2015•2016
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: elektronica-ICT

Masterproef

A Testable True Random Number Generator for Linux Security Applications

Promotor :
Prof. dr. ir. Nele MENTENS

Promotor :
Dhr. BOHAN YANG

Daniel Wietrzychowski

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

2015•2016

Faculteit Industriële ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterproef

A Testable True Random Number Generator for Linux Security Applications

Promotor :
Prof. dr. ir. Nele MENTENS

Promotor :
Dhr. BOHAN YANG

Daniel Wietrzychowski

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

universiteit hasselt | KU LEUVEN

Preface

This master's thesis is the condensed form of a year of effort. I learned a great deal about an interesting subject and achieved to create a working prototype.

I would like to thank my internal promotor dr. prof. Nele Mentens, my external promotor Bohan Yang and his colleague Vladimir Rozic for their help and guidance during the period of my thesis.

Furthermore, I would like to thank my closest friends and family members for a great deal of support and patience.

I hope this thesis may prove to be useful to you.

Daniel Wietrzychowski

Diepenbeek, August 22, 2016.

Table of Contents

Table of Tables

Table of Figures

Glossary

D flip-flop – A type of flip-flop which stores the input on the rising edge of the clock signal.

Fibonacci Architecture – A many-to-one feedback architecture where all enabled taps are mixed into 1 feedback bit.

Field Programmable Gate Array [FPGA] – A FPGA is an IC which can be customized after manufacturing, several Configurable Logic Blocks are provided which could be wired into a custom electronic circuit.

Galois Architecture – A one-to-many feedback architecture where 1 bit is mixed into all enabled taps.

Hardware Description Language [HDL] – A computer language to describe the structure and behavior of electronic circuits.

Initialization Vector [IV] – Starting value of a cyclic process.

Linear Feedback Shift Register [LFSR] – A shift register in which some of its outputs are used with logic elements to produce the next input.

Module – A module is part of a Verilog program as a means to group logic which serves a discrete purpose.

Ring Oscillator [RO] – A device composed of an odd number of NOT-gates in a feedback configuration.

Ring Oscillator with Fibonacci architecture [FIRO] – A ring oscillator design based on a linear feedback shift register with a many-to-one or Fibonacci architecture.

Ring Oscillator with Galois architecture [GARO] - A ring oscillator design based on a linear feedback shift register with a one-to-many or Galois architecture.

T flip-flop – A type of flip-flop which toggles the output on the rising edge of the clock signal while the input is high.

Abstract

Security application extensively utilize randomly generated numbers. Generating these strictly unpredictable numbers is a challenge. My Master's thesis is about delivering a stable stream of entropy to security applications.

A system on chip design which generates random numbers will need 3 distinct modules: A module as entropy source, a module for post processing and a module for communication with another device. The entropy generated originates from a randomly oscillating signal which is sampled. The sampled data will be whitened, to reduce any hardware bias. Finally, the generated data will be tested for randomness and be prepared for transmission to a receiving device.

The proposed design will be made with an "AVNet S6LX9 MicroBoard" development board. A ring oscillator based design will generate the entropy for the system. Any hardware bias will be reduced from the generated data stream. Finally, a MicroBlaze softcore processor will evaluate the generated data and proceed to transfer the stream if the evaluation is successful.

Abstract in Nederlands

Beveilligingstoepassingen maken extensief gebruik van willekeurige nummers, het genereren van deze strikt onvoorspelbare nummers is een uitdaging. Het doel van deze masterproef is het leveren van een stabiele onvoorspelbare entropiebron aan beveiligingstoepassingen.

Het ontwerp voor een system-on-chip willekeurige nummergenerator vereist 3 modules: Een entropiebron, naverwerking en connectiviteit. De entropiebron bemonstert een willekeurig oscilerend signaal, data uit de entropiebron wordt geanalyseerd in een testmodule. De naverwerking van de data zorgt voor een statistische uniforme uitvoer. Tenslotte zorgt een module voor connectiviteit tussen het hardwareplatform en de beveiligingstoepassing.

Het voorgestelde ontwerp wordt uitgevoerd op het "AVNet S6LX9 Microboard" developmentboard. De entropie wordt gegenereerd via een op ring-oscilator gebasseerd ontwerp. De data wordt gefilterd om enige hardware voorkeur te reduceren. Tenslotte wordt de gegenereerde data in een MicroBlaze softcore geëvalueerd om, indien de evaluatie positief is, te worden verzonden naar een ontvangend apparaat.

1 Introduction

Security applications utilize strong random numbers to generate session keys, certificates and random challenges in various authentication protocols. These numbers should be generated with a true random number generator which produces uniformly distributed values. To ensure the quality of the true random number generator, a real-time evaluation of randomness is required. If the evaluation succeeds, the random information is kept, otherwise the output of the random number generator is blocked. [1]

Research will be conducted for the KU Leuven at (Computer Security and Industrial Cryptography) COSIC[2] situated in Leuven and (Embedded Systems & Security) ES&S[3] situated in Diepenbeek.

1.1 Problem Description

In Unix-like operating systems, a special file /dev/random is provided to generate strong random numbers. It collects entropy from device drivers and from user behavior which is stored in an entropy pool. However, in certain environments like in network servers, lack of user interaction might prove a poor source of entropy, resulting in the depletion of the entropy-pool when random numbers are needed. This might set a limitation to security applications. [1]

According to an article of Acunetix [4], whom conducted a survey of the top million utilized sites in 2010, more than 40% of websites could be confirmed to use a Unix-like system.

Security applications running on the machines within the affected scope, i.e. Unix-like systems, could stall due to a depleted entropy-pool which is used for /dev/random/. A dedicated hardware true random number generator with built in tests, could significantly increase reliability of security applications, while ensuring equally strong random numbers as with Unix's /dev/random.

1.2 Objectives

The main objective of the thesis is to develop and construct a true random number generator with on-board real-time evaluation, to improve the random entropy source of Linux.

To complete the main objective, several HDL cores will need to be created.

First, a HDL block for the generation of entropy will need to be designed and tested, the output of this block is random data.

Next, a HDL block needs to be created to remove the hardware bias of the entropy source.

Finally, a softcore processor needs to be assigned, this will evaluate generated whitened information and when successful, transfer the data to a Linux-based system.

These HDL-cores need to be ported to an embedded system with the aim for a small form factor, low energy consumption and high throughput. Power will be drawn from an active USB port, which could also be used for the data transmission.

1.3 Materials and Methods

To realize a testable true random number generator, an extensive literature study is conducted to gain in depth information about random number generation. The literature study will help to delimit the hardware design and implemented algorithms. Furthermore, it will help to gain insight in implementing custom drivers for the Unix-kernel.

As an embedded system a FPGA development board will be used to test and design specialized HDL cores. The following hardware blocks will need to be created:

- As an entropy source, the jitter of a ring oscillator among a few other possibilities will be evaluated to generate random noise.
- Hardware generated entropy is biased, a whitening algorithm must be implemented to achieve a statistical ideal source of noise.
- The evaluation of the generated entropy will be evaluated with an algorithm described in the publication of the National Institute of Standards and Technology[5]. The aim is for a hardware-friendly On-the-fly method of entropy measurement/estimation.
- Finally, the generated data must be transmitted to the receiving Unix-kernel.

When the hardware tests are implemented and tested on an FPGA development board, the hardware will be ported and redesigned to an embedded system.

2 Hardware Platform

The hardware platform used is the “AVNet Spartan-6 LX9 MicroBoard” hardware development kit.

Several advantages were present to choose this particular platform, they include:

a low cost platform,
a small footprint,
a decent on board Spartan 6 FPGA,
a serial interface and
an Ethernet interface.

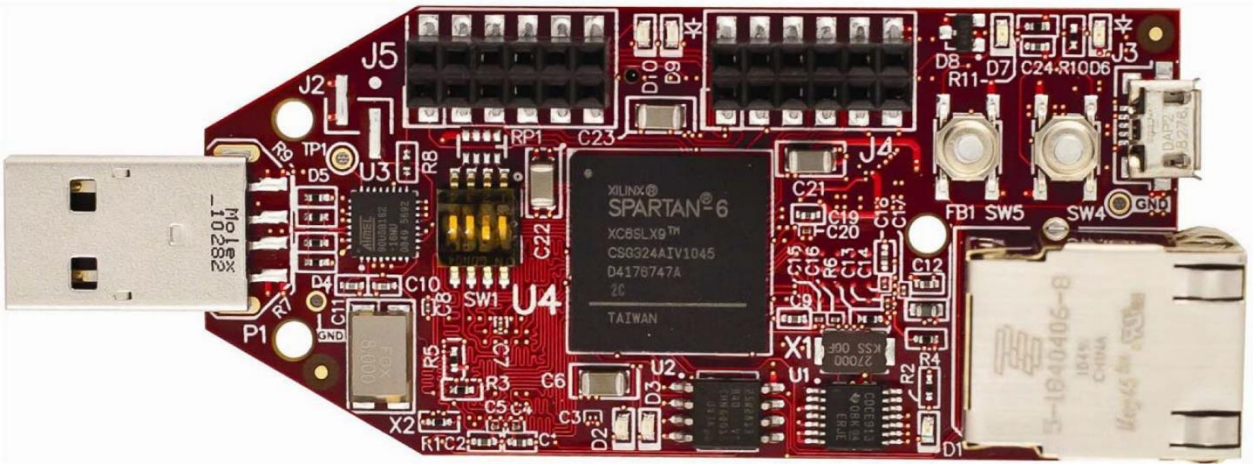

Figure 2.1 Top view of an AVNet Spartan-6 LX9 MicroBoard [6]

The Spartan-6 FPGA development board (Figure 2.1) can be programmed using the USB-A connector which is connected to a JTAG-controller. The micro USB-B port is connected to a CP2102 chip, a serial to USB converter. To utilize the UART-to-USB port, the necessary driver (a driver for the CP2102 chipset) need to be installed. The Ethernet interface is a convenient way to transport a stream of data to another device, although it is not as easy as serial communication, it can allow a transfer speed [7] of up to 25Mbit/s, surpassing the serial connection speed (115200 baud) with a great margin.

2.1 Hardware Development Kit

The Spartan-6 FPGA is preprogrammed using the Xilinx hardware development kit, hardware modules were developed using the “ISE Design Suite v14.7” while adding existing modules, such as the “Micro Blaze softcore processor” and the “Ethernet Lite Media Access Controller”, was done using “Xilinx Platform Studio v14.7”. To implement software written in C/C++, the “Xilinx Software Development Kit v14.7” was used.

2.2 Unix-Based Receiving Hardware

To demonstrate the developed hardware, a small Unix-based computer was used, this will act as the receiving server of the entropy. The chosen computer system was, for demonstration purposes, a Raspberry Pi version 3, due to relative ease of use and its small size.

The used operating system for the Raspberry Pi, Raspbian "Jessie" with a compiled Unix kernel version 4.4 [8], was compatible with the CP2102 serial-to-USB chip without making kernel modifications.

2.3 A Simplified Overview

To summarize the test setup, a computer with a Debian-based operating system was used to improve its entropy source. To generate entropy for the computer, custom hardware was developed using the AVNet development board. A diagram of the setup is shown in Figure 2.2. Within the development board, a random data stream is generated. The generated data will be post processed to remove hardware bias of numbers. Finally, a MicroBlaze softcore processor will be used to evaluate the random data and transfer the data to another system.

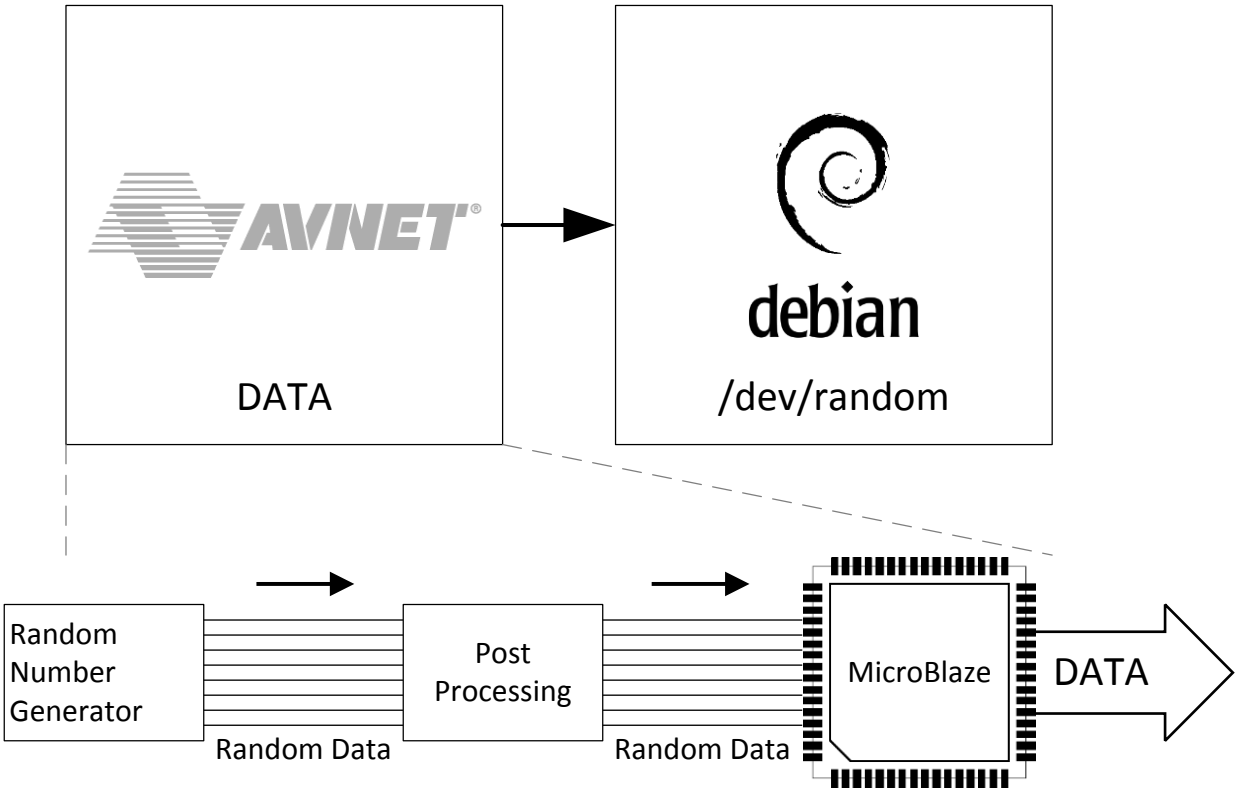

Figure 2.2 General test setup overview

3 Generating Random Data

Random data could be generated using 2 main methods: one relies on a secret and often complicated algorithm which generates numbers; another relies on sampling an apparently random phenomenon such as white noise. The former is referred to as a pseudorandom number generator and the latter is referred to as a true random number generator.

- **Pseudorandom Numbers**

  An algorithm can generate statistical random numbers, though these numbers may appear strong random numbers, given the correct algorithm, all numbers in the sequence can be determined. This is often referred to as pseudorandom. A pseudorandom number generator is easily implemented using a linear feedback shift register [LFSR], this will generate a sequence based on the architecture of the device, the polynomial given for the feedback and the initial seed. Given all different settings for the LFSR, the complete sequence of numbers could be calculated. If one were to guess all settings of an LFSR (i.e. after viewing a sequence of generated numbers), all future numbers could be accurately predicted. [9]

  These pseudorandom numbers can be engineered to produce a perfectly uniform distribution and appear to be perfectly random, though when the algorithm is known, the sequence of numbers contain no entropy.

  Multiple approaches exist to design a pseudorandom number generator, but this is beyond the scope of this Master's thesis.

- **True Random Numbers**

  True random numbers are based on sampling random phenomena, the strength of these numbers rely on the unpredictability of the sampled source. One could generate a sequence of bits which are truly random by performing a sequence of coin tosses with a fair coin, the state of the next coin toss cannot be accurately determined, due to the random nature of a coin toss.

  While a coin toss may seem perfectly random, it is impractical to implement on a chip. The methods explored in this Master's thesis are based upon ring oscillators, these are loops of an odd number of logical inverter which are chained together. These ring Oscillators will generate an oscillating signal of with an unknown frequency based on a variety of environmental factors. Ring oscillators are also known to exhibit a fair amount of jitter. While this is not an ideal source of noise, the random factors present allow entropy to be extracted.

  Data extracted from a source of noise is often biased and unlike a fair coin, it behaves more like weighted dice. Post processing of the generated data, can reduce the hardware bias and therefore increase the entropy of the data.

3.1 Noise source

Random data originates from a noise source, the noise is periodically sampled which produces a stream of bits. Multiple approaches exist to create a noise source. The methods tested upon are described below. These methods use oscillating loops of logic without clock circuitry and may be destructively optimized away by the FPGA software. Within the Xilinx Constraints Guide[10], the KEEP_HIERARCHY constraint exists to preserve the hierarchy of the applied modules.

To test each of the noise sources, a data width of 20000 bits was used, this is the amount of data required to perform a test described within the FIPS140-2 [11]. The preliminary tests conducted with a python script visually represented the generated data as a plot of the value in function of the sample number and as a distribution of said data. The python script *Analyze_data.py* is provided in Appendix A.

Intended use of *Analyze_data.py* given a Unix based system with python and matplotlib installed:

```
#> cat /location/of/device | ./Analyse_data.py [data width in bytes]
```

The calculated value plot should contain no recognizable patterns and the data distribution should be close to a uniform distribution. The script also calculated a few indicators to quickly identify random data of acceptable quality. The indicators used and their expected values if the data was random are:

- Average value: ~127
- Median value: ~127
- Percentage of binary Ones: ~50%
- Distribution Minimum and Maximum: ~0,39%
- Shannon Entropy: ~8 bit

E.g. Visual characteristics of /dev/random given in Figure 3.1

Command: `#> cat /dev/random | ./Analyse_data.py 2500`

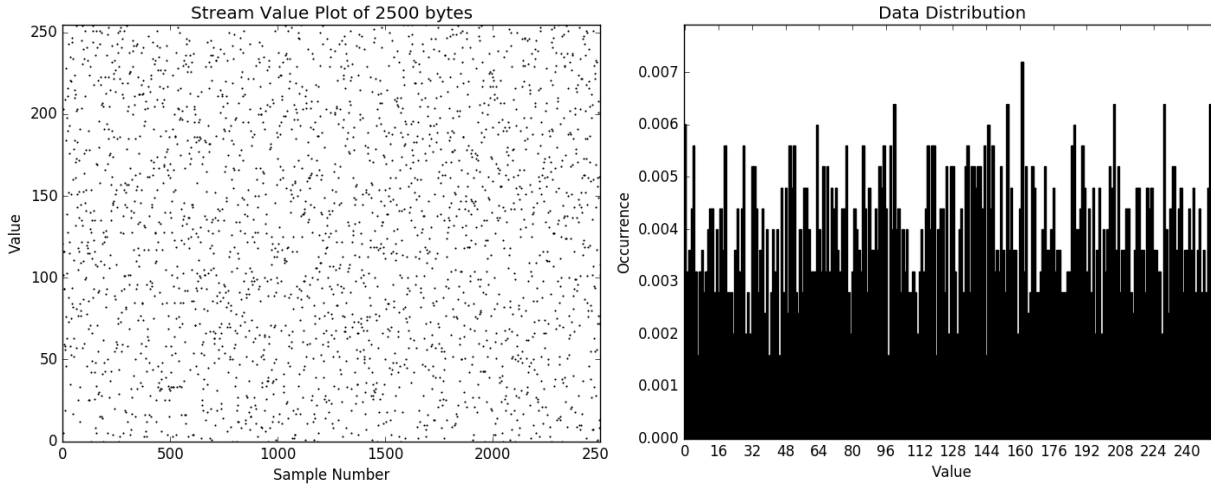

Figure 3.1 Visual Representation of /dev/random after gathering random 2500 bytes

3.1.1 Ring Oscillator Design

Gathering random data from a ring oscillator (Figure 3.2) can be achieved by sampling a fast ring oscillator by a slow ring oscillator. To get different speeds of operation, more delay elements (i.e. a set of 2 consecutive inverters) can be daisy-chained together. The model describing the hardware shown in Figure 3.2 is done in the file *RO_PARAM.v* which is provided in Appendix B.

Required signals:

- EN: Enable signal
- OUT: Serial output data

Optional Parameter:

- DELAY: Amount of consecutive *delay-blocks*, defaults to 1

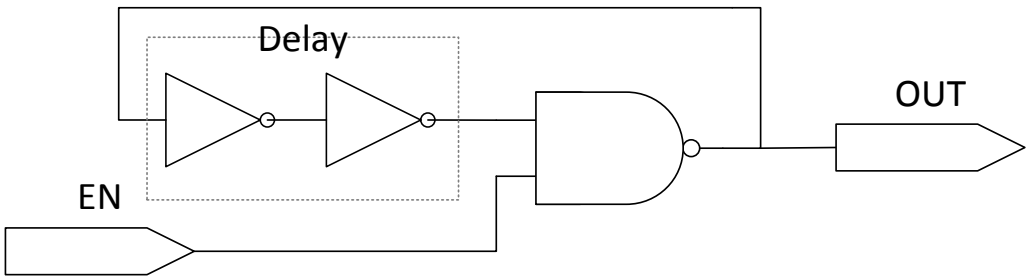

Figure 3.2 Example of a 3 inverter based Ring Oscillator with an Enable option

The output of the source of noise can be sampled with a clock signal, this will result in a serial data stream of oscillating values. Figure 3.3 shows 2500 bytes sampled from a ring oscillator with 1 Delay without post processing, these values were obtained by sampling the source using a D flip-flop and a clock signal of 100MHz. The graphical representation of the data was generated by piping the data from the ring oscillator through python script, the used script *Analyze_Data.py* is provided in Appendix A. The python script provided will visually represent the data and it will output statistics about the collected data such as:

- The average numerical value observed;
- The median of the observed values;
- The percentage of binary ones within the dataset;
- The extrema observed in the histogram representation;
- The Shannon entropy observed per byte, max 8bit.

The results of a default ring oscillator design as shown in Figure 3.2 are poor, the predominant values generated are 0x00 and 0xFF. In comparison to the data generated from /dev/random shown in Figure 3.1, unlike noise, a few values seem to be favored other values are never generated. The distribution is not uniform unlike is expected of the distribution of true random values.

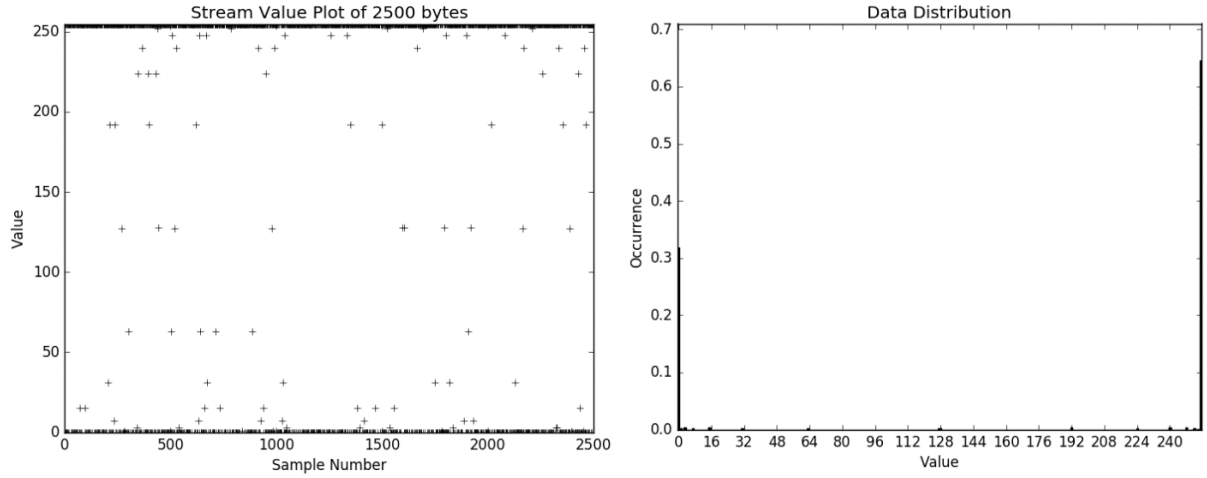

Figure 3.3 Graphical representation of 2500 bytes collected by the default Ring Oscillator described in Appendix A

The statistics shown in Table 3.1 are poor in comparison to the expected values from a random source. The ring oscillator has a bias toward generating ones. The Shannon entropy per generated byte is 1.247 bit out a maximum of 8 bit. Therefore, each byte generated contains only a little more entropy as could be achieved by flipping a coin.

Data analyzed: 2500 bytes	
Average	169,15
Median	255
Ones	66,31%
Distribution Min	0%
Distribution Max	64,56%
Shannon Entropy	1,247 bit

Table 3.1 Properties of the 2500 byte data set generated by the default Ring Oscillator

3.1.2 Fibonacci Ring Oscillator with static polynomial

A Fibonacci ring oscillator shares properties with a ring oscillator and with a LFSR with Fibonacci Architecture. The Verilog code to generate the FIRO shown in Figure 3.4 is included in Appendix C. Polynomials are programmed using short notation, e.g. $x^{16} + x^{15} + x^{13} + x^4 + 1$ will be the same as 0xD008.

Required signals:

- EN: Enable signal
- Poly: Polynomial, width of bus determined by parameter TAPS
- OUT: Serial output Data

Optional parameter:

- TAPS: Amount of available feedback taps, defaults to 16

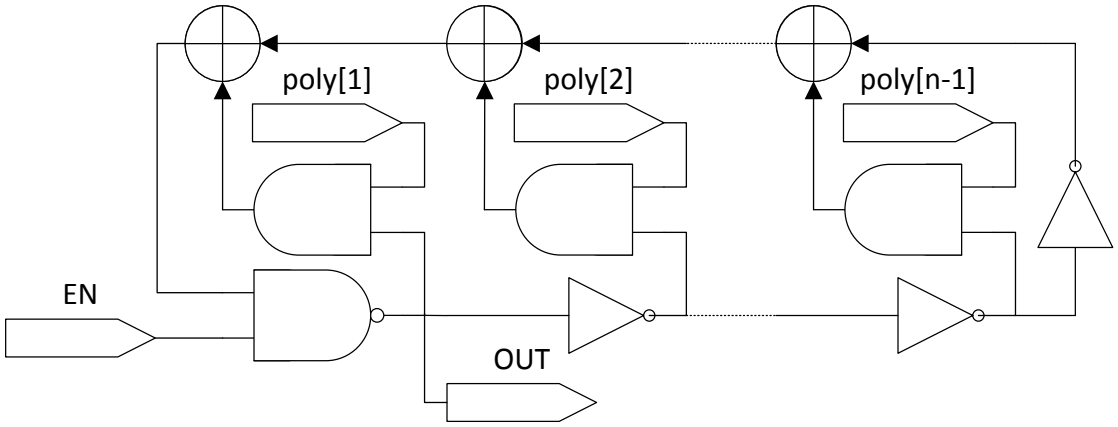

Figure 3.4 Ring Oscillator with Fibonacci architecture and on-the-fly changeable polynomial

A Fibonacci Ring Oscillator was tested using 16 possible taps and a static polynomial of 0x4218, this source was sampled using a D-flop and a clock signal of 100MHz. Enhancing a ring oscillator with a Fibonacci architecture causes the circuit to behave like a Pseudo random number generator and a True random number generator combined. While true random behavior is caused by transition delays in components, pseudo randomness is added by using a feedback polynomial.

The results of gathered data shown in Figure 3.5 are better than generated with the default ring oscillator although the observed results are still poor. The generator favors certain sets of numbers, while other values are never observed. The distribution is not uniform without post processing.

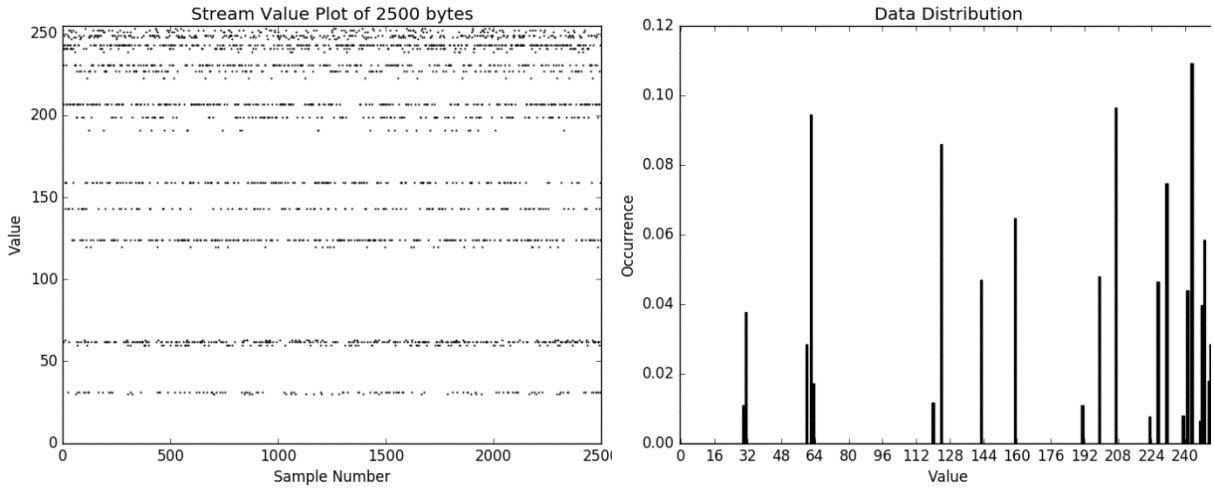

Figure 3.5 Graphical representation of 2500 bytes collected by the FIRO using the static polynomial 0x4218

The statistics in Table 3.2 show a strong deviation from the expected values. The generator could be observed to have a bias toward generating ones. The Shannon entropy calculated is rather poor with an entropy of 4.173 bits.

Data analyzed: 2500 bytes	
Average	178.55
Median	207
Ones	68.91%
Distribution Min	0%
Distribution Max	10.92%
Shannon Entropy	4.173 bit

Table 3.2 Properties of the 2500 byte data set generated by the FIRO with the static polynomial 0x4218

3.1.3 Fibonacci Ring Oscillator with dynamic polynomial

While a static polynomial could be used with the Fibonacci ring oscillator, the design permits changing the polynomial on the fly. This was tested using a linear feedback shift register of similar size to the Fibonacci Ring Oscillator, set up to produce a maximum length sequence. The LFSR was tied to change each clock cycle, which caused the polynomial to change every 10ns with a clock frequency of 100MHz. The used FIRO and LFSR had a size of 16 possible taps, and the LSFR chosen produced a maximum sequence length, which resulted in 65535 polynomials to be cycled through.

Figure 3.6 shows how the LFSR is connected to the FIRO, The complete design of the LFSR with Galois architecture is available in Appendix D

. The LFSR described in *LFSR_Galois.v* works synchronous with the clock signal, it will progress each clock cycle the module is enabled according to the set polynomial. When the LFSR is reset, the IV will be the data present in the LFSR memory, the IV cannot be all zeroes.

Required signals for the LFSR:

- CLK: Clock signal
- EN: Enable signal, when low, shifting of data is disabled
- RST: Resets output to IV
- IV: Initialization Vector, nonzero starting value
- POLY: Polynomial in short notation
- OUT: Output but

Optional parameter for the LFSR:

- TAPS: Amount of possible feedback taps, should be equal to the TAPS of FIRO

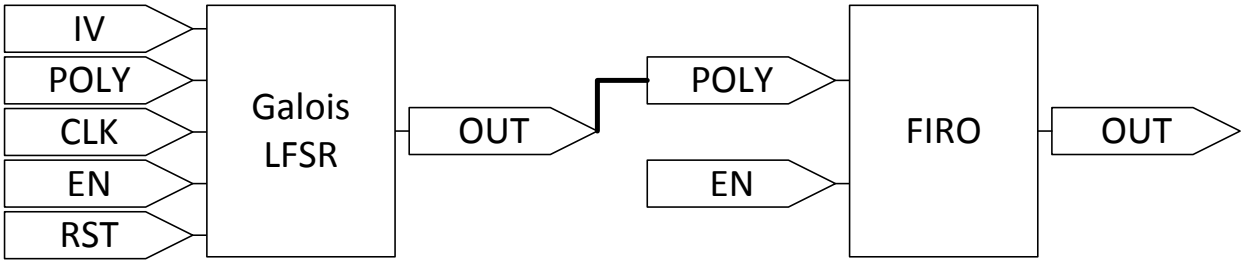

Figure 3.6 Partial block diagram of a FIRO with dynamic polynomial

Figure 3.7 shows the visual representation of 2500 bytes after applying a dynamic polynomial to the FIRO. The used LFSR has a Galois architecture and was set to use the polynomial 0xB401. Although the results shown are not great, certain values are still more prominent than others, some values are not generated within the span of 2500 bytes. The distribution appears to be more centered despite it not being uniformly distributed.

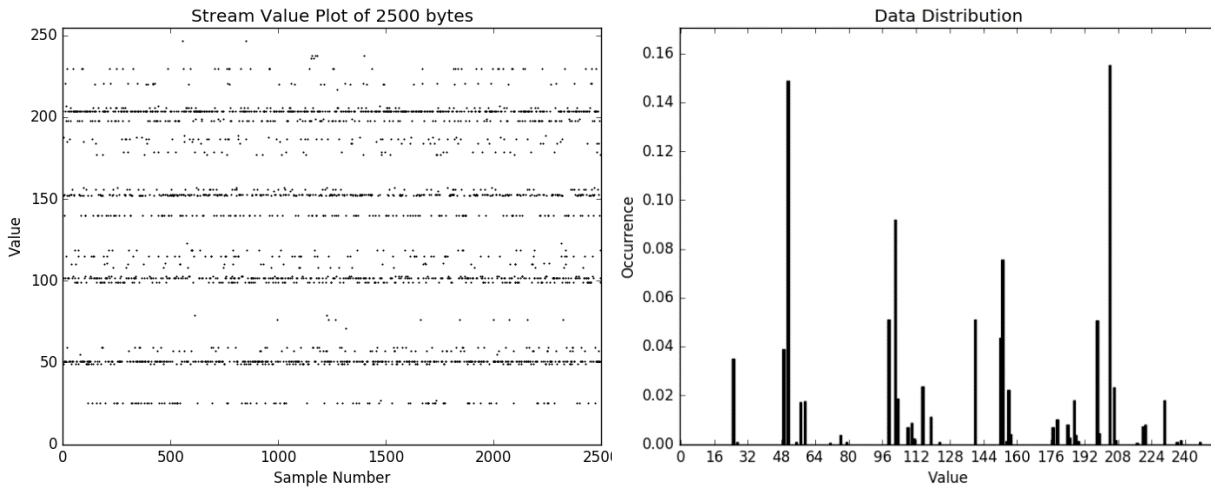

Figure 3.7 Graphical representation of 2500 bytes collected by the FIRO using a LFSR to generate a dynamic polynomial

The statistics from Table 3.3 confirm the centered distribution with the average and median value being closer to 128. There is no obvious binary bias with the amount of ones being about 50% of the generated data. The Shannon Entropy still is rather low with 4.267 bit per collected byte.

Data analyzed: 2500 bytes	
Average	130.30
Median	140
Ones	50.82%
Distribution Min	0%
Distribution Max	15.52%
Shannon Entropy	4.267 bit

Table 3.3 Properties of the 2500 byte data set generated by the FIRO with a dynamic polynomial

3.1.4 Galois Ring Oscillator with static polynomial

The Galois Ring Oscillator shares properties with a ring oscillator and with a LFSR with Galois architecture. The verilog code, *GARO.v*, describing the GARO design shown in Figure 3.8 is included as Appendix E. The tested noise source of type GARO, had a static polynomial of 0x4BFE and was sampled at 100MHz using a D flip-flop.

Required signals:

- EN: Enable signal
- Poly: Polynomial, width of bus determined by parameter TAPS
- OUT: Serial output signal

Optional parameter:

- TAPS: Amount of available feedback taps, defaults to 16

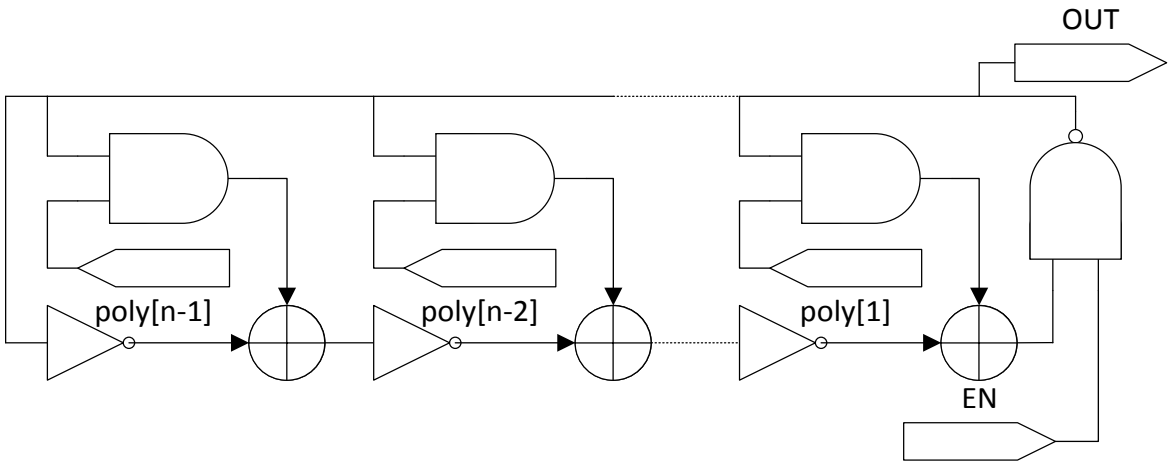

Figure 3.8 Ring Oscillator with Galois architecture and on-the-fly changeable polynomial

The gathered visual data of 2500 values as seen in Figure 3.9 are rather poor. The most prominent observed values are 0x00 and 0xFF. The value plot shows several values being favored. The observed distribution of the sample is not uniform.

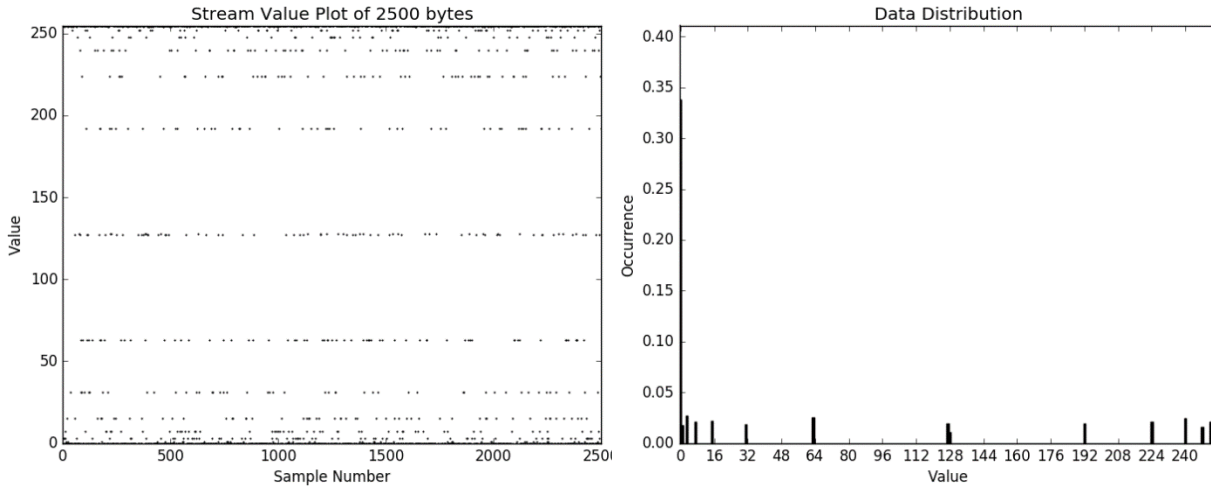

Figure 3.9 Graphical representation of 2500 bytes collected by the GARO using the static polynomial 0x4BFE

The gathered statistical values as shown in Table 3.4, display lacking entropy present in the sample. The generated data is slightly biased towards generating ones.

Data analyzed: 2500 bytes	
Average	131.84
Median	192
Ones	52.34%
Distribution Min	0%
Distribution Max	37.36%
Shannon Entropy	2.667 bit

Table 3.4 Properties of the 2500 byte data set generated by the GARO with static polynomial 0x4BFE

3.1.5 Galois Ring Oscillator with dynamic polynomial

The design of the Galois ring oscillator allows an on-the-fly changeable polynomial. This is achieved using the setup as shown in Figure 3.10, which combines a LFSR with Galois architecture with the GARO. The polynomial used for the LFSR was 0xB401. The Verilog code *LFSR_Galois.v*, describing the LFSR, van be found in Appendix D.

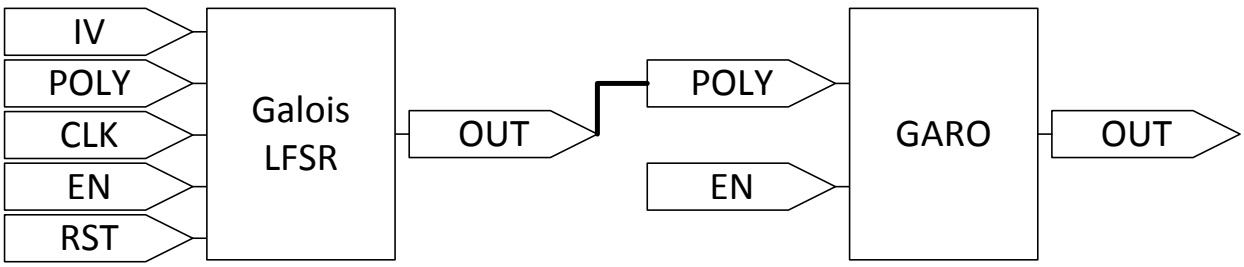

Figure 3.10 Partial block diagram of a GARO with dynamic polynomial

The results of the GARO are shown in Figure 3.11. More noise can be observed as with the GARO with static polynomial. The generator still favors some values, while other values are not observed in the sample. The distribution of the dataset is also not uniform.

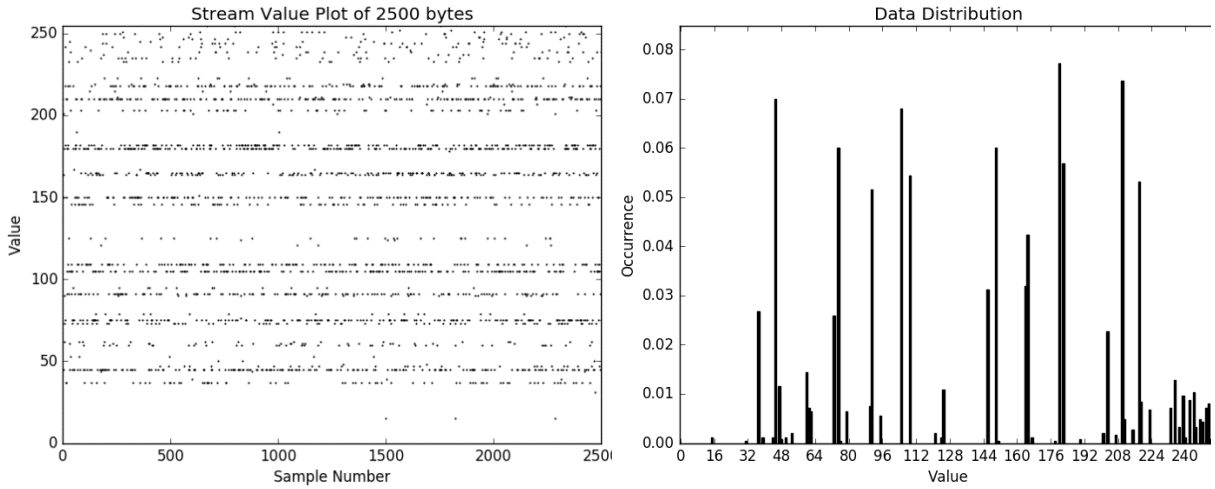

Figure 3.11 Graphical representation of 2500 bytes collected by the GARO using a LFSR to generate a dynamic polynomial

Given the statistics returned by the python script, shown in Table 3.5, a considerable increase of entropy can be observed in comparison to the GARO with static polynomial. The Shannon entropy is still low and a bias toward generating binary ones can be seen.

Data analyzed: 2500 bytes	
Average	142.45
Median	150
Ones	54.99%
Distribution Min	0%
Distribution Max	7.72%
Shannon Entropy	4.805 bit

Table 3.5 Properties of the 2500 byte data set generated by the GARO with a dynamic polynomial

3.1.6 Combined Ring Oscillator with Fibonacci and Galois architecture and dynamic polynomial

Combining the previously described dynamic FIRO and dynamic GARO with a XOR-operation, led to a dynamic FIGARO setup. The FIGARO set up with a static polynomial has been documented[12]. The design requires 2 independent polynomials to set up the pattern followed by the LFSR's, an independent IV which differs from 0 is also needed. The polynomial generating the pattern for the FIRO was set up to be 0xE946 and the polynomial generating the pattern for the GARO was set up to be 0xE976. The output of the dynamic FIGARO was sampled using a D flip-flop at 100MHz.

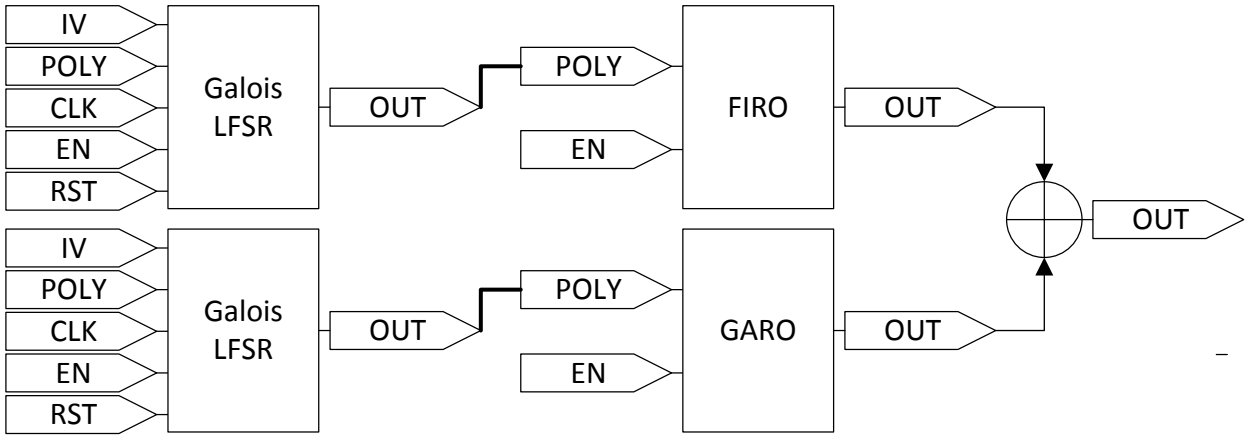

Figure 3.12 Partial block diagram of a dynamic FIGARO setup

The results of gathering entropy from the FIGARO noise source can be seen in Figure 3.13. The value plot resembles noise although some values appear to be less prominently visible. The distribution is not completely uniform. Nevertheless, all values seem to be represented within a 2500 byte sample.

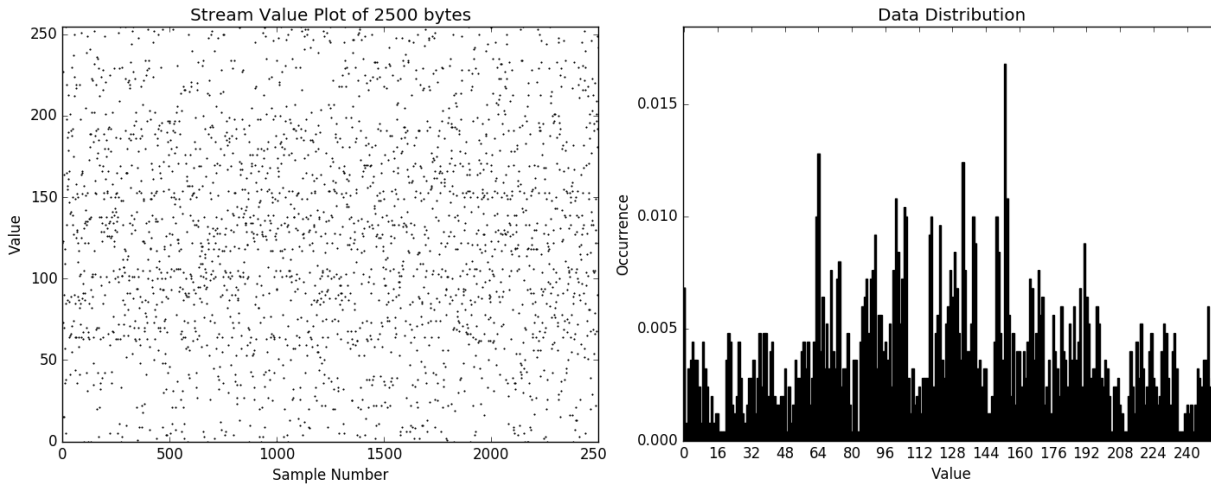

Figure 3.13 Graphical representation of 2500 bytes collected by the dynamic FIGARO

The statistical properties reported by the python script, as seen in Table 3.6, show a good amount of entropy present in the sample. The average and median values are comparable with the expected values of a random source. There is a balanced amount of binary ones against binary zero. There still is a deviation from the expected peeks of the calculated distribution.

Data analyzed: 2500 bytes	
Average	127.10
Median	127
Ones	49.67%
Distribution Min	0.04%
Distribution Max	1.68%
Shannon Entropy	7.711 bit

Table 3.6 Properties of the 2500 byte data set generated by the dynamic FIGARO

3.2 Sampling

The first step after generating noise, is to sample it with a stable and known clock signal. Sampling causes each clock cycle to generate 1 bit of data. Previous examples have a known clock of 100MHz and will therefore generate 100Mbps per parallel method of sampling.

3.2.1 D flip-flop

Sampling of the noise like signal is done with a D flip-flop, this device will store the binary value present at the input-pin D when the clock signal transitions from low to high (i.e. active on the rising edge of the clock signal). Since noise is sampled on the rising edge of a clock signal, each rising edge generates 1 random bit. A clock signal of 100MHz will sample 100Mbit per second. The provided verilog-code describes the simplest of the aforementioned D flip-flop.

Required signals:

- Clock: Data is sampled at the rising edge of the clock signal.
- D: Data input
- Q: Output

```
module dflop(Clock, D, Q);
    input      Clock;
    input      D    ;
    output reg Q = 0;
    always @(posedge clock)
        begin
            Q = D;
        end
endmodule
```

3.2.2 T flip-flop

A T flip-flop also uses a 1-bit memory like a D flip-flop. The T flip-flop does not store the bit provided at the T input, it will toggle the output when the clock signal has a rising edge and the T input is high. The provided verilog code describes the T flip-flop.

Required signals:

- Clock: Data is sampled at the rising edge of the clock signal.
- T: The output is toggled if a high signal is sampled
- Q: Output

```
module tflop(Clock, T, Q);
    input       Clock;
    input       T    ;
    output reg Q = 0;
    always @(posedge clock)
        begin
            Q = T?!Q:Q;
        end
endmodule
```

3.2.3 Shift Register

Shift Registers shift serial data into a memory. They consist out of daisy-chained D flip-flops which shift data on each rising edge of the provided clock signal. A shift register is used in the design to convert a serial stream of data, from the noise source, to parallel data. Figure 3.14 shows the block diagram of converting serial to parallel data.

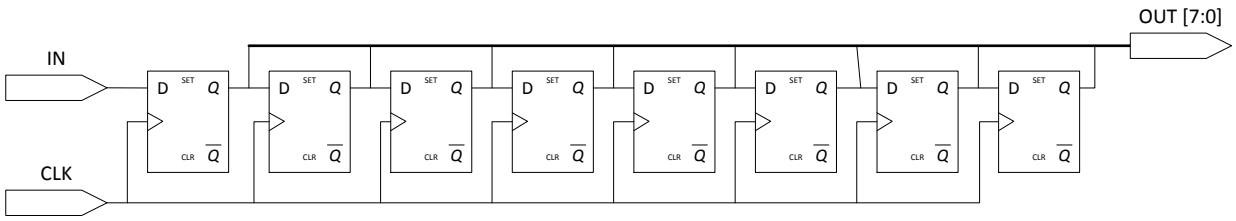

Figure 3.14 Block diagram of a serial to parallel shift register

The block diagram given in Figure 3.14 could be simplified using a model description in verilog HDL such as described below.

```
module shift_register(Clock, D, OUT);
    input            Clock ;
    input            IN    ;
    output reg [7:0] OUT   ;
    always @(posedge Clock)
        begin
            OUT = {IN,OUT[7:1]};
        end
endmodule
```

4 Post Processing

When the generated noise is sampled, some amount of bias may be present. The first samples of the different noise sources all show an amount of bias, some generators behaved with a binary bias, other generators favored certain values.
Post processing is a method to apply an algorithm to mix and compress the generated data, this will generally lead to an increase of entropy.

4.1 XOR Filter

A XOR filter will concatenate multiple inputs using exclusive-or logic elements. Each input is linked to a stream of random bits which may be biased. Each XOR-operation will increase the entropy of the output. A XOR-filter which can mix n amount of serial data streams is shown in Figure 4.1.

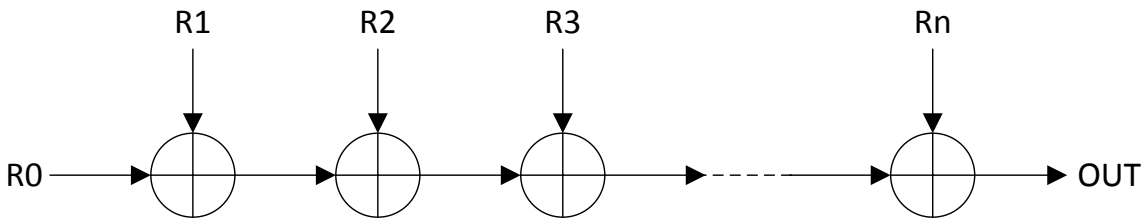

Figure 4.1 Block diagram of a XOR-filter

The block diagram shown in Figure 4.1 can be described with few lines of verilog HDL such as shown below.

Required signals:

- IN: Input bus, width determined by parameter N
- OUT: Serial output

Optional parameter:

- N: Amount of streams to merge

```
module XOR_filter(IN, OUT);
    parameter N=2;
    input [N-1:0] IN ;
    output        OUT;
    assign OUT =  ^IN;
endmodule
```

Use of Verilog module:

```
XOR_Filter #(N) XF({R0,R1, ... , RN},OUT);
```

4.2 Parity Filter

The parity filter as shown in Figure 4.2 mixes the previous bit with the input an operation which reduces bias. The verilog code, *PostProcessing.v*, which describes the parity filter can be found in Appendix F

Appendix F also elaborates on parallelizing the parity filter and improving it with a blocking filter.

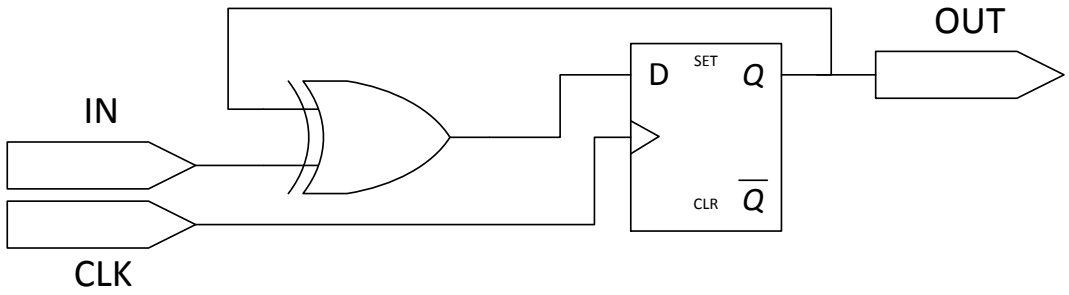

Figure 4.2 Serial Parity Filter

Several designs of parity filters were tested, multiple stages of filtering achieved different results. To test the necessary amount of filtering, a python script was created to simulate different stages of filtering on a saved data stream. The script used, *PP_Parity.py* can be found in Appendix G

4.2.1 Parity Filter Applied to the Default Ring Oscillator

The Default Ring Oscillator had poor performance as shown in the previously mentioned Figure 3.3. When the parity filter was run on the same data, the results shown in Figure 4.3.
The observed results are still not random, but an improvement is visible comparing to the unfiltered data. Although the filter dropped every other value, there seems to be an increase in correlation of the generated data. The observed statistical values were also improved. The Shannon entropy increased from 1.247 bits to 6.622 bits.

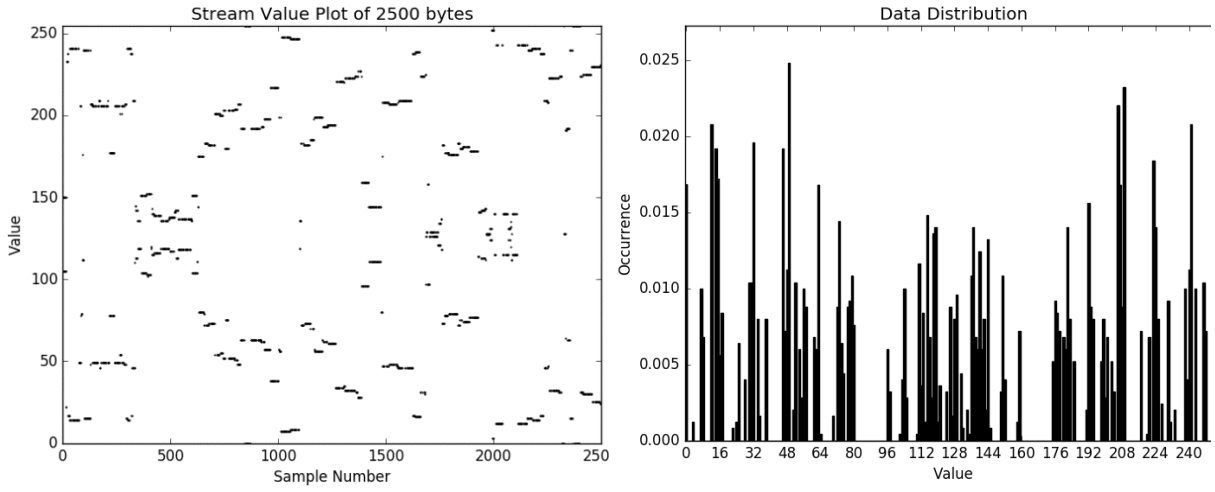

Figure 4.3 Results of the Default Ring Oscillator after 1 step of Post Processing

4.2.2 Parity Filter Applied to the FIRO

The parity filter was also applied to the Static FIRO, the results are shown in Figure 4.4. The value plot seems pattern free and the distribution of the gathered sample seems close to uniform. The statistical values reported by the script are close to the expected values of a true random sample. The bias towards binary ones is filtered away, the Shannon entropy has risen to 7.91 bit. A real statistical test is needed to determine randomness.

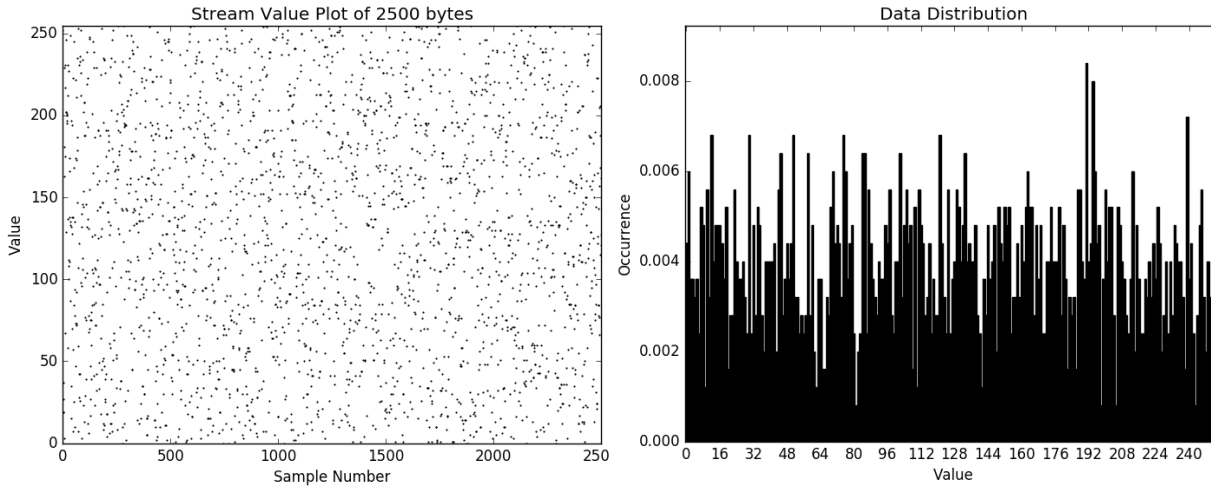

Figure 4.4 Results of the Static FIRO with 1 step of Post Processing

The filter was also applied to the Dynamic FIRO, the observations are shown in Figure 4.5. Similar observations can be made as with the Static FIRO. The reported statistics are also improved to the point they're indistinguishable from the expected values of a random source. The Shannon entropy increased to 7.92 bit.

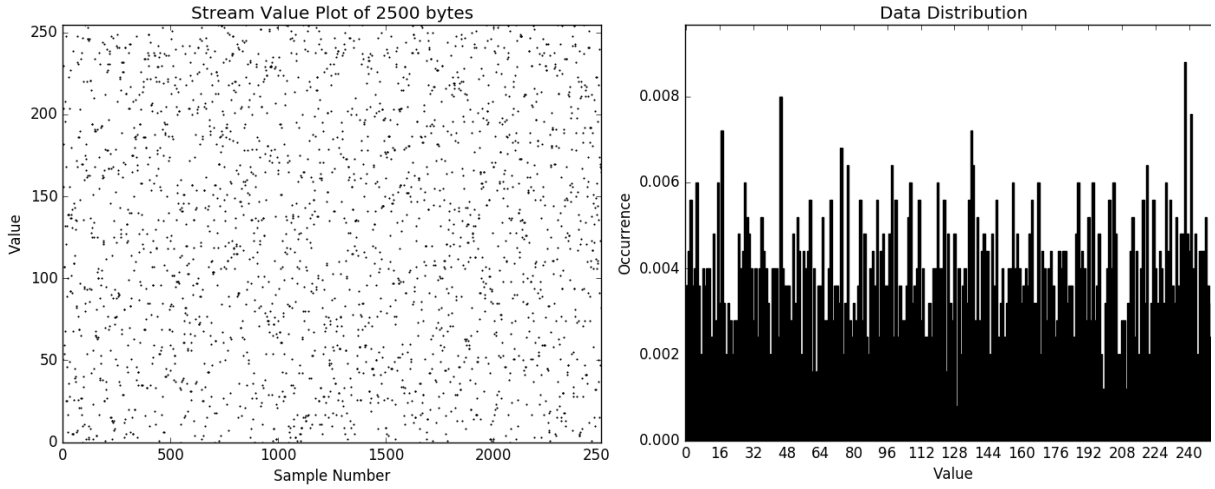

Figure 4.5 Results of the Dynamic FIRO with 1 step of Post Processing

4.2.3 Parity Filter Applied to the GARO

After applying 1 step of parity filtering, an improvement is visible to the Static GARO. Visual inspection of Figure 4.6 shows some recognizable patterns and the distribution of the sample does not appear uniformly distributed. The Shannon entropy has increased to 7.796 bit.

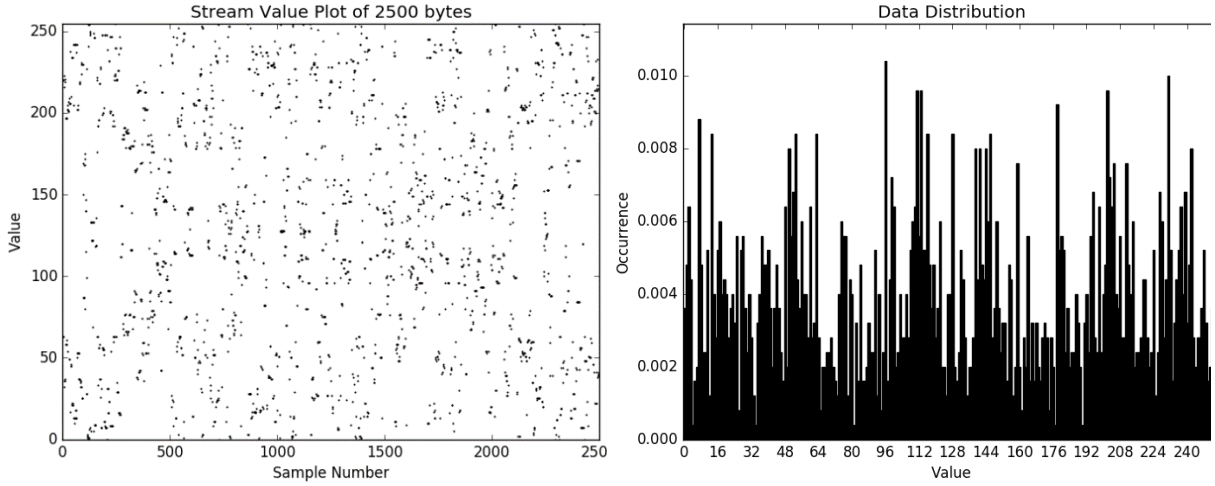

Figure 4.6 Results of the Static GARO with 1 step of Post Processing

The parity filter applied to the Dynamic GARO achieved the results shown in Figure 4.7. Compared to the results of the Static GARO, an improvement is seen. The values calculated within the python script show no obvious deviation from the expected values of a true random set. The Shannon entropy has increased to 7.928 bit.

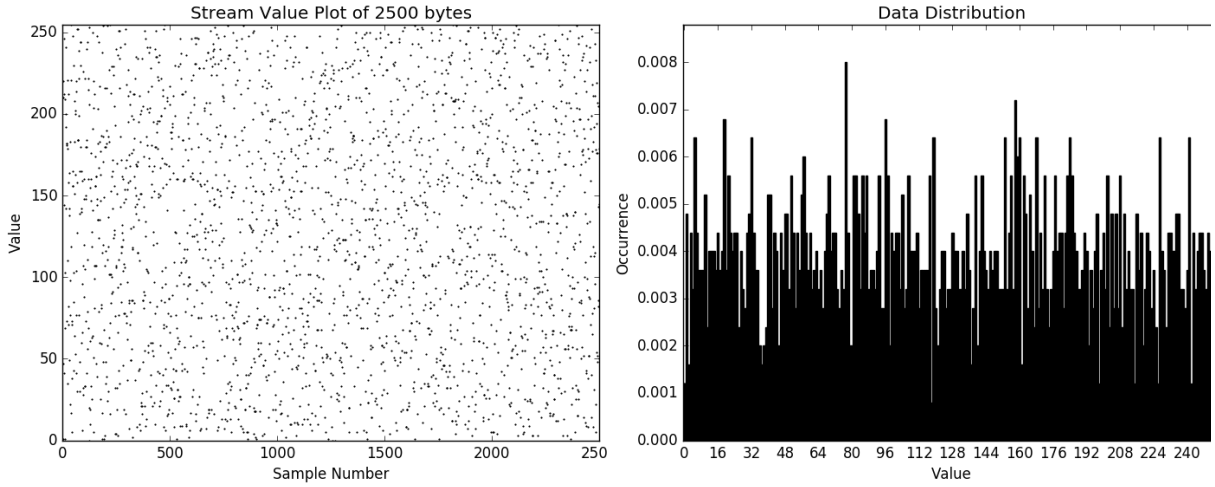

Figure 4.7 Results of the Dynamic GARO with 1 step of Post Processing

4.2.4 Parity Filter Applied to the FIGARO

The results shown in Figure 4.8 are of a single step of parity filter post processing after the Dynamic FIGARO. The calculated distribution is close to being uniform, no recognizable patterns can be seen in the noise plot and the reported statistical values are close to the expected values. The Shannon entropy of the sample is 7.928 bit per byte.

A statistical test suite was used to evaluate the generated data, such as FIPS140-2 and the NIST Statistical Test Suite.

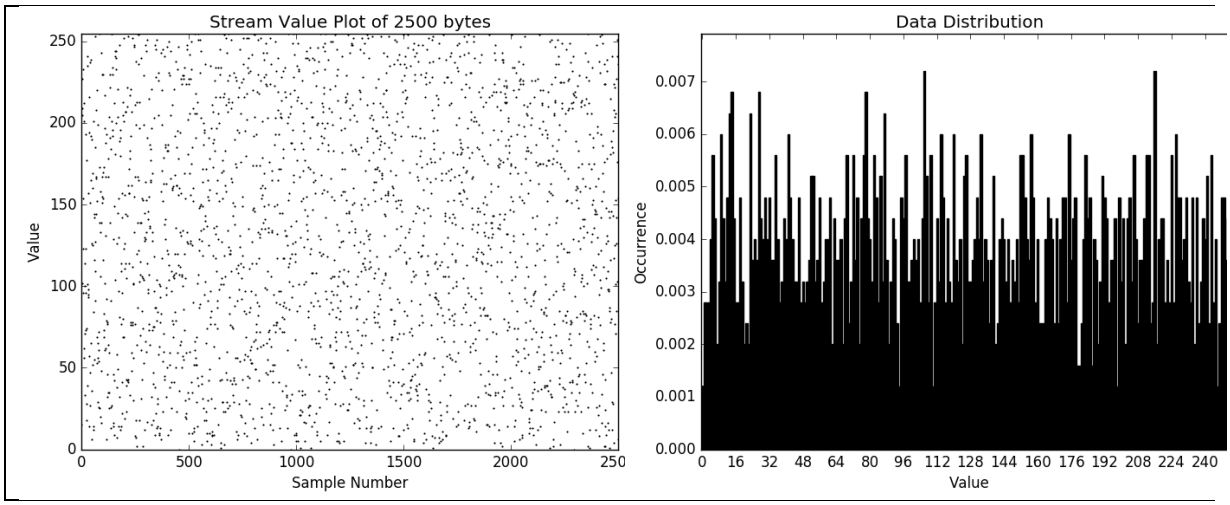

Figure 4.8 Results of the Dynamic FIGARO with 1 step of Post Processing

5 Communication using the Xilinx MicroBlaze

The Xilinx Platform Studio can work with the provided board support package, which is provided from AVNET store page[13]. Using XPS and the tutorial on LwIP for the Spartan-6 LX9 Microboard[14] a core capable of Ethernet communication could be generated.

The generated core includes:

- Xilinx MicroBlaze
- 64MB DDR RAM
- Ethernet MAC
- USB UART
- GPIO DIP Switches
- GPIO LEDs
- GPIO RNG Connection
- AXI4Lite switch fabric

The LwIP tutorial provides a Reference Design Block Diagram[14] (Figure 5.1) which shows a system is similar to the generated system.

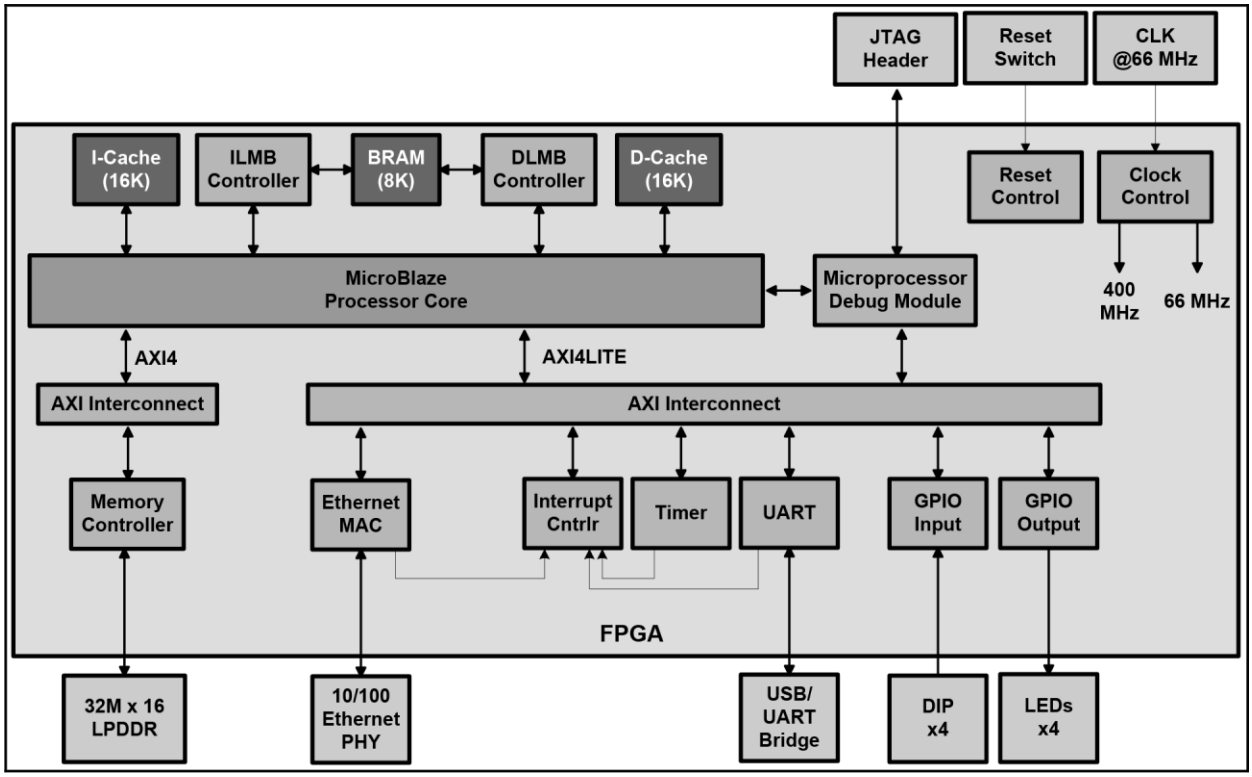

Figure 5.1 Xilinx XPS Block Diagram as provided from the LwIP turorial for the LX9[14]

The generated design has a relatively large footprint. It uses 30% of the Slice Registers, 77% of the Slice LUTs and occupies all available Slices.

5.1 From Generator to MicroBlaze

The implemented noise generator are 4 Dynamic FIGARO as shown in Figure 5.2. These are sampled using 2x4 serial to parallel shift registers, 4 of which operate on the rising edge of the clock signal, the remaining operate on the falling edge of the clock signal. A constant output is achieved by multiplexing the output of the 8 shift registers, causing each register to be sequentially sent as an output.

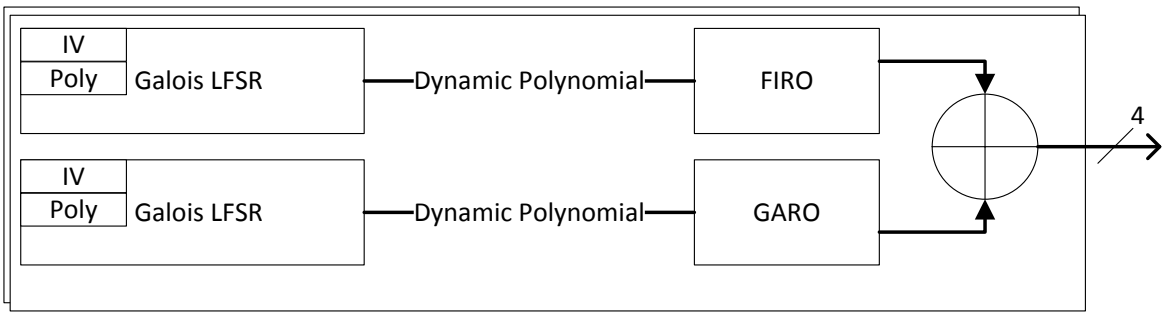

Figure 5.2 Implemented Noise Generator Block Diagram

The generator delivers parallel bits every clock cycle, the generator operates at 100MHz, the theoretical data rate is 800Mbps. This high bitrate is slightly biased and 1 step of post processing using a parity filter is added. Adding one step of post processing reduces the theoretical bitrate to 400Mbps. A general overview of the setup is displayed in Figure 5.3

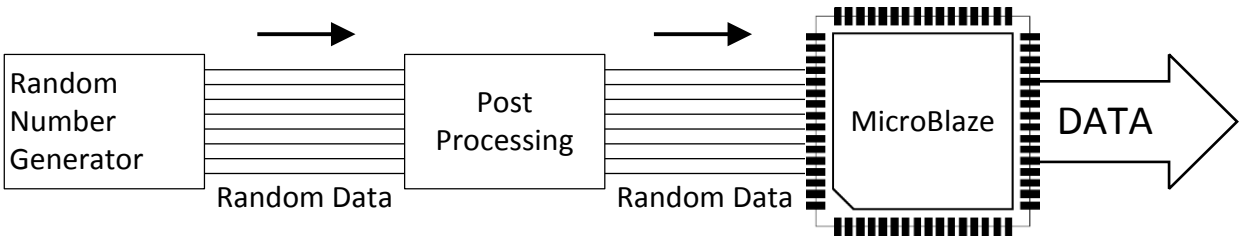

Figure 5.3 Generator to MicroBlaze Block Diagram

5.2 MicroBlaze Processor Core

The Xilinx MicroBlaze is programmed in C. Data from the FIGARO core can be loaded using pointers.
Since a GPIO module was used to load the data, a simple rule is implemented to read the next byte: The next byte should be different from the previous byte. The C-code of *RNG.c* is provided in Appendix H

The Monobit test is implemented as described in FIPS-140-2[11]. This test will count the amount of binary ones in a sequence of 20 000 bit. The test is successful if 9725 < N <10275. The Monobit test is based on the binomial distribution, where the rejection region is chosen to be 0.1% and the chance of N binary ones out of 20 000 bits is calculated, when the probability is lower than 0.1% the set of 20 000 bits is rejected.

5.3 From MicroBlaze to Computer

Transferring data is done using USB UART at 115200 baud. As displayed in Table 5.1, the Speed of /dev/random is rather low with the fastest time recorded at 1.8 byte per second. The lowers speed recorded with the serial connection was 11490 bytes per second.

Transfer Speed			
/dev/random 2500 bytes		/dev/ttyS00 100000 bytes	
[s]	[byte/s]	[s]	[byte/s]
1409,08	1,774207	8,67285	11530,24
1521,63	1,642975	8,67286	11530,22
1542,28	1,620977	8,67389	11528,85
1609,56	1,55322	8,67394	11528,79
1659,9	1,506115	8,67457	11527,95
1756,31	1,423439	8,6746	11527,91
1897,16	1,317759	8,67467	11527,82
1898,97	1,316503	8,67516	11527,16
1909,54	1,309216	8,68027	11520,38
1913,96	1,306192	8,68046	11520,13
1942,1	1,287266	8,68183	11518,31
1961,06	1,274821	8,6824	11517,55
2001,23	1,249232	8,68541	11513,56
2095,27	1,193164	8,68597	11512,82
2122,05	1,178106	8,70289	11490,44
2136,3	1,170248	8,70398	11489

Table 5.1 Comparative speed test /dev/random

6 Interfacing to /dev/random

Generated data can be used to enhance the Unix entropy pool. A few statistical tests were conducted to ensure the good quality of the generated data. The statistical tests conducted were conform FIPS140-2 and the statistical tests from the "NIST Statistical Test Suite"

6.1 Feeding into an entropy sink using the Random Number Generator Daemon

Entropy generated from hardware can be inserted into the Unix entropy pool using *rngd*[15], a part of rng-tools. The daemon is set up to evaluate the incoming data using FIPS140-2 test and if the tests are successful, add the entropy to the entropy pool.

To enhance the entropy pool the following command is used:

```
#> rngd –rng-device=/dev/RandomNumberGenerator
```

Options of the rngd can be changed by adjusting the settings file located at `/etc/default/rng-tools` on a Debian based system. Optionally the service rng-tools can be set up to run each time on boot, ensuring more available entropy for the system.

6.2 Achieved randomness according to a statistical tests.

A file was generated containing enough data to conduct 10 000 FIPS 140-2 tests, this resulted in a 25Mb file. This was repeated for several setups as random number generator. These files contained 1 step of post processing using a parity filter and the option of discarding data was disabled.

6.2.1 Results of FIPS 140-2

A wide range of noise sources was tested using the Avnet LX9 MicroBoard. Each generator generated 25Mb of data to be analyzed before and after post processing. The FIPS140-2 test is included in the rng-tools package and could be easily used to automate many tests. Conducting a test can be done with an easy command:

```
#> cat file_containing_random_data | rngtest
```

After running the *rngtest* command, a report is generated to show the number of failed tests. The report generated from the implemented hardware random number generator is displayed in Table 6.1

```
rngtest: bits received from input: 200008000
rngtest: FIPS 140-2 successes: 9993
rngtest: FIPS 140-2 failures: 7
rngtest: FIPS 140-2(2001-10-10) Monobit: 1
rngtest: FIPS 140-2(2001-10-10) Poker: 0
rngtest: FIPS 140-2(2001-10-10) Runs: 5
rngtest: FIPS 140-2(2001-10-10) Long run: 1
rngtest: FIPS 140-2(2001-10-10) Continuous run: 0
```

Table 6.1 Results of a FIPS 140-2 test conducted on 10 000 samples

6.2.2 Results of NIST Statistical Test Suite

The National Institute of Standards and Technology has made a Statistical Test Suite available to evaluate the output of a random number generator. After the file *sts-2.1.2.zip*[16] was downloaded, unpacked and compiled, the tests can be executed using the program *./assess*.

While the test suite is equipped with an interface to customize each battery of tests, it requires user interaction which can't be included in the parameters.

To test the hypothetical file FILENAME with a size of FILESIZE following command is issued:

```
#> ./assess FILESIZE
```

This will result in an interface where several questions need to be answered. The filled form to test FILENAME is given as Table 6.2.

```
          G E N E R A T O R    S E L E C T I O N


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

   Enter Choice: 0

               User Prescribed Input File: FILENAME
               S T A T I S T I C A L   T E S T S
               _________________________________

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

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

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

   Select Test (0 to continue): 0
   How many bitstreams?
   Input File Format: 10
    [0] ASCII - A sequence of ASCII 0's and 1's
    [1] Binary - Each byte in data file contains 8 bits of data

   Select input mode: 1
     Statistical Testing In Progress.........

READ ERROR:  Insufficient data in file.
     Statistical Testing Complete!!!!!!!!!!!!
```

Table 6.2 NIST Statistical Test Suite, Test Procedure

Fortunately, there is a workaround to automate user interaction. Keyboard combinations can be automated by piping the keystrokes into the program. The shell-script "*NIST_Auto.sh*" was created to automate the NIST Statistical Test Suite which code is displayed to Table 6.3.

The created script will automate user interaction, opening the correct file and backup the results.

```
#!/bin/bash
#NIST automated test
function gen_auto {
txt="0
$1
1
0
10
1
"
echo "$txt"
}

inputfile=$1
filesize=$(stat --printf="%s" $inputfile)
gen_auto $inputfile|./assess $filesize
cd ./experiments/AlgorithmTesting/
tar c . | xz > ../../$inputfile.tar.xz
echo "$inputfile done."
```

Table 6.3 NIST Statistical Test Suite Automation script

The reports generated for many proposed generator helped to reject several faulty designs of hardware generators. The report generated for the implemented design is included in Appendix I

7 Conclusion

Among many tested proposed designs as random number generators, a combination has been chosen which passes FIPS140-2 and the NIST Statistical Test Suite.

First, the implemented design utilizes 4 Dynamic FIGARO as a noise source, these FIGARO are sampled using 8 Shift Registers, half of which sample on the rising edge of the clock signal, the other 4 sample on the falling edge. Finally, the generator cycles through the contents of the shift registers to output a byte from a different FIGARO each clock cycle.

Then, the output of the generator is filtered using 8 parallel parity filters which removes any possible bias the noise source may have, the results of which can be observed in.

After filtering, the data is read into the Xillinx MicroBlaze softcore processor. The c-program reads and stores the random data in a memory of 2500 bytes.

Finally, a block of generated data is tested against the implemented Monobit test and if the test passes, the data is forwarded to the connected computer.

The data from the hardware random number generator is subsequently mixed into the Unix entropy pool to enhance /dev/random. Since entropy is added in a faster manner to the system, security applications relying on /dev/random can achieve better performance.

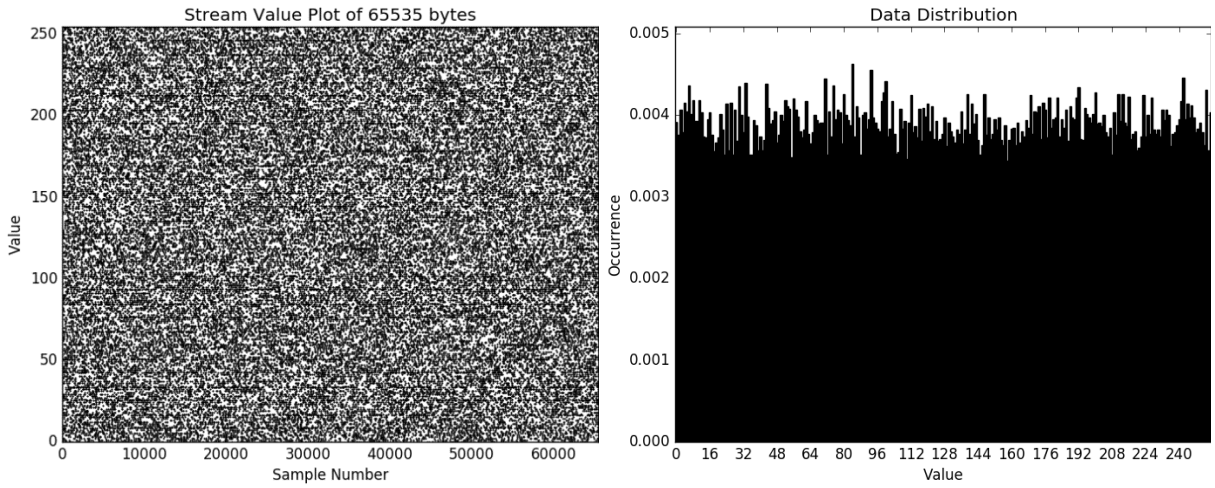

Figure 7.1 The Visual representation 65535 bytes generated from the implemented design. The reported Shannon entropy was 7.998 bit per byte

Future improvements include implementing more on-board tests, only one test is currently implemented. Further improvements are to reduce the footprint of the design, since all available slices are occupied. Finally, the throughput speed could be improved, sending the data over Ethernet may increase the throughput speed dramatically.

Bibliography

[1] N. Mentens, “A Testable True Random Number Generator for Linux Security Applications.” UHasselt, Hasselt, 2014.

[2] KU Leuven, “Computer Security and Industrial Cryptography - COSIC,” 2015. [Online]. Available: https://securewww.esat.kuleuven.be/cosic/. [Accessed: 30-Oct-2015].

[3] A. KHLim, “ARCO - ES&S,” 2015. [Online]. Available: http://acro.be/NL/ess.php?id=102. [Accessed: 30-Oct-2015].

[4] Acunetix, “Acunetix,” 2010. [Online]. Available: http://www.acunetix.com/blog/articles/statistics-from-the-top-1000000-websites/. [Accessed: 30-Oct-2015].

[5] S. Rukhin, A., Soto, J., Nechvatal, J., Smid M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, A., Vo, “Statistical Test Suite for Random and Pseudorando Number Generators for Cryptographic Applications,” *Spec. Publ. 800-22 Natl. Inst. Stand. Technol.*, no. April, 2010.

[6] Avnet, “Xilinx ® Spartan ® -6 FPGA LX9 MicroBoard User Guide,” p. 6, 2011.

[7] Avnet, “LwIP Applications Software 201 for the Spartan-6 LX9 MicroBoard,” p. 27, 2011.

[8] RASPBERRY FOUNDATION, “Download Raspbian for Raspberry Pi,” 2016. [Online]. Available: https://www.raspberrypi.org/downloads/raspbian/. [Accessed: 02-Jun-2016].

[9] G. Feng, K. K. Tzeng, and S. Member, “Algorithm for Multisequence Shift-Register Cyclic Codes,” vol. 37, no. 5, pp. 1274–1287, 1991.

[10] Xilinx_EDK, “Xilinx Constraints Guide,” *Constraints*, vol. 612, pp. 1–248, 2008.

[11] A. Region and D. Function, “FIPS 140 - 2(Change Notice 1) Random Number Tests —,” vol. 2, no. 1, 2003.

[12] J. Golić, “True Random Number Generation with Logic Gates Only,” *Security*, pp. 1–34, 2008.

[13] AVNET, “AES-S6MB-LX9-G.” [Online]. Available: https://products.avnet.com/shop/en/emea/kits-and-tools/development-kits/aes-s6mb-lx9-g-3074457345629535808. [Accessed: 12-Jul-2016].

[14] Avnet, “LwIP Applications Software 201 for the Spartan-6 LX9 MicroBoard,” 2013.

[15] M. S. Philipp Rumpf, Jeff Garzik, “rngd(8) - Linux man page.” .

[16] NIST, “NIST Statistical Test Suite.” [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/documents/sts-2.1.2.zip.

Appendix A

Python script intended to analyze the data generated from the multiple designed random number generators. The script will output a plot of the values in the given interval and the distribution of values in the same interval. The script also writes a few indicator values which could be compared to values expected with true random data.

Suggested use:

```
#> head -c [number of bytes] /location/of/device | ./Analyse_data.py [number of bytes]
```

Analyze_data.py

```
#!/usr/bin/python3
'''
* Analysis of generated data, output: plot, distribution, stats
* Author : Daniel Wietrzychowski
* License: GPL-2.0
'''
import sys
import matplotlib.pyplot as plt
import numpy as np
import math

def sys_args(): # Read parameters of program
    if len(sys.argv)>1:
        return int(sys.argv[1])
    else:
        print("Intended use: ", sys.argv[0], " bytes_to_process ")
        exit()

def percentage(val, limit):
    result = (val/float(limit))*100
    return int(result)

def amount_of_ones(data):
    if (data < 256):
        ctr = 0
        for i in range(0,8):
            ctr += 1 if (data & 1) else 0
            data >>= 1
        return ctr
    else: # Error condition
        return -1

def shannon_entropy(probability_list):
    sum  = 0.0
    entr = 0.0
    for elem in probability_list:
        sum += elem
        # Conditional sum, log(0) generates error
        entr += 0 if (elem == 0.0) else (elem * math.log2(elem))
    if (sum<0.99999):
        print("ERROR: sum probability_list:", sum)
    return (-entr)

def write_file(filename, datatowrite):
    try:
        with open(filename, 'w') as f:
            f.write(datatowrite)
    except IOError as e:
        print('open_files failed: %s' % e.strerror)
```

```
def main():
    bytes_to_process = sys_args()
    data = sys.stdin.buffer.read() #Read binary data stdin
    bytes_to_read = len(data)
    cycles = bytes_to_read//bytes_to_process

    divider = int(bytes_to_read // 100) # progress monitor
    # open virtual figure , 8x6 inches
    plt.figure(num=None, figsize=(8, 6), dpi=96, facecolor='w', edgecolor='k')
    for i in range(0,cycles * bytes_to_process, bytes_to_process):
        dataset = []
        num_ones = 0
        for j in range(0,bytes_to_process):
            current_byte = int(data[i+j])
            dataset.append(current_byte)
            num_ones += amount_of_ones(current_byte)
            if not ((i+j)%divider): #display of percentage
                sys.stdout.write("\rAnalysing: %02d%%" % \
                                 percentage(i,cycles * bytes_to_process))
                sys.stdout.flush()
        # Write plot data
        plt.plot(range(0,bytes_to_process),dataset,'ko', markersize=1)
        plt.axis([0, bytes_to_process, 0, 255])
        plt.xlabel('Sample Number')
        plt.ylabel('Value')
        plt.title('Stream Value Plot of {:d} bytes'.format(bytes_to_process))
        num = i//bytes_to_process
        plt.savefig(str(num) + " ValuePlot.png", bbox_inches='tight')
        plt.cla()
        plt.clf()
        # Statistics
        stat = "Amount of data analysed: {:.0f} bytes\n".format(bytes_to_process).
        stat += "Avg:\t{:.3f}\n".format(np.mean(dataset))
        stat += "Med:\t" + str(np.median(dataset)) + "\n"
        ones = num_ones /(bytes_to_process*8)
        stat += "Percentage ones:\t{:.2%}\n".format(ones)
        stat += "\nHistogram statistics:\n"
        # Distribution
        y,_,_ = plt.hist(dataset, range(0,256+1), histtype='bar',
                         rwidth = 1, color='k', normed=1, align='left')
        plt.axis([-0.5, 255.5,0,y.max() + y.max()*0.1])
        plt.xticks(list(filter(lambda x:x%16==0,range(0,256))))
        plt.xlabel('Value')
        plt.ylabel('Occurrence')
        plt.title('Data Distribution')
        plt.savefig(str(num) + " DistributionPlot.png", bbox_inches='tight')
        plt.cla()
        plt.clf()
        stat += "Distribution Min:\t{:.3%}\n".format(y.min())
        stat += "Distribution Max:\t{:.3%}\n".format(y.max())
        stat += "Per Byte Shannon Entropy:\t{:.3f}
bits\n".format(shannon_entropy(y))
        write_file(str(num) + " Stats.log", stat)
    print("\n\nDone")
# Run main program
main()
```

Appendix B

Verilog module to enable Simple Ring Oscillator in the Xilinx Hardware Development Kit

RO_PARAM.v

```
`timescale 1ns / 1ps
/**
* Parameterized Ring Oscillator Module
* Author: Daniel Wietrzychowski
* License: LGPL-2.1
**/

// USE: RO #(DELAY) RingOscillator(EN, out);
module RO(
        input EN,
        output out);
    parameter DELAY=1;

    wire [DELAY:0] interconnect;
    assign interconnect[0] = out;

    genvar i;
    generate
        for(i=1; i <= DELAY; i=i+1)
            begin
                (* S = "TRUE" *)delay WAIT(interconnect[i-1],interconnect[i]);
            end
    endgenerate

    NAND2  EN_inv (
        .I0(EN),
        .I1(interconnect[DELAY]),
        .O (out));
endmodule

module delay(input i, output o); // ouput = inv(inv(input)
    wire inv_wire0;

    INV inv_0(
        .I(i),
        .O(inv_wire0));

    INV inv_1(
        .I(inv_wire0),
        .O(o));
endmodule
```

Appendix C

Verilog module which produces a FIRO of length parameterized value TAPS.

FIRO.v

```
/**
* Parameterized Fibonacci Ring Oscillator Module
* Author : Daniel Wietrzychowski
* License: LGPL-2.1
**/
module FIRO(
        EN       ,
        POLYNOME ,
        O        );
    // Ring oscillator with feedback taps, Many-to-one logic
    parameter TAPS = 16;

    input               EN;
    input [TAPS-1:0]    POLYNOME;
    output              O;
    wire [TAPS-1:0]     WXOR,
                        WINV;
    assign O =          WINV[0];
    NAND2 EN_INV(
            .I0(WXOR[0]),
            .I1(EN),
            .O(WINV[0])
        );
    genvar i;
    generate
        for(i=1; i<TAPS; i=i+1)
            begin
                (* S = "TRUE" *)Block_Fibonacci BF(
                    .in_xor(WXOR[i])   ,.out_xor(WXOR[i-1]) ,
                    .in_inv(WINV[i-1]),.out_inv(WINV[i])    ,
                    .enable_tap(POLYNOME[i])
                );
            end
    endgenerate
    // Terminate and loopback
    assign WXOR[TAPS-1] = WINV[TAPS-1];
endmodule

module Block_Fibonacci(
        input  in_xor    ,
        input  in_inv    ,
        input  enable_tap,
        output out_xor   ,
        output out_inv   );
    wire flip;
    XOR2 X(
        .I0(in_xor),
        .I1(flip),
        .O (out_xor));
    AND2 A(
        .I0(enable_tap),
        .I1(in_inv    ),
        .O (flip      ));
    INV INVERT(
        .I(in_inv ),
        .O(out_inv));
endmodule
```

Appendix D

Parameterized LFSR module with Galois architecture

LFSR_Galois.v

```
/**
* Parameterized LFSR with Galois architecture
* Author : Daniel Wietrzychowski
* License: LGPL-2.1
**/
module Galois_LFSR(
        clock,
        enable,
        reset,
        polynome,
        IV,
        out);
    parameter TAPS = 16;
    input   clock,
            enable,
            reset;
    input  [TAPS-1:0]   polynome, // bit0 = 0!
                        IV;
    output [TAPS-1:0]   out;
    wire [TAPS:0] ic;
    genvar i;
    generate
        for(i=0; i<TAPS; i=i+1)
            begin
                Block_Galois_LFSR BG_LFSR(
                    .CLK(enable & clock),
                    .IV(IV[i])          ,
                    .feedback(i[0])     ,
                    .poly(polynome[i])  ,
                    .reset(reset)       ,
                    .in(ic[i+1])        ,
                    .out(ic[i])
                );
            end
    endgenerate
    assign ic[TAPS] = ic[0];
    assign out = ic[TAPS-1:0];
endmodule
module Block_Galois_LFSR(
        input CLK,
        input IV,
        input feedback,
        input poly,
        input reset,
        input in,
        output out);
    wire inv,flip, to_xor;
    XOR2 X(
        .I0(to_xor),
        .I1(flip),
        .O(out));
    AND2 A(
        .I0(poly),
        .I1(feedback),
        .O(flip));
    LFSR_D_FLOP MEM(
        .CLK(CLK),
        .D(in),
        .RESET(reset),
        .IV(IV),
        .Q(to_xor));
endmodule
```

```
module LFSR_D_FLOP(CLK, D, RESET, IV, Q);
    input CLK,D,RESET,IV;
    output reg Q;
    always @(posedge CLK)
        begin
            if(RESET)
                begin
                    Q = IV;
                end
            else
                begin
                    Q = D;
                end
        end
endmodule
```

Appendix E

Verilog module which produces a GARO of length parameterized value TAPS.

GARO.v

```
/**
* Parameterized Galois Ring Oscillator Module
* Author : Daniel Wietrzychowski
* License: LGPL-2.1
**/
module GARO(
        EN       ,
        POLYNOME ,
        O        );
    // Ring oscillator with feedback taps, One-to-many logic
    parameter TAPS = 16;
    input               EN;
    input [TAPS-1:0]    POLYNOME;
    output              O;
    wire [TAPS-1:0]     WXOR;
    wire                feedback;
    assign O =          feedback;
    NAND2 EN_INV(
            .I0(WXOR[0] ),
            .I1(EN      ),
            .O (feedback)
        );
    genvar i;
    generate
        for(i=1; i<TAPS; i=i+1)
            begin
                (* S = "TRUE" *)Block_Galois BG(
                    .in_tap(feedback)     ,
                    .in_inv(WXOR[i])      ,
                    .out_xor(WXOR[i-1])   ,
                    .enable_tap(POLYNOME[i])
                );
            end
    endgenerate
    // Terminate and loopback
    assign WXOR[TAPS-1] = feedback;
endmodule

module Block_Galois(
        input in_tap,
        input in_inv,
        input enable_tap,
        output out_xor);
    wire inv,flip;
    XOR2 X(
        .I0(inv),
        .I1(flip),
        .O(out_xor));
    AND2 A(
        .I0(enable_tap),
        .I1(in_tap),
        .O(flip));
    INV INVERT(
        .I(in_inv),
        .O(inv));
endmodule
```

Appendix F

Verilog module responsible for post processing the random data

PostProcessing.v

```
`timescale 1ns / 1ps
/**
* Parallel Post Processing using blocking parity filter for an 8-bit bus
* Author : Daniel Wietrzychowski
* License: LGPL-2.1
**/

module PP_parity_parallel(
                        input clock,
              input enable, // inverted
              input reset,
                        input  [7:0] in,
                        output [7:0] out,
              output  block);

    wire [7:0] parity;

    genvar i;
    generate
        for(i=0;i<8;i=i+1)
            begin : parity_filter
                PP_parity_bit (clock,in[i],parity[i]);
            end
    endgenerate

    BlockingFilter
        #(  .N(1),
            .M(1))    BlockFilter(  clock,
                                    enable,
                                    reset,
                                    parity,
                                    out,
                                    block);
endmodule

module PP_parity_bit(clock,in,out);
        input clock, in;
        output reg out;
        reg mem = 1'b0;
always @(posedge clock)
        begin
                mem = in ^ mem;
                out = mem;
        end
endmodule
```

```
module BlockingFilter(
        clock,
        enable,
        reset,
        in,
        out,
        block);

    parameter       N=1;
    parameter       M=1;      // drop N /(M+1) packets
    input             clock ,
                      reset ,
                      enable;
    input       [7:0] in    ;
    output reg  [7:0] out   ;
    output reg        block ;

    reg [3:0] counter=4'b0;   // 1 < N < 15

    always @(posedge clock)
        begin
            if(reset)
                begin
                    out = 8'b0;
                    block = 1'b1;
                    counter = 4'b0;
                end
            if(!enable)
                begin
                    block   = counter < N ;
                    out     = block ?    // if Block (every N out of M+1 values)
                                out :    // TRUE:  don't change value
                                in  ;    // FALSE: pass-through input

                    counter = (counter < M)*(counter + 1); // loop counter 0->N
                end
            else
                begin
                    block = 1'b1;
                end
        end
endmodule
```

Appendix G

Simulation of parity filter on data.

Use of *PP_Parity.py* assuming a Unix-like operating system:

```
#> cat inputData | ./PP_Parity [amount of filters] > outputData
```

PP_Parity.py

```
#!/usr/bin/python3
'''
* PostProcessing of stdin to stdout using parityfilter
* Author : Daniel Wietrzychowski
* License: GPL-2.0
'''
import sys
class Parallel_Parity_Filter:
    'Use Filter'
    def __init__(self):
        self.prev = 0x00
    def filter(self, val):
        tmp = (int(val) ^ int(self.prev))
        self.prev = tmp
        return tmp

if len(sys.argv)>1:
    num_filters = int(sys.argv[1])
    filters = [Parallel_Parity_Filter() for i in range(num_filters)]
else:
    print("Intended use: ", sys.argv[0], " #filters")
    exit()
data = sys.stdin.buffer.read() #Read binary data stdin
bytes_to_read = len(data)
for i in range(0, bytes_to_read):
    active_byte = data[i]
    for j in range(len(filters)):
        active_byte = filters[j].filter(active_byte)
    processed_byte = (active_byte).to_bytes(1,byteorder='big')
    if not(i % (2*num_filters)):
        # push to stdout
        sys.stdout.buffer.write(processed_byte)
```

Appendix H

The C-code to transfer generated data to a computer.

RNG.c

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"

#define MEMSIZE 2500
#define MONOBIT_THRESHOLD 275

void print(char *str);

char count_ones(char byte){
    char c=0;
    c += byte & 1;
    c += (byte >> 1) & 1;
    c += (byte >> 1) & 1;
    c += (byte >> 1) & 1;
    c += (byte >> 1) & 1;
    c += (byte >> 1) & 1;
    c += (byte >> 1) & 1;
    c += (byte >> 1) & 1;
    return c;
}

int main(){
    init_platform();
    // Pointers to hardware: DIP, LED, Generator
          int   * ptr_led = (int *)XPAR_LEDS_4BITS_BASEADDR            ;
    const int   * ptr_dip = (const int *)XPAR_DIP_SWITCHES_4BITS_BASEADDR    ;
    const int   * ptr_rng = (const int *)XPAR_AXI_INTERCONNECT_8BITS_BASEADDR;
    int prev_data = 0;
    int rng_data = 0;
    int i;
    char tests;
    int monobit_sum;
    char mem[MEMSIZE];
    while(1){
        for( i = 0; i < MEMSIZE; i++){
            while(prev_data == rng_data){
                        (*ptr_led) = *ptr_dip; // (*out) = *in
                        rng_data   = *ptr_rng & 0xFF; // get data from pins, truncate to 8bits
                        mem[i]=rng_data;
            }
            prev_data = rng_data;
        }
        char tests = 0;
        //monobit test
        monobit_sum =0;
        for( i = 0; i < MEMSIZE; i++){
            monobit_sum += count_ones(mem[i]);
        }
        char tests |= ((monobit_sum < (10000+MONOBIT_THRESHOLD))?
                            (monobit_sum > (10000-MONOBIT_THRESHOLD)) : 0);
        if( tests == 1){
            for( i = 0; i < MEMSIZE; i++){
                xil_printf("%c", mem[i]);
            }
        }
    }
    cleanup_platform();
    return 0;
}
```

Appendix I

Included is the successful report generated by the NIST Statistical Test Suite for Random number generators. The conducted tests are described in “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications”[5]

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <Reduced TFlop PP1Block>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  1   4   1   0   0   0   0   1   1   2  0.122325     10/10      Frequency
  1   0   0   0   1   0   1   5   2   0  0.008879     10/10      BlockFrequency
  2   0   3   0   1   1   0   0   2   1  0.350485     10/10      CumulativeSums
  1   1   4   0   0   0   0   1   2   1  0.122325     10/10      CumulativeSums
  0   4   1   1   1   0   2   1   0   0  0.122325     10/10      Runs
  1   0   2   0   4   1   1   1   0   0  0.122325     10/10      LongestRun
  0   0   0   0   2   5   0   0   2   1  0.004301     10/10      Rank
  2   0   0   0   1   1   5   1   0   0  0.008879     10/10      FFT
  1   0   0   2   1   2   0   0   4   0  0.066882     10/10      NonOverlappingTemplate
  0   1   2   2   0   0   0   1   4   0  0.066882     10/10      NonOverlappingTemplate
  1   1   2   0   1   0   1   1   3   0  0.534146     10/10      NonOverlappingTemplate
  2   0   0   0   0   1   1   0   4   2  0.066882     10/10      NonOverlappingTemplate
  0   0   2   0   0   1   0   2   4   1  0.066882     10/10      NonOverlappingTemplate
  2   1   0   0   2   1   0   0   4   0  0.066882     10/10      NonOverlappingTemplate
  1   0   1   0   1   1   0   3   3   0  0.213309     10/10      NonOverlappingTemplate
  0   0   0   0   1   4   0   1   4   0  0.004301     10/10      NonOverlappingTemplate
  0   0   4   0   0   1   0   1   4   0  0.004301     10/10      NonOverlappingTemplate
  1   1   0   1   0   0   1   0   6   0  0.000439      9/10      NonOverlappingTemplate
  1   1   1   0   1   1   0   0   4   1  0.213309     10/10      NonOverlappingTemplate
  2   0   1   0   0   1   1   1   4   0  0.122325     10/10      NonOverlappingTemplate
  1   0   0   0   2   0   2   1   3   1  0.350485     10/10      NonOverlappingTemplate
  0   2   1   0   3   0   1   0   3   0  0.122325     10/10      NonOverlappingTemplate
  1   0   0   2   0   0   1   2   3   1  0.350485     10/10      NonOverlappingTemplate
  1   0   0   1   0   0   0   0   4   4  0.004301     10/10      NonOverlappingTemplate
  0   2   2   1   1   0   0   0   4   0  0.066882     10/10      NonOverlappingTemplate
  0   1   1   0   0   0   3   1   4   0  0.035174     10/10      NonOverlappingTemplate
  1   0   0   2   2   0   1   0   4   0  0.066882     10/10      NonOverlappingTemplate
  1   0   2   0   0   1   1   0   5   0  0.008879     10/10      NonOverlappingTemplate
  1   1   1   0   0   1   1   1   3   1  0.739918     10/10      NonOverlappingTemplate
  3   1   0   0   0   0   1   0   5   0  0.002043     10/10      NonOverlappingTemplate
  1   2   1   1   0   0   1   1   3   0  0.534146     10/10      NonOverlappingTemplate
  1   1   1   1   1   0   0   0   5   0  0.017912     10/10      NonOverlappingTemplate
  0   0   0   1   2   1   1   0   5   0  0.008879     10/10      NonOverlappingTemplate
  0   1   1   2   0   0   1   0   4   1  0.122325     10/10      NonOverlappingTemplate
  1   0   1   0   0   0   0   3   4   1  0.035174     10/10      NonOverlappingTemplate
  0   2   0   0   1   2   0   2   3   0  0.213309     10/10      NonOverlappingTemplate
  0   1   0   0   0   0   1   0   5   3  0.002043     10/10      NonOverlappingTemplate
  1   0   0   2   0   0   1   0   3   3  0.122325      9/10      NonOverlappingTemplate
  2   0   1   0   2   0   1   0   4   0  0.066882     10/10      NonOverlappingTemplate
  0   0   0   2   2   0   1   0   4   1  0.066882     10/10      NonOverlappingTemplate
  0   0   1   2   1   2   0   0   3   1  0.350485     10/10      NonOverlappingTemplate
  1   1   1   0   0   2   0   0   4   1  0.122325     10/10      NonOverlappingTemplate
  1   0   3   1   1   0   0   1   3   0  0.213309     10/10      NonOverlappingTemplate
  2   0   1   1   1   0   0   0   4   1  0.122325     10/10      NonOverlappingTemplate
  1   0   0   2   0   2   0   0   4   1  0.066882     10/10      NonOverlappingTemplate
  0   1   0   0   2   1   0   1   5   0  0.008879     10/10      NonOverlappingTemplate
  2   1   1   0   0   0   2   0   4   0  0.066882      9/10      NonOverlappingTemplate
  1   2   0   0   2   0   0   0   5   0  0.004301     10/10      NonOverlappingTemplate
  0   0   0   0   0   2   1   1   4   2  0.066882     10/10      NonOverlappingTemplate
  1   1   1   0   1   1   1   1   3   0  0.739918     10/10      NonOverlappingTemplate
  1   0   1   1   0   0   1   1   3   2  0.534146     10/10      NonOverlappingTemplate
  1   2   3   0   0   1   0   0   3   0  0.122325     10/10      NonOverlappingTemplate
  0   0   0   1   1   0   2   2   3   1  0.350485     10/10      NonOverlappingTemplate
  0   0   0   0   1   2   1   1   4   1  0.122325     10/10      NonOverlappingTemplate
  1   0   1   1   2   1   0   0   3   1  0.534146     10/10      NonOverlappingTemplate
  1   3   1   1   0   1   0   0   3   0  0.213309     10/10      NonOverlappingTemplate
  1   0   0   3   0   0   0   1   3   2  0.122325      9/10      NonOverlappingTemplate
  0   1   1   0   0   0   0   2   5   1  0.008879     10/10      NonOverlappingTemplate
  0   0   0   1   1   2   2   0   3   1  0.350485     10/10      NonOverlappingTemplate
  0   1   2   0   0   2   0   0   4   1  0.066882     10/10      NonOverlappingTemplate
  0   2   1   0   3   0   1   0   3   0  0.122325     10/10      NonOverlappingTemplate
  1   2   0   0   0   1   0   1   5   0  0.008879     10/10      NonOverlappingTemplate
  0   2   2   0   0   1   0   1   4   0  0.066882     10/10      NonOverlappingTemplate
```

1	0	2	0	3	0	0	1	3	0	0.122325	10/10	NonOverlappingTemplate
2	3	0	0	0	2	0	0	3	0	0.066882	10/10	NonOverlappingTemplate
1	0	0	0	0	0	3	1	3	2	0.122325	10/10	NonOverlappingTemplate
0	1	0	1	0	3	0	1	3	1	0.213309	10/10	NonOverlappingTemplate
0	0	1	1	0	2	1	0	4	1	0.122325	10/10	NonOverlappingTemplate
0	0	0	2	1	0	0	1	6	0	0.000199	10/10	NonOverlappingTemplate
1	0	2	1	1	0	1	0	3	1	0.534146	10/10	NonOverlappingTemplate
0	0	1	3	0	1	0	0	4	1	0.035174	10/10	NonOverlappingTemplate
0	1	3	0	0	0	0	2	4	0	0.017912	10/10	NonOverlappingTemplate
0	0	0	2	0	1	1	0	4	2	0.066882	10/10	NonOverlappingTemplate
1	0	1	0	4	0	0	0	3	1	0.035174	10/10	NonOverlappingTemplate
1	0	2	1	0	0	0	1	4	1	0.122325	10/10	NonOverlappingTemplate
1	0	0	0	1	1	2	0	4	1	0.122325	10/10	NonOverlappingTemplate
2	0	0	1	1	1	0	0	3	2	0.350485	10/10	NonOverlappingTemplate
1	0	0	0	2	1	1	0	4	1	0.122325	10/10	NonOverlappingTemplate
2	0	0	0	0	0	1	2	3	2	0.213309	10/10	NonOverlappingTemplate
1	0	2	0	1	0	0	2	4	0	0.066882	10/10	NonOverlappingTemplate
2	0	0	0	3	0	0	0	4	1	0.017912	10/10	NonOverlappingTemplate
0	0	1	1	1	0	2	0	5	0	0.008879	10/10	NonOverlappingTemplate
1	0	0	2	1	2	0	0	4	0	0.066882	10/10	NonOverlappingTemplate
0	1	2	0	2	1	0	0	4	0	0.066882	10/10	NonOverlappingTemplate
2	0	1	0	0	1	1	2	3	0	0.350485	10/10	NonOverlappingTemplate
0	0	2	1	1	2	0	0	3	1	0.350485	10/10	NonOverlappingTemplate
0	0	0	1	1	0	1	1	5	1	0.017912	10/10	NonOverlappingTemplate
0	1	0	0	0	1	1	1	4	2	0.122325	10/10	NonOverlappingTemplate
1	3	0	0	0	0	0	1	5	0	0.002043	10/10	NonOverlappingTemplate
0	1	0	2	0	1	0	2	4	0	0.066882	10/10	NonOverlappingTemplate
0	1	1	0	1	1	1	0	4	1	0.213309	10/10	NonOverlappingTemplate
0	1	1	2	0	0	0	1	3	2	0.350485	10/10	NonOverlappingTemplate
2	2	0	0	0	0	0	2	3	1	0.213309	10/10	NonOverlappingTemplate
1	0	1	2	1	0	0	0	5	0	0.008879	10/10	NonOverlappingTemplate
2	0	0	1	1	0	0	1	5	0	0.008879	10/10	NonOverlappingTemplate
1	0	3	0	1	1	0	0	3	1	0.213309	9/10	NonOverlappingTemplate
1	1	1	1	0	1	1	0	3	1	0.739918	10/10	NonOverlappingTemplate
1	0	2	1	0	1	1	0	4	0	0.122325	10/10	NonOverlappingTemplate
1	1	2	0	0	0	0	2	3	1	0.350485	10/10	NonOverlappingTemplate
1	1	1	1	0	0	0	0	3	3	0.213309	10/10	NonOverlappingTemplate
1	2	0	1	0	0	0	1	5	0	0.008879	10/10	NonOverlappingTemplate
1	0	2	0	0	2	0	1	4	0	0.066882	9/10	NonOverlappingTemplate
2	1	0	1	1	1	0	0	3	1	0.534146	10/10	NonOverlappingTemplate
1	1	0	1	2	0	0	1	3	1	0.534146	10/10	NonOverlappingTemplate
2	1	0	1	0	1	0	1	4	0	0.122325	10/10	NonOverlappingTemplate
0	0	0	0	1	2	0	1	4	2	0.066882	10/10	NonOverlappingTemplate
1	2	0	1	1	0	0	1	3	1	0.534146	10/10	NonOverlappingTemplate
3	0	1	0	0	1	0	2	3	0	0.122325	10/10	NonOverlappingTemplate
0	0	1	1	2	0	1	0	4	1	0.122325	10/10	NonOverlappingTemplate
0	0	0	1	3	1	0	0	4	1	0.035174	10/10	NonOverlappingTemplate
0	1	0	0	0	0	0	4	5	0	0.000199	10/10	NonOverlappingTemplate
0	1	2	1	1	0	1	0	4	0	0.122325	10/10	NonOverlappingTemplate
0	0	0	1	3	0	3	0	3	0	0.035174	10/10	NonOverlappingTemplate
2	0	1	0	0	0	0	1	5	1	0.008879	10/10	NonOverlappingTemplate
0	2	2	0	0	1	0	0	4	1	0.066882	10/10	NonOverlappingTemplate
0	1	0	1	1	1	0	0	4	2	0.122325	10/10	NonOverlappingTemplate
0	2	0	0	0	2	0	0	3	3	0.066882	10/10	NonOverlappingTemplate
0	2	1	0	1	1	0	1	3	1	0.534146	10/10	NonOverlappingTemplate
1	0	0	2	0	0	2	0	5	0	0.004301	10/10	NonOverlappingTemplate
1	0	0	1	3	1	0	0	3	1	0.213309	10/10	NonOverlappingTemplate
0	0	2	1	0	0	1	1	5	0	0.008879	10/10	NonOverlappingTemplate
0	0	0	2	0	3	1	0	4	0	0.017912	10/10	NonOverlappingTemplate
1	1	0	1	0	2	0	1	3	1	0.534146	10/10	NonOverlappingTemplate
2	1	0	0	0	0	3	1	3	0	0.122325	10/10	NonOverlappingTemplate
0	1	1	1	0	0	1	0	5	1	0.017912	10/10	NonOverlappingTemplate
0	1	0	1	3	0	0	0	3	2	0.122325	10/10	NonOverlappingTemplate
0	0	0	1	3	0	0	2	3	1	0.122325	10/10	NonOverlappingTemplate
0	2	1	0	0	0	1	2	3	1	0.350485	10/10	NonOverlappingTemplate
0	0	1	0	1	3	1	0	3	1	0.213309	10/10	NonOverlappingTemplate
0	0	1	0	2	2	1	0	3	1	0.350485	10/10	NonOverlappingTemplate
0	1	0	0	0	0	1	2	6	0	0.000199	10/10	NonOverlappingTemplate
1	0	0	0	1	1	0	0	5	2	0.008879	10/10	NonOverlappingTemplate
1	0	1	2	1	1	0	1	3	0	0.534146	10/10	NonOverlappingTemplate
1	1	1	0	1	1	1	0	4	0	0.213309	9/10	NonOverlappingTemplate
0	1	2	0	0	3	1	0	3	0	0.122325	10/10	NonOverlappingTemplate
0	1	0	0	3	2	0	0	3	1	0.122325	10/10	NonOverlappingTemplate
0	1	0	0	1	2	2	1	3	0	0.350485	10/10	NonOverlappingTemplate
2	1	0	1	1	0	0	1	3	1	0.534146	10/10	NonOverlappingTemplate
1	0	0	2	2	1	0	0	3	1	0.350485	10/10	NonOverlappingTemplate
0	0	3	0	0	4	0	0	3	0	0.004301	10/10	NonOverlappingTemplate
0	1	0	0	0	1	0	3	4	1	0.035174	10/10	NonOverlappingTemplate
1	0	0	1	1	1	0	1	4	1	0.213309	10/10	NonOverlappingTemplate
0	1	0	1	1	0	0	1	6	0	0.000439	10/10	NonOverlappingTemplate

```
  0   0   2   2   0   1   1   0   4   0  0.066882     10/10      NonOverlappingTemplate
  0   1   0   1   1   0   0   0   5   2  0.008879     10/10      NonOverlappingTemplate
  1   1   1   1   1   0   0   0   4   1  0.213309     10/10      NonOverlappingTemplate
  1   0   1   0   1   1   0   1   4   1  0.213309     10/10      NonOverlappingTemplate
  1   0   0   1   1   0   1   1   4   1  0.213309      9/10      NonOverlappingTemplate
  0   0   1   1   1   0   1   1   3   2  0.534146     10/10      NonOverlappingTemplate
  1   1   0   1   0   1   0   1   4   1  0.213309      9/10      NonOverlappingTemplate
  1   0   0   2   0   1   0   1   4   1  0.122325     10/10      NonOverlappingTemplate
  0   1   1   1   0   0   2   2   3   0  0.350485     10/10      NonOverlappingTemplate
  0   0   2   1   2   0   0   0   4   1  0.066882     10/10      NonOverlappingTemplate
  0   1   0   0   2   1   1   0   4   1  0.122325     10/10      NonOverlappingTemplate
  1   0   1   1   1   1   0   1   4   0  0.213309     10/10      NonOverlappingTemplate
  0   0   1   1   1   0   2   0   5   0  0.008879     10/10      NonOverlappingTemplate
  2   5   0   0   0   1   2   0   0   0  0.004301     10/10      OverlappingTemplate
  0   4   1   0   0   0   0   2   3   0  0.017912     10/10      Universal
  1   1   4   0   0   1   0   1   2   0  0.122325     10/10      ApproximateEntropy
  0   2   0   0   3   0   1   4   0   0  0.017912     10/10      RandomExcursions
  1   2   1   0   3   0   0   1   2   0  0.350485     10/10      RandomExcursions
  1   0   1   0   4   1   1   2   0   0  0.122325     10/10      RandomExcursions
  1   0   1   1   3   1   0   1   1   1  0.739918     10/10      RandomExcursions
  0   1   2   1   3   1   0   0   2   0  0.350485     10/10      RandomExcursions
  0   0   2   1   3   0   2   1   1   0  0.350485     10/10      RandomExcursions
  1   3   0   2   3   1   0   0   0   0  0.122325     10/10      RandomExcursions
  1   0   0   1   5   1   0   1   0   1  0.017912     10/10      RandomExcursions
  1   0   0   1   1   1   0   1   0   5  0.017912     10/10      RandomExcursionsVariant
  0   0   1   1   0   1   0   2   1   4  0.122325     10/10      RandomExcursionsVariant
  0   0   0   1   2   0   0   1   1   5  0.008879     10/10      RandomExcursionsVariant
  0   0   1   0   0   2   2   1   1   3  0.350485     10/10      RandomExcursionsVariant
  0   0   0   2   0   0   4   1   0   3  0.017912     10/10      RandomExcursionsVariant
  0   1   0   1   0   3   0   0   1   4  0.035174     10/10      RandomExcursionsVariant
  1   0   0   2   1   2   0   1   0   3  0.350485     10/10      RandomExcursionsVariant
  1   2   0   2   0   0   1   1   0   3  0.350485     10/10      RandomExcursionsVariant
  1   1   0   0   2   2   1   0   0   3  0.350485     10/10      RandomExcursionsVariant
  1   0   1   1   0   0   0   2   1   4  0.122325     10/10      RandomExcursionsVariant
  1   1   0   0   1   0   0   0   2   5  0.008879     10/10      RandomExcursionsVariant
  0   1   1   1   2   0   0   0   2   3  0.350485     10/10      RandomExcursionsVariant
  1   1   2   0   0   0   0   1   2   3  0.350485     10/10      RandomExcursionsVariant
  1   1   1   2   1   0   0   1   0   3  0.534146     10/10      RandomExcursionsVariant
  0   2   1   1   3   0   0   0   0   3  0.122325     10/10      RandomExcursionsVariant
  0   1   3   0   0   2   0   1   0   3  0.122325     10/10      RandomExcursionsVariant
  0   2   0   2   0   0   1   1   1   3  0.350485     10/10      RandomExcursionsVariant
  0   2   0   0   0   0   2   1   1   4  0.066882     10/10      RandomExcursionsVariant
  0   0   0   4   2   2   1   1   0   0  0.066882     10/10      Serial
  0   0   0   6   0   1   1   0   2   0  0.000199     10/10      Serial
  1   1   2   0   0   0   0   5   0   1  0.008879     10/10      LinearComplexity


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 8 for a
sample size = 10 binary sequences.

The minimum pass rate for the random excursion (variant) test
is approximately = 8 for a sample size = 10 binary sequences.

For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
A Testable True Random Number Generator for Linux Security Applications

Richting: **master in de industriële wetenschappen: elektronica-ICT**
Jaar: **2016**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Wietrzychowski, Daniel

Datum: **22/08/2016**