Exploiting Uniform Spatial Distribution to Design Efficient Random Number Source for Stochastic Computing

Kuncai Zhong[1], Zexi Li[1], Haoran Jin[1], Weikang Qian[1,2,*]
[1]University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, China
[2]MoE Key Lab of Artificial Intelligence, Shanghai Jiao Tong University, China
{kczhong,lzx12138,allenjin,qianwk}@sjtu.edu.cn

ABSTRACT

Stochastic computing (SC) generally suffers from long latency. One solution is to apply proper random number sources (RNSs). Nevertheless, current RNS designs either have high hardware cost or low accuracy. To address the issue, motivated by that the uniform spatial distribution generally leads to a high accuracy for an SC circuit, we propose a basic architecture to generate the uniform spatial distribution and a further detailed implementation of it. For the implementation, we further propose a method to optimize its hardware cost and a method to optimize its accuracy. The method for hardware cost optimization can optimize the hardware cost without affecting the accuracy. The experimental results show that our proposed implementation can achieve both low hardware cost and high accuracy. Compared to the state-of-the-art stochastic number generator design, the proposed design can reduce 88% area with close accuracy.

KEYWORDS

stochastic computing, random number source, uniform spatial distribution

1 INTRODUCTION

Stochastic computing (*SC*), an unconventional computing paradigm proposed in 1960s [1], has attracted much attention in recent years [2]. It does complex computation by simple circuits based on stochastic bit streams, which encode values by the ratios of ones [3]. For example, SC can implement multiplication by a single AND gate. SC also has strong fault tolerance. Owing to these advantages, SC has been successfully applied in many domains, such as image processing [4] and neural networks [5].

However, to ensure high accuracy, SC generally requires long stochastic bit streams, which lead to long latency and high energy consumption. Note that a stochastic bit stream is generated by a *stochastic number generator* (*SNG*) in an SC circuit, and a main component in an SNG that controls the randomness is a *random number source* (*RNS*).[1] Thus, one solution to the problem of long stochastic bit stream is to design a proper RNS.

To this end, there have been many novel RNS designs proposed [6–9]. They achieve high accuracy by generating random outputs that are properly distributed. For example, the Sobol sequence generators are proposed to generate low discrepancy sequences, which are uniformly distributed in the time domain [7]. However, these RNS designs generally suffer from high hardware cost and occupy most hardware cost of an SC circuit. For example, for a 2-input SC multiplier, 2 8-bit Sobol sequence generators can even occupy 95.7% area. Thus, it is crucial to design efficient RNSs with high accuracy and low hardware cost, which remains as an open problem.

An SC circuit with m inputs needs m RNSs, which output m random binary numbers in a clock cycle. The m numbers can be treated as a point in an m-dimensional space Ω, which we refer to as an *output point*. The output points generated over all the clock cycles within a computation period form a distribution in the space Ω. In the following, we call the distribution the *spatial distribution of the RNS outputs*, or *spatial distribution* for short. An example of a random spatial distribution of the RNS outputs for $m = 2$ is shown in Fig. 1(a). For an SC circuit, an output point of its RNSs determines one bit in the output stochastic bit stream of the circuit, and a spatial distribution determines the entire output stochastic bit stream and hence, the accuracy of the circuit. Therefore, in order to achieve a high accuracy, we need to design RNSs generating a good spatial distribution.

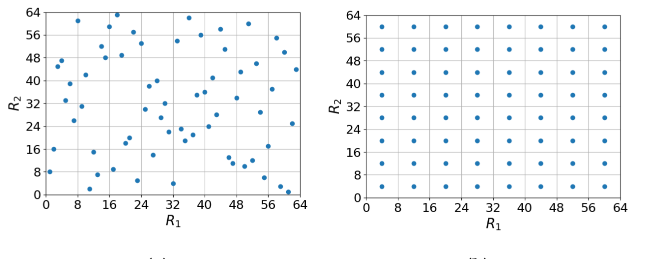


Figure 1: Example of the spatial distribution of the RNS outputs for $m = 2$: (a) random distribution; (b) uniform distribution.

A prior work points out that a uniform spatial distribution will generally lead to a high accuracy for an SC circuit [6]. An example of a uniform spatial distribution is shown in Fig. 1(b), where each grid has exactly one point. Nevertheless, the work [6] only gives a preliminary discussion about it. To the best of our knowledge, there is still no systematic study on how to generate a uniform spatial

[*]Corresponding author: Weikang Qian.

[1]We will describe them in detail in Section 2.

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549396

distribution with low hardware cost, and there is even no formal definition about it in the context of SC.

In this work, to address the above issues, we systematically study the uniform spatial distribution and exploit it to design low-cost high-accuracy RNSs. Our main contributions are as follows.

- We give a formal definition of the uniform spatial distribution and a necessary and sufficient condition for generating it (see Section 3). We also show its difference from the traditional notion of *uniform distribution*.
- We propose a basic architecture and an implementation based on linear feedback shift registers (LFSRs) to satisfy the necessary and sufficient condition (see Section 4). The output points generated by the proposed implementation form a uniform spatial distribution.
- We propose methods to optimize the hardware cost and the accuracy of the proposed implementation (see Section 5). The hardware cost optimization method optimizes the cost of the proposed implementation without affecting the accuracy.

The experimental results show that our proposed implementation achieves both low hardware cost and high accuracy. Compared to the state-of-the-art SNG design, with close accuracy, the proposed design can achieve 88% area reduction.

2 BACKGROUND AND RELATED WORKS

In this section, we introduce the background on SC circuits and the related works on reducing the hardware cost of SNGs and improving the accuracy of RNSs.

2.1 Stochastic Computing Circuits

To ensure a high computation accuracy, an SC circuit usually requires its stochastic bit streams to be independent or with low correlation. Thus, as shown in Fig. 2, an m-input SC circuit generally has m SNGs, which produce m independent stochastic bit streams. The SC circuit also has an SC core that computes on the bit streams. As shown in Fig. 2, an SNG typically consists of an RNS and a comparator, where the RNS generates a random binary number R in a clock cycle and the comparator compares R with the input binary number X to generate a bit 1 when $R < X$, and 0 otherwise.

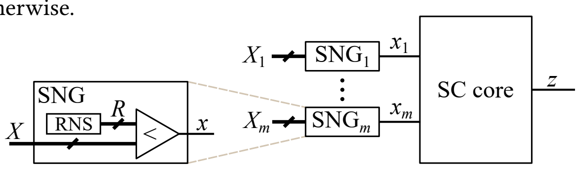


Figure 2: General architecture for an m-input SC circuit.

2.2 Methods to Reduce the Hardware Cost of SNGs

SNGs usually occupy most area of an SC circuit [10]. It is crucial to reduce the hardware cost of SNGs. There are several methods proposed for this purpose, such as inserting D flip-flops (DFFs) and sharing RNSs. By inserting DFFs, we can duplicate a stochastic bit stream and delay it for some clock cycles to generate a new stochastic bit stream, which has the same value as the original one but is uncorrelated to it [1]. Therefore, it helps reduce the number of SNGs used to generate stochastic bit streams of the same value [11, 12]. Sharing RNSs is the method to share an RNSs for different SNGs. It applies circular shift or scrambling to generate different random binary numbers for different SNGs in a clock cycle [13, 14]. This method usually reduces the number of RNSs to 1 and achieves the minimum hardware cost for RNSs. However, these two methods usually degrade the accuracy.

2.3 Methods to Improve the Accuracy of RNSs

Generally speaking, there are two types of methods to improve the accuracy of the RNSs.

The first type is to design better RNSs [6–9, 15–20]. To obtain accurate results, an RNS that performs deterministic computation for SC is proposed [15]. However, it needs very long stochastic bit streams. To achieve high accuracy with short stochastic bit streams, several novel RNS designs are proposed to produce properly-distributed random number sequences based on low discrepancy sequences [6, 7, 16, 18, 20] or pseudo-random sequences [8, 9, 17]. However, different from our work, these works mainly focus on producing a random number sequence properly distributed in the *time domain* and do not consider generating a good spatial distribution of the RNS outputs [6–9, 16–18]. Furthermore, they either suffer from high hardware cost [6–9, 17–19] or are only applicable for some special circuits like multipliers [16].

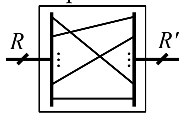


Figure 3: Illustration of scrambling.

The second type is to change the outputs of an RNS. Typical examples are seeding and scrambling [21]. Seeding is applied to LFSR-based RNS. It is to choose a proper initial value, known as a *seed*, for an LFSR. Under different seeds, the output sequences of an LFSR differ. Scrambling is to permute the output of an RNS from R to R', as shown in Fig. 3. By applying scrambling, the output sequence of an RNS also changes. By applying these methods to some RNSs of an SC circuit, the spatial distribution of the RNS outputs changes and hence, the accuracy may improve. Furthermore, these methods can be implemented with no hardware overhead. However, the existing works on these two methods do not discuss which spatial distribution is good and how to generate it with the two methods.

3 DEFINITION AND CONDITION FOR THE UNIFORM SPATIAL DISTRIBUTION

In this section, we first formally define the uniform spatial distribution and compare it with the random spatial distribution. Then, based on the definition, we derive a necessary and sufficient condition for the uniform spatial distribution.

3.1 Uniform Spatial Distribution

Consider m n-bit RNSs. They output 2^n different output points in the period of 2^n cycles. We denote each point as $(R_1, R_2, \ldots, R_m)$, where R_i is the output of the i-th RNS and ranges from 0 to $2^n - 1$. To define the uniform spatial distribution on these 2^n points, we first uniformly split the m-dimensional space $\Omega = [0, 2^n - 1]^m$ into 2^n grids. For the special case where n is a multiple of m, each dimension is split into $2^{\frac{n}{m}}$ uniform intervals. An example for $n = 6$

and $m = 2$ is shown in Fig. 1(b). For the general case, assume that $n = m\lfloor \frac{n}{m} \rfloor + p$, where $0 \le p < m$. Without loss of generality, we choose the first p dimensions and split each of them into $2^{\lceil \frac{n}{m} \rceil}$ uniform intervals, and for the rest $(m - p)$ dimensions, we split each of them into $2^{\lfloor \frac{n}{m} \rfloor}$ uniform intervals.

Given 2^n output points generated in the period of 2^n cycles, we say that they form a *uniform spatial distribution* if each of the 2^n grids has exactly one output point in it. Fig. 1(b) shows an example of a uniform spatial distribution on 2^6 points.

We remark that the proposed uniform spatial distribution is different from the uniform distribution in the time domain, which is a sequence of random numbers $r_1, r_2, \ldots$, such that for any $i \ge 1$, r_i is a random number uniformly distributed in a given range. In the context of SC, the range is typically $\{0, 1, \ldots, 2^n - 1\}$. To generate such a uniform distribution, there have been several RNS designs proposed [6–9, 15–20] to randomly generate a number in the range with the same probability as $\frac{1}{2^n}$. However, this type of uniform distributions only focuses on a single RNS and does not consider the spatial distribution of the output points of m RNSs [18]. Therefore, it cannot ensure a high accuracy for the SC circuits. In the context of SC, it is more essential to study the spatial distribution.

Next, we compare the uniform spatial distribution with the random spatial distribution to show its strength. The prior work [6], which is the most related work to ours, has done a preliminary comparison of these two distributions by just using a special example. In contrast, we perform a more comprehensive comparison here by considering a large number of random samples to draw a more solid conclusion. Specifically, we consider a 2-input SC multiplier and 1000 random spatial distributions and 1000 uniform spatial distributions. For each distribution, we randomly generate 1000 input pairs and calculate the mean absolute error (MAE) over these 1000 pairs as follows

$$MAE = \frac{1}{1000} \sum_{i=1}^{1000} \left| f(a_i, b_i) - \frac{a_i \times b_i}{2^{2n}} \right|, \quad (1)$$

where a_i and b_i are the inputs in the i-th input pair and $f(a_i, b_i)$ is the output of the 2-input SC multiplier with a_i and b_i as the inputs.

We compare the minimum, the maximum, and the average MAEs for the two types of spatial distributions over the 1000 samples. The comparison is shown in Table 1 for $n = 6, 7, 8$, where Min, Max, and Ave denote the minimum, the maximum, and the average MAEs, respectively. Clearly, the uniform spatial distribution has much lower error than the random spatial distribution. Therefore, it is desirable for m RNSs to produce a uniform spatial distribution.

Table 1: MAE comparison for different distributions.

n	Random			Uniform		
	Min	Max	Ave	Min	Max	Ave
6	0.016	0.094	0.034	0.010	0.017	0.013
7	0.011	0.062	0.024	0.0063	0.010	0.0080
8	0.0081	0.045	0.017	0.0039	0.0057	0.0047

3.2 A Necessary and Sufficient Condition for Uniform Spatial Distribution

Suppose that the i-th $(1 \le i \le m)$ dimension of the space Ω has 2^{j_i} intervals. According to our space partition rule mentioned in Section 3.1, we have $j_i = \lceil \frac{n}{m} \rceil$ for $1 \le i \le p$ and $j_i = \lfloor \frac{n}{m} \rfloor$ for $p+1 \le i \le m$. Consider a grid with the i-th dimension mapped to the D_i-th $(1 \le i \le m, 0 \le D_i < 2^{j_i})$ interval in the i-th dimension of the space Ω. Then, the grid can be uniquely indexed by a combination as $(D_1, D_2, \ldots, D_m)$. Since $0 \le D_i < 2^{j_i}$, D_i can be represented as a j_i-bit binary number. Since $\sum_{i=1}^{m} j_i = n$, by cascading the binary representations of $D_1, D_2, \ldots, D_m$ in sequence, we will obtain an n-bit binary number, denoted as $C(D_1, \ldots, D_m)$. Therefore, each grid can be uniquely indexed by the binary number $C(D_1, \ldots, D_m)$. For example, for the bottom-left grid in Fig. 1(b), it can be indexed by a combination $(D_1, D_2) = ((000)_2, (000)_2)$ or a 6-bit binary number $C(D_1, D_2) = (000000)_2$.

Now consider an output point $P = (R_1, R_2, \ldots, R_m)$. If the j_i most significant bits (MSBs) of R_i form a binary number D_i, then the point is in the D_i-th interval in the i-th dimension of the space Ω. In the following, we call the j_i MSBs of R_i its *leading bits* and the rest bits of R_i its *trailing bits*. Therefore, the index $C(D_1, \ldots, D_m)$ of the grid where the point P locates equals the cascading of the leading bits of $R_1, \ldots, R_m$, denoted as $L(R_1, \ldots, R_m)$, or $L(P)$ for short. For example, for the output point $(R_1, R_2) = ((000100)_2, (000100)_2)$ in Fig. 1(b), the cascading of the leading bits of R_1 and R_2 is $(000000)_2$. Therefore, it is located in the bottom-left grid.

By the above analysis and the definition of the uniform spatial distribution, we can reach the following necessary and sufficient for the uniform spatial distribution.

Claim 1. *2^n output points $P_1, \ldots, P_{2^n}$ generated by the RNSs in the period of 2^n cycles form a uniform spatial distribution if and only if the set $\{L(P_1), \ldots, L(P_{2^n})\}$ equals the set $\{0, 1, \ldots, 2^n - 1\}$.*

4 BASIC ARCHITECTURE AND IMPLEMENTATION

In order to generate a uniform distribution, we only need to satisfy the necessary and sufficient condition stated in Claim 1. In this section, we first propose a basic architecture to satisfy the condition. Then, we propose a detailed implementation of it using LFSRs.

4.1 Basic Architecture

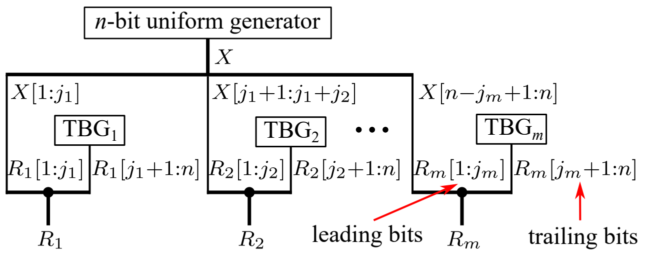


Figure 4: The basic architecture for generating the uniform spatial distribution, where TBG$_i$ denotes the i-th trailing-bit generator.

Our proposed basic architecture that satisfies the necessary and sufficient condition in Claim 1 is shown in Fig. 4. In the figure, the notation $N[j : k]$ denotes the j-th to the k-th MSBs of a binary number N. For example, $R_i[1 : j_i]$ denotes the 1-st to the j_i-th MSBs of R_i, i.e., the leading bits of R_i. The architecture has an n-bit *uniform generator*, which is a binary number generator that generates all n-bit binary numbers in the period of 2^n. Examples include an n-bit counter and an n-bit pseudo-random number generator.

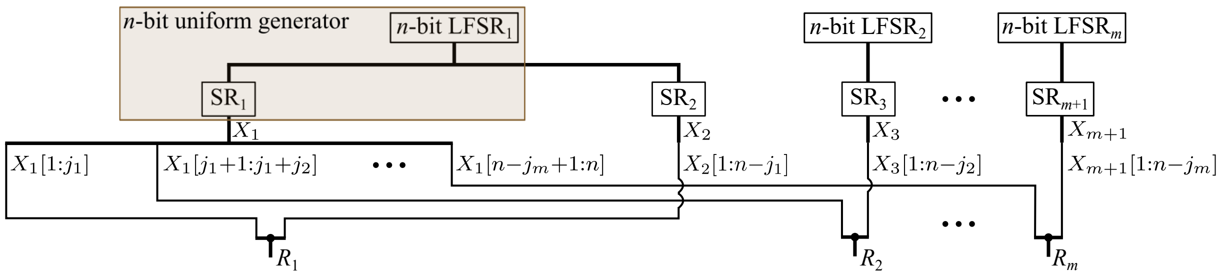


Figure 5: Implementation to generate the uniform spatial distribution, where SR_i denotes the i-th scrambling module.

The architecture uses the n-bit uniform generator to produce an n-bit binary number X in each clock cycle. To generate an output point $P = (R_1, R_2, \ldots, R_m)$, we apply the $(\sum_{k=1}^{i-1} j_k + 1)$-th to the $(\sum_{k=1}^{i} j_k)$-th MSBs of X as the leading bits of R_i, for $1 \leq i \leq m$. Also, the architecture includes m *trailing-bit generators* (i.e., TBG$_i$'s), which are used to generate the trailing bits of R_i's. By this design, we have $L(P) = X$. According to the definition of the n-bit uniform generator, we further have $\{L(P_1), \ldots, L(P_{2^n})\} = \{0, 1, \ldots, 2^n - 1\}$ for 2^n output points $P_1, \ldots, P_{2^n}$ generated by the architecture in a period of 2^n cycles. Thus, by Claim 1, the basic architecture produces a uniform spatial distribution.

4.2 Proposed Implementation Based on LFSR

In this section, we proposed a detailed implementation of the basic architecture using LFSRs. LFSR is the most widely used RNS with low hardware cost. An n-bit LFSR outputs all positive n-bit integers in the period of $(2^n - 1)$ cycles. Ignoring that it does not output 0, we can roughly treat an LFSR as a uniform generator. Furthermore, if we scramble the output bits of an n-bit uniform generator by the scrambling module shown in Fig. 3, we still get all n-bit binary numbers in the period of 2^n. Therefore, a uniform generator with its outputs scrambled is still a uniform generator.

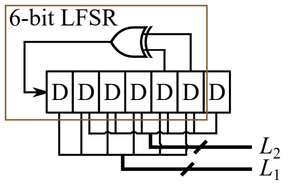
Based on the above discussion, we propose a detailed implementation shown in Fig. 5, where SR$_i$ denotes the i-th scrambling module. It applies an LFSR (i.e., LFSR$_1$) with its outputs scrambled (i.e., by the module SR$_1$) as the uniform generator to generate the leading bits of $R_1, R_2, \ldots, R_m$. To generate the trailing bits of $R_1, \ldots, R_m$, we introduce $(m - 1)$ extra LFSRs (i.e., LFSR$_2, \ldots,$ LFSR$_m$) and m extra scrambling modules (i.e., SR$_2, \ldots,$ SR$_{m+1}$). For each $1 \leq i \leq m$, we apply the 1-st to the $(n - j_i)$-th MSBs of the outputs of LFSR$_i$ scrambled by SR$_{i+1}$ as the trailing bits of R_i. We note that to reduce the hardware cost, LFSR$_1$ are used twice to generate the leading bits of $R_1, \ldots, R_m$ and the trailing bits of R_1, respectively. Based on this implementation, we can generate a uniform spatial distribution. Note that a scrambling module is logic-free. Hence, the main hardware cost of the proposed implementation is that of m LFSRs.

5 IMPLEMENTATION OPTIMIZATION

The proposed implementation has several configuration parameters. They include 1) the feedback polynomials of the m LFSRs, 2) the seeds of the m LFSRs, and 3) the scrambling ways of the $(m+1)$ scrambling modules. Different configurations of them will lead to different hardware costs. Furthermore, although the proposed implementation guarantees to produce a uniform spatial distribution, which typically leads to a high accuracy, we can even tune its configuration to maximize the accuracy. Thus, in this section, we further optimize the proposed implementation in both hardware cost and accuracy by considering these configuration parameters.

5.1 Hardware Cost Optimization

Hardware optimization of the proposed implementation is mainly determined by the feedback polynomials of the m LFSRs. As shown in [11], for 2 LFSRs, if they have the same feedback polynomial but different seeds, they can be implemented by realizing one LFSR first and then the other by inserting DFFs after the first. An example of implementing 2 6-bit LFSRs with the same feedback polynomial but different seeds is shown in Fig. 6. Note that in this example, by the selected seeds, the output sequence of L_2 is a delayed version of that of L_1 by 1 clock cycle. Thus, we can implement L_2 by inserting a single DFF after L_1, reducing the total hardware cost. Inspired by this, *we let all the m LFSRs have the same feedback polynomial to reduce the hardware cost.* We note that although this design choice significantly limits the configuration space, we still have many other parameters for configuration to reach a high accuracy.



Clock cycle	1	2	3	4	⋯
L_1	7	3	1	32	⋯
L_2	14	7	3	1	⋯

Figure 6: Implementing 2 6-bit LFSRs with the same feedback polynomial but different seeds by 1 6-bit LFSR and 1 DFF.

Under the above design choice, we further propose a method to optimize the hardware cost. The optimization depends on the seeds of the m LFSRs, which are obtained by the method in Section 5.2. Assume that the seed of LFSR$_i$ $(1 \leq i \leq m)$ is S_i. For convenience, we also introduce a reference LFSR LFSR$_{ref}$, which has the same feedback polynomial as these LFSRs and its seed as 1. For any value $1 \leq X \leq 2^n - 1$, we assume that it is produced by the reference LFSR LFSR$_{ref}$ in the clock cycle $T(X)$, where $1 \leq T(X) \leq 2^n - 1$. We assume that the seed (*i.e.*, the initial value) of an LFSR is produced at the clock cycle 1. Thus, by definition, we have $T(1) = 1$.

In general, given 2 n-bit LFSRs L_1 and L_2 with the same feedback polynomial as LFSR$_{ref}$ and the seeds as X and Y, respectively, if $T(X) > T(Y)$, then the output sequence of L_2 is a delayed version of that of L_1 by $(T(X) - T(Y))$ clock cycles. Fig. 6 shows an example with $T(X) - T(Y) = 1$. Consequently, we can implement L_2 by inserting $(T(X) - T(Y))$ DFFs after L_1, as shown in Fig. 6. We refer to this operation as *merging* two LFSRs. Note that if $T(X) - T(Y) \leq n$, merging two LFSRs reduces the total hardware cost.[2] However, if $T(X) - T(Y) > n$, the merge needs more DFFs than an LFSR has. Thus, in this case, using two separate LFSRs is more hardware-efficient than merging two LFSRs.

Based on the above observation, we propose a method to optimize the hardware cost of the proposed implementation. It has two steps. The first step is to merge m LFSRs in our proposed implementation based on the above merging criteria. After the first step, there may still exist some redundant DFFs. Then, the second step removes them. Next, we elaborate these two steps in Sections 5.1.1 and 5.1.2, respectively. We conclude with some properties of the proposed hardware cost optimization method in Section 5.1.3.

5.1.1 Merging LFSRs. The procedure to merge the m LFSRs based on the above merging criteria is shown in Algorithm 1. Given the seeds of the m LFSRs as $S_1, \dots, S_m$, it first sorts the sequence $T(S_1), \dots, T(S_m)$ in the descending order and obtains a new sequence $T(S_{r_1}), \dots, T(S_{r_m})$, where $(r_1, \dots, r_m)$ is a permutation of $(1, 2, \dots, m)$ satisfying that $T(S_{r_1}) > T(S_{r_2}) > \dots > T(S_{r_m})$. Then, for each $1 \le i \le m-1$, it tries to merge LFSR$_{r_i}$ and LFSR$_{r_{i+1}}$. If $d = T(S_{r_i}) - T(S_{r_{i+1}}) \le n$, it inserts d DFFs after LFSR$_{r_i}$ to implement LFSR$_{r_{i+1}}$. Otherwise, it introduces a new LFSR with the same feedback polynomial as LFSR$_{r_1}$ to implement LFSR$_{r_{i+1}}$.

Algorithm 1: LFSR merging procedure.

```
1 input: The seeds $S_1, \dots, S_m$ of the $m$ LFSRs;
2 output: LFSR$_{r_1}$, LFSR$_{r_2}$, ..., LFSR$_{r_m}$;
3 $(T(S_{r_1}), \dots, T(S_{r_m})) \leftarrow sort(T(S_1), \dots, T(S_m))$;
4 Use an LFSR to implement LFSR$_{r_1}$;
5 for $i \leftarrow 1$ to $m-1$ do
6     if $d = T(S_{r_i}) - T(S_{r_{i+1}}) \le n$ then
7         Insert $d$ DFFs after LFSR$_{r_i}$ to implement LFSR$_{r_{i+1}}$;
8     else Use a new LFSR to implement LFSR$_{r_{i+1}}$;
9 return LFSR$_{r_1}$, LFSR$_{r_2}$, ..., LFSR$_{r_m}$;
```

The output of Algorithm 1 is an implementation of m LFSRs by multiple sequences of DFFs, each for a subset of the m LFSRs.

Example 1. *Consider merging 3 6-bit LFSRs with the seeds S_1, S_2, and S_3. Suppose that $(T(S_1), T(S_2), T(S_3)) = (10, 30, 28)$. First, we sort $T(S_1), T(S_2), T(S_3)$ in descending order as $T(S_2), T(S_3), T(S_1)$. Then, we use an LFSR to implement LFSR$_2$. Since $T(S_2) - T(S_3) = 2 < 6$ and $T(S_3) - T(S_1) = 10 > 6$, we insert 2 DFFs after LFSR$_2$ to implement LFSR$_3$ and use a separate LFSR with the same feedback polynomial as LFSR$_2$ to implement LFSR$_1$. Therefore, by applying Algorithm 1, the 3 LFSRs can be implemented by 2 DFF sequences as shown in Fig. 7, where the first sequence of 8 DFFs implements LFSR$_2$ and LFSR$_3$ and the second one of 6 DFFs implements LFSR$_1$.*

5.1.2 Removing Redundant DFFs. By the proposed implementation shown in Fig. 5, for any $2 \le i \le m$, not all the outputs of LFSR$_i$ are used. Indeed, only $(n - j_i)$ outputs of LFSR$_i$ are used. This gives us a further opportunity to reduce DFFs. For example, suppose that for LFSR$_2$ and LFSR$_3$ in Fig. 7, each of them has 3 bits used, which are indicated by T_2 and T_3 in Fig. 7, respectively. Then, the rightmost 2 DFFs in the first DFF sequence, which are shown in grey, are not needed: Removing them does not affect the correct generation of T_2 and T_3. Therefore, they are redundant and hence, can be removed.

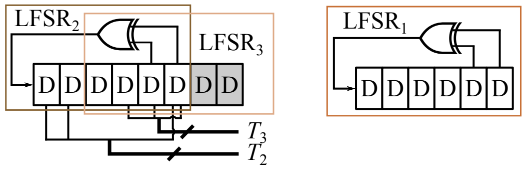


Figure 7: An implementation of three 6-bit LFSRs by two sequences of DFFs after LFSR merging, where T_2 and T_3 are the used bits from LFSR$_2$ and LFSR$_3$, respectively.

Inspired by the above example, we propose a method to remove the redundant DFFs. The method works on each DFF sequence in turn. For each sequence, it starts from the rightmost DFF of the sequence. If the output of the DFF is neither used as a bit in a random number R_i nor as an input to an XOR gate, then it is redundant and removed. With the rightmost DFF removed, we repeat the above step on the *new* rightmost DFF. The entire procedure stops until the current rightmost DFF is either used as a bit in a random number R_i or as an input to an XOR gate.

5.1.3 Properties of the Proposed Hardware Cost Optimization Method. The proposed method has the following properties.

(1) The method does not affect the accuracy. This is because the method keeps the function of each bit in the random numbers $R_1, \dots, R_m$.
(2) The total number of DFFs in the design optimized by the proposed method is no more than $(n + D)$, where $D = \max\{T(S_1), \dots, T(S_m)\} - \min\{T(S_1), \dots, T(S_m)\}$. A sketch of the proof of this claim is as follows. According to the LFSR merging procedure shown in Algorithm 1, LFSR$_{r_1}$ has n DFFs. For any $2 \le i \le m$, the additional number of DFFs needed to implement LFSR$_{r_i}$ is at most $(T(S_{r_{i-1}}) - T(S_{r_i}))$. Thus, the total number of DFFs needed to implement all the m LFSRs after the LFSR merging procedure is at most $n + \sum_{i=2}^{m}(T(S_{r_{i-1}}) - T(S_{r_i})) = n + T(S_{r_1}) - T(S_{r_m})$, where $T(S_{r_1}) = \max\{T(S_1), \dots, T(S_m)\}$ and $T(S_{r_m}) = \min\{T(S_1), \dots, T(S_m)\}$. Thus, the total number of DFFs after the LFSR merging procedure is at most $(n + D)$. The second step, removing redundant DFF, may further reduce some DFFs. Thus, the total number of DFFs in the design optimized by the proposed method is no more than $(n + D)$.

5.2 Accuracy Optimization

In this section, under the design choice from Section 5.1, i.e., all the LFSRs having the same feedback polynomial, we propose a method to optimize the accuracy of the proposed implementation by configuring the available design parameters, including 1) the common feedback polynomial, 2) the seeds of the m LFSRs, and 3) the scrambling ways of the $(m+1)$ scrambling modules. With an optimized configuration, a better uniform spatial distribution is generated and the correlation among the outputs of the RNSs can be further reduced, leading to an improved accuracy [21, 22].

There are $(2^n - 1)$ possible seeds for an n-bit LFSR and $n!$ possible scrambling ways for an n-bit scrambling module. Assume that there are f possible feedback polynomials for an n-bit LFSR. The size of the entire configuration space is $(2^n-1)^m (n!)^{m+1} f \approx 2^{nm} (n!)^{m+1} f$,

[2]When $T(X) - T(Y) = n$, merging reduces the number of XOR gates needed for the two LFSRs.

which is very large even when n and m is small. Therefore, it is impossible to search the entire configuration space to find the best configuration. To efficiently find a good configuration, we propose a heuristic method as shown in Algorithm 2.

Algorithm 2: Accuracy optimization procedure.

```
1  input: an SC core, LFSR bit-width $n$, LFSR number $m$, number of
     feedback polynomials of an $n$-bit LFSR $f$, a parameter $u \geq 1$
     controlling the total number of DFFs, and parameters $t_d$ and $t_b$;
2  output: minimum MAE $MAE_{\min}$ and the best configuration $C^*$;
3  $MAE_{\min} \leftarrow +\infty$, $C^* \leftarrow null$;
4  for $i \leftarrow 1$ to $f$ do
5      Choose the $i$-th possible feedback polynomial for the $m$ LFSRs;
6      Choose LFSR$_1$'s seed $S_1$ so that $T(S_1) = u$;
7      for $j \leftarrow 1$ to $t_d$ do
8          Randomly choose LFSR$_k$'s seed $S_k$ so that $1 \leq S_k \leq 2^n - 1$
             and $1 \leq T(S_k) \leq u$, for $k = 2, \dots, m$;
9          for $k \leftarrow 1$ to $t_b$ do
10             Randomly chooses a scrambling way for SR$_r$, for
                 $r = 1, \dots, m+1$;
11             Simulate the circuit with the current configuration and
                 the specified SC core, and obtain the $MAE$;
12             if $MAE < MAE_{\min}$ then
13                 $MAE_{\min} \leftarrow MAE$;
14                 update $C^*$ to the current configuration;
15 return $MAE_{\min}$ and $C^*$;
```

The proposed method takes as inputs an SC core to which the proposed RNS implementation is applied and several parameters. The main part of the proposed method is a triple loop. The outermost loop iterates over all the feedback polynomials of an n-bit LFSR and sets the seed S_1 of the 1-st LFSR so that $T(S_1) = u$, where $u \geq 1$ is a parameter to control the total number of DFFs. In the intermediate loop, for $2 \leq k \leq m$, it randomly chooses the seed S_k of LFSR$_k$ so that $1 \leq T(S_k) \leq u$. In the innermost loop, it randomly chooses a scrambling way for each of the $(m+1)$ scrambling modules. The number of iterations for the intermediate and the innermost loops are t_d and t_b, respectively, which are two parameters. For each configuration of the feedback polynomial, the seeds, and the scrambling ways generated within the triple loop, it simulates the circuit with that configuration and the specified SC core to obtain its MAE. Finally, the configuration with the minimum MAE is returned. The time complexity of the method is $O(ft_dt_b)$, which is far smaller than $f2^{nm}(n!)^m$ when t_d and t_b are small.

By Algorithm 2, we have $\max\{T(S_1), \dots, T(S_m)\} = T(S_1) = u$ and $\min\{T(S_1), \dots, T(S_m)\} = 1$. Thus, by Property 2 described in Section 5.1.3, the number of total DFFs of the proposed optimized implementation is at most $(n + u - 1)$.

6 EXPERIMENTAL RESULTS

This section shows the experimental results comparing the SNG using our optimized RNS design with some existing SNG designs.

6.1 Experimental Setup

To show whether the proposed RNS design are suitable for different types of SC circuits, we choose 8 SC circuits as benchmarks. The first three are the 2-input, 3-input, and 4-input SC multipliers. We denote them as *MUL*-2, *MUL*-3, and *MUL*-4, respectively. The next three are the circuits implementing $\cos(x)$, $\sin(x)$, and $\tanh(x)$ proposed in [12]. The last two are the circuits implementing $x^{0.45}$ and e^{-2x} synthesized by the method in [23] with degree and precision as 4 and 6, respectively. For these 2 circuits, we apply 4 SNGs to generate 4 stochastic bit streams with the probability x and reuse one RNS in these 4 SNGs to generate 6 stochastic bit streams with the constant coefficient of $\frac{1}{2}$.

We choose 4 existing SNG designs for comparison, which are denoted as *m-SSG*, *m-LFSR*, 1-*LFSR*, and *m-FSM* [7, 14, 20, 21]. *m-SSG*, *m-LFSR*, and 1-*LFSR* consist of RNSs and comparators, as shown in Fig. 2. They differ by the RNSs. Specifically, for an SC circuit with m inputs, *m-SSG* has m Sobol sequence generators, each responsible for the generation of the stochastic bit stream of one input [7]. It usually has the highest accuracy with short stochastic bit streams. *m-LFSR* is similar to *m-SSG* except that each Sobol sequence generator is replaced by an LFSR [21]. It is the most widely used SNG design. 1-*LFSR* has only one LFSR for the entire SC circuit and it is shared among the SNGs for different inputs [14]. It has the lowest hardware cost. *m-FSM* is based on finite state machine (FSM) [20]. For an SC circuit with m inputs, it applies m FSMs, each responsible for the generation of the stochastic bit stream of one input. The states of each FSM are determined by a Sobol sequence. It is the state-of-the-art SNG design with high enough accuracy and low hardware cost. However, it does not include an RNS. Note that for the circuits for $x^{0.45}$ and e^{-2x}, which are synthesized by the method in [23], they need an n-bit RNS to generate the stochastic bit streams with the constant coefficient of $\frac{1}{2}$. Since *m-FSM* is essentially based on Sobol sequences, we use a Sobol sequence generator as the n-bit RNS in the circuits for $x^{0.45}$ and e^{-2x} when considering *m-FSM*-based SNG.

For our proposed implementation, as shown in Section 5.2, we need to determine a parameter u to configure it. We consider a choice of $u = \max((m-2)n-1, 1)$, where the function $\max(X, 1)$ ensures that for any $m \geq 2$, $u \geq 1$, a requirement on u in Algorithm 2. By the discussion at the end of Section 5.2, this choice of u also ensures that the total DFF number of our proposed implementation is no more than $(n+max((m-2)n-1, 1)-1 = max((m-1)n-2, n))$. Note that the number of DFFs in *m-LFSR* is nm. Thus, this choice of u ensures that the hardware cost of the SNGs using our proposed implementation is less than that of *m-LFSR* by the cost of at least one LFSR. In the following, we denote the SNG design using the proposed implementation as *Proposed*.

For the accuracy optimization method, we set both t_d and t_b as 100. Thus, the total number of configurations searched by our method is $10000f$. Note that for *m-SSG* and *m-FSM*, we need to properly choose its Sobol sequence generators or FSMs, and for *m-LFSR* and 1-*LFSR*, we need to properly determine their feedback polynomials, seeds, and scrambling ways. For a fair comparison, we also randomly configure these SNG designs for the same runtime as we use to configure our design and choose the best configuration with the minimum MAE. In the following, we compare the 4 existing SNG designs and *Proposed* for the bit-width $n = 8$.[3] We repeat each experiment for 10 times and obtain the average experimental results.

[3]We also did experiments for bit-width $n = 6$ and 7. The results are similar to $n = 8$ and hence, are omitted due to space limit.

The average runtime for configuring *Proposed* by Algorithm 2 for each benchmark over 10 times is listed in Table 2.

Table 2: The average runtime for configuring *Proposed* for each benchmark.

Benchmark	*MUL*-2	*MUL*-3	*MUL*-4	$\cos(x)$
Runtime (s)	551.36	879.51	1414.71	1246.92
Benchmark	$\sin(x)$	$\tanh(x)$	$x^{0.45}$	e^{-2x}
Runtime (s)	1003.14	1308.48	517.81	498.23

6.2 Accuracy Comparison

We first compare the accuracy of the SNG designs. For the SC multipliers, we obtain the MAE over 1000 random input groups. For the other benchmarks, we obtain the MAE over all possible inputs. The experimental results are listed in Table 3, where *Ave-MUL* and *Ave-All* denote the average MAEs over the SC multipliers and all benchmarks, respectively. To show the comparison more clearly, in the table, we highlight the minimum MAE among all the SNG designs in bold, and underline the cases where *Proposed* are better than *m-LFSR*.

Table 3: Accuracy comparison for different SNG designs.

Design	*m-SSG*	*m-FSM*	*m-LFSR*	1-*LFSR*	*Proposed*
MUL-2	0.00171	0.00158	0.00189	**0.00155**	0.00178
MUL-3	0.00265	**0.00260**	0.00349	0.01004	0.00334
MUL-4	0.00321	**0.00306**	0.00414	0.01837	0.00413
$\cos(x)$	0.00261	0.00262	0.00239	**0.00221**	0.00232
$\sin(x)$	**0.00168**	0.00188	0.00228	0.00243	0.00262
$\tanh(x)$	**0.00231**	0.00268	0.00269	0.00282	0.00291
$x^{0.45}$	**0.00555**	0.00637	0.00647	0.0110	0.00618
e^{-2x}	0.00340	**0.00336**	0.00372	0.00642	0.00389
Ave-MUL	0.00252	**0.00241**	0.00317	0.00998	0.00308
Ave-All	**0.00289**	0.00302	0.00338	0.00685	0.00340

As shown in this table, *Proposed* achieves very close accuracy as *m-SSG* and *m-FSM*, which generally have the highest accuracy. Furthermore, *Proposed* has almost the same accuracy as *m-LFSR* and far higher accuracy than 1-*LFSR* on average. Overall, it achieves a high accuracy for these benchmarks.

6.3 Hardware Cost Comparison

For the hardware cost, we first compare the areas of the SC circuits using different SNG designs. Note that the SNGs generally occupy most area of an SC circuit. Thus, for the area comparison, we focus on the SNG area. For all the SNG designs except *m-FSM*, their areas equal the sum of the areas of the RNSs and the comparators. For *m-FSM*, its area equals the sum of the areas of the FSM-based SNGs and the Sobol sequence generator, where the area of the Sobol sequence generator is only considered for the circuits for $x^{0.45}$ and e^{-2x}. In the following, we obtain the area of SNGs by summing up the area of its components. These components are specified by hardware description language and synthesized by Synopsys Design Compiler [24] using the Nangate 45nm library [25] to obtain their areas. The area of each component is listed in Table 4, where SNG_{FSM} denotes an FSM-based SNG [20] and SSG denotes a Sobol sequence generator [7].

The area comparison for different SNG designs is shown in Figs. 8. Clearly, the area of *Proposed* is far smaller than that of *m-SSG* and *m-FSM*, and is always smaller than that of *m-LFSR*. Compared to *m-SSG*, *m-FSM*, and *m-LFSR*, the area of *Proposed* is reduced by 92%, 88%, and 27%, respectively, on average.

Table 4: Area of each component.

Component	SSG	SNG_{FSM}	LFSR	Comparator	DFF
Area (μm^2)	613.66	380.65	43.76	26.87	4.77

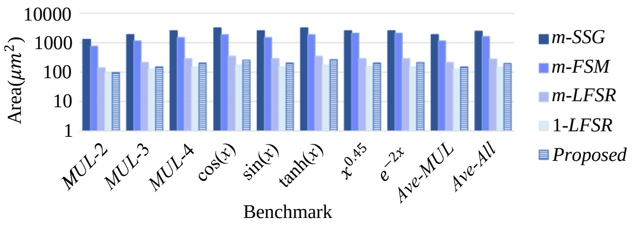


Figure 8: Area comparison for different SNG designs.

Then, we compare the powers and the delays of the SC circuits using different SNG designs. Note that for these circuits, the differences of their powers and delays lie in their SNGs. Therefore, to compare their powers and delays, we only need to compare their SNGs. For each existing SNG design, we obtain its power by summing up those of its components. For *Proposed*, as described in Section 6.1, its hardware cost is less than that of *m-LFSR* by the cost of at least one LFSR. Thus, its cost is no more than the total cost of $(m-1)$ LFSRs and m comparators for an m-input SC circuit. Note that $(m-1)$ LFSRs and $(m-1)$ comparators constitute $(m-1)$ LFSR-based SNGs. Therefore, for *Proposed*, its power is no more than the total power of $(m-1)$ LFSR-based SNGs and 1 comparator for an m-input SC circuit. By Synopsys Design Compiler [24] based on the Nangate 45nm library [25], we obtain the power of a comparator, a Sobol sequence generator-based SNG, an FSM-based SNG, and an LFSR-based SNG as $1.10\mu W$, $30.1\mu W$, $9.06\mu W$, and $4.54\mu W$, respectively. Then, based on the above power model, we obtain the powers of different SNGs for an m-input SC circuit, which are listed in Table 5. For the delay of an SNG design, since it is independent of m, we measure the delay of an SNG design by setting $m = 1$. The delays of different SNGs for an m-input SC circuit are also listed in Table 5. As shown in the table, *Proposed* has a much smaller power and a smaller delay than *m-SSG*. Compared to *m-FSM*, *Proposed* has a smaller power and a 45% delay reduction. Compared to *m-LFSR*, *Proposed* has a smaller power and the same delay.

Table 5: Powers (μW) and delays (ns) of different SNGs for an m-input SC circuit.

Design	*m-SSG*	*m-FSM*	*m-LFSR*	1-*LFSR*	*Proposed*
Power	$30.1m$	$9.06m$	$4.54m$	$3.44 + 1.10m$	$\leq 4.54m - 3.44$
Delay	2.00	3.39	1.88	1.88	1.88

In summary, compared to *m-SSG* and *m-FSM*, *Proposed* can achieve far lower hardware cost and very close accuracy. Furthermore, compared to *m-LFSR*, *Proposed* can achieve lower hardware cost and almost the same accuracy. In conclusion, *Proposed* has both low hardware cost and high accuracy.

6.4 Accuracy-Area Trade-off Comparison

As shown in Section 5.2, our proposed RNS design has an important parameter u. In this section, we study the influence of u on the accuracy and the hardware cost. We choose 5 us, which are $u_1 = 1$, $u_2 = \max(\lfloor \frac{(m-2)n-1}{3} \rfloor, 1)$, $u_3 = \max(\lfloor \frac{(2m-4)n-2}{3} \rfloor, 1)$, $u_4 = \max((m-2)n-1, 1)$, and $u_5 = 100$ in ascending order. We obtain 5 corresponding SNG designs based on the proposed RNS implementation. Then, we apply these SNG designs to the 8 benchmarks and obtain the average MAEs and the average SNG areas. In Fig. 9, we plot how the average SNG area and the average MAE of the proposed implementation changes with u. For comparison, we also add the results of *m-SSG*, *m-FSM*, *m-LFSR*, and 1-*LFSR*.

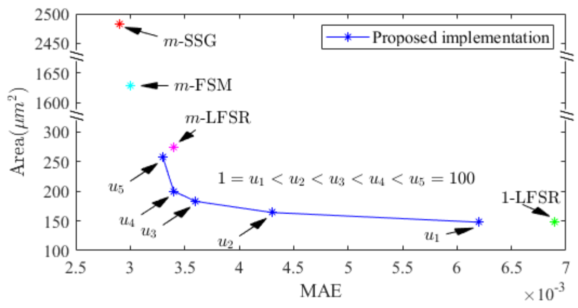


Figure 9: The SNG areas and the MAEs of the proposed implementation for different choices of the parameter u.

As the figure shows, for the proposed implementation, the SNG area increases with u, while the MAE decreases with u. This is expected. With a larger u, on the one hand, the design space is larger and hence, it is possible to find a configuration with a lower MAE. On the other hand, $T(S_1), \ldots T(S_m)$ span a larger range, hence potentially requiring more DFFs to implement all the m RNSs. This indicates that by tuning the parameter u, we can trade the accuracy for area for the proposed implementation.

Compared to other SNG designs, the SNG designs using the proposed implementation have far lower hardware cost than *m-SSG* and *m-FSM*, and the design point with $u = u_5$ can achieve both higher accuracy and lower hardware cost than *m-LFSR*. Furthermore, for the case where $u = 1$, the SNG design using the proposed implementation has higher accuracy than 1-*LFSR*. Note that in this case, it has only one LFSR and has the same hardware cost as 1-*LFSR*. Therefore, the proposed implementation can achieve higher accuracy than 1-*LFSR* with the same hardware cost.

Finally, we remark that this study also demonstrates another advantage of our proposed design, that is, it gives design flexibility. By changing the value of u, we can generate different SNG designs to adapt to different requirements of accuracy and hardware cost.

6.5 Case Study: Gamma Correction

To show the application-level quality, we apply the different SNG designs to a gamma correction circuit, which is just the design for the function $x^{0.45}$ described in Section 6.1. The input image and the output image by the accurate gamma correction circuit are shown in Figs. 10(a) and (b), respectively. The output images by the SC gamma correction circuits based on *m-SSG*, *m-FSM*, *m-LFSR*, 1-*LFSR*, and *Proposed* are shown in Figs. 10(c)-(g), respectively. To more clearly show the quality, we calculate the peak signal-to-noise ratios (PSNRs) and structural similarities (SSIMs) for Figs. 10(c)-(g), which are listed below each figure. We can see that the output image by the SC gamma correction circuit based on *Proposed* has the highest PSNR among all the SNG designs. It also has a high enough SSIM. The PSNR and the SSIM values are above 40dB and 0.99, respectively, which indicates only a slight degradation in the quality compared to the accurate output image. Therefore, *Proposed* has a good application effect.

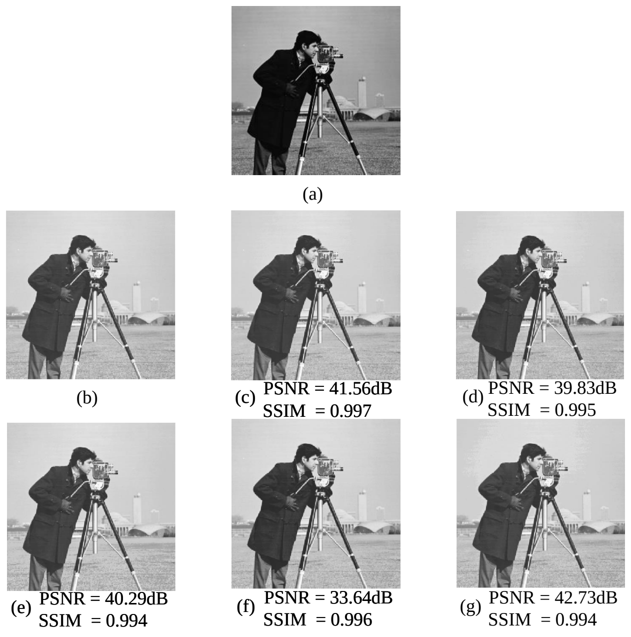


Figure 10: Comparison of various SNGs for the gamma correction application. (a): the input image; (b): the output image by the accurate gamma correction circuit; (c)–(g): the output images by the SC gamma correction circuit based on *m-SSG*, *m-FSM*, *m-LFSR*, 1-*LFSR*, and *Proposed*, respectively.

7 CONCLUSION

In this paper, aiming at designing RNSs with low hardware cost and high accuracy for SC, we propose a basic architecture and an LFSR-based implementation to generate output points uniformly distributed in space. We also propose methods to optimize the hardware cost and accuracy of the proposed implementation. The experimental results show that compared to the existing SNG designs, our proposed one achieves far lower hardware cost with close accuracy or higher accuracy with the same hardware cost. In addition, our method gives design flexibility to adapt to different requirements of accuracy and hardware cost. Our work shows the great potential of the uniform spatial distribution for SC. Our future work will further exploit it to design a more efficient RNS for SC.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China under grant number 2020YFB2205501. The authors thank Mr. Sina Asadi and Prof. M. Hassan Najafi from the University of Louisiana at Lafayette for the help on the experiments.

REFERENCES

[1] B. R. Gaines. Stochastic computing. In *AFIPS Spring Joint Computer Conference*, pages 149–156, 1967.
[2] S. Liu and J. Han. Dynamic stochastic computing for digital signal processing applications. In *DATE*, pages 604–609, 2020.
[3] A. Alaghi, W. Qian, and J. P. Hayes. The promise and challenge of stochastic computing. *IEEE TCAD*, 37(8):1515–1531, 2018.
[4] A. Alaghi et al. Stochastic circuits for real-time image-processing applications. In *DAC*, pages 136:1–136:6, 2013.
[5] Y. Liu, S. Liu, et al. A survey of stochastic computing neural networks for machine learning applications. *IEEE TNNLS*, 32(7):2809–2824, 2021.
[6] A. Alaghi and J.P. Hayes. Fast and accurate computation using stochastic circuits. In *DATE*, pages 1–4, 2014.
[7] S. Liu and J. Han. Energy efficient stochastic computing with sobol sequences. In *DATE*, pages 650–653, 2017.
[8] K. Kim, J. Lee, and K. Choi. An energy-efficient random number generator for stochastic circuits. In *ASP-DAC*, pages 256–261, 2016.
[9] F. Neugebauer, I. Polian, and J. P. Hayes. S-box-based random number generation for stochastic computing. *Microprocessors and Microsystems*, 61:316–326, 2018.
[10] W. Qian et al. An architecture for fault-tolerant computation with stochastic logic. *IEEE TC*, 60(1):93–105, 2011.
[11] Z. Li et al. Simultaneous area and latency optimization for stochastic circuits by D flip-flop insertion. *IEEE TCAD*, 38(7):1251–1264, 2019.
[12] K. Parhi and Y. Liu. Computing arithmetic functions using stochastic logic by series expansion. *IEEE TETC*, 7(1):44–59, 2019.
[13] H. Ichihara et al. Compact and accurate digital filters based on stochastic computing. *IEEE TETC*, 7(1):31–43, 2019.
[14] S. A. Salehi. Low-cost stochastic number generators for stochastic computing. *IEEE TVLSI*, 28(4):992–1001, 2020.
[15] D. Jenson and M. Riedel. A deterministic approach to stochastic computation. In *ICCAD*, pages 1–8, 2016.
[16] H. Sim and J. Lee. A new stochastic computing multiplier with application to deep convolutional neural networks. In *DAC*, pages 29:1–29:6, 2017.
[17] M. H. Najafi and D. J. Lilja. High quality down-sampling for deterministic approaches to stochastic computing. *IEEE Transactions on Emerging Topics in Computing*, 9(1):7–14, 2018.
[18] M. H. Najafi, D. J. Lilja, and M. Riedel. Deterministic methods for stochastic computing using low-discrepancy sequences. In *ICCAD*, pages 1–8, 2018.
[19] Z. Wang et al. Deterministic shuffling networks to implement stochastic circuits in parallel. *IEEE TVLSI*, 28(8):1821–1832, 2020.
[20] S. Asadi, M. H. Najafi, and M. Imani. A low-cost fsm-based bit-stream generator for low-discrepancy stochastic computing. In *DATE*, pages 1–6, 2021.
[21] J. H. Anderson, Y. Hara-Azumi, and S. Yamashita. Effect of LFSR seeding, scrambling and feedback polynomial on stochastic computing accuracy. In *DATE*, pages 1550–1555, 2016.
[22] A. Alaghi and J. P. Hayes. Exploiting correlation in stochastic circuit design. In *ICCD*, pages 39–46, 2013.
[23] X. Peng and W. Qian. Stochastic circuit synthesis by cube assignment. *IEEE TCAD*, 37(12):3109–3122, 2018.
[24] Synopsys Inc. http://www.synopsys.com, 2021.
[25] Nangate Inc. http://www.nangate.com, 2021.