2010

Digital Implementation of a True Random Number Generator

Sam Mitchum
Virginia Commonwealth University

Digital Design and Implementation of True Random Number Generators

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

Samuel Theodore Mitchum, Junior
Bachelor of Science in Engineering, University of South Carolina, 1981
Master of Engineering, University of South Carolina, 1986

Director: Dr. Robert H. Klenke
Associate Professor
Department of Electrical and Computer Engineering

Virginia Commonwealth University
Richmond, Virginia
December 2010

TABLE OF CONTENTS

ACKNOWLEDGEMENT

This dissertation is a continuation of development begun at Philips Semiconductor. The high cost and long lead times required to obtain analog chip design help on the existing random number generator circuit inspired the System Architect to charge me with developing a truly digital design containing no analog components. The Glen Allen office was shut down before this work was completed. Philips Semiconductor granted me permission to continue this development and to publish my findings. I would like to recognize the contributions of the following people from Philips Semiconductor:

Raj Maini, Director of Strategic Marketing, for giving me permission to continue

David Deland, Manager of the Glen Allen office

Dr. Jack Ehrhardt, System Architect and Bill Chauncey, Mathematician

Mark Harrison, Lucien Rainville, Bill Lester, team members

I would also like to thank my advisory committee from VCU:

Dr. Bob Klenke, Department of Electrical and Computer Engineering, Advisor

Dr. Jim Ames, Department of Computer Science

Dr. Jim Deveney, Professor Emeritus, Department of Mathematics

Dr. Alen Docef, Department of Electrical and Computer Engineering

Dr. Mike McCollum, Department of Electrical and Computer Engineering

I would like to thank Donovan Davis, Naveen Elangho and Jin Xu for helping with the layout of the RNG chip.

Finally I would like to thank the School of Engineering, Virginia Commonwealth University and especially the Administrative Assistants who have been very helpful.

CHAPTER 1 INTRODUCTION

1.1 Statement of Purpose

The purpose of this research is to develop a digital true random number generator that can be synthesized using standard digital design tools. Random number generators are used in security for generating secrets such as session keys and large primes for key exchange and exponentiation[1]. Random number generators are also used for simulating random events and for professional gaming. The security applications are of primary importance as the number and complexity of networks continues to grow. Random number generators will be required to protect the medical, financial and personal data of entities connected to these networks. A digital true random number generator that can be synthesized using standard digital tools will enable designers to address these privacy concerns more efficiently.

1.2 Problem Definition

Random number generators may be divided into two classes – pseudo random number generators and true random number generators. Pseudo random number generators generate a stream of numbers in a known pattern. The pattern is typically very long and it is hard to recognize the sequence of numbers is ordered. However, perfect knowledge of the generating circuit and the most recently generated number will enable the next generated number to be determined. For this reason pseudo random number generators are often called deterministic random number generators. Pseudo random

number generators are easily built from a Linear Feedback Shift Register (LFSR) assuming judicious selection of the XOR taps[2]. In order to disguise the fact that they are deterministic, PRNGs are often built with many more generated bits than are used per number. For example, a 128 bit LFSR may be used to implement a 32 bit PRNG. Protection comes from the fact that it is more difficult to discover the 96 hidden bits than the 32 bits used for the random number. Hence it is anticipated that the 128 bit implementation would be more secure. As with many real-world aspects of security, it is assumed that simply extending the length of the PRNG would provide acceptable privacy. However the issue of perfect knowledge of the generator determining the next output value is not addressed by this solution.

True random number generators produce a stream of truly random numbers. That is, knowing the generating circuit and the past history of numbers generated is insufficient to determine the next number. True random number generators are often called non-deterministic random number generators since the next number to be generated cannot be determined in advance. For a perfect true random number generator, the probability of the next generated number being any specific value should be equal to the probability of the next generated number being any other specific value. Since a certainty is always a probability of 1 and since some specific value will certainly be generated, the probability of any particular value being generated next should be equal to 1 (certainty) divided by the number of possible values in the range. As a simple illustration, consider a six sided die. The probability of any one of the six sides facing up after rolling the die is 1 (certainty) divided by 6 (the number of possible values). For a digital random number composed of N bits where N is positive definite, the range of

values has 2^N possible values. So the probability of any particular value being generated next by a true N-bit digital random number generator is:

Equation 1

$$P = \frac{1}{2^N}$$

At the inception of this research, true random number generators always required some analog components be included in ICs (Integrated Circuits). True random number generators were always based on an analog property like junction or thermal noise that was often whitened and scaled to produce a uniformly distributed random number generator. Whenever a SOC (System On a Chip) required random number generation, an analog IC designer was required to complete the design. A digital random number generator that can be designed using standard digital design tools would significantly reduce the cost and complexity of including a true random number generator in a design.

When a true random number generator is implemented in an FPGA (Field Programmable Gate Array), either several additional analog components such as resistors and operational amplifiers must be added to the design, or the designer must measure and match performance of individual logic blocks to achieve acceptable performance[3][4]. Again, a digital random number generator that can be designed and implemented using standard digital design tools would alleviate the need for extra components and/or the tedious hand-matching of logic blocks. This dissertation documents the design and implementation of a true random number generator using standard digital design methodology.

1.3 Overview

The remaining chapters of this dissertation are organized as follows; Chapter 2 covers methods for grading random number generators. Chapter 3 covers existing random number generator designs. Chapter 4 introduces the concept of divergent path RNGs. Chapter 5 covers two architecture proposals for non-deterministic digital random number generators. Chapter 6 presents the hardware and software used to test the generators. Chapter 7 contains the test results from the two proposed architectures. Finally chapter 8 presents possible future work on digital random number generation.

1.4 Abbreviations and Acronyms

The following abbreviations and acronyms are used throughout this paper.

Acronym	Stands For	Meaning
FPGA	Field Programmable Gate Array	A digital device whose architecture can be readily reconfigured.
LFSR	Logical Feedback Shift Register	A multibit parallel FIFO whose incoming bit is a logical combination of current bits.
NIST	National Institute of Standards and Technology	The government organization that promotes and publishes standards.
PRNG	Pseudo (Deterministic) Random Number Generator	A number generator that creates a repeatable sequence of numbers that appear to be random.
RN	Random Number	A number whose value cannot be predicted merely by knowing previous numbers in the sequence and the generating circuit.
RNG	Random Number Generator	A circuit for creating a sequence of numbers.
SOC	System On a Chip	A complete microprocessor based design incorporated on a single semiconductor.
TRNG	True (Non-deterministic) Random Number Generator	A number generator that creates a non-repeatable, non-predictable sequence of numbers.

CHAPTER 2 DEFINING AND MEASURING RANDOMNESS

2.1. Definition

In order to discuss random number generation, it is necessary to define what random means. Webster's New World College Dictionary gives three definitions for random[5]:

1) lacking aim or method; purposeless; haphazard

2) not uniform; especially of different sizes

3) Statistics of statistical sample selection in which all possible samples have equal probability of selection

No one of these three definitions is sufficient to describe a true random number generator. The first definition, haphazard, implies that the number stream from the RNG is not predictable. But is says nothing about the distribution of numbers generated. This definition would be satisfied with only a small portion of potential numbers actually being produced; for example if a six sided die was cast and always came up either 3 or 6 but never 1, 2, 4 or 5. The second definition, not uniform, adds nothing to the first definition. But the third definition, equal probability of selection, means that over time, the probability of each possible number being produced should be the same. To continue the analogy with a six sided die, this definition means that after rolling the die for a long time, each value in the set {1, 2, 3, 4, 5, 6} should have come up about the same number of times. How many samples constitute a long time and how close to equal constitutes about the same number of times are defined by the statistical tests for randomness. One other characteristic is defined by these tests: how haphazard (or unpredictable) the stream is. For example, a simple counter will produce every value in the count range and with

equal distribution. But a counter is not considered to be a RNG because the output is not haphazard; that is, the output is very predictable.

A random number generator outputs a stream of numbers. If the order of the numbers in the stream is exactly known then the stream is completely determined and there is no randomness in it. If the order of the numbers is not known then the stream has some degree of randomness. For most systems, a random number stream that is uniformly distributed is ideal. For such a system, the probability of any number being generated is equal to the probability of any other number being generated at any time. The tests used to measure the randomness in this paper all assume a uniform distribution. Each test has a unique way to measure the distribution.

2.2. Tests For Randomness

There are many ways to test the randomness of a stream of numbers. A few simple ones would be:

- count the number of "1" bits and the number "0" bits and make sure they are approximately the same
- break the stream into groups of say four bits and make sure that each possible four-tuple occurs roughly the same number of times (0000, 0001, 0010, 0011, …. 1111)
- pick a bit size and a particular pattern for that size and count how many bits are produced before that exact pattern is produced again.

There are many published sets of tests for randomness, including FIPS 140-1 and George Masaglia's Diehard Tests. The National Institute for Standards and Technology (NIST) has published a suite of statistical tests for determining the quality of a random number generator in publication 800-22 [6]. These are the tests that have been used to determine

the quality of the RNGs in this paper. This suite contains sixteen different tests which are detailed below.

1) Frequency Test: This test compares the proportion of 1's to 0's in the data. The proportion of 1's should be about half the number of bits. The test fails if there are too many or too few 1's in the bit stream.
2) Block Frequency Test: This test computes the proportion of 1's to 0's in a specified block size. For random data the frequency should be about half the block size. This test fails if there are too many blocks which have either too many or too few 1's.
3) Cumulative Sums Test: This test identifies the maximal excursion from 0 of a random walk using the values [-1, +1]. In other words, start at a point in the bit stream and move forward to the adjacent bit. If it is a "0" then SUM = SUM – 1. If it is a "1" then SUM = SUM + 1. If the bits alternated perfectly then the cumulative sum would remain low. If there are too many 1's or 0's in a row, however, the cumulative sum gets large. This test fails if the cumulative sum is either too large or too small.
4) Runs Test: This test counts the number of occurrences of runs of 1's. A run is defined as a continuous stream of bits of the same value bounded at the start and the end by bits of the opposite value. The expected results are more runs of shorter numbers of 1's and fewer runs of longer numbers of 1's. The test fails if there is significant deviation from the expected number of runs for any length of consecutive bits.

5) Longest Runs Test: This test counts the longest number of consecutive bits in each block of m bits. The test fails if there are too many consecutive 1's in the block.

6) Rank Test: This test divides the stream of binary bits into rows and columns of matrices. It then calculates the rank of each resulting matrix as a way of testing for linear dependence – hence too many repeated patterns. The test fails if ranks of the resulting matrices are incorrectly distributed.

7) Discrete Fourier Transform Test: This test examines the peak heights in the discrete Fourier transform of the sequence. The purpose is to detect repetitive patterns in the sequence. The test fails if the number of peaks exceeding a given threshold is too large.

8) Non-overlapping Template Matching Test: This test searches the bit stream for specific, aperiodic patterns. If the pattern is found, the search is started again just beyond the end of the pattern. If the pattern is not found, the search is started again at the next bit position. The test fails if too many occurrences of the pattern are found.

9) Overlapping Template Test: This test is similar to the non-overlapping template matching test except if the pattern is found, the search is continued from the next bit following the start of the pattern so that patterns which overlap are detected. Again the test fails if too many occurrences of the pattern are found.

10) Maurer's Universal Statistical Test: This test counts the number of bits between matching patterns in the data stream. This measure is related to how well the stream can be compressed. The test fails if the bit stream is compressible.

11) Approximate Entropy Test: This test compares the frequency of occurrence of all patterns of a certain bit length with the frequency of occurrence of all patterns that are one bit longer. The test fails if the difference in frequency of occurrence for the two lengths is not as expected for random data.

12) Random Excursions Test: This test is similar to the cumulative sum test in that a sum is calculated by taking a random walk from a point considered to be the origin and returning to that point. For each bit traversed, subtract 1 if the bit is a "0" and add 1 if the bit is a "1". The test actually examines eight different measurements – how many times each of the sums in the set [-4, -3, -2, -1, +1, +2, +3, +4] are encountered during a random walk. The test fails if the number of times each sum is encountered does not match that predicted for random data.

13) Random Excursions Variant Test: This test is a more stringent variation of the random excursions test. The difference is the number of sums. This test uses a total of eighteen sums, [-9, … -1, +1, … +9] where the random excursions test only uses eight. The test fails when the number of times each sum occurs does not match that expected for random data.

14) Serial Test: This test measures the frequency of occurrence of all possible overlapping patterns of a specified bit size. In a random stream, each pattern should occur approximately the same number of times. The test fails if the number of occurrences of each pattern is not approximately the same. Note for the case of 1 bit patterns, this test degenerates to the frequency test.

15) Lempel-Ziv Test: This test counts the number of cumulatively distinct patterns in the sequence. It is a measure of how much the bit stream can be compressed. The test fails if the bit stream can be compressed.

16) Linear Complexity Test: This test calculates the size of a LFSR that would be required to produce the bit stream. The test fails if the required LFSR is too small.

These are the sixteen tests that make up the NIST800-22 test suite for randomness. They are quite complete – a fact that may be inferred as NIST has not seen the need to update them. The scoring is somewhat arbitrarily as suggested by NIST at 96%; that is, a score of 95.9% fails while a score of 96.0% passes.

2.3 How Random is Random Enough?

Continuing the analogy with a six sided die, the value defined by rolling the die would be random. Assuming the die was well-constructed and balanced, there should be no way to predict what number would come up on any given roll. The probability should be equal that any one of the range of numbers {1, 2, 3, 4, 5, 6} would come up on any given roll. That is, if the die were cast many times, then the number of 1's should match the number of 2's, the number of 3's, the number of 4's, the number of 5's and the number of 6's. If this is true then the die is called "fair" and the probability of being rolled is equally distributed across all six elements of the range. Such a die would be called statistically random.

Suppose an extra dot is added to the 2 face of the die. Now there is no 2 face and there are two 3 faces. Now if the die is rolled there is still a 1 in 6 chance of the other four faces (1, 4, 5, 6) coming up. But there is no chance a 2 face will come up and there

is a 2 in 6 (or 1 in 3) chance of a 3 face coming up. Now, with this change, a haphazard number is still generated by rolling the die because the number that will come up is not perfectly predictable. But the probabilities are no longer evenly distributed. This uneven distribution is one aspect that tests like those in the NIST 800-22 suite are designed to detect.

Suppose another change is made to the die. The extra dot is removed so that all six numbers are present, but we make the entire 1 dot face heavier than the other faces. Now all six values can be rolled but the face across from the 1 dot will come up more often. If the heavy face is 1, the faces adjacent to the heavy face are {2, 3, 4, 5} and the face opposite the heavy face is 6. If the die is rolled many times then 1 will come up the least frequently, {2, 3, 4, 5} will come up with about equal frequency and 6 will come up most frequently. This variation in frequency is another aspect that statistical tests will measure.

Finally, in a perfectly fair die it is not possible to predict which number will be rolled following any other number. If it were possible to predict the order of numbers, for example {1, 4, 3, 6, 2, 5, 1, 4, 3, 6, 2, 5 …}, then the die would be of little use and that is obvious. But a more basic (and more likely) example of this same aspect would be if every time a 3 was rolled then a 1 would be rolled. The numbers rolled before the 3 and after the 1 could be perfectly haphazard. But every time a 3 was rolled, a 1 would be rolled next. While this would be obvious after a few dozen rolls with a six sided die, it would be much harder to detect with a random number generator having four billion output values. The statistical tests also find and measure these small patterns within the larger output sample.

2.4 Not-So-Random Number Generators

2.4.1 Linear Feedback Shift Register

A Linear Feedback Shift Register is a circuit composed of a chain of flip-flops. Each flip-flop output is tied to the next flip-flop input. All are clocked by a common clock. The input to the first flip-flop in the chain is a linear combination of the outputs of one or more of the flip-flops in the chain. See Figure 1 for an example of a LFSR.

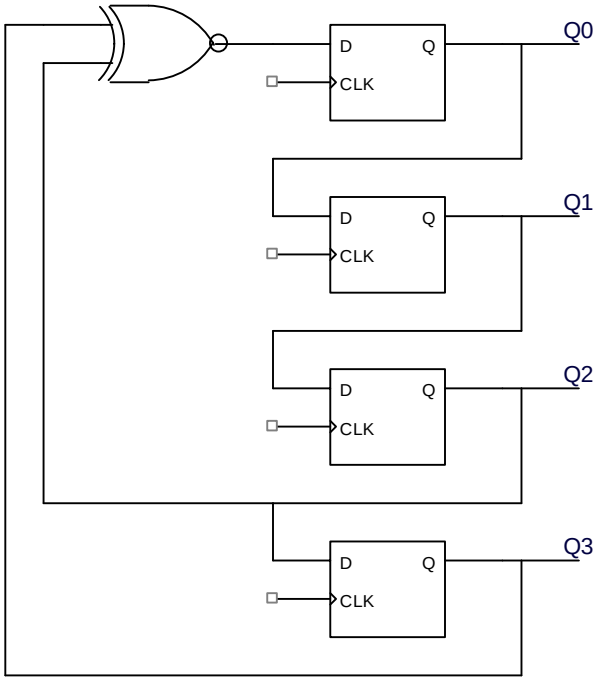

Figure 1

If the feedback is chosen correctly then a maximal length LFSR is obtained. A maximal length LFSR will cycle through every possible output value over and over – much like a counter except with a scrambled count. The output of a maximal length LFSR appears to be random though it is actually well ordered. The output of a maximal length LFSR will typically pass all statistical tests for randomness. These RNGs are often called PRNGs (pseudo random number generators) or deterministic RNGs because the output stream can be predicted (determined) mathematically.

Xilinx has published an application note that describes maximal length LFSRs and gives topography for up to 168 bits[7]. This application note also contains a bibliography for scholars interested in learning how appropriate feedback taps are chosen.

Unlike security applications and gaming, with simulation it is often more desirable to guarantee a particular distribution of random numbers than to guarantee lack of predictability of random numbers. In these applications a PRNG is an optimal choice. Simulation software frequently implements one or more PRNG distributions in order to model various combinations of timing delays, power fluctuations and other real world phenomena. If the calculation of new random numbers could be accelerated that would improve the speed of simulator and reduce the amount of time needed to run simulations. McCollum and others present a FPGA implementation for accelerating a PRNG and for controlling the distribution of the random number stream in [8].

2.4.2 Whitening

According to statistics, the variance of the linear combination of two sets of numbers is equal to the sum of the variances of the individual sets as long as there is no correlation between the two sets[9]. The linear combination can be realized by adding, subtracting or XORing the two sets. Equation 2 illustrates this principle.

Equation 2 Variance of TRNG XOR PRNG

$$V(x \hat{} y) = V(x) + V(y)$$

Since a TRNG is independent of any LFSR based PRNG by virtue of its definition, Equation 2 shows how the output of a TRNG can be statistically improved by XORing it with the output of an LFSR. Combining the output of the TRNG with the output of a

LFSR will increase the variance, and hence the randomness of the TRNG. This process is termed whitening and is well known in the practice of cryptology. Cusick and Stanica state "LFSRs can be applied in generating pseudorandom numbers, pseudonoise sequences, fast digital counters, *whitening sequences*, cryptography…"[10].

Thamrin, Witjaksono and others describe a RNG whose output is whitened by XORing with a LFSR[11]. Figure 2 below is a reproduction of Figure 7 in this paper for the purpose of illustrating such a circuit.

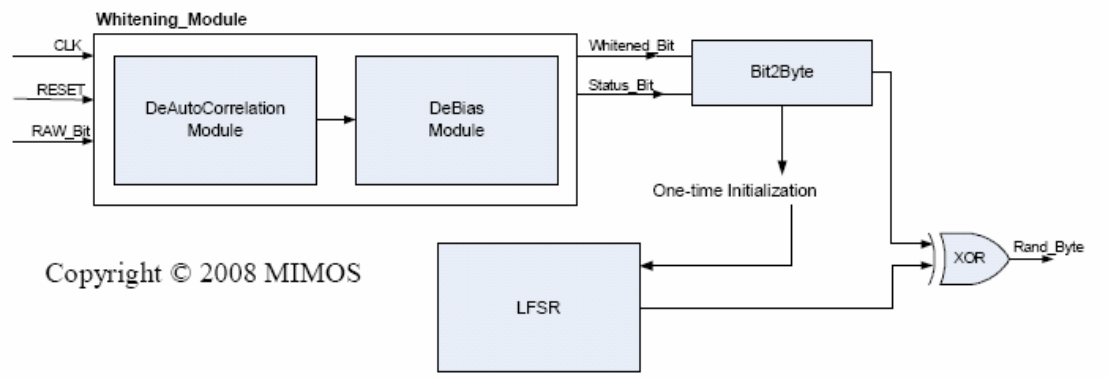

Figure 2 RNG Using LFSR Whitened By LFSR

Marsaglia presented a paper in 1968 proving that LFSRs do contain some frequency related correlation in the stream of generated numbers[12]. Therefore a TRNG linearly combined with a LFSR would exhibit better statistical properties than either the TRNG or the LFSR alone. Expressed another way, the TRNG could be linearly combined with the PRNG and the resulting combination would be more statistically random than either of the two input streams.

CHAPTER 3 BACKGROUND

3.1 Electronic RNG History

3.1.1 Analog RNGs

As mentioned before, analog electronic RNGs have been used for some time. Early analog RNG designs are based on amplifying electrical noise then converting the amplified signal to a digital signal. The circuitry for converting to a digital signal can be as simple as a clocked comparator. An example of such a TRNG is given in [13]. The block diagram from this Analog RNG, Figure 1 in [13], is reproduced in Figure 3 for convenient reference.

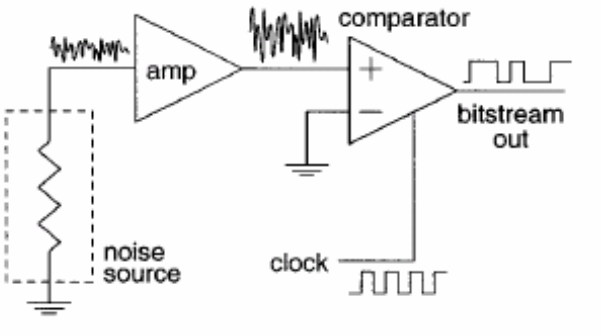

Figure 3 Simple Analog RNG [9]

If the frequency characteristics of the Analog RNG are inadequate for the application, a whitening filter can be placed either before or after the amplifier. If 32 bit random numbers are required then a 32 bit SIPO (serial in, parallel out) can be used to collect the bits into words. "This is the most popular RNG technique for single-chip or board-level solutions where shielding of the noise source is possible"[13].

Another publication of a similar analog RNG but having digital post processing is given in [14]. The analog noise source for this RNG is not just a resistor; rather it is an

A/D output compared to the reference voltage. Note this analog RNG requires whitening as shown by this quote "The proposed RNG exploits the direct amplification technique, using an accurate offset zeroing system, and, when its output is fed to a XOR-based decorrelating algorithm, the FIPS … and correlation randomness tests can be easily passed"[14]. This design illustrates an example of whitening RNG output by XORing with a PRNG output.

Each analog RNG starts with an analog noise source and amplifier. This reliance on analog circuitry presents an extra requirement for IC designers – that an analog IC designer be a part of the design team. Two other problems IC designers face with analog RNG designs are: "The lack of adequate shielding from power supply and substrate signals in an IC environment prohibits the exclusive use of this method for IC-based cryptographic systems"[13].

An interesting analog RNG design is given by Walsh and Beisterfeldt in [15]. The block diagram for this RNG is reproduced on the following page. This design utilizes a voltage controlled oscillator (VCO) to generate a waveform with a varying frequency. The control voltage on the VCO is generated by the output of a D/A converter. The input to the D/A converter is generated by a LFSR. The LFSR is clocked by the sample clock and its input bit comes from the VCO output sampled by a D flipflop that shares the same sampling clock as the LFSR. The output of the D flipflop is sent through a CRC32 generator to whiten it. The output of the CRC32 block is multiplexed with ground (0 volts) to form the random number output – 32 bits wide. The multiplexer select is generated by a counter so that 0 is output while the random number is being

formed then the random number is output after it is formed. Variations on this design (fast clock sampled by a much slower clock) have also been published as digital designs.

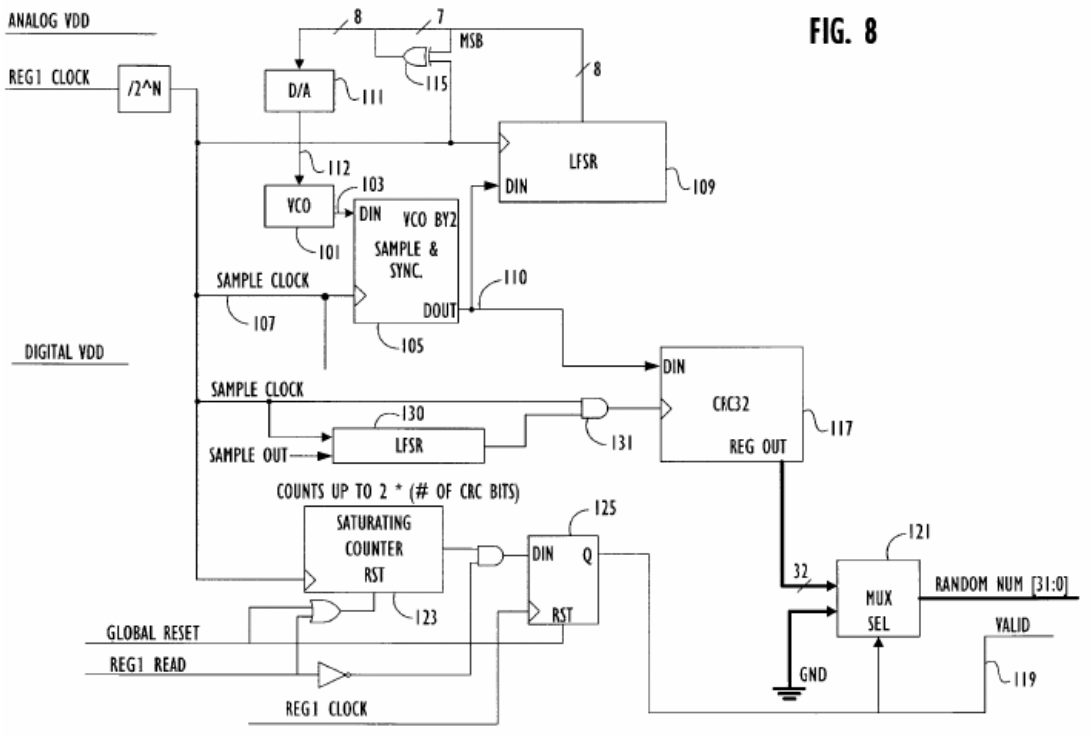

Figure 4 Oscillator Sampling RNG [15]

3.1.2 Chaos RNGs

RNGs have been designed using chaos theory. An excellent pair of papers on chaos based RNG design has been presented by Stojanovski and Kocarev[16][17]. In the first paper they cover the theory of chaos based RNG design while in the second paper they present an example of a chaos based RNG. The basic theory behind chaos based RNGs is that chaotic circuits tend to operate chaotically when observed at a course grain level, though they are deterministic in microscopic space and by their defining equations. The equation defining a chaos based TRNG appears to be perfectly solvable; however the solution requires infinite calculation resolution and infinite granularity in measurement. Since a digital number is being produced, neither the infinite calculation resolution nor

the infinite measurement granularity is available. Hence the domain must be partitioned into the same number of partitions as the number of digital possibilities. For example, to generate a 32 bit random number it would be necessary to partition the domain into 2^{32} states. Now the generator output will always be in one of the predefined states (it will have one of 2^{32} states which can be conveniently named [0 .. 4,294,967,295]. But since each state represents multiple values, the actual function output may have one of many values. Hence the next state, which can be perfectly calculated using the chaotic equation and the real value, is not visible in the digital world and will appear to be random. A simple example may clarify: Assume a really simple equation, $X_{n+1} = 2*X_n$, and a 2 bit RNG. Then we would partition the output domain into the following four states: 0 = [0..0.4999], 1 = [0.5 .. 0.9999], 2=[1.0 .. 1.4999] and 3=[1.5 .. 1.9999]. Table 1 illustrates how the digital output can vary because the actual value varies for this simple equation.

Table 1 Chaos RNG Example

X_n	random number	$X_{n+1}=2*X$	next random number
0.2499	0	0.4998	0
0.2500	0	0.5000	1
0.7499	1	1.4998	2
0.7500	1	1.5000	3

The second paper by these authors presents a RNG based on a single attractor. The structure of this RNG is shown in Figure 4 below. The structure is simple with only 29 MOS transistors and 2 resistors required. However, the presence of the resistors, and of course the balancing of the transistors, will require analog IC design methodology. This RNG cannot be designed using digital methods only.

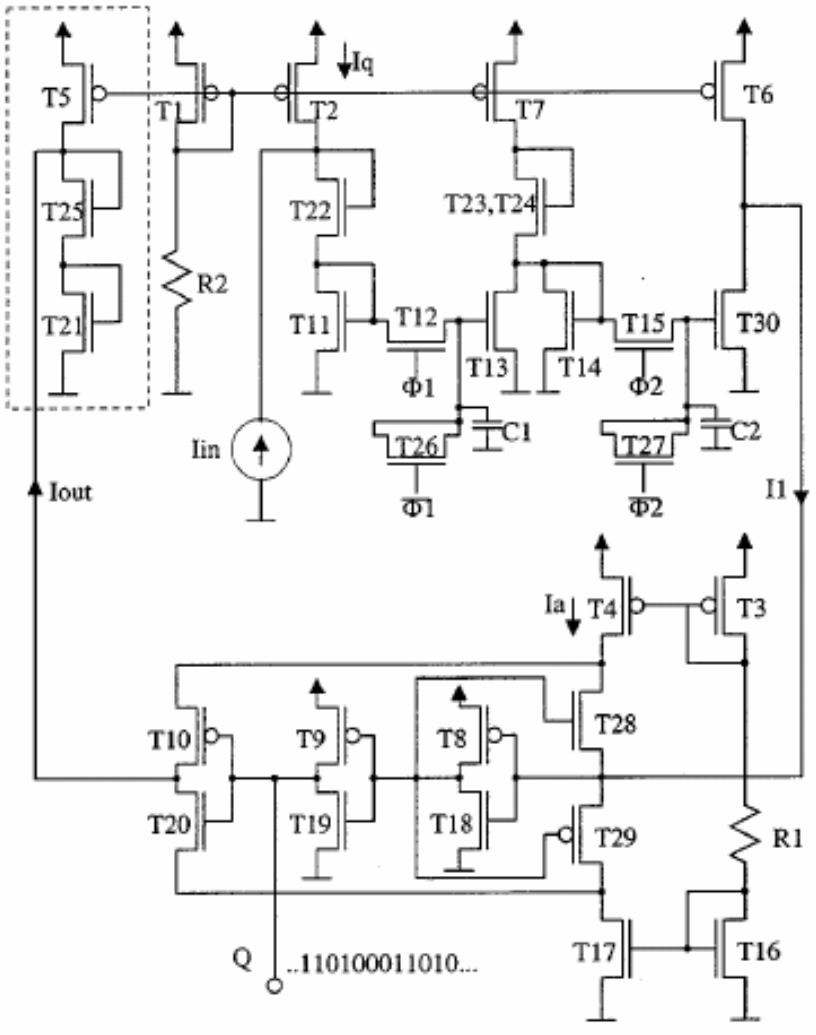

Figure 5 Chaos Based RNG [17]

3.1.3 Digital RNGs

The first practical digital RNGs were built from LFSRs as covered in Chapter 2. LFSRs produce a deterministic sequence that appears to be random but in fact repeats, at least within the size of the LFSR. For example, a 32 bit LFSR will produce at best a stream of 32 bit numbers that repeat after (2^{32} - 1) numbers. The exact sequence is controlled by the precise layout of the LFSR; specifically where the taps for feedback are located and whether they represent an XOR or an XNOR. The components of a LFSR are (1) one flipflop per bit and (2) the input to the LSBit is an XOR combination of the

outputs of the LFSR. Chu and Jones have documented not only the LFSR operation but several different architectures in [18].

Predating the formalization of LFSR design, Knuth patented a design for a random number generator. The difference between his design and a traditional LFSR based PRNG is that whether various stages in the shift register are either complemented or not is based on the value being shifted out[19]. Tausworthe[20] and MacLaren and Marsaglia[21] were contemporaries of Knuth who published similar designs. Each of these designs exhibited similar probability distributions.

A variant on a simple LFSR is an LFSR with more bits than are required. One popular choice has been a 128 bit LFSR with only 32 bits of random number used. It is hoped that failure to expose all the bits will keep third parties from predicting the stream of numbers. Nonetheless this design still contains the weakness of a known generating circuit and a pattern that can eventually be traced.

As mentioned before, another popular structure for a digital RNG is a slow clock sampling a fast clock. Some papers refer to this style as oscillator sampling. Figure 2 in [13] shows such a design and is reproduced below in Figure 6.

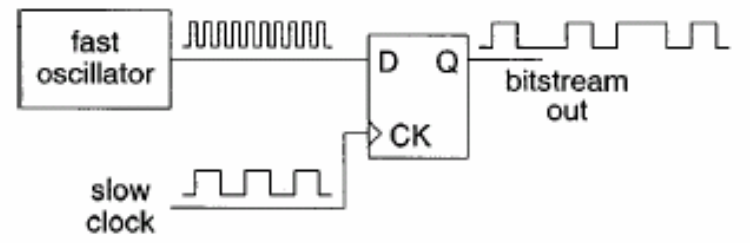

Figure 6 Oscillator Sampling from Fig2 in [13]

While this design looks deceptively simple, there is often considerable post processing required to whiten the resulting random number stream as it is often impractical to

completely isolate the sample clock from the faster oscillator. That is, some common frequency remnants must be whitened out. Some papers refer to this whitening as decorrelating the output.

Another realization of a digital RNG constructed from oscillator sampling is given by [3]. This design features a single D flipflop that handles the sampling. The TRNG output is available but is also used to seed a PRNG. Using a TRNG to seed a PRNG has also become popular because the unpredictability of the TRNG is merged with the statistically desirable qualities of the PRNG. FIPS publication 140-2 specifically recommends using a TRNG to seed a PRNG as the safest way to construct a RNG [22].

A more complex architecture for an oscillator sampled RNG is given in [23]. Note in this design, illustrated in Figure 7, the low frequency clock is the jittery one while the high frequency oscillator is being sampled. Note also the presence of post processing to whiten the RNG output.

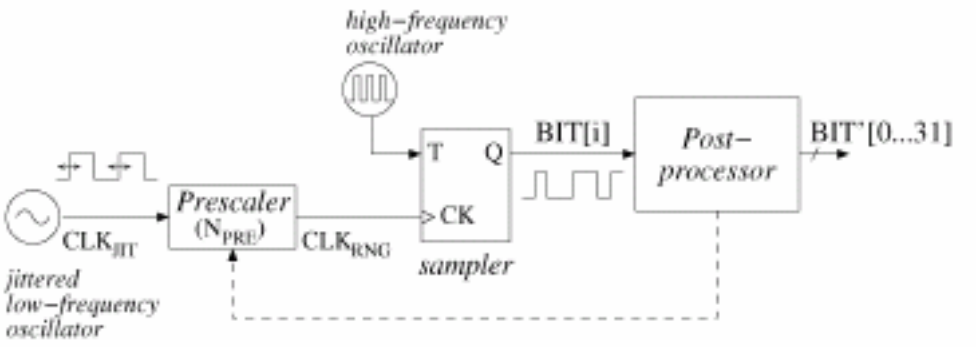

Figure 7 Oscillator Sampled RNG from [23]

3.1.4 RNG's Similar To This Work

Several designs similar to the one documented herein have been presented. McTaggert and Burson have presented a TRNG based on free running clocks[24]. In this RNG, there are two LFSRs, each clocked by a separate and unrelated free running

oscillator. One of the oscillators is crystal based. The other oscillator is not crystal based but its architecture is not disclosed. The LFSRs are of different lengths – one is 39 stages and one is 23 stages. The LSBits of the two LFSRs are XORed together. The output of the XOR is sampled by a flipflop to form the random number. The clock for the flipflop is independent of the two LFSR clocks. Provisions have been made for seeding the random number by preloading parts of the LFSRs. Figure 8 below shows the circuitry for this random number generator.

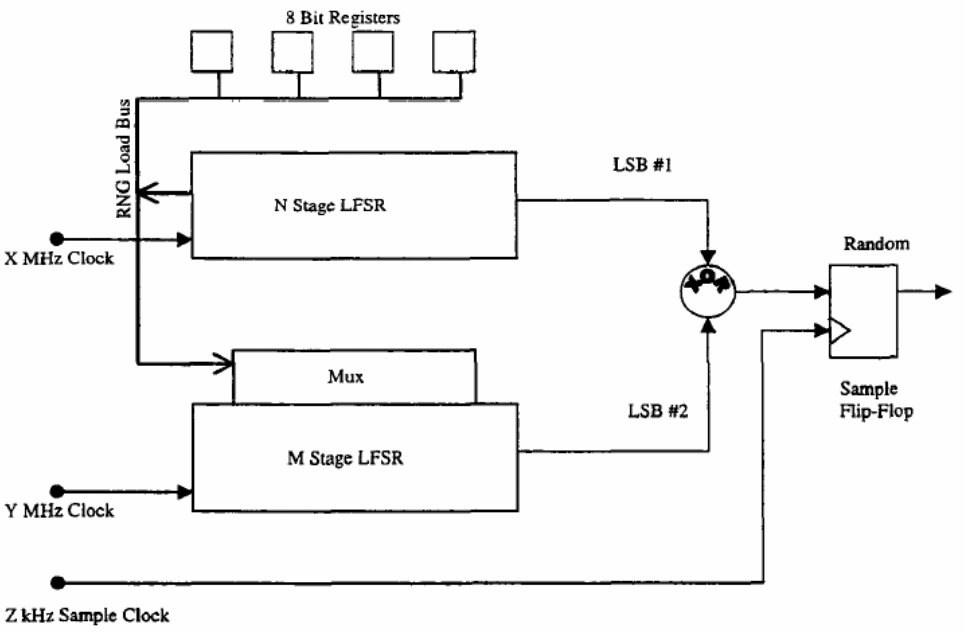

Figure 8 Multi Clock RNG[24]

A TRNG based on a LFSR clocked by a random clock is presented in [25]. The random clock is composed of the XOR of several unrelated ring oscillators. That signal is then sampled by a flipflop and presented as the clock for a maximal length LFSR. The number of inverters in each of the various ring oscillators is required to be prime relative to the number of inverters in the other ring oscillators in order to lessen the possibility of

the oscillators locking into the same frequency or a related harmonic. The use of at least three oscillators is advised in case one oscillator locks to the bus clock. This concept is interesting because all of the randomness comes from the clock waveform as opposed to any shuffling, scrambling or whitening of the produced bit stream. Figure 9 shows a typical realization of this type of random number generator.

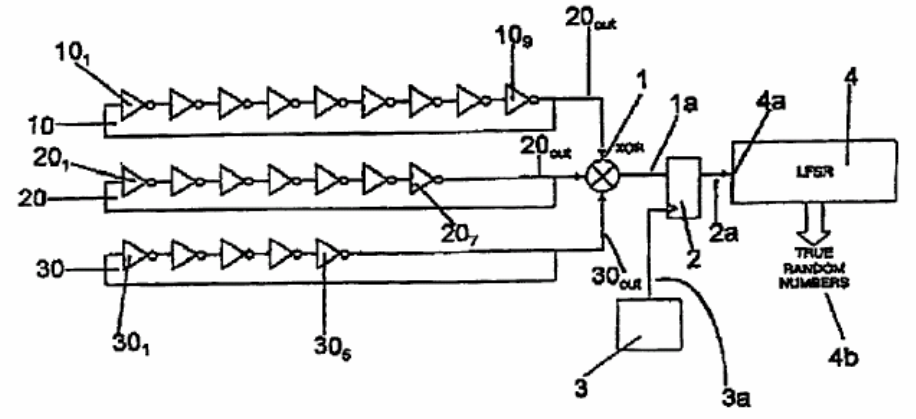

Figure 9 LFSR with random clock[25]

Wilbur has patented another RNG implementation involving multiple ring oscillators[26], Note the presence of two ring oscillators generating randomness. Each ring oscillator has an "enhanced" output which is the XOR of several taps from the ring.

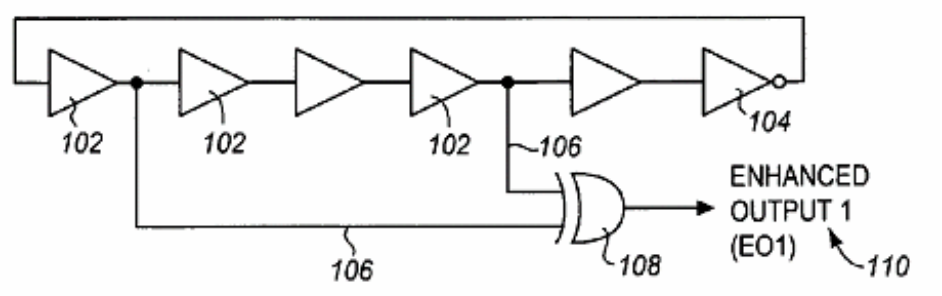

Figure 10 Ring Oscillator With Enhanced Output[26]

Figure 11 illustrates the entire TRNG block diagram as presented in the patent disclosure. The outputs of the ring oscillators drive a delay line. The delay line provides

multiple taps into the Combiner-Sampler. The Combiner-Sampler XORs the taps together. The output of the Combiner-Sampler is XORd with the output of the sample flipflop. The output of the sample flipflop is fed to a second sampling flipflop to produce the random output.

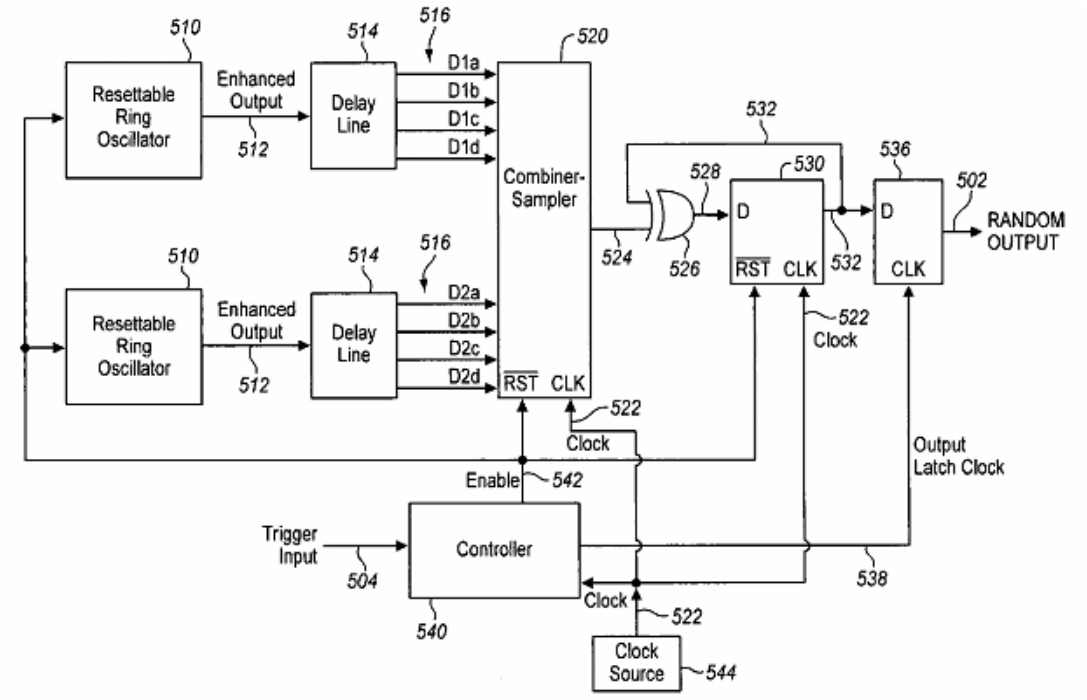

Figure 11 TRNG Based On Scrambled Clocks[26]

CHAPTER 4 DIGITAL TRNG DESIGNS

4.1. Overview

The purpose of this research is to develop a digital true random number generator that can be synthesized using standard digital design tools. Developing a digital TRNG composed of standard digital components is important because:

- It alleviates the need for analog circuit design.
- The RNG can be incorporated with other digital cryptographic components.
- No external components are required for FPGA implementations.

A general architecture for digital TRNGs will be developed.

4.2 Randomness

4.2.1 Randomness In The Analog Domain

Randomness in the analog domain has long been accepted in the form of signal noise. The signal noise is usually present as a small amplitude signal superimposed over the intended signal and may often be seen as tiny vertical perturbations in an oscilloscope trace of the signal. The noise is comprised of several aspects including thermal noise, noise picked up from a power supply and junction noise. Often the frequency characteristic of signal noise is such that a suitable RNG can be obtained by simply subtracting the signal then scaling the noise. Otherwise one or more frequencies may be filtered out of the noise source. At that point the noise can be sampled and converted to digital values as required.

4.2.2 Randomness In The Digital Domain

In the digital domain, every effort is usually made to prevent randomness in amplitude. Each piece of information is represented by a bit which is resolved to a "0" or a "1". The clock rate for a digital circuit is usually chosen so that all transitions from "0" to "1" or from "1" to "0" are allowed to complete between clocks. Hence digital circuits are designed to be immune to amplitude noise. Many digital clock signals are designed to be quite repeatable to enable not only a predictable period for the afore-mentioned transitions but also to enable accurate timing. Often the clock signal is fed back through a crystal to restrict the frequency of operation. Such crystal controlled oscillators can be accurate to 20 ppm (parts per million) depending on how exactly the mechanical crystal properties are controlled when the crystal is cut. There is very little noise in such a crystal controlled clock source and hence little randomness to recover in either the amplitude or the period of oscillation.

A noisy oscillator may be formed by connecting inverters in series then connecting the last output to the first input to form a ring. Each inverter alters the phase of the signal by 180 degrees so in order for the circuit to oscillate, there must be an odd number of inverters in the ring. Such an oscillator is called a ring oscillator and oscillates with a period equal to twice the total propagation delay around the ring. Figure 12 is an example of a ring oscillator.

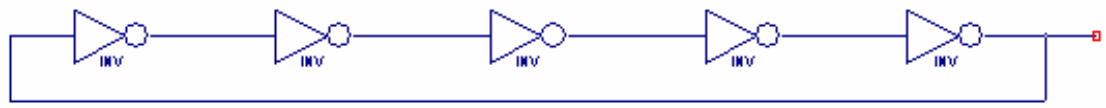

Figure 12

These ring oscillators require very little chip area at only two transistors per CMOS inverter stage and are easy to build with either schematics or HDL. Ring

oscillators are also susceptible to substantial perturbations in the oscillation period called jitter which can easily be several percent of the nominal oscillator period[27]. This jitter makes a ring oscillator a reasonable source of randomness. A different method will be needed to capture this randomness since it is in the clock period rather than in the signal amplitude, however. Where the noise source in an analog TRNG would be visible as vertical perturbations on an oscilloscope trace, the noise source in a digital TRNG would be visible as horizontal perturbations on an oscilloscope trace.

4.2.3 Distilling Randomness From Clock Jitter

Distilling randomness from an analog noise source is straightforward. Use a sample and hold to freeze the noisy signal long enough to perform a digital conversion to the required number of bits. Distilling randomness from digital clock jitter is more complex. If a straight conversion, analogous to the noise source conversion, is performed then the converting circuit would need to resolve to (2^N) values where N is the number of bits required for the random number. The resolution would have to occur in one (maximum period – minimum period) time and that would require a counter with a count interval as shown in equation 3.

Equation 3

$$CountInterval = \frac{(MaxPeriod - MinPeriod)}{2^N}$$

Therefore if the change in period is 10 nanoseconds and the random number is 32 bits, the counter interval would be 2.3 x 10^{-18} seconds. Direct conversion to obtain the random number is not possible using current technology. Hence it will be necessary to gather randomness a few bits at a time and build it up into a significant word size.

4.2.4 Range of Numbers and Paths Through The Range

A digital RNG generates numbers which are based on powers of 2. The number of bits in the generated number defines the range of the generated number since each bit can have one of two possible values, 0 or 1. The number of possible values that can be represented is 2^N where N is the number of bits. Thus the range of numbers that can be generated by a RNG having N bits is [0 .. 2^N-1]. A digital RNG is constrained to generate a random number within this specified range. For instance, a four bit RNG can only generate numbers from the set [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15].

When arithmetic operations result in carries beyond the most significant bit (MSB) position these carries are typically ignored. When a RNG would attempt to generate a number outside of the allowed range, a modulo operation is realized to bring the number back within range. For example, if +1 is added to the maximum count for a four bit RNG, the resulting number would be beyond the range that can be expressed in four bits. So the carry beyond the MSB is ignored and the result of 15 + 1 is 16 MOD 16 or 0.

A generating function traverses a path through the range of numbers generated. A very trivial example is a counter. On each cycle "1" is added to the number. The sequence generated is {0, 1, 2, 3, 4, .. (2^N-1), 0, 1, 2, …}. Notice the modulo operation returns the generator output back to the allowed range at 0. The particular example of a counter has some interesting properties:

- Each number in the range of numbers is generated – no numbers are skipped.
- The frequency of each generated number is the same; that is, a "2" is generated just as often as a "3".

- The pattern or sequence of numbers generated is always the same. Expressed another way, the counting generator traverses the same path through the range of numbers over and over.

Not all functions share these properties. For instance doubling, or multiplying by two, also traverses a path through the range of numbers. But unlike counting, doubling will not generate every value in the range – only the powers of 2. Any function used to construct a RNG should be capable of generating all numbers in the range.

There are many functions that can generate all the numbers in the range. Instead of a counter, adding any constant that is relatively prime with the size of the range (the modulus) will generate every value in the range. The proof is straightforward. Let a be the relatively prime constant used to generate the range and m (the size of the range) be the modulus. Then starting at zero and adding a each time followed by a modulo operation, the results are shown in Equation 4. Note there would be a total of m values generated – one value generated for each value in the range.

Equation 4 Generating Range Using a Relatively Prime Constant

$$((0 * a) + a) \bmod m = a$$
$$((1 * a) + a) \bmod m = (2 * a) \bmod m$$
$$((i * a) + a) \bmod m = ((i + 1) * a) \bmod m$$

Each calculated number must fall in the inclusive range [0 to (m-1)] by definition of the modulus operation. There are as many calculated numbers as there are values in the range. Each calculated number must be unique as stated by the Modified Cancellation Law for Congruences illustrated by Equation 5. Therefore all values in the range are generated by successively adding a constant that is relatively prime with the modulus.

Equation 5 Modified Cancellation Law

$$((i \bullet a) \bmod m) = ((j \bullet a) \bmod m)$$
$$\therefore i = j$$

Since $i = j$ then every unique value of i ε [0 .. (m-1)] must generate a unique number within the range of possible numbers. A simple example is easily computed for the case of a 3 bit range (m = 8) and a relatively prime a = 3 as shown by Equation 6.

Equation 6 Example of Relatively Prime Generation

$$(0 * 3) \bmod 8 = 0$$
$$(1 * 3) \bmod 8 = 3$$
$$(2 * 3) \bmod 8 = 6$$
$$(3 * 3) \bmod 8 = 1$$
$$(4 * 3) \bmod 8 = 4$$
$$(5 * 3) \bmod 8 = 7$$
$$(6 * 3) \bmod 8 = 2$$
$$(7 * 3) \bmod 8 = 5$$

4.2.5 Capturing Randomness Using Divergent Paths

In order to explore the concept of randomness within a generator it is necessary to make an abstraction, separating the generator and the sampler. If the sampling is not at perfectly uniform intervals then it will affect the apparent randomness of the generator. Normally it is expected any randomness within the sampler will increase the apparent randomness of the generator. Therefore this paper postulates a random number generator producing random numbers and a sampler reading random numbers at fixed intervals.

If a RNG is constructed from a generating function such as a counter that is clocked by a noisy source such as a ring oscillator then some slight amount of randomness may be observed in the following way. Let the generator be a free running counter clocked by a ring oscillator at an average rate of 100MHz but with a noise of +/- 1MHz and let the sample interval be 1 microsecond (1MHz) as shown in Figure 13.

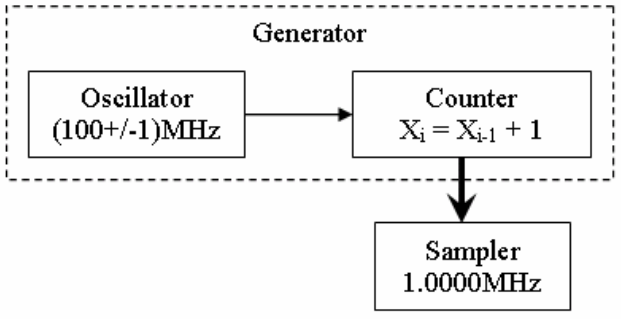

Figure 13 Simple Generator and Sampler

The counter runs at 100MHz on average. But at any given sample, the generator could have run at 99MHz, 100MHz or 101MHz because of the noisy oscillator driving the counter. Now, ignoring terminal count issues, the value of this generator at any sample time can be predicted within 3 counts by multiplying the number of sample intervals times the average oscillator frequency divided by the sample frequency. Although the count can be predicted within 3 counts, it cannot be predicted exactly – hence there is some randomness in the value. For example, on the fifth sample the value of this generator could be 499, 500 or 501. On the seventh sample the value of this generator could be 699, 700 or 701. On the nth sample, the value of this generator would be (n x 100) + {-1, 0, 1}. There is a small amount of uncertainty about the value that is due to the noisy oscillator in the generator. That uncertainty in the generator needs to be collected and preserved. A simple counter cannot preserve it. In fact no generator that

has a single path through the range of generated numbers can preserve this uncertainty. The value at any point in the future may be found be interpreting the generator once for each sample interval and then adding the uncertainty to the final value. In general, the value at the Nth sample from this type of generator is given by Equation 7.

Equation 7 Values From a Single Path

$$R_0 + R_1 + R_2 + R_3 ... = (N \bullet \Delta R_{AVG}) + U$$
$$where U \in [-1,0,+1]$$

The uncertainty is not preserved across multiple samples as is shown by Equation 7. All of the uncertainty or randomness can be considered independent of how many samples were taken. In order to create a TRNG or True Random Number Generator, a way is needed to capture and store the randomness generated at each sample. Capturing and storing the randomness at each sample can be accomplished by altering the path that the generator traverses through the range of possible values at each sample. If the path is altered at each sample then the state of the generator, including whatever randomness is captured, can be preserved in the output value. Figure 14 is a visualization of how altering the path of the generator at each sample can preserve randomness.

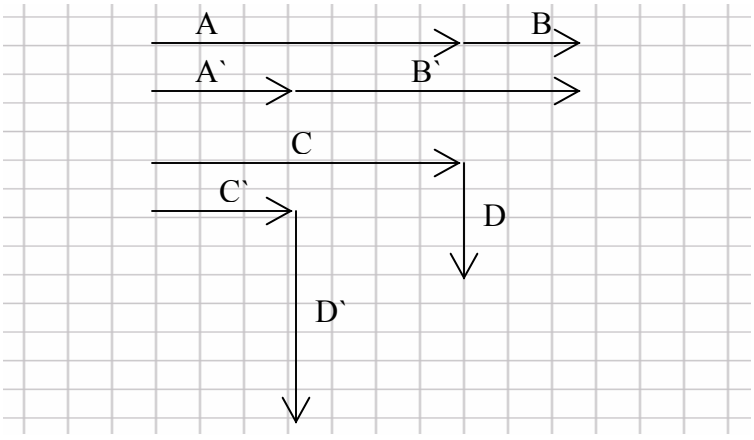

Figure 14 Altering Path to Preserve Randomness

Notice in Figure 14 if the relative lengths of segments A and B are reversed, as shown with segments A` and B`, then the end result is the same – that is, the variations in length have no noticeable effect on the final position because there is only a single path. If the path is altered between segments, as shown with segments C and D and the alternate segments C` and D`, then the final result is different. The variations in length of the individual segments are preserved by the alternate path. It is necessary to vary the path at each sample in order to preserve the randomness at each sample.

In order to affect a new path at each period, the generator must employ a second function independent of the generation function. In order to illustrate this requirement, consider two cases. The first case will involve a simple generating function, +1 and at every sample a dependent function, +3. The dependent function is derived from the independent function by multiplying the independent function by 3. See Figure 15 for a block diagram of the generator with a dependent function added. Another more subtle change is also required to the generator. The number read by the sampler must be re-introduced to the generator as a counter preload. Randomness is captured by feeding it back into the system so the sampled value is loaded into the counter each time it is read.

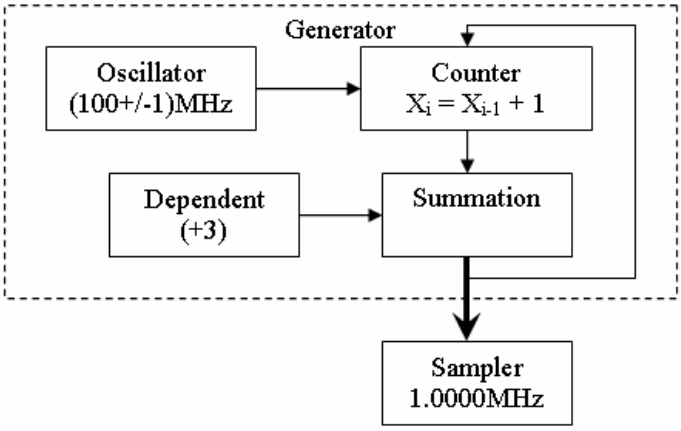

Figure 15 Generator With Dependent Function

This block diagram is represented by Equation 8 below where R_0 is the first random number, R_1 is the second, R_2 the third, etc. T is the number of sample periods, ΔR_{AVG} is the average value of the change in random number (the average number of clocks) per sample period and U is the uncertainty at each sample. The equation says each random number is the product of the average number of clocks per sample times the number of samples plus the uncertainty. The uncertainty term, U, is outside the summation indicating that the uncertainty is not captured at each sample period.

Equation 8 Uncertainty From a Single Path

$$R_0 + R_1 + R_2 + R_3 \ldots = (T \bullet (\Delta R_{AVG} + 3)) + U$$

$$R = (\sum_{1}^{T} (100 + 3)) + U$$

$$where U \in [-1,0,+1]$$

The second case will involve the same simple generating function, +1, but paired with the function x10 as shown in the block diagram in Figure 16. The function x10 is independent of the generating function +1. Independent here is used in the algebraic sense: that is, there is no correlation between $\{X_2 = X_1 + 1\}$ and $\{X_2 = X_1 \text{ x } 10\}$.

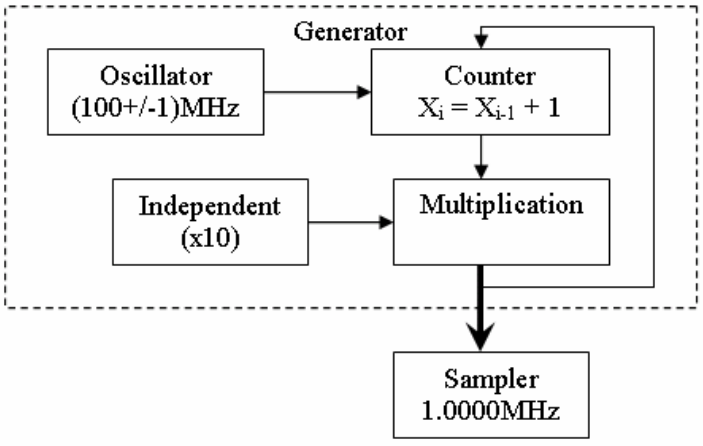

Figure 16 Generator with Independend Function

Equation 9 describes the generator in Figure 16. U_T is the uncertainty at sample T. Unlike the generator with a dependent second function, the exact value of U at the sample instant is required to compute the next random number.

Equation 9 Uncertainty From Multiple Path

$$R = \sum_{1}^{T}((100 + U_T) * 10)$$
$$where U_T \in [-1,0,+1]$$

In Equation 9, the uncertainty cannot be factored out of the summation hence the randomness is collected from each sample period. The second function in the second case, times 10, is independent of the first function, +1. The uncertainty cannot be factored out because the final value generated depends on what the uncertainty was at the sample instant as well as how long it has been since the last sample. Hence randomness is captured from each sample. Table 2 shows two iterations of output from the generator and the result of reversing the order of the oscillator fluctuations. Once the independent second function is added, the random number is different based on the order of the oscillator fluctuations. Hence the randomness has been captured by the generator.

Table 2 RN Generated Versus Oscillator Frequency

Osc Freq	Counter	RN		Osc Freq	Counter	RN
101	101	1010		99	99	990
99	1109	11090		101	1091	10910

The independent function in the generator alters the path of the generating function by moving the generated value to a new point in the range. Since the uncertainty is present at each period, it is the uncertainty that is being preserved by relocating the value at each period. Since the value is relocated, a discontinuity is introduced at each period.

The path of the generation function is forced to diverge from its normal trajectory through the range of generated values. A simple example will illustrate this process.

4.2.6 Simple Divergent Path RNG

Assume a 4 bit random number, implying a range of [0 .. 15], a generating function of +7, a secondary function of x 2, an oscillator for the generation circuit running at 10 Hz average with +/- 10% noise and a sample frequency of 1 Hz. At this point it is necessary to make a few modifications for the sake of the actual implementation. First, since there are 4 bits holding the value, math will be done Modulo 16 as shown in Equation 10.

Equation 10 Modulo 16

$$(15+1)MOD16 = 0$$

Second, a straight multiply by 2 is problematic for two reasons; (a) it tends to push digits beyond the 4 bit limit and (b) it removes randomness from bit0. Instead the multiply function will be replaced by a rotate left function. This will preserve all the generated bits and has no tendency to push bits beyond the 4 bit limit. With these two concessions to the requirements of digital synthesis, the state machine for the example RNG is shown in Figure 17.

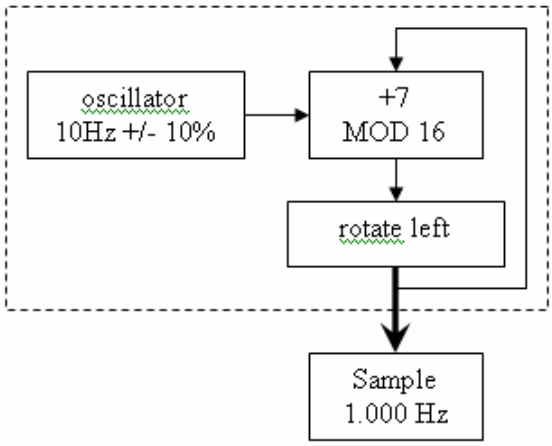

Figure 17

Assuming the reset value is 0, Table 3 shows the first two iterations of this RNG.

Table 3

Iteration	(+7 x {9,10,11}) MOD 16	Rotate Left
0	0	0
1	(15, 6, 13)	(15, 12, 11)
2	{(6,12,2),(11,14,1),(2,4,6)}	{(12,9,4),(7,13,2),(4,8,12)}

The middle column is calculated by multiplying +7 times the number of oscillations in the sample period (9, 10 or 11) then taking MOD 16 of the resulting number. See Figure 18 for examples of this calculation.

$$(9x7)MOD16 = 15$$
$$(10x7)MOD16 = 6$$
$$(11x7)MOD16 = 13$$

Figure 18 Example Calculations

The right column is just a Left Shift of the middle column with bit3 rotated into bit0. See Figure 19 for examples of this calculation.

$$15 = \%1111 RotateLeft = \%1111 = 15$$
$$6 = \%0110 RotateLeft = \%1100 = 12$$
$$13 = \%1101 RotateLeft = \%1011 = 11$$

Figure 19 Example Rotate Left Calculations

Note that if there were no uncertainty, only one value of each three-tuple would be generated. Since the oscillator jitter is assumed to be symmetrically distributed about the average value, the middle value of each three-tuple would be generated in the absence of jitter. The produced sequence would be [0, 12, 13]. Figure 20 illustrates the divergent paths generated by this RNG.

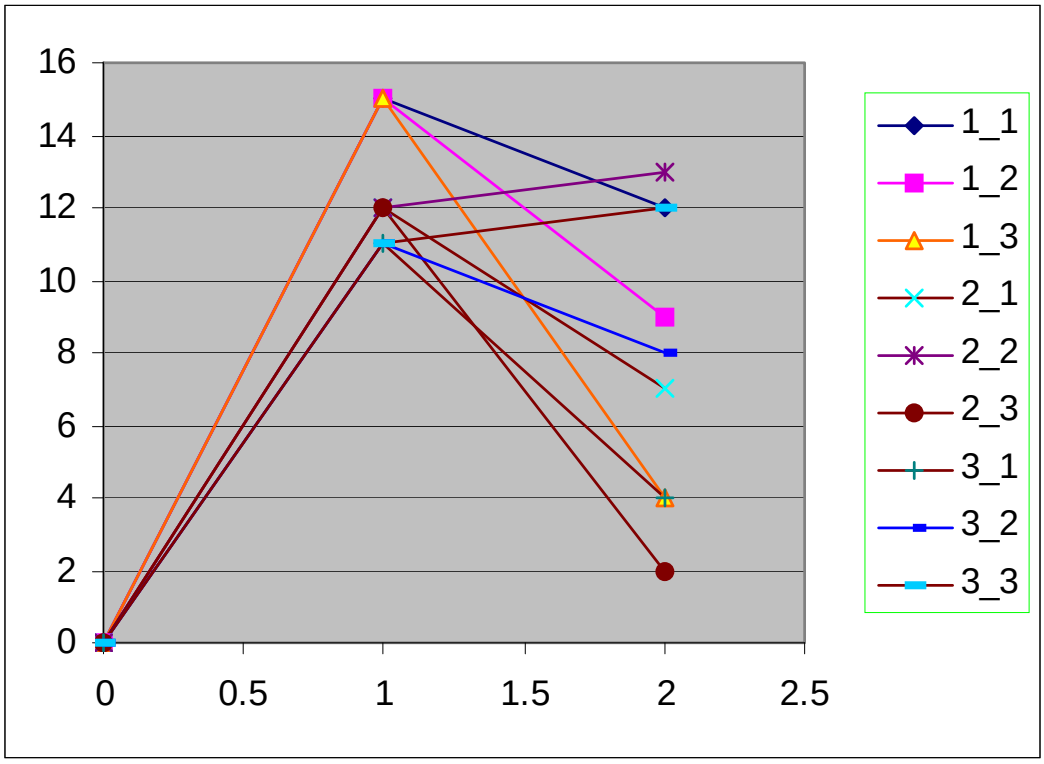

Figure 20

The paths diverge from each point at the rate of 3 paths per point because the clock has an uncertainty of +/- 1 counts per period. Hence there are three possible values for each generated number. The independent function ROTATE-LEFT preserves the randomness from each path.

4.2.7 Divergent Path Formulae

As shown in Figure 20, there are three possible paths at each point of divergence. After the first iteration there are 3 possible values. After the second iteration there are 9 possible values. Should a third iteration be performed there would be 27 different values calculated. At each iteration the number of possible values becomes multiplied by the divergence from each point. By inspection the total number of possible values after some arbitrary number of iterations is given by Equation 11.

Equation 11 Number of Possible Values

$$N = D^i$$

where N = number of values, D = divergence (number of paths from each point) and i = iterations.

Another observation may be made about this simple generator. By the third iteration, the number of possible values that can be calculated is $3^3 = 27$. The range of values for a 4 bit generator is only $2^4 = 16$. So the number of values that can be calculated exceeds the range of the generator. That is, any generated value must be in the range [0 .. 15], a total of 16 possible values. But the total number of calculations which must be performed to guarantee that the actual generated number has been calculated on the third iteration away is 27 according to Equation 11. At this point, calculating the expected output of the generator becomes futile as there are more calculations required than simply listing the possible values. Hence by Chaitin's criterion for randomness, that no simpler representation for the set exists other than listing the set, this generator will mimic truly random behavior on the third iteration of values [28]. The number of iterations necessary to mimic true random number generator behavior can be derived from Equation 11 by setting it equal to the range of possible values, 2^N, as shown in Equation 12. That is, after how many iterations does the number of possible values equal the size of the range?

Equation 12 Iterations When RNG Appears Random

$$2^N = D^i$$
$$N \log(2) = i \log(D)$$
$$i = \frac{N \log(2)}{\log(D)}$$

where N = number of bits (4) and D = divergence (3) at each point. The term on the top left, 2^N, represents the range of values – or for Chaitin's criterion the size of a list of all the possible values. The term on the right, D^i, represents the number of values which must be calculated to guarantee prediction of the random number generated with a divergence D on the i^{th} iteration. For the simple RNG example with D = 3and N = 4, this equation yields i = 2.5 in agreement with the empirical measurement of 3 (see the plot in Figure 20). Hence for a divergence of 3, prediction of more than two samples into the future is futile as the list of calculated values is longer than the list of possible values.

4.2.8 Differences From Other Architectures

The Multi-Clock Generator presented by McTaggart and Burson[24] is an example of the simplest form of a divergent path RNG. The two LFSRs, clocked by different clocks, represent two functions. The two functions are independent since they are each maximal length and the numbers of bits of the two LFSRs are relatively prime. The divergence for this particular generator would be 2 as there are two independent operations.

The RNG presented by Oerlemans[25] is quite different from a divergent path RNG. Oerlemans has presented a single LFSR clocked by an unpredictable clock so that the stream of numbers is not predictable. But the randomness introduced by the clock uncertainty is not preserved – each sample is only as random as the clock frequency which is derived from the XOR of the oscillators. The average and standard deviation of the clock frequency of this generator could be calculated by observing enough generated numbers. Then the value at any time could be predicted to fall within a subset of the range of numbers. Hence there is not necessarily any point in the future at which

prediction of the output value becomes futile. The RNG presented by Oerlemans is not guaranteed to ever meet Chaitin's criterion for randomness. A divergent path generator will capture the randomness from each sample. The farther into the future prediction is attempted, the more randomness the generator has captured. As shown in Equation 12 above, there is a point at which prediction of the output from a divergent path generator becomes futile. Hence a divergent path RNG will always meet Chaitin's criterion for randomness at some future sample. The number of samples at which the RNG becomes random by Chaitin's definition is governed by the number of independent functions comprising the RNG. More independent functions means less samples before the RNG achieves randomness.

CHAPTER 5 DIVERGENT PATH ARCHITECTURES

5.1 Adder-Shifter Based TRNG Architecture

In chapter 4 a simple example for a TRNG was presented. A free-running adder was used to generate bits and a shifter was used at sample intervals to capture and preserve randomness. This type of TRNG can be abstracted to a general architecture consisting of an adder for bit generation followed by a shifter for randomness preservation. A TRNG that has an adder-shifter architecture will be referred to as an ASTRNG in the remainder of this paper. The ASTRNG illustrated in Figure 21 has two additional features: (1) a transposing unit attached after the shifter and (2) the adder is split into two counters.

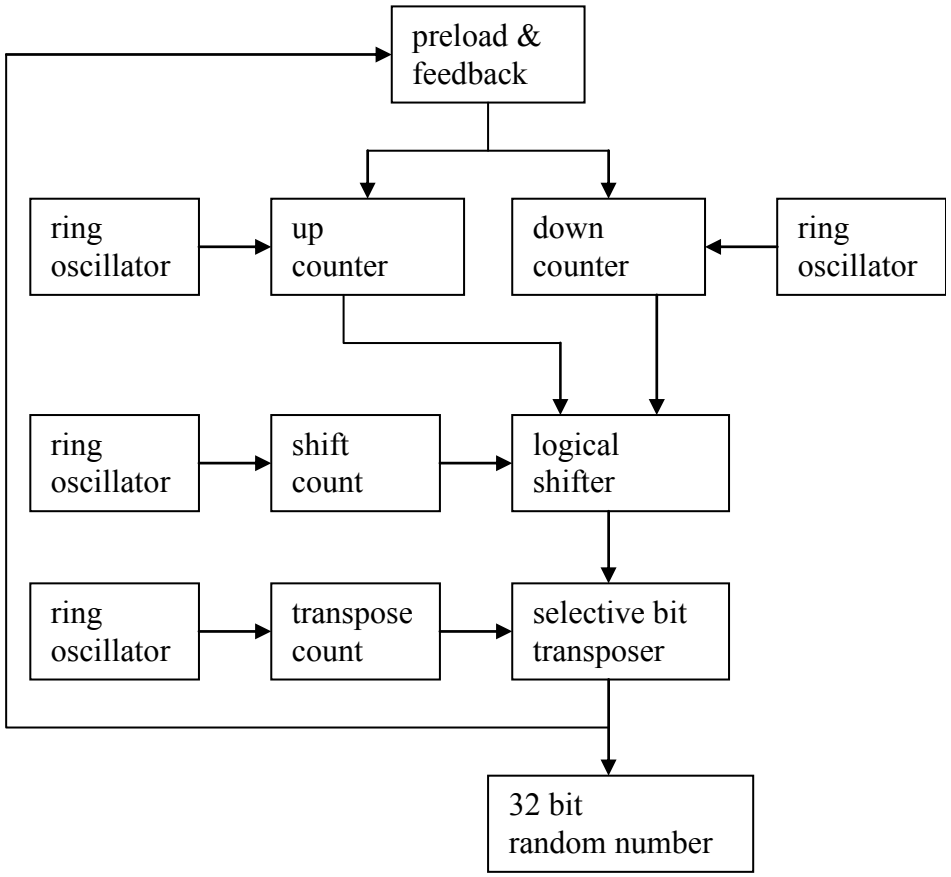

Figure 21 Block Diagram of ASTRNG

The bit generation for this ASTRNG has been configured as a 16 bit up counter and a 16 bit down counter for two reasons. First, splitting the counter reduced the time required to produce every possible value from 2^{32} clocks to 2^{16} clocks. Second, it helps balance the number of high bits (1's) with the number of low bits (0's). Each counter has its own noisy ring oscillator. Each counter runs all the time. Each counter has an associated latch that is not shown in the block diagram in order to avoid confusion. All four counters are latched when a value is read from the ASTRNG. The counters continue to run and the latched counts are used for creating the value read. This architecture avoids synchronizing the ring oscillators at every read. Whenever a random number is read, the random number is fed back into the counters using the preload function. Following the bit generation stage is the shift stage for preserving randomness. This stage is a 32 bit barrel shifter. The shift count is not fixed; rather a free running 5 bit counter is clocked by a third noisy ring oscillator as shown. Whenever a value is read from the ASTRNG, the shift count is latched and the value from the bit generator is rotated by that count. As discussed earlier, bits are rotated left and the leftmost bit (b31) is rotated back into the rightmost bit (b0) position. The final stage in this ASTRNG transposes the individual bits in the shifted value. There are four transpositions implemented. A free running 2 bit counter, clocked by a fourth noisy ring oscillator, is latched whenever a value is read from the ASTRNG. The latched 2 bit count determines which transposition is applied to the value. Table 4 illustrates how the bits are transformed according to the transpose count. The four different transpositions implemented are: (1) 32 bit, (2) 16 bit, (3) 8 bit and (4) 4 bit.

Table 4 Transposer Bit Transpositions Per Count

output bit	count=00	count=01	count=10	count=11
0	31	15	7	3
1	30	14	6	2
2	29	13	5	1
3	28	12	4	0
4	27	11	3	7
5	26	10	2	6
6	25	9	1	5
7	24	8	0	4
8	23	7	15	11
9	22	6	14	10
10	21	5	13	9
11	20	4	12	8
12	19	3	11	15
13	18	2	10	14
14	17	1	9	13
15	16	0	8	12
16	15	31	23	19
17	14	30	22	18
18	13	29	21	17
19	12	28	20	16
20	11	27	19	23
21	10	26	18	22
22	9	25	17	21
23	8	24	16	20
24	7	23	31	27
25	6	22	30	26
26	5	21	29	25
27	4	20	28	24
28	3	19	27	31
29	2	18	26	30
30	1	17	25	29
31	0	16	24	28

This ASTRNG was constructed using the Mentor Graphics chip design tools. The resulting design was translated to a GDS-II plot file and that file was sent to MOSIS for fabrication in AMI05 technology. Five (5) of the chips were bonded and packaged in the typical MOSIS 40 pin ceramic DIP package and returned from MOSIS. The five chips

were tested for functional operation and current requirements and the results sent back to MOSIS. Then the five TRNG chips were tested for divergence and shown to be truly random. Finally bit streams were gathered from each of the five TRNG chips and tested using the NIST 800-22 tests for randomness.

5.2 Design of the ASTRNG Chip

5.2.1 ASTRNG Design Methodology

The architecture of this TRNG is shown in Figure 21. The ASTRNG design was begun using VHDL. However, when problems were encountered with the synthesis tool, Leonardo Spectrum, the VHDL design was abandoned in favor of a digital schematic design based on Mentor Graphics Design Architect, abbreviated DA. DA comes with a generic AMI05 parts library. The library contains most commonly used gates, latches and flipflops. This library was used in the ASTRNG design to minimize the amount of time spent designing individual gates and allow concentration on the ASTRNG layout and testing.

Fortunately switching to schematic design did not decrease the ability to test individual blocks as they were completed. AccuSim was used to verify operation of each component as it was designed. As each block layout was completed, a spice extraction was performed and the resultant circuit was simulated using MachTA. While this represents a lot of time spent designing and running tests, such attention to detail at the block design level made the final layout much easier to test and gave a high degree of confidence the resulting IC would work.

5.2.2 ASTRNG Development

This ASTRNG architecture required substantial development effort. First a more detailed block diagram was prepared. The transistor count in each of these blocks was estimated using Leonard Spectrum output statistics with no attempts at optimizing. Then each block was constructed using the following methodology:

a) Draw a schematic using Design Architect

b) Simulate the schematic using Accusim

c) Layout the schematic using IC Station

d) Verify the layout using LVS (Layout Versus Schematic)

e) Check the layout using DRC (Design Rules Check)

f) Perform a parasitic extraction to properly model the fabricated part

g) Verify correct operation using MachTA on the extraction

There were a number of blocks to be designed including those shown in the block diagram as well as some that were required but not shown in the block diagram. The blocks that were designed as a part of this project are tabulated in Table 5.

Table 5 Blocks In ASTRNG

Name	Number	X Size	Y Size
Input Mux	1	200	1100
Up Counter	1	796	1080
Down Counter	1	796	1080
Logical Shifter	1	1000	2200
Transposer	1	600	2200
Ouput Mux	1	160	2000
Latch16	1	193	1050
Control Register	1	992	360
Ring Oscillator	4	500	600
5 Bit Counter	1	631	360
2 Bit Counter	1	320	240
Counter Latch	2	322	970

A down counter bit slice and two versions of an up counter bit slice (one with a preload and one without) were designed using AMI05 standard cells. Sixteen up counter bit slices with preload were combined in the up counter while sixteen down counter slices were combined in the down counter. A small amount of logic was added to each counter to permit synchronous preload.

The logical barrel shifter was prepared in a different manner. Since an instantaneous multiple bit shift was required, a block of five multiplexers implemented each bit. Then a five bit count selected which of the multiplexer inputs were selected. The five bit counter was built from five of the non-preload up counter bit slices mentioned above. The logical shifter posed the greatest routing challenge of any module because of the large number of interconnects.

The transposer was designed as a 32 bit wide four to one multiplexer. As such it could be broken down into bit slices of a 4 to 1 multiplexer. Then thirty-two of the bit slices were combined along with some control logic to create the transposer.

As shown in the block diagram, there were four ring oscillators required for this design. Each ring oscillator had a different number of stages. Each ring oscillator is controllable to some extent so the frequency of oscillation can be altered by switching inverter stages in and out of the design. The fewer the stages, the faster the ring oscillates. More stages cause the ring to oscillate slower. The rings were designed to permit frequency shifting among four different oscillation frequencies. In order to permit changes to the topology during operation, the changes had to be synchronized to the operation of the ring. Digital control of the ring oscillators including frequency shifting is covered in more detail in this paper[29].

There are two more entities required for this ASTRNG in addition to those in the block diagram. A control register is required to allow stopping, starting and configuring the ASTRNG and latches are required for holding data. The control register is formed from D flipflops and has read and write capability. It contains three bits per ring oscillator; one to select between fixed or rotated frequencies and two to select which frequency. The control register also contains a run bit; when set the ASTRNG generates numbers and when cleared it does not. Finally the control register contains an Inhibit Feedback bit; when set the current output of the ASTRNG is not preloaded into the counters and cleared the current output is preloaded to seed generation of the next number.

Two different types of latches were constructed. One type of latch, ctlat, was designed to latch the output of the up and down counters during random number generation. There are two of these, one for the up counter and one for the down counter. The other type of latch, lat16, is just a simple array of latches to latch the most significant half of the 32 bit random number for the 16 bit bus implementation.

With all blocks completed and tested, the entire ASTRNG was simulated using MachTA. Several operations were simulated. Reset and run was tested to be sure the random number generator would come out of reset and produce random numbers with no intervention from control logic. Reading from and writing to control registers was also simulated. Finally several random numbers were simulated.

A padring was designed from AMI05 pads and the ASTRNG was placed in the padring. The simulation was adapted to run through the padring. A parasitic extraction was performed on the complete chip, ASTRNG plus padring. Finally MachTA was used

to verify functionality of the completed chip. Then the completed chip was converted to GDS-II format and transferred electronically to MOSIS for production and packaging.

When the decision was made to have ASTRNG chips fabricated, it was necessary to select a package and design the pinout. The standard MOSIS AMI05 package is the 40 pin DIP which limits the design somewhat. Once the required pins had been listed, there were not enough pins to build a 32 bit bus. Eight pins, 4 inputs and 4 outputs, were needed for bringing the ring oscillators out and supplying external oscillators in case the internal oscillators were not adequate. Therefore it was necessary to build a 16 bit bus on the IC itself and a latch inside the ASTRNG so that all 32 bits could be read from the ASTRNG at once – the lower 16 bits are presented out the bus while the upper 16 bits are held in the latch until a second read could be performed to collect them. The pins and their functions are listed in Table 6.

Table 6 Pin Definitions for ASTRNG Chip

Pin Name	Number	Function
RST	6	1 means reset – 0 means run
IRUN	11	1 means generate after reset – 0 means halt after reset
CS	8	1 means ASTRNG selected – 0 means not selected
RnW	7	1 means read – 0 means write
MSW	10	1 means acess upper 16 bits – 0 means lower 16 bits
CReg	9	1 means access control register = 0 means access data
Ack	5	1 means transfer can complete – 0 means hold transfer
D0 .. D15	14-17,19-22, 24-27,28-31	Data lines
OBO	34	1 means use onboard oscillators – means use external
INxx	1,2,3,40	external oscillator inputs (4)
OUTxx	35,36,37,38	external oscillator outputs(4)
Vdd	12,13,18,23	Power supply voltage
Gnd	4,32,33,39	ground connection

Figure 22 shows the IC layout for the ASTRNG. Note this layout required a double allocation from MOSIS as the area of the completed layout was larger than would fit in a single allocation. That is why the top and bottom sides of the layout below look (and are) much longer than the left and right sides. Note also that with poor planning, the top row has many more connections (13) than the bottom row (7). For future reference, it is much easier to connect the pads to pins if there are equal numbers of pads on each side of the layout. Looking from left to right, the input bus latch is left most. Next are the counter preload latches followed closely by the counters and the counter output latches. The barrel shifter is next and the transposer is the rightmost tall block. Then along the top are the fliflops that make up the control register. Just under them is a 16 bit output latch. Then on the right are the four ring oscillators. Just below the ring oscillators are the 5 bit shift counter and the 2 bit transpose counter.

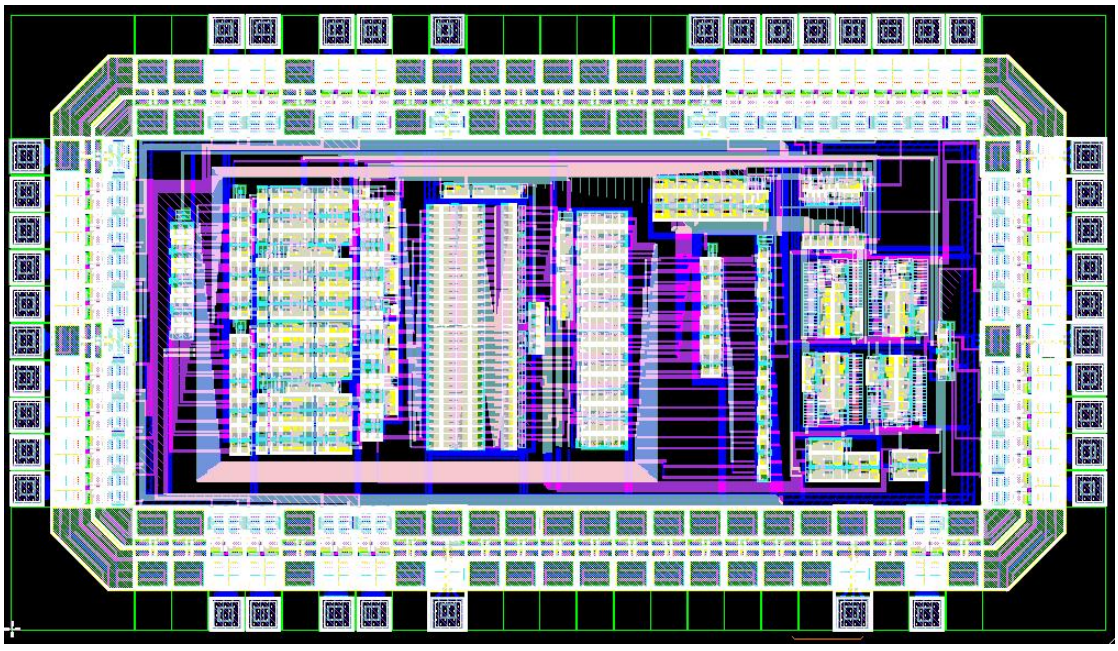

Figure 22 ASTRNG IC Layout

5.2.3 ASTRNG IC Fabrication Testing

Whenever MOSIS fabricates an IC in their education program they request a report on the results of the fabrication to help them improve their process. MOSIS fabricated five of the ASTRNG ICs and bonded them into the 40 pin DIP packages. When the ICs were returned to the University, each was tested for functionality by writing the control register and reading it back, reading at least two random numbers and measuring the power supply current for each chip with reset asserted and while the chip was active. The results of the functionality test and the current measurements are reproduced in Table 7.

Table 7 ASTRNG Functional and Current Results

Chip	Functional Test	Reset mA	Active mA
1	pass	3.5	42.3
2	pass	3.4	42.3
3	pass	3.7	41.9
4	pass	3.8	41.2
5	pass	3.7	41.0

The oscillation frequency for each of the four settings of each of the four ring oscillators was measured on each ASTRNG chip and is illustrated in Figure 23.

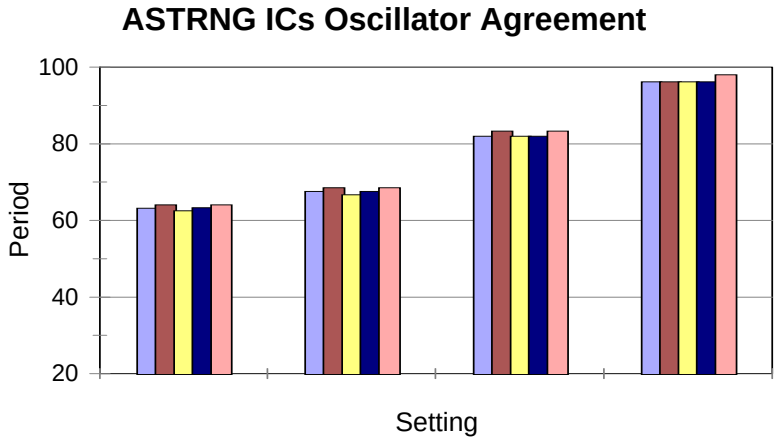

Figure 23 ASTRNG Oscillator Comparison

As seen in Figure 23, the agreement in period between oscillators with similar settings is very high. The results for all settings on all oscillators are in Table 8.

Table 8 RNG Oscillator Frequency

Chip	Setting	Osc1	Osc2	Osc3	Osc4
1	00	64.1	56.2	53.2	49.5
1	01	68.5	64.1	58.8	52.6
1	10	83.3	70.4	69.4	58.8
1	11	98.0	86.2	74.6	68.5
2	00	63.3	55.6	52.1	49.0
2	01	67.6	64.9	58.8	52.1
2	10	82.0	69.4	67.6	58.1
2	11	96.2	86.2	75.8	67.6
3	00	62.5	55.6	52.1	48.5
3	01	66.7	63.3	58.8	51.5
3	10	82.0	69.4	67.6	58.8
3	11	96.2	84.7	74.6	67.6
4	00	64.1	54.9	52.1	50.0
4	01	68.5	64.9	58.8	52.1
4	10	83.3	69.4	68.5	58.1
4	11	96.2	84.7	73.6	67.6
5	00	63.2	56.2	52.1	50.0
5	01	67.6	64.9	58.1	52.6
5	10	82.0	70.4	67.6	58.1
5	11	96.2	84.7	74.6	68.5

5.2.4 ASTRNG IC Statistical Testing

The ASTRNG IC was tested statistically using the NIST800-22 test suite. First the raw output of each of the five ASTRNGs was gathered at 20 microsecond intervals. The data was gathered from one IC at a time. The data was read at a rate of 32 bits every 20 microseconds or 1.6Mbits/second. Then the raw output was run through the NIST800-22 test suite. Since the data did not pass all of the NIST800-22 tests, it was whitened in software by reading the raw TRNG data file, XORing it with the output of a PRNG and writing a new data file containing the whitened data. The PRNG used to whiten the data

contained a 32 bit LFSR that was clocked exactly 32 times between samples. Clocking the LFSR 32 times does not reduce the maximal length property since 2^{32} and $(2^{32} - 1)$ are relatively prime but it does insure that all bits in the LFSR change every sample instant. The LFSR was derived from the Xilinx white paper on maximal length LFSRs[7]. The taps for the XOR were at bits 31, 21, 1 and 0 though they are numbered 32, 22, 2 and 1 in the Xilinx paper. The results from the raw data and the whitened data are presented in chapter 7 section 7.1.

5.2.5 ASTRNG Realization In FPGA

The same random number generator was described in VHDL and synthesized in an FPGA in order to demonstrate the ease with which the design can be translated to any digital platform. The design was synthesized to a Xilinx Spartan2 FPGA (XC2S100) in an Avnet Mini-Spartan2 Development Board. In order to connect it to the same test bed as the RNG ICs, an additional wrapper that duplicated the functionality of the ASTRNG IC test board was created. Then a small PC board with two 40 pin connectors was wire wrapped. This small board translated the pinout from the Spartan2 Mini Development Board to the ASTRNG IC test board so that the same software and hardware could be used to test the ASTRNG IC and any digital ASTRNG implementation on the Spartan2 Mini Development Board. As with the IC version, the output from the FPGA version was whitened using the software PRNG previously described. Test results for the ASTRNG in FPGA are discussed in chapter 7, section 7.2. Similar results were achieved between the ASTRNG IC and the ASTRNG FPGA, meeting the requirement that the design be realizable on any digital platform.

5.3 Concatenated LFSR Based Architectures

As mentioned earlier, the purpose of the transposer in the ASTRNG was just to scramble the order of bits. Since a LFSR acts like a counter with a pseudo random count it was decided to construct a digital TRNG from concatenated LFSRs instead of from the adder-shifter architecture. For these CLTRNGs, the divergent path would come from preloading the LFSR with different values depending on the relative rates of the oscillators. The CLTRNG designs were easy to build using the Xilinx development board. Each one was described in VHDL, synthesized for the Xilinx Spartan2 development board and tested using the same test setup as the ASTRNGs. Each of the LFSRs used in CLTRNGs was based on the Xilinx application note previously cited.

A LFSR is just a bit trickier to work with than a straight counter. The reason is that for either the XOR or the XNOR implementation there is a lockup value. Care must be exercised to keep from preloading the LFSR with the lockup value. For these implementations, instead of risking lockup, each LFSR was preloaded with its own output bits. In order to implement a divergent path, the bits are scrambled before preloading.

5.3.1 CLTRNG Realization With 9, 13 and 16 Bit LFSRs

Figure 24 is a block diagram for a CLTRNG. This Concatenated LFSR TRNG was built from three LFSRs; one 16 bit LFSR, one 13 bit LFSR and one 9 bit LFSR. Combined there would be 38 bits (16 + 13 + 9) so the upper two bits of each LFSR are not concatenated to form the random number. Instead the upper two bits of each LFSR are tied to the ring oscillators in the other two LFSRs to force frequency rotation in the ring oscillators. The oscillators are all cross coupled: that is, each LFSR has two

frequency control outputs. Each ring oscillator has two frequency control inputs. Each LFSR's frequency control outputs are tied to the other two LFSRs' frequency control inputs – one output to each of the other two inputs. This design was described in VHDL.

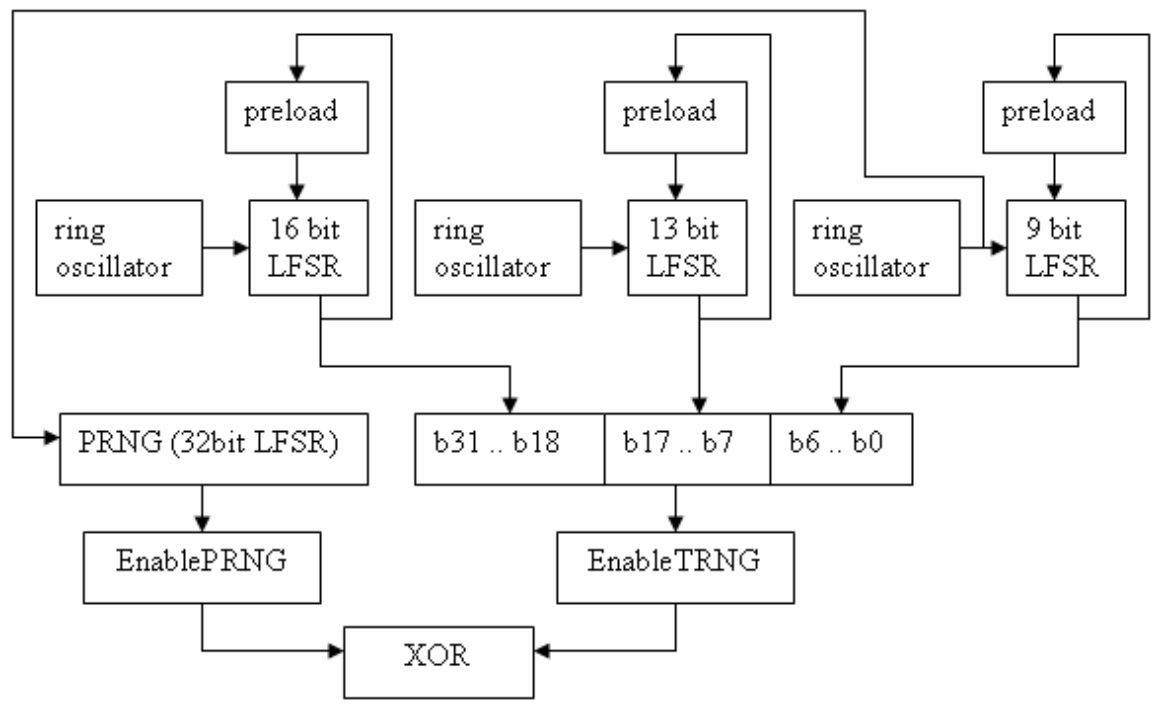

Figure 24 CLTRNG Bock Diagram

A 32 bit whitening PRNG constructed from a 32 bit LFSR was included in the design. The PRNG contains a 32 bit LFSR and a state machine that clocks the LFSR exactly 32 times between samples – updating all bits in the PRNG every sample. The control register was adapted so that either the 32 bit CLTRNG or the 32 bit PRNG or the 32 bit XOR of the CLTRNG and PRNG could be selected for output. The output of the PRNG was of interest to insure proper operation. Tests were made with the CLTRNG only enabled, with the PRNG only enabled and with the whitened CLTRNG. Test results are given in chapter 7, section 7.3.

5.3.2 LFSR Realization With 11, 11 and 10 Bit LFSR Contributions

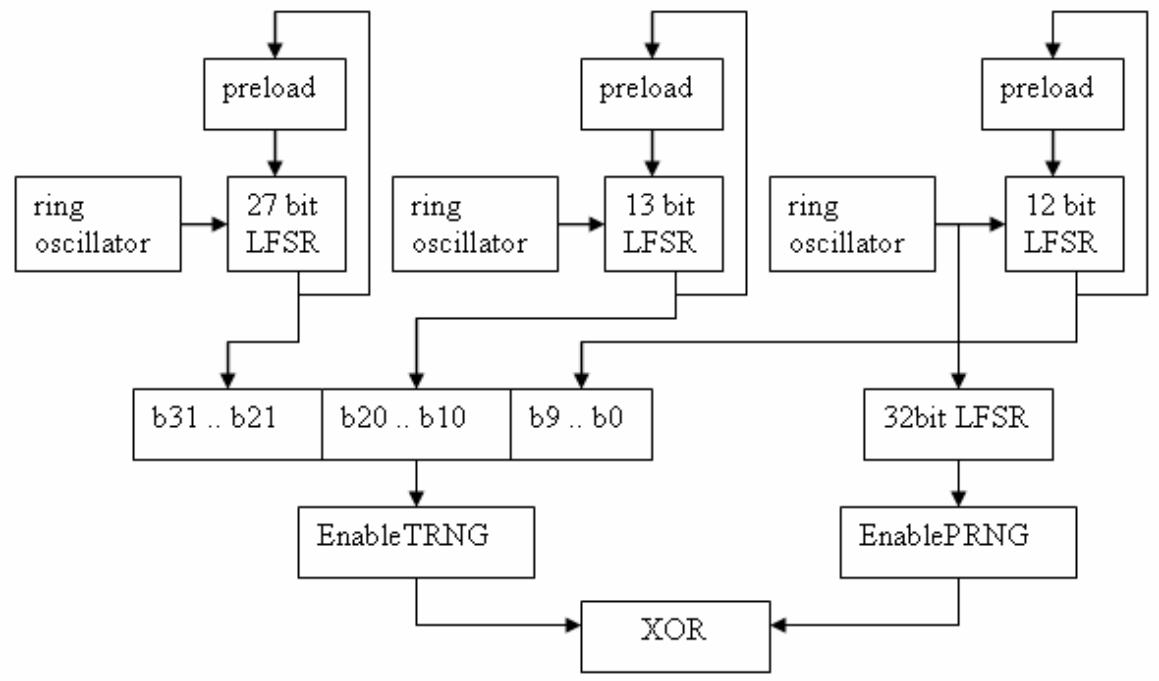

Figure 25 CLTRNG With 11, 11 and 10 Bit Contributions

This CLTRNG was also built from three LFSRs; one 27 bit LFSR, one 13 bit LFSR and one 12 bit LFSR. Figure 25 is a block diagram of this CLTRNG. The previous design was altered somewhat to explore whether a different LFSR would make a significant difference in the statistical scoring of the RNG. From the previous RNG, the 9 bit LFSR was expanded to a 10 bit LFSR. The 13 bit LFSR was not altered and the 16 bit LFSR was expanded to a 27 bit LFSR. For all LFSRs, maximal length circuits as specified by the Xilinx application note were implemented. The 32 bit whitening PRNG from the previous design was retained. The oscillator cross coupling from the previous design was also retained. The design was synthesized for the Spartan2 development board. Data was gathered with and without whitening and statistical tests were run on the data. Chapter 7 section 7.4 documents test results from this CLTRNG. The PRNG data is the same as that documented in section 7.3 so it is not repeated.

5.3.3 CLTRNG Realization With 7, 9, 11 and 13 Bit LFSR Contributions

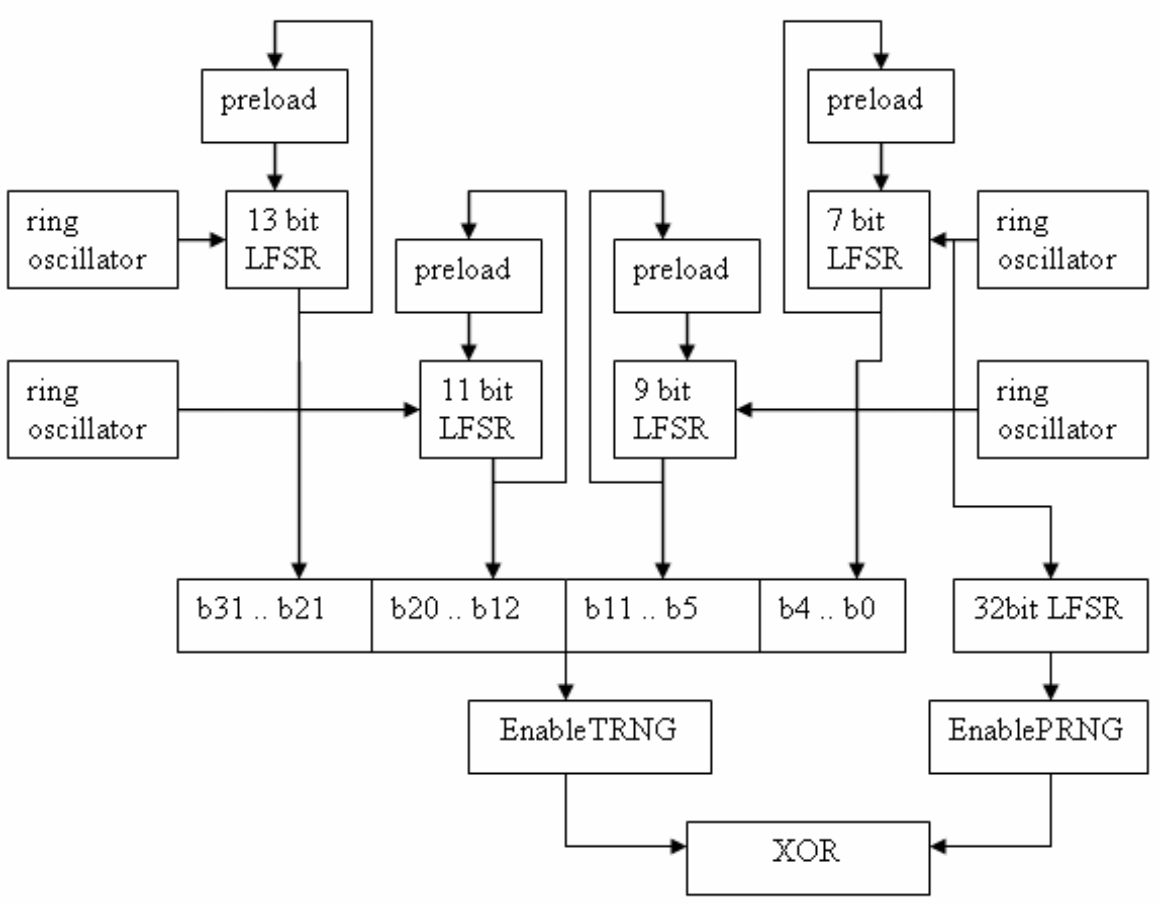

Figure 26 CLTRNG With 4 LFSR

This CLTRNG realization was built from four concatenated LFSRs instead of three to explore whether increasing the number of LFSRs made a significant difference in the statistical scoring of the RNG. Since there are now four oscillators instead of three, there are 33% (4/3) more paths from any one point. Hence it is expected that the divergence would be higher for this CLTRNG than for the three LFSR version of the CLTRNG. A block diagram for this CLTRNG is shown in Figure 26. In this case a total of 40 bits are produced, 32 of which are the random number. The other 8 bits are used to control the 4 ring oscillators. The oscillators are cross coupled as before; that is, no frequency control output is tied to the oscillator which drives its generating LFSR. Both non-whitened and

whitened data were gathered as before. Statistical tests were run on this data. The results are documented in chapter 7 section 7.5. Once again the PRNG results are not reported for this generator as they are identical to those documented in chapter 7, section 7.3.

5.3.4 Using a TRNG to Whiten a PRNG

One of the more interesting possibilities to come out of this work is the ability to whiten an LFSR based PRNG. The LFSR based PRNG is simple to build in hardware and software as documented earlier. But there is a frequency component present in maximal length LFSRs due to the repetition of certain bit patterns as the LFSR proceeds through the range of output values. This frequency component can be whitened by XORing the PRNG output with a divergent path TRNG built of LFSRs clocked by ring oscillators – in other words a CLTRNG. Experimental data supporting this conclusion are presented in chapter 7 section 7.6.

CHAPTER 6 TEST EQUIPMENT

6.1 TRNG Test Equipment

In order to test the random number generators, a hardware and software platform was put together. The test equipment hardware is based on a microprocessor development board. It contained a Motorola 68306 processor running at 16MHz, 2 Mbytes of FlashPROM, 2 Mbytes of static ram, a RS232 port and a 16 bit bus. Figure 27 shows a block diagram of the hardware used to test RNGs.

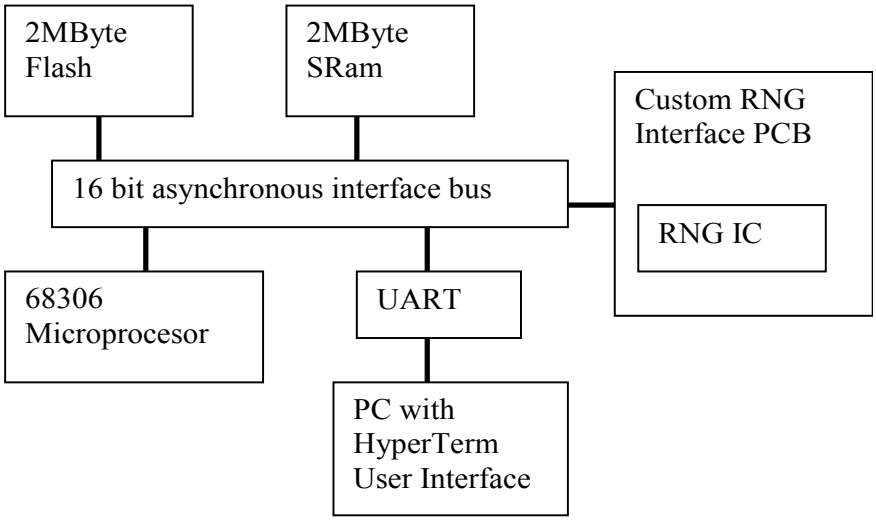

Figure 27 RNG IC Test Equipment

Crosscode C for the 68000 was used to compile and assemble the sotware for the test equipment. Time critical routines were written in assembly language while other software was written in C. The software provided a simple monitor, with peek and poke capability, interrupt driven communications and a custom command set for testing various aspects of the RNG. Table 9 lists the software modules and their contents.

Table 9 Testboard Software Files and Contents

File	Contents
start.s	powerup reset and speed dependent routines
duart.s	interrupt driver for duart
zos.c	basic operating system (malloc free printf scanc etc.)
fips.c	routines to implement FIPS140-1 & 140-2 tests
trng.c	custom routines for testing RNGs

Figure 28 shows a photograph of the test equipment. The larger board on the left is the MC68306 microprocessor board. The MC68306 is in the top left corner. Below it are two flash proms. This board was designed to run in as an EISA peripheral but the EISA portion is unused. The large connector on the left is for power. The empty socket where the ribbon connector is attached was originally for a coprocessor so all address and data lines were available. The 40 pin ribbon connector attaches to one of several interface boards as detailed later in this chapter.

Figure 28 RNG IC Test Equipment

6.2 RNG Test Equipment Built In Tests

The tests that are built into the test equipment follow a certain format. Each is represented by a command which is of the form DoXyz where Xyz represent the name or an abbreviation of the function or functions to be performed. The tests were originated in response to FIPS140-1 and so several numbers are specified as follows. A buffer is defined to be 625 each of 32 bit words or 2500 bytes. The built in tests have access to 100 of these buffers meaning a total of 62,500 each of 32 bit random numbers (or 250,000 bytes) may be generated at once and buffered for testing. The FIPS140 specification calls out failure rates in terms of so many per 10,000. So the normal amount of data gathered for transfer to the PC host is 10,000 buffers of 625 each of 32 bit words (25,000,000 bytes). The built in test routines are enumerated in the following paragraphs.

The DoAlgo test is a routine that checks to insure the FIPS140-1 and/or 140-2 algorithms are performing correctly. First a buffer full of pseudo random data is generated from either a LFSR or the rand() function built into the C compiler. Pseudo random data is used in order to verify FIPS algorithm – as pseudo random data should pass these tests. FIPS specifies a buffer to be 625 each of 32 bit words; that is, 2500 bytes. Then the data is evaluated based on either FIPS140-1 or 140-2 standards. The operator specifies on the command line whether to generate with the LFSR or the rand() function and whether to use FIPS140-1 or 140-2 testing.

The next routine, DoDump, merely dumps one or more of the 100 buffers to the screen. Again, the operator specifies which buffer to dump on the command line.

The third routine, DoFIPS, runs the FIPS test for statistical randomness on the data buffer or buffers specified. The operator chooses which buffer(s) are to be evaluated and whether 140-1 or 140-2 specifications will be applied to the data. At the conclusion of the test, the number of buffers which fail one or more aspects of the test are output by test category.

The fourth routine, DoCLat, writes a value into the control latch on the test board. The control latch, as the name suggests, latches the control signals for the RNG IC. The three signals latched are tabulated in Table 10.

Table 10 Bits in Control Latch

Bit	Meaning
0	1=RUN; 0=RESET
1	1=RUN after RESET;0=HALT after RESET
2	1=Use Built-In Oscillators;0=Use Offboard Oscillators

The fifth routine, DoRand, reads a random number from either the normal RNG IC interface board or the special interface board that permitted all five RNG ICs to be read at one time. The user specifies which board to read on the command line. The random number which is read is shown on the screen.

The sixth routine, DoOsc, writes the control latch bit that selects whether to use onboard oscillators (built into the RNG IC) or external oscillators in the random number generating circuit.

The seventh routine, DoCReg, reads and writes the control register onboard the RNG IC. Bits in the control register control whether the RNG is generating or stopped and control the oscillation of each of the four onboard ring oscillators. For each oscillator, a

bit controls whether the oscillator runs at a fixed frequency or rotates the frequency while the other two bits specify which frequency if a fixed frequency is selected.

The eighth routine, DoLoad, loads one or more buffers with random numbers. The options which can be specified on the command line are whether to read numbers from the single RNG IC interface board or from the five RNG IC interface board and which of the 100 buffers should be filled. The time between each random number read is controlled by the global variable nBetween which defaults to 20 microseconds. Another command exists to change the sample period.

The ninth routine, DoPeri, allows the user to set the sample period between random number generations. The new value for the global variable nBetween is the command parameter.

The tenth routine, DoCont, continuously loads buffers full of random numbers and tests them for statistical randomness. The command parameters accepted for this routine control whether the RNG IC is attached via a single IC interface board or the multi IC interface board, whether to use FIPS140-1 or 140-2 specifications and whether or not to whiten the data by XORing it with a 32 bit LFSR output before testing. The sample period is set by the global variable nBetween. This routine will fill all buffers with data then test all buffers, repeating until it is stopped by the user entering a key. When it is finished, a final tally of the results of each phase of the FIPS test will be displayed.

The eleventh routine, DoReset, will reset the RNG IC if the parameter is 0. It will permit the RNG IC to run if the parameter is 1. If the user does not type a parameter then it will show the value of the current reset bit.

The twelfth routine, DoGath, gathers numbers from the random number generator then dumps them to the screen continuously. This is the mechanism used to gather random numbers for processing on the PC. The user can specify on the command line whether to gather the numbers from a multi or single RNG IC board and how many 625 word buffers to gather (minimum of 1, maximum of 20,000). This routine alternately calls Load and Dump to load the buffers then dump them to the screen.

The thirteenth routine, DoRR, resets the RNG then reads random numbers from it. This routine evolved to prove the divergent nature of this series of TRNGs. It resets the RNG then waits a precise amount of time. Then the routine loops reading a RNG and waiting a precise amount of time until the user presses a key. If the routine is reading from the Multi board then it reads four consecutive sets of random numbers and displays them. All signals including reset and read are presented to each of the five boards at exactly the same time. The data from each of the five boards is read into a buffer at exactly the same time. Then thedata from each buffer is read into the test routine in sequence. This test gives a very accurate picture of the divergent nature of these RNGs. If the routine is reading random numbers from a single RNG IC, then it reads 20 numbers, resets the board, reads 20 numbers, resets the board and continues until it has repeated this procedure eight times. It then displays the eight columns of numbers side by side and resumes operation. This routine also places a start beside any columns that have data duplicated with another column.

The fourteenth and fifteenth routines, DoTC and DoPound, were used to test the counters associated with a RNG and to to continually write a location. These two routines have no relevance to random number testing.

6.3 RNG Test Interface PCBs

Two printed circuit boards were designed to interface the RNG ICs to the test equipment. One interface PCB was designed to test a single RNG IC exhaustively. The second interface PCB was designed to operate all five RNG ICs in parallel. Both boards were designed using OrCad Schematic Capture to generated the schematic and OrCad Layout Plus to lay out the printed circuit boards. In order to reduce cross talk and power supply noise, both boards were fabricated as four layer boards. Each board has a top and bottom signal plane and a power plane and a ground plane on the inner layers. Once the layout was complete, the board stack was sent to Sierra Proto Express for board fabrication. Two of each board was ordered; one spare and one to populate. The boards were fabricated and populated. The RNG ICs were placed in sockets for easy installation and removal. One small modification from the original schematic was necessary on the single RNG IC interface. No modifications were necessary on the multi-board.

6.3.1 Single RNG IC Interface PCB

The single RNG IC interface board has the ability to measure power supply current by having a jumper on the +5V supply line. If the shorting block is removed and a current meter is plugged into the jumper then the current from the +5V supply can be read. The single RNG IC interface board has all four of the onboard oscillator outputs attached to headers so that an oscilloscope or other measuring device can be attached to each oscillator output. There are also headers so that four external oscillators can be tied to the RNG IC in case the onboard oscillators are not functional. The final diagnostic tool on the single RNG IC interface board is a circuit to capture any accesses that are not properly terminated. Figure 29 is a picture of the single RNG IC interface.

Figure 29 Single RNG IC Interface

6.3.2 Multiple RNG IC Interface PCB

The multiple RNG IC interface board (multi-board) holds 5 each of RNG ICs. The multi-board can only perform limited operations on the RNG ICs. It can reset them all simultaneously and it can read them all simultaneously. It cannot write the control register nor can it operate the RNG ICs in any manner other than free running out of reset. The multi-board has special circuitry so that the reset, chip select and most significant word lines are presented to all 5 RNG ICs at the same time. This arrangement causes all 5 RNG ICs to generate random numbers at the same time. The multi-board also has latches for each RNG IC output so that the random number generated by each RNG IC is latched at the same time. This board was designed to test whether or not the RNG ICs would produce different numbers even with identical power supplies reset timing and control signals. Figure 30 is a picture of the multi-board.

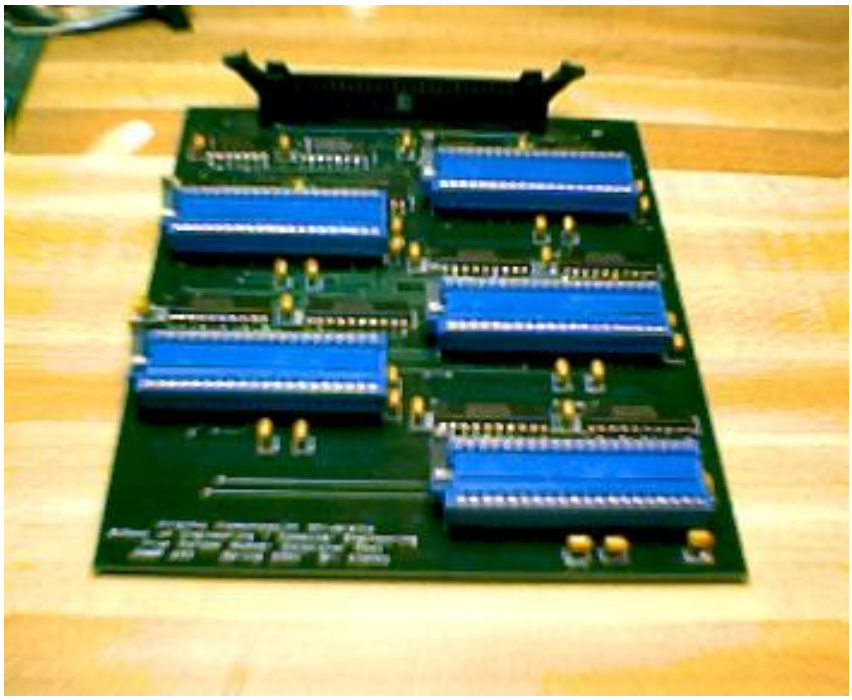

Figure 30 RNG Multi-Board

6.3.3 Avnet Development Board for Xilinx Spartan2 FPGA

The single RNG IC interface and the multi-board were designed to attach to the MC68306 bus as explained earlier. In order to provide a programmable digital hardware solution, a development board containing a Spartan2 FPGA was attached to the test equipment using the same bus port as the single and multi-board. The Spartan2 FPGA has plenty of I/O lines and the development board has them brought out to a 40 pin header. All that remained was to construct a simple wire-wrap board with two 40 pin connectors: one connector plugged into the test equipment and the other connector plugged into the Xilinx board. This arrangement allowed RNG designs to be built and tested more rapidly as well as providing proof that this family of RNGs could be implemented in any digital platform with analog components. Figure 31 is a photograph of the Spartan2 development board and wire-wrapped interface.

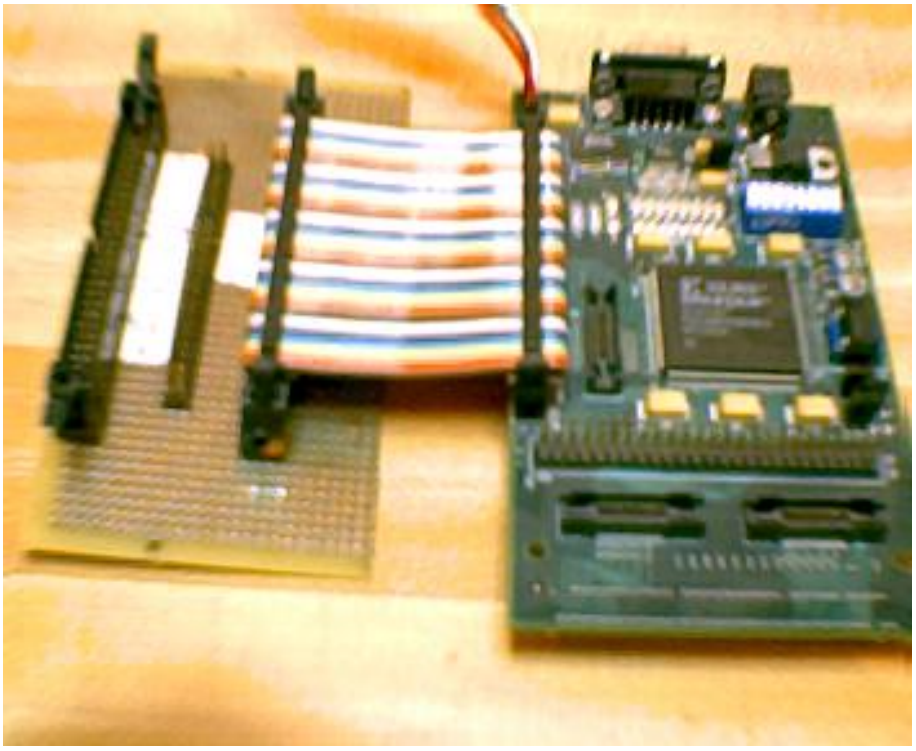

Figure 31 Xilinx Spartan2 Interface

CHAPTER 7 TEST RESULTS

7.1 RNG IC Test Results

The RNG ICs were tested in several ways. First, as detailed before, each IC was tested for functionality and for current drain and the results were sent to MOSIS. This testing is designed to help MOSIS improve its ability to support education by fabricating ICs for universities. Second, the RNG ICs were tested to see if they would really produce divergent streams of random numbers using the DoRR (do reset and read) test as described in chapter 6. Third, the RNG ICs were each subjected to statistical testing of the resulting stream of random numbers using the NIST800-22 test suite.

7.1.1 Reset and Read Test Results for RNG ICs

All five RNG IC chips were reset simultaneously, released from reset simultaneously and read simultaneously at 20 microsecond intervals. A total of 4 sets of 5 readings (one from each RNG IC) are taken. After the readings are taken, the board is reset and another 4 sets of 5 readings are taken at 20 microsecond intervals. This cycle continues until the operator presses a key to stop the test. Looking at the data reproduced on the next page, each column represents one of the five RNG ICs. Each row represents one reading taken from each of the five RNG ICs simultaneously. There are two important comparisons made by this test. First, each of the five RNG ICs should generate a different stream of numbers. That is, each column should be different. Second, each group of four numbers within a column should be different. The number of readings before the data started to diverge is an important indicator of the randomness of the RNGs. If there is no randomness associated with the RNG design then it is expected that each RNG would

output the same sequence of numbers. However, not only did each RNG IC output a different sequence of numbers than the other four, but each RNG never repeated the same sequence of numbers. Figure 29 shows the Reset and Read test for the RNG ICs.

```
Reset and read MULTI at 20 usec  20 [any key to quit]

E240A6BB F67501C4 BEAE68AB 5025C7DB 0AD8898E
DABB4A77 4C16ABE9 95C4B2A0 B5D747A7 16325610
0394D052 9EE06414 A78B7534 8C63E9B9 89077F72
F458F753 3C963B83 51F9495B D55EB858 901E3A32
[any key to quit]

D5804D67 8812E3ED F81088ED DABE218A 1B5B4130
A718ECD2 2E7F583B E14DE4D8 9F813B27 624A8D6C
23C895E0 9C557345 78ECE6E7 964682C6 755D34E7
4F1CAA2F F24E9F5A CEB5E4B4 5F35AE8A 862E4257
[any key to quit]

EDA810B2 5BC72510 87804C65 5025D7DB A6BEBA40
366D964D 6D1324F6 85C10031 5E9F8A18 AACD7C55
26ACFAB8 0BADD6C5 F89A8EAF 14F2A9B9 E5AEA400
50088C88 E593A84A 8F5A0B9E BAC9AD2A EBF1AA36
[any key to quit]

2B018ADE C75B1025 87804C67 5025C75B 1B5B4130
484469DA 720B20CC 179000C6 14B03D3D 46D8A42A
7FDB2C01 DC4987BA 49179C1A E4CA094F 6C301765
6AE5BB1E 9D5E581B 4947C4A2 C1BF03FB 574819B6
[any key to quit]

C64A20B6 ABB6284B F81088AD D67501D4 C4765B0C
BE9BC8AF 59B43287 A9AD9A4D 3070F265 9D4DACD8
A48F8D97 8B254C6C 80899067 B1314A1A 611996F5
80EE49AD 4D867E84 CF01E247 C4117AEB FB0F1052
[any key to quit]

ADA810BB 75F6CC09 3C046A2B DA3E218A 0314BDB1
0DD6A15F 393BDBC4 EE547F53 9FD52195 F66D817F
212105B9 6DE801FD EB929AB8 4F6D8B6A B560E8EB
169B946B BE942421 70948B4C 809D5181 AED5476D
[any key to quit]

175201B5 6BEAB402 F81088AD DABE218A 623B2306
4A90F065 366C593F 953B54DA 7C8C59BB 1A4DDFD9
CA7EA44A 5C919543 94AE6C77 774B7A6C 0DD3A8FB
B8925A8B FE89FD08 38C38D0E 49D81155 5B2D5000
[any key to quit]
TRNG>
```

Figure 32 Reset and Read at 20 uSec Intervals Multi RNG IC

There are no duplicate columns as expected, so each RNG IC is producing a different stream of random numbers. There are three cases where a RNG IC produced the same number after reset on two occasions. These duplicates are annunciated by the green, red and blue circles. However, in no case is a second number duplicated, indicating a high divergence of each RNG IC when sampled at 20 microsecond intervals as expected.

7.1.2 NIST800-22 Results for RNG ICs

The output random number streams of each of the five RNG ICs were also subjected to the NIST800-22 statistical tests. A total of 200,000,000 bits (100 sets of 2,000,000 bits) were gathered from each RNG IC at the rate of 20 microseconds per 32 bit random number. The data was input to the NIST800-22 test suite. The results of testing the raw data from the RNG ICs are tabulated in Table 10. A star to the right of any test result indicates the data failed that particular test for statistical randomness.

Table 10 Raw data NIST800-22 Results

	Raw data taken and processed by NIST800-22									
test	Chip 1		Chip 2		Chip 3		Chip 4		Chip 5	
Frequency	0.990		0.990		0.960		0.990		0.980	
Block Frequency	0.000	*	0.000	*	0.000	*	0.000	*	0.000	*
Cumulative Sums	0.990		0.975		0.955	*	0.990		0.980	
Runs	0.940	*	0.930	*	0.910	*	0.940	*	0.950	*
Longest Run	1.000		0.950	*	0.940	*	0.930	*	0.980	
Rank	0.980		1.000		0.990		1.000		1.000	
FFT	1.000		1.000		1.000		0.990		1.000	
Nonperiodic Templates	0.963		0.961		0.962		0.958	*	0.962	
Overlapping Templates	0.970		0.990		0.970		0.990		0.970	
Universal	0.890	*	0.870	*	0.950	*	0.980		0.920	*
Approximate Entropy	0.000	*	0.000	*	0.000	*	0.000	*	0.000	*
Random Excursions	0.988		0.990		0.998		0.984		0.991	
Random Excur Variant	0.996		0.992		0.993		0.986		0.983	
Serial	0.540	*	0.545	*	0.490	*	0.535	*	0.540	*
Lempel-Ziv	1.000		1.000		1.000		1.000		1.000	
Linear Complexity	0.960		0.980		0.990		1.000		0.990	

By inspection, the raw data output from each RNG IC passed more tests than it failed. For each RNG IC, the raw data failed from 5 to 7 of the 16 statistical tests, however. In particular, the raw data from all RNG ICs failed the Block Frequency test, the Runs test, the Approximate Entropy test and the Serial test. Consistently failing these tests indicates that some bit pattern (or patterns) is produced more often than the average and some bit pattern is produced less often than the average. The data is still not predictable because the Lempel Ziv compression algorithm is unable to compress the data at all.

7.1.3 Whitened NIST800-22 Results for RNG ICs

The output random number streams of each of the five RNG ICs were whitened and the whitened data was subjected to the NIST800-22 statistical tests. Two different methods of whitening were used. The first method was simply to XOR the output from the RNG ICs with a 32 bit maximal length LFSR as defined by the Xilinx Application Note[7]. The test results from running the NIST800-22 tests over the XOR whitened data is tabulated in Table 11. Whitened data from every chip passes the NIST800-22 tests.

Table 11 XOR Whitened Data NIST800-22 Results

	Data whitened by XOR and processed by NIST800-22				
test	Chip 1	Chip 2	Chip 3	Chip 4	Chip 5
Frequency	1.000	1.000	0.980	1.000	0.990
Block Frequency	0.980	1.000	0.990	0.990	0.990
Cumulative Sums	1.000	1.000	0.985	0.995	0.990
Runs	1.000	0.990	0.990	0.990	0.990
Longest Run	1.000	0.980	0.990	0.990	0.990
Rank	0.980	0.990	0.990	1.000	0.990
FFT	0.970	0.980	0.970	1.000	1.000
Nonperiodic Templates	0.990	0.990	0.991	0.989	0.990
Overlapping Templates	1.000	0.990	1.000	0.980	1.000
Universal	1.000	0.960	1.000	0.990	0.990
Approximate Entropy	0.990	0.990	0.990	1.000	0.990
Random Excursions	0.991	0.984	0.995	0.979	0.988
Random Excur Variant	0.991	0.987	0.995	0.979	0.995
Serial	0.985	0.995	0.985	0.980	0.995
Lempel-Ziv	1.000	1.000	1.000	1.000	1.000
Linear Complexity	1.000	0.990	1.000	1.000	1.000

One other variation of whitening was done as an experiment. Instead of simply XORing each 32 bit number produced by the RNG with the 32 bit output of the LFSR, half of the bits were XORd and half were XNORd in an attempt to help balance the number of 1's and 0's produced. Table 12 has the results of the XOR/XNOR whitening. As can be seen by inspection of Tables 11 and 12, there is no statistical difference apparent between a straight XOR and a split XOR/XNOR. Any form of whitening seems adequate to clean up the discrepancies and even up the distribution across the range.

Table 12 Whitened by XOR/XNOR Results

	Whitened by XOR,XNOR and processed by NIST800-22				
test	Chip 1	Chip 2	Chip 3	Chip 4	Chip 5
Frequency	1.000	1.000	0.980	1.000	0.990
Block Frequency	0.980	1.000	0.990	0.990	0.990
Cumulative Sums	1.000	1.000	0.985	0.995	0.990
Runs	1.000	0.990	0.990	0.990	0.990
Longest Run	1.000	0.990	0.980	0.980	0.970
Rank	0.990	1.000	1.000	0.990	1.000
FFT	0.970	0.980	0.970	1.000	1.000
Nonperiodic Templates	0.990	0.990	0.991	0.989	0.990
Overlapping Templates	0.990	0.980	0.980	1.000	0.990
Universal	1.000	0.960	1.000	0.990	0.990
Approximate Entropy	0.990	0.990	0.990	1.000	0.990
Random Excursions	0.991	0.984	0.995	0.979	0.988
Random Excur Variant	0.991	0.987	0.995	0.979	0.995
Serial	0.985	0.995	0.985	0.980	0.995
Lempel-Ziv	1.000	1.000	1.000	1.000	1.000
Linear Complexity	1.000	0.990	1.000	1.000	0.980

7.2 FPGA RNG Test Results

The RNG IC architecture was reproduced in a Spartan II FPGA in order to show how readily the design could be moved from one digital design suite to another. The design was coded in VHDL and realized using Xilinx xst. Data was gathered from this RNG configuration and processed using the NIST800-22 suite. Then the data was whitened as it was with the RNG ICs and tested again.

7.2.1 NIST800-22 Test Results for FPGA ASTRNG

Two sets of data were gathered from the FPGA ASTRNG at 20 microsecond intervals in the same method as data from the ASTRNG ICs was gathered. The NIST800-22 suite was run over each set of data as before. The results are in Table 13. A quick comparison shows excellent experimental agreement between the ASTRNG ICs and the ASTRNG as realized in the FPGA. Both implementations have similar results for the NIST800-22 tests on the raw data. In both cases the raw data shows problems with the Block Frequency, Longest Runs, Universal, Approximate Entropy and Serial tests. There is a little difference; the ASTRNG ICs have a problem with Runs test also while the FPGA has a problem with the Overlapping Templates test. Both the ASTRNG IC data and the FPGA realization data was whitened using a 32 bit LFSR clocked 32 times per sample – so that each bit in the LFSR would be generated anew for each data point. Both the ASTRNG IC and the ASTRNG FPGA realization passed all NIST800-22 tests after whitening.

Table 13 FPGA Based ASTRNG Raw and Whitened NIST Results

	Raw data		NIST800-22		Whitened
test	Data 1	Data 2	Wht 1	Wht 2	
Frequency	0.970	0.990	0.990	1.000	
Block Frequency	0.390 *	0.780 *	0.990	0.980	
Cumulative Sums	0.965	0.970	0.990	1.000	
Runs	1.000	0.980	1.000	0.960	
Longest Run	0.650 *	0.780 *	0.980	1.000	
Rank	1.000	0.990	1.000	1.000	
FFT	1.000	0.990	0.990	1.000	
Nonperiodic Templates	0.971	0.979	0.989	0.990	
Overlapping Templates	0.720 *	0.770 *	0.990	0.980	
Universal	0.010 *	0.340 *	0.990	0.990	
Approximate Entropy	0.000 *	0.010 *	1.000	0.990	
Random Excursions	0.990	0.994	0.993	0.993	
Random Excur Variant	0.997	0.997	0.993	0.997	
Serial	0.760 *	0.905 *	0.990	0.995	
Lempel-Ziv	1.000	1.000	1.000	1.000	
Linear Complexity	0.980	0.990	1.000	1.000	

7.3 Three LFSR Based FPGA CLTRNG Test Results

Instead of up and down counters, a barrel shifter and a transpose unit, a TRNG was constructed by concatenating the output of three different LFSRs to form a 32 bit random number. The LFSRs were each of different size and hence would produce different output sequences. Divergence comes as the ring oscillator for each of the LFSRs experiences noise. The PRNG that was used to whiten both the IC ATRNG and the FPGA ASTRNG was included in each of the CLTRNG realizations. As with the ASTRNG ICs, reset and read tests to insure the CLTRNG was diverging properly and NIST800-22 statistical tests to insure the CLTRNG generated the appropriate distribution were run on the data collected from the CLTRNGs.

7.3.1 Reset and Read Test Results for 3-LFSR CLTRNG

Since there was only one FPGA and therefore only one CLTRNG, the reset and read test was run a little differently. The CLTRNG was reset then a set of 20 readings was taken at 10 microsecond intervals. This process was repeated 8 times. Then the eight sets of readings were shown one each in eight columns. Each column had 20 readings taken consecutively. An asterisk is placed at the end of each row that has at least one duplicate entry. Starting at the beginning, count down each line that has at least one duplicate entry. The number of lines that have at least one duplicate is a reciprocal indicator of how fast the CLTRNG is diverging. As with the multi-chip reset and read, two aspects were important. First, the data does diverge after only a few samples. This illustrates that the CLTRNG is not repeatable even given circumstances where reset and sample timing is tightly controlled by a microprocessor and the tests are run within milliseconds of each other, precluding the possibility of significant temperature change.

Second, the data does not converge again after it diverges. The lack of convergence illustrates that the streams are truly divergent and not just a noise impulse.

```
[any key to quit]
F5507AB6 F5507AB6 F5507AB6 F5507AB6   F5507AB6 F5507AB6 F5507AB6 F5507AB6 *
FDA91E70 19470E71 19451E73 19451E73   19451E71 19472E70 19451E73 19451E73 *
7B26EEB3 69CD16F3 7A9161B3 A38C01C1   4C185522 54820C33 61D561B3 27F2954B
7BA1D0E0 CBE7572B 2F48246B 013106E0   7DBA38BA 6A18FBD9 E9C64905 16B79C4F
294A4A21 838EDF2B 94DFC210 ECF4A40C   8968A203 5BD55F10 BF56A659 1CB786C8
57684183 5474507F 6A4E9A70 04A00B38   9A64E6E9 C829A9C1 9D5C68F7 8697AD4C
9E926C2A F66FF995 41CEF6BF CA2A73CF   627A12A1 36B7E3BA 62CA849D 089A3C56
BB06F184 8A626AE8 2AB4D8F8 E1E81256   8C6E7383 6AE8C712 0B1F225D D339B475
AC9ED3A0 37407EE1 9DCAD36D B09308C4   8580ECBC 2C445E05 30BF3EA5 3CDFCAB5
0EE596B0 5B157CD6 EEFE8618 3978F7BC   F25C0699 81A125A8 9DFF01EF E09012F3
A75341CB 876032B3 9D790948 27F11493   73C374FE E046F59D 3FFC6A41 F808ED74
43AAF0CA 0F3DD157 50950740 34B0E9AA   29D9BBA2 7032912A EB0226F4 FE2EB7EA
558E527E 33C91846 6C48CEC0 DF55A5FD   B9330857 AC4A0950 90469BCA 3C97890D
0A091FFF C04AA121 3843F8CD 35899381   3C2E3200 BB1DE42B B5682889 39318BDA
FB0FA372 88E64E9D E6E5E9E9 9FEDF04C   64094361 E477E65B 20119DA8 F3BA548A
ACF5A901 BD088D5A 3CDCABD9 F8CAFE28   987468CA FDD56E6F 6429FF7A 47E72A54
A3A68B53 D697AAE0 470839D4 ADECEA91   2EA8E979 D128C07B CB2B003A 1EDB7680
AA1BDD5A ABF10449 05744276 18EDB692   0690AC6E F27B1CB8 70D7F92E D33F5DC5
063B9307 A0EDDC57 3F49E816 FA371F59   B69FF9B0 81482A82 69223362 67631CE7
4264385D B63ED630 7E825DD1 21962B82   1C166574 7211A019 BFF3FE66 33BC6D28
```

Figure 33 Reset and Read at 10 uSec Intervals for 3-LFSR RNG

7.3.2 NIST800-22 Test Results for 3-LFSR CLTRNG

Two sets of data were taken with the original 3-LFSR CLTRNG architecture. One set of data was taken using only the TRNG while the other set was taken using the TRNG whitened by the PRNG. Both of these data sets were taken by reading a random number from the CLTRNG every 40 microseconds. The NIST800-22 statistical tests were run on the data and the results are tabulated in Table 14. As can be seen by examining the NIST test results, the CLTRNG passes all but one of the NIST tests without whitening. With whitening it easily passes all NIST tests.

Table 14 NIST Results for 3-LFSR CLTRNG 40 uSec Interval

test	Raw data Data 1		Whitened Wht 1
Frequency	0.950	*	0.990
Block Frequency	1.000		0.990
Cumulative Sums	0.960		1.000
Runs	0.980		0.990
Longest Run	1.000		1.000
Rank	0.980		0.990
FFT	0.990		0.980
Nonperiodic Templates	0.988		0.991
Overlapping Templates	0.990		0.970
Universal	1.000		0.990
Approximate Entropy	0.980		0.970
Random Excursions	0.990		0.993
Random Excur Variant	0.982		0.989
Serial	0.990		0.990
Lempel-Ziv	1.000		1.000
Linear Complexity	0.990		0.980

7.4 Another Three LFSR Based FPGA CLTRNG Test Results

The 3-LFSR CLTRNG architecture was changed to different LFSR's. Instead of 16, 13 and 9 bit LFSRs with the most significant 2 bits ignored, the LFSRs used were 27, 13 and 12 bits long with 11, 11 and 10 bits respectively concatenated to form the 32 bit random number.

7.4.1 Reset and Run Test Results for Second 3-LFSR CLTRNG

A reset and read test was run on the second 3 LFSR CLTRNG design. As with the first 3 LFSR CLTRNG, the data were read in 32 bit random numbers at an interval of 40 microseconds. The results are shown in Figure 34. Note this configuration does not diverge as fast as the previous CLTRNG.

```
[any key to quit]
8A54BB81 8A54BB81 92B11503 8A54BB81   8A54BB81 92B11503 8A54BB81 8A54BB81 *
02E14EEC 02E14EEC 1DB7D31E 02E14EEC   02E14EEC 1DB7D31E 02E14EEC 02E14EEC *
D3492028 D3492028 03BED68E D3492028   D3492028 03BED68E D3492028 D3492028 *
4B41B334 4B41B334 1B1C9057 4B41B334   4B41B334 1B096457 F781DF24 4B41B334 *
EC4574F2 EC4574F2 F763E132 EC4574F2   EC4574F2 F7757132 24A9F585 EC4574F2 *
81047CAE 81047CAE 3F1120D0 81047CAE   81047CAE 3F0E5CD0 1954C8B2 81047CAE *
D4B56D9B D4B56D9B E40B413F D4B56C0E   D4B56D9B E419BD3F 0CA5FA32 D4B56C0E *
FF54D278 FF54D278 6778BE9A E994D282   FF54D278 6778129A 1BA92E2C FF54D282 *
EC11082B EC11082B 67EFF73B 47710BA6   EC11082B 67FA0B3B 54D37ECA A7510BA6 *
3A72A11A 30C432F3 29B2A705 2FB2A158   BA72A11A 29B11B6A 2CA05F9B 1912A158
27ACF398 6CBA04D4 282F85DD 4E6CF200   CF2CF398 283F2DD9 85ACB92C A96CF03B
1857FF54 141446DF 04C7E6CE 6297FFB0   48B7FF54 04D83980 FF67523B EA77FFDB
E46814E2 FB32405B 250D98F3 75E814DA   950814E2 5F7345D3 E3945B48 C1E8170B
1F461724 7438960A 20FD07F6 A6A617FE   2B5D8724 DEF06B46 BF7C3222 29A61412
5ACA5718 512F820D 7B92B59D 205F9902   5F8B2318 78CE48FC 01198F7F 74BF9943
A16244D2 D487B2A2 CB8C1AA7 7DEBB442   047FD4D2 263CB3C8 48EEF578 530BB688
B3203579 8146092A 749AB3F7 F0EC5EA4   82FBD979 5D2CE7E5 3C346DBE B6CC5F15
D9772276 A04641EE AD9FFF64 CA281FE1   2DA13276 8C495682 68B3A38B 46D1970D
7F5BB458 EC6DB116 6A5C9AD9 CB58D30E   705FBC58 2B4B3A70 0288CEB2 DC216245
031FDD30 CF3D648C ED192F6D 9C616BC2   FF3C2D30 3CC7976A E20E18BC A7D82CD9
```

Figure 34 Reset and Run Test 2nd 3-LFSR CLTRNG at 40 uSec Interval

7.4.2 NIST800-22 Test Results for Second 3-LFSR CLTRNG

Two sets of data were taken from this second 3-LFSR CLTRNG. One set was taken with the PRNG disabled and is just the TRNG output. The second set of data was taken with both the TRNG and the PRNG enabled. The results are in Table 15.

Table 15 NIST Results for Second 3-LFSR CLTRNG

	Raw data	Whitened
test	Data 1	Wht 1
Frequency	1.000	0.990
Block Frequency	0.970	0.990
Cumulative Sums	0.995	0.990
Runs	1.000	1.000
Longest Run	0.970	0.990
Rank	1.000	1.000
FFT	0.990	0.990
Nonperiodic Templates	0.989	0.988
Overlapping Templates	0.980	0.990
Universal	1.000	0.990
Approximate Entropy	0.980	1.000
Random Excursions	0.988	0.990
Random Excur Variant	0.996	0.994
Serial	0.975	0.990
Lempel-Ziv	1.000	1.000
Linear Complexity	1.000	0.980

7.4 Four LFSR Based FPGA CLTRNG Test Results

The 4-LFSR CLTRNG architecture was designed using 13, 11, 9 and 7 bit LFSRs. 32 of the 40 bits are concatenated to form the RNG. The four oscillators for the LFSRs are cross-coupled as described in Table 16.

Table 8 4 Bit LFSR OScillator Cross-coupling

ring oscillator	coupled to
lfsr7	lfsr9 & lfsr11
lfsr9	Lfsr11 & lfsr13
lfsr11	lfsr13 & lfsr7
lfsr13	lfsr7 & lfsr9

One interesting aspect of a CLTRNG made from 4 LFSRs as opposed to 3 LFSRs is the relative rate of divergence of the two CLTRNGs. Since the CLTRNG with 4 LFSRs has 4 points of divergence whereas the CLTRNG with 3 LFSR has only 3 points of divergence, it seems logical that the 4 LFSR CLTRNG would show a faster divergence. The classic Reset and Read test only takes 8 sets of data. In order to make a more meaningful measure of divergence, the Reset and Read test was changed to run continuously and store how many rows of the 8 sets of readings have duplicates per set of 8 readings. When the test is stopped, 20 counts are printed out. Each count represents the number of rows of readings with duplicates. For example, if the first count, count[0], had a value of 10 then that would mean 10 groups of 8 sets of data have no (zero) rows with duplicates. If the second count, count[1], had a value of 100 then that would mean 100 groups of 8 sets of data had only 1 row with duplicate readings on it. In short, the 20 counts taken together represent a histogram of duplicate readings. The histogram can be used to measure divergence. A large, steep bell indicates a high rate of divergence as few

duplicate readings are produced. A short, gently sloping bell indicates low divergence as there are many duplicate readings. For example, consider the following two sets of counts. The set labeled A was taken at 40 microsecond intervals. The set labeled B was taken at 100 microsecond intervals. Since there is more opportunity for noise to affect the slower readings, it is expected that set B would show a higher divergence. The graph of the data as percentages bears this out in Figure 35.

A: 40uS	B:100uS
0	0
27	224
486	1524
2281	4571
4638	4244
8885	3056
8457	1792
8534	922
7997	425
6226	191
4022	84
3359	40
2079	18
1556	5
964	4
472	0
291	1
198	0
93	0
53	0

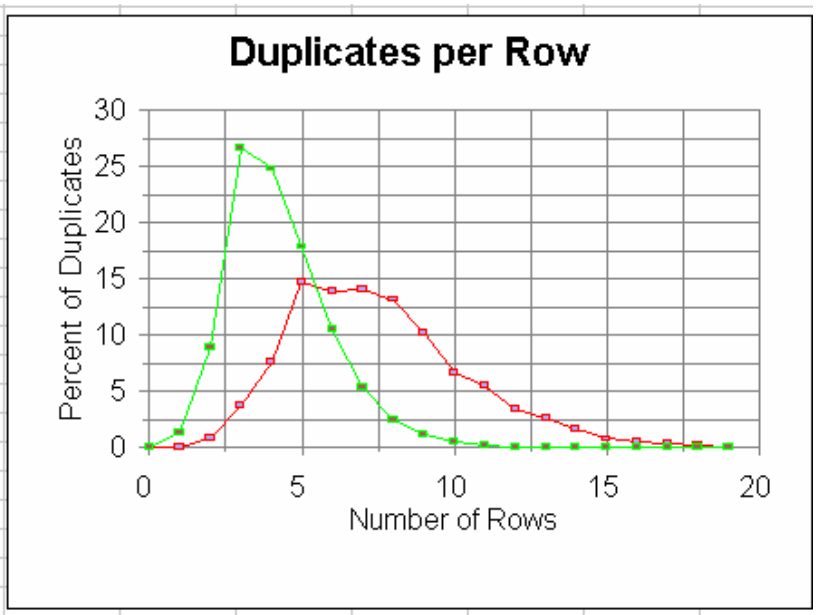

Figure 35 Reset and Read Histograms for CLTRNG

7.4.1 Reset and Read Test Results for 4-LFSR CLTRNG

The reset and read tests for the 4-LFSR CLTRNGs show an improved divergence as expected. Since there are 4 ring oscillators instead of 3, each ring oscillator can have multiple next values and the divergence has been shown to be exponentially related to the number of possible next values, it is expected that the divergence of the 4-LFSR CLTRNG will be higher than the divergence of the 3-LFSR CLTRNG. The Reset and Read data for both the 3-LFSR and the 4-LFSR CLTRNG are tabulated and charted in Figure 36. Both sets of data were taken at 40 microsecond intervals.

3 LFSR	4 LFSR
0	0
8	41870
2047	293
1649	0
398	0
72	0
20	0
2	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0

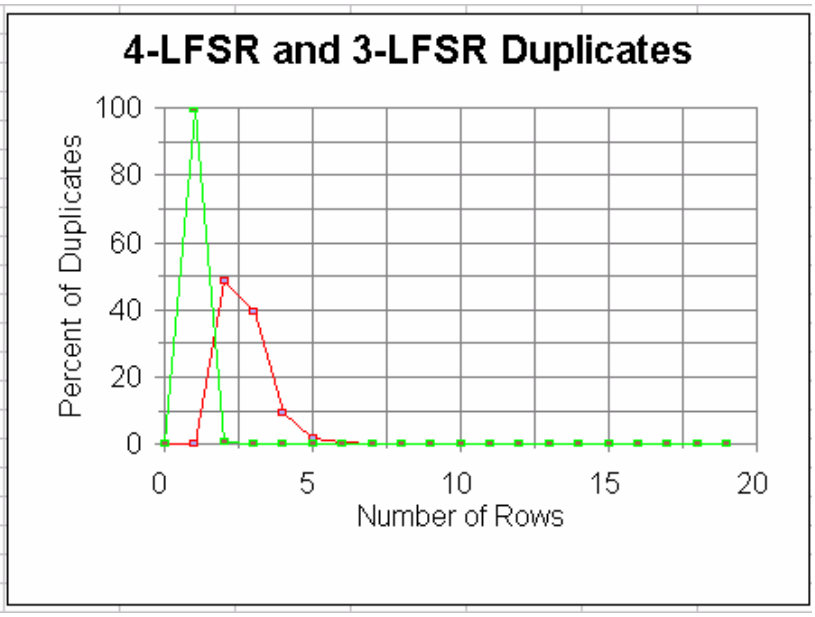

Figure 36 3-LFSR and 4-LFSR Divergence

7.4.2 NIST800-22 Test Results for 4-LFSR CLTRNG

Two pairs of two sets of data were taken with the 4-LFSR CLTRNG architecture. Each pair consists of one set of data from the TRNG only and one set of whitened data. Test results for the 4 LFSR CLTRNG are tabulated in Table 17.

Table 9 4-LFSR CLTRNG Nist800-22 Results

	Raw data				Whitened data	
test	Data 1		Data 2		Wht 1	Wht 2
Frequency	0.590	*	0.480	*	0.990	0.970
Block Frequency	0.960		0.980		0.990	0.990
Cumulative Sums	0.605	*	0.485	*	0.990	0.965
Runs	0.690	*	0.630	*	0.980	1.000
Longest Run	0.960		0.930	*	0.980	1.000
Rank	0.990		0.990		1.000	1.000
FFT	0.990		0.990		0.980	0.990
Nonperiodic Templates	0.981		0.983		0.991	0.991
Overlapping Templates	0.770	*	0.740	*	0.980	1.000
Universal	0.970		0.970		1.000	0.980
Approximate Entropy	0.950	*	1.000		0.960	0.990
Random Excursions	0.986		0.983		0.984	0.998
Random Excur Variant	0.995		0.998		0.984	0.988
Serial	0.995		0.990		0.995	0.990
Lempel-Ziv	1.000		1.000		1.000	1.000
Linear Complexity	0.990		1.000		0.960	0.990

7.5 Using a TRNG to Whiten a PRNG

As mentioned earlier, an interesting result from this work is the discovery of a way to whiten the output from a LFSR based PRNG. The 3-LFSR CLTRNG architecture includes a PRNG and provisions were made for reading (1) the TRNG, (2) the PRNG or (3) the XOR of the TRNG and PRNG. As is shown by the data in Figure 37, PRNG output fails the Rank test in the NIST800 test suite. But after XORing with the TRNG, the combined stream passes the Rank test.

test	PRNG data		TRNG data		PRNG^TRNG
Frequency	0.970		0.950	*	0.990
Block Frequency	0.980		1.000		0.990
Cumulative Sums	0.970		0.960		1.000
Runs	0.990		0.980		0.990
Longest Run	0.990		1.000		1.000
Rank	0.000	*	0.980		0.990
FFT	0.980		0.990		0.980
Nonperiodic Templates	0.989		0.988		0.991
Overlapping Templates	0.980		0.990		0.970
Universal	0.990		1.000		0.990
Approximate Entropy	0.980		0.980		0.970
Random Excursions	0.992		0.990		0.993
Random Excur Variant	0.993		0.982		0.989
Serial	1.000		0.990		0.990
Lempel-Ziv	1.000		1.000		1.000
Linear Complexity	0.990		0.990		0.980

Figure 37 LFSR Data Whitened By TRNG

CHAPTER 8 CONCLUSION

8.1 Summary of Work

A methodology for the digital design of digital true random number generators has been presented. Digital schematics and the Mentor Graphics digital ASIC design software was used to design and fabricate digital true random number generators. VHDL and Xilinx FPGA design software was used to reproduce the original true random number generator. Data measurements from both the ASTRNG ICs and the FPGA ASTRNG realizations were taken and showed very similar results, proving the design was portable across digital platforms. One other architecture for the divergent path random number generator was realized by replacing the counters, shifter and transposer with concatenated LFSRs. Three realizations of this CLTRNG architecture were constructed and tested. The two CLTRNG realizations composed of three LFSRs generated a stream of random numbers that scored significantly higher on the NIST800-22 tests. The CLTRNG realization composed of four LFSRs showed a higher divergence as expected but it also showed an unexpectedly poorer statistical composition.

8.2 Lessons Learned

One important lesson learned was that each experiment in the sequence of any research has value. In particular, each time the Reset and Read test was modified the previous experiments should have been repeated to include data from the older experiments for comparison with data from the newer ones.

Another important lesson learned was that of keeping accurate records. In some cases where years passed between gathering the data and writing up the experiment it was difficult to reproduce the thinking behind each experiment. In the future more accurate records will be kept. A statement of purpose for the experiment, the method, the data and the results will be written up for each experiment to help clarify thinking, eliminate errors and guide future work. The experimental writeup will be completed as soon as reasonably possible during and after the experiment.

Another important lesson learned was the power of technology to increase the pace of experimentation. This research contained large amounts of data – each NIST800-22 run required 25,000,000 bytes of data. Work was started on a 700MHz P3 machine with 512M of 133 MHz SDRam. Crunching one set of NIST800-22 statistical tests took about 5 hours. Later work was done on 2GHz P4 machines with 1G of 400MHz DDRam. The same NIST800-22 statistical tests ran in 20 minutes on these machines.

8.3 Future Work

As stated earlier, the original TRNG architecture was an attempt to jumble bits randomly. Counters were used to generate bits then the resulting bits were shifted and transposed. NIST800-22 tests brought out a critical weakness of this design – it has detectable frequency components. This weakness was somewhat overcome by switching from counters/shifter/transposer to concatenated LFSRs for the generating circuit. After having time to consider more carefully, it was not necessary to use counters. Instead, since any prime, or indeed any number that is relatively prime with respect to the range of numbers, would generate every possible combination, a generator could be constructed from an adder that would iteratively add the prime in the place of a counter that could

only add “1”. Then if a prime were chosen that was roughly half the length of the range, it would serve to toggle many bits on every iteration which would alleviate the need for the down counter and the need for bit balancing (trying to make the number of 1’s and 0’s equal over the long term). Whereas the CLTRNG formed by concatenation of three LFSRs only alters 3 bits per generation period (one in each LFSR), a generator built from a prime-adder circuit (ASTRNG) could easily alter half of the bits in the number at each iteration. Hence the ASTRNG should be able to be sampled considerably faster than the CLTRNG. An interesting research project would be to use the FPGA to derive the best possible CLTRNG and ASTRNG and then to layout both designs and have ICs constructed and tested.

REFERENCES

[1] Stallings, William, *Network Security Essentials – Applications and Standards*, Prentice-Hall Inc., 2000, pp. 69.

[2] Brent, Richard P., "Note on Marsaglia's Xorshift Random Number Generators", *Journal of Statistical Software*, Vol. 11, Issue 5, August 2004.

[3] K.H. Tsoi, K. H. and K.H. Leung et. al., "Compact FPGA-based True and Pseudo Random Number Generators", *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*, 2003, pp. 51-61.

[4] Kohlbrenner, Paul and Kris Gaj, "An Embedded True Random Number Generator for FPGAs", *Proceedings Of The 2004 ACM/SIGDA 12th International Symposium On Field Programmable Gate Arrays (FPGA '04)*, pp. 71-78.

[5] *Collins English Dictionary – Complete and Unabridged 6th Edition 2003*. Harper Collins Publishers 2003.

[6] Rukhin, Andrew and Juan Soto et. al., "Statistical Test Suite For Random And Pseudorandom Number Generators For Cryptographic Applications (NIST Special Publication 800-22)", *National Institute of Standards and Technology*, May 15 2001.

[7] Alfke, Peter, *Efficient Shift Registers, LFSR Counters and Long Pseudo-Random Sequence Generators*, Xilinx Publication XAPP052, July 7, 1996.

[8] McCollum, James M. and Joseph M. Lancaster et. al., "Hardware Acceleration of Pseudo-Random Number Generation for Simulation Applications", *Proceedings of the 35th IEEE Southeastern Symposium on System Theory (SST '03)*, March 2003, pp. 299-303.

[9] Griffiths, Dawn, *Head First Statistics*, O'Reilly Media Inc., 2009, pp. 230.

[10] Cusick, Thomas W. and Pantelimon Stanica, *Cryptograhic Boolean Functions and Applications*, Academic Press, 2009, pp. 19.

[11] Thamrin, N.M., G. Witjaksono. et. al., "An Enhanced Hardware-based Hybrid Random Number Generator For Cryptosystem", *ICIME '09 Proceedings of the International Conference on Information Management and Engineering*, April 03 – April 05 2009, pp. 152-156.

[12] Marsaglia, George, "Random Numbers Fall Mainly in the Planes", *Proceedings of the National Academy of Sciences*, Vol. 61, September 1968, pp. 25-28.

[13] Petrie, Craig S. and J. Alvin Connelly, "A Noise-Based IC Random Number Generator for Applications in Cryptography", *IEEE Transactions On Circuits And Systems—I: Fundamental Theory And Applications*, Vol. 47, No. 5, MAY 2000, pp. 615-621.

[14] Bucci, Marco and Lucia Germani et.al., "A High-Speed IC Random-Number Source for SmartCard Microcontrollers", *IEEE Transactions On Circuits And Systems—I: Fundamental Theory And Applications*, Vol. 50, No. 11, November 2003, pp. 1373-1380.

[15] Walsh, James J. and Randall Paul Biesterfeldt, "Method And Apparatus For Generating Random Numbers", US Patent 6,480,072 B1, November 12 2002.

[16] Stojanovski, Toni and Ljupco Kocarev, "Chaos-Based Random Number Generators—Part I: Analysis", *IEEE Transactions On Circuits And Systems—I: Fundamental Theory And Applications*, Vol. 48, No. 3, March 2001, pp. 281-288.

[17] Stojanovski, Toni and Johnny Pihl et. al., "Chaos-Based Random Number Generators—Part II: Practical Realization", *IEEE Transactions On Circuits And Systems—I: Fundamental Theory And Applications*, Vol. 48, No. 3, March 2001, pp. 382-385.

[18] Chu, Pong P. and Jones, Robert E., *Design Techniques of FPGA Based Random Number*. Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, Ohio and NASA Glen Research Center, Cleveland Ohio, October 26, 1999.

[19] Knuth, Donald E., "Random Number Generator", US Patent 3,548,174, August 10 1966.

[20] Tausworthe, Robert C., "Random Numbers Generated By Linear Recurrence Modulo Two", *Mathematics of Computation*, Vol. 19, 1965, pp. 201-209.

[21] MacLaren, M. Donald and George Marsaglia, "Uniform Random Number Generators", *Journal of the ACM*, Vol. 12, Issue 1, 1965, pp. 83-89.

[22] FIPS 140-2: Federal Information Processing Standards Publication 140-2, "Security Requirements for Cryptographic Modules", *U.S. Department of Commerce / National Institute of Standards and Technology*, December 03 2002.

[23] Bucci, Marco and Lucia Germani et.al., "A High-Speed Oscillator-Based Truly Random Number Source for Cryptographic Applications on a Smart Card IC", *IEEE Transactions On Computers*, Vol. 52, No. 4, April 2003, pp. 403-409.

[24] McTaggert, Jeff and Brook Burson, “Multi-Clock Random Number Generator”, *Motorola White Paper*, September 1999.

[25] Oerlemans, Robert Vincent Michael, “Digital True Random Number Generating Circuit”, US Patent 6,807,553 B2, October 19, 2004.

[26] Wilber, Scott A., “Integrated True Random Number Generator”, US Patent Application Publication US 2010/0281088 A1, Nov 04 2010.

[27] Weigandt, Todd C. and Beomsup Kim et. al., ”Analysis of Timing Jitter In CMOS Ring Oscillators”, *IEEE International Symposium on Circuits and Systems, 1994 (ISCAS’94)*, Vol. 4, May 30 1994, pp. 27-30.

[28] Chaitin, Gregory J., *Exploring Randomness (Discrete Mathematics and Theoretical Computer Science)*, Springer-Verlag, 2001, pp. 111-129.

[29] Mitchum, Sam and Robert H. Klenke, “Design and Fabrication of a Digitally Synthesized, Digitally Controlled Ring Oscillator”, *Proceedings of the Third IASTED International Conference on Circuits, Signals and Systems (CSS 2005)*, October 24-26 2005, pp. 26-30.