|| Volume 5 || Issue 12 || December 2020 || ISSN (Online) 2456-0774
INTERNATIONAL JOURNAL OF ADVANCE SCIENTIFIC RESEARCH
AND ENGINEERING TRENDS

DESIGN OF CHAOS BASED PSEUDORANDOM NUMBER GENERATOR USING BEAT FREQUENCY DETECTION

[1]SHAIK RAJIYA SULTHANA, [2]D.MOHAN AJAY,

[1] *M. Tech scholar, Dept of VLSI, Jntu Kakinada,AP India, 9959989378, rajiyashaik124@gmail.com.*
[2]*ASSISTANT PROFESSOR, Dept of VLSI, Jntu Kakinada,AP India, 9177441569 mohanajay.777@gmail.com. ,*

-- ***---

Abstract: - Random numbers are used in a extensive variety of packages. Pseudo random variety mills are gradual and high-priced for many programs whilst pseudo random wide variety mills (PRNG) suffice for most packages. Although a majority of random wide variety generators had been applied in software program degree, growing call for exists for hardware implementation because of the arrival of quicker and excessive density Field Programmable Gate Arrays (FPGA). FPGAs make it possible to implement complex structures, including numerical calculations, genetic programs, simulation algorithms and many others. at hardware level.

This paper discusses in element the hardware implementation of several PRNGs and their characteristics. Random range generator is needed drastically by means of many applications like cryptography, simulation, numerical analysis, textual content-to-speech and so forth. Most C libraries have a couple of library routines for initializing, after which producing random numbers. For parametric speech synthesis application, a random quantity generator is needed to produce noise samples. Therefore, a need has been felt for the design of a committed hardware for random number generator that generates one random variety per cycle so that text-to speech conversion is finished in actual time.

--- ***---

I INTRODUCTION

Pseudo Random Number Generators (PRNGs) have become indispensable component in many cryptographic systems, including PIN/password generation, authentication protocols, key generation, random padding and nonce generation. PRNG circuits utilize a non-deterministic random process, usually in the form of electrical noise, as a basic source of randomness. Along with the noise source, a noise harvesting mechanism to extract the noise, and a post-processing stage to provide a uniform statistical distribution are other important components of the PRNG. Our focus is to design an improved FPGA based PRNGs, using purely digital components. Using digital building blocks for PRNGs has the advantage that the designs are relatively simple and well-suited to the FPGA design flow, as they can suitably leverage the CAD software tools available for FPGA design. However, digital circuits exhibit comparatively limited number of sources of random noise, e.g. metastability of circuit elements, frequency of free running oscillators and jitters (random phase shifts) in clock signals. As would be evident, our proposed PRNG circuit utilizes the frequency difference of two oscillators and oscillator jitter as sources of randomness.

Reconfigurable devices have become an integral part of many embedded digital systems, and predicted to become the platform of choice for general computing in near future. From being mainly prototyping devices, reconfigurable systems including FPGAs are being widely employed in cryptographic applications, as they can provide acceptable to high processing rate at much lower cost and faster design cycle time. Hence, many embedded systems in the domain of security require a high quality PRNG implementable on FPGA as a component. We present a PRNG for Xilinx FPGA based applications, which has a tunable jitter control capability based on DPR capabilities available on Xilinx FPGAs. The major contribution of this paper is the development of an architecture which allows on–the–fly tunabilty of statistical qualities of a PRNG by utilizing DPR capabilities of modern FPGAs for varying the DCM modeling parameters. To the best of our knowledge this is the first reported work which incorporates tunability in a PRNG.

This approach is only applicable for Xilinx FPGAs which provide programmable clock generation mechanism, and capability of DPR.DPR is a relatively new enhancement in FPGA technology, whereby modifications to predefined portions of the FPGA logic fabric is possible on–the–fly, without affecting the normal functionality of the FPGA. Xilinx Clock Management Tiles (CMTs) contain Dynamic Reconfiguration Port (DRP) which allow DPR to be performed through much simpler means

[1]. Using DPR, the clock frequencies generated can be changed on–the–fly by adjusting the corresponding DCM parameters. DPR via DRP is an added advantage in FPGAs as it allows the user to tune the clock frequency as per the need. Design techniques exist to prevent any malicious manipulations via DPR which in other ways may detrimentally affect the security of the system [2].

Mersenne Twister (MT) is a widely-used fast pseudorandom number generator (PRNG), designed by Matsumoto [8]. More CPU time is required for initialization than for generation in MT and hence, next to Mersenne Twisters, WELL generators were introduced by Panneton [9]. CPUs for personal computers later, acquired new features of SIMD operations (i.e., 128- bit operations) and multi-stage pipelines.128-bit based PRNG was proposed which was named as SIMD-oriented Fast Mersenne Twister (SFMT), which is analogous to MT using SIMD operations proposed by Saito[7]. Tsoi[10] mentioned that if the function call is avoided, WELL may be slower than MT for some CPUs. The SFMT pseudorandom number generator is a very fast generator with satisfactorily high-dimensional equidistribution property. Then Random number generators based on linear recurrences modulo 2 were introduced. Linear Feedback Shift Register random number generators, also called Tausworthe generators, which work on linear recurrences modulo 2. Trinomial-based generators have important statistical defects, but combining them can yield generators that are relatively fast and robust. Such combinations have been proposed and analyzed by Matsumoto and Wang [11, 12]. The generators given in are for 32-bit computers. Nowadays 64-bit computers are becoming increasingly common and so it is important to have good generators designed to fully use the 64-bit words given by P. L'Ecuyer [6] The huge-period generators proposed thereafter were not quite optimal. New generators with better equidistribution and bit-mixing properties were required.

A generator with a period of can be implemented using k flip-flops and k LUTs, and provides k random output bits each cycle. Despite these advantages, FPGA-optimized generators are not widely used in practice, as the process of constructing a generator for a given parameterization is time consuming, in terms of both developer man hours and CPU time. While it is possible to construct all possible generators ahead of time, the resulting set of cores would require many megabytes, and be difficult to integrate into existing tools and design flows. Faced with these unpalatable choices, engineers under time constraints understandably choose less efficient methods, such as combined Tausworthe generators [3] or parallel linear feedback shift registers (LFSRs). using cheap bit-wise shift-registers to provide long periods and good quality without requiring expensive resources. The number of bits generated per cycle is chosen generally to meet the needs of the application.

Permutation of the resulting outputs is given to the XOR gates. The output of the XOR gates are then given to the PIPO SRs, where the XOR gate outputs are shifted and thus random number generation takes place successfully. The simulations are performed in Model Sim 6.4a which is a tool and synthesized using Xilinx Plan Ahead Virtex5 kit verified on the Spartan 3E kit and the programming is written using Verilog.

The results that are obtained from the tools and the design summary obtained from Xilinx 8.1i are shown below. The initial seed is given as input. The seed is permuted.

Random Number Generation: Types and Techniques

Because randomness is so inherent in everyday life, many researchers have tried to either harvest or simulate its effect inside the digital realm. Before accomplishing this feat, however, many important questions need to be answered. What does it mean to be random? How does a person go about creating randomness, and how can he capture the randomness he encounters? How can someone know if an event or number sequence is random or not? Over generations, the answers to these questions have progressively been developed.

This paper takes a look at the current solutions, and attempts to organize the methods for creating chaos. Defining Random It is impossible to appreciate a random number generator without first understanding what it means to be random. Developing a well-rounded definition of randomness can be accomplished by studying a random phenomenon, such as a dice roll, and exploring what qualities makes it random. To begin, imagine that a family game includes a die to make things more interesting. In the first turn, the die rolls a five. By itself, the roll of five is completely random. However, as the game goes on, the sequence of rolls is five, five, five, and five. The family playing the game will not take long to realize that the die they received probably is not random.

From this illustration, it is apparent that when discussing randomness, a sequence of random numbers should be the focus of the description, as opposed to the individual numbers themselves (Kenny, 2005). To make sure the next die the family buys is random, they roll it 200 times. This time, the die did not land on the same face every time, but half of the rolls came up as a one. This die would not be considered random either, because it has a disproportionate bias toward a specific number. To be random, the die should land on all possible values equally. In a third scenario, the dice manufacturer guarantees that now all its dice land on all numbers equally. Cautious, a family role this new die 200 times to verify. Although the numbers were hit uniformly, the family realized that throughout the entire experiment the numbers always followed a sequence: five, six, one, two, etc. Once again, the randomness of the die would be questioned. For the die to be accepted as random, it could not have any obvious patterns in a sequence of dice rolls.

If it can be predicted what will happen next, or anywhere in the future, the die cannot truly be random. From the results of these dice illustrations a more formal definition of randomness can be constructed. A generally accepted and basic definition of a random number sequence is as follows: a random number sequence is uniformly distributed over all possible values and each number is independent of the numbers generated before it (Marsaglia,2005). A random number generator can be defined as any system that creates random sequences like the one just defined. Unfortunately, time has shown that the requirements for a random number generator change greatly depending on the context in which it is used.

Objective

The goal of this paper is the design, analysis and implementation of an easy-to-design, improved, low-overhead, tunable PRNG for the FPGA platform. The following are our major contributions:

1) We investigate the limitations of the BFD–PRNG [3] when implemented on a FPGA design platform. To solve the shortcomings, we propose an improved BFD–PRNG architecture suitable for FPGA based applications. To the best of our knowledge this is the first reported work which incorporates tunability in a fully digital PRNG.

2) We analyze the modified proposed architecture mathematically and experimentally.

3) Our experimental results strongly support the mathematical model proposed. The proposed PRNG has low hardware overhead, and the random bitstreams derived from the proposed PRNG passes all tests in the NIST statistical test suite.

II LITERATURE REVIEW

From the rigorous review of related work and published literature, it is observed that many researchers have designed random number generation by applying different techniques. Researchers have undertaken different systems, processes or phenomena with regard to design and analyze PRNG content and attempted to find the unknown parameters. A pseudorandom number generator (PRNG), is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. These sequences are not truly random. Although sequences that are closer to truly random can be generated using hardware random number generators, pseudorandom numbers are important in practice for simulations (e.g., of physical systems with the Monte Carlo method), and are important in the practice of cryptography. Ray C. C. Cheung, Dong-U Lee, John D. Villasenor [1], presented an automated methodology for producing hardware-based random number generator (PRNG) designs for arbitrary distributions using the inverse cumulative distribution function (ICDF).

The ICDF is evaluated via piecewise polynomial approximation with a hierarchical segmentation scheme that involves uniform segments and segments with size varying by powers of two which can adapt to local function nonlinearities. Analytical error analysis is used to guarantee accuracy to one unit in the last place (ulp). Compact and efficient PRNGs that can reach arbitrary multiples of the standard deviation can be generated. For instance, a Gaussian PRNG based on our approach for a Xilinx Virtex-4 XC4VLX100-12 field programmable gate array produces 16-bit random samples up to 8.2delta. It occupies 487 slices, 2 block-RAMs, and 2 DSP-blocks.

A normal LFSR could only generate one random bit per cycle. As multi-bits is required to form a random number in most applications, Multi-LFSRs architecture is used to implement a UPRNG. This means 32 different LFSRs are needed in a 32- bit output UPRNG. But Leap-Ahead architecture could avoid this and generate one multi-bits random number per cycle using only one LFSR. The Leap-Ahead architecture consumes less than 10% of

slices which the Multi-LFSR architecture consumes. One of the reasons for this is that the Leap-Ahead architecture has only 1LFSR in the UPRNG hardware, while the Multi-LFSR architecture has 18. The other reason is that every register in the UPRNG has to be initialed separately when the circuit is restarted, and the logic for this is complicated.

As the Multi-LFSR architecture has 18×18registers, while the Leap-Ahead architecture has only 23registers, it needs more slices for the initializing function. By implementing the Leap-Ahead LFSR architecture and Multi-LFSR architecture of both Galois type and Fibonacci type on Xilinx Vertex 4 FPGA, we acquire the conclusion that, with only very little lost in speed, Leap-Ahead LFSR architecture consumes only 10% slices of what the MultiLFSR architecture does to generate the random numbers that have the same period.

Cellular Automata (CA) have been found to make good pseudo-random number generators (PRNGs), and these CA-based PRNGs are well suited for implementation on Field Programmable Gate Arrays (FPGAs). To improve the quality of the random numbers that are generated, the basic CA structure is enhanced in two ways. First, the addition of a super-rule to each CA cell is considered. The overviews of the design of linear feedback shift register (LFSRs) and cellular automata (CA), followed by a review of related works that have utilized LFSR and CA for generating random numbers. Therefore, evaluated the performance of CA-based PRNGs suitable for implementation on FPGAs. Synthesis results for the Xilinx Spartan 3E FPGA give a good idea of the relative resources required for each configuration.

Pawel Dabal, Ryszard Pelka [4] presented FPGA Implementation of Chaotic Pseudo-Random Bit Generators‖ Modern communication systems (including mobile systems) require the use of advanced methods of information protection against unauthorized access. Therefore, one of the essential problems of modern cryptography is the generation of keys having relevant statistical properties. In recent years, the cryptographers pay an increasing attention to digital systems based on chaos theory. Carlos Arturo Gayoso, C. González, L. Arnone, M. Rabini, Jorge Castiñeira Moreira, [5] presented ―Pseudorandom Number Generator Based on the Residue Number System and its FPGA Implementation‖ Residue Number System (RNS), which allows us to design a very fast circuit that has a very different way of operating with respect to other generators.

PSEUDORANDOM number generators (PRNG) have many applications among diverse fields such as cryptography [1], communications [2], or procedural generation [3]. Specifically, in the field of instrumentation and measurements, PRNGs are needed in many applications such as statistical sampling, Monte Carlo simulations, evaluating the immunity to noise of digital systems and, in general, testing of physical, biological, and electrical systems: code density tests and determination of Wiener and Volterra kernels in nonlinear systems [4], [5]. Some of the most commonly used PRNGs are based on linear congruential generators (LCG) or linear feedback shift registers (LFSR). Many of these systems, however, present some correlations or short periods, which make them unsuitable for many applications [6]. In this context, chaos-based

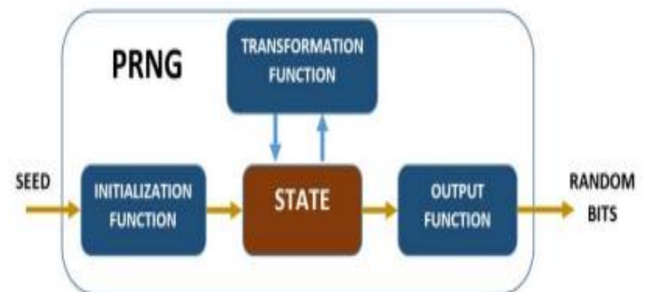


Fig. 1. Overall scheme of a generic PRNG

PRNGs have arisen as a good alternative, thanks to their properties of ergodicity, and randomlike behavior [7]. In this paper, we propose a random generator based on the logistic map that, in order to improve its statistical properties, dynamically changes its chaotic parameter. The system has been implemented in a Virtex 7 field-programmable gate array (FPGA), using 510 lookup tables (LUTs) and 120 registers. To test the good statistical properties of the proposed generator, its generated sequences have been subjected to the National Institute of Standards and Technology (NIST) tests. The sequences have passed all of these tests, proving that they are undistinguishable from a truly random sequence. The main contribution of this paper is the proposal of a novel chaos-based PRNG that:

1) Offers better randomness results than other PRNGs commonly used in simulations such as LCGs and LFSRs;

2) Requires a very small amount of resources to be implemented on an FPGA compared to other previously proposed chaos-based PRNGs.

I. PRNG ALGORITHM

A. Generic Structure of a Chaotic PRNG A PRNG is an algorithm that, starting with a seed, by using a transformation function, generates a sequence that appears to be random, and its length is much bigger than the seed length (Fig. 1). A chaotic PRNG can be easily implemented by using a digitized chaotic map $x_{i+1} = f(x_i,\gamma)$ (1) where each x_i is an element of the sequence and γ is a constant parameter that determines the behavior of the system. Using a map of this kind, starting from a seed composed by x_0 and γ , a sequence of elements $\{x_i\}$ is generated. Since each element is represented by a certain number of bits, it is possible to use all of them to build a binary random generator. However, there can be correlations among the bits within an element x_i . Therefore, to obtain better statistical properties,

only a few bits of each x_i are usually used to build the random sequences, typically the least significant bits (LSBs) since they present a low correlation. In this paper, we have based the algorithm on the logistic map, given by $x_{i+1} = \gamma x_i(1 - x_i)$ (2) where in order to work on the chaotic region, necessary for obtaining good random properties, the values of the parameter γ must be in the interval [3.57, 4]. If $\gamma < 3.57$, fixed points or periodic orbits (not suitable for PRNG applications) are obtained and, if $\gamma > 4$, the orbits usually diverge [8]. This map has been chosen due to its simplicity and its randomlike behavior that has been widely studied in the past decades [9].

B. Randomness Degradation Caused by the Digitization of the System

When a chaotic map such as the logistic map is digitized with a word length of n bits, each x_i can only take 2^n different values. Furthermore, for a given γ , the value of a certain x_i determines the value of the next element x_{i+1}. Therefore, after a maximum number of 2^n iterations, the sequence will repeat itself. Although the maximum period is 2^n, much shorter periods, on the order of $\sim 2^{n/2}$, are usually found of [10]. These short-period sequences fail most of the NIST randomness tests [8]. Furthermore, if a big number of random numbers were needed for signal processing or simulation, the sequence would start to repeat itself which could affect the simulation results. A possible strategy to reduce this problem consists on using bigger word lengths. For example, a word length of 500 bits is used in [11]. Unfortunately, this approach requires to use a big amount of extra resources to obtain longer periods. In this paper, this issue has been solved using an alternative approach that improves the random properties of a chaotic PRNG by using a small number of extra resources.

C. Random-Enhancement Proposal

The random-enhancement approach used in this paper consists of using several values of γ : $\gamma_1, \gamma_2, ... \gamma_m$ instead of a single one. The sequence $\{x_i\}$ is generated by changing the value of γ according to a sequence partition $\{k_i\}$. With this method, the first k_1 elements are obtained by $x_i = f(x_{i-1}, \gamma_1)$, the next k_2 elements are obtained as $x_i = f(x_{i-1}, \gamma_2)$, and so on. After having used all of the values of γ_i and having generated $\sum_{i=1}^{m} k_i$ elements in total, the initial value of γ and γ_1 is reused, continuing the process in a circular way. Furthermore, it has been applied to a simpler chaotic map than the skew tent map (used in [16] and [17]), the logistic map, obtaining a cost-effective high-performance PRNG.

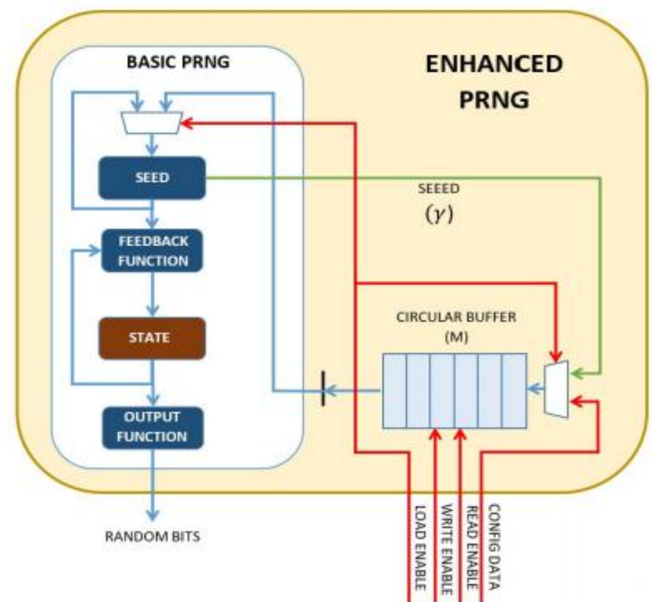


Fig. 2. Diagram of the enhanced PRNG. A FIFO is used to store the values of each γ_i . This is done with the config data signal. Write enable signal allows the system to write the value of config data inside the FIFO. Finally, read enable and load enable allow to read the next value of γ_i from the FIFO and load it into the seed register. The "config. data," "read," "write," and "load" enable inputs are driven by an external control block (not shown in the

picture) that has been designed to follow the proposed algorithm.

TABLE I
IMPLEMENTATION RESULTS

Configuration	32-bit Logistic Map	64-bit Logistic Map	32-bit Proposed PRNG
LUTs	439	903	510
Registers	46	70	120
Slices[1]	116	235	143
DSPs	13	18	13
NIST passing rate	0.252	0.979	0.989

The proposed system has been fully implemented in an FPGA and exhaustively analyzed. A scheme of the enhanced PRNG is shown in Fig. 2.

III PROPOSED SYSTEM

Design Overview

Tunability is established by setting the DCM parameters on–the– fly using DPR capabilities using DRP ports. This capability provides the design greater flexibility than the ring oscillator-based BFD-PRNG. The difference in the frequencies of the two generated clock signals is captured using a DFF. The DFF sets when the faster oscillator completes one cycle more than the slower one (at the beat frequency interval). A counter is driven by one of the generated clock signals, and is reset when the DFF is set. Effectively, the counter increases the throughput of the generated random numbers. The last three LSBs of the maximum count values reached by the count were found to show good randomness properties.

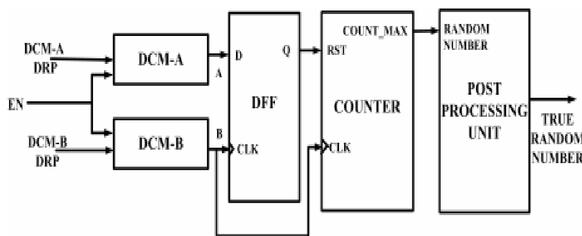


Fig. 3: Overall architecture of proposed Digital Clock Manager based tunable BFD–PRNG

Fig. 3 shows the overall architecture of the proposed PRNG. In place of two ring oscillators, two DCM modules generate the oscillation waveforms. The DCM primitives are parameterized to generate slightly different frequencies, by adjusting two design parameters M (Multiplication Factor) and D (Division Factor). In the proposed design, the source of randomness is the jitter presented in the DCM circuitry. The DCM modules allow greater designer control over the clock waveforms, and their usage eliminates the need for initial calibration [3].

The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a wellchosen feedback function can produce a sequence of bits which appears random and which has a very long cycle. Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences. Both hardware and software implementations of LFSRs are common.

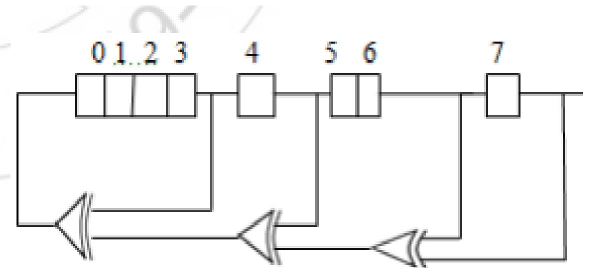


Fig 4 Galois LFSRs

The feedback tap numbers correspond to a primitive polynomial in the table so the register cycles through the maximum number of 256 states excluding the all-zeroes state. The bit positions that affect the next state are called the taps. In the diagram the taps are [7,6,4,3]. The rightmost bit of the LFSR is called the output bit. The taps are XOR'd sequentially with the output bit and then fed back into the leftmost bit. The sequence of bits in the rightmost position is called the output stream. The arrangement of taps for feedback in an LFSR can be expressed in finite field arithmetic as a polynomial mod 2. This means that the coefficients of the polynomial must be 1's or 0's.

This is called the feedback polynomial or reciprocal characteristic and 3rd bits (as shown), the feedback polynomial is X 7 +X6 +X4 +X3 +1 The 'one' in the polynomial does not correspond to a tap – it corresponds to the input to the first bit (i.e. x 0, which is equivalent to 1). The powers of the terms represent the tapped bits, counting from the left. The first and last bits are always connected as an input and output tap

respectively. The LFSR is maximal-length if and only if the corresponding feedback polynomial is primitive. This means that the following conditions are necessary (but not sufficient): The number of taps should be even. The set of taps – taken all together, not pairwise (i.e. as pairs of elements) – must be relatively prime. In other words, there must be no divisor other than 1 common to all taps.

Galois LFSRs:

Named after the French mathematician Évariste Galois, an LFSR in Galois configuration, which is also known as modular, internal XORs as well as one-to-many LFSR, is an alternate structure that can generate the same output stream as a conventional LFSR (but offset in time). In the Galois configuration, when the system is clocked, bits that are not taps are shifted one position to the right unchanged. The taps, on the other hand, are XOR'd with the output bit before they are stored in the next position. The new output bit is the next input bit. The effect of this is that when the output bit is zero all the bits in the register shift to the right unchanged, and the input bit becomes zero. When the output bit is one, the bits in the tap positions all flip (if they are 0, they become 1, and if they are 1, they become 0), and then the entire register is shifted to the right and the input bit becomes 1.

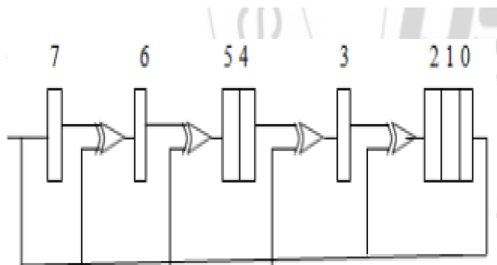


Fig 5 Galois register LFSRs

To generate the same output stream, the order of the taps is the counterpart (see above) of the order for the conventional LFSR, otherwise the stream will be in reverse. Note that the internal state of the LFSR is not necessarily the same.

The Galois register shown has the same output stream as the Fibonacci register in the first section. A time offset exists between the streams, so a different start point will be needed to get the same output each cycle. Galois LFSRs do not concatenate every tap to produce the new input (the XOR'ing is done within the LFSR and no XOR gates are run in serial, therefore the propagation times are reduced to that of one XOR rather than a whole chain), thus it is possible for each tap to be computed in parallel, increasing the speed of execution. In a software implementation of an LFSR, the Galois form is more efficient as the XOR operations can be implemented a word at a time: only the output bit must be examined individually.

It is a pseudorandom number generator proposed in 1986by Lenore Blum, Manuel Blum and Michael Shub (Blum et al., 1986). Blum Blum Shub takes the form: Xn+1 = Xn2mod n Where n=p x q is the product of two large primes p and q. At each step of the algorithm, some output is derived from xn+1; the output is commonly the bit parity of Xn+1 or one or more of the least significant bits of Xn+1. The two primes, p and q, should both be congruent to 3 (mod 4) . Steps for executing Blum Blum Shub Generator algorithm:

The Blum Blum Shub Generator is known to be the cryptographically secure pseudo random number generator (CSPRNG). The algorithm for BBS generator is as follows: Select two big prime numbers p and q, such that both the numbers leave a remainder of 3 when divided by 4. Choose n = p * q Choose seeds, such that s is relatively prime to n which means that neither p nor q is a factor of s. Xo = s2 mod n The consequent values are generated according to the formula Xi = (Xi1)2mod n A sequence of binary digits is produced according to the formula Bi= Xi mod2 The output sequence is B1, B2, B3, B4…… Pipelining Introduction

a) Pipelining Comes from the idea of a water pipe: continue sending water without waiting the water in the pipe to be out leads to a reduction in the critical path Either increases the clock speed (or sampling speed) or reduces the power consumption at same speed in a DSP system

b) Parallel Processing Multiple outputs are computed in parallel in a clock period The effective sampling speed is increased by the level of parallelism Can also be used to reduce the power consumption water pipe An instruction pipeline is a technique used in the design of computers to increase their instruction throughput (the number of instructions that can be executed in a unit of time).

The first step is always to fetch the instruction from memory; the final step is usually writing the results of the instruction to processor Registers or to memory. Pipelining seeks to let the processor work on as many instructions as there are dependent steps, just as an

assembly line builds many vehicles at once, rather than waiting until one vehicle has passed through the line before admitting the next one. Just as the goal of the assembly line is to keep each assembler productive at all times, pipelining seeks to keep every portion of the processor busy with some instruction. Pipelining lets the computer's cycle time be the time of the slowest step, and ideally lets one instruction complete in every cycle.

A) The LUT-SR PRNG The LUT-SR generators provide a middle performance between the LUT-Opt [1] and LUT-FIFO generator [1], by Implementation of PRNG in FPGA using Efficient Resource Utilization 91 using cheap bit-wise shift-registers to provide long periods and good quality without requiring expensive resources. The number of bits generated per cycle is chosen generally to meet the needs of the application.

B) Algorithm LUT-SR generator family uses a short and precise algorithm for expanding the full PRNG structure. The algorithm [1] uses r, t and k with period 2 − 1 where r is the number of random output bits generated per cycle, t is the XOR gate input count, k is the maximum shift register length. The parameters (r, t, k) describe the properties of the generator in terms of application requirements and architectural restrictions. The algorithmic steps are as follows, • Initial loading Initially the loading step is done by giving a seed. For rbit generator the seed size is r. As soon as the seed is given the bits are permuted. Any seed other than "all-zero state" can be given. The seed is also known as initial seed. All zero state condition cancels random number generation and makes the generator idle [1].

Permutation The simple dependency between adjoining bits is masked up using a final output permutation. The model is shown in Fig.1. Loading XOR connections, the permuted outputs are given as inputs to XOR gates. The number of inputs should not exceed r. The number of rounds should be t or t-1[1], where t is the number of XOR gates given. Each permuted output bit is used at most t times. Some bits will be assigned the same FIFO bit in multiple rounds. The XOR- ed outputs are given to the PIPO SR and fed back to the FIFO extensions [1]. PIPO SR Universal shift register performs shifting operation in addition to the parallel-in-parallel-out function.

At a time, multiple input processing happens in Parallel-in-parallelout-shift register. The purpose of the parallel-in parallel-out shift register is to take in parallel data, shifts it, then output the data [1]. FIFO Extension 1-bit shift registers are used. Bitwise shift registers improve the rate of mixing [1]. For 8-bit PRNG, the length of FIFO SR should not exceed k, where k=8.The length of the shift register is given by the number of flip-flops. The outputs from PIPO SR are fed back to FIFO or SISO SR. A FIFO is a sequential data buffer that is very easy to use. Very small FIFOs can be implemented with flip-flops or register arrays, sometimes even with shift registers [1].

The resulting sequence is fed back to the SISO SR or FIFO SR. Permutation of the resulting outputs is given to the XOR gates. The output of the XOR gates are then given to the PIPO SRs, where the XOR gate outputs are shifted and thus random number generation takes place successfully. The Random Number Generation is performed as per the methodology.

The simulations are performed in Model Sim 6.4a which is a tool and synthesized using Xilinx PlanAhead Virtex5 kit verified on the Spartan 3E kit and the programming is written using Verilog. The results that are obtained from the tools and the design summary obtained from Xilinx 8.1i are shown below. The initial seed is given as input. The seed is permuted. The results for 8-bit PRNG are discussed below. The same scheme is carried out for 64-bit PRNG. The permuted bits' output is given to the XOR gates. For 8-bit PRNG the number of XOR gates is 8(t=8). The concept of permutation is used up for improving randomness among bits and thus employing unpredictability. The first and last bits are interchanged. The same concept of permutation is used for different bit PRNGs. The permuted outputs are fed into the XOR gates and for remaining inputs to XOR gates round basis is used. Hence, the obtained XOR gate output bits are fed in a parallel basis into the PIPO SR. resulting outputs generate the random number cycle. The cycle is fed into the SISO SR [FIFO] of varying lengths (length=k).

The length should not exceed r. As each bit crosses the flip-flop, it will be set to zero. Hence, random number generation takes place. The resulting random numbers are generated such that their period is 2r -1. If the number of bits is 16, then the period is 216-1. The count of all zero state is reduced since the all zero state leads to idle condition. The period is the duration after which the entire sequence goes on repeating based on the initial seed and the permutations. So, the period for 32,

64, 128 and 512-bit PRNGs are 232-1, 264-1, 2128-1, 2512-1.

Tuning Circuitry

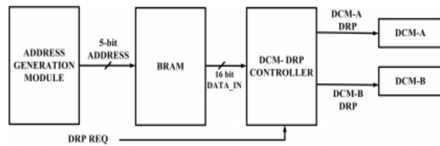


Fig. 6: Architecture of tuning circuity.

The architecture of the tuning circuitry is shown in Fig. 3. The target clock frequency is determined by the set of parameter values actually selected. The random values reached by the counter, as well as the jitter are related to the chosen parameters M and D (details are discussed in Section IV). This makes it possible to tune the proposed PRNG using the predetermined stored M and D values. As unrestricted DPR has been shown to be a potential threat to the circuit [6], the safe operational value combinations of the D and M parameters for each DCM are predetermined during the design time, and stored on an on-chip Block RAM (BRAM) memory block in the FPGA.

There are actually two different options for the clock generators – one can use the Phase Locked Loop (PLL) hard macros available on Xilinx FPGAs, or the DCMs. We next describe analytical and experimental results which compelled us to choose DCM in favor of the PLL modules for clock waveform generation.

Circuit Behavior with PLL as Clock Generator

We first consider the operational principle for the PLL, and its feasibility as a component of the proposed PRNG. The Xilinx PLL synthesizes a clock signal whose frequency is given by: FCLKFX = FCLKIN · M D (1) where FCLKIN is the frequency of input clock signal, and M and D are the multiplication and division factors previously mentioned. Values of M and D can be varied to generate the required clock frequency. The two PLLs can be parametrized with the necessary set of (M, D) values to generate two slightly different clock frequencies. Without loss of generality, assume P LLA is set up to be slightly faster than P LLB, i.e. the time periods are related by TA < TB. On reaching the beat frequency interval (say, n clock cycles), by definition, P LLA completes one cycle more than the slower one.

The following equation depicts this simple model: TA TB = N N + 1 (2) N = 2.n, where n is the estimated maximum counter value. For the first n clock cycles, the counter does not increment, and then increments by one for each of the next n clock cycles. Hence, the maximum counter values reached is n. Then, Eqn. (2) leads to: n = TB /2(TB − TA) (3) Using design configuration parameters (M and D) one of the oscillators is made to run faster than the other. This is done in order to limit the range of counter values produced.

If both the oscillators were configured to run at the same frequency we may get random numbers, but the maximum counter value produced will be very high (theoretically infinite) as per Eqn. (3). In other words, the latency of the circuit will be very high, since the counter sets and resets only after reaching a very large count value. When the Xilinx PLLs are used as clock generators, the predicted and observed counter values for all combinations of (M, D) values remain the same. This confirms that the Xilinx PLL instances demonstrate close-toideal behavior and are quasi-identical, and have negligible jitter between the waveforms generated by them. Since the BFDPRNG is critically dependent on the presence of jitter between the two generated clock waveforms, PLLs seem unsuitable as components of the proposed PRNG. Hence, next we examine the DCM as clock generators.

Circuit Behavior with DCM as Clock Generator

Without loss of generality, the clock signals produced by one of the DCM (say, DCMA) is slightly faster than the other (DCMB), implying TA < TB. This is ensured by assigning the design parameters M and D as in Eqn. (7). More details are discussed in Section IV-C. Timing diagrams of the DCM clock outputs and the resultant DFF response is shown in Fig. 4.

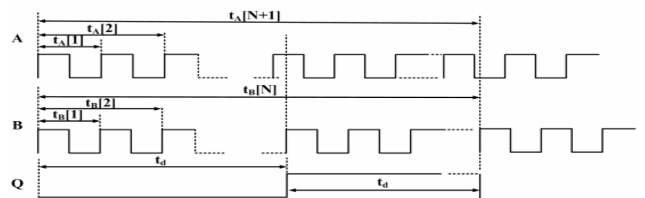


Fig. 7: Timing diagram of DCM output waveforms and the corresponding and DFF response.

Let N be the number of clock cycles of the slower clock signal in which the faster clock signal completes exactly one cycle more. Then, $t_A [N + 1] = (N + 1)\ T_A + €_A$(4) and $t_B [N] = NT_B + €_B$ (5) where $€_A$ and $€_B$ are the uncertainties due to jitter in DCMA and DCMB

respectively. The uncertainties due to jitter in DCMA and DCMB are different, this is because the DCMs are designed with distinct modeling parameters M and D. The corresponding jitter for each of the DCMs used in the proposed design is presented in Table III. For example, consider the configuration presented in Sl.No. 1. In this case, DCMA is configured with M=15 and D=31 and DCMB is configured with M=14 and D=29. This results in peak-to-peak jitter of 0.600 ns and 0.568 ns for DCMA and DCMB respectively. Of course, we also have: $t_A[N + 1] = t_B[N]$.

Assuming there is no metastability for the DFF if signal transitions occur in the setup-hold timing window around its driving clock edge (the metastability issue can be avoided by cascaded DFF combination), the transition time (td) of the DFF, the time interval after which it sets (i.e. the counter driven by the DFF resets), is estimated by: $td = t_A[N + 1] + t_B[N]\ 2 = (N + 1)T_A + NT_B + €_A + €_B\ 2$ (6) From Eqn. (6), the transition time of DFF is a random process. The output of the DFF, i.e. the time interval (td) after which the counter resets, is thus a random function. As a result, the count value obtained when the counter resets is also a random quantity.

IV SIMULATION RESULTS

Power report

```
2.  Summary
2.1.  On-Chip Power Summary
-----------------------------------------------------------------------------
|                          On-Chip Power Summary                            |
-----------------------------------------------------------------------------
|       On-Chip        | Power (mW) |  Used  | Available | Utilization (%) |
-----------------------------------------------------------------------------
| Clocks               |       1.30 |      3 |    ---    |       ---       |
| Logic                |       0.00 |     10 |   11776   |         0       |
| Signals              |       0.00 |     20 |    ---    |       ---       |
| IOs                  |       0.00 |     20 |    372    |         5       |
| Quiescent            |      31.52 |        |           |                 |
| Total                |      32.83 |        |           |                 |
-----------------------------------------------------------------------------
```

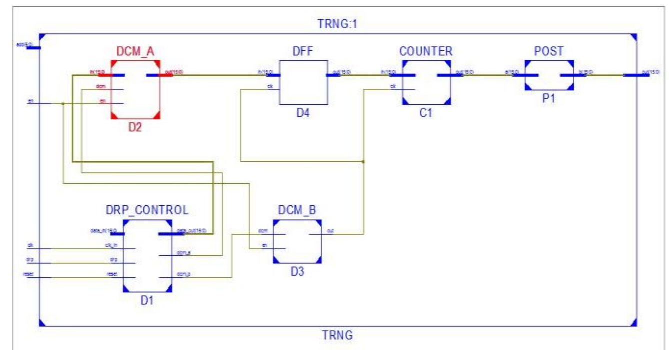
Timing report

```
========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk_out'
  Total number of paths / destination ports: 9 / 7
------------------------------------------------------------------------
Offset:              6.769ns (Levels of Logic = 2)
  Source:            C1/out_7 (FF)
  Destination:       out<8> (PAD)
  Source Clock:      clk_out rising

  Data Path: C1/out_7 to out<8>
                              Gate     Net
    Cell:in->out      fanout  Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FD:C->Q              2   0.591   0.590  C1/out_7 (C1/out_7)
     LUT2:I0->O           1   0.648   0.420  P1/Mxor_b<8>_Result1 (out_8_OBUF)
     OBUF:I->O                4.520          out_8_OBUF (out<8>)
    ----------------------------------------
    Total                     6.769ns (5.759ns logic, 1.010ns route)
                                      (85.1% logic, 14.9% route)

========================================================================
```

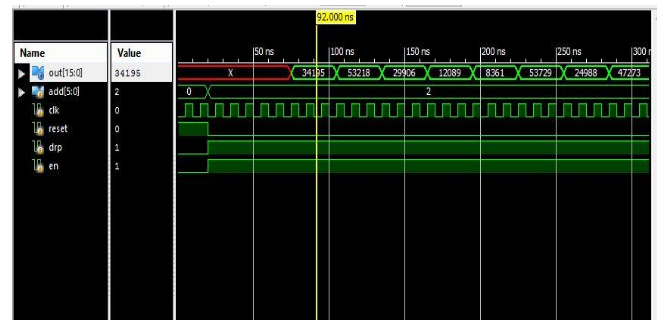
Design Summary

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	9	5888	0%
Number of Slice Flip Flops	10	11776	0%
Number of 4 input LUTs	17	11776	0%
Number of bonded IOBs	20	372	5%
Number of GCLKs	1	24	4%

RTL Schematic



SIMULATION RESULTS



V CONCLUSION

We have presented an improved fully digital tunable PRNG for FPGA based applications, based on the principle of Beat Frequency Detection and clock jitter, and with in-built error correction capabilities. The PRNG utilizes this tunability feature for determining the degree of randomness, thus providing a high degree of flexibility for various applications.

VI FUTURE SCOPE

Presently, we are dealing with the 4-digit numeric OTP system in this project. So, in order to improve the security, we need to develop an Alpha numeric (symbol base) OTP systems.

REFERENCES

[1] D. B. Thomas and W. Luk, "The LUT-SR Family of Uniform Random Number Generators for FPGA

Architectures," IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, March 2012.

[2] D. B. Thomas and W. Luk, "FPGA-optimized uniform random number generators using lot and shift registers," in Proc. Int. Conf. Field Program. Logic Appl., 2010, pp. 77–82.

[3] D. B. Thomas and W. Luk, "FPGA- optimized high - quality uniform random number generators," in Proc. Field Program. Logic Appl. Int.Conf., 2008, pp. 235-244.

[4] D. B. Thomas and W. Luk, "High quality uniform random number generation using LUT optimized state-transition matrices," J. VLSI Signal Process., vol. 47, no. 1, pp. 77–92, 2007.

[5] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long period generators based on linear recurrences modulo 2," ACM Trans. Math. Software, vol. 32, no. 1, pp. 1–16, 2006.

[6] P. L'Ecuyer, "Tables of maximally equidistributed combined LFSR generators," Math.Comput., vol. 68, no. 225, pp. 261– 269, 1999.

[7] M. Saito and M. Matsumoto, "SIMD-oriented fast mersenne twister: A 128-bit Pseudo random number generator," in MonteCarlo and Quasi- Monte Carlo Methods. NewYork: SpringerVerlag, 2006, pp. 607–622.

[8] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623- dimensionally equidistributed uniform pseudo-random number generator," ACM Trans. Modeling Comput. Simulate., vol.8, no. 1, pp. 3–30, Jan. 1998.

[9] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved longperiod generators based on linear recurrences modulo 2," ACM Trans. Math. Software, vol. 32, no. 1, pp. 1–16, 2006.

[10] K. H. Tsoi, K. H. Leung, and P. H. W. Leong, "Compact FPGAbased Pseudo Random Number Generators", in IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society, Washington, DC, 2003, p. 51.