Implementing True Random Number Generators Based on High Fanout Nets

Octavian CRET\textsuperscript{1}, Tamás GYÖRFI\textsuperscript{2}, Alin SUICIU\textsuperscript{1}

\textsuperscript{1} Technical University of Cluj-Napoca, Romania, Computer Science Department
E-mail: Octavian.Cret@cs.utcluj.ro

\textsuperscript{2} National Instruments Romania, Cluj-Napoca branch
E-mail: tamas.gyorfi@ni.com

Abstract. The main component of a true random number generator (TRNG) is its source of entropy. Field Programmable Gate Arrays (FPGAs) are emerging as a popular platform for implementing single-chip TRNGs. Most of the generators implemented in FPGAs exploit the jitter from ring oscillators as source of entropy. The present paper proposes a new method for implementing TRNGs in FPGAs based on high fanout circuits operating at high frequency. The advantage of this design is its solid source of entropy and its high throughput. The quality of the random number stream is further increased by applying post-processing functions. All the theoretical assumptions related to the analysis of the entropy source are validated by practical implementations and results.

Key-words: Random number generation, fanout, entropy, FPGA, cryptography.

1. Introduction

The increasing need for true random numbers is due to the emergence of many application fields where these numbers are indispensable (e.g. Quantum Cryptography). Random numbers are useful for a variety of purposes, such as generating data encryption keys, simulating and modeling complex phenomena and for selecting random samples from larger data sets. In many applications, a supplementary constraint on the random number generator is to provide numbers at a reasonably high rate (high throughput).
The evaluation of the empirical quality of a random number generator is often done by performing some statistical tests on its output sequences; these tests are grouped in batteries like Diehard [1], NIST [2] and more recently TestU01 [3].

There are three main approaches to generating random numbers using a computer, with quite different characteristics:

- **Pseudo Random Number Generators** (PRNGs) are algorithms that use mathematical formulae to produce sequences of numbers that appear random to the user that is not aware of these formulae. If the algorithm is complex and the period of the PRNG sequence is long enough, it can produce high quality numbers. The advantage is that they are easy to implement in software; their major disadvantage in the field of cryptography and security applications is their repeatability: for equal external seeds the output sequence will always be the same.

- **Unpredictable Random Number Generators** (URNGs) are algorithms that basically rely on unpredictable human-computer interaction to capture entropy. Such examples are the mouse movement on a given area by a human operator or the amount of time between keystrokes. Even though the source is not a truly entropic one (a good knowledge of the operator’s habits can ease the prediction of the next event), the results generated show a good quality and this type of methods are used in several cryptographic products (e.g. PGP). By combining several such entropy sources the results can be improved.

- **True Random Number Generators** (TRNGs) produce the random numbers based on some physical phenomena (e.g. radiation, jitter, ring oscillators, thermal noise etc.) that are digitized. They do not have an internal state like the PRNGs and the next generated bit is based entirely on the physical process. In general these raw bits are not uniformly distributed (the probability of a ‘1’ is not 50%) so they require some post-processing in order to reduce the bias (almost all TRNGs implementations reported in the literature include at least one post-processing block, that is also called resilience function).

At this moment, the main techniques that were reported in the literature for creating TRNGs are:

- **Ring oscillators** (ROs): basically, this method is exploiting the jitter of a clock signal in a purely digital design. Most reported implementations use this approach [4], [5], [6], [7], [8], [9];

- **Direct amplification of the noise** that is intrinsic in analog signals: this method relies on the amplification of the shot noise, the thermal noise, the atmospheric noise or the nuclear decay. The noise is amplified and then, using comparators and analog-to-digital converters, bits are extracted from it [10];

- **Chaos-based TRNGs**: this method is based on a well-defined deterministic analog signal that exhibits chaotic behavior. Existing implementations exploit Markov’s chaotic sources theory [11] and use mixed analog-digital chips.
Implementing True Random Number Generators Based on High Fanout Nets

- **Metastability-based**: this method is based on an odd plurality of ring-oscillators put into metastable mode. After a few periods, the ring-oscillators are switched into generation mode where the noise accumulated in the metastable mode is sampled [16];

- **BRAM access** (only for Xilinx FPGAs): in this method, the primary source of entropy is a True Dual-Port BlockRAM operating at high frequency, which is used in a special architecture that creates a concurrent write conflict (Xilinx BRAMs allow simultaneous write access to the same memory location). That conflict is converted into a strong entropy source [12], [13].

- **Crosstalk-based TRNGs**: this method relies on filling a region or the whole FPGA chip close to its maximal capacity and exploiting the interconnection network as intensely as possible. This way, electromagnetic interferences and crosstalk effects appear inside the chip and there are strong chances for the design to exhibit a nondeterministic behavior [14], [15].

The present paper proposes a new method for implementing TRNGs in FPGAs based on high fanout circuits operating at high frequency. The advantage of this design is its solid source of entropy and high throughput. Compared to other reported TRNGs implemented in FPGA devices, the present design is portable on any FPGA chip from any vendor, with at most some minor design changes or minor settings in the vendor’s synthesis tools. The quality of the random number stream is further increased by applying post-processing functions.

The paper is organized as follows: section 2 presents the entropy source of the TRNG, section 3 presents the architecture of the proposed generator, while in section 4 some experimental results are shown; we draw some conclusions in section 5 and suggest some future research directions.

### 2. The Entropy Source

Most of the TRNGs implemented in FPGAs are based on the timing jitter from ring-oscillator clocks - the fundamental principles are well described in [4]. These implementations do not take into consideration the global, manipulable deterministic jitter which accumulates faster than the random jitter [7]. This decreases the generator’s security.

As the FPGA technology is continuously evolving, the exploration of other physical phenomena that can be used for implementing TRNGs becomes substantial. Recent implementations employed the metastability as source of entropy [16], but these are still based on ring-oscillators.

According to [16], naturally occurred metastability events are relatively rare and when they occur are sensitive to temperature and voltage changes. The probability that the physical flip-flop circuit will stay in the metastable region is very small. Therefore, straightforward employment of metastability phenomena in flip-flop circuits is inefficient due to the rare occurrence of natural metastability event.
The aim of our research was to develop new digital circuits that exploit electronic noise that might be present in FPGA chips. High fanout nets operating on high frequency proved to be a solid entropy source for TRNGs.

The fanout of a gate is defined as the number of gates that can be connected to its output without exceeding the current ratings of the gate. In order to maintain correct logic levels, the output of a gate can not drive an infinite number of other gates. However, real gates have input capacitances and resistances. So, if the output of a gate drives many other components, this gate will face a high load capacitance, which slows the transition of the output, hence increasing the signal propagation delay (Figure 1).

![Figure 1](image.png)

**Fig. 1.** A driver’s output transition facing different capacitive loads, which are proportional to the driver’s fanout (screenshot from AIM-Spice circuit simulator).

Figure 1 shows the simulation of the output transitions of a circuit composed of a driver component that faces different capacitive loads (the number of driven components varies). $V(\text{out})$ shows the transitions for a small load (15 pF) which results in a nearly square waveform. As the capacitive load increases ($V(\text{out1})$ has 1.5 nF and $V(\text{out2})$ has 6.5 nF), the transition time of the output signal facing the load also increases.

This means that fanout is not a fixed number, but it limits the maximum frequency on which the overall system can operate. If the frequency of the system is greater than this maximum (for a given fanout), nondeterministic behavior may occur.

A test design was set up to examine the behavior of circuits that have high fanout and operate at high frequency. This design was tested on a Xilinx Virtex-II Pro 30 FPGA that populates a Digilent XUP board and all the numeric values presented in the following sections refer to this board.

The design is shown in Figure 2. It was used for studying the propagation delay induced by different values of the fanout. Blocks *Driver* and *Sampler* are clocked by...
Implementing True Random Number Generators Based on High Fanout Nets

a high adjustable frequency. The output of the Driver block is called $sig_{test}$, which is a periodic signal connected to the inputs of the Load and Sampler blocks.

![Test design: studying high-fan-out high-frequency circuits.](image)

The Load block consists of $n$ Flip-Flops, each one of them being sourced by $sig_{test}$. The $n$ parameter is the actual fanout of the Driver block. The Sampler block measures the propagation delay of the driver output transitions. If the propagation delay is small (low fanout), then both D Flip-Flops ($dff_1$ and $dff_2$) will accurately capture the transitions of the $sig_{test}$ signal. $dff_1$ is active on the falling edge of the clock, so if the propagation delay of the $sig_{test}$ signal is greater than the half-period of the clock (for a high fanout), then $dff_1$ will miss $sig_{test}$'s transition, because this transition appears after the next falling edge of the clock. Thus the Sampler will yield a '0' at its output. If the fanout is further increased, both $dff_1$ and $dff_2$ Flip-Flops will miss the transitions.

The waveform diagram from Figure 3 presents the circuit’s behavior for different fanouts. It must be mentioned, that blocks Driver and Sampler were placed on the FPGA board close to each other (using LOC constraints), so the delay caused by signal path-lengths is negligible.

On the wave diagram, the first row shows the clk periodic signal. This signal is connected to the Clock input of the $dff_1$ and $dff_2$ Flip-Flops, where the first one is falling-edge active while the second one is rising-edge active, as illustrated in the waveform diagram on the second and third lines. The fourth line represents the output of the Driver source block ($sig_{test}$), as it would appear with zero fanout (i.e. zero delay).
In reality, the transition of the signal driver is not instantaneous; moreover the time needed for a transition is proportional to the fanout. The time differences $\Delta t_1$, $\Delta t_2$ and $\Delta t_3$ illustrate the propagation delay of the signal driver which might be smaller (fifth row) or greater (sixth row) than the half of the clk period. A special attention must be paid to the third case (seventh row), where the propagation delay induced by fanout is approximately equal to the half-period of clk. When the design was tested for this configuration, a nondeterministic behavior was observed, meaning that $dff_1$ has sometimes captured, sometimes missed the transition of the Driver, in an unpredictable manner. This happens because as the fanout increases, the periodic signal generated by the Driver will present phase uncertainties, resulting in ambiguity in its rising and falling edge (jitter).

Non-deterministic behavior was also achieved by fixing the fanout while changing the clock frequency. The waveform diagram from Figure 4 shows the results of this experiment: the output of $dff_2$ (Figure 2) was inspected at different clock frequencies of the design while keeping the fanout at a constant value and also preserving the implementation of the components. For this test, the frequency of the $sig\_test$ signal was doubled, so that $dff_2$ would sample the same value in each cycle.

As Figure 4 shows it, for a clock frequency of 290 MHz, $dff_2$ captures the low part of $sig\_test$ as the bias is 0% ($dff_2$ always captures a logic ‘0’). This means that the delay of $sig\_test$ is less than the half-period of the clock signal. If the frequency is set to 320 MHz, the delay is greater than the half-period of the clock signal, thus the output is always ‘1’, i.e. the bias is 100%. By using a DCM, two intermediary frequencies
can be configured for which the behavior \( \text{dff} 2 \) was unpredictable: as Figure 4 shows it, the bias is between 0 and 100% both for a frequency of 300 MHz and 310 MHz.

![Bias in \( \text{dff} 2 \)'s output for varying clock frequencies and constant fanout of 1280.](image)

The non-determinism of the design appears due to the following factors:

- Decreased slew rate due to high fanout – this results in phase uncertainties of the \( \text{Driver} \) output signal, \text{i.e.} jitter;
- Setup and / or hold time violations at the input of the Flip-Flops in the \( \text{Sampler} \) block;
- Duty-cycle distortion, meaning that for a high fanout, \( \text{sig.test} \) will be in state High for a short interval of time, which might be not enough to drive the Flip-Flops in the \( \text{Sampler} \) block.

3. Generator Architecture

3.1. Collecting Entropy

The test design presented in the previous section follows the generic architecture of a TRNG where the source of randomness is the signal generated by the \( \text{Driver} \) block.
that spans over the whole \textit{Load} block, while the \textit{Sampler}'s output is unpredictable for a certain value of the fanout. By applying a corresponding post-processing function, random numbers of a good quality can be obtained.

In that design, Flip-Flops from the \textit{Load} block contributed to the \textit{Driver}'s block fanout, but these Flip-Flops were redundant as their output was unused. By considering each Flip-Flop from the \textit{Load} block as an individual \textit{Sampler}, the randomness of the generator can be further increased by applying a function that collects the entropy from each Flip-Flop. We used an \textit{n}-bit XOR-gate for this purpose. The generator obtained this way is shown in Figure 5.

\begin{figure}[h]
  \centering
  \includegraphics[width=\textwidth]{fanout_based_TRNG.png}
  \caption{Fanout-based TRNG.}
\end{figure}

In order to obtain random bits in every clock cycle, the frequency of the signal generated by the \textit{Driver} block has to be equal to the clock frequency. This is achieved by using two T type flip-flops, as shown in Figure 5.

On first inspection it might appear that each Flip-Flop in the fanout-based TRNG samples the same noise signal on the same clock edge, thus their output would be the same and the XOR gates would always generate '0'. This is false; the Flip-Flops do not sample the same noise signal. Although the clock rising-edges will arrive at almost the same time at each Flip-Flop due to the global clock trees and buffers, the \textit{sig-test} pulses will arrive with different propagation delays due to the different lengths of the routes from the \textit{Driver} block to the Flip-Flops' inputs. For a large number of Flip-Flops, the resource utilization and the physical area occupied by the design is also expanding, thus the propagation delays caused by path-lengths become significant. Thus, the overall propagation delay from the \textit{Driver} block to the Flip-Flops is influenced by two factors:
Implementing True Random Number Generators Based on High Fanout Nets

- the fanout \( n \);
- the distance from the Driver block to each Flip-Flop.

Out of these two factors, it was demonstrated in the previous section that the first one leads to non-deterministic delays. In conclusion, the noise in the \( \text{sig}_\text{test} \) signal will manifest itself differently at the input of each Flip-Flop, thus these Flip-Flops will sample uncorrelated noise signals. We do not state that every Flip-Flop in the fanout-based TRNG will have an unpredictable output. As an example, it is possible that \( \text{sig}_\text{test} \) pulses will never reach the Flip-Flop, if the route-length between the Driver block and the Flip-Flop is too long. It is recommended to constrain the placement of the design in the smallest possible area, so that the overall propagation delay will be caused mostly by fanout and not by path-lengths. The design presented in Figure 6 was set up to test how many of the Flip-Flops exhibit non-determinism for a given fan-out and a given operational frequency.

![Fig. 6. Serial inspection of each Flip-Flop's output.](image)

In this configuration, the Driver block is implemented to have the same frequency as the clock signal. Each Flip-Flop is connected to one input of the multiplexer, so totally there are \( n \) Flip-Flops. The selection is done by the Counter that counts from 0 to \( n-1 \), so all Flip-Flops are examined one-by-one. The Counter is clocked by a Frequency Divider which divides the initial clock frequency by a factor of 212. This means that one Flip-Flop remains selected during 212 clock cycles, so its behavior can be analyzed by examining the Multiplexer’s output.
The result of this measurement is expressed as a normalized bias histogram which shows the number of Flip-Flops having a specific bias in their output for a given fanout. Figure 7 shows several bias histograms for different configurations of the fanout.

Based on Figure 7, it is clear that a large number of Flip-Flops have a deterministic output of ‘0’ or ‘1’, i.e. they always capture or miss the transitions of the Driver block. However, a certain percentage of the Flip-Flops (20% in average) have an unpredictable output which collected together create a strong source of entropy.

3.2. Optimization

One can not know in advance, which are the Flip-Flops that exhibit non-determinism (this also depends on the place-and-route algorithm, the effort level set in Xilinx tools etc.), thus the XOR gate is applied on each Flip-Flop’s output.

As Figure 7 shows it, non-deterministic output can be obtained not just for one single specific value of the fanout. If the fanout \(n\) is greater than a certain value, there will always be Flip-Flops that latch their inputs exactly when the input signal transition takes place. The fact that the generator will always provide unpredictable output for a fanout greater than a certain threshold value greatly reduces the implementation difficulties. Finding the optimal implementation, however, might be challenging.

The quality of the generated numbers depends on three factors:

- the fanout \(n\);
- the clock frequency;
- the relative placement of the components.
Implementing True Random Number Generators Based on High Fanout Nets

The optimal implementation is the one characterized by a high quality of the random number stream and high throughput, while resource utilization and power consumption is as low as possible. The following issues have to be taken into consideration:

1. The quality of the generated random numbers stream depends on the number of Flip-Flops that exhibit non-determinism, which in turn depends on the total number of Flip-Flops \( n \);

2. If quality is good and the output is unbiased, there is no need for extra post-processing function, thus saving throughput (post-processing improves quality but lowers the throughput);

3. The resource utilization ratio increases with the total number of Flip-Flops in the design;

4. If resources utilization is high, then:
   - There might be not enough space for implementing another application on the board;
   - Power consumption increases.

So, the optimal design structure is a trade-off between resource utilization and throughput.

It must be mentioned that the optimal value for \( n \) and the clock frequency may vary for different FPGA boards, packages, speed grades and a whole set of measurements may have to be done for finding them. As an example, Figure 8 shows the bias in the output for three different clock frequencies where \( n \) is constantly set to 1536. This test was also carried out on a Digilent XUP board featuring a Xilinx Virtex-II Pro30 FPGA.

![Bias in the output for different frequencies](image)

**Fig. 8.** Bias in the output for three different frequencies where the fanout is set to \( n = 1536 \).
Based on the data from Figure 8, one can conclude that the frequency range in which the output of the generator provides unpredictable results is wide, from 290 to 310 MHz. The optimal configuration for a fanout equal to 1536 would be to set the clock frequency to 300 MHz, where ‘0’ and ‘1’ appear nearly with the same probability (the output is unbiased).

However, if neither the power consumption nor the throughput is critical, the implementation of the generator is not challenging. As an example, after setting up the design presented in Figure 5 for a high value of $n$ (greater than 500 for a Virtex2PRO FPGA), the worst-case delay can be examined in the Static Timing Report or Asynchronous Delay Report which are part of the Xilinx ISE tools. By setting the clock period close to this delay (using a Digital Clock Manager), the output will become non-deterministic.

3.3. Post-processing

The main feature of a TRNG is its unpredictable output, but usually this output appears biased, meaning that ‘0’s and ‘1’s do not appear with the same probability. To overcome this problem, a post-processing function can be applied on the output of the generator. One of the most well-known and popular post-processing functions is the XOR corrector which is applied on the consecutive output bits of the generator.

As stated in the Introduction, most TRNGs implementations reported in the literature, by all methods, use a post-processing block, as it is almost impossible to obtain unbiased random bits directly from the entropy source.

The XOR corrector reduces bias, moreover, provides tolerance in the presence of environmental changes. The drawback of this function is that it reduces the throughput of the generator by a factor of $m$, where $m$ is the width of the shift register.

Figure 10 illustrates the effect of a XOR-based post-processing function applied on the output of the fanout-based TRNG and Figure 11 shows the architecture of the whole TRNG design.
4. Testing and Results

After setting these theoretical bases, the design was thoroughly tested on Xilinx FPGAs; then, its portability was verified by synthesizing it (the VHDL source files) on other FPGA devices from other vendors, like ALTERA. For a Xilinx Virtex-II Pro 30 FPGA, the fanout is set to 1536 and the design operates on 300 MHz. It must be mentioned that by applying the default compilation settings, both the Xilinx and Altera synthesis tools would reduce the number of Flip-Flops in the Samplers.
block (Figure 11) to one Flip-Flop as they are equivalent. This issue can be solved by un-checking the “Equivalent Register Removal” (Xilinx ISE) or “Remove Duplicate Registers” (Quartus) options. Depending on the FPGA target, the “Maximum Fanout” option also has to be increased above the number of flip-flops instantiated in the Samplers block, otherwise the compilation tools would duplicate the Driver block.

The XOR-based post-processing block is applied on 5 consecutive bits in order to eliminate the small bias variations. Thus the throughput of the generator is 5 cycles/bit, i.e. 60 Mbps. The bit stream was transmitted to the PC via a Fast-Ethernet interface where it was analyzed using the most recent and complete statistical test suites, NIST [2] and TestU01 [3]. The generator configured this way has successfully passed 100% of the TestU01 tests (the Alphabit and Rabbit batteries). A detailed NIST report for a stream of 240 MB is shown in Appendix A, also indicating excellent results (the TestU01 only displays a message).

As mentioned in Section 3, subsection 3.3, post-processing is mandatory for all reported TRNG designs. In this case (Xilinx FPGAs), without applying a post-processing function, the percentage of the passed tests is 65% for the NIST battery and 25% for the TestU01 Rabbit battery.

If the XOR-based post-processing is applied on 2 to 4 consecutive bits the output does not pass all the TestU01 tests; if the post-processing is applied on more than 5 consecutive bits the quality remains roughly the same (i.e. very good, all tests are passed) but the throughput of the generator decreases accordingly. Therefore we have chosen the size of the post-processing block to be 5 bits for our experiments.

This TRNG can be compared with other implementations in terms of throughput and logic (slices) utilization, in Xilinx FPGAs. We will compare only the single-core designs – in most cases, several TRNG cores can be instantiated in the same FPGA physical chip, which scales up the throughput by a factor of \( k \), where \( k \) is the number of TRNG cores.

Table 1 shows a brief comparison of this generator with other generators reported in the literature.

Without pretending to make a comprehensive survey, there are a few aspects that can be noticed about reported TRNG implementations. There are several RO-based TRNG implementations reported in the literature. The method was gradually improved, from the first reported implementation ([4]) to [9], [5] and [6]. The main problem is the extended need for qualified user intervention in the design, in order to tune the ROs to have sensibly equal oscillating periods. As the method evolved, this intervention was gradually limited, but is still important - only experienced FPGA designers could create such an implementation. It is important to mention that the only reported implementations were on Xilinx FPGAs. Proving that this approach is also valid for other FPGAs is a new task that has not been reported done. Due to particular architectural details, other FPGAs may require different types of constraints, so the same design is very unlikely to work when ported directly to another hardware host. Table 1 presents the best results reported in the literature, in terms of occupied space in Xilinx FPGAs and throughput.
Table. 1. Comparison of the proposed method with other existing TRNGs implemented on VIRTEX-II PRO FPGA

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of slices</th>
<th>Throughput (Mbps)</th>
<th>Efficiency (Kbps/Slice)</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring oscillators-based</td>
<td>565</td>
<td>2.5</td>
<td>4.4</td>
<td>Xilinx only (no other implementation reported)</td>
</tr>
<tr>
<td>(as implemented in [9])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ring oscillators-based</td>
<td>100</td>
<td>12.5</td>
<td>125</td>
<td>Xilinx only (no other implementation reported)</td>
</tr>
<tr>
<td>(as implemented in [5])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ring oscillators-based</td>
<td>232</td>
<td>7</td>
<td>30.2</td>
<td>Xilinx only (no other implementation reported)</td>
</tr>
<tr>
<td>(as implemented in [6])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRAM-based</td>
<td>686</td>
<td>25</td>
<td>36.4</td>
<td>Xilinx only</td>
</tr>
<tr>
<td>(as in [12], [13])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crosstalk-based</td>
<td>147</td>
<td>12.5</td>
<td>85</td>
<td>Xilinx only</td>
</tr>
<tr>
<td>(as in [15])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fanout-based method</td>
<td>1536</td>
<td>60</td>
<td>39</td>
<td>No restrictions</td>
</tr>
<tr>
<td>(this paper)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The BRAM-based method was first presented in [12], and then it was independently rediscovered in [13]. Naturally, it is only functional in Xilinx FPGAs (BRAMs are specific to Xilinx FPGAs).

The crosstalk-based method was first presented in [14], and then it was developed in [15]. Since it is based on filling the FPGA chip on a given region of it, it occupied more active logic. It has only been tested for Xilinx FPGA devices because it exploits a feature that is specific to Xilinx FPGAs: the carry chain lines, which are only vertical in some FPGA families and which exhibit some interesting properties [15].

Portability is an important feature of a TRNG implemented in hardware. In this context, a portable TRNG is a TRNG whose entropy source itself is portable. One can implement a “race condition”-based (as in the BlockRAM implementations [12], [13]) or a “jitter”-based TRNG (as in [4], [5], [6] and [9]), on several FPGA families, etc., but the architecture of the generator that runs correctly in one FPGA family will not be identical to the one that runs correctly in another FPGA family. In the present case of our fanout-based TRNG, if the same VHDL architecture is synthesized both on Xilinx and Altera FPGAs, the random quality of the output stream is the same without having to modify or adapt the design, as shown below.

In order to examine the design’s portability, the fanout-based generator was also implemented on other FPGA chips, from different families of the same vendor and also from different vendors. Both for a Xilinx Virtex-4 FX12 and Virtex 5 LX110T FPGA, the parameters of the design were set exactly as for the Virtex-IIPro30 FPGA and
for this configuration the same good quality of random number stream was obtained (i.e. the fanout value is 1536, the post-processing is on 5 bits, yielding a throughput of 60 Mbps and the TRNG’s output stream passes 100% of the TestU01 and NIST statistical tests).

The same configuration (fanout = 1536, clk_freq = 300 MHz) of the generator yields a slightly biased result on a Cyclone II 2C20 FPGA chip from ALTERA (49.5%–51.5%). This is corrected by applying the XOR-corrector post-processing function on 8 consecutive bits, resulting in a 37.5 Mbps throughput.

From all the other reported TRNG designs, RO-based TRNGs implementations are the most likely to be portable (BRAM-based and crosstalk-based exploit some constructive features that are specific to Xilinx FPGAs only). But this portability would come at a high price in terms of design effort. Practically, the only implementations reported in the literature were on a single type of FPGAs (mostly Xilinx FPGAs). If a given architecture described in VHDL works correctly, for instance, on a Xilinx Spartan-3E FPGA, it must be significantly modified in order to make it work on a Xilinx FPGA from another family, like Virtex-5 (and obviously, for Altera FPGAs another version of this code must be written, etc.). The parameters that need adjustment are, in this case, the length and number of ring-oscillators and their relative placement. This must be done in order to exploit the underlying source of entropy on those particular FPGA families - this action is equivalent to moving the entropy source from a hardware host (or FPGA family) to another.

The throughput of this fanout-based TRNG is only exceeded by the one of the crosstalk-based TRNG [15] (Table 1), but this TRNG design method easily allows placing multiple units of the generator on the same chip, thus increasing throughput, while this is a problem for other generators (e.g. RO-based TRNGs, because they have to be placed manually [4], [5], [6], [9], or crosstalk-based TRNGs, because of other specific aspects of that design [14], [15]).

The resource utilization relative to the maximum amount of resources on a FPGA board is low, but greater than the other implementations presented in Table 1 – which can be considered a drawback of this method. However, a logic resources occupation ratio of 2% is still very acceptable. The Sampler block contains 1536 Flip-Flops while the XOR gates use 500 LUTs. Thus in a Xilinx Virtex 5 LX110T FPGA, the generator can be mapped in 450 slices, including the Ethernet transmission module that is necessary for writing random bits into a file on the PC workstation (2% of the total amount of available logic). The remaining resources can be used to implement the actual cryptographic algorithm or any another application that uses random numbers (like a Monte Carlo simulator, a stochastic automaton etc.). It is also possible to implement several of these fanout-based TRNG cores to operate in parallel; such a design is fully scalable in terms of throughput (instantiating $k$ TRNG cores will scale up the throughput by a factor of $k$).

The power consumption of different TRNGs was analyzed in [17], but only for Actel FPGAs. The power consumption of the present generator was analyzed using the Xilinx XPowar Analyzer (for Xilinx FPGAs). In this design, the Samplers block consists of 1536 Flip-Flops and runs on 300 MHz. For the analysis, the Flip-Flop toggle rate (a parameter used in the XPower Analyzer software) is set to 100%, as
the signal generated by the Driver block toggles at each clock cycle.

The design was first implemented on a Xilinx Spartan 3S100E device. The power consumption report is summarized in Table 2. It must be noted that the Total Quiescent Power represents the power drawn by the device when there is no switching activity. The power consumed by the TRNG is represented by the Total Dynamic Power.

Table 2. Power analysis report of the proposed TRNG implemented on Spartan3E XC3S100E

<table>
<thead>
<tr>
<th>Name</th>
<th>Value (W)</th>
<th>Used</th>
<th>Total Available</th>
<th>Utilization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocks</td>
<td>0.01812</td>
<td>2</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Logic</td>
<td>0.00020</td>
<td>1556</td>
<td>1920</td>
<td>81.0</td>
</tr>
<tr>
<td>Signals</td>
<td>0.00024</td>
<td>2169</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>I/Os</td>
<td>0.00018</td>
<td>10</td>
<td>108</td>
<td>9.3</td>
</tr>
<tr>
<td>DCMs</td>
<td>0.02395</td>
<td>1</td>
<td>2</td>
<td>50.0</td>
</tr>
</tbody>
</table>

Total Quiescent Power: 0.04910 (W)
Total Dynamic Power: 0.02162 (W)
Total Power: 0.06302 (W)

 Junction Temp: 28.6 (degrees C)

For the sake of comparison, we have also implemented the RO-based TRNG (as reported in [6]) on the same Xilinx Spartan 3S100E device. In order to obtain the power consumption report, one needs to specify a value for the working frequency; we set this value to 400 MHz. The reported power consumption was 1.57 mW (Table 3).

Table 3. Dynamic power consumption and throughput values of the proposed TRNG on Spartan and Virtex FPGA families

<table>
<thead>
<tr>
<th>TRNG</th>
<th>Spartan3E XC3S100E</th>
<th>Virtex5 XC5VLX110T</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RO-based</td>
<td>1.57</td>
<td>22.2</td>
<td>7</td>
</tr>
<tr>
<td>Fanout-based</td>
<td>21.6</td>
<td>178.7</td>
<td>60</td>
</tr>
</tbody>
</table>
As Table 3 indicates it, when porting the design from Spartan3E to Virtex5 family the power consumption increases more dramatically for the RO-based TRNG (by a factor of 15), compared to the fanout-based TRNG (by a factor of 8).

A good measure of the efficiency is the Throughput / Power consumption ratio. As Table 3 indicates, this metric presents almost similar values in the case of the two designs, but the fanout-based design holds the very important advantage of being portable on any FPGA device.

The generator presented is based on a physical phenomenon, thus assuring an output of nondeterministic and uncorrelated sequences of numbers. In order to experimentally prove the true randomness of the design, several samples were generated, separated by a physical shut-down, and then compared. If the generator is a true random one, then bits in the same position from two different samples should be equal with the probability of 50%. Figure 12 shows the result of this comparison for two samples of 100 MB each, where the samples are divided into smaller blocks of 10 kbytes. As one can see, the condition is met and the TRNG is undoubtedly working correctly.

5. Conclusions and Future Work

A new alternative for implementing TRNGs in hardware was presented. This method is based on exploiting nondeterminism introduced by high fanout nets. Its correctness was proved by means of physical prototypes implemented on several boards populated with FPGA devices from Xilinx (Virtex2PRO, Virtex4 and Virtex5 families) and Altera (Cyclone II family).

Until recently, the most popular method for building TRNGs in FPGAs was the one based on ring oscillators. The principle that lies underneath RO-based TRNGs is the manipulation of electronic jitter. It is commonly accepted that jitter is a phenomenon that appears in any electronic circuit. It does not constitute a problem if kept under control - the manufacturer guarantees a maximal level of jitter in its products, which is also the case of FPGA devices. However, RO-based TRNGs include
Implementing True Random Number Generators Based on High Fanout Nets

a design that aims at increasing the occurrence of jitter and exploiting it in order to generate random bits.

As any other design, the design presented in this paper is not totally jitter-free, but it is not specially conceived to exploit jitter as the main entropy source. It is possible that jitter contributes to the exhibited nondeterminism, but the tests that were performed (as illustrated in Figure 4, Figure 7 and Figure 8) clearly show the decisive influence of fanout on the global level of entropy. The design’s jitter alone is not strong enough to produce non-determinism.

The greatest advantage of the fanout-based TRNG is its portability. It clearly differentiates it from all the other reported TRNGs implementations, which suffer from the same problem: the fact that it is often required to perform a manual place and route of the design’s components (for instance, the coupled ROs oscillating periods must be equal, with a very small admitted tolerance – see [4]). This makes the other TRNG design methods hard to approach by non-experts, both for implementing the reported designs from the literature and for porting an existing TRNG from a hardware support to another one. The method presented in this paper is easily portable on any hardware host, with a minimal degree of intervention required on the design architecture (VHDL source files). This was proven by our Altera FPGA implementation that was done by just synthesizing the same architecture that yielded excellent results on Xilinx FPGAs.

The statistical test results show the strength of the entropy source. The generator’s throughput is 60 Mbps for a single TRNG block or core, but more TRNG cores can be instantiated in parallel, thus proportionally increasing the throughput. We have performed physical tests by instantiating up to 50 cores in a Virtex5 FPGA, thus obtaining a throughput of 3 Gbps. The resource utilization is kept low (2% of the available logic resources on a Virtex5 LX110T, for a single TRNG core, but it depends on the capacity of the physical board available), thus permitting the implementation of the whole system that uses true random numbers (for instance, a cryptographic system) on the same chip. Such an embedded architecture increases the system’s security – keeping everything into a single chip makes the generator far more resistant to any attacks.

This paper brings some important theoretical novelties:

- determining a universal method that can allow the implementation of performing TRNGs on any hardware support that has enough active logic available;

- explaining the mechanism from the entropy generation point of view;

- in-depth analysis of the fanout-based entropy source (normalized bias histograms, bias analysis for different clock frequencies);

- proposing a new testing method for TRNGs for consecutive runs (separated by a physical shut-down of the TRNG).

It is possible to combine this generator with a TRNG based on another physical principle, like jitter (as used in ring oscillators-based designs), BRAM access or
crosstalk, thus gaining entropy from more than one source, which leads to an even better quality of the random number stream. One such implementation would be to replace the DCM with a ring-oscillator (see Figure 11).

A future research direction will be to test the proposed design when exposed to environment variations and to analyze whether the generator can be attacked by manipulating external conditions (temperature, radiations etc.). Also, the parameters of the generator (size, throughput) could be further improved by increasing the percentage of Flip-Flops having non-deterministic behavior in the generator (∼20% in the current implementation).

Acknowledgements. This work was supported by National Instruments under grant no. 24222 / 09.10.2011.

References


Implementing True Random Number Generators Based on High Fanout Nets


## Appendix – NIST Statistical Test Suite Results

Results for the uniformity of p-values and the proportion of passing sequences

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
<th>C9</th>
<th>C10</th>
<th>P-value</th>
<th>Prop.</th>
<th>Statistical test</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>105</td>
<td>109</td>
<td>91</td>
<td>102</td>
<td>112</td>
<td>87</td>
<td>99</td>
<td>102</td>
<td>97</td>
<td>0.292870</td>
<td>0.9908</td>
<td>frequency</td>
</tr>
<tr>
<td>87</td>
<td>88</td>
<td>111</td>
<td>96</td>
<td>93</td>
<td>93</td>
<td>98</td>
<td>112</td>
<td>96</td>
<td>106</td>
<td>0.613759</td>
<td>0.9918</td>
<td>block-frequency</td>
</tr>
<tr>
<td>80</td>
<td>97</td>
<td>104</td>
<td>98</td>
<td>112</td>
<td>94</td>
<td>105</td>
<td>116</td>
<td>82</td>
<td>92</td>
<td>0.179945</td>
<td>0.9908</td>
<td>cumulative sums</td>
</tr>
<tr>
<td>76</td>
<td>102</td>
<td>112</td>
<td>98</td>
<td>99</td>
<td>93</td>
<td>102</td>
<td>101</td>
<td>111</td>
<td>86</td>
<td>0.288509</td>
<td>0.9888</td>
<td>cumulative sums</td>
</tr>
<tr>
<td>87</td>
<td>118</td>
<td>100</td>
<td>91</td>
<td>101</td>
<td>107</td>
<td>117</td>
<td>89</td>
<td>71</td>
<td>99</td>
<td>0.027546</td>
<td>0.9888</td>
<td>runs</td>
</tr>
<tr>
<td>94</td>
<td>82</td>
<td>110</td>
<td>107</td>
<td>100</td>
<td>110</td>
<td>95</td>
<td>92</td>
<td>84</td>
<td>106</td>
<td>0.375831</td>
<td>0.9888</td>
<td>longest-run</td>
</tr>
<tr>
<td>104</td>
<td>103</td>
<td>98</td>
<td>101</td>
<td>95</td>
<td>98</td>
<td>102</td>
<td>84</td>
<td>95</td>
<td>100</td>
<td>0.960111</td>
<td>0.9878</td>
<td>rank</td>
</tr>
<tr>
<td>121</td>
<td>111</td>
<td>79</td>
<td>93</td>
<td>86</td>
<td>100</td>
<td>82</td>
<td>99</td>
<td>107</td>
<td>102</td>
<td>0.065344</td>
<td>0.9898</td>
<td>fft</td>
</tr>
<tr>
<td>86</td>
<td>112</td>
<td>90</td>
<td>108</td>
<td>96</td>
<td>108</td>
<td>99</td>
<td>100</td>
<td>93</td>
<td>88</td>
<td>0.582500</td>
<td>0.9929</td>
<td>nonperiodic templates</td>
</tr>
<tr>
<td>108</td>
<td>90</td>
<td>91</td>
<td>100</td>
<td>107</td>
<td>94</td>
<td>94</td>
<td>88</td>
<td>112</td>
<td>96</td>
<td>0.696376</td>
<td>0.9898</td>
<td>overlapping templates</td>
</tr>
<tr>
<td>85</td>
<td>106</td>
<td>109</td>
<td>107</td>
<td>98</td>
<td>78</td>
<td>101</td>
<td>78</td>
<td>99</td>
<td>119</td>
<td>0.045614</td>
<td>0.9949</td>
<td>apen</td>
</tr>
<tr>
<td>70</td>
<td>88</td>
<td>65</td>
<td>59</td>
<td>79</td>
<td>77</td>
<td>65</td>
<td>63</td>
<td>73</td>
<td>74</td>
<td>0.382393</td>
<td>0.9902</td>
<td>random excursions</td>
</tr>
<tr>
<td>68</td>
<td>86</td>
<td>76</td>
<td>64</td>
<td>70</td>
<td>64</td>
<td>61</td>
<td>82</td>
<td>66</td>
<td>76</td>
<td>0.450367</td>
<td>0.9930</td>
<td>random excursions variant</td>
</tr>
<tr>
<td>93</td>
<td>108</td>
<td>98</td>
<td>97</td>
<td>100</td>
<td>87</td>
<td>107</td>
<td>82</td>
<td>99</td>
<td>109</td>
<td>0.611639</td>
<td>0.9918</td>
<td>serial</td>
</tr>
<tr>
<td>100</td>
<td>105</td>
<td>88</td>
<td>96</td>
<td>110</td>
<td>103</td>
<td>107</td>
<td>88</td>
<td>88</td>
<td>95</td>
<td>0.711017</td>
<td>0.9929</td>
<td>serial</td>
</tr>
<tr>
<td>94</td>
<td>104</td>
<td>94</td>
<td>105</td>
<td>112</td>
<td>109</td>
<td>88</td>
<td>98</td>
<td>84</td>
<td>92</td>
<td>0.552757</td>
<td>0.9888</td>
<td>linear complexity</td>
</tr>
</tbody>
</table>

The minimum pass rate for each statistical test with the exception of the random excursion (variant) test is approximately = 0.980465 for a sample size = 980 binary sequences. The minimum pass rate for the random excursion (variant) test is approximately 0.978821 for a sample size = 713 binary sequences. For further guidelines construct a probability table using the MAPLE program provided in the addendum section of the documentation.