Asynchronous Quasi-Random Number Generator: Taking Advantage of PVT Variations

Rodrigo N. Wuerdig, Marcos L. L. Sartori, Ney L. V. Calazans
PUCRS - School of Technology - Ipiranga Av., 6681 - Porto Alegre - Brazil, 90619-900
rodrigo.wuerdig@acad.pucrs.br, marcos.sartori@acad.pucrs.br, ney.calazans@pucrs.br

***Abstract*— Random number generators find application in many fields, including cryptography, digital signatures and network equipment testers, to cite a few. Two main classes of such generators are usually proposed, pseudo-random number generators and true-random number generators. The former are simple to build and use, but cannot be employed in every application, especially in those where randomness is meant to support security. The later can be complicated to build, since they often must rely on hard-to-predict events that are hard to produce in the deterministic world of digital circuits. This work proposes a quasi-random number generator hardware implementation, intended to provide most of the benefits of true-random number generators with costs closer to those of pseudo-random number generators. The quasi-random number generator described here relies on the use of asynchronous circuit design techniques allied to process, voltage and temperature variability to achieve relatively high degrees of randomness. An FPGA prototype demonstrates the feasibility of the approach.**

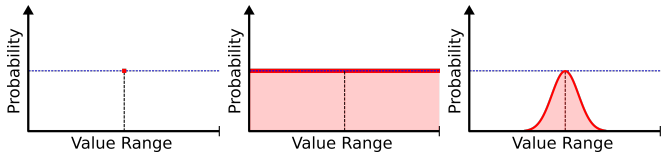
I. Introduction to Random Number Generators

Given the need to use random sequences of symbols, most often in the form of numbers, in several applications, this paper explores the design and construction of hardware modules to produces such sequences. Ideally, true-random generators are the most indicated, but if repetition of results is required, they may not be the best choice. This paper deals with the design of random number generators that set a compromise between pseudo-random and true-random generators.

A true-random number generator (TRNG) has equal probability of producing any value at any given time. The probability of generating a value is not bound by previous or future values. Conversely, a pseudo-random number generator (PRNG) is deterministic, meaning that at any given moment it is bound to generate a specific value. This makes possible to predict its output, based on previously generated values or on the instant of the value generation. Thus, a PRNG is not in fact random. A quasi-random number [1] generator (QRNG) produces results with a range with some given uncertainty. The narrower the range the more similar its behaviour is to PRNGs. Conversely, the wider this uncertainty range is, the more its behaviour resembles a TRNG. Figure 1 shows the typical probability distribution of values produced by the three classes of RNGs discussed in this Section.

A. PRNGs

A PRNG is a class of number generators that produces values in an evenly distributed, but predictable, sequence. This class of generators is deterministic, they are useful on applications requiring reproducible values in different executions. Common examples of applications where PRNGs are useful include simulation and application modeling. Usually, PRNGs are implemented using linear-feedback shift registers (LFSRs), as exemplified in Figure 2. An LFSR generates a sequence of numbers that appears to be random. It comprises a register with a fixed number of bits



(a) Pseudo-Random (b) True-Random (c) Quasi-Random

Fig. 1: Probability distributions for PRNGs, TRNGs and QRNGs. Assuming a specific number generation moment, (a) has probability 1 for a single value and 0 for all other values in the range; (b) has equal probability of generating any value at any moment, and (c) has a probability distribution between these extremes. Note: area under the curves (b), (c) is 1.

and a set of XOR (or XNOR) gates. Every cycle the register in shifted and its least significant bit (LSB) is set according to the characteristic polynomial expression. The polynomial expression defines which bits in the register are combined (XORed or XNORed) to produce the next LSB.

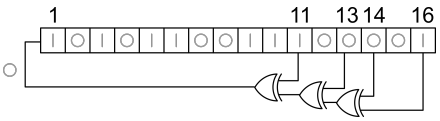


Fig. 2: An example 16-bit LFSR [2].

If an LFSR of length n generates all possible $2^n - 1$ values before repeating itself, it is *maximum*. There are multiple possible polynomial expressions for an LFSR with a given length, not all of which yield a maximum LFSR. Pre-computed polynomials for maximum LFSRs can be found at [3]. Equation (1) is the polynomial expression of the 16-bit LFSR in Figure 2.

$$x^{16} + x^{14} + x^{13} + x^{11} + 1 \quad (1)$$

Since PRNGs produce numbers in a known sequence, they have poor entropy. This makes PRNGs not fit for cryptography applications, which are sensitive to entropy attacks.

B. TRNGs

Digital circuits are by design deterministic. Often, it is a desirable feature that circuits have predictable results. However, some applications may require unpredictable data to be useful, or even safe.

Entropy is a measurement of chaos, or surprise. A perfectly ordered sequence following a pattern, e.g. $0, 1, 2, 3...$, may be considered to have no entropy, since it is possible to predict any next or previous value. Conversely, a sequence without any detectable pattern has maximum entropy, if it is not possible to predict any next value. Sources of high entropy are found in nature, e.g. in patterns of background radiation. These can

978-1-7281-0453-9/19/$31.00 ©2019 IEEE

be harvested to produce uniformly distributed random numbers. However, the nature sampling process can be complex.

Several recent works propose efficient TRNGs. For example, Wieczorek and Golofit [4] employ the instability of a flip-flop resolve time, added by a chaotic random source to produce a TRNG with high quality randomness. In another effort, Liu et al. [5] suggest using two ring oscillators as an entropy source, added by a sampler of these, instrumented with a post-processing digital part that produces random bitstreams.

The quality of TRNGs has received attention as well. The National Institute of Standards and Technology (NIST) publishes a statistical test suite to verify the randomness of TRNGs and other RNGs [6]. Proposals employ the NIST suite to verify the quality of their implementations. Some authors have used these to show that some TRNGs are not in fact TRNGs, see e.g. [7].

C. QRNGs

For some applications, generators with a some degree of (un)predictability may be enough. These applications can benefit from Quasi-random number generators (QRNGs). QRNGs provide a compromise between predictability and complexity. Quasi-random denotes that the sequence of values produced is neither completely unpredictable nor fully deterministic. Values generated by a QRNG fall in a range of uncertainty as in Figure 3. The uncertainty range Δ impacts the entropy of the produced sequence. Infinite uncertainty would produce a maximum entropy, as in TRNGs. Conversely, zero uncertainty produces a deterministic sequence, as in PRNGs.

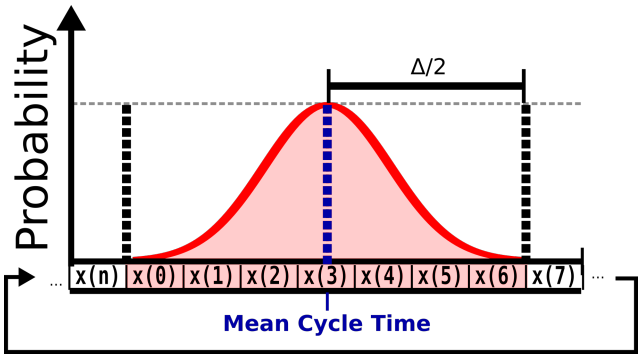


Fig. 3: Example value distribution probability by QRNGs, assuming a specific generation moment. The curve is centered at the value generated by a equivalent LFSR circuit running at the Mean Cycle Time. The Δ range derives from cycle time variations.

The uncertainty range can be widened by gathering entropy from the environment, thus producing higher quality results. This paper proposes to use asynchronous circuits to gather entropy from PVT variations, thus increasing the uncertainty range of the RNG. A final remark about Figure 3 is that the normal probability distribution it shows is just an example of possible QRNG behavior, not a rule.

II. Asynchronous Circuits

Synchronous circuits rely on a periodic global clock signal to provide a discrete common time reference and to synchronize actions, guaranteeing correct circuit behavior. The frequency of this global clock signal is defined during circuit design and is not subject to change due to operation condition variations.

Asynchronous circuits in turn do not possess such a common time reference. They operate on local communication, with timing subject to PVT variations. This characteristic makes them capable of functioning under a broader range of conditions. The variable delays of asynchronous circuits under environmental conditions variation provides an interesting solution to harvest entropy from the environment. Synchronization in these circuits takes place through the use of *handshake channels* [8] between communicating entities. Handshake protocols are characterized by two distinct steps: (i) *request*, which announces data availability for processing; and (ii) *acknowledgement*, which acknowledges the reception of and processing data.

There are different choices of handshake protocols and different circuit templates to achieve such protocols. Popular protocol choices are 4-phase (level sensitive) and 2-phase (edge sensitive) handshake protocols. 4-phase protocols take less hardware to implement, but often present less performance than 2-phase protocols.

Besides the handshake protocol, it is possible to organize sequential computation is stages that employ either one of two strategies: full-buffer and half-buffer. The first type allow that its input and its output have different data tokens, while the second cannot simultaneously have distinct data tokens at its input and output, forcing one of these to contain no data token at any moment (i.e. a *spacer*). half-buffers can lead to very fast circuits, but they sub-utilize hardware resources. The contrary is true for full-buffers.

Templates for implementing sequential hardware are often divided in two main categories: (i) quasi-delay insensitive (QDI); and (ii) bundled-data (BD).

QDI templates use delay-insensitive codes and special logic components to detect computation completion. The circuitry used to implement logic that manipulates delay-insensitive encoded data often requires more than twice the area and often employ custom special logic gates, e.g. Muller C-Elements. However, this approach allows the design of very robust circuits with timing assumptions restricted to specific places in the circuit, called isochronic forks [9].

Conversely, BD templates use local timing assumptions to determine when a pipeline stage computation completes. The request signal in bundled-data circuits are delay-matched with the propagation of the stage datapath, to guarantee that it will not arrive prior to the stage computation completion. Bundle-data templates eliminate the need for special data encoding and associated excessive circuitry, yielding lower area at the expense of a more restrictive set of timing assumptions.

A. The Mousetrap Asynchronous Template

This Section describes the Mousetrap asynchronous design template to be used in our QRNG proposal. Singh and Nowick proposed the Mousetrap template as an organization for building asynchronous pipelines [10]. Mousetrap is a 2-phase BD template that relies on conventional logic gates only. It implements a half-buffer pipeline controlled by 4-phase handshake protocol.

A Mousetrap pipeline stage comprises a latch and a controller that determines when the latch should be either transparent or opaque. A Mousetrap controller includes: (i) a *done* bit, responsible for signaling the presence or absence of data on a determined pipeline stage latch; (ii) an XNOR gate, responsible for generating the local enable signal controlling the stage latch(es).

The intuition behind Mousetrap comes from the fact that the current pipeline stage is ready to receive a new spacer (or data) from the previous stage after the data (or spacer) present in the current stage has propagated to the next stage. It achieves this by

comparing (XNORing) the current stage *done* bit with the next stage *done* bit. The basic structure of the Mousetrap pipeline can be observed in the upper part of Figure 4 (showing instances of Mousetrap controllers) combined with the latches. The *done* bits are output of the 1-bit control latch attached to the data latch, all controlled by the stage XNOR gate output.

On any stage, the *done* bit is latched together with the data. This *done* bit acts both as an acknowledge signal to the previous stage and as a request signal to the next stage. The request is delayed matched with the stage datapath by the big Δ delay line. On our circuit, the acknowledge signal is delayed by the little Δ delay to avoid hold timing violations.

The delayed version of the request signal arrives at a stage latch and is captured if the latch is transparent, becoming the next stage *done* bit. Once it is captured, the XNOR gate closes the latch, holding the data (or spacer). The captured done bit is fed back to the previous stage XNOR gate as an acknowledgement signal. This, in turn, make the previous stage latch transparent, allowing the reception of a new spacer (or data). This template has as an additional advantage that it makes straightforward to convert a synchronous circuit to an asynchronous version.

III. The Asynchronous QRNG Architecture Proposal

The practicality of using the Mousetrap template relies in that it does not require special cells like Muller C-elements. The starting point was a 128-bit PRNG design built with an LFSR using a Fibonacci representation. LFSRs provide a simple method to produce a broad variety of pseudo-random sequences. To enhance variability, the LFSR employs the polynomial expression showed in Equation (2), taken from [11], with a periodicity of $3.4028237 * 10^{38}$, calculated using the expression: $2^n - 1$.

$$x^{128} + x^{126} + x^{101} + x^{99} + 1 \tag{2}$$

Two different versions of the circuit were developed, both implementing a 4-stage Mousetrap pipeline. In the first, the combinational logic (3 2-input XOR gates or a single 4-input XOR gate) occupies a single stage and all subsequent stages exist for data propagation only. This type of implementation is not optimized for maximum throughput but it is very simple to troubleshoot the design and to tune the delays and fix hold violations. The second architecture, depicted in Figure 4, spreads the combinational logic across all pipeline stages (a 2-input XOR in each pipeline stage except the first. Distributing the logic does provide advantages, because a stage empty of combinational logic still has a minimum delay, required to avoid timing violations. When logic is distributed along several stages, the mentioned minimum delays become part of the logic gates delay and delay lines are incremented by smaller values.

Note that each stage computes part of the new data, and the fourth cycle feeds the new data back to the first stage. After that each new cycle produces a new random datum.

The architecture starts with a reset signal. This signal clears the datapath, setting requests and acknowledge signals for the initial value, loading the seed value, and making every latch initially transparent.

Handshake stretching, or holding pipeline propagation, is a quite difficult task to do without asynchronous specific cells, like asymmetric C-Elements. Most of the difficulty of developing holding mechanisms is that there are a lot of timing constraints that need to be taken into account to avoid metastability. A solution is to use a latch in every acknowledge signal where the transparency is set by the *Hold* input.

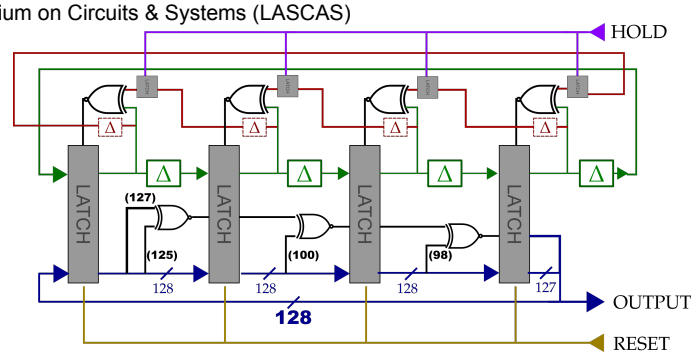


Fig. 4: The Mousetrap QRNG with combinational logic spread across the pipeline (second architecture). Red dotted Δ boxes are delay lines inserted to avoid hold violations; green Δ boxes are delay lines matching the stage delay. *Hold* halts the pipeline, stalling its operation.

Data acquisition is a relatively simple process, easy to interface to ordinary logic circuitry. The first step to fetch a random value is to hold the pipeline (or as called on the previous paragraph, do a handshake stretch). This can be done by setting the *Hold* signal high. The second step is to wait for a short period of time, to ensure that the *Hold* signal propagated through the latches. After this, data is available for reading on the output bus and *Hold* is set low again. This kind of data acquisition method is similar to a clock stretching process, it simplifies circuit usage.

Data throughput depends mostly on the worst request Δ (delay) added with the worst latch setup and hold time. When calculating circuit throughput it is important to notice that delays may fluctuate along time, because of dynamic variations (due e. g. to circuit temperature changes). The latency of the QRNG is the summation of all Δ values plus the setup and hold time of every latch in the circuit, except for the first latch, since it is there where the seed value is loaded into the datapath.

$$Latency = \sum_{n=1}^{3} \Delta Req(n) + Latch(n)_{Hold} + Latch(n)_{Setup}$$

A. Behavioral Simulation

Some particular care must be taken when simulating asynchronous circuits with conventional logic simulators such as Modelsim, since traditional zero-delay simulation can often break when dealing with combinational loops present in almost any asynchronous logic design. Accordingly, to simulate the proposed QRNG a unit-model delay mode was employed to avoid problems. A unit-model delay mode sets a minimum defined delay for every logic component, which can even lead to pessimistic simulations.

B. FPGA Timing Simulation and Prototyping

Most of the difficulty for simulating and prototyping asynchronous circuits, especially BD ones, relates to the optimization and trimming of the delay lines during the synthesis process, which can cause circuit malfunctioning. This is due to the underlying assumptions of most synthesis tools, that are designed primarily to support the mainstream synchronous design paradigm. When working with FPGAs it is necessary to use an RTL attribute called *keep*, assigned to every wire and component forming delay lines, to avoid logic trimming during synthesis. The circuit was thus prototyped on a Nexys board equipped with

a Xilinx XC3S200 Spartan-3 FPGA, where the *Hold* signal was associated to a toggle switch and values were mapped to the seven segment displays. One of the acknowledge signals was set to an external pin of the board, to allow the use of an oscilloscope to measure the cycle time and variations in frequency due to external conditions (e.g. temperature changes). The FPGA logic utilization of the proposed QRNG appears in Table I.

TABLE I: FPGA logic utilization for the proposed QRNG.

Logic Utilization	Used	Available	Utilization
Slice Registers	552	3840	14%
4 input LUTs	31	3840	1%

Two Nexys boards were used to prototype two instances of the proposed QRNG and were simultaneously monitored using an oscilloscope while both boards were kept running for 24 hours. An interesting part of the experiment was that the cycle time frequency was not the same on otherwise identical boards, which is clearly due to fabrication process variations between the two FPGAs. This divergence in cycle time endorses the assumptions and the expected functionality. Table II shows these results.

TABLE II: FPGA mean cycle times and divergence periods.

	Board 1	Board 2	ΔPeriod
Average Cycle Time	13.89ns	13.33ns	$\sim$ 560ps

Considering the obtained experimental values, it is possible to estimate an approximate period of data divergence between the boards. For calculating this divergence Equation (3) was used:

$$DivergencePeriod \simeq \frac{\overline{P_{b1}P_{b2}}}{|\overline{P_{b1}} - \overline{P_{b2}}|} \quad (3)$$

The formula was applied to both FPGAs with the collected experimental values. In the formula, $\overline{P_{bi}}$ stands for the average period of board bi.

$$DivergencePeriod \simeq \frac{13.89 * 13.33}{|13.89 - 13.33|} \quad (4)$$

$$DivergencePeriod \simeq 330.631ns \quad (5)$$

Considering an average frequency of 75MHz on the FPGA, the circuit would have an estimated throughput of 9.6Gigabit/s on the data output bus.

C. VLSI Logic Synthesis and Timing Simulation

Synthesizing VLSI circuits with delay lines is not a trivial task. With the advance on commercial frameworks for logic and physical synthesis, several improvements help the designer to obtain better circuits. Commercial tools improvements, on the other hand, create a distance between the circuit and the designer, since what the former describes is not necessarily what is implemented in the latter. To avoid that delay lines be interpreted as unnecessary, setting specific constraints is necessary to guide synthesis tools.

The QRNG was synthesized using the ARM Sage-X Library for the bulk CMOS TSMC 180nm technology node. The Cadence commercial framework was employed. One relevant concern when developing the circuit was latch hold and setup violations, due to the presence of short datapaths between stages, especially in the first architecture. The design was synthesized with commands to preserve the component instances over each delay line. To fix hold violations, small delay lines were needed on the acknowledge signal to compensate the latch hold time as depicted by dotted components in Figure 4. This was not needed in the FPGA versions of the circuit. Table III displays the synthesized circuit results. The circuit was simulated with physical synthesis post-layout annotated delays and worked correctly.

TABLE III: VLSI logic cells utilization for the proposed QRNG for 180nm bulk CMOS technology.

Design	Area (μm^2)	Cell Count
QRNG	42735	1392
Request Delay Lines	1005	72
Acknowledge Delay Lines	714	24

IV. Conclusions and Future Work

A small and simple QRNG. was designed, prototyped successfully in FPGAs and synthesized in one VLSI technology. This type of circuitry can easily be used in devices that do not require strong security. One future work is to investigate the use of QDI design instead of a BD template, because QDI design can enhance robustness, being useful to accommodate extreme PVT variations on the circuit. Such extreme variations can increase the QRNG randomness, Δ from Figure 3. Another relevant future work is to formalize the approach to randomness tests of the proposed QRNG, comparing it to PRNGs and TRNGs.

Acknowledgements

This research was partially funded by grants/scholarships from CNPq (under grant no. 312556/2014-4) and FAPERGS, Brazilian research funding organisms, as well as by a partial scholarship from HP Brazil.

References

[1] P. Bratley, B. Fox, and H. Niederreiter, "Algorithm 738: Programs to Generate Niederreiter's Low-discrepancy Sequences," *ACM Transactions on Mathematical Software*, vol. 20, no. 4, pp. 494–495, Dec. 1994.

[2] Wikipedia, "Linear-feedback shift register," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Linear-feedback_shift_register.

[3] R. Ward and T. Molteno, "Table of linear feedback shift registers," Texas A&M University, Tech. Rep., Oct. 2007. [Online]. Available: https://web.archive.org/web/20161007061934/http://courses.cse.tamu.edu/csce680/walker/lfsr_table.pdf.

[4] P. Z. Wieczorek and K. Golofit, "True Random Number Generator Based on Flip-Flop Resolve Time Instability Boosted by Random Chaotic Choice," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 4, pp. 1279–1292, 2018.

[5] Y. Liu, R. C. C. Cheung, and H. Wong, "A Bias-Bounded Digital True Random Number Generator with Architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 1, pp. 133–144, 2017.

[6] A. Rukhin *et al.*, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," NIST, Tech. Rep. 800-22, Apr. 2010. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf.

[7] E. Salih, "On the Security of a Double-Scroll Based "True" Random Bit Number Generator," in *European Signal Processing Conference (EUSIPCO)*, 2015, pp. 2058–2061.

[8] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design – A Systems Perspective*. Springer, 2001.

[9] A. J. Martin, "The Limitations to Delay-Insensitivity in Asynchronous Circuits," in *6th MIT Conference on Advanced Research in VLSI*, 1990, pp. 263–278.

[10] M. Singh and S. M. Nowick, "Mousetrap: High-speed transition-signaling asynchronous pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007.

[11] P. Alfke, "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators," 1996. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf.