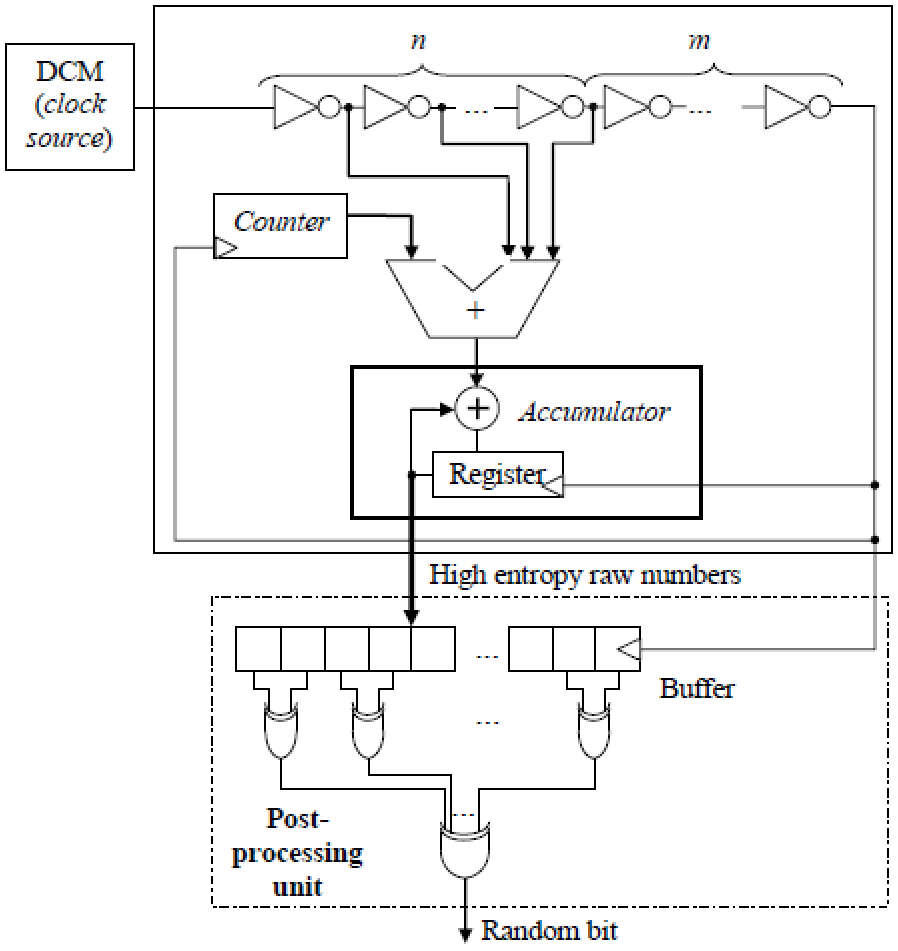


Figure 2-11 - Crosstalk based Design [24]

The proposed design has been proven to provide high quality random numbers satisfying statistical test suites such as NIST and TestU01. The provided throughput of the proposed design is also shown to be quite high (up to 0.7Gbps for Spartan3E100 FPGA) [24].

The advantages of the crosstalk based design:

- Very high throughput.
- Entropy level increases with the temperature or any other extreme condition which could influence the FPGA [24].
- The disadvantages of crosstalk based design:
- Dedicated to Xilinx FPGAs.

- FPGA is required to be used at a relatively large capacity levels.
- Manual placement is required for exploiting the interconnection intensely.
- High power consumption.
- There is no mathematical model for the entropy source.

2.3.9 Write Collisions of Memory Blocks Based Design

An alternative method for implementing TRNGs in FPGAs, which is based on write collisions in dual-port block memories (BRAM) appeared in [25],[26]. The author reports that the write collision of block rams provides efficient entropy to be employed in cryptography or for other security issues such as device identification and true random number generation [26]. When a write collision occurs on a field of memory cell, the cell is likely to remain in a metastable state before it goes into a stable state again. The stable states of the cell are influenced by the respective drivers, adjacent components, manufacturing and process anomalies, thermal vibrations of materials and other minor factors [25]. These external and internal factors in FPGA obviously affect some individual bits of the memory cell. These individual memory cells are then employed as the entropy source for the proposed design which provides a fast and robust true random number generation method with a throughput of more than 100 Mbits per second. The randomness quality is extended by post processing and confirmed by Diehard and standardized BSI AIS 31 test suites. This random number generator can also be instantiated many times on contemporary FPGA devices to support even higher throughputs of random data (>100 Mbits/s).

The proposed TRNG employs an evaluation design that is shown in Figure 2-12 for distinguishing the memory bits of block memories which utilizes sufficient entropy after the write collisions. The evaluation design can drive all input lines of the memory to either zero or one at the same time. Furthermore, a finite state machine (CTL) performs repeated queries on all bits at each memory address. By using this design the TRNG determines which of the individual bits of the memory blocks are suitable for entropy generation. This determination is done by monobit test with 2^{15} tries of collision measurements. After this evaluation process the TRNG uses the chosen bits of the memory block for write collision which is the entropy source of the proposed design.

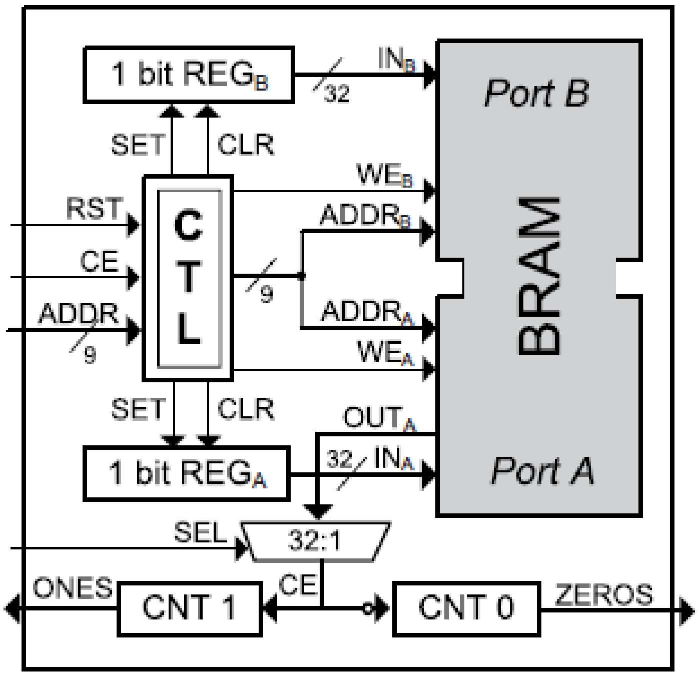


Figure 2-12 - Bram Based Design [25]

The advantages of write collision of BRAMs based design:

- Relatively high throughput.
- The throughput can be increased by employing multiple TRNGs in the same FPGA.
- Relatively low resource usage but BRAMs are valuable resource of FPGAs.
- Low power consumption.

The disadvantages of write collision of BRAMs based design:

- Device specific characteristics of TRNG.
- Potential risks of defects on the used BRAM because of write collisions
- There is no mathematical model for the entropy source

- The employed post processing method can hide the abnormalities of the generator.

2.3.10 Coherent Sampling Design

The coherent sampling design of true random number generator is proposed in [5]. The randomness source of the proposed design is the intrinsic jitter contained in ring oscillator clocks. The generator can produce random bits at speeds up to 0.5 Mbits/second with good statistical characteristics. The design uses resources commonly encountered in all FPGA families. The generated output bit streams are also confirmed by NIST Test suite [5].

The source of randomness of this design is the ring oscillator's clock jitter. Jitter on the ring oscillator clocks arises from unstable propagation delays of the logic gates that are included in the ring oscillator's loop. The method employs two ring oscillators in order to generate two clock signals with jitter. The critical part of the clock generation is that the generated clocks must have different but very close periods. In order to extract the randomness from the intrinsic jitter of ring oscillator clocks, coherent sampling method has been used.

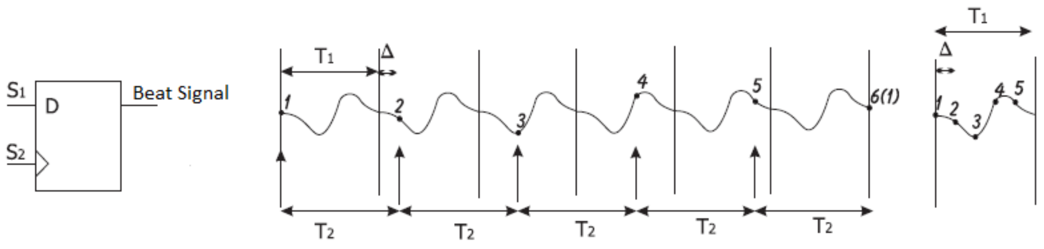


Figure 2-13 - Coherent Sampling [27]

The coherent sampling (see Figure 2-13) is a method which samples a periodic signal S_1 with another periodic signal S_2 where the period of S_1 (T_1) is slightly different than the period of S_2 (T_2). The output signal of the sampling (beat signal) will also be periodic with a period that is inversely proportional to period difference of the signals. Beside of this, the period of beat signal is equal to integer multiple of

T_2. The size of beat signal will be a random count of periods T_2 because of the unstable T_1. In order to get the random bit from the jitter, it is sufficient to count the period of T_2 during one period of beat signal [27].

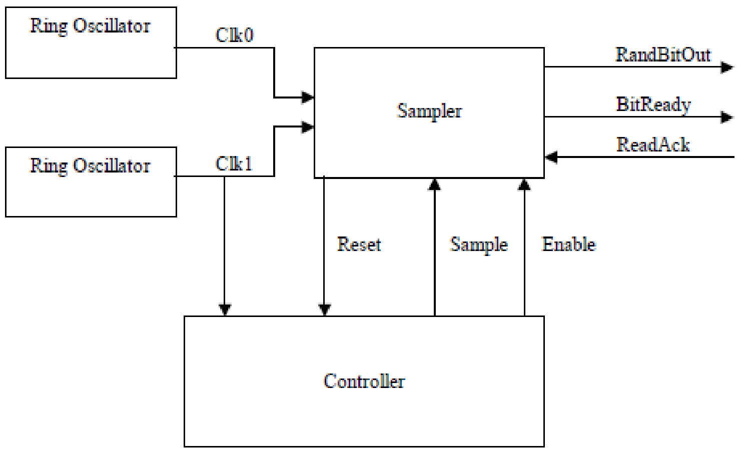


Figure 2-14 - Coherent Sampling Based TRNG [5]

The general structure of the coherent sampling design based TRNG is presented in Figure 2-14. This design contains two separate ring oscillators, sampling module and controller module. Ring oscillators supplies two clock signals to the sampler. The sampler module extracts random bit from the jitter by the help of the controller module using coherent sampling method.

The most critical part of the generator is the requirement of that the two generated clock frequencies have to be close but not the same for generating high quality random numbers. The period difference of these two clocks should be tens of pico seconds. In order to solve this problem the ring oscillator components must be manually placed in the FPGA to the close CLBs.

The advantages of coherent sampling design:

- Uses common resources of all FPGA families.
- Mathematical model is feasible (not given in the original paper)
- Uses very few logic resources.

- Low power consumption.

The disadvantages of coherent sampling design:

- Manual placement is needed in order to ensure the generator proper operation in all circumstances.
- The output bit-stream required a post-processing in order to reduce the bias.

Table 2-6 - Comparison Table of TRNG Methods

	Ring Oscillator	**PLL**	**State Machine**	**FIGARO**	**META-RO**	**Metastable FF**	**TERO**	**Crosstalk Effect**	**BRAM**	**Coherent Sampling**
Can be realized in all kind of FPGAs	Yes	No	Yes	Yes	Yes	No	Yes	No	No	Yes
Automated Synthesis	Yes	Yes	Yes	Yes	No	No	No	No	Yes	No
Design Simplicity	High	High	Mid Range	Mid Range	Low	Low	Low	Low	Low	High
Throughput	High	Low	High	High	High	High	Low	Very High	High	High
Power Consumption	Very High	Low	High	Low	Low	Low	Low	High	Low	Low
Post Process Requirement	Yes	No	No	Yes	No	No	No	No	No	Yes
Math Model Availability	No	Yes	No	No	No	No	Yes	No	No	Yes
Required FPGA resource	Very High	Low	Mid Range	Mid Range	Low	Low	Very Low	High	Low	Low
Randomness Quality	High	High	Low	Low	High	High	High	High	High	High
Robustness	Low	High	High	Low	Low	High	High	High	High	High

As a result of the previous comparative table, we have chosen to implement coherent sampling technique because of its advantages especially the use of resources that are commonly encountered for all kind of FPGAs and the design simplicity. The next chapter is prepared for a deeper understanding of the method and presenting its implementation details. Our implementation has the same features of the originally proposed design. We also used NIST statistical test suite and Diehard battery of tests to evaluate the randomness quality of our implementation. The generated one Gigabit output of our implementation is used and confirmed by both of the test suites. In addition to the design proposed in [5], we have implemented the embedded tests of FIPS 140-1 in the same FPGA. By implementing these embedded tests, our design gained a concurrent control of randomness feature, which makes the design more secure and reliable for external attacks and environment changes.

CHAPTER 3

IMPLEMENTATION OF TRUE RANDOM NUMBER GENERATOR

In this chapter, we will present the implementation details of the TRNG that we have realized on Virtex-5 FPGA with Xilinx ML507 Development Platform. We have chosen to implement coherent sampling method which was briefly overviewed in the first chapter because of its advantages especially its design simplicity and very low resource requirement. Besides, the coherent sampling method employs a robust high quality randomness source that can be realized by using commonly encountered FPGA resources. We have also implemented the embedded tests published by FIPS 140-1 [8]. We have designed the TRNG and its embedded tests in a compact module and described it in VHDL language. We have also confirmed our generated random numbers with the NIST statistical test suite. In the following parts of this chapter we will report on the obtained experience and the engineering challenges encountered during this implementation. Our TRNG design will be explained hierarchically in a top-down fashion.

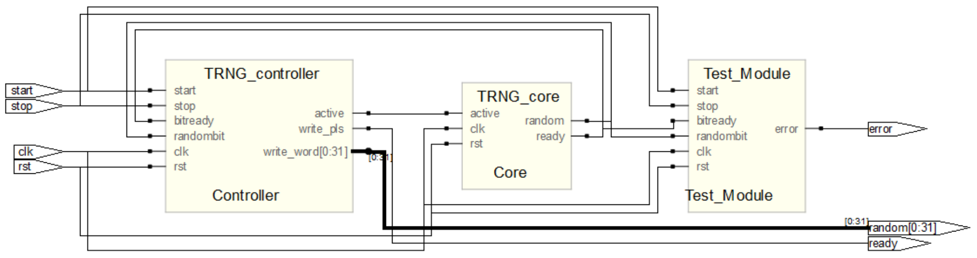


Figure 3-1- TRNG Design (Top view)

Our design includes three main modules that are named as TRNG core, TRNG controller and Test module. The TRNG core is the heart of the generator where the randomness source, extraction method and the post processing are realized. Test module includes the embedded tests which are defined in FIPS 140-1 standard. These tests run concurrently while the random numbers are generated. The controller module controls the test module and the TRNG core module. It also provides an interface to an upper TRNG driver. Block diagram of the top level design of our TRNG is illustrated in RTL schematic as an output of Synplify Pro synthesis tool in Figure 3-1.

3.1 TRNG Core Module

The TRNG core module is the most critical part of the generator because the employed randomness source, extraction method and post processing are all realized in this module.

The module has three inputs that are named “rst”, “clk” and active. The “clk” signal is the interface clock of the TRNG, which should be supplied from an upper control layer of the TRNG. The period of this clock signal is critical because of the preset time configuration of the sampler module. This requirement is explained in the design details of the sampler. We have implemented the controller and the sampler for 50 MHz input clock signal. The input signal “rst” can be used to stop the clock generation of the ring oscillators while the TRNG is in reset mode. The active input of the TRNG is used to enable or disable the controller module. The TRNG core also has two output ports that are named as random and ready. Whenever the TRNG core generates a new random bit at its random port, it generates a ready pulse that to indicate this event.

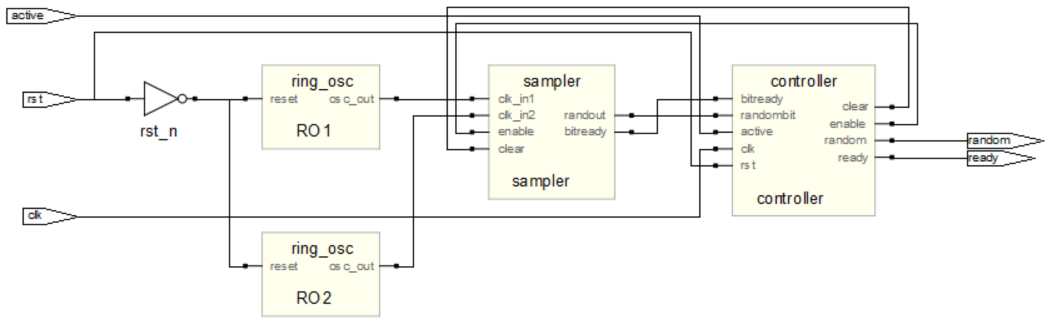


Figure 3-2 - TRNG Core Sub Modules

The TRNG core module includes three sub modules that are named as ring oscillator, sampler and controller as shown in Figure 3-2. Two instances of a ring oscillator are employed in the TRNG core which has close but non-identical oscillation frequencies. We use these two ring oscillators to supply streams of pulses to the sampler logic. The sampler logic extracts the random bit by the help of the controller and produces “bitready” signal for every random bit. The controller module drives the sampler module to get true random bits. Generated random bits are post processed by the controller. The controller generates ready pulse for every post processed random bit. The operation and the implementation details of each module are explained below.

3.1.1 Ring Oscillators

A ring oscillator is a combinational loop of delay elements (logic gates), which contains an odd number of inverters. The employed inverter(s) in the loop causes an oscillation whose period is the total propagation time in the loop. The propagation delay of logic gates in FPGAs is unstable because of the physical processes of the underlying technology. This instability reveals itself as jitter on the generated clock signal of a ring oscillator.

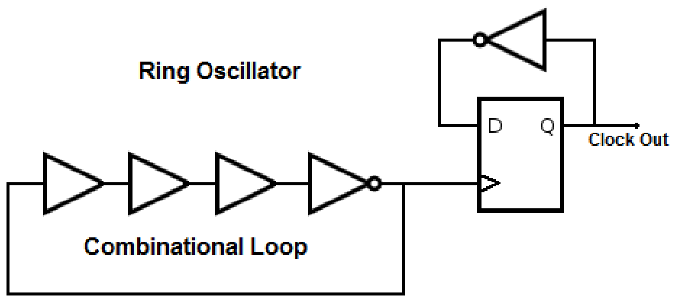


Figure 3-3 - Ring Oscillator

The frequency of the ring oscillator clock signal is directly dependent on the number of gates employed in the loop. On the other hand the entropy of the ring oscillator's clock jitter is also related to the number of delay elements employed in the loop. Considering these and a couple of experiments together, we have chosen to use three buffers and one inverter for the construction of our ring oscillator. The oscillation on the loop is supplied to a toggle flip flop in order to obtain a better square shape clock signal. The general structure of our ring oscillator is shown in Figure 3-3.

The ring oscillators are asynchronous designs. Asynchronous circuits are not commonly encountered in FPGA based designs. VHDL implementation of a ring oscillator is not an ordinary VHDL code, since the vendor's synthesizing tool or the mapping tool may prune the delay elements in the loop automatically. In order to prevent this, we construct the ring oscillator in Xilinx schematics using "KEEP" constraint [28]. Using [29], we have determined the primitive elements, associated with KEEP attribute with each and then we have describe the connections in VHDL as shown in Appendix-A. "KEEP" is an advanced mapping and synthesis constraint which is used with the name of the net that one wants not to be pruned [28]. For every FPGA family, a similar design attribute can be found to implement ring oscillators.

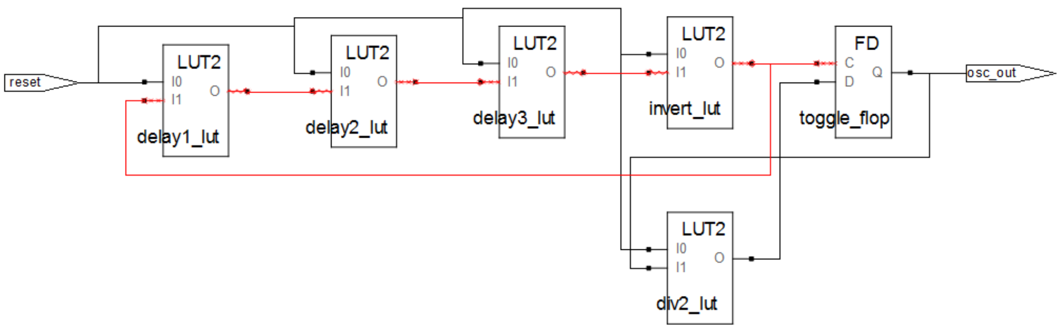
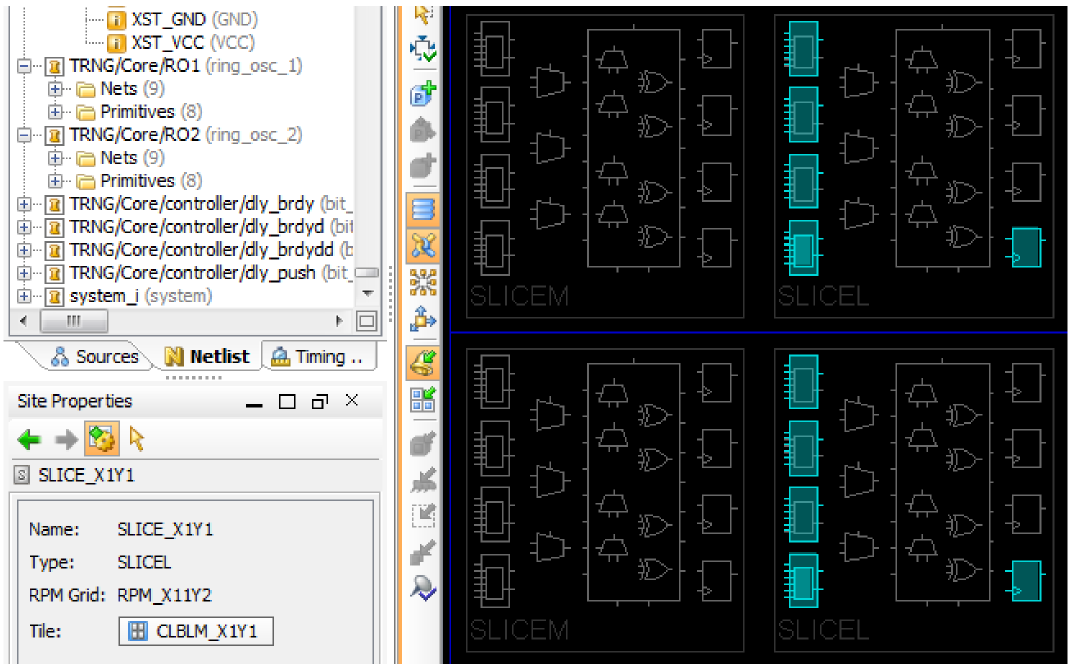


Figure 3-4 - Ring Oscillator Implementation Schematic

The RTL schematic of our ring oscillator is presented in Figure 3-4. LUT2 and FD modules in the schematic are Xilinx primitive design elements. The reset input of the ring oscillator is connected to all LUT2 elements and it resets the outputs of all LUT2s. Each LUT2 primitive that are named as delay1_lut, delay2_lut and delay3_lut is configured to behave like a buffer. But each LUT2 element that are named as invert_lut and the div2_lut is configured to behave like an inverter. The FD is the regular D Flip Flop.


XST_GND (GND)
XST_VCC (VCC)
TRNG/Core/RO1 (ring_osc_1)
Nets (9)
Primitives (8)
TRNG/Core/RO2 (ring_osc_2)
Nets (9)
Primitives (8)
TRNG/Core/controller/dly_brdy (bit_
TRNG/Core/controller/dly_brdyd (bit
TRNG/Core/controller/dly_brdydd (b
TRNG/Core/controller/dly_push (bit_
system_i (system)

Site Properties

SLICE_X1Y1

Name: SLICE_X1Y1
Type: SLICEL
RPM Grid: RPM_X11Y2
Tile: CLBLM_X1Y1

Figure 3-5 - Plan Ahead Screen Shot

Our implemented ring oscillator fits into a single slice of Virtex – 5 FPGA as shown in Figure 3-5. The generated clock frequency of ring oscillator is around 195 MHz. For our purpose the frequencies of the generated clocks should be close to each other but not identical. This can be achieved by placing each oscillator into individual slices that are very close to each other. Hence we place the ring oscillators manually into two separate slices that are adjacent to each other by using the Plan Ahead tool of Xilinx ISE Project Navigator. The coordinates of the slices that are used for ring oscillators are arbitrarily chosen as Slice_X1Y0 and Slice_X1Y1. This constraint of placement can be done by using "LOC" attribute of Xilinx as shown below.

```
Attribute LOC: string;
Attribute LOC of RO1: label is "SLICE_X1Y0";
Attribute LOC of RO2: label is "SLICE_X1Y1";
```

In addition, we have placed each element of a ring oscillator in to a specific location in the slice for providing the same conditions in each ring oscillator. We have first done the manual placements by using Plan Ahead visual interface. Then we described these manual placement constraints in the VHDL code using "BEL" attribute of Xilinx as shown below. "BEL" is an advanced placement constraint, which locks the logical symbol into a specific site of the slice. This means that if these constraints are used for a LUT, this LUT will only be placed in the specified site of the slice.

```
attribute bel : string ;
attribute bel of delay1_lut : label is "D6LUT";
attribute bel of delay2_lut : label is "C6LUT";
attribute bel of delay3_lut : label is "B6LUT";
attribute bel of div2_lut    : label is "A6LUT";
attribute bel of invert_lut  : label is "A5LUT";
```

We monitored the clocks of our implemented ring oscillators with an Agilent MSO6104A oscilloscope. The screen shot of the oscilloscope can be seen in Figure 3-6. The frequencies of ring oscillators are around 195 MHz and the jitter can be observed easily. The measurement is done when the oscilloscope is in edge triggered mode and the trigger is on CLOCK2 signal.

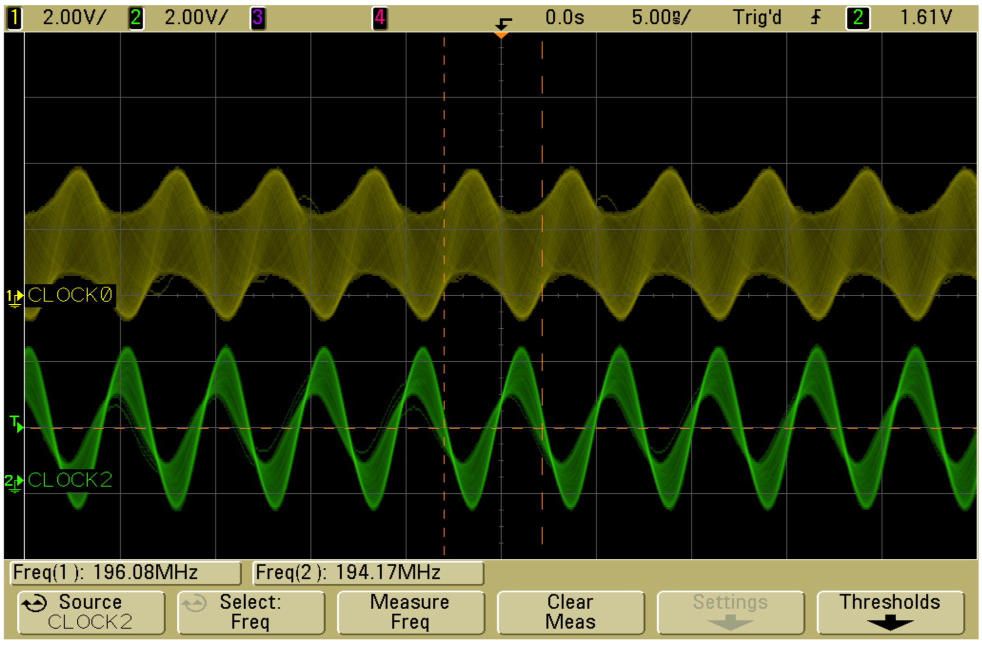


Figure 3-6 - Ring oscillator clocks

3.1.2 The Sampler Module

The sampler circuit extracts randomness from the jitter on the input ring oscillator clocks by using the coherent sampling method. In order to use coherent sampling the frequency of the input clocks have to be close but non-identical. So if we sample one of them with the rising edge of the other one, the sampled output signal (beat signal) will have long sequences of 1s or 0s because of period drifting of the clocks. Due to jitter noise on input clock signals, one period of the beat signal includes many numbers of cycles of the sampling clock, this number being a random quantity. The sampler circuit counts the number of cycles that the beat signal is not changing by using a single bit counter (T-flip flop). The output of the counter flip flop has a random bit for each rising edge of the beat signal. The sampler module that extracts the random bit mentioned above is presented in Figure 3-7.

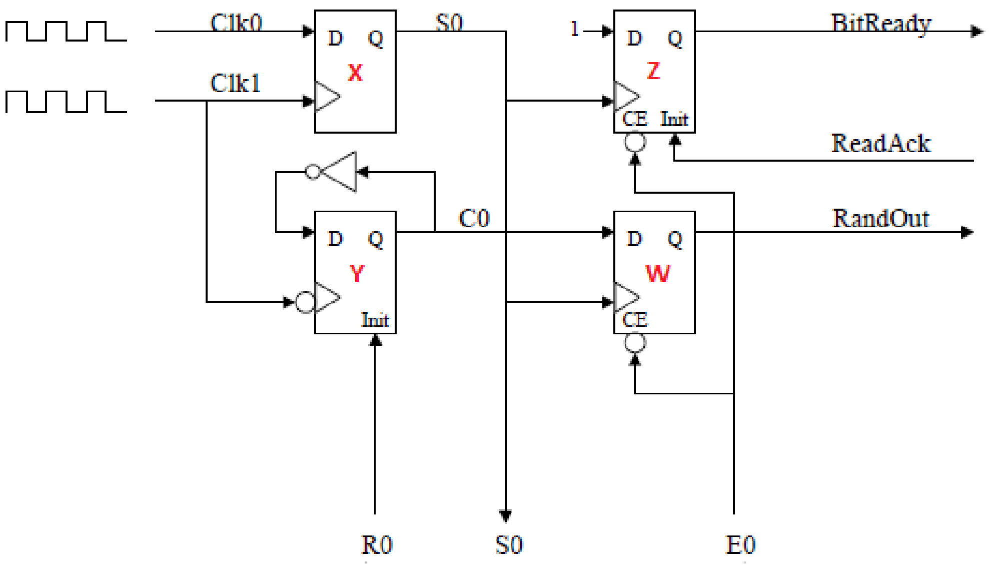


Figure 3-7 - Sampler Design [5]

The sampler circuit contains four flip flops that are denoted as (X,Y,Z,W) in Figure 3-7. Flip flop X is used to sample clk0 with the rising edge of clk1. The output of this flip flop is the beat signal in coherent sampling method and is denoted as S0. During the sampling process the toggle flip flop (Y) toggles its output for each falling edge of clk1. Sampler module utilizes this flip flop for counting the cycles of clk1 during one period of S0 in modulo 2. This toggle flip flop can be reset by the signal denoted R0. It is a critical need of the sampler mechanism to reset this flip flop before each generation in order to prevent the correlation between two successive generated bits. The output of counter flip flop is denoted as C0. Flip flop W latches the C0 signal for every rising edge of S0, meaning that the number of cycles passed during one period of S0 is latched at the end of the period. This is now the generated random bit for this period of the beat signal (S0). The output of Z flip flop (bitready signal) gets high for each newly generated bit and this flip flop can be cleared only by an external signal (ReadAck). E0 is the enable input of the sampler circuit which is used for disabling W and Z flip flops and for keeping the random bit unchanged before it is read from outside. After each rising or falling edge of the S0 signal, C0 signal has many state changes with short cycles because of jitter noise. Therefore after the first rising edge of S0, enable signal (E0) should be made low by the controller module to keep the random bit unchanged. The controller then read the random bit while keeping the enable signal at low for a predetermined period in order to wait for S0 to settle up. To let the sampler generate the next random bit the controller module of the TRNG core should send "ReadAck" and re-enable flip flops W and Z.

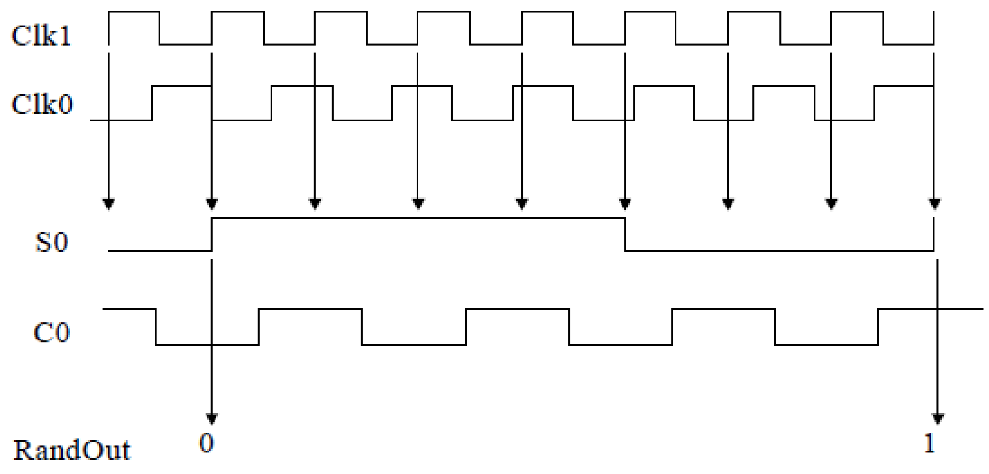


Figure 3-8 - Sampler Signals

Figure 3-8 illustrates the timing of the signals. It is seen that C0 toggles for each falling edge of clk1 and S0 has a long period relative to clk1. The random bit is latched with the rising edge of S0.

In our implementation of the above simple sampler we have used schematic design features of Xilinx as we have done for the ring oscillator case. For the schematic design of the sampler we used LUT2 and FDCE primitives in order to construct our sampler module. The FDCE primitive is a D flip flop with chip enable and clear inputs features. The Synplify schematic diagram of our sampler is shown in Figure 3-9.

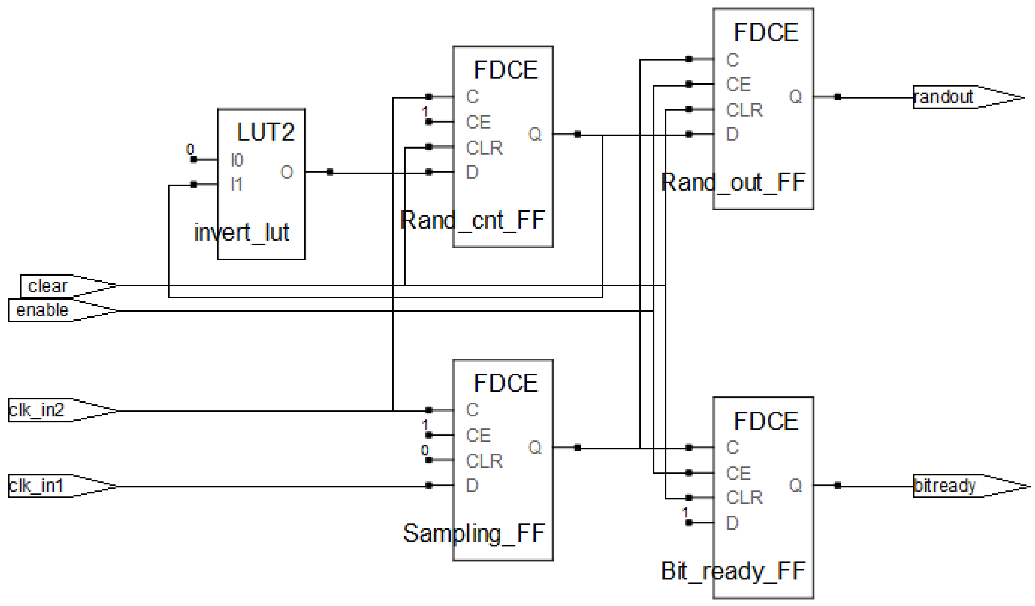


Figure 3-9 - Sampler Module Synplify Schematic

In the implementation, X, Y, Z and W flip flops are denoted with sampling_FF, Rand_cnt_FF, Bit_ready_FF and Rand_out_FF, respectively. The enable input of the circuit is the same as E0 in our implementation. We combined R0 and ReadAck signals also because they work with the same polarity for random bit generation. We also used a clear input instead of R0 and ReadAck signals.

While implementing the sampler we utilized “BEL” and “LOC” constraints as in the ring oscillator case. These constraints and the VHDL description of the sampler module are presented in Appendix – B.

During the implementation of ring oscillator we have observed that the jitter of the output clock can be affected by the circuit that is employed in adjacent slices of the ring oscillator. Because of this the quality of the generated numbers decreases whenever the ring oscillators of our TRNG is placed into slices that are surrounded with circuit that are used for other purposes. In order to overcome this problem, we placed the TRNG core, which contains the ring oscillator and the sampler circuit on one of the corners of the FPGA. Besides, we have also surrounded our compact

TRNG core design with a slice fence that is prohibited to be used for other purposes as shown in Figure 3-10.

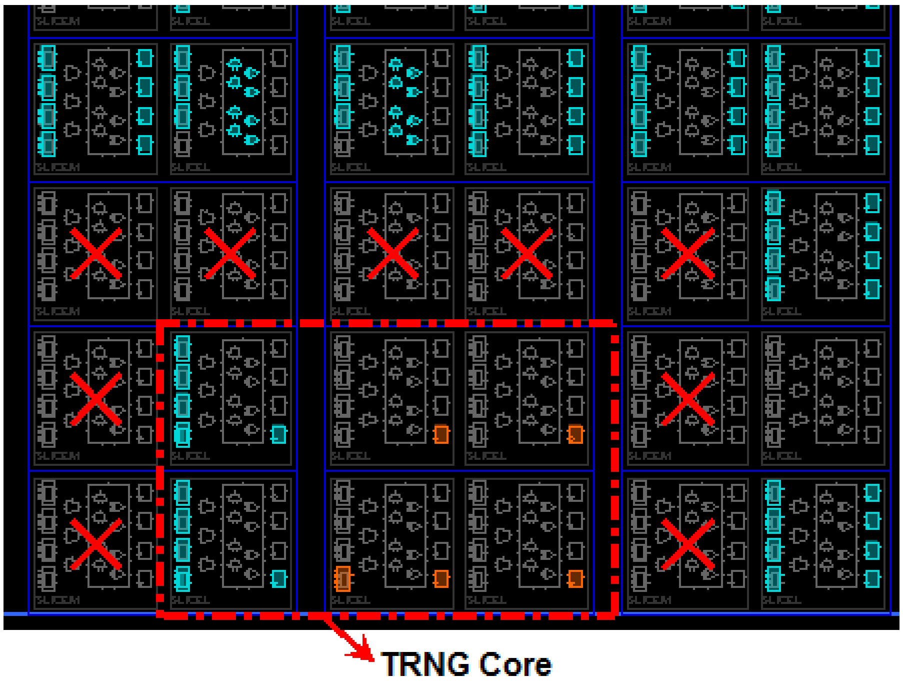


Figure 3-10 - Prohibit fence of TRNG core

This prohibition of usage of the slices can be done by the "PROHIBIT" constraint that is shown below.

CONFIG PROHIBIT = SLICE_X0Y0;

CONFIG PROHIBIT = SLICE_X0Y1;

CONFIG PROHIBIT = SLICE_X0Y2;

CONFIG PROHIBIT = SLICE_X2Y3;

3.1.3 The Controller Module

The controller module senses the output control signals of the sampler and drives the sampler for sampling correctly. The controller manages enabling or disabling of the sampler and clears the counter flip flop for each run. In order to generate a single random bit, the controller enables the sampler and waits for the bitready signal. Whenever a random bit is latched by the sampler, the bitready signal goes high and the controller disables the sampler and then reads the random bit. After reading, it waits for a predetermined amount of time (3 clock cycles of 50MHz in our case) for re-enabling the sampler. In this way the controller forces the sampler circuit to ignore the short beat cycles that occur on both rising and falling edges of the beat signal. Following this preset waiting time the controller re-enables the sampler and resets the counter value and bitready signal for the next bit generation, cycle. The controller prevents correlation between successive bits by resetting the counter flip flop for each new generation operation.

Meanwhile the controller uses an exclusive - or corrector in order to prevent the bias on the output. Two successive incoming bits are input to the exclusive or corrector, which halves the throughput of the generator but increases the quality of its statistical properties. The controller generates a ready pulse for every generated and post processed random bit. The random bit and ready pulse of that bit is connected to the output ports of the TRNG core module.

3.2 Embedded Test Module

The randomness of a TRNG is generally confirmed by statistical test suites before it can be employed by a cryptographic system. However these tests are generally executed offline using output streams of a TRNG. It is equally important to care about whether any environmental change (such as temperature, voltage, current requirement of FPGA) could decrease the randomness quality of the employed TRNG. Besides, the quality of a TRNG can also be degraded by external attacks, which are directly targeting the TRNG. The generated random numbers that are used during the real operation of the system are usually not tested with statistical randomness tests. Because these tests are complex and heavy to be executed for each random number set generated online. Statistical tests are generally executed at the start up of the system and are repeated later periodically. NIST has proposed a subset of its statistical tests to be realized in embedded hardware in FIPS 140-1.

These tests have already been explained in chapter 2. We have implemented these tests also in our TRNG as was illustrated in Figure 3-11. The implementation of these tests occupies 827 slices of the Virtex-5 (xc5vfx70t) FPGA.

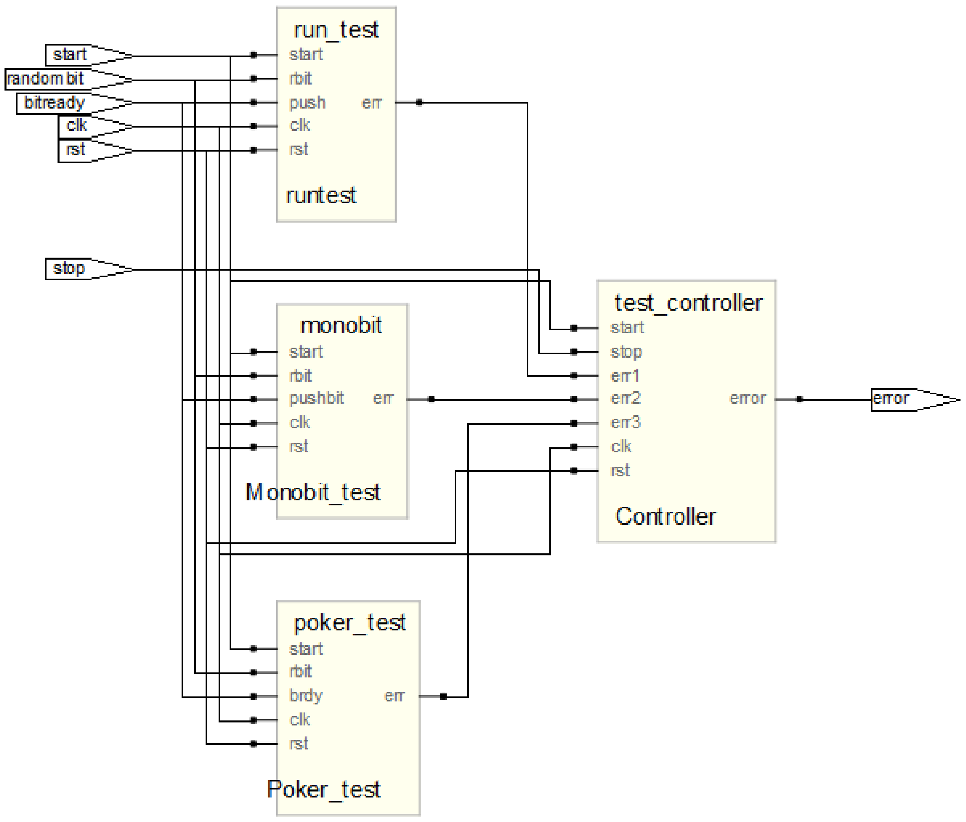


Figure 3-11 - Embedded Test Module Schematic (Top View)

FIPS 140-1 contains four basic randomness tests that are feasible to implement in FPGA. These tests are monobit, poker, run and long run tests. In our TRNG these tests are running concurrently with generator module. The test module generates an error signal for failure of any of these tests. The upper layer TRNG controller can consider this error signal in order to determine whether or not it should use the last set of random numbers generated. Even for some systems, this error signal can be considered as an alarm that indicates an external attack on the system.

The proposed tests of FIPS 140-1 require 20.000 bits of random data for each run of the tests. The implementation of our test module has four sub modules that are illustrated in Figure 3-11. The controller of the test module generates the error signal if any one of the implemented test fails. The runs_test module includes the runs and long run tests. Monobit and poker tests are realized in two separate modules. The following subsections presents simulations and verification of these test modules.

3.2.1 Simulation of Test Module on Modelsim

Modelsim is a hardware simulation and debug tool developed by Mentor Graphics Inc. In order to verify the functionality, we simulated our test module design on Modelsim simulator. Modelsim simulates the input signals of the device under test (DUT) and shows the responses of this the device via a proper visual interface. Modelsim compiles the device and test bench codes by using standard and vendor specific libraries and the tester can observe all internal and external signals of the design in a timing diagram while simulating it.

In the present case, DUT is the test module of our TRNG. The test module has 6 inputs i.e. start, stop, bitready, randombit, rst and clk. The clk input of our test module is a 50 MHz clock and the rst is an active low reset signal. The start and stop input signals carry initiate and halt commands respectively, which are formed as single pulses. The critical part of this test bench is the simulation of the randombit signal because the random input of the test module is not a periodic signal and even it is required to have good randomness features to simulate for correct random bit generation. Generating a random signal is not an easy task for Modelsim by VHDL code. Therefore we decided to use the original random bits that are generated by our custom TRNG as an input to the test bench of our design. The generated numbers are saved in a text file in binary format and are used as an input for the test bench. During simulation the test bench reads the random bits from the file and produces randombit and bitready signals. By using the visual interface, we have verified that the test module of our TRNG works as expected and any deviation from the randomness according to FIPS 140-1 is detected by this module. In the next sub sections of this chapter, we present some simulation diagrams for the test module.

3.2.1.1 Simulation of Monobit (Frequency) Test

The implementation of monobit test is the simplest one of the FIPS 140-1 tests. This test checks the zero and one ratio of every 20.000 bits of input sequence. The implementation contains just two counters which are named as bit_counter and one_counter as in Figure 3-12. The bit_counter counts the number of input bits up to 20.000 and restarts counting. For every 20.000 bits of random input a stop signal is generated to reset the counter value. The start input also clears the counters. The one_counter counts ones in the input sequence until a clear signal is generated.

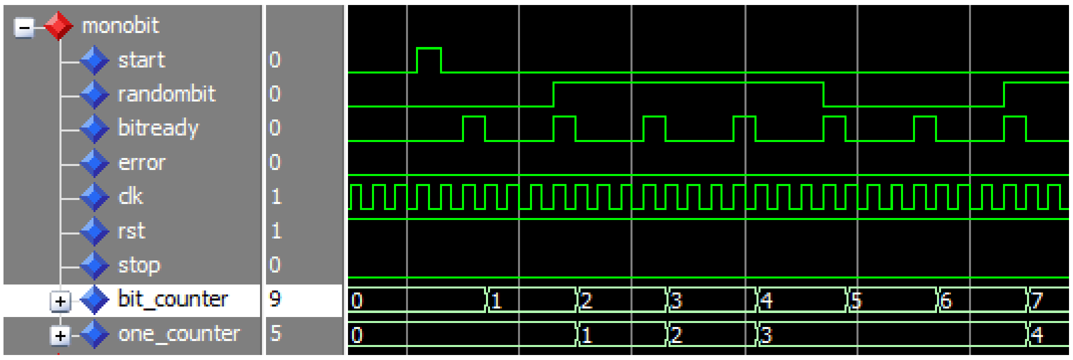


Figure 3-12 - Monobit Test Signals (Start of a sequence)

The error output signal gets high when the stop signal is high and the one counter value is out of the acceptance interval.

$$9,654 < \text{\# of ones in } 20.000 \text{ bits} < 10,346$$

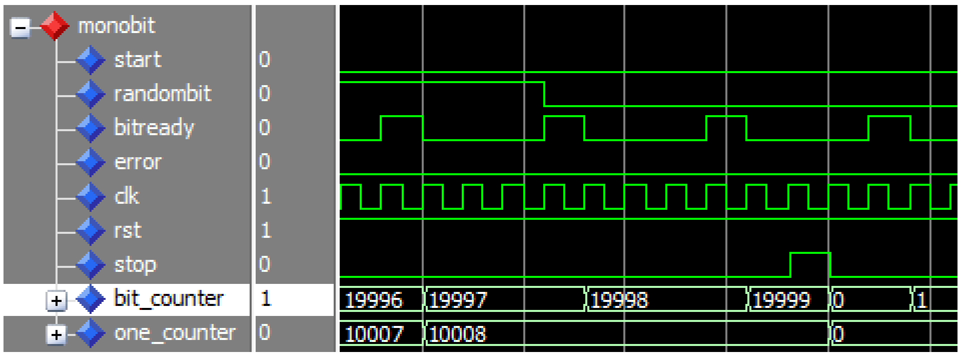


Figure 3-13 - Monobit Test Signal (End of a sequence)

3.2.1.2 Simulation of Runs Test

The runs and long run tests check if the tested sequence contains more than expected number of consecutive ones or zeros. “Run” is a terminology, which corresponds to a sequence of identical bits that are bounded before and after with a bit of the opposite value. The length and number of occurrence of the runs are critical for randomness. The runs with length up to 6 and larger than 6 have their own required intervals which is in Figure 3-14. On the other hand according to long runs test, a run with length 34 and larger is not acceptable for a random sequence. We have implemented runs and long run tests in the same module which counts the number of runs that are occurred in a 20.000 bits long sequence.

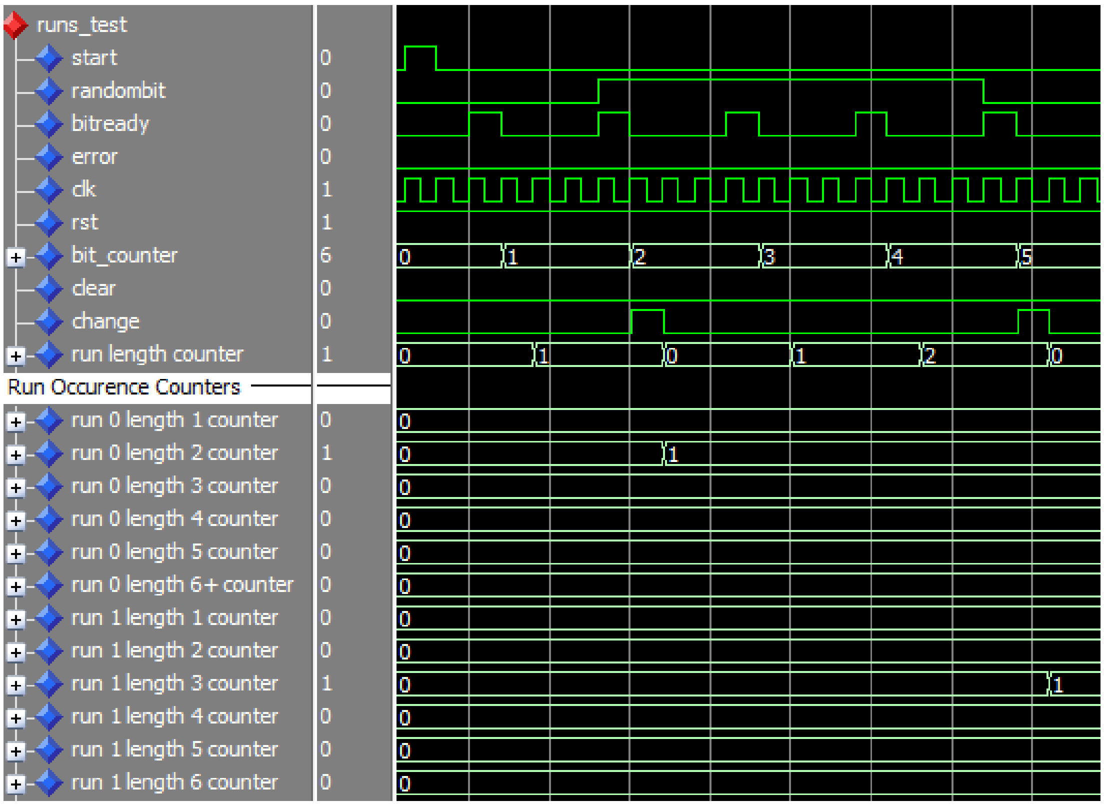
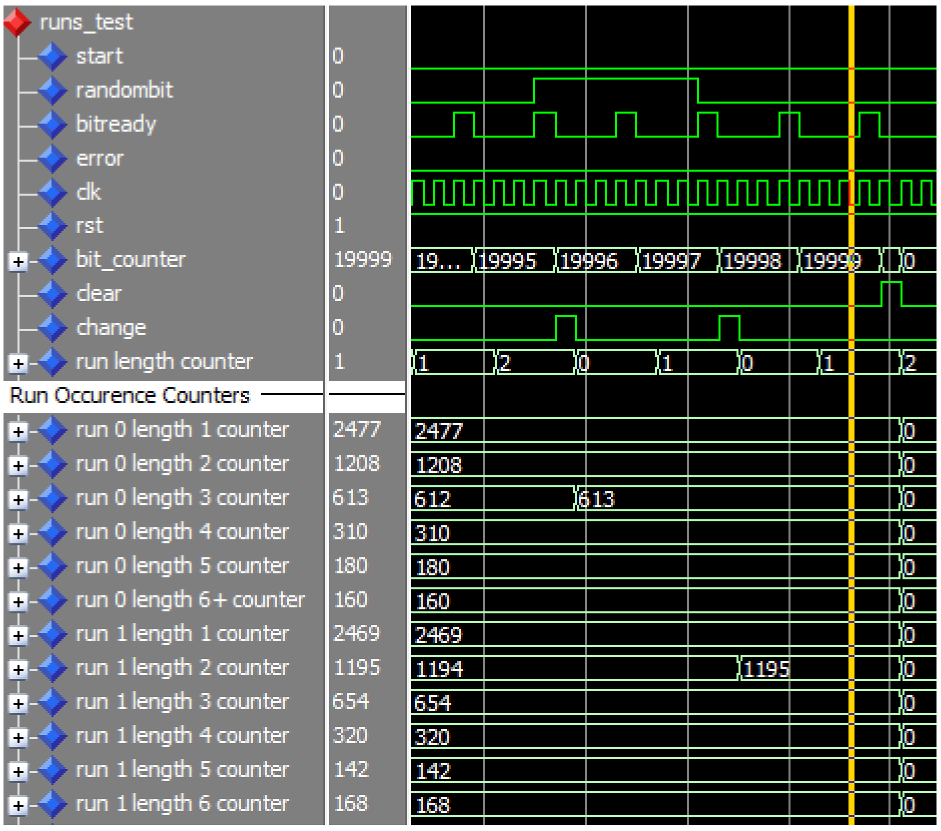


Figure 3-14 - Runs Test Signals

The generated signals and the counters used in the implementation of run test module are presented in Figure 3-14. The module generates change signal, which indicates the starting point of all runs in the incoming sequence. The run counter is cleared by each change signal and counts up until the next change pulse. For every change pulse one of the run occurrence counters, counts up.

The bit counters counts incoming bits and is used to generate clear signal for every 20.000 bits. Whenever the clear signal is generated, the numbers of occurrence counters are controlled if it is in the required interval of not as can be seen in Figure 3-15. If any occurrence number of run is not in the required interval the error signal gets high to indicate non randomness. Besides if the run counter value exceeds 34, the error signal also gets high.



runs_test
start 0
randombit 0
bitready 0
error 0
clk 0
rst 1
bit_counter 19999
19... 19995 19996 19997 19998 19999 0
clear 0
change 0
run length counter 1
1 2 0 1 0 1 2
Run Occurence Counters
run 0 length 1 counter 2477
2477 0
run 0 length 2 counter 1208
1208 0
run 0 length 3 counter 613
612 613 0
run 0 length 4 counter 310
310 0
run 0 length 5 counter 180
180 0
run 0 length 6+ counter 160
160 0
run 1 length 1 counter 2469
2469 0
run 1 length 2 counter 1195
1194 1195 0
run 1 length 3 counter 654
654 0
run 1 length 4 counter 320
320 0
run 1 length 5 counter 142
142 0
run 1 length 6 counter 168
168 0

Figure 3-15 - End of Runs Test

3.2.1.3 Simulation of Poker Test

The poker test considers a 20.000 bit random stream as 5000 distinct numbers that are represented in 4 bits. Each distinct number has 16 possible decimal values. The poker tests controls if the occurrence numbers of these 16 possible values are in a required interval as given below.

$$1.03 < \frac{16}{5000} * \left(\sum_{i=0}^{15} (f(i)^2) - 5000 \right) < 57.4$$

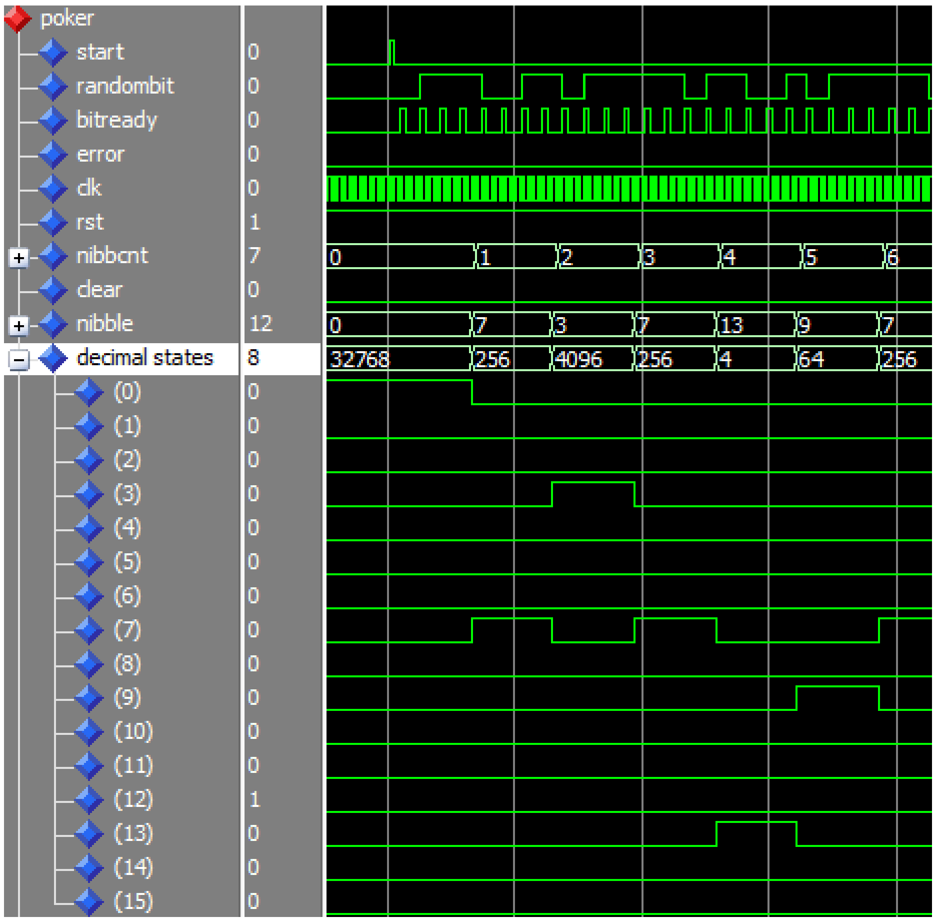


Figure 3-16 - Poker Test Signals

The signals of the poker test implementation are illustrated in Figure 3-16.

In our implementation, the poker module packs the each 4 successive bits into a nibble and generates a nibbready signal. This generated nibble is input to a hex to decimal decoder. The decoder generates 16 state signals, which are then used for counting the occurrence numbers of 16 possible decimal numbers. A sub module of poker test is named count_then_square counts the number of occurrences and calculates the square of this count values.

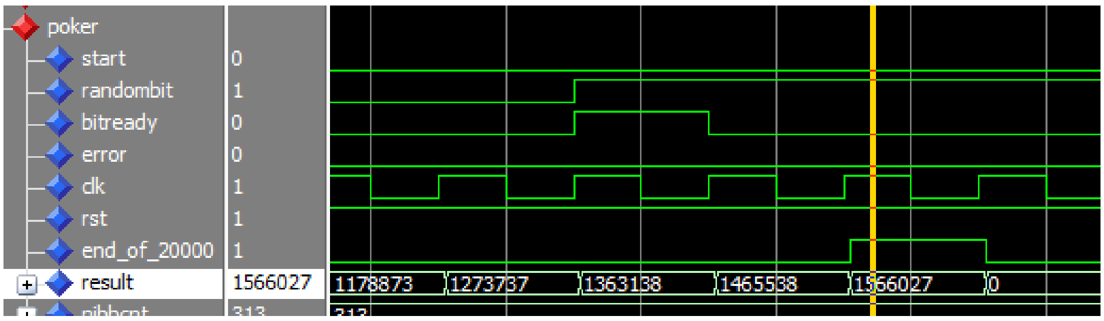


Figure 3-17 - Poker Result signals

The results of the square operation are added together and controlled if the sum is in the following interval or not.

$$1562906 < \sum_{i=0}^{15} (f(i)^2) < 1580438$$

.

CHAPTER 4

TESTS AND RESULTS

The quality and security evaluation of a TRNG should be carried out before it is employed in a cryptographic system. The strength and the security of a system directly depend on the randomness quality of the TRNG. Generated random bit streams have to be tested generally with various tests in order to detect any deviation from randomness. The deviation may arise from either a poorly designed generator or an external attack. Besides a certain number of failures can also be expected in random sequences that are generated by a custom TRNG. A tester should interpret the test results and make a conclusion about the correctness/incorrectness or validity/invalidity issues.

The statistical evaluation of the bit streams that are generated by a TRNG can be done internally or externally. In spite of the fact that internal and concurrent evaluation is more secure and reliable, the generated bit streams are generally tested out of the TRNG because the statistical tests require complex and heavy calculations and the implementation of these on hardware are generally not feasible and efficient. In our TRNG implementation we have implemented the tests of FIPS 140-1 in hardware because this in a way reduced test suite is feasible and efficient to implement in hardware to enhance the security of the TRNG. On the other hand, we still have to evaluate the randomness of our TRNG externally by using some other test suites. For this, we designed a test platform, which generates required amount of random bits using our TRNG and saves them in a file on the computer. We then use this file for statistical evaluation of our custom TRNG.

Besides determining whether our custom TRNG is qualified to be employed in a cryptographic system or not, we also compared it with a commercial TRNG that is already used in many cryptographic systems. RPG100 TRNG from the FDK corporation [10] is our choice in this comparative evaluation. RPG100 is an IC that generates a true random bit stream with a 250 Kbps throughput [30] [31]. The circuit is composed of only CMOS and no other external components are needed. Statistical random number generator test circuits of FIPS 140-1 are also built into the chip for checking randomness easily. The FDK corporation provides an evaluation board for RPG100 [32]. By using this evaluation board, we can easily reach to the pins of RPG100 chip and can operate it with external signals that are supplied by another board.

In our evaluation, RPG100 is required to be operated with in the same conditions hence we used RPG100 in the same test platform. We connected the evaluation board of RPG100 to the ML507 board via external pins of Virtex-5 FPGA. Then by the help of our test platform designed FPGA, we have accumulated the generated random bits of RPG100 and save then also in a file on the computer as was done for custom TRNG.

In this chapter firstly, we first explain the components and connections of the test set up and then the FPGA project that includes our custom TRNG and the driver for RPG100. The statistical evaluations of the generated bit streams, which are performed off-line, are also reported in this chapter. At the end of the chapter our custom TRNG and RPG100 are compared according to several aspects.

4.1 The Test Set Up

The test set up contains three main elements that are computer, Xilinx ML507 Development Board and RPG100 Evaluation Board [33]. The computer is connected to FPGA (Virtex – 5) of the ML507 Board via Xilinx USB Platform Cable which can reach to the FPGA from the JTAG port. This port is used for loading and the debugging of the code running on the FPGA. The RS232 port of the board is used for communication of computer and the test platform project that runs on the FPGA. The RPG100 Evaluation board is also connected to ML507 board via the general purpose external pins of the FPGA. The general structure of the test set up is presented in Figure 4-1.

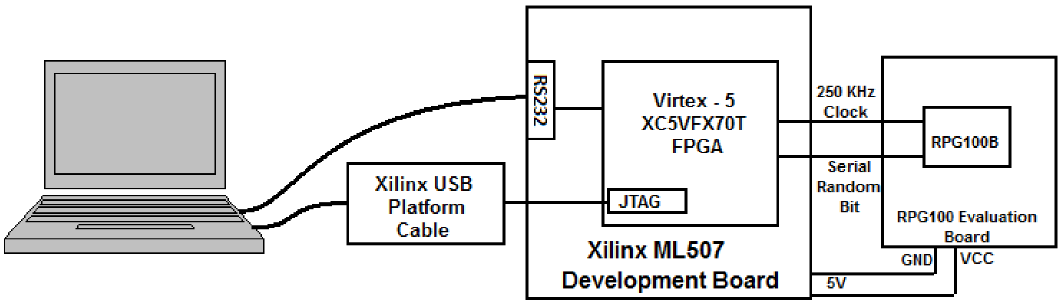


Figure 4-1 - Test Set Up

The test set up provides the facilities listed below.

- Generating required length of random bit streams by using custom TRNG or RPG100 chip.
- Accumulating the generated bit streams in a file on the computer.
- Running statistical evaluation processes on the generated bit streams.
- Measuring the throughput of the generators exactly.
- Monitoring the results of the embedded tests of custom TRNG.

4.2 Test Platform-FPGA Part

The test platform project, which runs on the test set up, provides a user interface for generating and saving the random bit streams of our custom TRNG or RPG100 on a file. By using this set up we can also calculate the throughput of our custom TRNG. Main sub modules of the test platform are listed below:

- An embedded microprocessor (Power PC “PPC440”),
- A shared two port block RAM (BRAM),
- TRNG driver module,
- Our custom TRNG (explained in the previous chapter),
- RPG100 controller module,

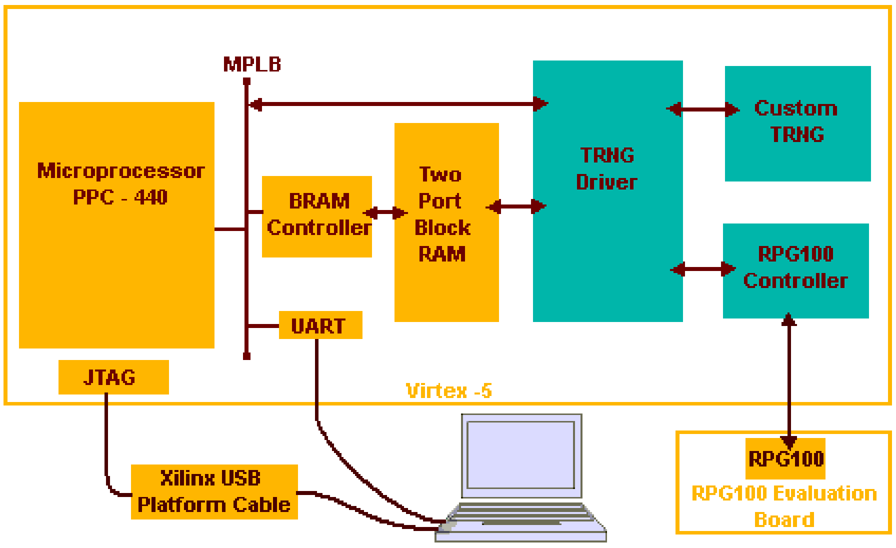
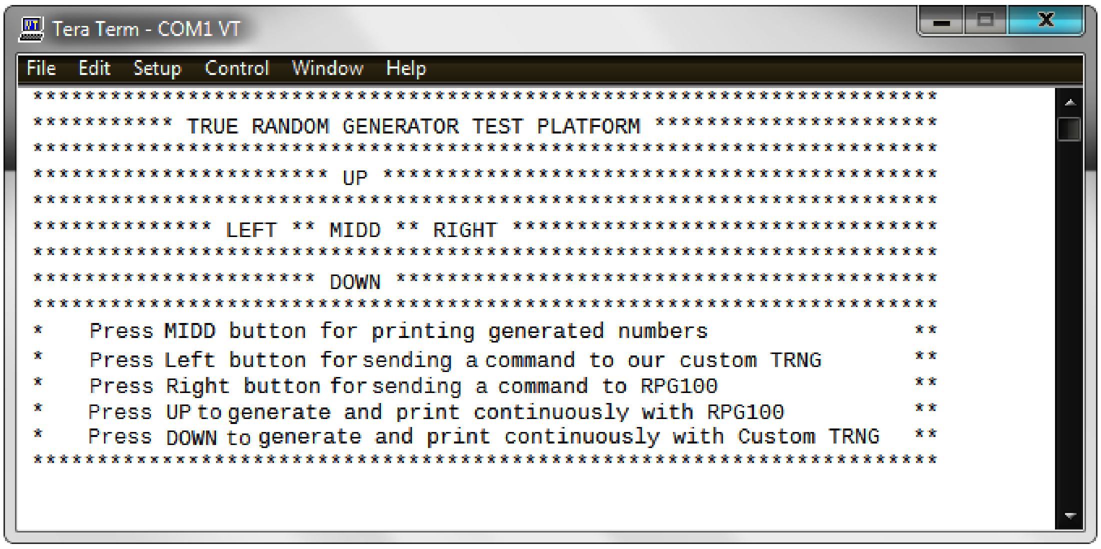


Figure 4-2 - Test Platform FPGA Project Top Design

The general structure of our test platform is presented in Figure 4-2. The modules in orange color are cores that already exist in all Virtex-5 FPGAs but the modules in green color are our custom designs for the test platform.

The embedded PPC440 microprocessor communicates with the peripheral devices via the peripheral local bus (PLB). Using software one can read/write from/to BRAM by using BRAM controller. And one can also send commands and read responses from the TRNG driver by using custom peripheral controller register which is already included in the TRNG driver. The TRNG driver shares the generated random bits via a shared BRAM. Both of the microprocessor and the TRNG driver can write/ read to/from the same BRAM.

The software that is running on the PPC440 provides a user interface by communicating with the computer via RS232 port and sensing the buttons of the board. This user interface informs the user using the terminal connected to the serial port while polling the buttons on the board. Each button of the board has its own functionality that is explained in the interface menu shown in Figure 4-3.



```
**********************************************************************
*********** TRUE RANDOM GENERATOR TEST PLATFORM **********************
**********************************************************************
*********************** UP *******************************************
**********************************************************************
************** LEFT ** MIDD ** RIGHT *********************************
**********************************************************************
********************** DOWN ******************************************
**********************************************************************
*    Press MIDD button for printing generated numbers              **
*    Press Left button for sending a command to our custom TRNG    **
*    Press Right button for sending a command to RPG100            **
*    Press UP to generate and print continuously with RPG100       **
*    Press DOWN to generate and print continuously with Custom TRNG  **
**********************************************************************
```

Figure 4-3 - Test Platform Menu

Whenever the generate button of the board is pressed, software side of the system sends generate command to TRNG driver module. After sending start command, the software waits for the response of the TRNG driver. The TRNG driver operates the chosen TRNG in order to generate a 1 mega random bit. Whenever the generation process is completed the TRNG driver informs the software that the requested random stream is generated and written on the shared block RAM. Also by using different buttons of the board, these generated random numbers can be printed on the serial port terminal. By using the log to a file function of the terminal, the user can save this random bit stream in a file on the computer. This file can then be used for the statistical evaluation of the TRNGs. If the user wants more than one mega random bits there is also a button on the board that is configured to generate and print the random numbers continuously.

4.2.1 The TRNG Driver Module

The TRNG driver module is designed to operate one of the true random number generation modules, i.e. our custom TRNG or one of RPG100. This module communicates with the microprocessor by using the custom peripheral controller registers. A custom peripheral controller register is a software accessible register which both the microprocessor and the custom design can write/read to/from it. The TRNG driver module gets the commands from the microprocessor and it operates the relevant TRNG. The custom TRNG or the RPG controller then send the random bit streams to the TRNG driver via 32 bit long output ports. TRNG driver writes the required amount of incoming random numbers into the two port block RAM then send acknowledgement to the microprocessor which means bit stream is ready to be read from the block RAM. The TRNG driver also stops the generation operation whenever the required amount of random bit streams are generated and written into the shared block RAM.

Besides, TRNG driver employs a counter for measuring the generation rate of the related TRNG. This counter is cleared before the generation process is started and then counts up during the generation of 1 megabits of random data. When generation is completed, this counter value can also be read by our software. The counter counts up with a 50 MHz clock. So the total time for the generation of 1 mega random bit is equal to counter value multiplied by 20 ns. Using this total time information the generation bit rate is calculated.

4.2.2 The RPG100 Controller Module

For the statistical evaluation of RPG100, we designed a module named RPG100 controller, which is connected to the RPG100 Evaluation Board via the general purpose I/O pins of the FPGA. RPG100 controller module supplies a 250 KHz clock to the RPG100 evaluation board, reads the generated random bits, saves them on a file as is done for the custom TRNG. The RPG100 controller module shown in Figure 4-4 has a top design view similar to the custom TRNG that was explained in the previous chapter.

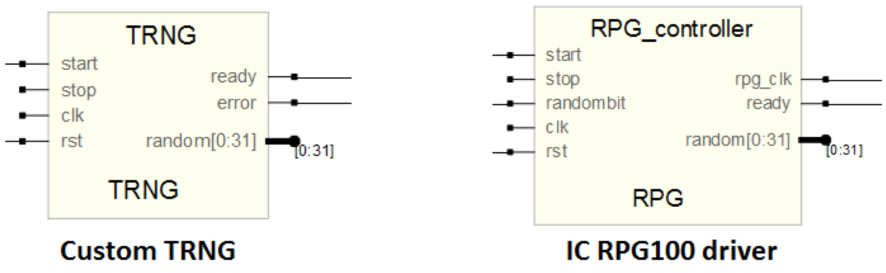


Figure 4-4 - Top Modules of TRNG and RPG100 controller

Both of these modules begin generating 32 bit long random numbers with the start pulse and continue generating up to the stop pulse. Both modules form a ready pulse for every new random number generated. The RPG controller module has two different pins which are “randombit” and “rpg_clk”, which are used for supplying the clock and for reading the random bit from the RPG100 evaluation board. These two pins are both connected to the external pins of the FPGA and wired to the RPG100 evaluation board.

4.3 Statistical Test Results

We have utilized the NIST test suite and Diehard battery of tests in our project for statistical evaluation of our custom TRNG and the RPG100 IC. We captured one gigabit output of both TRNGs into files on the computer by using the test platform design explained above. We then executed the tests on these random streams by following the recommendations of each relevant test suite. We assessed the outputs of the test suites by the method following NIST SP800-22 explanations. In this part of the chapter, the execution of the tests is explained in detail and our assessments of the results are reported.

We utilized the same statistical testing methodology used by the FDK Corporation, which produces the RPG100 IC. They have also tested their RPG100 silicon chip by NIST and Diehard test suites. The employed testing procedure of FDK Corporation is reported in [32].

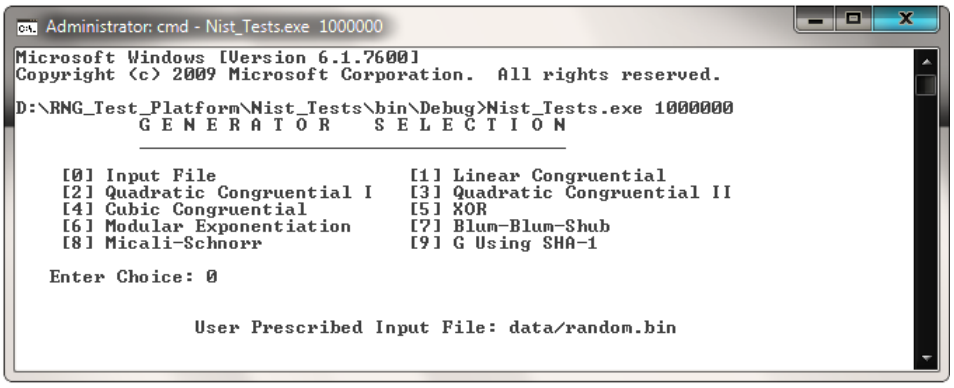
4.3.1 The NIST Test Suite Execution and Results

The NIST Test Suite is a statistical test suite that includes 15 different tests that were developed for randomness evaluation of binary sequences that are produced by cryptographic TRNGs. These tests check if there exist any different types of non-randomness in the sequence under test. The test suite produces a P-value for each test. The P-value (probability value) is a probabilistic measure that indicates the probability that the tested sequence is random. If a P-value of a test is equal to 1, then the sequence appears to have perfect randomness according to this test. On the other hand a P-value of zero indicates that the sequence appears to be completely non-random according to the test. Even if the tested sequence is truly random, the test results may conclude that the sequence is non-random with a small percentage. The probability of this kind of conclusion is called level of significance of the tests and denoted by “α”. The level of significance of a random sequence should be about 0.01 – 0.001 for cryptography. In order to decide whether the sequence passes a test or not, both the P-value and the level of significance are used. If the P-value is greater than (or equal to) the level of significance then the sequence under test is passed. On the other hand the distribution of P-values between (0, 1] is also important for the interpretation of the results. These P-values should be uniformly distributed in the interval (0, 1]. The NIST test suite generates a final analysis report, which includes the distribution of the P-values and the pass/fail proportions of each test in the suite.

4.3.1.1 The NIST test suite execution process

NIST has published ANSI C codes of the test suite in SP800-22. We used these C codes in our test process. We first compiled and built these codes to generate an executable for the test suite. Sample run of this executable for testing of 1.000.000 bits long bit stream is presented step by step below.

In order to invoke the NIST statistical test suite the user types the name of executable followed by the desired bit stream length (n) on the command prompt of windows. The following screen in Figure 4-5 is displayed first.

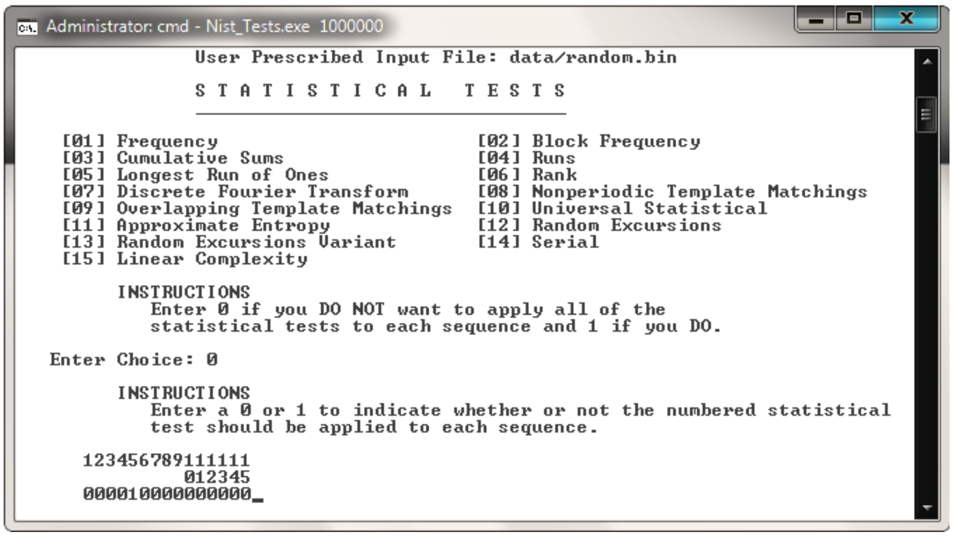


```
Administrator: cmd - Nist_Tests.exe 1000000

Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

D:\RNG_Test_Platform\Nist_Tests\bin\Debug>Nist_Tests.exe 1000000
               G E N E R A T O R    S E L E C T I O N
               ______________________________________

      [0] Input File                     [1] Linear Congruential
      [2] Quadratic Congruential I       [3] Quadratic Congruential II
      [4] Cubic Congruential             [5] XOR
      [6] Modular Exponentiation         [7] Blum-Blum-Shub
      [8] Micali-Schnorr                 [9] G Using SHA-1

    Enter Choice: 0

                      User Prescribed Input File: data/random.bin
```

Figure 4-5 - NIST Test Execution Screen - 1

The test suite can run the tests on the outputs of the pseudo random number generators that are listed in menu. However, option “0” is prepared for testing a custom bit stream. The user chooses the option “0” then types the folder link of the file that contains the bit stream.

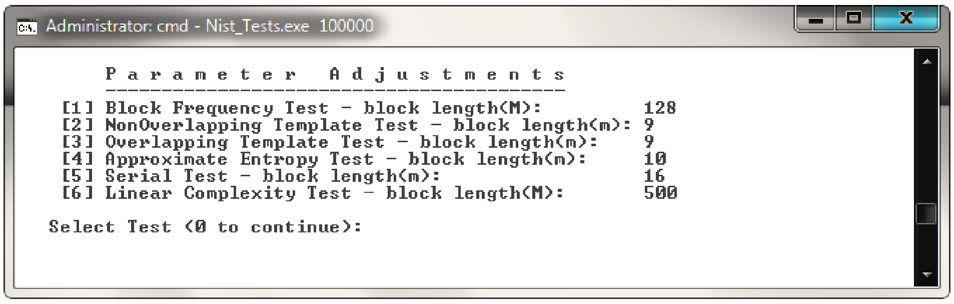
After specifying the input string file the statistical test list comes to screen as shown in Figure 4-6. In this console, the user can choose the tests that are going to be executed. If the user’s choice is “1” then all tests are going to be executed. But if the user’s choice is “0” then the user can determine which ones of the tests are going to be executed by typing a string. The string consists of 15 consecutive zeros or ones. The sequence numbers of ones in the string indicate the sequence numbers of the tests that are going to be executed.



```
Administrator: cmd - Nist_Tests.exe  1000000
                    User Prescribed Input File: data/random.bin

                    S T A T I S T I C A L   T E S T S
                    _________________________________

    [01] Frequency                       [02] Block Frequency
    [03] Cumulative Sums                 [04] Runs
    [05] Longest Run of Ones             [06] Rank
    [07] Discrete Fourier Transform      [08] Nonperiodic Template Matchings
    [09] Overlapping Template Matchings  [10] Universal Statistical
    [11] Approximate Entropy             [12] Random Excursions
    [13] Random Excursions Variant       [14] Serial
    [15] Linear Complexity

         INSTRUCTIONS
            Enter 0 if you DO NOT want to apply all of the
            statistical tests to each sequence and 1 if you DO.

   Enter Choice: 0

         INSTRUCTIONS
            Enter a 0 or 1 to indicate whether or not the numbered statistical
            test should be applied to each sequence.

      123456789111111
               012345
      000010000000000_
```

Figure 4-6 - NIST Test Execution Screen - 2

As an example in Figure 4-6, only the longest run of one’s test is going to be executed since only 5th bit of the string is set, indicating the fifth test. After choosing the tests the tests parameter adjustment screen appears as shown in Figure 4-7.

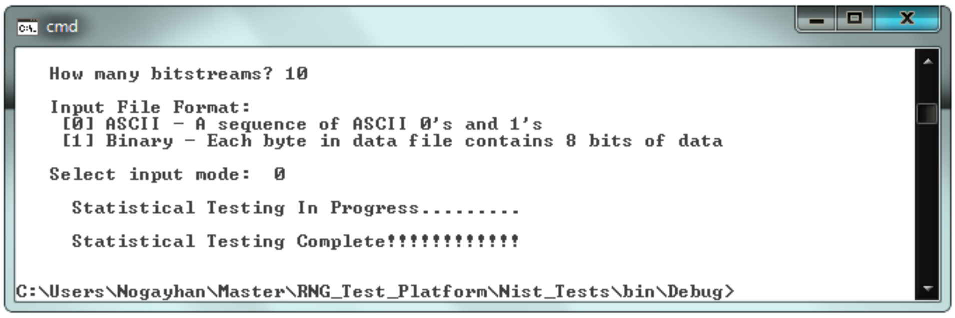


```
Administrator: cmd - Nist_Tests.exe  100000
        P a r a m e t e r   A d j u s t m e n t s
        -----------------------------------------
   [1] Block Frequency Test - block length(M):         128
   [2] NonOverlapping Template Test - block length(m): 9
   [3] Overlapping Template Test - block length(m):    9
   [4] Approximate Entropy Test - block length(m):     10
   [5] Serial Test - block length(m):                  16
   [6] Linear Complexity Test - block length(M):       500

   Select Test (0 to continue):
```

Figure 4-7 - NIST Test Execution Screen - 3

The parameter adjustment of the tests is also critical for the reliability of the tests. Generally, these parameter adjustments are done according to the length and number of samples that are going to be tested. The recommendations on the choice of parameters for each test are also explained in [6]. This console screen enables one to change these parameters during the test. After configuration of the test parameters the user types the number of samples (m) in the input file as shown in

Figure 4-8. So the input file has to include m times n bits. For the example given in Figure 4-8, the input file has 10 (m) times 100.000 (n) bits long streams.



```
  How many bitstreams? 10

  Input File Format:
   [0] ASCII - A sequence of ASCII 0's and 1's
   [1] Binary - Each byte in data file contains 8 bits of data

  Select input mode:  0

     Statistical Testing In Progress.........

     Statistical Testing Complete!!!!!!!!!!!!

C:\Users\Nogayhan\Master\RNG_Test_Platform\Nist_Tests\bin\Debug>
```

Figure 4-8 - NIST Test Execution Screen - 4

NIST test suite executable file can read random streams from an input file that are prepared in one of two different formats, which are ASCII and Binary formats. An ASCII formatted file includes a sequence of ASCII characters of 0's and 1's. Each bit of the random string is represented by an ASCII character. The binary formatted file includes the binary data that each byte in file contains 8 bits of random data. The user prepared file format is chosen in the screen shown in Figure 4-8. After this last step, the tests are executed and the results of all tests are written on specific result files. Besides, an interpretation of all tests is prepared and written on a file as a final analysis report of the test suite. This report includes results for the uniformity of P-values and the pass proportions of the tested sequences.

We performed the NIST statistical tests for evaluating our custom TRNG and RPG100 by following the strategy and recommendations in [6]. We used one gigabit generated bit streams. The generated random sequences are input to the test as a set of m (=1000) one Mbit sequences. The test suite provided the set of *P*-values for each generator (some typical values are shown in Table 4-1 and Table 4-2) for a significance level (α) of 0.01. The proportion of the passing sequences is within the expected confidence interval for both of the performed tests and *P*-values are uniformly distributed over (0, 1) interval. Some of the results that are taken from the final analysis report of executed NIST test of the two TRNGs are shown in the following tables. The final test report generated by NIST test suite executable file is available in the Appendix – C.

Table 4-1 - NIST Final Test Report for Custom TRNG (α = 0.01)

Results for the Uniformity of P-values and the Proportion of Passing sequences													
Test Name	**Proportion**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	χ^2	P_T
Frequency	989/1000	95	107	108	108	100	100	84	100	94	104	5.1	0.825505
Block Frequency	989/1000	114	86	95	93	110	108	112	88	95	99	10.88	0.397688
Cum Sums	988/1000	94	108	99	116	75	108	115	113	91	81	18.82	0.026768
Runs	995/1000	87	98	95	106	100	88	112	116	105	93	8.52	0.482707
Longest Run	989/1000	104	114	107	112	80	95	114	92	84	98	13.5	0.141256
Rank	989/1000	103	89	101	119	91	98	108	94	102	95	7.06	0.630872
FFT	983/1000	110	110	98	104	89	108	95	93	91	102	5.64	0.775337
Non Overlapping Temp.	989/1000	94	103	96	99	119	91	101	93	97	97	5.72	0.603841
Overlapping Temp.	995/1000	136	100	83	95	103	83	90	115	104	91	23.3	0.005557
Universal	990/1000	103	100	119	99	96	99	103	97	98	86	6.06	0.733899
Approximate Entropy	987/1000	109	102	92	81	121	101	94	100	103	97	10.06	0.345650
Random Excursion	617/628	58	67	62	75	59	65	61	52	69	60	5.986	0.723673
Rand. Exc. Variant	623/628	58	62	71	75	39	60	67	68	56	72	15.75	0.065546
Serial	993/1000	117	112	90	83	107	107	81	96	100	107	12.18	0.142872
Linear Complexity	993/1000	101	96	104	93	98	125	91	110	82	100	12.16	0.204439

Table 4-2 - NIST Final Test Report for RPG100

Results for the Uniformity of P-values and the Proportion of Passing sequences													
Test Name	**Proportion**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	χ^2	P_T
Frequency	994/1000	96	110	101	104	110	111	78	93	114	83	13.72	0.132640
Block Frequency	991/1000	96	92	91	99	114	101	92	117	95	103	7.46	0.589341
Cum Sums	994/1000	92	107	108	106	109	92	91	93	110	92	6.52	0.686955
Runs	987/1000	107	110	106	86	103	104	104	96	91	93	5.28	0.771469
Longest Run	987/1000	90	96	116	105	109	103	99	86	107	89	8.54	0.480771
Rank	990/1000	104	109	95	102	89	102	94	89	114	102	6.08	0.731886
FFT	987/1000	111	14	99	107	99	88	106	85	87	104	9.57	0.385543
Non Overlapping Temp.	992/1000	100	89	95	90	93	99	109	118	104	103	7.26	0.610070
Overlapping Temp.	987/1000	106	113	105	99	98	112	96	98	98	75	10.28	0.328297
Universal	991/1000	99	113	102	94	110	76	87	90	94	135	24.16	0.004055
Approximate Entropy	992/1000	107	91	100	97	93	100	82	118	117	95	8.26	0.242986
Random Excursion	608/618	75	71	56	61	45	57	67	66	46	74	16.85	0.045966
Rand. Exc. Variant	610/618	65	67	71	55	68	59	55	65	53	60	5.689	0.752969
Serial	994/1000	91	103	99	78	114	110	99	94	98	114	11.08	0.270265
Linear Complexity	987/1000	94	119	105	102	87	113	90	103	95	92	9.02	0.382115

The final analysis report of NIST test suite contains a summary distribution of the P-values and passing rate of each test. The results are represented as a table with x rows and y columns. The rows correspond to statistical tests applied, while columns are distributed as follows: columns 1-10 corresponding to the frequency of P-values, column 11 corresponding to the P-value that arises via the application of a chi-square test, column 12 corresponding to the proportion of binary sequences that have passed, and column 13 being name of the corresponding statistical test.

4.3.1.2 Interpretation of NIST test results

The interpretation of empirical results can be done in several ways. But the NIST test suite has adopted two approaches for this. The first one is the examining the proportion of sequences that pass a statistical test and the second one is checking the uniformity of distribution of P-values.

4.3.1.2.1 The pass/fail assessment of the tests

The test suite determines if an individual test is passed or not according to level of significance. If the P-value of an individual test is greater or equal to the level of significance then the test is accepted to be passed. The level of significance is 0.01 for all our tests. For example if 1000 binary sequences were tested and 996 binary sequences had P-values ≥ .01, then the passing ratio is 996/1000 = 0.9960.

The final analysis report of the NIST test suite contains the passing ratios of each test. In order to determine that the input stream is truly random these passing ratios should be in an acceptable range. NIST recommends that the acceptable range of passing rates is determined using the confidence interval defined by the equation below.

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}} \quad \text{where } \hat{p} = 1 - \alpha \text{ and m is the sample size}$$

If the proportion falls outside this interval, then there is evidence that the data is non-random. In our case:

Test sample (m)	1000
Length of each bit stream (n)	1 Mbit
Level of significance α	0.01
The acceptance region	$0.980561 \leq \hat{p} \leq 0.999499$

For both of the tests that are executed using the outputs of custom TRNG and RPG100 we obtained passing ratios as in Table 4-1 and Table 4-2, which verifies that all of these passing rates are in the acceptance region.

4.3.1.2.2 Uniformity of distribution of P-values

The final analysis report of the NIST test suite also includes the distribution of P-values to ensure uniformity. The interval between 0 and 1 is divided into 10 sub-intervals, and the P-values that lie within each sub-interval are counted and displayed in the report for each test. The report contains a master P-value for each test which is calculated via an application of a chi-square test and Goodness-of-Fit Distributional Test on the P-values obtained for each individual statistical test. This master P-value of each test should be greater than or equal to 0.0001 for uniformly distributed P-values. The master P-value (P_T) of a test is calculated by the procedure defined below.

Firstly the chi-square test is run as follows:

$\chi^2 = \sum_{i=1}^{10} \frac{(F_i - s/10)^2}{s/10}$ where F_i is the number of P-values in sub-interval i and s is the sample size.

Then master P-value is calculated as

$P_T = \mathrm{igamc}(9/2, \chi^2/2)$ where the igamc is an incomplete Gama function.

In order to obtain a P_T, which is greater than (or equal to) 0.0001 NIST recommends that if the level of significance of test is 0.01 then the acceptance region of the chi-square test output should meet the equation $\chi^2 \leq 33.72$. In our case all calculated χ^2 values are observed in the acceptance region as seen in Table 4-1 and Table 4-2.

The results of both TRNGs are similar to each other according to passing rates and distribution of P-values. Results of both TRNGs are in the safe interval of randomness tests of the NIST.

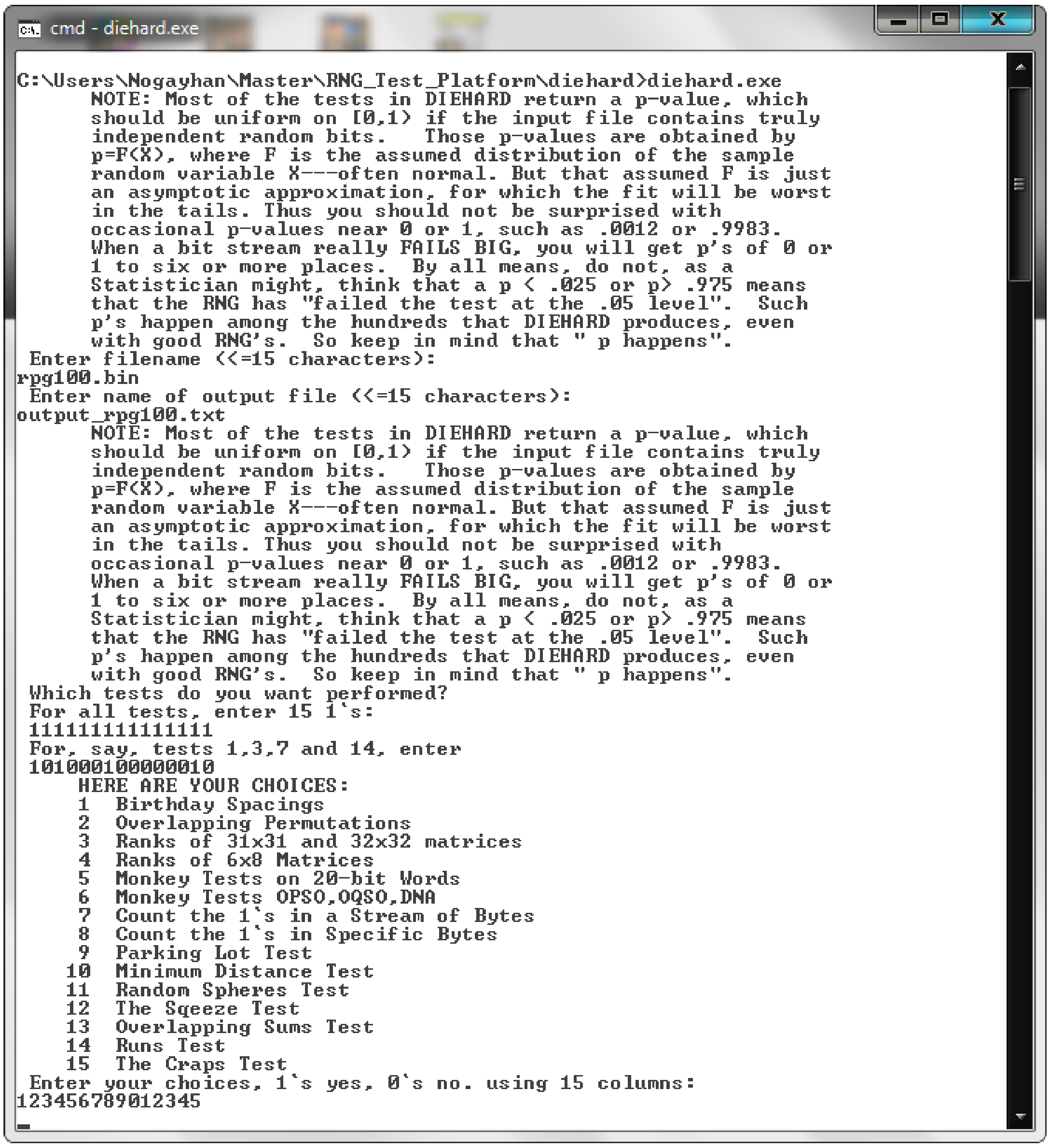
4.3.2 The Diehard Battery of Tests and Results

The diehard battery of tests includes 15 distinct statistical tests. For each execution, the test suite generates 220 P-values for the evaluation of 11,468,800 bytes of random data. The number of generated P-values is different each test of the battery. A statistical test of diehard is considered a pass if the P-value is in range $[(0+\alpha/2),(1-\alpha/2)]$ where α is the level of significance of the test. The level of significance (α) is recommended to be 0.05 for the diehard test suite. Therefore, these P-values are supposed to be uniformly distributed in [0, 1) if the tested stream is truly random.

4.3.2.1 Diehard Battery of the Tests execution process

The source code and the executable file of the diehard test suite (diehard.exe) are also available on the internet [7]. We used this executable for the evaluation of our bit streams. A sample run of the diehard test suite is explained below. Diehard test suite requires data in binary format, where conversion can be done by using “asc2bin.exe”, which is also available on the internet together with the diehard test executable file.

The executable file of diehard test is invoked without any parameter. Then it waits for the name of a file of size 87.5 Mbits. The user then identifies an output file name and selects the tests to be executed as shown in Figure 4-9. For the evaluation of our custom TRNG and the RPG100 we used 1 Gbit output streams from each TRNG. In order to evaluate the randomness of these bit streams with diehard test suite, we split each 1 gigabit stream of data into 12 distinct 87.5 Mbits set of streams. Then we ran diehard test suite 12 times with each individual set. The result file of these tests includes a total of 2640 P-values. One of the generated test result files of diehard test suite is given in the Appendix - D.



```
C:\Users\Nogayhan\Master\RNG_Test_Platform\diehard>diehard.exe
      NOTE: Most of the tests in DIEHARD return a p-value, which
      should be uniform on [0,1) if the input file contains truly
      independent random bits.   Those p-values are obtained by
      p=F(X), where F is the assumed distribution of the sample
      random variable X---often normal. But that assumed F is just
      an asymptotic approximation, for which the fit will be worst
      in the tails. Thus you should not be surprised with
      occasional p-values near 0 or 1, such as .0012 or .9983.
      When a bit stream really FAILS BIG, you will get p's of 0 or
      1 to six or more places.  By all means, do not, as a
      Statistician might, think that a p < .025 or p> .975 means
      that the RNG has "failed the test at the .05 level".  Such
      p's happen among the hundreds that DIEHARD produces, even
      with good RNG's.  So keep in mind that " p happens".
 Enter filename (<=15 characters):
rpg100.bin
 Enter name of output file (<=15 characters):
output_rpg100.txt
      NOTE: Most of the tests in DIEHARD return a p-value, which
      should be uniform on [0,1) if the input file contains truly
      independent random bits.   Those p-values are obtained by
      p=F(X), where F is the assumed distribution of the sample
      random variable X---often normal. But that assumed F is just
      an asymptotic approximation, for which the fit will be worst
      in the tails. Thus you should not be surprised with
      occasional p-values near 0 or 1, such as .0012 or .9983.
      When a bit stream really FAILS BIG, you will get p's of 0 or
      1 to six or more places.  By all means, do not, as a
      Statistician might, think that a p < .025 or p> .975 means
      that the RNG has "failed the test at the .05 level".  Such
      p's happen among the hundreds that DIEHARD produces, even
      with good RNG's.  So keep in mind that " p happens".
 Which tests do you want performed?
 For all tests, enter 15 1`s:
 111111111111111
 For, say, tests 1,3,7 and 14, enter
 101000100000010
     HERE ARE YOUR CHOICES:
     1  Birthday Spacings
     2  Overlapping Permutations
     3  Ranks of 31x31 and 32x32 matrices
     4  Ranks of 6x8 Matrices
     5  Monkey Tests on 20-bit Words
     6  Monkey Tests OPSO,OQSO,DNA
     7  Count the 1`s in a Stream of Bytes
     8  Count the 1`s in Specific Bytes
     9  Parking Lot Test
    10  Minimum Distance Test
    11  Random Spheres Test
    12  The Sqeeze Test
    13  Overlapping Sums Test
    14  Runs Test
    15  The Craps Test
 Enter your choices, 1`s yes, 0`s no. using 15 columns:
123456789012345
_
```

Figure 4-9 - Diehard Execution Screen

4.3.2.2 Interpretation of the Diehard Tests Results

The output of diehard test suite contains P-values of each performed test for the tested stream. The numbers of generated P-values for each test is different and are given in Table 4-3 and Table 4-4.

In contrast with NIST test suite, diehard suite generates only the P-values. There exists no interpretation of these P-values and no pass/fail assessment of the tests. In order to interpret these P-values we employed the same method as in NIST case. These methods are (1) the examination of the proportion of sequences that pass a statistical test and (2) the distribution of P-values to check for uniformity.

4.3.2.2.1 The pass/fail assessment of Diehard tests

The pass and fail assessment of diehard tests is different than the NIST test suite. A statistical test of diehard is considered to be a pass if the P-value is in range [(0+α/2),(1-α/2)] where α is the level of significance [7]. Diehard test suite set the level of significance of each test as %5 for assessment then the passing interval of each tests is [0.025,0.975]. We performed the diehard test for 12 times for each TRNG. For each run we obtained 220 P-values. The results of our pass/fail assessment are summarized in Table 4-3 and Table 4-4. Table 4-3 and Table 4-4 contain the results of custom TRNG and RPG100 respectively. These tables include the number of P-values that are out of the pass region (0.025 – 0.975) for each individual Diehard test.

Table 4-3 - Diehard Pass/Fail Assessment for Custom TRNG

Test Name (Total # of generated P-values)	**Number of generated P-values that are out of pass region for each execution of test**											
	#1	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**	**#11**	**#12**
Birthday Spacing	1	1	0	1	0	0	0	0	1	0	0	1
OPERM5 (2)	1	0	0	1	0	0	1	1	0	0	0	0
Binary Rank 31x31	0	0	1	0	0	0	0	0	0	0	0	0
Binary Rank 32x32	0	0	0	0	0	0	0	0	0	0	0	0

Binary Rank 6x8	0	0	2	2	3	0	0	1	0	1	1	1
Bitstream (20)	1	1	0	1	0	1	0	1	1	3	3	2
OPSO (23)	0	1	0	1	0	1	0	1	1	3	3	1
OQSO (28)	0	2	3	3	1	2	1	0	2	4	1	4
DNA (31)	2	0	2	0	0	1	1	2	3	2	1	0
1's on stream of bytes (2)	0	0	1	0	0	0	0	1	0	0	0	1
1's on specific bytes (25)	0	1	1	0	2	1	0	2	0	1	2	2
Parking Lot (11)	0	1	0	1	2	0	1	3	0	0	0	1
Minimum Distance	0	0	0	0	0	0	0	0	0	0	0	0
3D Spheres (21)	1	1	1	0	2	0	0	2	1	0	1	1
Squeeze (1)	0	0	0	0	0	0	0	0	0	0	0	0
Overlapping Sums	2	0	0	1	0	1	1	0	0	1	1	0
Runs (4)	0	0	0	0	0	1	0	1	0	0	0	0
Craps (2)	0	0	0	0	0	0	1	0	0	0	0	0

Table 4-4 - Diehard Pass/Fail Assessment for RPG100

Test Name (Total # of generated P-values)	**Number of generated P-values that are out of pass region for each execution of test**											
	#1	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**	**#11**	**#12**
Birthday Spacing	0	0	0	0	1	0	1	0	0	3	1	0
OPERM5 (2)	0	0	1	0	2	0	0	0	0	0	2	0
Binary Rank 31x31	0	0	0	0	0	0	0	0	0	0	0	0

Binary Rank 32x32	0	0	0	0	0	0	0	0	0	0	0	0
Binary Rank 6x8	2	0	0	0	0	2	3	2	0	3	2	4
Bitstream (20)	3	1	0	1	1	1	2	1	0	1	1	0
OPSO (23)	0	0	0	0	1	0	2	0	1	1	1	1
OQSO (28)	2	2	0	1	2	1	1	1	1	2	1	1
DNA (31)	2	1	1	1	2	5	4	2	1	0	0	2
1's on stream of bytes (2)	0	0	0	0	1	0	0	0	0	0	0	0
1's on specific bytes (25)	1	1	3	2	1	1	2	1	0	1	1	0
Parking Lot (11)	1	0	1	0	0	0	0	0	1	1	0	0
Minimum Distance	0	0	0	0	1	0	0	0	0	0	0	0
3D Spheres (21)	2	3	0	0	1	1	0	3	2	0	1	2
Squeeze (1)	0	0	0	0	0	0	0	0	0	1	0	0
Overlapping Sums	0	0	1	1	1	0	0	0	0	0	0	0
Runs (4)	0	0	0	0	1	0	0	1	0	0	0	1
Craps (2)	0	0	0	0	0	0	0	0	0	0	0	0

For the evaluation of our 1 Gigabit data we performed diehard test suite 12 times. These 12 distinct run of diehard test suite produces a total of 2640 P-values. In order to make a pass/fail assessment with diehard test results, we specified the number of P-values that are out of the pass region (0.025-0.975). Then we determined the proportion of fails and passes in these 2640 P-values. This ratio must be in a specified interval, which is named as the acceptance interval.

The acceptance interval of our test is calculated using the same method as the NIST case. We have used the number of P-value that are generated by diehard test suite as the number of test samples and the level of significance of the test is 0.05.

Test sample $\rightarrow$ m = 2640

Level of significance $\rightarrow$ $\alpha = 0.05$ then

$$\hat{p} = 0.95 \pm 3\sqrt{0.95 \ x \ 0.05/2640} = 0.95 \pm 0.127252$$

The acceptance region $\rightarrow$ $0.937275 \leq \hat{p} \leq 0.962725$

The results of the tests show that 128 out of 2640 P-values are out of the pass region for the custom TRNG. On the other hand 126 out of 2640 P-values are out of the pass region for RPG100. Both these results are in the acceptance region and listed in the Table 4-5.

Table 4-5 - Pass/Fail Assessment of both TRNGs

	$\hat{p}$	**RESULT**
Custom TRNG output	0.95151	SUCCESS
RPG100 output	0.95227	SUCCESS

4.3.2.2.2 Uniformity of distribution of P-values of Diehard tests

We have checked the uniformity of the P-values of diehard test suite using the same method as in NIST case. We have parsed the output files of 12 diehard executions for each TRNG then calculated the χ^2 values. The distribution of P-values and calculated χ^2 values for the custom TRNG test results is listed in Table 4-6.

Table 4-6 - Uniformity list of the custom TRNG Diehard Test Results

Execution Number	**Distribution of generated 220 P-values over (0,1] for each execution of Diehard**										
	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	χ^2
1	19	20	22	21	26	22	24	20	22	24	1.909
2	23	17	22	23	22	16	28	27	24	18	6.545
3	22	18	19	25	15	30	26	22	18	25	8.545
4	18	19	24	23	20	23	18	28	24	23	4.181
5	24	16	20	22	27	19	18	21	28	25	6.363
6	22	25	14	30	21	25	16	22	17	28	11.09
7	14	15	33	15	24	21	23	25	25	25	14.36
8	22	22	20	17	22	29	16	18	21	33	11.45
9	28	17	16	17	22	20	23	18	23	36	15.45
10	11	17	16	22	23	21	27	19	29	35	19.81
11	18	21	19	24	16	27	22	28	20	25	6.363
12	16	13	23	19	25	24	18	13	34	35	25

The distribution of P-values and calculated χ^2 values for the custom TRNG test results is listed in Table 4-7.

Table 4-7 - Uniformity list of the RPG100 Diehard Test Results

Execution Number	**Distribution of generated 220 P-values over (0,1] for each execution of Diehard**										
	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	χ^2
1	26	19	16	22	25	18	23	23	17	31	8.045
2	17	23	27	17	22	20	32	30	18	14	14.72
3	26	21	14	18	22	26	26	21	16	30	10.45
4	14	24	17	22	18	20	20	34	22	29	14.09
5	27	14	16	23	22	31	15	26	22	24	12.54
6	19	14	19	33	13	22	33	19	21	27	20
7	22	25	21	23	21	17	22	18	24	27	4.22
8	26	22	23	24	26	23	17	20	21	18	3.81
9	18	24	21	17	17	22	22	32	25	22	8.18
10	24	27	16	25	25	20	21	20	19	23	4.63
11	24	19	19	21	25	22	16	29	19	26	6.45
12	14	26	27	21	20	24	15	22	25	26	8.544

The assessment of these χ^2 values of both TRNGs is done using $\chi^2 \leq 33.72$ and the results indicate that the distribution of 220 P-values that are generated by the diehard test suite for each execution with the random streams of custom TRNG and RPG100 have sufficient distribution uniformity.

4.3.3 Comparison of custom TRNG and RPG100

Cryptographic systems generally employ an external silicon chip TRNG such as RPG100 for random number generation. Employing a RPG100 chip on crypto system for random number generation has its own pros and cons. In this section of the thesis, we compare the RPG100 and our custom embedded TRNG from several aspects such as randomness quality, design security and throughput.

The randomness quality of a TRNG is evaluated by statistical tests. The FDK Corporation also confirmed the randomness quality of RPG100 by diehard and NIST test suites. This corporation shares results of these tests but their statistical results file do not contain detailed information. In order to be able to compare our custom TRNG and RPG100, we had to performed the same tests with FDK with the same parameters for confirming our custom TRNG and RPG100. The results of our tests revealed that both of the TRNGs have sufficient randomness to be used in cryptographic applications.

Besides external statistical evaluation of TRNGs the concurrent randomness tests are also recommended for security of TRNGs. The RPG100 chip has its own embedded FIPS 140-1, test which can be executed in real time during the generation. Therefore we also implemented these tests in the FPGA fabric integrated to our custom TRNG. These tests provide a concurrent and real time evaluation of the random numbers before they are used for any cryptographic issue. The security level of the random numbers generated is upgraded with such concurrent control features.

The throughput is another important feature of TRNGs because the random number need of a typical cryptographic system is getting larger and larger every day. The RPG100 chip generates true random bits with a constant throughput of 250 Kbps. However our custom TRNG has a variable throughput which is around 500 Kbps on the average. The throughput of our custom TRNG has a narrow standard deviation also. Although variable throughput of our custom TRNG is always higher than the throughput of RPG100.

If the TRNGs are used in an uncontrollable environment, the generated random bits should never be available for at the outside of the system. The RPG100 is an external device so it can easily be sniffed from its pins by attackers. On the other hand our custom TRNG is embedded in the FPGA, where the actual cryptographic application is also running on. Hence the attackers will not being able to sniff the generated random numbers, which is a feature that definitely, enhances the security of the overall system.

The effects of external changes (i.e. temperature) are not analyzed for our custom TRNG within the scope of this work but RPG100 makes a promise of supporting the same quality specifications for industrial operating conditions also.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Cryptographic systems requires random sequences for many processes such as key generation, authentication, padding and even for counter measures of side channel attacks. However random sequences that are employed in cryptographic systems must meet stringent requirements since the security of the system directly depends on these numbers. Simply stated these random numbers must be uncontrollable, unpredictable and irreproducible and must have good statistical properties in order to be eligible in cryptography. In addition to these, even if the employed random numbers have these features mentioned, a cryptographic system has to keep these numbers confidential for overall security issues. Hence it is generally recommended to realize the TRNGs inside the same implementation platform of the cryptographic algorithms. In contrast to older approaches where ASICs are mostly chosen for cryptographic applications, most of the implementations are realized with FPGAs during recent years. FPGA is common choice as the implementation platform because of its design simplicity, flexibility and re-programmability. Considering the prevalence of FPGAs in cryptographic systems, a TRNG built in FPGA is expected to improve the security and quality of a general cryptographic system.

In this thesis a TRNG, which is suitable for cryptographic applications is investigated, implemented and evaluated. The TRNG and its embedded statistical tests are described in VHDL language and then realized on an FPGA platform. The true random number extraction method proposed in [5] is employed for the implementation of the TRNG. The implementation can be realized in any FPGA of any vendor because it needs only the very common primitive resources of the FPGAs. For the concurrent statistical evaluation of the generated random numbers the randomness tests that are described in FIPS 140-1 are realized on the FPGA. Besides, the external statistical evaluation of the developed TRNG is also performed by using NIST statistical tests and Diehard battery of tests. These test suites are executed on a general purpose computer using the generated random streams of the TRNG. The implemented TRNG has a throughput up to 0.5 Mbps and the generation core occupies only 25 slices of the Xilinx Virtex-5 FPGA. This design shows us the possibility of generating and confirming true random bit sequences by using only the internal resources of FPGAs.

We have also compared our custom TRNG with RPG100, which is an external IC TRNG of FDK Corporation, from several aspects. We have performed the same statistical tests on the bit streams of RPG100. The statistical evaluation of both TRNGs shows that both are suitable to be employed in cryptographic applications. The throughput of our custom TRNG is higher than the throughput of RPG100 on average. The most important advantage of custom TRNG is being embedded in the FPGA that hosts the cryptographic application also. In contrast RPG100 is an external device, which is vulnerable to attacks of an adversary. As a result, using embedded TRNG in FPGA is more advantageous than using a separate IC such as RPG100 for true random number generation. This approach decreases vendor dependency and cost of a complete design. Besides, security of the whole system increases with embedded TRNG against external attacks. This thesis demonstrates that random number generation requirement of cryptographic systems can be satisfied by a TRNG built in FPGA where the system is already running on.

As a future study, other random number generation methods found in the literature can be implemented and tested in order to compare those with the already implemented one. The statistical evaluation of TRNGs can be done by the other statistical tests that exist in the literature such as the test of BSI [30]. ASIC implementation of the custom TRNG can also be considered together with the embedded test suite and an appropriate communication interface.

REFERENCES

[1] V. Fischer and M. Drutarovsky, “True Random Number Generator Embedded in Reconfigurable Hardware”, In Cryptographic Hardware and Embedded Systems - CHES 2002, Redwood Shores, CA, USA, Springer Verlag, Vol. 2523 of LNCS, 2002, pp 415-430.

[2] B. Sunar and D. Schellekens, “Secure Integrated Circuits and Systems Book, Chapter 6 - Random Number Generators for Integrated Circuits and FPGAs” 2010, pp 107-124.

[3] T. Williger, J. Guajardo, C. Paar, “Security on FPGAs: State of the art implementations and attacks”, Journal ACM Transactions on Embedded Computing Systems (TECS) , Volume 3, Issue 3, August 2004, pp 534 – 574.

[4] M. Varchola and M. Drutarovsky, “Embedded platform for automatic testing and optimizing of FPGA based cryptographic true random number generators” Radioengineering, Vol. 18, No. 4, 2009, pp 2-3.

[5] P. Kohlbrenner and K. Gaj, “An embedded true random number generator for FPGAs”, FPGA '04 Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, 2004, pp 71-78.

[6] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo, “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications”, NIST Special Publication SP800-22 rev 1a. (Revised: April 2010).

[7] G. Marsaglia, “The marsaglia random number cdrom with the diehard battery of tests of randomness” Supercomputer Computations Research Institute and

Department of Statistics, Florida State University, http://www.csis.hku.hk/diehard , last visited date, 10/12/2012 .

[8] FIPS Special Publication 140-1, “Security Requirements for Cryptographic Modules, Federal Information Processing Standards Publication 140-1”, U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Online http://csrc.nist.gov/publications/fips/fips1401.htm, 1994,last visited date, 10/12/2012 .

[9] FIPS Special Publication 140-2, “Security Requirements for Cryptographic Modules, Federal Information Processing Standards Publication 140-2”, U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Online. http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf, March 2002, last visited date, 10/04/2012 .

[10] FDK Corporation, “True Random Number Generator (TRNG) RPG100 datasheet Rev08”, http://www.fdk.com/cyber-e/pdf/HM-RAE106.pdf, last visited date, 10/11/2012.

[11] V. Fischer, A. Aubert, F. Bernard, B. Valtchanov, J.-L. Danger, and N. Bochard, “True random number generators in configurable logic devices,” Project ANR – ICTeR, February - 2009.

[12] M. Varchola, “FPGA Based True Random Number Generators for Embedded Cryptographic Applications (Thesis to the dissertation examination)”, Technical University of Kosice Faculty of Electrical Engineering and Informatics Department of Electronics and Multimedia Communications, December 2008.

[13] B. Sunar, W.J. Martin, and D.R. Stinson. “A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks”, IEEE TRANSACTIONS ON COMPUTERS, 2007 pp 109–119.

[14] NIST, “A Statistical Test Suite For Random And Pseudorandom Number Generators For Cryptographic Applications”, ITL Security Bulletin http://csrc.nist.gov/publications/nistbul/12-00.pdf, last visited date, 10/12/2012 .

[15] M. Dichtl and J.D. Golic, “High-Speed True Random Number Generation with Logic Gates Only” In Cryptographic Hardware and Embedded Systems - CHES 2007, Vienna, Austria, Springer Verlag, Vol. 4727 of LNCS, 2007, pp 45–61.

[16] S.K. Yoo, D. Karakoyunlu, B. Birand, B. Sunar, “Improving the Robustness of Ring Oscillator TRNGs”, http://ece.wpi.edu/~sunar/preprints/rings.pdf, 2008, last visited date, 13/07/2012 .

[17] T. E. Tkacik, “A hardware random number generator in Cryptographic Hardware and Embedded Systems”, CHES 2002, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers, ser. LNCS, Vol. 2523. Springer, 2003, pp 450-453.

[18] J.D. Golic, “New paradigms for digital generation and post-processing of random data” Technical report, Cryptology ePrint Archive, Report 2004/254, 2004.Online. Available: http://eprint.iacr.org/2004/254.ps, 2004, last visited date, 11/08/2012.

[19] J.D. Golic. “New Methods for Digital Generation and Postprocessing of Random Data” IEEE TRANSACTIONS ON COMPUTERS, 2006, pp. 1217–1229.

[20] I. Vasyltsov, E. Hambardzumyan, Y.-S. Kim, and B. Karpinskyy. Fast Digital TRNG Based on Metastable Ring Oscillator. In Elisabeth Oswald and Pankaj Rohatgi, editors, Cryptographic Hardware and Embedded Systems – CHES 2008, volume 5154 of LNCS, Springer, 2008, pp 164–180.

[21] M. Majzoobi, F. Koushanfar, and S. Devadas, “FPGA-based true random number generation using circuit metastability with adaptive feedback control,”

Cryptographic Hardware and Embedded Systems–CHES 2011, 2011, pp. 17–32.

[22] M. Varchola and M. Drutarovsky , "New High Entropy Element for FPGA based True Random Number Generators", Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop Santa Barbara, USA, Agust 2010, Springer, 2010, pp. 351-365.

[23] R. Tudoran , O. Cret , S. Banescu , A. Suciu, "Implementing true random number generators by generating crosstalk effects in FPGA chips", Proceedings of the 6th FPGA world Conference, September 2009, pp.25-31.

[24] O. Cret, R. Tudoran, A. Suciu, and T. Gyorfi, "Implementing True Random Number Generators in FPGAs by Chip Filling.", In Proceedings of the International Conference on Security and Cryptography, SECRYPT'09, 2009, pp. 62-67.

[25] T. Güneysu , "True Random Number Generation in Block Memories of Reconfigurable Devices" , Field-Programmable Technology (FPT), 2010 , pp 200 – 207.

[26] T. Güneysu , "Using Data Contention in Dual-ported Memories for Security Applications" , Journal of Signal Processing Systems, April 2012 Pages 15-29.

[27] B. Valtchanov, V. Fischer, A. Aubert, "Enhanced TRNG Based on the Coherent Sampling", 2009 International Conference on Signals, Circuits and Systems, 2009.

[28] Xilinx Constraints Guide http://www.xilinx.com/ support /documentation /sw_manuals /xilinx14_2 /cgd.pdf, UG625 (v. 13.4) ,last visited date, 10/12/2012.

[29] Xilinx Virtex-5 Libraries Guide for Schematic Designs http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_4/virtex5_scm.pdf, last visited date, 10/12/2012.

[30] BSI, “Anwendungshinweise und Interpretationen zum Schema (AIS 31)”, https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen /ais31.pdf, 2011, last visited date, 10/12/2012 .

[31] FDK Corporation, True Random Number Generation IC RPG100 / RPG100F catalog, http://www.fdk.com/cyber-e/pdf/HM-RAE101.pdf, last visited date, 10/11/2012.

[32] FDK Corporation RPG Business Promotion Dept., The Evaluation of Randomness of RPG100 by using NIST and Diehard Tests, http://www.fdk.com/cyber-e/pdf /HM-RAE104.pdf, last visited date, 10/11/2012.

[33] FDK Corporation, RPG100 Evaluation Board RPG100-TB(MB) User’s Manual Rev2 , http://www.fdk.com/cyber-e/pdf/HM-RAE102.pdf, last visited date, 10/11/2012.

APPENDIX – A

```
library unisim;

use unisim.vcomponents.all;

entity ring_osc is  port (

            osc_out            : out std_logic;

             reset             : in std_logic   );

  end ring_osc;

architecture low_level_definition of ring_osc is

signal ring_delay1      : std_logic;

signal ring_delay2      : std_logic;

signal ring_delay3      : std_logic;

signal ring_invert      : std_logic;

signal toggle           : std_logic;

signal clk_div2         : std_logic;

attribute KEEP : string;

attribute KEEP of ring_delay1 : signal is "true";

attribute KEEP of ring_delay2 : signal is "true";

attribute KEEP of ring_delay3 : signal is "true";

attribute INIT : string;

attribute INIT of div2_lut              : label is "1";
attribute INIT of delay1_lut          : label is "4";
attribute INIT of delay2_lut          : label is "4";
attribute INIT of delay3_lut          : label is "4";
attribute INIT of invert_lut          : label is "B";
-- Attribute for manually placement
attribute bel : string ;
attribute bel of delay1_lut : label is "D6LUT";
attribute bel of delay2_lut : label is "C6LUT";
attribute bel of delay3_lut : label is "B6LUT";
attribute bel of div2_lut    : label is "A6LUT";
attribute bel of invert_lut  : label is "A5LUT";
begin
  osc_out <= clk_div2;
  toggle_flop : FDCE
  port map ( D   => toggle,
            CLR => '0' ,
            CE  => '1' ,
            Q   => clk_div2,
            C   => ring_invert);
  div2_lut: LUT2
    generic map (INIT => X"1")
  port map( I0 => reset,
           I1 => clk_div2,
            O => toggle );
  delay1_lut: LUT2
    generic map (INIT => X"4")
  port map( I0 => reset,
            I1 => ring_invert,
             O => ring_delay1 );
  delay2_lut: LUT2
    generic map (INIT => X"4")
  port map( I0 => reset,
            I1 => ring_delay1,
             O => ring_delay2 );
  delay3_lut: LUT2
    generic map (INIT => X"4")
  port map( I0 => reset,
            I1 => ring_delay2,
             O => ring_delay3 );
  invert_lut: LUT2
    generic map (INIT => X"B")
  port map( I0 => reset,
            I1 => ring_delay3,
             O => ring_invert );
end low_level_definition;
```

APPENDIX – B

```
entity sampler is
  port(   clk_in1     : in  std_logic;
          clk_in2     : in  std_logic;
          enable      : in  std_logic;
          clear       : in  std_logic;
          randout     : out std_logic;
          bitready    : out std_logic
          );
  end sampler;
architecture low_level_definition of sampler is
signal rand_clk        : std_logic;
signal cnt0_n          : std_logic;
signal cnt0            : std_logic;
-- Attribute for manually placement
attribute bel : string ;
attribute bel of Sampling_FF : label is "AFF";
attribute bel of Rand_out_FF : label is "AFF";
attribute bel of Rand_cnt_FF : label is "AFF";
attribute bel of Bit_ready_FF    : label is "AFF";
attribute bel of invert_lut  : label is "A6LUT";
-- Attributes LOC
  attribute LOC : string ;
  attribute LOC of invert_lut : label is "SLICE_X2Y0";
  attribute LOC of Sampling_FF : label is "SLICE_X2Y1";
  attribute LOC of Bit_ready_FF : label is "SLICE_X3Y1";
  attribute LOC of Rand_cnt_FF : label is "SLICE_X2Y0";
  attribute LOC of Rand_out_FF : label is "SLICE_X3Y0";
begin
  Sampling_FF : FDCE
  port map ( D   => clk_in1,
             CLR => '0' ,
             CE  => '1' ,
             Q   => rand_clk,
             C   => clk_in2);
  Rand_cnt_FF : FDC_1
    port map ( D    => cnt0_n,
               Q    => cnt0,
               CLR  => clear,
               C    => clk_in2
               );
  invert_lut: LUT2
  generic map (INIT => X"B")
  port map( I0 => '0',
                  I1 => cnt0,
                  O => cnt0_n );
  Rand_out_FF : FDCE
  port map ( D   => cnt0,
             CLR => clear,
             CE  => enable,
             Q   => randout,
             C   => rand_clk);
  Bit_ready_FF : FDCE
  port map ( D   => '1',
             CLR => clear,
             CE  => enable,
             Q   => bitready,
             C   => rand_clk);

end low_level_definition;
```

APPENDIX – C

RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES

```
------------------------------------------------------------------------------
   generator is <data/custom_total.bin>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 95 107 108 108 100 100  84 100  94 104  0.825505    989/1000    Frequency
114  86  95  93 110 108 112  88  95  99  0.397688    989/1000    BlockFrequency
 94 108  99 116  75 108 115 113  91  81  0.026768    988/1000    CumulativeSums
 99  96 113 105 114  93  96  84 102  98  0.599693    987/1000    CumulativeSums
 87  98  95 106 100  88 112 116 105  93  0.482707    995/1000    Runs
104 114 107 112  80  95 114  92  84  98  0.141256    989/1000    LongestRun
103  89 101 119  91  98 108  94 102  95  0.630872    989/1000    Rank
110 110  98 104  89 108  95  93  91 102  0.775337    983/1000    FFT
 94 113  96  99 119  91 101  93  97  97  0.603841    989/1000    NonOverlappingTemplate
 89  95  85 106 110 105 108  88 108 106  0.494392    993/1000    NonOverlappingTemplate
136 100  83  95 103  83  90 115 104  91  0.005557    995/1000    OverlappingTemplate
103 100 119  99  96  99 103  97  98  86  0.733899    990/1000    Universal
109 102  92  81 121 101  94 100 103  97  0.345650    987/1000    ApproximateEntropy
 58  67  62  75  59  65  61  52  69  60  0.723673    617/628     RandomExcursions
 62  73  60  49  61  72  68  58  63  62  0.613623    622/628     RandomExcursions
 67  63  74  59  54  60  67  71  48  65  0.437274    623/628     RandomExcursions
 57  66  76  54  56  63  69  64  61  62  0.707249    620/628     RandomExcursions
 62  61  63  66  69  57  65  64  62  59  0.993769    622/628     RandomExcursions
 53  52  59  71  54  76  64  63  67  69  0.363700    620/628     RandomExcursions
 63  63  66  49  60  83  57  67  56  64  0.234060    623/628     RandomExcursions
 70  61  67  58  59  54  65  68  60  66  0.915607    614/628     RandomExcursions
 58  62  71  75  39  60  67  68  56  72  0.065546    623/628     RandomExcursionsVariant
 66  57  74  57  61  71  57  54  61  70  0.616976    621/628     RandomExcursionsVariant
 68  62  50  69  58  68  54  63  72  64  0.610271    622/628     RandomExcursionsVariant
 68  62  51  55  69  61  66  70  65  61  0.781044    623/628     RandomExcursionsVariant
 72  60  53  52  60  64  79  72  57  59  0.239982    622/628     RandomExcursionsVariant
 75  59  57  60  56  58  69  70  73  51  0.358376    619/628     RandomExcursionsVariant
 62  58  68  60  72  49  63  58  67  71  0.613623    623/628     RandomExcursionsVariant
 58  62  63  53  63  50  67  55  68  89  0.042096    623/628     RandomExcursionsVariant
 58  64  58  64  60  60  54  81  52  77  0.168532    619/628     RandomExcursionsVariant
 68  42  72  58  60  70  71  63  66  58  0.222556    621/628     RandomExcursionsVariant
 65  54  61  67  57  75  69  65  58  57  0.707249    620/628     RandomExcursionsVariant
 62  66  62  59  74  64  64  51  64  62  0.845408    624/628     RandomExcursionsVariant
 58  65  55  56  65  71  72  76  51  59  0.342708    621/628     RandomExcursionsVariant
 56  53  59  60  62  65  62  67  69  75  0.723673    618/628     RandomExcursionsVariant
 60  57  58  47  69  59  71  63  74  70  0.353103    620/628     RandomExcursionsVariant
 68  50  52  54  68  72  65  64  77  58  0.222556    618/628     RandomExcursionsVariant
 59  52  54  57  66  70  66  78  63  63  0.446255    620/628     RandomExcursionsVariant
 55  48  58  61  70  56  77  71  56  76  0.105114    620/628     RandomExcursionsVariant
117 112  90  83 107 107  81  96 100 107  0.142872    993/1000    Serial
113  96  97  88 103 108 106  98  90 101  0.786830    983/1000    Serial
101  96 104  93  98 125  91 110  82 100  0.204439    993/1000    LinearComplexity
```

- -

The minimum pass rate for each statistical test with the exception of the

random excursion (variant) test is approximately = 980 for a

sample size = 1000 binary sequences.

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

is approximately = 614 for a sample size = 628 binary sequences.

For further guidelines construct a probability table using the MAPLE program

provided in the addendum section of the documentation.

- -

APPENDIX – D

NOTE: Most of the tests in DIEHARD return a p-value, which should be uniform on [0,1) if the input file contains truly independent random bits. Those p-values are obtained by p=F(X), where F is the assumed distribution of the sample random variable X---often normal. But that assumed F is justan asymptotic approximation, for which the fit will be worst in the tails. Thus you should not be surprised with occasional p-values near 0 or 1, such as .0012 or .9983. When a bit stream really FAILS BIG, you will get p's of 0 or 1 to six or more places. By all means, do not, as a Statistician might, think that a p < .025 or p> .975 means that the RNG has "failed the test at the .05 level". Such p's happen among the hundreds that DIEHARD produces, even with good RNG's. So keep in mind that " p happens".

This is the BIRTHDAY SPACINGS TEST

Choose m birthdays in a year of n days. List the spacings between the birthdays. If j is the number of values that occur more than once in that list, then j is asymptotically Poisson distributed with mean m^3/(4n). Experience shows n must be quite large, say n>=2^18, for comparing the results to the Poisson distribution with that mean. This test uses n=2^24 and m=2^9, so that the underlying distribution for j is taken to be Poisson with lambda=2^27/(2^26)=2. A sample of 500 j's is taken, and a chi-square goodness of fit test provides a p value. The first test uses bits 1-24 (counting from the left) from integers in the specified file. Then the file is closed and reopened. Next, bits 2-25 are used to provide birthdays, then 3-26 and so on to bits 9-32. Each set of bits provides a p-value, and the nine p-values provide a sample for a KSTEST.

```
 BIRTHDAY SPACINGS TEST, M= 512 N=2**24 LAMBDA=  2.0000
```

```
        Results for custom_1.bin
              For a sample of size 500:     mean
  The 9 p-values were
      .335681   .623615   .512990   .638848   .052862
      .178181   .853718   .943854   .405282
 A KSTEST for the 9 p-values yields  .000665
```

THE OVERLAPPING 5-PERMUTATION TEST

This is the OPERM5 test. It looks at a sequence of one million 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. Thus the 5th, 6th, 7th,...numbers each provide a state. As many thousands of state transitions are observed, cumulative counts are made of the number of occurences of each state. Then the quadratic form in the weak inverse of the 120x120 covariance matrix yields a test equivalent to the likelihood ratio test that the 120 cell counts came from the specified (asymptotically) normal distribution with the specified 120x120 covariance matrix (with rank 99). This version uses 1,000,000 integers, twice.

```
        OPERM5 test for file custom_1.bin
For a sample of 1,000,000 consecutive 5-tuples,
 chisquare for 99 degrees of freedom= 97.795; p-value= .484594
        OPERM5 test for file custom_1.bin
    For a sample of 1,000,000 consecutive 5-tuples,
 chisquare for 99 degrees of freedom=140.726; p-value= .996237
```

This is the BINARY RANK TEST for 31x31 matrices.

The leftmost 31 bits of 31 random integers from the test sequence are used to form a 31x31 binary matrix over the field {0,1}. The rank is determined. That rank can be from 0 to 31, but ranks< 28 are rare, and their counts are pooled with those for rank 28. Ranks are found for 40,000 such random matrices and a chisquare test is performed on counts for ranks 31,30,29 and <=28.   ::

```
   Binary rank test for custom_1.bin
        Rank test for 31x31 binary matrices:
       rows from leftmost 31 bits of each 32-bit integer
     rank   observed  expected (o-e)^2/e  sum
       28       212     211.4   .001602     .002
       29      5033    5134.0  1.987349    1.989
       30     23247   23103.0   .896960    2.886
       31     11508   11551.5   .163993    3.050
  chisquare= 3.050 for 3 d. of f.; p-value= .656377
```

This is the BINARY RANK TEST for 32x32 matrices.

A random 32x32 binary matrix is formed, each row a 32-bit random integer. The rank is determined. That rank can be from 0 to 32, ranks less than 29 are rare, and their counts are pooled with those for rank 29. Ranks are found for 40,000 such random matrices and a chisquare test is performed on counts for ranks 32,31,30 and <=29.

```
   Binary rank test for custom_1.bin
        Rank test for 32x32 binary matrices:
       rows from leftmost 32 bits of each 32-bit integer
     rank   observed  expected (o-e)^2/e  sum
       29       196     211.4  1.124385    1.124
```

```
      30      4990    5134.0  4.039523    5.164
      31     23122   23103.0   .015549    5.179
      32     11692   11551.5  1.708293    6.888
  chisquare= 6.888 for 3 d. of f.; p-value= .928777
```

This is the BINARY RANK TEST for 6x8 matrices.

From each of six random 32-bit integers from the generator under test, a specified byte is chosen, and the resulting six bytes form a 6x8 binary matrix whose rank is determined. That rank can be from 0 to 6, but ranks 0,1,2,3 are rare; their counts are pooled with those for rank 4. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks 6,5 and <=4.

```
         Binary Rank Test for custom_1.bin
        Rank of a 6x8 binary matrix,
     rows formed from eight bits of the RNG custom_1.bin
     b-rank test for bits  1 to  8
   TEST SUMMARY, 25 tests on 100,000 random 6x8 matrices
  These should be 25 uniform [0,1] random variables:
     .750451     .071345     .421675     .155774     .827358
     .376755     .726652     .897187     .786412     .611982
     .303095     .129650     .127276     .429067     .609446
     .501428     .221000     .877219     .556928     .329686
     .369401     .314815     .970922     .116750     .752722
   brank test summary for custom_1.bin
       The KS test for those 25 supposed UNI's yields
                    KS p-value= .034584
```

THE BITSTREAM TEST

The file under test is viewed as a stream of bits. Call them b1,b2,... . Consider an alphabet with two "letters", 0 and 1and think of the stream of bits as a succession of 20-letter "words", overlapping. Thus the first word is b1b2...b20, the second is b2b3...b21, and so on. The bitstream test counts the number of missing 20-letter (20-bit) words in a string of 2^21 overlapping 20-letter words. There are 2^20 possible 20 letter words. For a truly random string of 2^21+19 bits, the number of missing words j should be (very close to) normally distributed with mean 141,909 and sigma 428. Thus (j-141909)/428 should be a standard normal variate (z score) that leads to a uniform [0,1) p value. The test is repeated twenty times.

```
THE OVERLAPPING 20-tuples BITSTREAM  TEST, 20 BITS PER WORD, N words

  This test uses N=2^21 and samples the bitstream 20 times.

 No. missing words should average  141909. with sigma=428.

tst no  1:  141469 missing words,   -1.03 sigmas from mean, p-value= .15179

 tst no  2:  141762 missing words,    -.34 sigmas from mean, p-value= .36534

 tst no  3:  142141 missing words,     .54 sigmas from mean, p-value= .70585

 tst no  4:  142516 missing words,    1.42 sigmas from mean, p-value= .92183

 tst no  5:  141984 missing words,     .17 sigmas from mean, p-value= .56925

 tst no  6:  141281 missing words,   -1.47 sigmas from mean, p-value= .07104

 tst no  7:  142413 missing words,    1.18 sigmas from mean, p-value= .88036

 tst no  8:  141494 missing words,    -.97 sigmas from mean, p-value= .16593

 tst no  9:  141636 missing words,    -.64 sigmas from mean, p-value= .26154

 tst no 10:  142805 missing words,    2.09 sigmas from mean, p-value= .98181

 tst no 11:  141948 missing words,     .09 sigmas from mean, p-value= .53600
 tst no 12:  141877 missing words,    -.08 sigmas from mean, p-value= .46990

 tst no 13:  141847 missing words,    -.15 sigmas from mean, p-value= .44211

 tst no 14:  141997 missing words,     .20 sigmas from mean, p-value= .58115

 tst no 15:  141887 missing words,    -.05 sigmas from mean, p-value= .47920

 tst no 16:  141490 missing words,    -.98 sigmas from mean, p-value= .16361

 tst no 17:  141875 missing words,    -.08 sigmas from mean, p-value= .46804

 tst no 18:  142332 missing words,     .99 sigmas from mean, p-value= .83831

 tst no 19:  142137 missing words,     .53 sigmas from mean, p-value= .70262

 tst no 20:  141694 missing words,    -.50 sigmas from mean, p-value= .30745
```

The tests OPSO, OQSO and DNA

OPSO means Overlapping-Pairs-Sparse-Occupancy The OPSO test 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 sequence to be tested. OPSO generates 2^21 (overlapping) 2-letter words (from 2^21+1 "keystrokes") and counts the number of missing words---that is 2-letter words which do not appear in the entire sequence. That count should be very close to normally distributed with mean 141,909, sigma 290. Thus (missingwrds-141909)/290 should be a standard normal variable. The OPSO test takes 32 bits at a time from the test file and uses a designated set of ten consecutive bits. It then restarts the file for the next designated 10 bits, and so on.

OQSO means Overlapping-Quadruples-Sparse-Occupancy The test OQSO is similar, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits from the test file, elements of which are assumed 32-bit random integers. The mean number of missing words in a sequence of 2^21 four letter words, (2^21+3 "keystrokes"), is again 141909, with sigma = 295. The mean is based on theory; sigma comes from extensive simulation.

The DNA test considers an alphabet of 4 letters C,G,A,T determined by two designated bits in the sequence of random integers being tested. It considers 10-letter words, so that as in OPSO and OQSO, there are 2^20 possible words, and the mean number of missing words from a string of 2^21 (over-lapping) 10-letter words (2^21+9 "keystrokes") is 141909. The standard deviation sigma=339 was determined as for OQSO by simulation. (Sigma for OPSO, 290, is the true value (to three places), not determined by simulation.

```
 OPSO test for generator custom_1.bin

  Output: No. missing words (mw), equiv normal variate (z), p-value (p)

                                                mw     z     p

   OPSO for custom_1.bin   using bits 23 to 32      142229  1.102  .8648
   OPSO for custom_1.bin   using bits 22 to 31      142369  1.585  .9435
   OPSO for custom_1.bin   using bits 21 to 30      142212  1.044  .8517
   OPSO for custom_1.bin   using bits 20 to 29      141672  -.818  .2066
   OPSO for custom_1.bin   using bits 19 to 28      141961   .178  .5707
   OPSO for custom_1.bin   using bits 18 to 27      142411  1.730  .9582
   OPSO for custom_1.bin   using bits 17 to 26      141994   .292  .6148
   OPSO for custom_1.bin   using bits 16 to 25      141853  -.194  .4230
   OPSO for custom_1.bin   using bits 15 to 24      141968   .202  .5802
   OPSO for custom_1.bin   using bits 14 to 23      141441 -1.615  .0532
   OPSO for custom_1.bin   using bits 13 to 22      141432 -1.646  .0499
   OPSO for custom_1.bin   using bits 12 to 21      141677  -.801  .2115
   OPSO for custom_1.bin   using bits 11 to 20      141948   .133  .5530
   OPSO for custom_1.bin    using bits 10 to 19        142017   .371  .6448
   OPSO for custom_1.bin    using bits  9 to 18       142267  1.233  .8913
   OPSO for custom_1.bin    using bits  8 to 17       141869  -.139  .4447
   OPSO for custom_1.bin    using bits  7 to 16       142165   .882  .8110
   OPSO for custom_1.bin    using bits  6 to 15       142239  1.137  .8722
   OPSO for custom_1.bin    using bits  5 to 14       142041   .454  .6751
   OPSO for custom_1.bin    using bits  4 to 13       141881  -.098  .4611
   OPSO for custom_1.bin    using bits  3 to 12       141376 -1.839  .0330
   OPSO for custom_1.bin    using bits  2 to 11       141798  -.384  .3505
   OPSO for custom_1.bin    using bits  1 to 10       142104   .671  .7490
 OQSO test for generator custom_1.bin
  Output: No. missing words (mw), equiv normal variate (z), p-value (p)
                                                mw     z     p
   OQSO for custom_1.bin    using bits 28 to 32        142003   .318  .6246
   OQSO for custom_1.bin    using bits 27 to 31        141829  -.272  .3927
   OQSO for custom_1.bin    using bits 26 to 30        142261  1.192  .8834
   OQSO for custom_1.bin    using bits 25 to 29        141573 -1.140  .1271
   OQSO for custom_1.bin    using bits 24 to 28        142092   .619  .7321
   OQSO for custom_1.bin    using bits 23 to 27        142247  1.145  .8738
   OQSO for custom_1.bin    using bits 22 to 26        141604 -1.035  .1503
   OQSO for custom_1.bin    using bits 21 to 25        141790  -.405  .3429
   OQSO for custom_1.bin    using bits 20 to 24        141726  -.621  .2672
   OQSO for custom_1.bin    using bits 19 to 23        141859  -.171  .4323
   OQSO for custom_1.bin    using bits 18 to 22        142236  1.107  .8659
   OQSO for custom_1.bin    using bits 17 to 21        142110   .680  .7518
   OQSO for custom_1.bin    using bits 16 to 20        141893  -.055  .4779
   OQSO for custom_1.bin    using bits 15 to 19        141873  -.123  .4510
   OQSO for custom_1.bin    using bits 14 to 18        141663  -.835  .2019
   OQSO for custom_1.bin    using bits 13 to 17        142148   .809  .7908
   OQSO for custom_1.bin    using bits 12 to 16        141528 -1.293  .0981
   OQSO for custom_1.bin    using bits 11 to 15        141761  -.503  .3075
   OQSO for custom_1.bin    using bits 10 to 14        142463  1.877  .9697
   OQSO for custom_1.bin    using bits  9 to 13        142191   .955  .8302
   OQSO for custom_1.bin    using bits  8 to 12        141942   .111  .5441
   OQSO for custom_1.bin    using bits  7 to 11        142075   .562  .7128
   OQSO for custom_1.bin    using bits  6 to 10        142082   .585  .7208
   OQSO for custom_1.bin    using bits  5 to  9        141856  -.181  .4283
   OQSO for custom_1.bin    using bits  4 to  8        141892  -.059  .4766
   OQSO for custom_1.bin    using bits  3 to  7        142510  2.036  .9791
   OQSO for custom_1.bin    using bits  2 to  6        141854  -.188  .4256
   OQSO for custom_1.bin    using bits  1 to  5        141845  -.218  .4137
 DNA test for generator custom_1.bin
 Output: No. missing words (mw), equiv normal variate (z), p-value (p)
                                       mw     z     p
    DNA for custom_1.bin    using bits 31 to 32        141819  -.266  .3949
    DNA for custom_1.bin    using bits 30 to 31        141559 -1.033  .1507
    DNA for custom_1.bin    using bits 29 to 30        141861  -.143  .4433
    DNA for custom_1.bin    using bits 28 to 29        141473 -1.287  .0990
 DNA for custom_1.bin   using bits 27 to 28     141487 -1.246  .1064
 DNA for custom_1.bin   using bits 26 to 27     141679  -.679  .2484
 DNA for custom_1.bin   using bits 25 to 26     142228   .940  .8264
 DNA for custom_1.bin   using bits 24 to 25     141723  -.550  .2913
 DNA for custom_1.bin   using bits 23 to 24     141563 -1.022  .1535
 DNA for custom_1.bin   using bits 22 to 23     142324  1.223  .8894
 DNA for custom_1.bin   using bits 21 to 22     142111   .595  .7240
 DNA for custom_1.bin   using bits 20 to 21     141967   .170  .5675
 DNA for custom_1.bin   using bits 19 to 20     141643  -.786  .2160
 DNA for custom_1.bin   using bits 18 to 19     142053   .424  .6641
 DNA for custom_1.bin   using bits 17 to 18     142310  1.182  .8814
 DNA for custom_1.bin   using bits 16 to 17     142064   .456  .6759
 DNA for custom_1.bin   using bits 15 to 16     141804  -.311  .3780
 DNA for custom_1.bin   using bits 14 to 15     141657  -.744  .2283
 DNA for custom_1.bin   using bits 13 to 14     142042   .391  .6522
 DNA for custom_1.bin   using bits 12 to 13     141224 -2.022  .0216
 DNA for custom_1.bin   using bits 11 to 12     142787  2.589  .9952
 DNA for custom_1.bin   using bits 10 to 11     142248   .999  .8411
 DNA for custom_1.bin   using bits  9 to 10     141920   .031  .5126
 DNA for custom_1.bin   using bits  8 to  9     142480  1.683  .9539
 DNA for custom_1.bin   using bits  7 to  8     142025   .341  .6335
 DNA for custom_1.bin   using bits  6 to  7     141714  -.576  .2822
 DNA for custom_1.bin   using bits  5 to  6     142027   .347  .6357
 DNA for custom_1.bin   using bits  4 to  5     141994   .250  .5986
```

```
   DNA for custom_1.bin    using bits  3 to  4       141646  -.777  .2186

   DNA for custom_1.bin    using bits  2 to  3       142041   .388  .6511

   DNA for custom_1.bin    using bits  1 to  2       141729  -.532  .2974
```

This is the COUNT-THE-1's TEST on a stream of bytes.

Consider the file under test as a stream of bytes (four per 32 bit integer). Each byte can contain from 0 to 8 1's, with probabilities 1,8,28,56,70,56,28,8,1 over 256. Now let the stream of bytes provide a string of overlapping 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's in a byte:: 0,1,or 2 yield A, 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E. Thus we have a monkey at a typewriter hitting five keys with various probabilities (37,56,70,56,37 over 256). There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chisquare test Q5-Q4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.

```
  Test results for custom_1.bin

 Chi-square with 5^5-5^4=2500 d.of f. for sample size:2560000

                        chisquare  equiv normal  p-value

  Results fo COUNT-THE-1's in successive bytes:

 byte stream for custom_1.bin     2510.48       .148     .558909

 byte stream for custom_1.bin     2421.22     -1.114      .132617
```

This is the COUNT-THE-1's TEST for specific bytes.

Consider the file under test as a stream of 32-bit integers. From each integer, a specific byte is chosen , say the leftmost bits 1 to 8. Each byte can contain from 0 to

8 1's, with probabilitie 1,8,28,56,70,56,28,8,1 over 256. Now let the specified bytes from successive integers provide a string of (overlapping) 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's, in that byte:: 0,1,or 2 ---> A, 3 ---> B, 4 ---> C, 5 ---> D, and 6,7 or 8 ---> E. Thus we have a monkey at a typewriter hitting five keys with with various probabilities 37,56,70,56,37 over 256. There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-letter words, counts are made on the frequencies for each word. The quadratic in the weak inverse of the covariance matrix of the cell counts provides a chisquare test:: Q5-Q4, the difference of the naive Pearson sums of (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.

```
Chi-square with 5^5-5^4=2500 d.of f. for sample size: 256000

                 chisquare  equiv normal  p value

  Results for COUNT-THE-1's in specified bytes:

          bits  1 to  8  2443.96      -.793      .214021

          bits  2 to  9  2416.46     -1.181      .118708

          bits  3 to 10  2526.06       .369      .643754

          bits  4 to 11  2491.55      -.119      .452443

          bits  5 to 12  2522.34       .316      .623997

          bits  6 to 13  2491.22      -.124      .450586

          bits  7 to 14  2625.15      1.770      .961623

          bits  8 to 15  2593.83      1.327      .907740

          bits  9 to 16  2449.62      -.712      .238096

          bits 10 to 17  2451.16      -.691      .244880

          bits 11 to 18  2507.65       .108      .543071

          bits 12 to 19  2434.88      -.921      .178539

          bits 13 to 20  2520.40       .288      .613514
```

bits 14 to 21	2525.94	.367	.643154
bits 15 to 22	2594.37	1.335	.909005
bits 16 to 23	2541.63	.589	.721994
bits 17 to 24	2591.28	1.291	.901629
bits 18 to 25	2509.22	.130	.551845
bits 19 to 26	2389.95	-1.556	.059817
bits 20 to 27	2439.70	-.853	.196882
bits 21 to 28	2479.03	-.297	.383389
bits 22 to 29	2562.59	.885	.811954
bits 23 to 30	2485.35	-.207	.417942
bits 24 to 31	2446.02	-.763	.222608
bits 25 to 32	2571.86	1.016	.845259

THIS IS A PARKING LOT TEST

In a square of side 100, randomly "park" a car---a circle of radius 1. Then try to park a 2nd, a 3rd, and so on, each time parking "by ear". That is, if an attempt to park a car causes a crash with one already parked, try again at a new random location. (To avoid path problems, consider parking helicopters rather than cars.) Each attempt leads to either a crash or a success, the latter followed by an increment to the list of cars already parked. If we plot n: the number of attempts, versus k the number successfully parked, we get a curve that should be similar to those provided by a perfect random number generator. Theory for the behavior of such a random curve seems beyond reach, and as graphics displays are not available for this battery of tests, a simple characterization of the random experiment is used: k, the number of cars successfully parked after n=12,000 attempts. Simulation shows that k should average 3523 with sigma 21.9 and is very close to normally distributed. Thus (k-

3523)/21.9 should be a standard normal variable, which, converted to a uniform variable, provides input to a KSTEST based on a sample of 10.

```
CDPARK: result of ten tests on file custom_1.bin
 Of 12,000 tries, the average no. of successes
      should be 3523 with sigma=21.9
 Successes: 3541    z-score:   .822 p-value: .794438
 Successes: 3561    z-score:  1.735 p-value: .958644
 Successes: 3555    z-score:  1.461 p-value: .928018
 Successes: 3548    z-score:  1.142 p-value: .873180
 Successes: 3520    z-score:  -.137 p-value: .445521
 Successes: 3553    z-score:  1.370 p-value: .914635
 Successes: 3497    z-score: -1.187 p-value: .117571
 Successes: 3489    z-score: -1.553 p-value: .060270
 Successes: 3534    z-score:   .502 p-value: .692266
 Successes: 3520    z-score:  -.137 p-value: .445521
square size   avg. no.  parked   sample sigma
  100.           3531.800       23.494
 KSTEST for the above 10: p=  .818998
```

THE MINIMUM DISTANCE TEST

It does this 100 times::   choose n=8000 random points in a square of side 10000. Find d, the minimum distance between the (n^2-n)/2 pairs of points. If the points are truly independent uniform, then d^2, the square of the minimum distance should be (very close to) exponentially distributed with mean .995 . Thus 1-exp(-d^2/.995) should be uniform on [0,1) and a KSTEST on the resulting 100 values serves as a test of uniformity for random points in the square. Test numbers=0 mod 5 are printed but the KSTEST is based on the full set of 100 random choices of 8000 points in the 10000x10000 square.

```
           This is the MINIMUM DISTANCE test
          for random integers in the file custom_1.bin
   Sample no.    d^2     avg     equiv uni
        5     .2568    .4633     .227466
       10     .3804    .6225     .317684
       15     .3710   1.0337     .311239
       20    2.7945   1.5499     .939706
       25     .0367   1.3904     .036204
       30     .9667   1.2787     .621501
       35     .1312   1.1506     .123576
       40    1.1381   1.1592     .681391
       45    1.4139   1.1469     .758522
       50     .2297   1.1258     .206137
       55    2.0995   1.1709     .878767
       60     .0429   1.1082     .042214
       65    5.2945   1.1434     .995113
       70     .2287   1.1530     .205350
       75     .2150   1.1202     .194297
       80    5.5842   1.1310     .996347
       85    1.7746   1.1254     .831956
       90    5.6926   1.1848     .996724
```

```
     95    2.3002   1.1930     .900917

    100    2.4611   1.1748     .915710

   MINIMUM DISTANCE TEST for custom_1.bin

        Result of KS test on 20 transformed mindist^2's:

                            p-value= .673039
```

THE 3DSPHERES TEST

Choose 4000 random points in a cube of edge 1000. At each point, center a sphere large enough to reach the next closest point. Then the volume of the smallest such sphere is (very close to) exponentially distributed with mean 120pi/3. Thus the radius cubed is exponential with mean 30. (The mean is obtained by extensive simulation). The 3DSPHERES test generates 4000 such spheres 20 times. Each min radius cubed leads to a uniform variable by means of 1-exp(-r^3/30.), then a KSTEST is done on the 20 p-values.

```
              The 3DSPHERES test for file custom_1.bin

 sample no:  1     r^3=  14.010     p-value= .37313

 sample no:  2     r^3=   4.483     p-value= .13881

 sample no:  3     r^3=   4.290     p-value= .13323

 sample no:  4     r^3=   8.167     p-value= .23833

 sample no:  5     r^3=  10.468     p-value= .29456

 sample no:  6     r^3=  11.586     p-value= .32037

 sample no:  7     r^3=   6.453     p-value= .19355

 sample no:  8     r^3=  27.781     p-value= .60388

 sample no:  9     r^3=  45.703     p-value= .78204

 sample no: 10     r^3=  11.222     p-value= .31207
```

```
sample no: 11    r^3= 124.081     p-value= .98401
sample no: 12    r^3=  18.753     p-value= .46479
sample no: 13    r^3=  36.836     p-value= .70709
sample no: 14    r^3=  27.195     p-value= .59606
sample no: 15    r^3=  26.891     p-value= .59195
sample no: 16    r^3=   2.189     p-value= .07038
sample no: 17    r^3=  25.173     p-value= .56789
sample no: 18    r^3=  44.948     p-value= .77648
sample no: 19    r^3=  11.705     p-value= .32305
sample no: 20    r^3=  88.002     p-value= .94678
 A KS test is applied to those 20 p-values.
      3DSPHERES test for file custom_1.bin        p-value= .199021
```

This is the SQEEZE test

Random integers are floated to get uniforms on [0,1). Starting with k=2^31=2147483647, the test finds j, the number of iterations necessary to reduce k to 1, using the reduction k=ceiling(k*U), with U provided by floating integers from the file being tested. Such j's are found 100,000 times, then counts for the number of times j was <=6,7,...,47,>=48 are used to provide a chi-square test for cell frequencies.

```
            RESULTS OF SQUEEZE TEST FOR custom_1.bin
         Table of standardized frequency counts
     ( (obs-exp)/sqrt(exp) )^2
        for j taking values <=6,7,8,...,47,>=48:
     1.3     -.3     .6     -.5     1.4     -.8
```

```
  1.3    -.4    -.9   -2.4    1.2     .5
 -1.9     .0    -.8     .4   -1.7     .8
 -1.0     .9    1.8     .8   -1.8    1.0
   .4    1.3     .1   -1.1    1.6    1.5
   .0    -.1     .9   -1.0    1.3    -.5
   .7     .8    1.3    -.7   -1.3     .0
  -.1
       Chi-square with 42 degrees of freedom: 50.305
          z-score=   .906  p-value= .822518
```

The OVERLAPPING SUMS test

Integers are floated to get a sequence U(1),U(2),... of uniform [0,1) variables. Then overlapping sums, S(1)=U(1)+...+U(100), S2=U(2)+...+U(101),... are formed. The S's are virtually normal with a certain covariance matrix. A linear transformation of the S's converts them to a sequence of independent standard normals, which are converted to uniform variables for a KSTEST. The p-values from ten KSTESTs are given still another KSTEST.

```
          Test no.  1      p-value  .250680
          Test no.  2      p-value  .636456
          Test no.  3      p-value  .306310
          Test no.  4      p-value  .204592
          Test no.  5      p-value  .001913
          Test no.  6      p-value  .956360
          Test no.  7      p-value  .606248
          Test no.  8      p-value  .223802
```

```
            Test no.  9      p-value  .562835
            Test no. 10      p-value  .001896
   Results of the OSUM test for custom_1.bin
        KSTEST on the above 10 p-values:  .931216
```

This is the RUNS test.

It counts runs up, and runs down, in a sequence of uniform [0,1) variables, obtained by floating the 32-bit integers in the specified file. This example shows how runs are counted: .123,.357,.789,.425,.224,.416,.95 contains an up-run of length 3, a down-run of length 2 and an up-run of (at least) 2, depending on the next values. The covariance matrices for the runs-up and runs-down are well known, leading to chisquare tests for quadratic forms in the weak inverses of the covariance matrices. Runs are counted for sequences of length 10,000. This is done ten times. Then repeated.

```
        The RUNS test for file custom_1.bin
    Up and down runs in a sample of 10000
             Run test for custom_1.bin   :
     runs up; ks test for 10 p's: .772382
    runs down; ks test for 10 p's: .953169
             Run test for custom_1.bin   :
     runs up; ks test for 10 p's: .597759
    runs down; ks test for 10 p's: .438695
```

This is the CRAPS TEST.

It plays 200,000 games of craps, finds the number of wins and the number of throws necessary to end each game. The number of wins should be (very close to) a normal with mean 200000p and variance 200000p(1-p), with p=244/495. Throws necessary to complete the game can vary from 1 to infinity, but counts for all>21 are lumped with 21. A chi-square test is made on the no.-of-throws cell counts. Each 32-bit integer from the test file provides the value for the throw of a die, by floating to [0,1), multiplying by 6 and taking 1 plus the integer part of the result. ::

```
              Results of craps test for custom_1.bin
 No. of wins:  Observed Expected
                         98483    98585.86
               98483= No. of wins, z-score= -.460 pvalue= .32274
  Analysis of Throws-per-Game:
Chisq=  19.90 for 20 degrees of freedom, p=  .53605
             Throws Observed Expected  Chisq     Sum
               1    66834    66666.7    .420     .420
               2    37818    37654.3    .711    1.131
               3    26537    26954.7   6.474    7.605
               4    19355    19313.5    .089    7.695
               5    13716    13851.4   1.324    9.018
               6    10025     9943.5    .667    9.686
               7     7207     7145.0    .538   10.223
               8     5191     5139.1    .525   10.748
               9     3735     3699.9    .334   11.082
              10     2586     2666.3   2.418   13.500
              11     1942     1923.3    .181   13.681
              12     1403     1388.7    .146   13.828
```

```
      13      969     1003.7   1.201   15.028
      14      752      726.1    .921   15.949
      15      533      525.8    .098   16.047
      16      377      381.2    .045   16.092
      17      275      276.5    .009   16.101
      18      222      200.8   2.232   18.332
      19      135      146.0    .826   19.159
      20      112      106.2    .315   19.474
      21      276      287.1    .430   19.904
   SUMMARY  FOR custom_1.bin
      p-value for no. of wins: .322741
      p-value for throws/game: .536055


Results of DIEHARD battery of tests sent to file custom_1.txt
```