Performing Stochastic Computation Deterministically

M. Hassan Najafi, *Member, IEEE,* Devon Jenson, *Student Member, IEEE,*
David Lilja, *Fellow, IEEE* and Marc Riedel, *Senior Member, IEEE,*

***Abstract*—Stochastic logic performs computation on data represented by random bit-streams. The representation allows complex arithmetic to be performed with very simple logic, but it suffers from high latency and poor precision. Furthermore, the results are always somewhat inaccurate due to random fluctuations. In this paper, we show that randomness is *not* a requirement for this computational paradigm. If properly structured, the same arithmetical constructs can operate on *deterministic* bit-streams, with the data represented uniformly by the fraction of 1's versus 0's. This paper presents three approaches for the computation: relatively prime stream lengths, rotation, and clock division. Unlike stochastic methods, all three of our deterministic methods produce completely accurate results. The cost of generating the deterministic streams is a small fraction of the cost of generating streams from random/pseudorandom sources. Most importantly, the latency is reduced by a factor of $\frac{1}{2^n}$, where n is the equivalent number of bits of precision.**

When computing in unary, the bit stream length increases with each level of logic. This is an inevitable consequence of the representation, but it can result in unmanageable bit streams lengths. We discuss two methods for maintaining constant bit-streams lengths via approximations, based on low-discrepancy sequences. These methods provide the best accuracy and area×delay product. They are fast-converging and so offer progressive precision.

***Index Terms*—Stochastic computing, deterministic computing, fast-converging process, unary bit-streams, pseudo-randomized bit-stream, low-discrepancy bit-streams.**

I. Introduction

In the paradigm of stochastic computation (SC), digital logic is used to perform computation on random bit-streams, where numbers are represented by the probability of observing a 1 versus a 0 [1], [2], [3], [4], [5]. The benefit of such a stochastic representation is that complex operations can be performed with very simple logic. For instance, multiplication can be performed with a single AND gate and scaled addition can be performed with a single multiplexer unit. One obvious drawback is that the computation has very high latency, due to the length of the bit-streams. Another is that the computation suffers from errors due to random fluctuations and correlations between the streams. These effects worsen as the circuit depth and the number of inputs increase [5]. A certain degree of accuracy can be maintained by re-randomizing bit-streams, but this is an additional expense [6]. While the logic to perform the computation is simple, generating random or pseudorandom bit-streams is costly. Indeed, in prior work, pseudorandom constructs such as linear feedback shift registers (LFSRs) accounted for as much as 90% of the area of stochastic circuit designs [3], [4]. This significantly diminishes the area benefits.

M. Hassan Najafi is with the School of Computing and Informatics, University of Louisiana at Lafayette, LA, E-mail: najafi@louisiana.edu. Devon Jenson, David Lilja, and Marc Riedel are with the ECE Department, University of Minnesota, Twin Cities, MN Email: {jens1172, lilja, mriedel}@umn.edu

This paper suggests that randomness is *not* a requirement for the paradigm. We show that the same computation can be performed on deterministically generated bit-streams.[1] The results are completely accurate, with no random fluctuations. Without the requirement of randomness, bit-streams can be generated inexpensively. Most importantly, with our approach, the latency is reduced by a factor of approximately $\frac{1}{2^n}$, where n is the equivalent number of bits of precision. (For example, for the equivalent of 10 bits of precision, the bit-stream length is reduced from 2^{20} to only 2^{10}.) As with SC, all bits in our deterministic streams are weighted equally. Accordingly, our circuits display the same high degree of tolerance to soft errors.

Related prior work includes Gupta and Kumaresan [8], who used LFSRs to generate and multiply stochastic numbers that are, in certain cases, guaranteed to be exact. They use an LFSR-based weighted binary stream generator that converts the input data to multiple bit-streams with different weights (e.g., 1/2, 1/4, 1/8, etc.) and non overlapping 1's. The deterministic methods we propose here are more general; we can use a variety of techniques for generating numbers (e.g., counters, LFSRs, or low-discrepancy sequence generators). Our methods converge faster and are more hardware efficient.

We propose three approaches: using relatively prime lengths; performing rotation; and clock dividing streams. We design low-cost counter-based structures for these approaches to generate and process "unary" bit-streams deterministically. When computing in unary, the bit-stream length increases with each level of logic: given initial streams of length n, the length increases to n^2 with the first level, n^3 with the second level, and n^{l+1} with l levels. This is a mathematical consequence: for an operation such as multiplication, the precision of the output values is always more than the precision of input data. This often results in unmanageable bit-streams lengths.

We discuss two methods for maintaining constant bit-stream lengths, based on low-discrepancy (LD) sequences [9]. We show that computation on LD bit-streams can be completely accurate. We introduce a LD-based deterministic method that converges quickly and produces completely accurate results. We then integrate one of the proposed deterministic methods, rotation, with the LD bit-streams. We show that the LD-based deterministic methods converge significantly faster to the expected output. We evaluate the scalability of the proposed methods as the precision and number of inputs increase.

[1]We note that *pseudorandom* bit-streams are often used in stochastic computing. Such bit-streams are, strictly speaking, deterministic. Here when we say *deterministic* we mean bit-streams that have simple patterns and lack any random attributes [7]. An example of a deterministic bit-stream is a "unary" bit-stream: one with first all 1's followed by all 0's.

This paper is structured as follows: Section II presents background information on SC. Section III gives an intuitive view of why SC works. Section IV shows how computation can be performed on deterministic bit-streams in a manner analogous to computation on stochastic bit-streams. Section V presents three deterministic methods and describes their circuit implementations. Section VI discusses two proposed LD deterministic methods. Section VII compares the performance, hardware cost, and area-delay product of prior stochastic methods to the proposed deterministic methods.

II. Background on Stochastic Logic

In a paradigm first advocated by Gaines, logical computation is performed on stochastic bit-streams [1]. There are two possible coding formats: a unipolar format and a bipolar format. These two formats are conceptually similar and can coexist in a single system. In the unipolar coding format, a real number x in the unit interval (i.e., $0 \leq x \leq 1$) corresponds to a bit-stream $X(t)$ of length L, where $t = 1,2,...,L$. The probability that each bit in the stream is one is $P(X = 1) = x$. For example, the value $x = 0.3$ could be represented by a random stream of bits such as 0100010100, where 30% of the bits are one and the remainder are zero. In the bipolar coding format, the range of a real number x is extended to $-1 \leq x \leq 1$. The probability that each bit in the stream is one is $P(X = 1) = \frac{x+1}{2}$. An advantage of the bipolar format is that it can deal with negative numbers directly. However, given the same bit-stream length, L, the precision of the unipolar format is twice that of the bipolar format. For what follows, unless stated otherwise, our examples will use the unipolar format.

The synthesis strategy with stochastic logic is to cast logical computations as arithmetic operations in terms of probabilities. Two simple arithmetic operations – multiplication and scaled addition – are illustrated in Fig. 1.

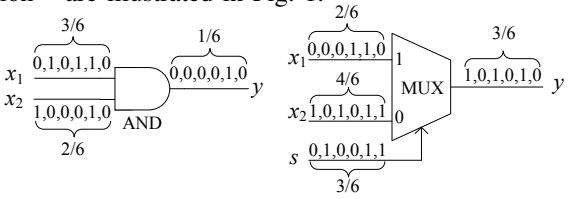


Fig. 1. Stochastic implementation of arithmetic operations: (a) Multiplication; (b) Scaled addition.

- **Multiplication**. Consider a two-input AND gate, shown in Fig. 1(a). Suppose that its inputs are two independent bit-streams X_1 and X_2. Its output is a bit-stream Y, where

$$\begin{aligned} y = P(Y = 1) &= P(X_1 = 1 \text{ and } X_2 = 1) \\ &= P(X_1 = 1)P(X_2 = 1) = x_1 x_2. \end{aligned}$$

  Thus, the AND gate computes the product of the two input probability values.

- **Scaled Addition**. Consider a two-input multiplexer, shown in Fig. 1(b). Suppose that its inputs are two independent stochastic bit-streams X_1 and X_2 and its selecting input is a stochastic bit-stream S. Its output is a bit-stream Y, where

$$\begin{aligned} y = P(Y = 1) &= P(S = 1)P(X_1 = 1) + P(S = 0)P(X_2 = 1) \\ &= sx_1 + (1 - s)x_2. \end{aligned}$$

  Thus, the multiplexer computes the scaled addition of the two input probability values.

III. Intuitive View of Stochastic Computation

Before presenting our methods, we present two intuitive explanations of why stochastic computation works: computing on averages and discrete convolution.

A. Taking a Look at the Average

Stochastic computation is framed as computation on *probabilities*. Computation is happening in a statistical sense on the average number of ones and zeros. Because the probability represented by a bit-stream is equivalent to its expected value, we can instead view bit-streams by the number of ones and zeros we would *expect* to see on average. For example, if we say that bit-stream A represents a probability $p_A = 2/3$, this is equivalent to saying that we expect to see two ones for every three bits. In general, the number formats (unipolar, bipolar, etc.) are all defined in terms of the average number of ones and zeros. For example, the probability p of unipolar and bipolar bit-streams are given by,

$$p_{\text{uni}} = \frac{N_1}{N_1 + N_0} \qquad p_{\text{bi}} = \frac{N_1 - N_0}{N_1 + N_0}, \tag{1}$$

where N_1 and N_0 are the average number of ones and zeros. Using two independent bit-streams in a unipolar format, an AND gate multiplies their probabilities. Labeling the input bit-streams as A and B, the probability of the output bit-stream C is given by,

$$p_C = p_A p_B = \frac{N_{C1}}{N_{C1} + N_{C0}} = \frac{N_{A1}}{N_{A1} + N_{A0}} \frac{N_{B1}}{N_{B1} + N_{B0}} \tag{2}$$

where N_{C1} and N_{C0} represent the average number of ones and zeros in bit-stream C. By multiplying out the right side of Equation 2 and organizing the terms,

$$p_C = \frac{N_{A1}N_{B1}}{N_{A1}N_{B1} + (N_{A1}N_{B0} + N_{A0}N_{B1} + N_{A0}N_{B0})} \tag{3}$$

it can be seen that the fraction has the same form as the unipolar probability of Equation 1. Therefore, the average number of ones and zeros in bit-stream C can be written in terms of the average number of ones and zeros in bit-streams A and B,

$$\begin{aligned} N_{C1} &= N_{A1}N_{B1} \\ N_{C0} &= N_{A1}N_{B0} + N_{A0}N_{B1} + N_{A0}N_{B0} \end{aligned} \tag{4}$$

We define a uniform number as a number in which all bits are weighted equally. Denote the average number of ones and zeros in a bit-stream X as the uniform number $N_{X1}\{1\} + N_{X0}\{0\}$. Distributing the AND operation (denoted by $\wedge$) gives the same result, as Equation 4:

$$\begin{aligned} &N_{C1}\{1\} + N_{C0}\{0\} = \\ &(N_{A1}\{1\} + N_{A0}\{0\}) \wedge (N_{B1}\{1\} + N_{B0}\{0\}) = \\ &N_{A1}N_{B1}\{1\} + (N_{A1}N_{B0} + N_{A0}N_{B1} + N_{A0}N_{B0})\{0\} \end{aligned} \tag{5}$$

This shows that, by representing probabilities with independent random bit-streams, an AND gate operates on average *proportions* of ones and zeros. In general, for any arbitrary logic gate with independent random bit-streams A and B as inputs, the proportion of bits at the output is given by,

$$N_{C1}\{1\}+N_{C0}\{0\} = (N_{A1}\{1\}+N_{A0}\{0\})\square(N_{B1}\{1\}+N_{B0}\{0\}) \quad (6)$$

where the $\square$ symbol is replaced with any Boolean operator. This demonstrates that independent random bit-streams passively maintain the property that the average bits of bit stream A are operated on with the average bits of bit-stream B. Independence guarantees that each outcome of a bit-stream (one or zero) will "see" the average number of ones and zeros of another bit-stream. We conclude this section with two examples demonstrating the application of Equation 6.

Example 1 Assume we have two independent bit-streams A and B with unipolar probabilities $p_A = 1/3$ and $p_B = 2/3$. This means *on average* we will observe a single one for every three bits of A and two ones every three bits of B. If these bit streams are used as inputs to an AND gate, the average output and probability are given by,

$$N_{C1}\{1\}+N_{C0}\{0\} = (1\{1\}+2\{0\})\wedge(2\{1\}+1\{0\}) =$$
$$2\{1\wedge 1\}+1\{1\wedge 0\}+4\{0\wedge 1\}+2\{0\wedge 0\} = 2\{1\}+7\{0\}$$
$$\Rightarrow p_C = 2/(2+7) = 2/9$$

Example 2 Assume we have two independent bit-streams A and B with bipolar probabilities $p_A = 4/6$ and $p_B = -3/5$. This means *on average* we will observe five ones for every six bits of A and a single one for every five bits of B. If these bit-streams are used as inputs to an XNOR gate, the average output and probability are given by,

$$N_{C1}\{1\}+N_{C0}\{0\} = (5\{1\}+1\{0\})\odot(1\{1\}+4\{0\}) =$$
$$5\{1\odot 1\}+20\{1\odot 0\}+1\{0\odot 1\}+4\{0\odot 0\} = 9\{1\}+21\{0\}$$
$$\Rightarrow p_C = (9-21)/30 = -12/30$$

We can see from Examples 1 and 2 that we can find the output of a stochastic logic gate by taking an *average view* of the random bit-streams and applying Equation 6.

B. Insight: Convolution

In basic terms, convolution consists of three operations: slide, multiply, and sum. For bit-streams X and Y, each with L bits, the discrete convolution operation is

$$\sum_{i=1}^{L}\sum_{j=1}^{L} X_i Y_j \quad (7)$$

The previous sections showed an AND gate multiplies proportions if each bit of one bit-stream "sees" every bit of the other bit-stream. Intuitively, this is equivalent to sliding one operand past the other.

Example 3 By sliding the following five-bit operands past each other,

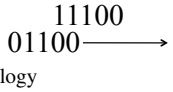


Fig. 2. Sliding operand analogy

each bit of the top operand sees two ones and three zeros and each bit of the bottom operand sees three ones and two zeros. In this way, a stochastic representation maintains the sliding of average bit-streams.

A significant attribute of the stochastic representation is that it is a uniform encoding. Uniform numbers have the interesting property that the order of elements does not matter (i.e., the values are not weighted). This means partial products can be summed by simple concatenation. The following example demonstrates how this contrasts with binary multiplication.

Example 4 To multiply binary numbers, we perform bitwise multiplication and sum the weighted partial products. It takes two operations, bitwise multiply and sum, to go from binary inputs to a binary output. In contrast, to multiply uniform numbers the partial products simply need to be concatenated. By performing bitwise multiplications sequentially in time, concatenation is performed passively.

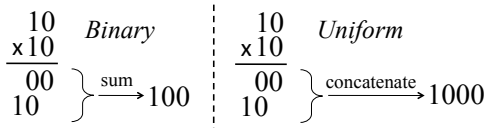


Fig. 3. Multiplication of binary and uniform numbers

When using a uniform encoding, we do not need to sum the output of a logic gate in a particular order to get back the same representation as the inputs. We have "proportions in", "proportions out". In contrast, a weighted encoding requires additional circuitry to add the partial products in the correct manner. This is why the arithmetic logic of a stochastic representation is so simple, the slide and sum operations of convolution are passively provided by the representation. Convolution of proportions only requires logic operations that result in bitwise (or element-wise) multiplication of the particular number format.

These insights lead us to ask: if the process can be described as multiplying every bit of one proportion by every bit of another proportion, or equivalently, by sliding and multiplying deterministic numbers, is randomness actually a requirement? Can the cost and latency be reduced if one approaches the problem deterministically?

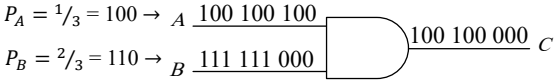
IV. Deterministic Interpretation

A. A Link Between Representations

Equation 6 gives us a link between independent stochastic bit-streams and deterministic bit-streams. We can substitute independent stochastic bit-streams for deterministic bit-streams if Equation 6 holds, that is, if we maintain the property that proportion A sees every bit of proportion B.

Example 5 Two registers contain deterministic unipolar proportions $p_A = 1/3$ and $p_B = 2/3$. How can we generate bit-streams such that a single AND gate performs multiplication?

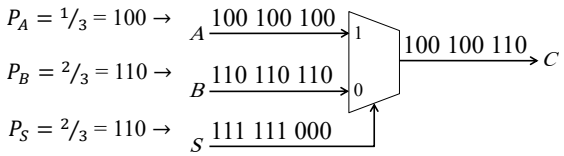
From Equation 6 we know each bit of p_A must be ANDed with each bit of p_B. Therefore, each bit-stream should be a redundant encoding that maintains Equation 6. One method, shown in a later section, is to clock divide one proportion while the other repeats:



$$p_C = p_A p_B = 2/9$$

Example 6 Three registers contain deterministic unipolar proportions $p_A = 1/3$, $p_B = 2/3$, and $p_S = 2/3$. How can we generate bit-streams such that a two-input multiplexer performs scaled addition?

A multiplexer performs the logical operation $(S \wedge A) \vee (\neg S \wedge B)$, where $\wedge$ is AND, $\vee$ is OR, and $\neg$ is NOT. It can be constructed using two AND gates, an inverter, and an OR gate. Because the circuit simply selects the output of either AND gate, bit-streams A and B do not need to be independent from each other. Only bit-stream S is required to be independent from A and B. Clock dividing S while A and B repeat performs scaled addition:



$$p_C = p_S p_A + (1 - p_S) p_B = 2/9 + 2/9 = 4/9$$

In these examples, Equation 6 is maintained on deterministic bit-streams.

B. Comparing the Representations

A stochastic representation passively maintains the property that each bit of one proportion sees every bit of the other proportion, but this property occurs *on average*, meaning the bit-streams have to be much longer than the resolution they represent due to random fluctuations. Equation 8 defines the bit-stream length N required to estimate the average proportion within an error margin ε [10].

$$N > \frac{p(1-p)}{\varepsilon^2} \tag{8}$$

To represent a value within a binary resolution $1/2^n$, the error margin ε must equal $1/2^{n+1}$. Therefore, the bit-stream must be greater than 2^{2n} uniform bits long, as the $p(1-p)$ term is at most equal to 2^{-2} [10]. This means the length of a stochastic bit-stream increases *exponentially* with the desired resolution. This results in enormously long bit-streams. For example, if we want to find the proportion of a random bit-stream with 10-bit resolution ($1/2^{10}$), we'll have to observe at least 2^{20} bits. This is over a thousand times longer than the bit-stream required by a deterministic uniform representation.

The computations also suffer from some level of correlation between bit-streams. This can cause the results to bias away from the correct answer. For these reasons, stochastic logic has only been used to perform approximate computations.

Another related issue is that the LFSRs must be at least as long as the desired resolution to produce bit streams that are sufficiently random. A "Randomizer Unit", described in [4], uses a comparator and LFSR to convert a binary encoded number into a random bit-stream. Each independent random bit-stream requires its own generator. Therefore, circuits requiring i independent inputs with n-bit resolution need i LFSRs with length L approximately equal to $2n$. This results in the LFSRs dominating a majority of the circuit area.

By using deterministic bit-streams, we avoid all problems associated with randomness while retaining all the computational benefits associated with a stochastic representation. For instance, the deterministic representation retains all the fault-tolerance properties attributed to a stochastic representation because it also uses a uniform encoding. To represent a value with resolution $1/2^n$ in a deterministic representation, the bit-stream must be 2^n bits long. The computations are also completely accurate; they do not suffer from correlation.

To utilize a deterministic representation, bit-stream generators must explicitly maintain Equation 6. The next section discusses three methods for generating independent deterministic bit-streams and gives their circuit implementations. Without the requirement of randomness, the hardware cost of the bit stream generators is small.

V. Deterministic Methods

Each method is implemented using a bit-stream generator formed by a group or interconnection of converter modules, as shown in Fig. 4. Each converter module uses the general circuit topology of Fig. 5. The generator takes in operands and generates bit-streams such that:

$$\begin{gathered} G(C_0, C_1, ..., C_{i-1}) \rightarrow \\ \big(C_0\{1\} + (2^{n_0} - C_0)\{0\}\big) \square \big(C_1\{1\} + (2^{n_1} - C_1)\{0\}\big) \square \\ ... \square \big(C_{i-1}\{1\} + (2^{n_{i-1}} - C_{i-1})\{0\}\big) \end{gathered} \tag{9}$$

where i is total number of converter modules that make up the generator, n_i is the binary resolution of the ith individual module, C_i is an operand defining the proportion (or encoded value) of the bit-stream, and each $\square$ can be any arbitrary logical operator.

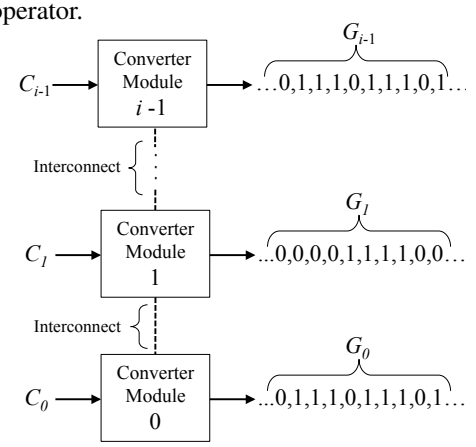


Fig. 4. Deterministic bit-stream generator

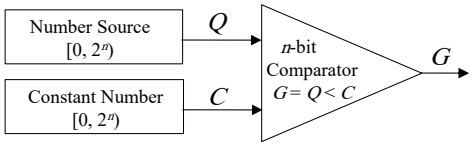


Fig. 5. Converter module

The methods maintain independence by using relatively prime bit lengths, rotation, or clock division. For each method, the hardware complexity of the circuit implementation is given. The computation time of each method is the same; each produces deterministic and completely accurate output.

A. Relatively Prime Bit Lengths

The "relatively prime" method maintains independence by using proportions that have relatively prime lengths (i.e., the ranges $[0, R_i)$ between converter modules are relatively prime).

Fig. 6 demonstrates the method with two bit-streams A and B, one with operand length four and the other with operand length three. The bit-streams are shown in array notation to show the position of each bit in time.

$$\begin{array}{cccccccccccc} a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_0 & b_1 & b_2 & b_0 & b_1 & b_2 & b_0 & b_1 & b_2 \end{array}$$

Fig. 6. Two bit-streams generated by the "relatively prime" method

Independence between bit-streams is maintained because the remainder, or overlap between proportions, always results in a new rotation (or initial phase) of a proportion. Intuitively, this occurs because the bit lengths share no common factors. This results in every bit of each operand seeing every bit of the other operand. For example, a_0 sees b_0, b_1, and b_2; b_0 sees a_0, a_3, a_2, and a_1; and so on. Using two bit-streams with relatively prime bit lengths j and k, the output of a logic gate repeats with period jk. This means with multi-level circuits the output of the logic gates will also be relatively prime. Fig. 7 demonstrates this with a two level circuit.

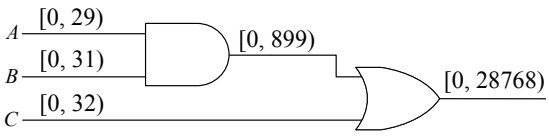


Fig. 7. Arbitrary multi-level circuit with streams generated by the "relatively prime" method

A circuit implementation of the "relatively prime" method is shown in Fig. 8. Each converter module uses a counter as a number source for iterating through each bit of the proportion. The state of the counter Q_i is compared with the proportion constant C_i. The relatively prime counter ranges R_i between modules maintain independence; there are no interconnections between modules. In terms of general circuit components, the circuit uses i counters and i comparators, where i is the number of generated independent bit-streams. Assuming the max range is a binary resolution 2^n and all modules are close to this value (i.e., 256, 255, 253, 251...), the circuit contains approximately i n-bit counters and i n-bit comparators.

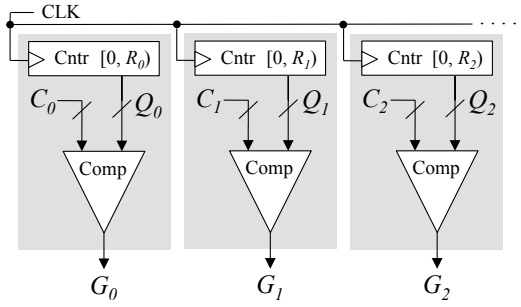


Fig. 8. Circuit implementation of the "relatively prime" method

An interesting property of the proposed "relatively prime bit lengths" method is that it maintains independence between bit-streams irrespective of the order of bits in each operand. As illustrated in Fig. 9, the operands of Fig. 6 can be permuted, and yet every bit of the first operand sees every bit of the second operand exactly once.

$$\begin{array}{cccccccccccc} a_0 & a_3 & a_1 & a_2 & a_0 & a_3 & a_1 & a_2 & a_0 & a_3 & a_1 & a_2 \\ b_1 & b_0 & b_2 & b_1 & b_0 & b_2 & b_1 & b_0 & b_2 & b_1 & b_0 & b_2 \end{array}$$

Fig. 9. Two permuted bit-streams generated by the "relatively prime" method

A limitation of this method is that it requires the inputs to have relatively prime lengths. In this paper, we focus on a digital representation of data, but the "relatively prime" method may also work well with an analog interpretation of the bit-streams, where the value is encoded as the fraction of time the signal is high and the independence property is maintained by using relatively prime (or inharmonic) frequencies [11], [12].

B. Rotation

In contrast to the "relatively prime" method, the "rotation" method allows proportions of arbitrary length to be used. Instead of relying on relatively prime bit lengths, the proportions are explicitly rotated. This requires the sequence generated by the number source to change after it iterates through its entire range. For example, a simple way to generate a bit-stream where the proportion rotates in time is to inhibit or stall a counter every 2^n clock cycles (where n is the length of the counter). Fig. 10 demonstrates this method with two bit-streams, both with proportions of length four.

$$\begin{array}{cccccccccccccccc} a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 & a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 & b_3 & b_0 & b_1 & b_2 & b_2 & b_3 & b_0 & b_1 & b_1 & b_2 & b_3 & b_0 \end{array}$$

Fig. 10. Two bit-streams generated by the rotation method

$$\begin{array}{cccccccccccccccc} a_0 & a_3 & a_1 & a_2 & a_0 & a_3 & a_1 & a_2 & a_0 & a_3 & a_1 & a_2 & a_0 & a_3 & a_1 & a_2 \\ b_1 & b_0 & b_3 & b_2 & b_2 & b_1 & b_0 & b_3 & b_3 & b_2 & b_1 & b_0 & b_0 & b_3 & b_2 & b_1 \end{array}$$

Fig. 11. Two permuted bit-streams generated by the "rotation" method

By rotating bit-stream B's proportion, it is straightforward to see that each bit of one bit-stream sees the other bit-stream's proportion. As with the "relatively prime" method, the operand of each bit-stream can be permuted and yet each bit of one operand sees every bit of the other operand exactly once, as illustrated in Fig. 11.

Assuming all proportions have the same length, we can extend the two bit-stream example to work with multiple bit-streams by inhibiting the number source at powers of the operand length. This allows the operands to rotate relative to longer bit-streams. For example, consider the circuit in Fig. 12. Bit-stream A does not rotate, bit-stream B rotates every 2^n clock cycles, and bit-stream C rotates every 2^{2n} clock cycles. The resultant bit-stream AB of the AND gate *repeats* every 2^{2n} clock cycles and bit-stream C *rotates* every 2^{2n} bits. Therefore bit-stream C rotates relative to the bit-stream AB, maintaining the rotation property for multi-level circuits.

A circuit implementation follows from the previous example. We can generate any number of independent bit-streams as long as the counter of every ith converter module is inhibited every 2^{ni} clock cycles. This can be managed by adding additional counters between each module. These counters control the phase of each converter module and maintain the property that each converter module rotates relative to the other

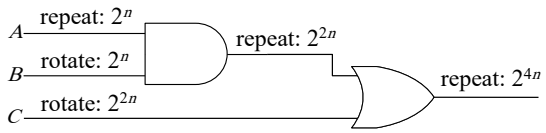


Fig. 12. An arbitrarily chosen multi-level circuit, with bit-streams generated by the "rotation" method

modules. Using n-bit binary counters and comparators, the circuit requires i n-bit comparators and $2i-1$ n-bit counters.

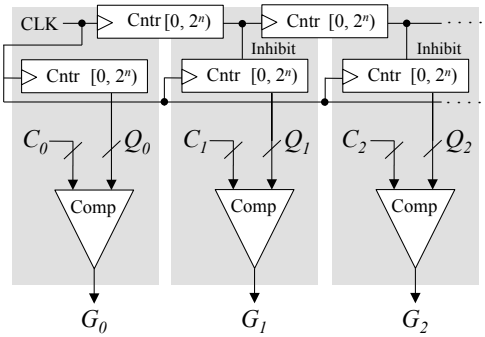


Fig. 13. Circuit implementation of the "rotation" method

The advantage of using rotation as a method for generating independent bit-streams is that we can use operands with the same resolution, but this requires slightly more circuitry than the "relatively prime" method.

C. Clock Division

The "clock division" method works by clock dividing operands. Similar to the "rotation" method, it allows proportions to have arbitrary lengths. This method was first seen in Example 5. Fig. 14 demonstrates this method with two bit-streams, both with proportions of length four. Fig. 15 depicts a permuted version of this method. In both figures bit-stream B is clock divided by the length of bit-stream A's proportion.

$$a_0\,a_1\,a_2\,a_3\,a_0\,a_1\,a_2\,a_3\,a_0\,a_1\,a_2\,a_3\,a_0\,a_1\,a_2\,a_3$$
$$b_0\,b_0\,b_0\,b_0\,b_1\,b_1\,b_1\,b_1\,b_2\,b_2\,b_2\,b_2\,b_3\,b_3\,b_3\,b_3$$

Fig. 14. Two bit-streams generated by the clock division method

$$a_0\,a_3\,a_1\,a_2\,a_0\,a_3\,a_1\,a_2\,a_0\,a_3\,a_1\,a_2\,a_0\,a_3\,a_1\,a_2$$
$$b_1\,b_1\,b_1\,b_1\,b_0\,b_0\,b_0\,b_0\,b_3\,b_3\,b_3\,b_3\,b_2\,b_2\,b_2\,b_2$$

Fig. 15. Two permuted bit-streams generated by the clock division method

Assuming all operands have the same length, we can generate an arbitrary number of independent bit-streams as long as the number source of every ith converter module generates a new number every 2^{ni} clock cycles. This can be implemented in circuit form by simply chaining the converter modules together, as shown in Fig. 16. Using n-bit binary counters and comparators, the circuit requires i n-bit comparators and i n-bit counters. This means the "clock division" method allows operands of the same length to be used with approximately the same hardware complexity as the "relatively prime" method.

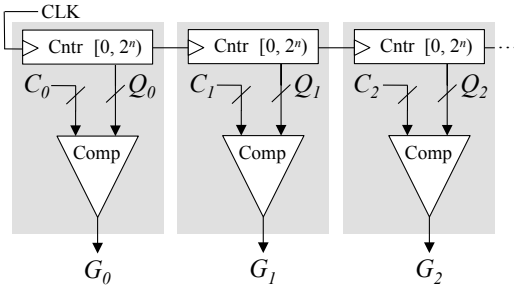


Fig. 16. Circuit implementation of the clock division method

D. Unary vs. Permuted Deterministic Bit-streams

With randomized bit-streams, the accuracy of the computation generally improves as the computation proceeds. This property can be exploited in applications of SC [7]. In contrast, deterministic computation with any of the three methods discussed above does not enjoy this property of "progressive precision." Due to the nature of the representation, – unary bit-streams with first all ones and then all zeros – truncating a stream can change its value significantly and hence result in relatively high error [11].

However, as discussed above, the bits in deterministic streams can be *permuted* without affecting the computation. So instead of using a counter to generate streams, we can use a pseudo-random source such as an LFSR in the converter modules. Here we are not using such constructs for the sake of introducing pseudo-randomness, but rather to permute the bits in streams effectively. The pseudo-random source must have a period equal to the length of the operand. Accordingly, we generate all numbers in the range $[0, R_i)$.

With the "rotation" and "clock division" methods discussed above, we can easily subsitute maximal period LFSRs for counters to improve the progressive precision of the streams.[2] However, with the "relatively prime" method, using an n-bit LFSR as a number source with a prime period of R_i means losing some numbers in the range of $(0, R_i)$.

From the point of view of hardware cost, an n-bit LFSR costs approximately the same as an n-bit counter. However, due to a higher number of bit transitions and so higher switching activity, an LFSR consumes more dynamic power than a counter. If we are going to run the computation to full accuracy, counters will be a better choice. However, for applications that can exploit progressive precision, making early decisions and so truncating the computation, LFSRs might be a better choice. In Section VII, we evaluate the performance and hardware cost of the three proposed deterministic methods, implemented with both counters and LFSRs.

VI. Low Discrepancy Deterministic Methods

In this section, we propose two fast-converging deterministic methods for computation with bit-streams using low-discrepancy (LD) sequences. While the first method is independent of the three methods discussed in Section V, the second method is an integration of the deterministic methods of Section V with LD bit-streams.

[2]We note that the 0-state is skipped in an LFSR. So an n-bit maximal period LFSR has a period of 2^n-1, while a counter has a maximum range and period of 2^n.

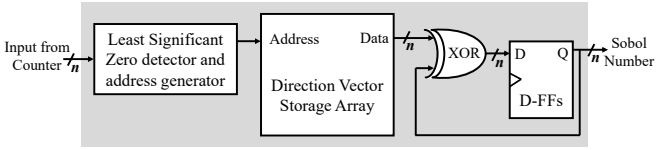


Fig. 17. A Sobol sequence generator [15][13].

A. Low-Discrepancy Sequences in SC

Low discrepancy (LD) sequences have traditionally been used to accelerate the convergence in Monte-Carlo simulations [13]. Recent work on SC [14][15][16] utilized these sequences to improve the speed of computation on stochastic bit-streams. With LD sequences, 1's and 0's in the stochastic streams are uniformly spaced, so the streams do not suffer from random fluctuations. The bit-streams can quickly and monotonically converge to the target value, producing acceptable results in a much shorter time [14].

Alaghi and Hayes proposed the use of LD Halton sequences for SC [14]. A Halton sequence generator consists of a binary-coded base-b counter, where b is a prime number. For d independent input streams in a SC system, d counters with different prime bases must be used. For instance, in the simplest case of multiplying two stochastic bit-streams using an AND gate, one base-2 and one base-3 counter is required. Stochastic bit-streams generated using Halton-based sequences can significantly improve the processing time of SC. However, the base conversion comes at the cost of significant additional hardware overhead [15].

Liu and Han [15] recently proposed a different LD-based stochastic stream generator based on Sobol sequences. Compared to generating Halton sequences, generating Sobol sequences does not require additional base-conversion hardware. The Sobol sequence generator, instead, consists of an address generator that detects the position of the least significant zero, a storage array storing the values of the direction vectors, and a pair of XOR gate and D-type flip-flop for recursively generating random numbers. The structure of the Sobol sequence generator, shown in Fig. 17 (b), was proposed in [13] and used in [15] for generating LD stochastic bit-streams. Different Sobol sequences can be generated by changing the values of the direction vectors.

The authors in [15] showed that the Halton-based stochastic multiplier takes about twice the sequence length to achieve a similar accuracy as the Sobol-based design. An n-bit Sobol generator, on the other hand, has a higher hardware footprint than an n-bit Halton generator. Both these designs consume a similar amount of energy if there is no parallelization [16]. An efficient parallel Sobol generator has been recently developed [16]. With parallelization, the processing time is significantly reduced: multiple Sobol numbers are generated in each cycle at the cost of some additional XOR gates.

B. First Method: Direct LD

The first method uses the LD Sobol sequences and is independent of prior deterministic methods. The required independence between the input bit-streams is guaranteed by simply using different Sobol sequences for generating the bit-streams and processing the streams for a specific number of cycles. The important point for this method is that the precision of the LD sequence generator should be i times the precision of the input data, where i is the number of independent bit-streams. Each input data must be converted to a stream of 2^{in} bits by comparing the input value to 2^{in} different numbers from the sequence generator. For example, to multiply two n-bit precision input data, two $2n$-bit precision Sobol sequence generators are required.

In the following, we see an example of multiplying two 2-bit precision input values using the first proposed method. The first input value is converted to a bit-stream representation using the simplest Sobol sequence (Sobol sequence 1 in Fig. 18). The second input value is converted using the second Sobol sequence from the MATLAB built-in Sobol sequence generator (Sobol sequence 2 in Fig. 18). Note that, when converting to a bit-stream representation, a one is generated if the Sobol number is *less* than the input target number.

Example 7 Deterministic 2-bit precision multiplication using the first proposed method:

$$\begin{array}{r} 1/4 = 1000\ 1000\ 1000\ 1000 \\ 3/4 = 1101\ 1110\ 0111\ 1011 \\ \hline 3/16 = 1000\ 1000\ 0000\ 1000 \end{array}$$

As can be seen, the accurate output of multiplying the two 2-bit precision input values is obtained by directly converting the inputs to 2^4-bit streams, by comparing them to the first 2^4 numbers of two Sobol sequences and ANDing the generated bit-streams.

To prove why the first proposed method produces deterministic and accurate results, we use two important properties of the Sobol sequences:

- The first 2^n numbers of any Sobol sequence include all n-bit precision values in the [0, 1) interval.
- If equally split [0, 1) into 2^n sub-intervals, in any consecutive group of 2^n Sobol numbers starting at positions $i \times 2^n (i = 0, 1, 2, \ldots)$, there is exactly one member in each sub-interval.

Fig. 18 categorizes consecutive groups of 2^2 numbers in the first four Sobol sequences. Each Sobol number in each group is labeled with a number from 0 to 3 depending on its sub-interval. For example 1/8 in Sobol sequence 1 is labeled with a_0 because it is a member of the first sub-interval, [0, 1/4). When converting a 2-bit precision input value into a 2^4-bit stream by comparing it to the first 2^4 numbers of a Sobol sequence, the result is the same for the Sobol numbers with the same label. For example, comparing 3/4 to 5/8 and 11/16 from the Sobol sequence 2 generates the same bit of '1' as both 5/8 and 11/16 are a member of [1/2, 3/4) (label b_2) and so are both less than the input value of 3/4. As can bee seen in Fig. 18, any selected group of 2^2 numbers includes all labels from 0 to 3, and as a result, all groups of the same Sobol sequence will produce the same number of 1s. All groups can accurately present the target input value and their difference will only be in the order of bits (order of labels).

In Section V, we observed that two input values can be multiplied, deterministically and accurately, by simply pairing every bit of one input stream with every bit of the other stream exactly once. As shown in Fig. 18 for $n = 2$, for any pair of two different Sobol sequences, every label u $(u = 0, 1, 2, 3)$

Sobol Seq 1	**0**	**1/2**	**1/4**	**3/4**	**1/8**	**5/8**	**3/8**	**7/8**	**1/16**	**9/16**	**5/16**	**13/16**	**3/16**	**11/16**	**7/16**	**15/16**
	a_0	a_2	a_1	a_3	a_0	a_2	a_1	a_3	a_0	a_2	a_1	a_3	a_0	a_2	a_1	a_3
Sobol Seq 2	**0**	**1/2**	**3/4**	**1/4**	**5/8**	**1/8**	**3/8**	**7/8**	**15/16**	**7/16**	**3/16**	**11/16**	**5/16**	**13/16**	**9/16**	**1/16**
	b_0	b_2	b_3	b_1	b_2	b_0	b_1	b_3	b_3	b_1	b_0	b_2	b_1	b_3	b_2	b_0
Sobol Seq 3	**0**	**1/2**	**1/4**	**3/4**	**7/8**	**3/8**	**5/8**	**1/8**	**11/16**	**3/16**	**15/16**	**7/16**	**5/16**	**13/16**	**1/16**	**9/16**
	c_0	c_2	c_1	c_3	c_3	c_1	c_2	c_0	c_2	c_0	c_3	c_1	c_1	c_3	c_0	c_2
Sobol Seq 4	**0**	**1/2**	**3/4**	**1/4**	**7/8**	**3/8**	**1/8**	**5/8**	**7/16**	**15/16**	**11/16**	**3/16**	**9/16**	**1/16**	**5/16**	**13/16**
	d_0	d_2	d_3	d_1	d_3	d_1	d_0	d_2	d_1	d_3	d_2	d_0	d_2	d_0	d_1	d_3

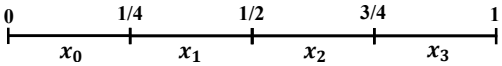


Fig. 18. First 16 numbers of the first four Sobol sequences from MATLAB built-in Sobol sequence generator, and the category of each one based on their position in the [0, 1] interval.

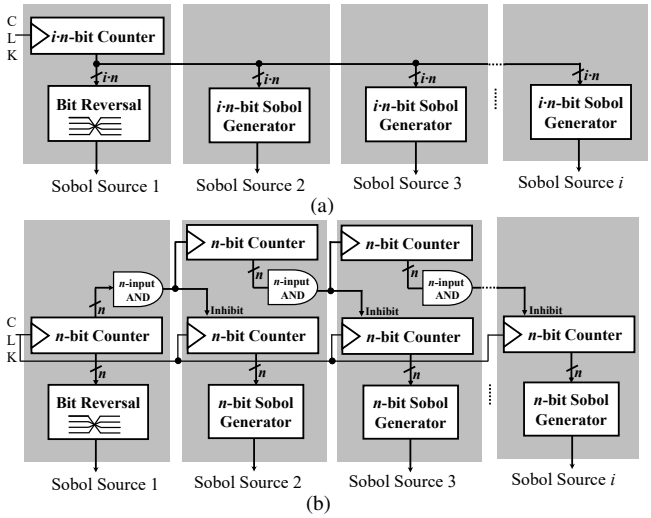


Fig. 19. Structures of the sources of generating Sobol sequences based on (a) first proposed method (b) second proposed method.

in x_u $(x = a,b,c,d)$ meets every label t $(t = 0,1,2,3)$ in y_t $(y = a,b,c,d)$ exactly once. So, the result of multiplying two 2-bit precision numbers by ANDing their 2^4-bit stream representation, generated based on two different Sobol sequences, is deterministic and completely accurate.

This argument can be easily extended to multiplication of i n-bit precision numbers when converting the input numbers to bit-streams of $2^{i.n}$-bit length by comparing them to $2^{i.n}$ numbers from i different Sobol sequences. The generated bit-streams can be divided into groups of 2^n bits. Every bit (label) from a bit-stream interacts with every bit (label) of the other bit-streams exactly once, which results in a deterministic and accurate output bit-stream.

Fig. 19 (a) shows the structure of the sources of generating Sobol sequences for the first proposed LD-based method. These are used as the number sources in the converter module. Note that the simplest Sobol sequence is simply the reverse of the output bits of a binary counter. Accordingly, we generate the first Sobol sequence by hard-wiring the output bits of a counter at no extra hardware cost.

C. Second Method: Integrated LD

The second method integrates LD sequences with the methods of Section V by using LD sequence generators as the number source in converter modules. In contrast to our first LD method, for the second method, the precision of the sequence generator is equal to the precision of the input data. For example, for multiplication of two n-bit precision input data, two n-bit LD sequence generators are required. When using Halton sequences which are generated based on prime numbers, the "relatively prime" method must be used. The Sobol sequences, on the other hand, must be integrated with the "clock division" or "rotation" methods. The operations then must continue for the product of the length of the bit-streams to produce deterministic and accurate results.

We will show in Section VII that the "rotation" method converges faster and is more energy-efficient than the "clock division" method. So, in what follows, we integrate LD Sobol sequences with the rotation method. While we limit our reported results to LD Sobol sequences and the "rotation" approach, the proposed idea can similarly be applied to LD Halton sequences and the "relatively prime" method.

The "rotation" method guarantees a deterministic and accurate output by rotating the bit-streams through inhibiting or stalling on powers of the stream lengths. Fig. 19 (b) shows the structure of the sources for generating Sobol sequences with the second LD method. The first Sobol source repeats every 2^n cycles but do not rotate. Other Sobol sources (source k=2, 3, ..., i) have a period of 2^n but rotate every $2^{(k-1)\cdot n}$ cycles by inhibiting. Additional counters control these inhibits. We will show that, due to the use of n-bit Sobol generators instead of expensive $i \cdot n$-bit generators, the second LD method incurs a lower hardware cost than the first LD method.

In the following, we see an example of multiplying two 2-bit precision input values using the second proposed LD method based on the first two Sobol sequences.

Example 8 Deterministic 2-bit precision multiplication using the second LD-based method:
Sobol source 1 with a period of 2^2 and no rotation:
0,1/2,1/4,3/4, 0,1/2,1/4,3/4, 0,1/2,1/4,3/4, 0,1/2,1/4,3/4
Sobol source 2 with a period of 2^2 and inhibiting after every 2^2 cycles:
0,1/2,3/4,1/4, 1/4,0,1/2,3/4, 3/4,1/4,0,1/2 1/2,3/4,1/4,0

$$\begin{array}{r} 2/4 = 1010\ 1010\ 1010\ 1010 \\ 3/4 = 1101\ 1110\ 0111\ 1011 \\ \hline 6/16 = 1000\ 1010\ 0010\ 1010 \end{array}$$

As can be seen, by exploiting the "rotation" approach, every number in the first four numbers of the Sobol source 1 pairs with every number in the first four numbers of the

TABLE I
MEAN ABSOLUTE ERROR (%) COMPARISON OF THE CONVENTIONAL SC AND THE PROPOSED DETERMINISTIC METHODS WHEN MULTIPLYING TWO 8-BIT PRECISION STOCHASTIC STREAMS WITH DIFFERENT NUMBERS OF OPERATION CYCLES.

Design Approach	Area(μm^2)	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}
Conventional SC - LFSR-32	1399	0.05	0.15	0.26	0.39	0.58	0.79	1.20	1.67	2.32	3.32	4.72
Deter. Rel. Prime Length - Counter	460	0.00	3.03	4.70	6.01	7.08	7.62	7.90	7.98	8.11	33.2	51.5
Deter. Rotation - Counter	617	0.00	3.10	4.84	6.15	7.08	7.66	7.99	8.17	8.26	33.1	51.8
Deter. Clock Division - Counter	460	0.00	12.3	18.7	21.8	23.4	24.0	24.4	24.5	24.9	49.6	62.2
Deter. Rel. Prime Length - LFSR	481	∼0.00	0.09	0.16	0.24	0.34	0.47	0.60	0.72	0.85	2.56	4.22
Deter. Rotation - LFSR	637	0.00	0.09	0.16	0.24	0.35	0.47	0.60	0.71	0.82	2.56	4.26
Deter. Clock Division - LFSR	481	0.00	1.44	2.48	3.74	5.28	7.18	9.91	14.2	24.8	25.0	25.8
Deter. First LD	3361	0.00	0.0003	0.0013	0.0035	0.009	0.019	0.041	0.092	0.190	0.451	0.921
Deter. Second LD	1277	0.00	0.0013	0.0033	0.0075	0.014	0.031	0.059	0.112	0.190	0.451	0.921

Sobol source 2 exactly once. This has led to a deterministic and accurate multiplication when these rotated sequences of numbers are used in converting the input values, 2/4 and 3/4, into bit-stream representation.

VII. EXPERIMENTS

In this section, we first compare the accuracy and hardware cost of the proposed deterministic methods with conventional stochastic method for multiplication. We then compare implementations of a well-known digital image processing algorithm, the Robert's cross edge detection, from the point of view of performance, hardware cost, and area-delay product.

A. Accuracy Comparison

For an accuracy comparison of the proposed deterministic methods with conventional SC, we exhaustively tested the multiplication of two 8-bit precision input data in the [0, 1] interval. For the conventional SC and for the deterministic LFSR-based approaches, the input data was selected from a large set of random input values. For the counter-based and the LD-based deterministic methods, we tested the multiplication operation on every possible pair of input values.

For the conventional SC and for the LFSR-based deterministic methods we used maximal period 32-bit and 8-bit LFSRs, respectively, as the number source in the converter module. Two different LFSRs (i.e., different designs with different seeds) were used in each case. An n-bit maximal period LFSR has a period of 2^n-1, as the 0-state in the LFSR is normally not used. Here, for a fair comparison between the LFSR-based deterministic methods and the counter-based and the LD-based deterministic approaches, we added a 0-state to the set of the states of each 8-bit LFSR to generate 2^8 unique numbers. For the LFSR-based relatively prime method, the period of the second LFSR was fixed at 2^8-1. For the LD-based methods, we used the first and the second Sobol sequences from the MATLAB built-in Sobol sequence generator (see Fig. 18).

Table I compares the mean absolute error (MAE) of different methods. We multiply the measured mean value of each method by 100 and report it as a percentage. Due to inherent random fluctuations in generating bit-streams and correlation between bit-streams, conventional SC cannot produce completely accurate results in 2^{16} cycles. All the proposed deterministic methods, on the other hand, produced completely accurate results in 2^{16} cycles. The results were the same as the results from a conventional binary radix-based multiplier.

Due to the nature of unary representation, truncating the bit-streams in the counter-based deterministic methods leads to a high truncation error. For example, when running the multiplication operation for 2^{15} cycles (processing 2^{15}-bit streams), the three counter-based deterministic methods showed a MAE of more than 3 percent. For applications where slight inaccuracy is acceptable this high truncation error makes the conventional SC a better choice than the counter-based deterministic methods.

We solve the high truncation error of the counter-based deterministic methods by bringing randomization back into representation. Instead of counters, we use LFSRs as the number source with the "relatively prime length," "rotation," and "clock division" methods. As can be seen in Table I, when truncating bit-streams, the three LFSR-based deterministic methods achieve a much lower MAE than their corresponding counter-based implementations. The LFSR-based relatively prime length and rotation methods even achieved a lower MAE than conventional SC.

The best MAEs, however, were produced by the two LD deterministic methods. When using the proposed LD methods, the MAEs of the truncated computation are significantly lower than those of the conventional SC and the counter-based and LFSR-based deterministic methods. For example, when running the two-input multiplication operation for 2^{15} cycles, the proposed LD methods achieve a MAE of approximately 10^{-3}, which is 150×, 100×, and 3000× lower than the MAE of the conventional SC, the deterministic LFSR-based rotation, and the counter-based rotation methods, respectively.

B. Cost Comparison

Perfectly precise computations require the output resolution to be at least equal to the product of the input resolutions. This is demonstrated in Equation 6, where to precisely compute the output of a logic gate given two proportions, each bit of one proportion must be operated on with every bit of the other proportion. For example, with proportions of size n and m, the precise output contains nm bits.

Assuming each independent input i has the same resolution $1/2^{n_{in}}$, the output resolution is given by $1/2^{n_{out}} = 1/2^{n_{in}{}^i}$. As discussed in Section IV, with a conventional stochastic representation, bit-streams of 2^{2n}-bit long are required to represent a value with $1/2^n$ precision. To ensure the generated bit-streams are sufficiently random and independent, each LFSR have at least as many states as the required output bit-stream.

TABLE II
HARDWARE AREA COST (μm^2) OF THE MULTIPLIER WITH DIFFERING DATA PRECISION (N) AND NUMBER OF INPUTS (I)

Design Approach	N=4 i=2	N=4 i=3	N=4 i=4	N=8 i=2	N=8 i=3	N=8 i=4	Stream Generator Structure
Binary radix	165	512	993	688	2104	4945	-
Conventional SC - LFSR	697	1496	2632	1399	2974	5166	i i*2*N-bit LFSRs + i N-bit Comp
Deter. Rel. Prime - Counter	212	318	424	460	690	920	i N-bit Counter + i N-bit Comp
Deter. Rotation - Counter	287	467	648	617	1003	1390	1 N-bit Counter + i-1 N-bit CounterEN + i N-bit Comp
Deter. Clock Div - Counter	212	318	424	460	690	920	i N-bit Counter + i N-bit Comp
Deter. Rel. Prime - LFSR	226	339	452	481	721	961	i N-bit LFSR + i N-bit Comp
Deter. Rotation - LFSR	301	489	676	637	1034	1431	1 N-bit LFSR + i-1 N-bit LFSREN + i N-bit Comp
Deter. Clock Div - LFSR	226	339	452	481	721	961	i N-bit LFSR + i N-bit Comp
Deter. First LD	1005	3740	9127	3361	13193	32406	1 i*N-bit Counter + i-1 i*N-bit Sobol Gen+ i N-bit Comp
Deter. First LD - 4x Parallel	1388	4456	10259	4119	14714	35090	1 i*N-bit Counter + i-1 i*N-bit 4x-Par. Sobol Gen [16]+ i*4 N-bit Comp
Deter. Second LD	456	806	1156	1277	2324	3371	i-1 N-bit Counter+i-1 N-bit (CounterEN+SobolGen)+i N-bit Comp

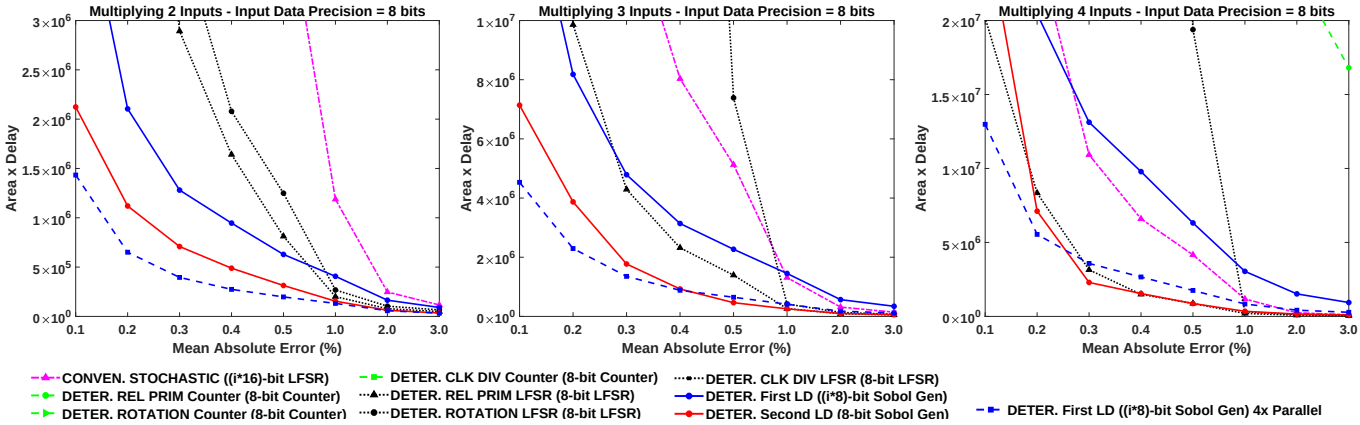


Fig. 20. Area×Delay of 8-bit precision multipliers for different MAEs. Note that the Area×Delay numbers for some methods were much larger than other method and out of the range shown in the figure.

Therefore, to compute with perfect precision, each LFSR must have at least length $2n_{in}i$.

With the deterministic methods, the resolution n of each input i is determined by the length of its converter module number source. The output resolution is simply the product of the number source ranges. For example, with the "clock division" method, each converter module number source is connected in series. With i inputs each with resolution n, the series connection forms a large number source with 2^{ni} states. This shows that output resolution is not determined by the length of each individual number source, but by their concatenation. This allows for a large reduction in circuit area compared to the conventional stochastic method.

The hardware area costs of the proposed deterministic methods for the case of implementing a 2-input 8-bit precision multiplier are compared in Table I. We synthesized the designs using the Synopsys Design Compiler vH2013.12 with a 45nm gate library. As can be seen in the table, the proposed counter-based and LFSR-based deterministic methods have a significantly lower hardware cost than the conventional stochastic method. The two LD-based deterministic methods have a higher cost than other deterministic methods due to LD Sobol sequence generators. The first LD method is 2.6 times more costly than the second LD method, due to the expense of generating 16-bit Sobol sequences.

C. Scalability Evaluation

To evaluate the scalability of the proposed methods, we extend our experiment to 3-input and 4-input stochastic multipliers, with 4-bit and 8-bit precision. The hardware area costs are reported in Table II. We also report the area numbers for a conventional binary radix multiplier. As can be seen in the reported numbers, among the bit-stream-based designs the "relatively prime length" and "clock division" methods have the lowest hardware cost. These two methods and the "rotation" method have the lowest cost increase rate ($\sim 2\times$) from 2 inputs to 4 inputs. The first LD method, which had the fastest convergence, as shown in Table I, has the highest hardware cost with highest cost increase rate ($9\times$) from 2 inputs to 4 inputs. The second LD method, on the other hand, is also fast converging, but has a cost increase rate ($2.5\times$) very close to the cost increase rates of the counter-based and LFSR-based deterministic methods. The conventional stochastic method is more costly than all implemented deterministic methods except the first LD method. The hardware cost of the binary radix-based multiplier increases by a factor of $7\times$ going from 2 inputs to 4 inputs. This increase is greater than all the proposed methods except the first LD method.

The important metric, however, to evaluate the efficiency of different methods is the area-delay product as an estimation of energy consumption. The area-delay of the implemented 8-bit precision multipliers for different MAEs is shown in Fig. 20. We first exhaustively tested each design approach with a large set of input values and found the average processing time of each one to achieve a specific MAE rate. We then multiplied the processing time by the corresponding design hardware area cost to produce the area-delay product. As can be seen in Fig. 20, the second LD method and the 4x parallel structure

of the first LD method have shown the lowest area-delay product among the implemented multipliers. When increasing the number of inputs, the LFSR-based "relatively prime" method and the conventional stochastic method approach these methods. The high hardware cost of the first LD method makes its area-delay product worse than that of the conventional SC and the relatively prime LFSR-based method as the number of inputs increases. The 4x parallel version of the first LD method, however, is more efficient due to its parallel structure. Its area-delay product grows comparatively slowly with the number of inputs.

From our scalability evaluation, we make the following conclusions:

- When completely accurate results are desired, the "relatively prime length" and the "clock division" methods are the best choices. They have the lowest hardware cost; so for the same operation time, equal to the product of the length of the bit-streams, these provide the minimum area-delay product.
- When an application can tolerate some small degree of inaccuracy and only a few independent bit-streams are used, the second and the parallel version of the first LD-based methods are the best choices. Note that some SC-based implementations of neural networks require only two independent sources for generating bit-streams [17]. In the next section, we show that these LD-based methods provide the minimum area-delay product for the Robert's cross edge detection circuit.
- When an application can tolerate relatively high inaccuracy and a large number of independent inputs are needed, a conventional stochastic implementation is the best choice.

The LFSR-based "relatively prime" method showed good scalability (see Fig. 20). However, implementing LFSRs with relatively prime periods can be difficult; also precisely representing arbitrary n-bit numbers with prime stream lengths can be problematic. Unlike conventional SC, the converter for the "relatively prime" method is also complex and costly since its bit-stream length is not a power of 2; a divider must be used. Accordingly a conventional stochastic implementation might be a better choice when a large number of independent *digital* bit-streams are required. Still, the "relatively prime" method may be an excellent solution for the mixed-signal design of stochastic circuits, where the values are encoded as the fraction of time the signal is high and represented using *analog* pulse-width modulated signals. A low-cost active integrator can be used as the converter in this case to average the output signal [11][12].

D. Robert's Cross Edge Detection Implementation

To further evaluate the proposed deterministic methods we studied implementations of a well-known digital image processing algorithm, the Robert's cross edge detection. In this edge detector, each operator consists of a pair of 2×2 convolution kernels that processes the pixels of the input images based on their three immediate neighbors:

$$Y_{i,j} = S \times (|X_{i,j} - X_{i+1,j+1}| + |X_{i,j+1} - X_{i+1,j}|)$$

where $X_{i,j}$ is the scaled value of the pixel at location (i, j) of the input image and $Y_{i,j}$ is the corresponding output value. Since the value of the pixels are originally in the range of [0, 255] we need to scale down the inputs to the [0,1] interval to process these with stochastic logic. S is also often set to 0.5 to scale down the output range from [0,2] to [0,1]. Fig. 21 shows the stochastic implementation of the Robert's cross algorithm proposed in [18]. The multiplexer (MUX) unit performs the scaled addition operation with the scaling factor of S. For accurate scaled addition, the bit-stream connected to the select input of the MUX should be independent of the MUX's two main input bit-streams [19]. Setting the S input to 0.5 would limit our experiment to a 2-bit precision select bit-stream. This could favor the LD-based bit-streams (see the properties of Sobol sequences in Section VI-B). So to generalize our evaluation and provide a fair comparison we use 8-bit precision random input values as the scaling factor. This changes the effective operation to:

$$Y_{i,j} = (1-S) \times |X_{i,j} - X_{i+1,j+1}| + S \times |X_{i,j+1} - X_{i+1,j}|$$

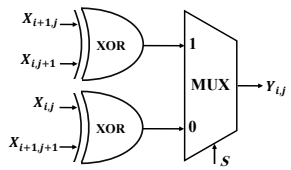


Fig. 21. Stochastic circuit for Robert's cross edge detection algorithm [18].

The two XOR gates compute absolute value subtraction if they are fed with correlated input streams (streams with maximum overlap between 1s) [18]. Sharing the same number source in generating the input streams provides correlated bit-streams. For the circuit shown in Fig. 21 the bit-streams connected to the inputs of the XOR gates should be correlated to each other but should be independent of the MUX's select input bit-stream. Two number generators are therefore required for this circuit; one for converting the four main inputs and one for converting the select input.

We evaluate the performance, hardware area, and area-delay product of the Robert's cross stochastic circuit in four different cases: 1) the conventional approach of processing random streams, 2) the proposed deterministic approaches of processing counter-based unary streams, 3) the proposed deterministic approaches of processing LFSR-based pseudo-random streams, and 4) the proposed deterministic methods of processing LD streams. The circuit shown in Fig. 21 is the core stochastic logic and will be shared between all cases.

For the relatively prime methods, we fix the period of the first and the second number sources on 2^8 and 2^8-1, respectively. LFSRs and counters with periods of 2^8 and 2^8-1 are therefore implemented. For the conventional SC, we use two different 8-bit or two different 16-bit LFSRs as the required sources of random numbers. For the first LD method, similar to Fig. 19(a), one 16-bit counter and one 16-bit precision Sobol generator (both with a period of 2^{16}) are implemented. For the 4x parallel structure of the first LD-based method we implement one 16-bit 4x parallel counter and one 16-bit 4x parallel Sobol generator, as proposed in [16]. Both the counter and the Sobol generator have a period of 2^{16} and generate four numbers in each cycle. For the second

TABLE III
AREA (um^2), DELAY (NUMBER OF CYCLES), AND AREA×DELAY/1000 OF THE ROBERT'S CROSS STOCHASTIC CIRCUIT SYNTHESIZED WITH THE 8- AND 16-BIT CONVENTIONAL SC APPROACH, AND ALSO THE PROPOSED DETERMINISTIC APPROACHES.

		Target Error (MAE)															
Design	Area	0%		0.01%		0.1%		0.3%		0.5%		1.0%		2.0%		3.0%	
Approach	(μm^2)	Cycle	A×D	Cycle	A×D	Cycle	A×D	Cycle	A×D	Cycle	A×D	Cycle	A×D	Cycle	A×D	Cycle	A×D
Conventional SC 8	700	-	-	-	-	-	-	-	-	-	-	-	-	166	116	100	70
Conventional SC 16	1000	-	-	-	-	51,081	51,101	10,721	10,725	4,021	4,023	1,216	1,216	291	291	116	116
Deter. Rel.Prime-Counter	679	65,280	44,351	65,255	44,334	64,430	43,774	62,640	42,558	60,960	41,416	56,730	38,542	48,550	32,985	40,805	27,723
Deter. Rotation-Counter	711	65,536	46,616	65,514	46,600	64,672	46,001	62,860	44,712	61,102	43,462	56,890	40,466	48,650	34,605	40,274	28,647
Deter. Clk Div-Counter	679	65,536	44,525	65,511	44,508	65,091	44,223	64,091	43,543	63,091	42,864	60,566	41,149	55,506	37,711	50,511	34,317
Deter. Rel.Prime-LFSR	700	65,280	45,709	64,225	44,970	27,800	19,466	4,540	3,179	1,685	1,180	445	312	165	116	100	70
Deter. Rotation-LFSR	755	65,536	49,512	64,626	48,825	27,866	21,053	4,746	3,586	1,661	1,255	446	337	161	122	96	73
Deter. Clk Div-LFSR	700	65,536	45,888	65,471	45,843	64,101	44,884	56,126	39,299	46,226	32,367	24,626	17,243	8,206	5,746	3,691	2,584
Deter. First LD	3581	65,536	234,678	8,188	29,320	1,008	3,610	368	1,318	192	688	92	329	40	143	28	100
Deter. First LD 4x Parallel	3581	16,384	91,898	2,048	11,487	252	1,413	92	516	48	269	24	135	10	56	8	45
Deter. Second LD	1496	65,536	98,062	14,384	21,523	1,272	1,903	382	572	192	287	92	138	40	60	28	42

LD method, similar to Fig. 19(b), one 8-bit counter, one 8-bit counter with enable input, and one 8-bit precision Sobol generator, all with a period of 2^8, are implemented. For all other methods, the number sources have a period of 2^8. Five 8-bit comparators are used in all different cases, except the 4x parallel design with 20 8-bit comparators, to compare the output of the number sources with the input values to generate the corresponding bit-streams.

Table III compares the hardware footprint, delay, and area-delay product numbers. To comprehensively test the designs, we simulate the operation of the Robert's cross circuit in each design approach by processing 2,000 sets of 8-bit precision random input values. For the accurate representation of input values in each design, we randomly choose integer values between zero and the period of the number generators and divide the integer value by their period.

When completely accurate results are expected, the deterministic designs must run for the same number of cycles (product of the periods of the number generators). Considering the higher hardware cost of the LD-based designs, the three counter-based and the three LFSR-based implementations have a lower area-delay product than the LD-based designs. Among these, the rotation-based methods have a slightly higher hardware cost and hence higher area-delay product than the "relatively prime" and "clock division" methods. LFSRs have a higher switching activity than counters. So, if factoring in the dynamic power consumption, for the case of implementing highly accurate Robert's cross circuit, the counter-based deterministic methods are more efficient than the LFSR-based deterministic methods.

The advantage of the LFSR-based methods compared to the counter-based methods becomes apparent when slight inaccuracy in the computation is acceptable. In such cases, the LFSR-based designs give a significantly lower area×delay, as they converge to the expected accuracy much quicker.

For the "relatively prime" and "rotation" approaches the proposed LFSR-based designs improve the processing time by 61% compared to the corresponding counter-based deterministic designs, when accepting an MAE of as low as 0.1%. This results in an area-delay saving of around 55%. For an MAE of 3.0%, these LFSR-based architectures showed $\sim$395× lower area-delay product by improving the processing time more than 400× compared to the counter-based architectures. For the "clock division" approach, the LFSR-based design of the Robert's cross circuit is more efficient when an MAE of 0.3% or greater is acceptable. The area-delay product decreases by a factor 13 with this method, with MAE of 3%.

Compared to conventional SC designs, LFSR-based deterministic structures had a smaller area-delay product when the conventional design implemented with 16-bit LFSRs, but roughly the same area-delay product when implemented with 8-bit LFSRs. The important point, however, is that a conventional SC design cannot achieve an MAE of 1.0% or lower when implemented with 8-bit LFSRs; a conventional SC design implemented with 16-bit LFSRs requires inordinately long processing times to produce completely accurate results.

Due to fast convergence, the LD-based implementations of the Robert's cross circuit can satisfy a fixed requirement for accuracy with much shorter processing time than the counter-based and LFSR-based deterministic implementations. This leads to a much lower area×delay for these implementations. As the results of Table III show, the 4x parallel structure of the first LD and also the second LD methods provide the best area-delay products for MAEs greater than 0.01%.

VIII. CONCLUSION

There has been widespread interest in the idea of stochastic logic in recent years. We point to [5] and [7] for surveys of work in the area. While numerous papers have advocated the advantages of the paradigm, the narrative has never been compelling. Yes, the paradigm permits complex arithmetic operations to be performed with remarkably simple logic, but generating the bit-streams is costly, essentially offsetting the benefit. The long latency, poor precision, and random fluctuations are near disastrous for most applications.

While it is easy conceptually to understand how SC works, randomness is costly. This paper argues that randomness is not necessary. Instead of relying upon statistical sampling to

operate on bit-streams, we can explicitly "convolve" them: we slide one operand past the other, performing bitwise operations. We argue that the logic to perform this convolution is less costly than that to randomized bit-streams. The results of our computation are predictable and completely accurate for all input values. Most importantly, we can use much shorter bit-streams to achieve the same accuracy as with statistical sampling through randomness. We conclude that there is no clear reason to use randomness. Even when randomness is free, say harvested from thermal noise or some other physical source, SC entails very high latency. In contrast, computation on deterministic uniform bit-streams is less costly, has much lower latency, and is completely accurate.

We do note that there is one drawback to the approach: bit-stream lengths grow with each level of logic. This is, in fact, a mathematical requirement. Consider the multiplication of two numbers, each encoded with a precision of n binary bits. Regardless of the encoding, the precision of the result must be greater than the precision of the two operands: up to n^2 bits are required. Stochastic encodings have the same requirement. However, with randomness it is easy to approximate the result, by simply truncating the length of the streams. Accordingly, most stochastic circuits keep constant bit-stream lengths regardless of the levels of logic.

To address this issue, we explored variants of our deterministic approach that permit truncation, and so limit the increase in the bit-stream lengths with multiple levels of logic. We proposed two methods based on low-discrepancy (LD) sequences. These methods provide the best accuracy and the lowest area$\times$delay. Although the hardware area cost of the LD-based methods is higher, they provide progressive precision: the longer the algorithm runs the more precise the computation becomes. In future work, we will explore applications of these techniques in areas such as image processing and machine learning.

ACKNOWLEDGMENT

This work was supported in part by National Science Foundation grant no. CCF-1408123 and CCF-1438286. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. Portions of this work were presented in the 35th IEEE/ACM International Conference on Computer Aided Design (ICCAD) [20], the 35th IEEE International Conference on Computer Design (ICCD) [21], and in the 37th IEEE/ACM International Conference on Computer Aided Design (ICCAD) [22].

REFERENCES

[1] B. R. Gaines, *Stochastic Computing Systems*, ser. Advances in Information Systems Science. Springer US, 1969.

[2] B. D. Brown and H. C. Card, "Stochastic neural computation i: Computational elements," *IEEE Transactions On Computers, vol. 50, no. 9, pages 891-905*, 2001.

[3] W. Qian and M. D. Riedel, "Synthesizing logical computation on stochastic bit streams," *DAC*, 2009.

[4] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE Transactions on Computers*, vol. 60, pp. 93–105, 2011.

[5] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Transaction on Embedded Computing*, vol. 12, 2013.

[6] S. S. Tehrani, A. Naderi, G.-A. Kamendje, S. Hemati, S. Mannor, and W. J. Gross, "Majority-based tracking forecast memories for stochastic ldpc decoding," *IEEE Transactions on Signal Processing*, vol. 58, pp. 4883–4896, 2010.

[7] A. Alaghi, W. Qian, and J. P. Hayes, "The Promise and Challenge of Stochastic Computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 8, 2018.

[8] P. K. Gupta and R. Kumaresan, "Binary multiplication with pn sequences," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 4, pp. 603–606, April 1988.

[9] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992.

[10] W. Qian, "Digital yet deliberately random: Synthesizing logical computation on stochastic bit streams," Ph.D. dissertation, University of Minnesota, 2011.

[11] M. H. Najafi and S. Jamali-Zavareh and D. J. Lilja and M. D. Riedel and K. Bazargan and R. Harjani, "Time-Encoded Values for Highly Efficient Stochastic Circuits," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1644–1657, May 2017.

[12] M. H. Najafi, S. Jamali-Zavareh, D. J. Lilja, M. D. Riedel, K. Bazargan, and R. Harjani, "An Overview of Time-Based Computing with Stochastic Constructs," *IEEE Micro*, vol. 37, no. 6, pp. 62–71, November 2017.

[13] I. L. Dalal, D. Stefan, and J. Harwayne-Gidansky, "Low discrepancy sequences for monte carlo simulations on reconfigurable platforms," in *2008 International Conference on Application-Specific Systems, Architectures and Processors*, July 2008, pp. 108–113.

[14] A. Alaghi and J. Hayes, "Fast and accurate computation using stochastic circuits," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–4.

[15] S. Liu and J. Han, "Energy efficient stochastic computing with Sobol sequences," in *DATE 2017*, March 2017, pp. 650–653.

[16] S. Liu and J. Han, "Toward Energy-Efficient Stochastic Circuits Using Parallel Sobol Sequences," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1326–1339, July 2018.

[17] S. R. Faraji, M. H. Najafi, B. Li, K. Bazargan, and D. J. Lilja, "Energy-Efficient Convolutional Neural Networks with Deterministic Bit-Stream Processing," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2019*, March 2019, pp. 1–6.

[18] A. Alaghi, C. Li, and J. Hayes, "Stochastic circuits for real-time image-processing applications," in *50th DAC*, May 2013, pp. 1–6.

[19] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *Computers, IEEE Trans. on*, vol. 60, no. 1, pp. 93–105, Jan 2011.

[20] D. Jenson and M. Riedel, "A Deterministic Approach to Stochastic Computation," in *the 35th ICCAD*, 2016, pp. 102:1–102:8.

[21] M. H. Najafi and D. J. Lilja, "High Quality Down-Sampling for Deterministic Approaches to Stochastic Computing," in *Computer Design (ICCD), 2017 IEEE 35th International Conference on*, Nov 2017.

[22] M. H. Najafi, D. J. Lilja, and M. Riedel, "Deterministic Methods for Stochastic Computing using Low-Discrepancy Sequences," in *Proceedings of the 37th International Conference on Computer-Aided Design*, ser. ICCAD '18, 2018.

M. Hassan Najafi received the B.Sc. degree in Computer Engineering from the University of Isfahan, Iran, the M.Sc. degree in Computer Architecture from the University of Tehran, Iran, and the Ph.D. degree in Electrical Engineering from the University of Minnesota, Twin Cities, USA, in 2011, 2014, and 2018, respectively. He is currently an Assistant Professor with the School of Computing and Informatics, University of Louisiana at Lafayette, Louisiana, USA. His research interests include stochastic and approximate computing, unary processing, computer-aided design, and machine-learning. In recognition of his research, he received the 2018 EDAA Outstanding Dissertation Award, the Doctoral Dissertation Fellowship from the University of Minnesota, and the Best Paper Award at the 2017 35th IEEE International Conference on Computer Design (ICCD).

David J. Lilja (F'06) received the B.S. degree in computer engineering from Iowa State University in Ames, IA, USA, and the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois at Urbana-Champaign in Urbana, IL, USA. He is currently the Schnell Professor of Electrical and Computer Engineering at the University of Minnesota in Minneapolis, MN, USA, where he also serves as a member of the graduate faculties in Computer Science, Scientific Computation, and Data Science. Previously, he served ten years as the head of the ECE department at the University of Minnesota, and worked as a research assistant at the Center for Supercomputing Research and Development at the University of Illinois, and as a development engineer at Tandem Computers Incorporated in Cupertino, California. He was elected a Fellow of the Institute of Electrical and Electronics Engineers (IEEE) and a Fellow of the American Association for the Advancement of Science (AAAS). His main research interests include computer architecture, parallel processing, computer systems performance analysis, approximate computing, and storage systems.

Marc D. Riedel (SM12) received the B.Eng. degree in electrical engineering from McGill University, Montreal, QC, Canada, and the M.Sc. and Ph.D. degrees in electrical engineering from the California Institute of Technology (Caltech), Pasadena, CA, USA. He is currently an Associate Professor of electrical and computer engineering with the University of Minnesota, Minneapolis, MN, USA, where he is a member of the Graduate Faculty of biomedical informatics and computational biology. From 2004 to 2005, he was a Lecturer of computation and neural systems with Caltech. He was with Marconi Canada, CAE Electronics, Toshiba, and Fujitsu Research Labs. Dr. Riedel was a recipient of the Charl H. Wilts Prize for the Best Doctoral Research in Electrical Engineering at Caltech, the Best Paper Award at the Design Automation Conference, and the U.S. National Science Foundation CAREER Award.