2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)

Reducing the Overhead of Stochastic Number Generators Without Increasing Error

Yudai Sakamoto
Graduate School of Information Science and Engineering
Ritsumeikan University, Shiga, Japan
saicho@ngc.is.ritumei.ac.jp

Shigeru Yamashita
College of Information Science and Engineering
Ritsumeikan University, Shiga, Japan
ger@cs.ritsumei.ac.jp

***Abstract*—Stochastic Computing (SC) is an approximation method to calculate functions with very low hardware cost. When we calculate $f(x)$ by SC, we need to generate many Stochastic Numbers (SNs) whose values are equal to x and constant values. To generate one SN, we need a linear-feedback shift register (LFSR) and a comparator, and thus such overhead may diminish the advantage of SC. Thus, there has been proposed an efficient method to duplicate x (for calculating a power of x) with low hardware overhead. However, it is not known how to reduce the hardware overhead for generating many constant SNs. Therefore, in this paper, we propose an efficient scheme to generate constant SNs with reasonably low hardware overhead without increasing errors. We provide some experimental results by which we can confirm that our proposed scheme is in general very useful to reduce the hardware overhead for generating constant SNs without increasing errors.**

***Index Terms*—Stochastic Computing; Stochastic Number; Correlation**

I. Introduction

Stochastic computing (SC, hereafter) is an approximation calculation method by using *stochastic numbers* (SNs, hereafter) which represent the ratio of 1 in bit strings [4]. There has been a great interest in the research for SC because SC can perform (especially) arithmetic operations with very low hardware cost and power compared to the conventional calculation methods based on binary radix encoding *if we admit some errors* [2]. Indeed, there have been proposed various applications of SC mainly in such as the area of image processing and neural networks (e.g., [3], [6], [7], [11]).

To generate an SN, the most popular way is to use a linear-feedback shift register (LFSR) and a comparator for each SN. Thus, if we need to generate many SNs, the hardware overhead becomes huge. However, when we calculate a function $f(x)$ by SC, we may need many SNs whose values are (1) x, and (2) some constants (coefficients in the expression of $f(x)$). As for (1), note that we need multiple different SNs for x to calculate a power of x although their values are the same (as x). This is because if we calculate x^2 by multiplying exactly the same two SNs for x in an SC manner, we get the value x not x^2, and so we need different SNs for x. More precisely as will be explained later in Section II, we need multiple different SNs for x which are not correlated with each other.

Fortunately, there has been proposed a very efficient method called RRR (Register based Re-arrangement circuit using a Random bit stream) duplicator (RRRD, hereafter) [10] which duplicates SNs to generate many SNs of the same value which are not correlated highly with each other by using few hardware.

JSPS KAKENHI Grant Number 15H02679

However, it is not known how to reduce the hardware overhead for generating many constant SNs (of the different values), which we will focus in this paper. There proposed a method to generate many SNs from very few SNs [8]. Thus, one may consider to use the method in [8] to reduce the hardware overhead for generating many constant SNs. However, that idea may not work well as we explained later in this paper because the method in [8] produce SNs which are highly correlated with each other and thus the calculation error may become huge. (This is because their motivation is not to generate coefficients in an expression of a function, but to generate SNs for another purpose which does not need SNs with low correlation.)

Considering the above-mentioned problem, in this paper, we propose a novel scheme to utilize the method [8] by lowering the correlation. Then we propose three methods in the scheme. We then confirm our proposed scheme is useful by some experiments.

This paper is organized as follows. The following Section II explains the basics for SC including how the correlation between SNs affects the calculation, and how an RRRD works. Section III explains our main idea to reduce the hardware overhead to generate many constant SNs without increasing errors too much. Then we propose our three methods in the same section. After that, Section IV shows some experimental results which confirm that our proposal is indeed useful. Finally, Section V concludes the paper with our future works.

II. Preliminaries and Previous Works

A. Stochastic Computing

In SC, a number is represented by a bit-stream in such a way that the probability (ratio) of "1" in the bit-steam is interpreted as the number itself [2]. For example, a bit-stream "00101000" represents $\frac{1}{4}$ because there are two "1" in the 8 bit-length. We refer to bit-streams of this type as *stochastic numbers* (SNs). We also refer to the probability that a stochastic number represents as its *value*. Thus, two bit-streams that contain "1" with the same probability represent the same number; we say that the values of the two stochastic numbers are the same. For example, both "101100" and "01010110" represents $\frac{1}{2}$, i.e.,

2380-6923/19/$31.00 ©2019 IEEE
DOI 10.1109/VLSID.2019.00028

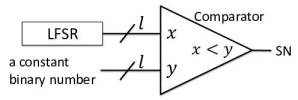

Fig. 1. Stochastic Number Generator

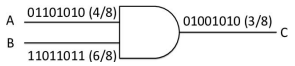

Fig. 2. An AND gate as a stochastic multiplier.

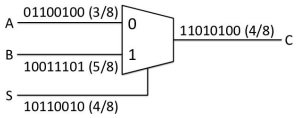

Fig. 3. Multiplexer used as a scaled stochastic adder

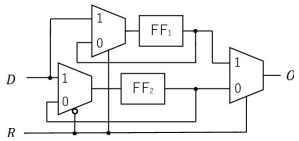

Fig. 4. Register based Re-arrangement circuit using a Random bit stream duplicator

the values of the two stochastic numbers are both $\frac{1}{2}$. We can increase the *precision* of the represented number by making the bit-stream longer.

By using a *stochastic number generator (SNG)* as shown in Fig. 1, we can generate an SN by comparing a constant binary number and a random number generated by *Linear Feedback Shift Register* (LFSR), etc. If we can use an ideal random number, the probability of getting 1 from such an SN can be controlled by the constant binary number. Thus, we can generate an SN representing any number as we want.

An issue we should consider is the obvious fact that an SN can represent only a real value in the range of $[0, 1]$. If we need to perform operations on numbers out of the range, we need to scale the inputs to fit into the range of $[0, 1]$, and then we need to scale again the final results appropriately after the whole SC.

Here, we briefly explain how basic arithmetic operations can be done in SC. Indeed, many operations can be done with very simple logic gates as we will see in the following. In SC, we can perform the multiplication of two SNs by simply inputting the two bit-streams to an AND gate as shown in Fig. 2. In the following, let $P(X = 1)$ mean the probability of getting 1 from the binary string X. If the two SNs, A and B, are independent with each other, we have the following: $P(A \; and \; B = 1) = P(A = 1) \times P(B = 1)$. From this, it is easy to see that a simple AND gate can be used as a *stochastic multiplier*. Indeed, in the example as shown in Fig. 2, we can surely get the correct result: $C = A \times B$ because $P(A = 1) = \frac{4}{8}$ and $P(B = 1) = \frac{6}{8}$, and $P(C = 1) = \frac{3}{8}$.

In SC, as shown in Fig. 3, we can perform the addition of two SNs by using a multiplexer (MUX) and an appropriate *scaling* operation if necessary. If an SN S is independent from both the two SNs, A and B, we have the following: $P(C = 1) = P(S = 1) \times P(A = 1) + P(S = 0) \times P(B = 1)$ where C is the output of the MUX. Especially, when $P(S = 1) = \frac{1}{2}$, $P(C = 1) = \frac{1}{2}(P(A = 1) + P(B = 1))$. Thus, we get an SN C which represents a number for the $\frac{1}{2}$-scaled addition result. Indeed, in the example as shown in Fig. 3, $P(A = 1) = \frac{5}{8}$, $P(B = 1) = \frac{3}{8}$, $P(S = 1) = \frac{4}{8}$ and $P(C = 1) = \frac{4}{8}$. This means we can get the half-scaled addition results. We may need to scale the result to get the real addition result if necessary.

So far we have seen that an AND gate and an MUX are enough to calculate the multiplication and addition of two SNs, respectively in SC. However, it should be noted if the two input SNs and/or the control input of a MUX (S in Fig. 3) are correlated, the calculated result is generally incorrect. Also, the more the SNs are correlated, the more the result of SC calculation is incorrect in most cases. Thus, in the research community, the value called "Stochastic Computing Correlation" (hereafter, SCC) [1] is often considered. The SCC of two SNs expresses how the SNs are correlated with each other. The range of SCC is from -1 to 1, and if the value of SCC is near to 0, the two SNs are less correlated.

Let us consider an extreme case when the two input SNs to the SC multiplication are exactly the same, i.e., highly correlated. Indeed, if the two SNs are exactly the same, the SCC of the two SNs is 1. In such a case, because the AND of the two same SNs is the same as its inputs (i.e., the AND of A and A is A), we cannot get the correct multiplication values A^2 although we want to multiply A and A by an AND gate. In general, we cannot get the correct multiplication result by an AND gate when the inputs are correlated. Especially when the value of input SNs to an AND gate is near to $\frac{1}{2}$, the absolute value of the error becomes large if there is a correlation of the two inputs. Therefore, we usually generate each SNs independently by a different independent SNG as shown in Fig. 1.

B. SN duplicators [10]

An SN duplicator refers to a generator that outputs an SN with the same value as the input SN, but has a different bit stream of the input SN. Therefore, an SN duplicator must satisfy the following condition.

- The values of input SN D and output SN O must be equal and the bit streams of them differ ($D \neq O$).

In [9], an SN duplicator using a single 1-bit flip-flop (FF) has been proposed. However, in case of 1-bit FF-based duplicators, we obtain an output SN that is exactly the same bit stream when we input the same SN to them; this duplicator can generate only one different SN from one SN. Therefore, this duplicator cannot generate more than one different SN. Indeed, an ideal SN duplicator requires the following conditions.

- Even if the same input SN D is given, the bit stream of its output SN O differs from each other every time the duplicator duplicates its input SN.

As an ideal SN duplicator, Register based Re-arrangement circuit using a Random bit stream duplicator (RRRD) has been proposed in [10]. An RRRD is shown in Fig. 4. An RRRD consists of three MUXs and two 1-bit FFs (FF_1 and FF_2). In Fig. 4, the bit stored in FF_2 is used as its i-th output O_i, the i-th input bit D_i is newly stored into FF_2, and the bit stored in FF_1 is not changed when $R_i = 0$ (R_i is the i-th bit in R). In the same way, the bit stored in FF_1 is used as O_i, D_i is newly stored into FF_1, and the bit stored in FF_2 is not changed when $R_i = 1$. In case of RRR, even if we input the same SN D to an RRRD, we obtain a different output SN O every time because we use different R every time.

Also, all the bits in D are used as O except for the last bit stored in FF_1 and FF_2. Instead, the initial bits stored in the FFs are outputted at first. This means that the erroneous bit at duplication using an RRRD is no more than two bits. Therefore, maximum duplication error of an RRRD is $\frac{2}{|d|}$ which is in inverse proportion to the bit length of the input SN. In [10], RRRDs are used to duplicate input x when calculating $f(x)$.

III. Reducing the Overhead to Generate Constant SNs

Any function can be approximated by a polynomial based on Maclaurin expansion. Then, the value of a polynomial can be calculated in an SC way because we need only multiplications (with AND gates) and additions (with MUXs with appropriate scaling if necessary) to calculate the value of a polynomial. Moreover, for many useful arithmetic functions, such as $\sin x$ and $\cos x$, we can calculate the function by only AND and NOT (or NAND) gates by transforming the functions by Horner's method [9].

For example, $\sin x$ can be expressed as:

$$\begin{aligned}\sin(x) &\simeq x - \tfrac{x^3}{3!} + \tfrac{x^5}{5!} - \tfrac{x^7}{7!} + \tfrac{x^9}{9!} - \tfrac{x^{11}}{11!} + \tfrac{x^{13}}{13!} - \tfrac{x^{15}}{15!} + \tfrac{x^{17}}{17!} \\ &= x(1 - \tfrac{1}{6}x^2(1 - \tfrac{1}{20}x^2(1 - \tfrac{1}{42}x^2(1 - \tfrac{1}{72}x^2(1 - \tfrac{1}{110}x^2 \\ &\quad (1 - \tfrac{1}{156}x^2(1 - \tfrac{1}{210}x^2(1 - \tfrac{1}{272}x^2)))))))). \end{aligned} \quad (1)$$

The above formula can be calculated by SC very easily because a single multiplication can be done by one AND gate, and $(1 - F)$ can be calculated from F by one NOT gate.

However, we need a lot of SNs (random bit strings); for the above $\sin x$, we need to generate eight different SNs for the eight constant values, and we also need to duplicate x to get x^2 and then duplicate x^2 seven times to generate eight different SNs for x^2. Note that we need eight different SNs for x^2 although we need the same values for them to calculate the function accurately. (As we explained in the previous section, the AND of the same two SNs (x^2) gives us x^2, not x^4. Thus we need different SNs for x^2 to calculate the above formula.) Accordingly, we need a substantial amount of hardware resource to generate all the necessary SNs which may diminish the advantage of SC. To reduce the hardware overhead to produce SNs for multiple x's, it is shown that RRRDs [10] work very well as we explained in the previous section.

However, RRRDs cannot be applied for the generation of different constant values because RRRDs just duplicate SNs and thus it cannot produce SNs of different values. Thus, we still need to generate a different SN for each constant value in the above formula. In the following, we propose an efficient scheme to decrease the overhead of generating SNs for constant values without increasing the error of the calculation too much.

A. Generating Many Constant SNs from Few SNs

In our proposed method, we will utilize a method proposed in [8] to generate many constant SNs from small number of SNs. The idea in the method [8] is as follows: suppose we have m SNs, $r_1, r_2, \cdots, r_m$, whose values are $R_1, R_2, \cdots, R_m$. By using an AND gate whose inputs are r_i or the negation of r_i, we can generate an SN whose value is the multiplication of R_i or $(1 - R_i)$. (Note that the negation of r_i is an SN whose value is $(1 - R_i)$.)

For example, $r_1 \cdot \overline{r_2} \cdot r_3$ generates an SN whose value is $R_1 \cdot (1 - R_2) \cdot R_3$. By ORing the outputs of such AND gates, we can generate many SNs. Thus, intuitively, we can generate exponentially many SNs from few SNs by adding logic circuits. Indeed the paper [8] proves that if we want to produce multiple SNs of the form $\frac{M}{1024}(1 \leq M \leq 1023)$ at the same time, we need at most four input SNs. For brevity, we call the above-mentioned method **GMCS (Generating Many Constant SNs from Few SNs)** in the following. Specifically, when we consider to generate many SNs of the form $\frac{M}{1024}(1 \leq M \leq 1023)$, GMCS can tell us how to generate such SNs (i.e., the logic circuits generating SNs) from four SNs whose values are $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$.

It seems that we can reduce the overhead for the constant SNs for general SC by GMCS. However it is not true by the following reason. Suppose we want to calculate a function in an SC way, and so we need many constant SNs whose values are coefficients (i.e., a_i in the expression of the above-mentioned $\sin x$) used in the expression of the function. If we use SNs produced by GMCS for each a_i (coefficients), we do not expect to get the correct function's value because there is a correlation between each a_i generated by GMCS. For example, when a_1 and a_2 are generated as $r_1 \cdot \overline{r_2} \cdot r_3$ and $r_1 \cdot r_2 \cdot \overline{r_3}$, respectively, there is a strong correlation between a_1 and a_2 because they use the same r_1, r_2 and r_3. Thus, we cannot utilize GMCS simply to reduce the number of constant SNs.

Note that GMCS is proposed in [8] to generate the constant SNs which are used for constant values in *Binary Combination Polynomial (BCP)*-based SC [12]; the constant SNs are only used as inputs of multiplexers in such a usage, and thus they are allowed to be correlated with each other [5]. In contrast, in this paper, we use constant SNs which are multiplied with each other. Thus we cannot simply use the idea of GMCS for our purpose.

B. Reducing the Correlation between SNs by RRRDs

Our main idea in this paper is to apply RRRDs to the inputs of GMCS so that we can decrease the correlation between SNs generated by GMCS. We explain our idea here.

Suppose we want to generate eight constant SNs, $a_1, \cdots, a_8$ to calculate some function in an SC manner. By using GMCS, we can do so as follows: first we prepare four SNs denoted by $r_1, \cdots, r_4$. (Specifically, the values of r_1, r_2, r_3, r_4 are set to $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ and $\frac{1}{16}$, respectively, in the paper [8].) Then, GMCS can produce logic circuits that generates desired constant SNs $(a_1, \cdots, a_m)$ at the same time from four SNs $(r_1, \cdots, r_4)$ where $a_i = \frac{M}{1024}(1 \leq M \leq 1023)$. Suppose that we can generate a_1 as $r_1 \cdot \overline{r_2} \cdot r_3$, and a_2 as $r_1 \cdot r_2 \cdot r_3$. In this case, as mentioned above, the correlation between a_1 and a_2 is high because they use the same r_1, r_2 and r_3. Thus we duplicate

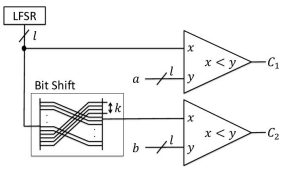

Fig. 5. Sharing an LSFR by Circular Shift

TABLE I
AVERAGE SCC ($l = 8$)
FOR DIFFERENT SHIFT AMOUNT

Shift Amount	$SCC(C_1, C_2)$
$k = 0$	1.000
$k = 1$	0.684
$k = 2$	0.451
$k = 3$	0.317
$k = 4$	0.274
$k = 5$	0.317
$k = 6$	0.451
$k = 7$	0.684

r_i by RRRDs to generate different SNs whose values are the same as r_i. Let us denote such duplicated SNs by $r_i^0 (= r_i)$, r_i^1, r_i^2, $\cdots$, r_i^m when we duplicate r_1 m times. Then we generate a_1 as $r_1^0 \cdot \overline{r_2^1} \cdot r_3^2$, and a_2 as $r_1^2 \cdot r_2^0 \cdot r_3^1$. Then we can expect that the correlation between a_1 and a_2 is reduced.

We performed the following experiment to check the validity of the above-mentioned our idea. In one trial, we compare two methods: (**Without RRRDs**) we generate eight random SNs from r_1, r_2, r_3 and r_4 by applying only GMS simply, and (**With RRRDs**) we generate the same eight random SNs by GMCS whose inputs are duplicated by RRRDs for each a_i as mentioned above. We performed this trial 100 times, and calculated SCC between any pair of a_i and a_j from eight generated SNs. The average SCC was 0.74 and 0.28 for (Without RRRDs) and (With RRRDs), respectively.

In conclusion, we expect RRRDs would be useful to increase the accuracy of SC when we use GMCS to reduce the overhead of generating many constant SNs.

C. Sharing LFSRs

In our proposed scheme, we may want to consider to decrease the number of LFSRs; we try to share LFSRs among many SNs. The idea taken from [5] is as follows: when we generate two (or more) SNs, we share an LFSR among the generation of multiple SNs. Of course, if we share an LFSR simply, the generated SNs are correlated highly with each other. To reduce the correlation, we perform a circular shift (of different shift amount) on each input of a comparator to generate an SN as Fig. 5 where we generate two SNs, C_1 and C_2, from one LFSR.

We checked the relation between SCC and the shift amount in the above scheme when the bit length of the LFSR (l) is 8. Table I shows the result; it shows the average value of $SCC(C_1, C_2)$ among all the combination of (a, b) (i.e., $(1,1), (1,2), (1,3), \cdots, (255, 254), (255, 255)$, which are $255^2 = 65,025$ combinations in total) for different shift amount (k).

From Table I, we can observe that the correlation between C_1 and C_2 can indeed be reduced by the circular shift, and the correlation becomes the lowest when the shift amount, k, equals $\frac{l}{2}$ where l is the bit-length of the LFSR. Thus, in our proposal in the following, we set the shift amount as evenly as possible among the inputs of different comparators which share an LFSR.

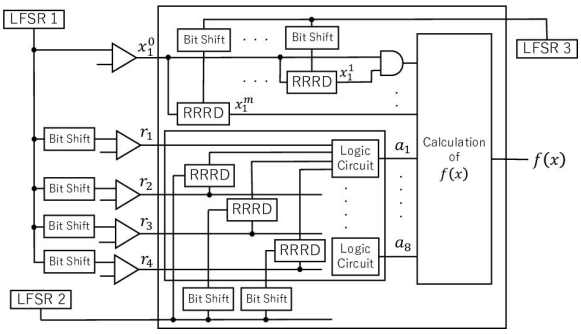

Fig. 6. Our Proposed Scheme to Perform Stochastic Computation

D. Proposed Methods to Reduce the Overhead for Constant SNs

Now we are ready to explain our proposal in this paper to reduce the overhead of generating constant SNs. Considering the above discussions, we propose the following scheme as shown in Fig. 6. The figure shows how we can generate duplicated many SNs for x, and different constant SNs, $a_1 \cdots a_m$, which are used to calculate $\sin x$ in an SC manner.

The figure shows a case in which we use eight constant SNs, $a_1, \cdots, a_8$, with the accuracy level $\frac{1}{1024}$, and thus we need four SNs, $r_1, \cdots, r_4$, for GMCS to produce the eight constant SNs. In the figure, LFSR 1 is shared to generate SNs that are used as x, and $r_1, \cdots, r_4$. Then, as mentioned above, we use RRRDs to duplicate r_i to generate $r_i^0 (= r_i)$, r_i^1, r_i^2, $\cdots$, r_i^7. We generate a_i as a logic circuit whose inputs are $r_1^{i_1}$, $r_2^{i_2}$, $r_3^{i_3}$, $r_4^{i_4}$. (Each logic circuit realizing a_i has four input SNs whose values are the same as $r_1, \cdots, r_4$, but the correlation between the generated $a_1, \cdots, a_8$ should become lower thanks to RRRDs as mentioned above.) In the proposed scheme, we need one SN (a random bit string) for each RRRDs; LFSR 3 are shared for generating all SNs used for RRRDs that duplicate x, and LFSR 2 are shared for generating all SNs used for RRRDs that duplicate r_i to generate $r_i^0 (= r_i)$, r_i^1, r_i^2, $\cdots$, r_i^7.

Accordingly, for the implementation of our proposal in Fig. 6, we need three LFSRs and five comparators and some overhead for RRRDs and Bit Shifters. The overhead for LFSRs and comparators should be dominant among them, so we consider mainly LFSRs and comparators for the overhead in the following.

Next, we seek a possibility that we may further reduce the overhead; we consider to share the above three LFSRs. As a preliminary experiment, we checked how SCC values (between any pair of a_i and a_j) change if we share LFSR 1 and LFSR 2 in the above scheme. We calculated SCC values (between any pair of a_i and a_j) for generating randomly selected $a_1, \cdots, a_8$, and the average SCC values (between any combination of pair of a_i and a_j) over 100 trials were 0.301 and 0.295 for with and without sharing LFSR 1 and LFSR 2, respectively.

From the above preliminary experiments, we conclude that there may be a small difference by sharing LFSRs in our proposal depending on the situation. Therefore, we propose the following three methods: the difference between the three

methods is which LFSRs are shared, and we will compare them by our experiment in the next section.

Method 1: In our proposed scheme, this method separately uses LFSR 1, LFSR 2 and LFSR 3, that is, this method needs three LFSRs.

Method 2: In our proposed scheme, this method shares one LFSR for all RRRDs, i.e., we share LFSR 2 and LFSR 3, that is, this method needs two LFSR.

Method 3: In our proposed scheme, this method shares all of LFSR 1, LFSR 2 and LFSR 3, that is, this method needs one LFSR.

IV. Experimental Results

In this paper, we propose to use GMCS and RRR to decrease the overhead of generating constant SNs without increasing errors. Then we proposed specifically Method 1 to Method 3 in the previous section. To confirm the efficiency of our proposal, we considered other methods to be compared with our methods. More concretely, we tried the following Method 4 to Method 8 in addition to our proposed three methods in our experiments.

In our experiments, we compared the errors when we calculate $\sin x$, $\cos x$, $\log(x+1)$ by our three methods and the following five other methods. These three functions are approximated as a series product of $(1-a_ix^2)$ or $(1-a_ix)$ by Horner's method [9] as mentioned in Section III. Thus, to calculate each function, we need eight constant SNs.

Method 4: This method uses GMCS to reduce the necessary constant eight SNs (i.e., $a_1, \cdots, a_8$) into four constant SNs (i.e., $r_1, \cdots, r_4$) as our proposal as shown in Fig. 6. However, this method does not use RRRDs as our proposal, thus this does not use LFSR 2 in Fig. 6. This method uses two LFSRs in total.

Method 5: This method is almost similar to Method 4, but one LFSR is shared for the two LFSRs used in Method 4. That is, this method uses only one LFSR.

Method 6: This method does not use GMCS, so it needs to generate eight SNs (i.e., $a_1, \cdots, a_8$), for which we require eight comparators. Because this method does not use GMCS, it does not use RRRDs for generating the constant eight SNs $(a_1, \cdots, a_8)$, and thus this does not use LFSR 2 in Fig. 6. This method uses two LFSRs; one is shared for generating x and eight SNs (i.e., $a_1, \cdots, a_8$) as LFSR 1 in Fig. 6. The other LFSR is used for RRRDs which duplicate x or x^2.

Method 7: This method is almost similar to Method 6, but one LFSR is shared for the two LFSRs used in Method 6. That is, this method uses only one LFSR.

Method 8: In the above Method 4 to Method 7, one LFSR (LFSR 1 in Fig. 6) is shared to generate x and the eight constant SNs (i.e., $a_1, \cdots, a_8$). In contrast, Method 8 prepares one dedicated LFSR separately for generating each a_i. (Thus, we do not need to use GMCS.) In this method, we also have other two LFSRs; one is for generating x, and the other is used for RRRDs which duplicate x or x^2. Thus, this method uses 10 LFSRs in total.

TABLE II
The Hardware Overhead of Each Method

Method	♯ LFSRs	♯ comparators	♯ RRRDs
1	3	5	36
2	2	5	36
3	1	5	36
4	2	5	8
5	1	5	8
6	2	9	8
7	1	9	8
8	10	9	8

Note that in all the eight methods, we use one LFSR to be shared for generating the inputs, i.e., x and constant SNs which are $(a_1, \cdots a_8)$ or $(r_1, \cdots, r_4)$ when we use GMCS to generate $a_1, \cdots a_8$ from $r_1, \cdots, r_4$. Note also that Method 8 should be the best in terms of accuracy of the function output, but it needs huge hardware resource as we see in the following.

In our experiment, we set the accuracy level as $\frac{1}{1024}$. Thus, each LFSR has 10 bits in our scheme, and so there are $2^{10} = 1024$ different initial values for one LFSR. For a method having only one LFSR, we tried all 1024 cases. However, for a method having more than one LFSRs, we cannot do all the cases; we randomly select 1024 cases for initial values of all the LFSRs in the method. There are 1024 values for x, i.e., $\frac{1}{1024}, \frac{2}{1024}, \cdots, \frac{1024}{1024}$. It is very time consuming to try all 1024 values for x, so for each case, we tried x from $\frac{2}{1024}$ with increasing $\frac{50}{1024}$, i.e., we tried $x = \frac{2}{1024}, \frac{52}{1024}, \frac{102}{1024}, \cdots, \frac{1002}{1024}$.

Other than the above simple functions, we also tried 100 randomly generated formulas of a series product of $(1-a_ix^2)$ based on Maclaurin expansion of $\sin x$. (In other words, we randomly change each constant a_i in "(1)".) For the randomly generated functions, we did two different sets of experiments; (Random A) each constant a_i in "(1)" is selected from $\frac{1}{1024}$ to $\frac{1023}{1024}$, and (Random B) each constant a_i in "(1)" is selected from $\frac{412}{1024}$ to $\frac{612}{1024}$. The reason we did (B) is that the effect of the correlation of two SNs for a stochastic multiplication of the two SNs would be very large if one of the two SNs is near to $\frac{1}{2}$.

A. Comparison of Hardware Overhead

First we compare the hardware overhead of the above-mentioned eight methods; Table II shows the numbers of LFSRs, comparators, and RRRDs for each method.

It is obvious that an LFSR (10-bit shift register) and an 10-bit comparator are large compared to an RRRD (which needs two flip-flops). Thus, we considered that the above eight methods can be categorized by the hardware overhead as follow:

- Very large overhead: Method 8
- Large overhead: Method 6 and Method 7
- Medium overhead: Method 1
- Small overhead: Method 2 and Method 3
- Very small overhead: Method 4 and Method 5

B. Comparison of Errors

Table III reports the root mean squared error between the accurate result and the actual result by each method. Each value in the table is the ratio to Method 1 (That is, lower

TABLE III
AVERAGE ERROR RATIO

Method	Random A	Random B	$\sin x$	$\cos x$	$\log(x+1)$
1	1.000	1.000	1.000	1.000	1.000
2	0.996	1.031	1.013	0.924	1.194
3	1.286	2.666	1.630	3.328	1.143
4	1.265	7.049	0.674	0.681	1.330
5	1.527	7.253	1.228	1.947	1.375
6	1.137	4.214	3.077	1.966	1.342
7	1.590	6.233	4.086	4.447	1.354
8	0.467	0.783	1.050	0.981	0.332

values are better in each column.) The second and the third columns show the results for randomly generated formulas. The second column is for the case where the constant (a_i) is chosen uniformly at random (Random A). The third is for the case where the constant (a_i) is chosen at random only from $\frac{412}{1024}$ to $\frac{612}{1024}$ (Random B). The forth, the fifth and the six-th columns are for $\sin x$, $\cos x$ and $\log(x+1)$, respectively.

From Table III, we can observe the following:

- Method 8 would be the best in terms of the accuracy (the exception is the case of $\sin x$; it is slightly worse than Method 1).
- Method 1 and Method 2 are not bad for all the cases.
- Method 3 to Method 7 would not be good in the sense that they become very bad in some (or almost all) cases.

As we mentioned in Section II-A, the correlation of the two inputs to a multiplication would have a huge bad impact on the result of the multiplication especially when the values of inputs are near to $\frac{1}{2}$. Indeed, Method 3 to Method 7 suffer from the correlation problem for Random B. However, it should be noted that our proposed Method 1 and Method 2 can overcome the correlation problem even for Random B, i.e., Method 1 and Method 2 works very well for Random B.

C. Discussion

From the experimental result, Method 1 and Method 2 are almost similar, and Method 3 is worse in terms of accuracy among the three of our proposed methods. Considering the hardware cost, we came to a conclusion that Method 2 gives us the best trade-off between the accuracy and the hardware overhead.

Let us compare our proposed methods with other methods, i.e., Method 4 to Method 8, in the following. As we expected, Method 8 seems to be the best in terms of the accuracy, but it requires large hardware overhead. As for Method 6 and Method 7, our methods would be better than them in terms of the both metrics, i.e., accuracy and the hardware cost. Method 4 and Method 5 would have smaller hardware overhead than our methods. Also Method 4 works well for $\sin x$ and $\cos x$ in terms of accuracy. However, we consider that Method 4 and Method 5 would become worse (in terms of accuracy) than our methods for general cases, especially when the constant values are near to $\frac{1}{2}$ by judging from the column "Random B."

In conclusion, we consider that Method 2 would be the best choice among the above eight methods in general because it works well for most cases in terms of accuracy, and the hardware overhead is reasonably small. Of course, we do not claim that Method 2 is always the best; other methods should be better than Method 2 for some specific cases.

V. CONCLUSION AND FUTURE WORK

This paper has investigated various methods to generate constant SNs considering the hardware overhead and errors together. The main idea utilized in our proposed method is to use GMCS to reduce the number of SNs to be generated and to use RRRDs to reduce the correlation caused by GMCS. Our experimental results suggest that Method 2 would provide us a very good trade-off between the hardware overhead and the increasing errors. From this we can conclude that in general our proposed scheme to generate constant SNs works well considering the hardware overhead and the increasing errors. For future work, we may try to perform experiments on other functions, and collect more results.

In this paper, we do not consider the initial values for LFSRs. However, it is known that such values also may affect the calculation errors, thus we may seek the effect of initial values of LFSRs in the future.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 15H02679.

REFERENCES

[1] A. Alaghi and J. P. Hayes. Exploiting correlation in stochastic circuit design. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, volume 00, pages 39–46, Oct. 2013.

[2] Armin Alaghi and John P. Hayes. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):92:1–92:19, May 2013.

[3] Armin Alaghi, Cheng Li, and John P. Hayes. Stochastic circuits for real-time image-processing applications. In *Proceedings of the 50th Annual Design Automation Conference*, pages 136:1–136:6, 2013.

[4] BR Gaines. Stochastic computing systems. *Advances in information systems science*, pages 37–172, 1969.

[5] Hideyuki Ichihara. Shota Ishii. Daiki Sunamori. Tsuyoshi Iwagaki. Tomoo Inoue. Compact and accurate stochastic circuits with shared random number sources. *IEEE 32nd International Conference on Computer Design (ICCD)*, pages 361–366, 2014.

[6] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 124:1–124:6, 2016.

[7] Peng Li and David J. Lilja. Using stochastic computing to implement digital image processing algorithms. In *Proceedings of the 2011 IEEE 29th International Conference on Computer Design*, ICCD '11, pages 154–161, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Ritsuko Muguruma and Shigeru Yamashita. Stochastic number generation with the minimum inputs. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E100.A(8):1661–1671, 2017.

[9] K. Parhi and Y. Liu. Computing arithmetic functions using stochastic logic by series expansion. *IEEE Transactions on Emerging Topics in Computing*, page 1, 2016.

[10] Ryota Ishikawa, Masashi Tawada, Masao Yanagisawa, Nozomu Togawa. Stochastic number duplicators based on bit re-arrangement using randomized bit streams. *IEICE Transactions on Fundamentals of Electronics,Communications and Computer Sciences*, E101-A(7), 2018, To Appear.

[11] Zhiheng Wang, Naman Saraf, Kia Bazargan, and Arnd Scheel. Randomness meets feedback: Stochastic implementation of logistic map dynamical system. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 132:1–132:7, New York, NY, USA, 2015. ACM.

[12] Zheng Zhao and Weikang Qian. A general design of stochastic circuit and its synthesis. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1467–1472, 2015.