ThundeRiNG: Generating Multiple Independent Random Number Sequences on FPGAs

Hongshi Tan[1], Xinyu Chen[1], Yao Chen[2], Bingsheng He[1], Weng-Fai Wong[1]
[1]School of Computing, National University of Singapore [2]Advanced Digital Sciences Center, Singapore

ABSTRACT

In this paper, we propose ThundeRiNG, a resource-efficient and high-throughput system for generating multiple independent sequences of random numbers (MISRN) on FPGAs. Generating MISRN can be a time-consuming step in many applications such as numeric computation and approximate computing. Despite that decades of studies on generating a single sequence of random numbers on FPGAs have achieved very high throughput and high quality of randomness, existing MISRN approaches either suffer from heavy resource consumption or fail to achieve statistical independence among sequences. In contrast, ThundeRiNG resolves the dependence by using a resource-efficient decorrelator among multiple sequences, guaranteeing a high statistical quality of randomness. Moreover, ThundeRiNG develops a novel state sharing among a massive number of pseudo-random number generator instances on FPGAs. The experimental results show that ThundeRiNG successfully passes the widely used statistical test, TestU01, only consumes a constant number of DSPs (less than 1% of the FPGA resource capacity) for generating any number of sequences, and achieves a throughput of 655 billion random numbers per second. Compared to the state-of-the-art GPU library, ThundeRiNG demonstrates a 10.62× speedup on MISRN and delivers up to 9.15× performance and 26.63× power efficiency improvement on two applications (π estimation and Monte Carlo option pricing). This work is open-sourced on Github at https://github.com/Xtra-Computing/ThundeRiNG.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Mathematics of computing → Random number generation**.

KEYWORDS

FPGA, pseudorandom number generation, statistical testing

ACM Reference Format:
Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, Weng-Fai Wong. 2021. ThundeRiNG: Generating Multiple Independent Random Number Sequences on FPGAs. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3461664

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICS '21, June 14–17, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8335-6/21/06.
https://doi.org/10.1145/3447818.3461664

1 INTRODUCTION

A pseudo-random number generator (PRNG) generates a sequence of uniformly distributed random numbers. It is a fundamental routine at the core of many modern applications, i.e., Monte Carlo simulation [44, 47] and approximated graph mining [24, 42, 45]. Many of these applications are inherently parallel and can cope well with the increasing amount of data by mapping the parallelism onto modern hardware. This has led to the need to generate a massive quantity of pseudo-random numbers with high quality of statistical randomness. In other words, the PRNG itself must also be scalable [53].

Field programmable gate arrays (FPGAs) have demonstrated promising performance on single sequence generation, benefiting from the good fit between the computation of PRNGs and the architecture of FPGAs. PRNG generally adopts recurrence algorithms [16] for generating a sequence of numbers and consists of two successive stages, as shown in the following equations.

$$x_n = f(x_{n-1}) \; n = 1, 2, 3, \ldots \tag{1}$$

$$u_n = g(x_n) \tag{2}$$

The state x_n belongs to $\boldsymbol{X}$, which is a finite set of states (the state space). $f: \boldsymbol{X} \rightarrow \boldsymbol{X}$ is the state transition function. $g: \boldsymbol{X} \rightarrow \boldsymbol{U}$ is the output function, where U is the output space. The generation of random numbers involves the following repeated steps: first, the state of the PRNG is updated according to Equation (1); then, Equation (2) extracts the random number u_n from the state x_n. To guarantee statistical randomness, existing FPGA-based PRNGs [10, 32, 51] usually implement the state transition with a large state space, requiring *block RAMs* (BRAMs) in the FPGAs to be used as storage for state processing. The output stage usually includes bitwise operations such as truncation or permutation to increase the unpredictability of the sequence. Leveraging the bit-level customization capability of FPGAs, the algorithm-specific permutation can be efficiently implemented and pipelined with the state transition to achieve high throughput in FPGAs. For instance, previous studies [1, 8–10, 32, 51] have shown that FPGAs deliver a better performance than CPU or GPU based single sequence generation.

While the single sequence of random number generation on FPGAs has been well studied, extending it to generate multiple independent sequences of random numbers (MISRN) is nontrivial. Despite that decades of studies on generating a single sequence of random numbers on FPGAs have achieved very high throughput and high quality of randomness, existing approaches for generating MISRN [10, 14, 20, 32, 55] either suffer from heavy resource consumption or fail to achieve independence among sequences. First, the resources of FPGAs can easily become a limitation for the concurrent generation. The state transition stage usually adopts states with large space or complex nonlinear arithmetic operations

Table 1: Survey of PRNG algorithms and implementations. The test suite for statistical quality is the TestU01 suite.

PRNG Algorithms	Platform	State width	#Multiplication for n instances	Single sequence Statistical quality	Multiple sequences Methods	Multiple sequences Statistical quality	Critical resources on FPGAs
Li et al. [32]	FPGA	19937	0	Crushable	Substream	Crushable	Block RAMs
Dalal et al.[10]		19937	0	Crushable	Substream	Crushable	Block RAMs
LUT-SR [51]		19937	0	Crushable	Substream	Crushable	LUTs
Philox4_32 [49]	GPU/CPU	256	$6n$	Crush-resistant	Multistream	Crush-resistant	DSP slices
MRG32k3a [29]		384	$4n$	Crush-resistant	Substream	Crushable	DSP slices
Xoroshiro128** [4]	CPU	128	$2n$	Crush-resistant	Substream	Crush-resistant	DSP slices
PCG_XSH_RS_64 [39]		64	n	Crush-resistant	Multistream	Crushable	DSP slices
LCG64 [35]		64	n	Crushable	Multistream	Crushable	DSP slices
ThundeRiNG	FPGA	192	**1**	**Crush-resistant**	Multistream	**Crush-resistant**	LUTs

in PRNG, which consumes the precious BRAM or DSP resources of FPGAs [10, 32]. Due to the heavy resource consumption, we cannot scale a large number of PRNG instances on a single FPGA. Second, the correlation among multiple sequences leads to low quality of randomness [10, 32]. Sequences generated by the same type of PRNG tend to have correlation, diminishing the quality of randomness [14, 20]. In fact, the quality of the generated sequences of the previous two designs is not guaranteed [10, 32], as they fail in some of the empirical tests such as TestU01 [30].

To our best knowledge, none of the previous studies on FPGAs achieved high quality of randomness as required in many applications, or the high throughput and scalability for MISRN. In this paper, we propose a high-throughput, high-quality, and scalable PRNG, called *ThundeRiNG*, to tackle the aforementioned two challenges. ThundeRiNG inherits *linear congruential generator* [31] (LCG) that natively supports affine transformation to generate distinct sequences. While the widely adopted LCG parallelization approaches such as state spacing [55] suffer from long-range correlation [14, 20] and efficiency problems [12, 52], we identified an opportunity to share the most resource-consuming stage between multiple PRNG instances on FPGAs, and found a technique to eliminate the correlation among the concurrently generated sequences.

Specifically, ThundeRiNG makes the following contributions:

- It enables state sharing for generating multiple independent sequences to solve the resource inefficiency problem when increasing the number of PRNGs instantiated on FPGAs.
- It has a resource-efficient decorrelation mechanism to remove the correlation among sequences to guarantee the quality of randomness.
- It consumes a constant number of DSPs for a varied number of generated sequences and achieves up to 655 billion random numbers per second (20.95 Tb/s), without compromising the quality of randomness.
- Compared with the state-of-the-art GPU implementation, it delivers up to 10.62× performance improvement. Furthermore, we demonstrate its effectiveness on two real world applications with delivering up to 9.15× speedup on throughput and 26.63× power efficiency.

The rest of the paper is organized as follows. Section 2 introduces the background and related work. Section 3 presents the design, followed by the implementation details on FPGA in Section 4. We present the experimental results and case studies in Sections 5 and 6, respectively. We conclude this paper in Section 7.

2 BACKGROUND AND RELATED WORK

In this section, we present the quality criteria and review existing approaches for generating MISRN (summarized in Table 1).

2.1 PRNG Quality Criteria

The statistical randomness of the generated sequences is the most important quality criterion of PRNG.

Statistical Randomness. Randomness is hard to measure due to its considerable evaluation space. Instead, statistical randomness is commonly used for measuring the quality of a PRNG. A numerical sequence is statistically random if it contains no recognizable pattern or regularities [54]. In essence, statistical randomness indicates how well the successive outputs of PRNG behave as independent and identically distributed (i.i.d) random variables.

Statistical Randomness Testing. There are two testing approaches for statistical randomness: theoretical test and empirical test. Theoretical test is a kind of prior test based on the knowledge of the PRNG algorithm, and thus it is not applicable for PRNGs without clear mathematical modeling [27]. In contrast, the empirical test is able to extract recognizable patterns from the generated sequences without knowledge of detailed mathematical modeling, and it is widely adopted in the evaluation of PRNGs [3, 7, 34].

The TestU01 suite[30], which is the most stringent empirical test suite, has been widely used and has become the standard for testing the statistical quality of a PRNG. It contains several test batteries, including SmallCrush (with 10 tests), Crush (with 96 tests), and BigCrush (with 160 tests). PRNGs that pass all tests in those test batteries can be referred as ***crush-resistant***, indicating a good quality of statistical randomness, while the PRNGs fail to do that is called ***crushable***, indicating that recognizable patterns exist [49]. All FPGA-based PRNGs (except this work) in Table 1 are crushable even for single sequence generation.

2.2 Multiple Sequence Generation Methods

In supporting MISRN, existing PRNGs usually adopt one of the two methods: substream and multistream (as shown in the column **Methods** for "Multiple sequences" in Table 1).

Substream. Substream based solutions equally divide the state space into many non-overlapped subspaces to generate disjoint

logical sequences. The practical criterion to guarantee nonoverlapping is maintaining at least 2^{63} skipped elements among the logical sequences [4]. This method is widely adopted in existing works [4, 19, 32, 36, 41].

Multistream. The multistream approach is that the same PRNG module is instantiated multiple times, and the instances run concurrently with different parameters for generating multiple distinct streams. All prior cited FPGA-based PRNGs use the substream solution, and only CPU/GPU based solutions adopt multistream based solutions, e.g., Philox4_32 [49] and PCG_XSH_RS_64 [39].

2.3 Challenges of MISRN Generation on FPGAs

Table 1 summarizes the existing FPGA-based PRNGs as well as CPU/GPU based PRNGs. We revisit those algorithms for potential adoption and thus analyze each method in terms of the state width, number of multiplications, statistical quality, multi-sequence generation method, and critical resources. We identify the limitations of existing works and the open challenges of multi-sequence generation on FPGAs.

2.3.1 Challenge 1: correlation among sequences. A common issue with existing methods of MISRN is the correlation between sequences. This leads to poor statistical randomness and may not satisfy the application requirements [15, 20]. Correlation violates the independence of the generated sequences and leads to inaccurate or biased results even in the simplest applications [15, 20]. All FPGA-based solutions are crushable for both single and multiple sequence generation.

2.3.2 Challenge 2: high throughput via parallelism. To increase throughput, multiple pseudo-random sequences must be generated concurrently. The recent methods [4, 32, 36, 39, 49] require instantiating one PRNG module for generating one sequence. On FPGAs, this translates to a significant resource consumption that is linearly proportional to the number of sequences. As a single FPGA has limited resources, this will severely limit the number of sequences that can be generated concurrently on one FPGA. Even worse, PRNGs usually require either a large state width (e.g., the 19937-bit state of FPGA-based solutions shown in Table 1), or complex arithmetic (e.g., the multiplication operation of CPU-based solutions shown in Table 1) to improve the randomness of the output. When instantiating on FPGAs, the large state width will consume the BRAM resources of FPGAs, and the complex arithmetic consumes heavily on DSPs. As a result, directly implementing existing CPU/GPU-based PRNGs on the FPGAs can be resource- and throughput-constrained.

3 DESIGN OF THUNDERING

We describe the design of our proposed ThundeRiNG in this section, followed by the implementation details on FPGA in the next section. As far as we know, ThundeRiNG is the first FPGA-based solution that solves the two above-mentioned challenges, providing a high-throughput, high-quality PRNG. ThundeRiNG is based on a well-studied *linear congruential generator* (LCG) PRNG [31]. To ensure the highest quality of the output, ThundeRiNG adopts a resource-efficient decorrelator that removes the correlation between the multiple sequences generated by parameterizing the LCG via increments. To scale the throughput, ThundeRiNG uses state sharing to reduce resource consumption when instantiating a massive number of PRNG instances on FPGAs.

3.1 Parameterizing LCG via Increment

The LCG algorithm has three parameters, labelled as m, a and c, where m is the ***modulus*** ($m \in \mathbb{Z}^+$), a is the ***multiplier*** ($0 < a < m$) and c is the ***increment*** ($0 \leq c < m$). The set of sequences generated with the same a, m, c parameters are represented as $\mathbb{X}_{a,m} = \{X^c_{a,m}, \ldots \mid 0 \leqslant c < m\}$, and the generation of each instance of $X^c_{a,m}$ is defined in the following equation:

$$x_{n+1} = (a \cdot x_n + c) \mod m, \; n \geq 0 \tag{3}$$

$$u_{n+1} = truncation(x_{n+1}) \tag{4}$$

Equation (3) is the state transition function, and Equation (4) is the output function, which conducts a simple truncation on x_n to guarantee that the state space is larger than the output space [39].

Instead of parameterizing PRNGs with modulus and multiplier [14], ThundeRiNG explores parameterization via the increment c, to enable a resource-efficient multiple sequence generation method. However, the generated distinct sequences also suffer the severe correlation problem [43], which motivates us to develop a decorrelation approach in the following subsection.

3.2 Decorrelation

As shown in previous studies [43], the sequences generated by LCGs with different increments still have severe correlations. There have been several existing approaches to eliminate the correlations, such as dynamic principal component analysis [46] or Cholesky matrix decomposition [17, 26]. However, these methods involve massive computation, which is resource inefficient and unpractical for high-throughput random number generation on FPGAs.

The number of possible combinations of sequences generated by LCG is very large, leading the correlations among multiple sequences hard to analyze. Therefore, we first consider the correlation between two sequences to simplify the problem, and then we extend it to multiple sequences.

3.2.1 Decorrelation on two sequences. Yao's XOR Lemma [57] states that the hardness of predication is amplified when the results of several independent instances are coupled by the exclusive disjunction. Therefore, we use the XOR operation to amplify the independence of the generated sequences by LCG. Specifically, we first adopt a light-weight but completely different algorithm from LCG for the generation of two sequences, even if they are weakly correlated, and then combine them to the sequences generated by the LCG algorithm with bitwise XOR operations.

Theorem 3.1 gives the theoretical proof of the improved independence for the newly generated sequences with our approach.

THEOREM 3.1. *Suppose $X^{c_1}_{a,m} = \{x^{c_1}_n\}_{n\in\mathbb{N}}$ and $X^{c_2}_{a,m} = \{x^{c_2}_n\}_{n\in\mathbb{N}}$ are two distinct sequences belong to $\mathbb{X}_{a,m}$, and there are two weakly correlated sequences $I = \{i_n\}_{n\in\mathbb{N}}$ and $J = \{j_n\}_{n\in\mathbb{N}}$,which are uncorrelated with the sequences in $\mathbb{X}_{a,m}$. Then the correlation between the combined sequences, $Z^1 = \{x^{c_1}_n \oplus i_n\}_{n\in\mathbb{N}}$ and $Z^2 = \{x^{c_2}_n \oplus j_n\}_{n\in\mathbb{N}}$ is weaker than the correlation between $X^{c_1}_{a,m}$ and $X^{c_2}_{a,m}$.*

Proof. First, we consider two binary uniformly distributed sequences, X and Y. As we cannot directly calculate the probability based on XOR, we transform the XOR operator to multiplication [18]. Specifically, we define a sequence transformation, $h(X) = 1 - 2X = \{1 - 2 \cdot x_n\}_{n \in \mathbb{N}}$, which maps the value of the elements in X from $\{0, 1\}$ to $\{-1, 1\}$. Then we have

$$h(X \oplus Y) = h(X) \cdot h(Y). \tag{5}$$

The mathematical expectation (E), variance (var) of $h(X)$, and the covariance (cov) between $h(X)$ and $h(Y)$ are calculated as follows:

$$E(h(X)) = 1 - 2E(X) \tag{6}$$

$$var(h(X)) = 4var(X) \tag{7}$$

$$cov(h(X), h(Y)) = 4cov(X, Y) \tag{8}$$

As X and Y are uniformly distributed, then we have

$$E(X) = \mu_X \approx 1/2. \tag{9}$$

Since $var(X) = E(X^2) - (E(X))^2$, we can approximate the variance of X:

$$var(X) = \mu_X - (\mu_X)^2 \approx 1/4 \tag{10}$$

Therefore, we can calculate the variance of the new sequence $X \oplus Y$ by Equations (7) and (10):

$$\begin{aligned} var(X \oplus Y) = \frac{var(h(X \oplus Y))}{4} &= \frac{1}{4} \cdot \Big(var(h(X)) \cdot var(h(Y)) \\ &+ var(h(Y)) \cdot E[h(X)]^2 + var(h(X)) \cdot E[h(Y)]^2 \Big) \\ &= \frac{var(h(X)) \cdot var(h(Y))}{4} \\ &= 4 \cdot var(X) \cdot var(Y) \approx 1/4 \end{aligned} \tag{11}$$

Taking two sequences in Z(in the Theorem definition), their correlation ρ_Z can be represented by the definition of correlation:

$$\rho_Z = \frac{cov(X^{c_1}_{a,m} \oplus I, X^{c_2}_{a,m} \oplus J)}{\sqrt{var(X^{c_1}_{a,m} \oplus I) \cdot var(X^{c_2}_{a,m} \oplus J)}} \tag{12}$$

Equation (12) can be further approximated using Equation (11):

$$\rho_Z \approx 4 \cdot cov(X^{c_1}_{a,m} \oplus I, X^{c_2}_{a,m} \oplus J) \tag{13}$$

where the covariance can be rewritten as

$$\begin{aligned} &cov(X^{c_1}_{a,m} \oplus I, X^{c_2}_{a,m} \oplus J) \\ &= \frac{cov[h(X^{c_1}_{a,m} \oplus I), h(X^{c_2}_{a,m} \oplus J)]}{4} \\ &= \frac{cov[h(X^{c_1}_{a,m}) \cdot h(I), h(X^{c_2}_{a,m}) \cdot h(J)]}{4} \\ &= \frac{1}{4} \cdot \Big(E[h(X^{c_1}_{a,m})h(X^{c_2}_{a,m})h(I)h(J)] \\ &\quad - E[h(X^{c_1}_{a,m})h(I))]E[h(X^{c_2}_{a,m})h(J)] \Big). \end{aligned} \tag{14}$$

As $X_{a,m}$ is independent of I and J, the first item in Equation (14) can be represented by their covariances:

$$\begin{aligned} &E[h(X^{c_1}_{a,m})h(X^{c_2}_{a,m})h(I)h(J)] \\ &= E[h(X^{c_1}_{a,m})h(X^{c_2}_{a,m})] \cdot E[h(I)h(J)] \\ &= \Big(cov[h(X^{c_1}_{a,m}), h(X^{c_2}_{a,m})] + E[h(X^{c_1}_{a,m})]E[h(X^{c_2}_{a,m})] \Big) \\ &\quad \cdot \Big(cov[h(I), h(J)] + E[h(I)]E[h(J)] \Big) \end{aligned} \tag{15}$$

Similar with Equation (13), we use the correlation ($corr$) to replace the covariance items in Equation (15):

$$\begin{aligned} &E[h(X^{c_1}_{a,m})h(X^{c_2}_{a,m})h(I)h(J)] \\ &\approx \frac{1}{4} \cdot \Big(corr[h(X^{c_1}_{a,m}), h(X^{c_2}_{a,m})] + 4E[h(X^{c_1}_{a,m})]E[h(X^{c_2}_{a,m})] \Big) \\ &\quad \cdot \Big(corr[h(I), h(J)] + 4E[h(I)]E[h(J)] \Big) \end{aligned} \tag{16}$$

As $X^{c_1}_{a,m}$ and $X^{c_2}_{a,m}$ from the same LCG set, the difference of their expectations can be ignored. We use $\mu_{X_{a,m}}$ to represent their expectation:

$$E(X^{c_1}_{a,m}) = E(X^{c_2}_{a,m}) = \mu_{\mathbb{X}_{a,m}} \tag{17}$$

Here, $\rho_{\mathbb{X}_{a,m}}$ represents the correlation of $X^{c_1}_{a,m}$ and $X^{c_2}_{a,m}$, and $\rho_{(I,J)}$ represents the correlation of I and J, Finally, combining Equations (13) to (16), Equation (12) can be simplified as

$$\rho_Z \approx \rho_{X_{a,m}} \cdot \rho_{(I,J)} + \rho_{X_{a,m}} \cdot (1 - 2\mu_{(I,J)}) + \rho_{(I,J)} \cdot (1 - 2\mu_{\mathbb{X}_{a,m}}). \tag{18}$$

The expectations of the sequence I, J and $X_{a,m}$ are very close to 1/2. Hence, the terms $\rho_{\mathbb{X}_{a,m}} \cdot (1 - 2\mu_{(I,J)})$ and $\rho_{(I,J)} \cdot (1 - 2\mu_{\mathbb{X}_{a,m}})$ in Equation (18) can be ignored. Finally, the correlation between Z^1 and Z^2 is simplified as

$$|\rho_Z| \approx \left|\rho_{X_{a,m}}\right| \cdot \left|\rho_{(I,J)}\right|. \tag{19}$$

We assume that the sequences in the $\mathbb{X}_{a,m}$ are highly correlated, so $\left|\rho_{\mathbb{X}_{a,m}}\right|$ is close to 1. $\left|\rho_{(I,J)}\right|$ is a small value (< 1) as the sequences I and J are only weakly correlated. Thus, $|\rho_Z| < \left|\rho_{\mathbb{X}_{a,m}}\right|$ from Equation (19), which proves that our XOR based approach decreases the correlation of the original LCG generated sequences. □

3.2.2 Decorrelation on multiple sequences. On top of Theorem 3.1, we further consider the correlation among the sequences generated from more than two generators. Typically, the independence of multiple random number sequences has two measurements, mutual independence and pairwise independence. Mutual independence is a strong notion of independence. It requires that each sequence is independent of all other sequences and any combination of other sequences in the set. Matsumoto et al. [37] have the hypothesis of mutual independence on the linear recurrence. However, there is no mathematically rigorous proof, and it is even impossible to evaluate all possible combinations in empirical tests when the number of sequences is large. Pairwise independence indicates that any two sequences in the domain are independent of each other, which is mostly considered measurement in PRNG [30]. Therefore, we

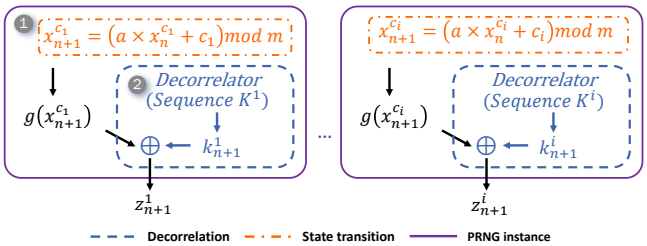


Figure 1: The decorrelation mechanism.

only extend Theorem 3.1 to pairwise independence of multiple sequences, as given in Theorem 3.2.

Theorem 3.2. *Suppose there is a set of independent sequences, denoted as $\mathbb{K}$, of which all members are uncorrelated with $\mathbb{X}_{a,m}$. Separately selecting n sequences from $\mathbb{K}$ and $\mathbb{X}_{a,m}$, denoted as $\{K^i\}_{i=1}^n$ $(n > 2)$ and $\{X_{a,m}^{c_i}\}_{i=1}^n$. The combined sequences, $\{Z^i = X_{a,m}^{c_i} \oplus K^i\}_{i=1}^n$ are pairwise independent from each other.*

Proof. We give the induction proof. Let $P(n)$ be the statement "the n decorrelated sequences $\{Z^i = X_{a,m}^{c_i} \oplus K^i\}_{i=1}^n$ are pairwise independent". We will prove that $P(n)$ is true for all $n > 2$.

We first prove $P(3)$ is true, which means three distinct sequences are pairwise independent (denote them as Z^1, Z^2 and Z^3). Based on our definition, we have $Z^1 = X_{a,m}^{c_1} \oplus K^1$ and $Z^2 = X_{a,m}^{c_2} \oplus K^2$. K^1 and K^2 belong to $\mathbb{K}$. Hence based on Theorem 3.1, we can get that Z^1 is independent from Z^2. In a similar way, Z^1 is independent from Z^3, and Z^2 is independent from Z^3. Therefore, $P(3)$ is true.

For the inductive step, assume $P(j)$ true, we will prove that $P(j+1)$ is also true. Compared to $P(j)$, Z^{j+1} is the newly introduced sequence, of which the K^{j+1} is independent with the sequences in $\{K^i\}_{i=1}^j$.

Successively combining Z^{j+1} with all sequences in $\{Z^i\}_{i=1}^j$ into pairs, and adopting Theorem 3.1 on all of the pairs, we derive that Z^{j+1} is independent from all the members in $\{Z^i\}_{i=1}^j$. Therefore, according to the definition of pairwise independence, $P(j+1)$ is true, completing the proof. □

3.2.3 The algorithm. We propose our decorrelation method based on Theorem 3.1 and Theorem 3.2, using the XOR operation to combine the correlated LCG sequences with a series of independent sequences.

The algorithm flow is shown in Figure 1, for the $\boldsymbol{i}$-th instance at Step $\boldsymbol{n+1}$, it transits the old state from $x_n^{c_i}$ to the new state $x_{n+1}^{c_i}$ by the LCG algorithm, and the inside decorrelator generates an element k_{n+1}^i and an XOR operation is performed to output z_{n+1}^i.

According to the proof in Theorem 3.2, there are two theoretical constraints on the sequences generated by the decorrelator. Define $\mathbb{K}$ to be the set of all candidate sequences for the decorrelator. The constraints are: (i) Every sequence in $\mathbb{K}$ should be independent of the sequences in LCG set $\mathbb{X}_{a,m}$. (ii) Any pair of sequences in $\mathbb{K}$ should not be strongly correlated with each other. These are the fundamental guidelines for choosing a suitable decorrelator.

Beyond the theoretical constraints, several practical factors could also be considered to reduce the selection space of the decorrelator when implementing the decorrelation method on FPGAs. First, since the XOR operation could reduce the statistical bias [57], the sequences generated by the decorrelator do not need to have perfect statistical randomness. Second, the decorrelator should be lightweight to be resource-efficient. For example, it is desirable to reduce the number of multiplications that are costly in FPGA resource consumption. Finally, the decorrelator should produce massive uncorrelated sequences, adapting to the different required degree of parallelism.

Based on the theoretical constraints and practical considerations, we adopt the xorshift algorithm [35] as the decorrelator for the following reasons. First, the generation process of xorshift is completely independent of the LCG algorithm, which guarantees the first theoretical constraint. Second, xorshift supports the substream method to generate long-period logical sequences, which can avoid long-range correlations [19] and hence meets the second theoretical constraint. Lastly, as xorshift is based on the binary linear recurrence, it only uses bit-shift and XOR operations that can be efficiently implemented on FPGAs.

3.3 State Sharing Mechanism

The decorrelation method introduced in the previous section solves the correlation issue of the LCG seqences $\mathbb{X}_{a,m}$. To use it on FPGA platforms, each LCG sequence is generated by an independent calculation process, and it requires one multiplication and one addition operation during each step of the state transition. This can require a large number of hardware multipliers in MISRN. To reduce the number of hardware multipliers, we propose the state sharing mechanism that reuses the intermediate results of state transition over distinct generators.

In order to enable intermediate result reuse, we further extend the LCG transition. Considering that ξ_h is an addition transition with a given constant integer h after the LCG transition (given in Equation (3)):

$$w_n = \xi_h(x_n) = (x_n + h) \mod m \tag{20}$$

We can get the transition from w_{n+1} to w_n by expanding the modulus and replacing x_n with Equation (3). That is

$$w_{n+1} = [a \cdot w_n + (l \cdot m + c - a \cdot h)] \mod m, \tag{21}$$

where l is an integer introduced during the expansion of the modulus operation. The transition from w_{n+1} to w_n has the same form as LCG. As the multiplier a is the same as the multiplier in Equation (3), the sequence W generated by Equation (20) belongs to $\mathbb{X}_{a,m}$. This indicates that selecting different h results in a unique sequence, which is the same as changing the increment c of LCG. Hence, we have the following representation:

$$w_{n+1} = \xi_h(\varphi_c(x_n)) = \varphi_{(l \cdot m + c - a \cdot h)}(w_n) \tag{22}$$

where φ_c is the LCG transition with a specific increment c.

Equation (22) allows us to share the output of φ_c in the generation of multiple sequences. We illustrate this state sharing mechanism, as shown in Figure 2. We refer φ_c as the root transition that can be shared among multiple sequences, and ξ_h as the leaf transition. Every leaf transition uses the same root state from a root transition and occupies a unique number h to guarantee that the sequences generated are distinct from others.

Combining with our decorrelation method, the flow of state sharing MISRN generation is as follows. First, the root transition generates an intermediate state at step $n+1$, x_{n+1}. Then, it is shared among p instances. Furthermore, the i-th instance transits to x_{n+1} by a unique leaf transition ξ_{h_i} to get a unique leaf state w^i_{n+1}. Finally, w^i_{n+1} goes through the output stage and then couples with the corresponding output of the decorrelator (illustrated in Section 3.2) to produce a random number z'^i_{n+1}.

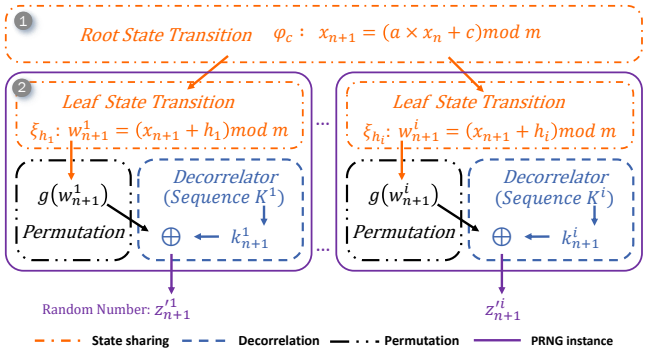


Figure 2: The State sharing mechanism.

To guarantee a maximum period of the sequences generated from the leaf transition, there is a constraint on the selection of h. First, referring to Hull-Dobell Theorem [22], $(l \cdot m + c - a \cdot h)$ must be an odd number. Second, as m is power-of-two, $l \cdot m$ is always even. Hence, we only need to consider the parity of $(c - a \cdot h)$. Again, c, the increment in the root transition, which is also under the constraint of the Hull-Dobell Theorem, is an odd number. Therefore, if $a \cdot h$ is an even number, $(l \cdot m - a \cdot h)$ is an odd number, and the Hull-Dobell Theorem holds. Finally, because a is a prime number, an even h will let the Hull-Dobell Theorem hold to guarantee the maximum period.

Comparing with all existing methods, which need p times multiplication for p distinct instances to transit the state at each step, our state sharing method **needs only one multiplication** along with p addition operations. Specifically, on the FPGA platform, our approach only needs one multiplier to support any number of PRNG instances. This completely resolves the bottleneck of existing methods to increase the number of high-quality PRNG instances to improve the throughput.

3.4 Permutation Function for Output

Since LCG is known to have weak statistical quality of low-order bits [33], ThundeRiNG adopts the random rotation permutation in the output function g as proposed by O'Neill [39]. Basically, it performs a uniform scrambling operation and then remaps the bits to enhance the statistical quality. The remapping operation is a bitwise rotation, of which the number of rotations is determined by the leaf state. As the leaf states are different from each other, the rotation operations of different sequences are different, which can further reduce the collinearity. We demonstrate the impact of the adopted permutation in Section 5.2.2.

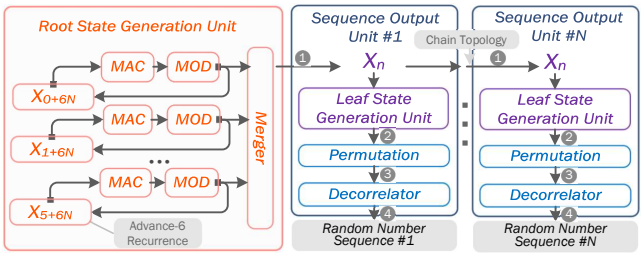


Figure 3: The overview of implementation of ThundeRiNG.

4 IMPLEMENTATION OF THUNDERING

4.1 Architecture Overview

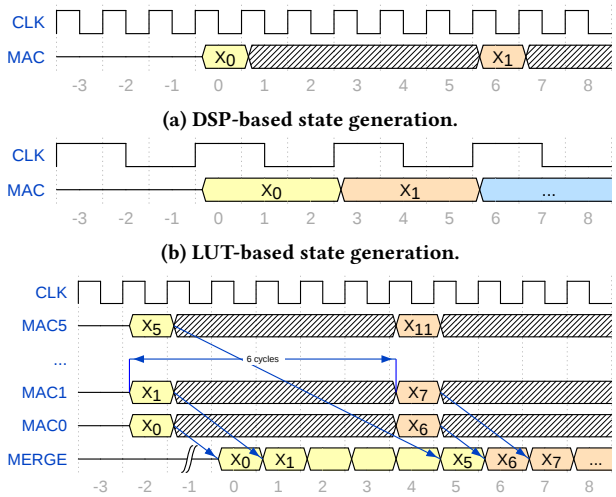
The overall architecture of ThundeRiNG implementation on FPGA is shown in Figure 3. The architecture mainly consists of one *root state generation unit* (**RSGU**) and multiple *sequence output unit* (**SOU**). Each SOU is responsible for generating a single random sequence. RSGU generates one root state per cycle by performing the transition function of LCG and shares it with all SOUs through a daisy-chaining topology [5] interconnection (Step ①). SOU is the main component to realize the decorrelation and state sharing as presented in Section 3. It is composed of three subcomponents, including the *leaf state generation unit* (LSGU), permutation unit, and decorrelator unit. Given the shared root state x_n, each SOU executes the leaf state generation in LSGU to generate a leaf state (Step ②) and then executes the permutation function to generate a distinct LCG sequence (Step ③). Finally, the decorrelator will output the random number sequence with the input from the permutation (Step ④).

4.2 Root State Generation Unit (RSGU)

The RSGU recursively generates root states by the following equation: $x_{n+1} = (a \cdot x_n + c) \mod m$, where the initial x_n is initialized with a random constant. Although the computation of one multiply-accumulate operation (MAC) is rather simple, generating one root state per cycle is nontrivial due to the true dependency [25] introduced by the recursive computation pattern. The calculation of the next state is based on the previous state, which means the calculation of a new state can only be started after the calculation of the previous state is finished. However, the latency of multiplication with DSPs on FPGA usually has multiple cycles (e.g., DSP48E2 takes 6 cycles as indicated in Figure 4(a).) As a result, the throughput is limited by the long latency of multiplication on FPGAs.

Another possible way of finishing the MAC in one cycle is to use logic resources such as LUTs and registers to construct the MAC directly. However, a large-bit multiplier costs a large number of LUTs. Moreover, it consists of many long combinational logic paths that result in a large propagation delay and lead to low frequency. As shown in Figure 4(b), the LUT-based 64-bit MAC can output x_1 after x_0 in one cycle. However, it runs at a much lower frequency, which degrades the generation performance.

Instead, ThundeRiNG hides the long multiplication latency of DSPs by leveraging the step-jump-ahead feature of LCG [6]. Although recursive state generation has flow dependency, LCG supports the arbitrary advance recurrence, which could generate states



(a) DSP-based state generation.

(b) LUT-based state generation.

(c) Proposed state generation with advance-6 recurrence.

Figure 4: The timing diagrams of different hardware designs for root state generation unit.

in a jump-wise manner: $x_{n+i} = f_{adv-i}(x_n)$, where i is an arbitrary integer number and f_{adv-i} is a new recurrence function whose parameters can be derived from the original LCG state transition function (Equation (3)). With it, the process of state generation can be partitioned into multiple portions and executed by multiple hardware units with different lags in parallel. For example, to generate $\{x_0, x_1, ...x_6, x_7...\}$ as the output state sequence, instead of using one state generator, we could use two dependent advance-2 state generators with the first one generating the state sequence $\{x_0, x_2, ..., x_6, ...\}$ and the second one generating the state sequence $\{x_1, x_3, ..., x_7, ...\}$.

The architecture of RSGU is shown in Figure 3. The RSGU consists of multiple independent state generators. Each state generator contains a MAC unit for multiplication operations, a modulus unit for modulus operation, and registers for storing the temporal state during the recurrence process. State generators perform the same f_{adv-i} on adjacent starting positions in parallel. Their outputs are further merged in the order of the original state sequence. Since the latency of state calculation is six cycles, we implement six state generators so that RSGU could generate one state per cycle, as indicated in Figure 4(c).

In addition, our design does not involve critical combinational logic paths, and the evaluation shows that the post-routing frequency can be scaled up to 550MHz using the HLS toolchain. Parameters for advance-i recurrence are calculated in compile-time following the algorithm proposed by Brown et al. [6], which has $O(\log(i))$ complexity. With i equal to 6 in our case, the overhead of calculating these parameters is negligible.

4.3 Sequence Output Unit (SOU)

With the root state as input, each SOU performs leaf state generation (LSGU), permutation, and decorrelation to finally output a random number sequence. Each LSGU consists of an integer adder. The LSGU adder in the i-th SOU performs the addition of the root state with a unique constant value h_i, which is calculated at compile time by the approach described in Section 3.3. The permutation unit is implemented by shift registers. The rotation operation in the permutation function is divided into three stages to reduce the length of the combinatorial logic path. In the first stage, it calculates the the number of required bits for the rotation with the output from LSGU as input. In the second stage, it splits the rotation operation into two small rotations. In the remaining stages, it performs these split rotations in parallel. These stages are executed in a pipelined manner to guarantee a throughput of one output per cycle. The decorrelator, which is a xorshift sequence generator and belongs to the linear feedback shift register generators, is implemented by the shift registers on FPGAs by following these previous works [1, 40].

When increasing the number of SOUs to provide a massive number of sequences, state sharing by simple data duplication may cause a high fan-out problem since all SOUs require the same input from the RSGU. This problem can be optimized by handcrafted register replication, but it loses the flexibility associated with HLS tools [11]. Therefore, we adopt a daisy chain scheme [5] for the internal data transfer that each SOU receives data from the front SOU and then directly forwards the received data to the next SOU. As there is no 1-to-N connection, it can keep the fan-out at a very low level at the cost of a slight increase in output latency. The extra latency is equal to the number of SOUs in the same topology times the period of the execution clock, which is only 1.82μs for 1000 SOUs running at a frequency of 550MHz.

4.4 Discussion

Although ThundeRiNG is specifically designed for FPGAs, we also explore the possibility of generalizing our decorrelation and state sharing methods to CPUs and GPUs with the results presented in Section 5.5. As this generalization is not the main focus of this paper, the implementations on CPUs and GPUs are rather straightforward, and we believe that more optimizations can be considered as future work. For CPU implementation, we utilize a single thread for root state generation and multiple threads for parallel sequence output. The root states are generated in a batch manner so that the root states of each batch can fit in the last level cache for good data locality. In addition, we explore a double buffering scheme to overlap the root state generation and sequence output processes to increase throughput. For the GPU implementation, the state sharing mechanism leverages the shared memory hierarchy and hardware-accelerated arrive/wait barrier [38]. In one stream processor of GPU, we use a single thread for root state generation and multiple threads for parallel sequence output. The synchronization among them is managed through the efficient cooperative group interface provided by CUDA [38].

5 EVALUATION

In this section, we evaluate both the quality and the throughput of ThundeRiNG.

5.1 Experimental Setup

5.1.1 Hardware Platform. The evaluation of statistical quality is conducted on a server with an Intel Xeon 6248R CPU and 768 GB DDR4 memory. The throughput benchmarks and case studies are

conducted on the following hardware platforms and corresponding development environments:

FPGA: Xilinx Alveo U250 accelerator card with Vitis HLS Toolchain 2020.1. The number of available hardware resources are 2,000 BRAMs, 11,508 DSP slices, and 1,341,000 LUTs.

GPU: NVIDIA Tesla P100 GPU with CUDA Toolkit 10.1.

CPU: Two Intel Xeon 6248R CPUs (96 cores after hyperthreading enabled) with oneAPI Math Kernel Library 2021.2.

5.1.2 Parameter setting. The parameters of the root state transition include the modulus m, multiplier a and increment c. According to the existing empirical evaluation [33, 39], we choose modulus m as 2^{64}, multiplier a as 6364136223846793005 and increment c as 54. To guarantee scalability, we choose the xorshift128 generator as the decorrelator since it has the period of $2^{128}-1$ and hence can generate 2^{64} nonoverlapping subsequences which satisfies the decorrelation requirement on 2^{63} distinct sequences [39] of LCG with the state size of 64-bit. With the above parameters, ThundeRiNG is able to generate up to 2^{63} uncorrelated sequences, and the period of each sequence is up to $2^{64}-1$.

5.1.3 Evaluation Methods for Statistical Quality. To our best knowledge, there is no systematic benchmark for MISRN. Thus, we evaluate the quality of the MISRN generated by ThundeRiNG with two kinds of tests: intra-stream correlation and inter-stream correlation. The intra-stream correlation indicates the dependence of random numbers from the same sequence (stream), while the inter-stream correlation indicates the dependence from different sequences.

Evaluation method on intra-stream correlation. Following previous studies [39, 49], we adopt a complete and stringent test suite, TestU01 suite [30], as the empirical test suite for statistical quality measurement. While existing works [2, 32] only conducted tests with the Crush battery, we evaluate ThundeRiNG with the BigCrush battery, which is more extensive and has 64 more tests than the Crush battery [30]. Despite the BigCrush battery testing approximately 2^{38} random samples from one sequence, it can still miss regular patterns with a long period. We hence adopt a complimentary test suite, PractRand [13], which allows for an unlimited test length of one sequence. PractRand runs in iterations. In each iteration, all tests are run at a given sample size. In the next iteration, the sample size is doubled until unacceptable failure occurs. Therefore, it is powerful to detect regular long-range patterns. As ThundeRiNG can generate up to 2^{63} distinct sequences, it is impractical to evaluate them all. Hence, we randomly select 64 distinct sequences for evaluations.

Evaluation method on inter-stream correlation. As TestU01 and PractRand test suits are not designed for testing the inter-stream correlation [23], we adopt the evaluation method from Li et al. [32] that interleaves multiple sequences into one single sequence before evaluating the interleaved sequence with the BigCrush and PractRand test suites. Specifically, the interleaved sequence is generated by selecting numbers from multiple sequences in a round-robin manner. Suppose there are k sequences in total and the i-th sequence is $\{x_0^i, x_1^i, ..., x_n^i, ...\}$, the interleaved sequence will be $\{x_0^0, x_0^1, ..., x_0^k, x_1^0, x_1^1, ...x_1^k, ...\}$. Besides TestU01 and PractRand, we also perform the Hamming weight dependency (HWD) test on the interleaved sequences. HWD, which is the dependency between the number of zeroes and ones in consecutive outputs, has been an important indicator of randomness and adopted by many test suites [30]. We use a powerful HWD testbench from Blackman et al. [4] that several existing crush-resistant PRNGs fail to pass [4].

Beyond this commonly adopted evaluation, to demonstrate the strength of our decorrelation method, we conduct a more stringent analysis on inter-stream correlation by three pairwise correlation evaluations. The experiments on pairwise correlation include Pearson's correlation coefficient, Spearman's rank correlation coefficient, and Kendall's rank correlation coefficient [50].

- The Pearson correlation is also known as cross-correlation, which measures the strength of the linear relationship between two sequences.
- Spearman's rank correlation represents the strength and direction of the monotonic relationship between two variables and is commonly used when the assumptions of Pearson correlation are markedly violated.
- Kendall rank correlation describes the similarity of the ordering of the data when sorted by each of the quantities.

The outcomes of the three pairwise correlation tests range from -1 to +1, where -1 indicates a strong negative correlation, +1 for a strong positive correlation, and 0 for independence. As it is hard to traverse and analyze the pairwise correlations of all candidate sequences, we randomly select a pair of distinct sequences to calculate their coefficients and report the maximal correlation for 1000 such pairs.

5.1.4 Methods for Throughput Evaluation. We first evaluate the throughput of ThundeRiNG by varying the number of PRNG instances, and then comparing it with the state-of-the-art FPGA-based designs as well as CPU/GPU designs. For each experiment, we repeat the execution for 10 times and report the median throughput.

There are, in general, two performance metrics for PRNG throughput evaluation: terabits generated per second (Tb/s), and giga samples generated per second (GSample/s). Tb/s is commonly used in FPGA-based evaluation since FPGA-based PRNGs tend to have a large and arbitrary number of output bits (e.g., LUT-SR [51] uses 624-bit output) to increase the throughput. GSample/s is used in CPU/GPU-based evaluation, where the size of a sample is usually aligned with 32-bit. Hence, we use Tb/s when comparing with other FPGA-based implementations, and GSample/s with the sample size of 32-bit when comparing with CPU/GPU-based implementations. For implementations with a larger sample size, we normalize it to 32-bit correspondingly. For example, Philox-4×32 uses the 128-bit round key [49], which produces 4×32-bit random numbers per output. We will count that as four samples per output, for a fair comparison.

5.2 Quality Evaluation

5.2.1 TestU01 and PractRand. Table 2 shows the testing results of ThundeRiNG and the state-of-the-art PRNG algorithms [4, 29, 39, 49, 51] on BigCrush testing battery and PractRand test suite.

The results indicate that ThundeRiNG passes all tests in the BigCrush battery for both single sequence and multiple sequences. The results of the PractRand suite show ThundeRiNG never encounters a defect even after outputting up to 8 terabytes random numbers. In summary, ThundeRiNG demonstrates a competitive

quality of statistical randomness compared to the state-of-the-art PRNG algorithms.

Table 2: Statistical testing of ThundeRiNG and state-of-the-art PRNG algorithms on BigCrush and PractRand test suites.

Algorithms	Intra-stream correlation		Inter-stream correlation	
	BigCrush	PractRand	BigCrush	PractRand
Xoroshiro128** [4]	Pass	>8TB	Pass	>8TB
Philox4_32 [49]	Pass	>8TB	Pass	1TB
PCG_XSH_RS_64 [39]	Pass	>8TB	105 failures	256MB
MRG32k3a [29]	Pass	>8TB	1 failure	2TB
LUT-SR [51]	2 failures	>1TB	Pass	16MB
ThundeRiNG	**Pass**	**>8TB**	**Pass**	**>8TB**

5.2.2 Pairwise Correlation Evaluation. Table 3 shows the evaluation of pairwise correlation analysis when we enable different techniques: original LCG, original LCG + decorrelation, original LCG + permutation, and ThundeRiNG. The results indicate that the three kinds of correlations of multiple sequences generated by ThundeRiNG are much smaller than those by other design solutions, demonstrating the good statistical randomness of ThundeRiNG.

Table 3: Pairwise correlation tests with different techniqes enabled.

Inter-stream Correlations	LCG Baseline	LCG + Decorrelation	LCG + Permutation	ThundeRiNG
Pearson	0.99764	0.00151	0.00019	**0.00003**
Spearman's rank	0.99764	0.00150	0.00020	**0.00003**
Kendall's rank	0.99843	0.00101	0.00013	**0.00002**

5.2.3 Hamming Weight Dependency Evaluation. Table 4 shows the evaluation results of Hamming weight dependency of different methods. The value of a result of the HWD test indicates the number of generated random numbers before an unexpected pattern is detected. Thus, a higher value of the result means better statistical quality.

We examine the impact of different techniques. If we only apply the permutation to the original LCG, there is no reduction in Hamming weight dependency, although it reduces the collinearity significantly (shown in Table 3). In contrast, our decorrelation method significantly reduces the Hamming weight dependency.

Table 4: Hamming weight dependency test with different techniqes enabled.

Inter-stream Correlations	LCG Baseline	LCG + Decorrelation	LCG + Permutation	ThundeRiNG
Blackman et al. [4]	$1.25e+08$	$> 1e+14$	$1.25e+08$	$>$ **1e + 14**

5.3 Throughput Evaluation on FPGA

We evaluate the throughput and resource consumption of the proposed FPGA implementation of ThundeRiNG. Figure 5 shows the resource consumption and the implementation frequency with the increasing number of PRNG instances. The results show that DSP consumption is less than 1% of the total capacity and, more importantly, it is oblivious to the number of instances. This is because only the root state generation unit that requires the multiplication operation consumes the DSP resource, and ThundeRiNG only needs

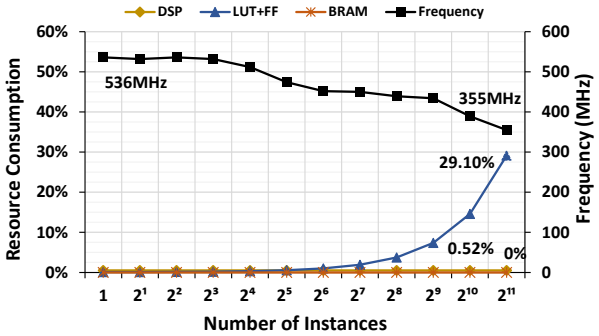


Figure 5: Resources consumption and clock frequency with varying number of SOU instances.

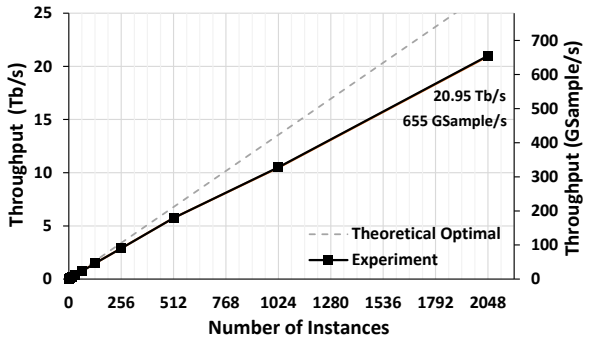


Figure 6: Throughput with varying number of SOU instances.

one root state generation unit for MISRN generation. In addition, ThundeRiNG does not occupy any BRAM resource since the state is small enough to fit into the registers, and thus the BRAM utilization is 0%. The frequency is up to 500MHz and gradually drops as increasing number of instances due to more resource (FF + LUT) consumption.

Figure 6 shows the overall throughput with increasing number of instances, where the solid black line is the measured throughput, and the dashed grey line is the optimal throughput under the frequency of 550MHz. The observed throughput is nearly proportional to the number of instances and can be up to 20.95 Tb/s with 2048 instances. The gap between the optimal line and our results is because of the frequency drop (from 536MHz to 355MHz).

5.3.1 Comparison with FPGA-based Works. Table 5 shows the performance comparison between ThundeRiNG and other state-of-the-art FPGA works as well as implementations with optimistic scaling. We estimate the throughput of the FPGA methods [32, 51] with optimistic scaling, where we assume the number of PRNG instances scales perfectly within the resource capacity, and the implementation frequency is fixed at 500MHz. In addition, we estimate the performance of porting high-quality CPU-based solutions [4, 49] to run on FPGAs. The estimated number of PRNG instances of CPU-based implementation on FPGAs is equal to the resource capacity of the FPGA platform divided by the resource consumption of one PRNG reported by the synthesis of the Vitis tool. We assume they have the 500MHz frequency on FPGA.

Table 5: Throughput, quality test and resource utilization of the state-of-the-art FPGA-based works and porting CPU-based designs to FPGAs.

PRNGs	Quality	Freq. (MHz)	Max #ins.	BRAM (%)	DSP (%)	Thr. (Tb/s)	Sp.
Implementation Benchmarking:							
ThundeRiNG	**Crush-resistant**	**355**	**2048**	**0%**	**0.5%**	**20.95**	1×
Li et al. [32]	Crushable	475	16	1.6%	0%	0.24	87.08×
LUT-SR [51]	Crushable	600	1	0%	0%	0.37	55.9×
Optimistic Scaling:							
Philox4_32 [49]	Crush-resistant	500	442	0%	100%	2.83	7.39×
Xoroshiro128** [4]	Crush-resistant	500	1150	0%	100%	18.40	1.14×
Li et al. [32]	Crushable	500	1000	100%	0%	16.00	1.37×

ThundeRiNG outperforms all other designs significantly, delivering 87.08× and 55.9× speedup over the state-of-the-art FPGA-based solutions [32, 51] while guaranteeing a high quality of randomness. More importantly, while Li et al. [32] achieve a throughput of 16Tb/s with optimistic scaling, ThundeRiNG still has 37% higher throughput. It is also noteworthy that ThundeRiNG consumes no BRAMs while they use up all BRAMs. Even assuming that Philox4_32 [49] and xoroshiro128** [4] are ideally ported to the FPGA platform, ThundeRiNG still delivers 7.39× and 1.15× speedups over them, respectively, with much lower resource consumption.

Table 6: Throughput of various GPU PRNG schemes running on Nvidia Tesla P100 compared to ThundeRiNG's throughput.

Algorithms (cuRAND)	BigCrush	Throughput GSample/s	Throughput Tb/s	ThundeRiNG's Speedup
Philox-4×32 [49]	Pass	61.6234	1.9719	**10.62×**
MT19937 [36]	Pass	51.7373	1.6556	12.65×
MRG32k3a [28]	1 failure	26.2662	0.8405	**24.92×**
xorwow [35]	1 failure	56.6053	1.8114	11.56×
MTGP32 [48]	1 failure	29.1273	0.9321	22.47×

5.4 Comparison with Existing Works on GPUs

We perform throughput and quality comparison with GPU-based PRNGs in cuRAND [38], which is the official library from Nvidia, as shown in Table 6. The statistical test results of cuRAND on the BigCrush battery are collected from the official document [38]. The results show the ThundeRiNG outperforms GPU-based solutions from 10.6× to 24.92×. On the other hand, three of the GPU-based PRNGs fail to pass the BigCrush test, while ThundeRiNG passes all tests. These experiments indicate that ThundeRiNG outperforms cuRAND in both throughput and quality.

5.5 ThundeRiNG on CPU and GPU

As a sanity check, we evaluate the design of ThundeRiNG on the CPU/GPU. Figure 7 compares the throughput of porting the design of ThundeRiNG to CPU/GPU with the state-of-the-art CPU/GPU-based PRNG implementations (Intel MKL and cuRAND). ThundeRiNG did not perform well on the CPU when the number of instances is larger than 2^4 because the overhead of CPU synchronization for state sharing rises dramatically. ThundeRiNG on GPU slightly outperforms cuRAND. To be fair, more optimizations for these implementations are needed in the future. For example, the

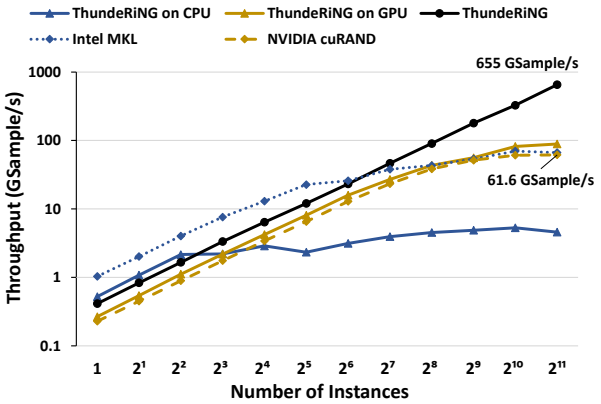


Figure 7: Performance comparison of the compatibility implementations of ThundeRiNG on CPU/GPU, Intel MKL PRNG and NVIDIA cuRAND with varying number of instances.

ThundeRiNG design on the CPU can be improved by utilizing SIMD, and ways can be found to reduce the synchronization overhead.

In summary, because of its fine-grained parallelism and synchronization, the state sharing and decorrelation proposed fits FPGAs better, allowing for the instantiation of many MISRN generators. Consequently, the throughput of ThundeRiNG on FPGA scales linearly with the number of generators and outperforms other designs significantly, even though it runs at a much slower frequency.

6 CASE STUDIES

To further demonstrate the advantages of ThundeRiNG, we apply it to two applications: the estimation of π and Monte Carlo option pricing. Furthermore, we compare the FPGA implementations with the GPU-based ones.

6.1 Implementation

Estimating the value of π is widely used as an application to demonstrate the efficiency of the PRNG [21, 32]. The basic idea is that assuming we have a circle and a square that encloses the circle, the value π can be calculated by dividing the area of the circle by the area of the square. In order to estimate the area of the circle and the square, we generate a large number of random points within the square and count how many of them falling in the enclosed circle. Random number generation is the bottleneck of this application as it consumes 87% of the total execution time of the GPU-based implementation according to our experiment.

Monte Carlo option pricing is commonly used in the mathematical finance domain. It calculates the values of options using multiple sources of random features, such as interest rates, stock prices, or exchange rates. It relies on PRNGs to generate a large number of possible but random price paths for the underlying of derivatives. The final decision is made by computing the associated exercise value of the option for each path. We choose the Black-Scholes model as the target model for option pricing. On the GPU-based implementation, random number generation accounts for 54% of the total execution time, according to our experiment.

We implement two applications on both GPU and FPGA platforms for comparison. For GPU-based implementations, we directly

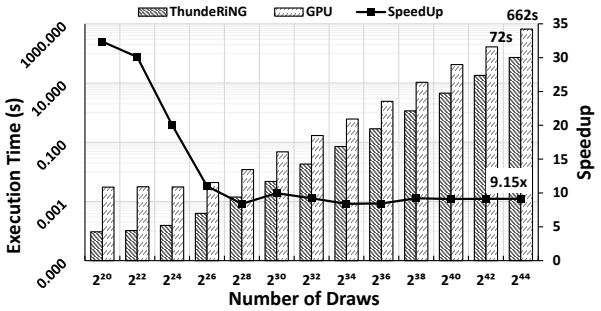


Figure 8: Execution time of estimation of π of FPGA-based solution (ThundeRiNG) and GPU-based solution with varying number of draws.

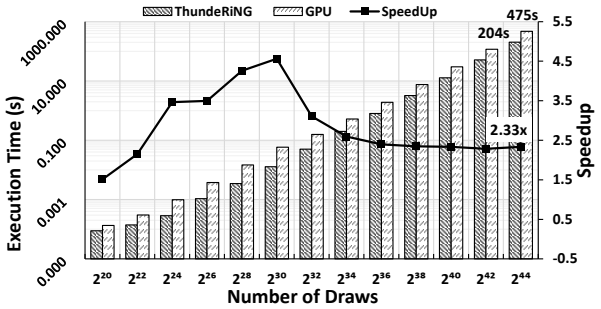


Figure 9: Execution time of Monte Carlo option pricing of FPGA-based solution (ThundeRiNG) and GPU-based solution with varying number of draws.

use the officially optimized designs of Nvidia using the cuRAND library [38], targeting the Nvidia Tesla P100 GPU. For our FPGA-based implementation, we use the ThundeRiNG for random number generation and design the rest of the logic in the application using HLS to achieve the same functionality as the GPU-based designs. For all implementations, single-precision floating points were used as the data type.

6.2 Results

Figure 8 shows the performance of FPGA-based solution (ThundeRiNG) and GPU-based solution on the estimation of $\boldsymbol{\pi}$ with varying the number of draws, where each draw requires two random numbers. The results show FPGA-based solution significantly outperforms GPU-based solution for all number of draws, and the speedup is stable and up to 9.15× for the massive number of draws. The downgrade trend of speedup is because GPU-based implementation cannot fully utilize the hardware capacity when the number of draws is not large.

Figure 9 shows the performance of FPGA-based solution (ThundeRiNG) and GPU-based solution on Monte Carlo option pricing with varying number of draws, each draw requiring a new random number. Our implementation with ThundeRiNG significantly outperforms the GPU-based solution for all number of draws. The speedup of the massive number of draws can be up to 2.33×.

In addition to the comparison on throughput, we also show the resource utilization of the FPGA platform and the power efficiency comparison between GPU and FPGA, as shown in Table 7,

Table 7: The comparison of throughput and power efficiency of two applications between FPGA and GPU.

Applications		Estimation of π	MC option pricing
FPGA: Alveo U250 (16nm FinFET)	Frequency (MHz)	304	335
	Number of instances	1600	256
	LUTs	1048235(70%)	735173(49%)
	FFs	1171130(38%)	751810(24%)
	DSPs	5512(45%)	5984(49%)
	Throughput (GSample/s)	480	86
	Power consumption (W)	45	43
GPU: Tesla P100 (16nm FinFET)	Frequency (MHz)	1,190	1,190
	Throughput (GSample/s)	53	33
	Power consumption (W)	131	126
ThundeRiNG's improvement	Throughput speedup	**9.15x**	**2.33x**
	Power efficiency	**26.63x**	**6.83x**

where the power consumption is reported by respective official tools, namely `nvidia-smi` for GPU and `xbutil` for FPGA, and the power efficiency is calculated by dividing the throughput by the power consumption. The results show the FPGA-based solutions outperform the GPU-based solutions by 6.83× and 26.63× for MC option pricing and the estimation of π, respectively. The end-to-end comparison of the two applications demonstrates that ThundeRiNG is able to generate massive independent random numbers with high throughput, and FPGA can be a promising platform for PRNG involved applications.

7 CONCLUSION

In this paper, we propose the first high-throughput FPGA-based crush-resistant PRNG called ThundeRiNG for generating multiple independent sequences of random numbers. Theoretical analysis shows that our decorrelation method can enable the concurrent generation of high-quality random numbers. By sharing the state, ThundeRiNG uses a constant number of multipliers and BRAM regardless of the number of sequences to be generated. Our results show that ThundeRiNG outperforms all current FPGA and GPU based pseudo-random number generators significantly in performance as well as quality of the output. Furthermore, ThundeRiNG is designed to be used as a 'plug-and-play' IP block on FPGAs for developer convenience. We believe that our work contributes to making the FPGA a promising platform for high performance computing applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback on this work. We thank the Xilinx Adaptive Compute Cluster (XACC) Program [56] for the generous donation. This work is supported by MoE AcRF Tier 1 grant (T1 251RES1824), Tier 2 grant (MOE2017-T2-1-122) in Singapore, and also partially supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

REFERENCES

[1] M. Bakiri, J. Couchot, and C. Guyeux. 2018. CIPRNG: A VLSI Family of Chaotic Iterations Post-Processings for $\mathbb{F}_2$ -Linear Pseudorandom Number Generation Based on Zynq MPSoC. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 5 (2018), 1628–1641. https://doi.org/10.1109/TCSI.2017.2754650

[2] Mohammed Bakiri, Christophe Guyeux, Jean-François Couchot, and Abdelkrim Kamel Oudjida. 2018. Survey on hardware implementation of random

number generators on FPGA: Theory and experimental analyses. *Computer Science Review* 27 (2018), 135–153.

[3] Lawrence E Bassham III, Andrew L Rukhin, Juan Soto, James R Nechvatal, Miles E Smid, Elaine B Barker, Stefan D Leigh, Mark Levenson, Mark Vangel, David L Banks, et al. 2010. *Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications*. National Institute of Standards & Technology.

[4] David Blackman and Sebastiano Vigna. 2018. Scrambled linear pseudorandom number generators. *arXiv preprint arXiv:1805.01407* (2018).

[5] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2018. You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11, 3 (2018), 1–23.

[6] Forrest B Brown. 1994. Random number generation with arbitrary strides. *Transactions of the American Nuclear Society* 71, CONF-941102- (1994).

[7] Karen H Brown. 1994. Security requirements for cryptographic modules. *Fed. Inf. Process. Stand. Publ* (1994), 1–53.

[8] Pawel Dabal and Ryszard Pelka. 2012. FPGA implementation of chaotic pseudo-random bit generators. In *Proceedings of the 19th International Conference Mixed Design of Integrated Circuits and Systems-MIXDES 2012*. IEEE, 260–264.

[9] Pawel Dabal and Ryszard Pelka. 2014. A study on fast pipelined pseudo-random number generator based on chaotic logistic map. In *17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*. IEEE, 195–200.

[10] Ishaan L. Dalal and Deian Stefan. 2008. A Hardware Framework for the Fast Generation of Multiple Long-Period Random Number Streams. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '08)*. Association for Computing Machinery, New York, NY, USA, 245–254. https://doi.org/10.1145/1344671.1344707

[11] Johannes de Fine Licht, Simon Meierhans, and Torsten Hoefler. 2018. Transformations of high-level synthesis codes for high-performance computing. *arXiv preprint arXiv:1805.08288* (2018).

[12] Lih-Yuan Deng, Jyh-Jen Horng Shiau, and Henry Horng-Shing Lu. 2012. Large-order multiple recursive generators with modulus 231- 1. *INFORMS Journal on Computing* 24, 4 (2012), 636–647.

[13] C Doty-Humphrey. [n.d.]. PractRand official site (2018). *URL http://pracrand. sourceforge. net* ([n. d.]).

[14] Mark J Durst. 1989. Using linear congruential generators for parallel random number generation. In *Proceedings of the 21st conference on Winter simulation*. 462–466.

[15] Karl Entacher, Andreas Uhl, and Stefan Wegenkittl. 1999. Parallel random number generation: long-range correlations among multiple processors. In *International Conference of the Austrian Center for Parallel Computation*. Springer, 107–116.

[16] Graham Everest, Alfred Jacobus Van Der Poorten, Igor Shparlinski, Thomas Ward, et al. 2003. *Recurrence sequences*. Vol. 104. American Mathematical Society Providence, RI.

[17] Roger G Ghanem and Pol D Spanos. 2003. *Stochastic finite elements: a spectral approach*. Courier Corporation.

[18] Oded Goldreich, Noam Nisan, and Avi Wigderson. 1995. On Yao's XOR lemma. Technical Report TR95–050, Electronic Colloquium on Computational Complexity.

[19] Hiroshi Haramoto, Makoto Matsumoto, Takuji Nishimura, François Panneton, and Pierre L'Ecuyer. 2008. Efficient Jump Ahead for F2-Linear Random Number Generators. *INFORMS Journal on Computing* 20, 3 (2008), 385–390.

[20] Peter Hellekalek. 1998. Don't trust parallel Monte Carlo! *ACM SIGSIM Simulation Digest* 28, 1 (1998), 82–89.

[21] Lee Howes and David Thomas. 2007. Efficient random number generation and application using CUDA. *GPU gems* 3 (2007), 805–830.

[22] Thomas E Hull and Alan R Dobell. 1962. Random number generators. *SIAM review* 4, 3 (1962), 230–254.

[23] Chester Ismay. 2013. *Testing Independence of Parallel Pseudorandom Number Streams: Incorporating the Data's Multivariate Nature*. Arizona State University.

[24] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 745–761. https://www.usenix.org/conference/osdi18/presentation/iyer

[25] Ken Kennedy and John R. Allen. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[26] Agnan Kessy, Alex Lewin, and Korbinian Strimmer. 2018. Optimal whitening and decorrelation. *The American Statistician* 72, 4 (2018), 309–314.

[27] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA.

[28] Pierre L'Ecuyer. 1996. Combined multiple recursive random number generators. *Operations research* 44, 5 (1996), 816–822.

[29] Pierre L'ecuyer. 1999. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research* 47, 1 (1999), 159–164.

[30] Pierre L'Ecuyer and Richard Simard. 2007. TestU01: AC library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)* 33, 4 (2007), 1–40.

[31] Derrick H Lehmer. 1951. Mathematical methods in large-scale computing units. *Annu. Comput. Lab. Harvard Univ.* 26 (1951), 141–146.

[32] Yuan Li, Paul Chow, Jiang Jiang, Minxuan Zhang, and Shaojun Wei. 2013. Software/Hardware Parallel Long-Period Random Number Generation Framework Based on the WELL Method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 5 (2013), 1054–1059.

[33] Pierre L'ecuyer. 1999. Tables of linear congruential generators of different sizes and good lattice structure. *Math. Comp.* 68, 225 (1999), 249–260.

[34] George Marsaglia. 1995. The diehard test suite, 1995. *URL http://stat. fsu. edu/~ geo/diehard. html* (1995).

[35] George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software, Articles* 8, 14 (2003), 1–6. https://doi.org/10.18637/jss.v008.i14

[36] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, 1 (1998), 3–30.

[37] Makoto Matsumoto and Takuji Nishimura. 2000. Dynamic creation of pseudorandom number generators. In *Monte-Carlo and Quasi-Monte Carlo Methods 1998*. Springer, 56–69.

[38] Nvidia. 2020 (accessed November 13, 2020). *CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/curand/device-api-overview.html#performance-notes

[39] Melissa E O'Neill. 2014. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Trans. Math. Software* (2014).

[40] Amit Kumar Panda, Praveena Rajput, and Bhawna Shukla. 2012. FPGA implementation of 8, 16 and 32 bit LFSR with maximum length feedback polynomial using VHDL. In *2012 International Conference on Communication Systems and Network Technologies*. IEEE, 769–773.

[41] François Panneton, Pierre L'ecuyer, and Makoto Matsumoto. 2006. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software (TOMS)* 32, 1 (2006), 1–16.

[42] Aduri Pavan, Srikanta Tirthapura, et al. 2013. Counting and sampling triangles from a graph stream. (2013).

[43] Ora E Percus and Malvin H Kalos. 1989. Random number generators for MIMD parallel processors. *Journal of parallel and distributed computing* 6, 3 (1989), 477–497.

[44] SJ Plimpton, SG Moore, A Borner, AK Stagg, TP Koehler, JR Torczynski, and MA Gallis. 2019. Direct simulation Monte Carlo on petaflop supercomputers and beyond. *Physics of Fluids* 31, 8 (2019), 086101.

[45] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetze, Volker Hilt, and Thorsten Strufe. 2017. Approximate Stream Analytics in Apache Flink and Apache Spark Streaming. *arXiv preprint arXiv:1709.02946* (2017).

[46] Tiago J Rato and Marco S Reis. 2013. Advantage of using decorrelated residuals in dynamic principal component analysis for monitoring large-scale systems. *Industrial & Engineering Chemistry Research* 52, 38 (2013), 13685–13698.

[47] Reuven Y Rubinstein and Dirk P Kroese. 2016. *Simulation and the Monte Carlo method*. Vol. 10. John Wiley & Sons.

[48] Mutsuo Saito. 2010. A Variant of Mersenne Twister Suitable for Graphic Processors. *CoRR* abs/1005.4973 (2010). arXiv:1005.4973 http://arxiv.org/abs/1005.4973

[49] John K Salmon, Mark A Moraes, Ron O Dror, and David E Shaw. 2011. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[50] Richard Taylor. 1990. Interpretation of the correlation coefficient: a basic review. *Journal of diagnostic medical sonography* 6, 1 (1990), 35–39.

[51] David B Thomas and Wayne Luk. 2012. The LUT-SR family of uniform random number generators for FPGA architectures. *IEEE transactions on very large scale integration (vlsi) systems* 21, 4 (2012), 761–770.

[52] Hubbul Walidainy and Zulfikar Zulfikar. 2015. An improved design of linear congruential generator based on wordlengths reduction technique into FPGA. *International Journal of Electrical and Computer Engineering* 5, 1 (2015), 55.

[53] Martin Weigel. 2018. Monte Carlo methods for massively parallel computers. In *Order, Disorder and Criticality: Advanced Problems of Phase Transition Theory*. World Scientific, 271–340.

[54] Wikipedia contributors. 2020. Statistical randomness — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Statistical_randomness&oldid=948430929 [Online; accessed 29-October-2020].

[55] Pei-Chi Wu and Kuo-Chan Huang. 2006. Parallel use of multiplicative congruential random number generators. *Computer Physics Communications* 175, 1 (2006), 25–29. https://www.sciencedirect.com/science/article/pii/S0010465506001007

[56] Xilinx. 2020. Xilinx Adaptive Compute Cluster (XACC) Program. https://www.xilinx.com/support/university/XUP-XACC.html.

[57] Andrew C Yao. 1982. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*. IEEE, 80–91.