A Deterministic Approach to Stochastic Computation

Devon Jenson and Marc Riedel

jens1172@umn.edu, mriedel@umn.edu

Department of Electrical and Computer Engineering, University of Minnesota

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. The random or pseudorandom sources required to generate the representation are costly, consuming a majority of the circuit area (and diminishing the overall gains in area). 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. The three methods are evaluated on a collection of arithmetical functions. 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.

I. INTRODUCTION

In the paradigm of stochastic computation, digital logic is used to perform computation on random bit streams, where numbers are represented by the probability of observing a one [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.

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. 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 stochastic computation, all bits in our deterministic streams are weighted equally. Accordingly, our deterministic circuits display the same high degree of tolerance to soft errors.

This paper is structured as follows: Section II presents background information on stochastic computing. Section III gives an intuitive view of why stochastic computation 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 compares the hardware complexity and latency of stochastic and 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 [1]. 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 Figure 1.
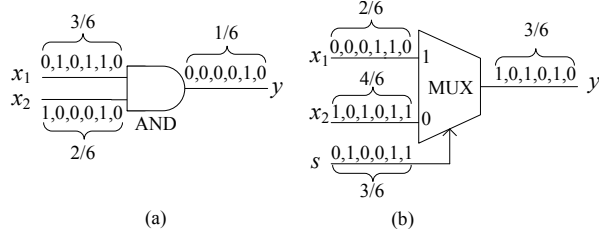
Fig. 1: Stochastic implementation of arithmetic operations: (a) Multiplication; (b) Scaled addition.

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

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

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

- **Scaled Addition**. Consider a two-input multiplexer, shown in Figure 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

$$y = P(Y = 1)$$
$$= P(S = 1)P(X_1 = 1) + P(S = 0)P(X_2 = 1)$$
$$= sx_1 + (1 - s)x_2.$$

  (Note that throughout the paper, multiplication and addition represent *arithmetic* operations, not Boolean AND and OR.) 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*. This is, of course, an abstraction of what is happening at the bit level. 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,

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

where N_1 and N_0 are the average number of ones and zeros. Therefore, by showing how each logic gate manipulates the average number of ones and zeros, the operation of the logic gate can be expressed independently of any particular number format.

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,

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

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,

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

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,

$$N_{C1} = N_{A1}N_{B1}$$
$$N_{C0} = N_{A1}N_{B0} + N_{A0}N_{B1} + N_{A0}N_{B0} \qquad (4)$$

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:

$$N_{C1}\{1\} + N_{C0}\{0\} =$$
$$(N_{A1}\{1\} + N_{A0}\{0\}) \wedge (N_{B1}\{1\} + N_{B0}\{0\}) = \qquad (5)$$
$$N_{A1}N_{B1}\{1\} + (N_{A1}N_{B0} + N_{A0}N_{B1} + N_{A0}N_{B0})\{0\}$$

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,

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

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. (Of course, if the bit streams are correlated, the output does not simply depend on the proportions of the bit streams in a straightforward way.) 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,

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

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,

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

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

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

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,

$$11100$$
$$01100 \longrightarrow$$

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.

$$\begin{array}{r} 10 \\ \times 10 \\ \hline 00 \\ 10 \end{array} \left.\begin{array}{r} \\ \\ \end{array}\right\} \xrightarrow{\text{sum}} 100 \qquad \begin{array}{r} 10 \\ \times 10 \\ \hline 00 \\ 10 \end{array} \left.\begin{array}{r} \\ \\ \end{array}\right\} \xrightarrow{\text{concatenate}} 1000$$

Binary *Uniform*

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. Looking back at Example 2, we can think of the bipolar format as containing positive entities ('1's) and negative entities ('0's). Multiplication of entities that can be both positive and negative is defined by the following truth tables:

TABLES 1 & 2
Truth tables for multiplication of positive (1) and negative (0) entities

a	b	$a \times b$	a	b	$a \equiv b$
-	-	+	0	0	1
-	+	-	0	1	0
+	-	-	1	0	0
+	+	+	1	1	1

where the truth table on the right is identical to the truth table implemented by an XNOR gate. Therefore, by using the logic gate that multiplies the format of the entities, the average bit streams are convolved. This is why, in particular, multiplication and scaled addition are extremely simple operations with stochastic logic.

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?
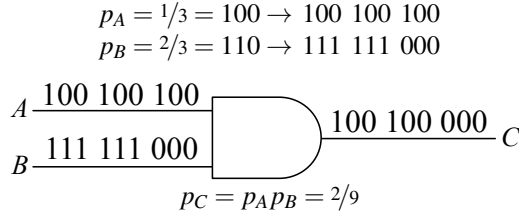
IV. DETERMINISTIC INTERPRETATION

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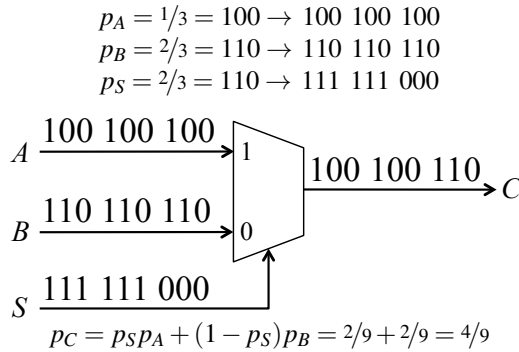
Example 5 Two registers contain deterministic unipolar proportions $p_A = 1/3$ and $p_B = 2/3$. How can we generate bit streams such that a single AND gate performs multiplication?

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:

$$p_A = 1/3 = 100 \rightarrow 100\ 100\ 100$$
$$p_B = 2/3 = 110 \rightarrow 111\ 111\ 000$$

$$A \frac{100\ 100\ 100}{}$$
$$B \frac{111\ 111\ 000}{}$$
$$100\ 100\ 000\ C$$
$$p_C = p_A p_B = 2/9$$

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:

$$p_A = 1/3 = 100 \rightarrow 100\ 100\ 100$$
$$p_B = 2/3 = 110 \rightarrow 110\ 110\ 110$$
$$p_S = 2/3 = 110 \rightarrow 111\ 111\ 000$$

$$A \xrightarrow{100\ 100\ 100} 1$$
$$B \xrightarrow{110\ 110\ 110} 0$$
$$S \xrightarrow{111\ 111\ 000}$$
$$100\ 100\ 110\ C$$
$$p_C = p_S p_A + (1 - p_S) p_B = 2/9 + 2/9 = 4/9$$

In these examples, Equation 6 is maintained on deterministic bit streams. (For convenience, we will use "independent" to describe both random and deterministic bit streams that obey Equation 6.)

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 [7].

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

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} [7]. 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 in order 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 Figure 4. Each converter module uses the general circuit topology of Figure 5. The modules are similar to the "Randomizer Unit"; the difference is that the LFSR is replaced by a deterministic number source. The generator takes in operands and generates bit streams such that:

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

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.
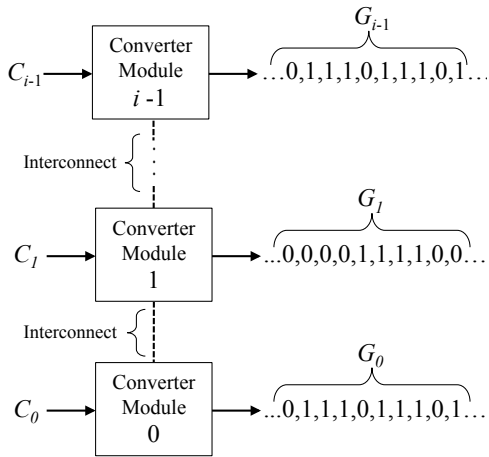
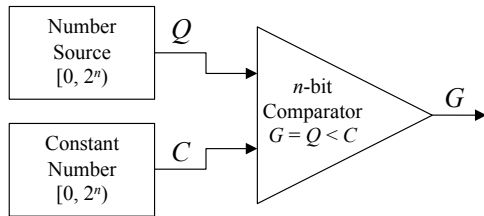**Fig. 4:** Deterministic bit stream generator

Fig. 5: Converter module

Conceptually, we can view the converter module circuit of Figure 5 as selecting bits from a collection or array a (see Figure 6). The binary constant C determines the contents of the array: any elements less than index C contain a one, otherwise they contain a zero. In this way, C defines a uniform proportion of ones and zeros: $C \rightarrow C\{1\} + (2^n - C)\{0\}$. Using the array analogy, we can represent the proportion as a sequence $a_0, a_1, \ldots a_{2^n-1}$. The number Q, determined by the deterministic number source, points to different indices of the array. Each clock cycle, the element that Q points to is chosen as the next output. To ensure the generated bit stream has the same proportion of ones and zeros (i.e., represents the number C), Q must point to each index an equal number of times (within some period). In other words, the input C defines the proportion or collection of bits from which the bit stream is *uniformly* generated. The sequence generated by the number source is used to maintain independence between bit streams. This requires the converter modules to have certain properties *relative* to one another. Depending on the method used, these properties manifest into either interconnections between modules or as differences in certain parameters.
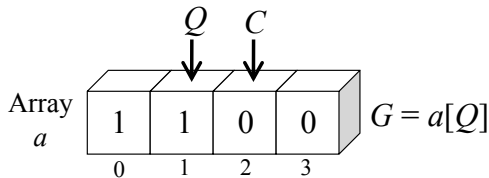

Fig. 6: Analogy of the circuit operation of Figure 5

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.

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). Figure 7 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.
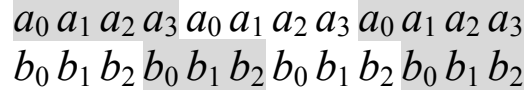

Fig. 7: Two bit streams generated by the relatively prime method

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. Figure 8 demonstrates this with a two level circuit.
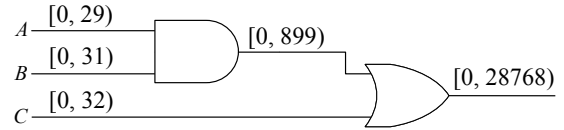

Fig. 8: Arbitrary multi-level circuit with streams generated by the relatively prime method

Therefore, by using relatively prime bit lengths up front, we can guarantee that Equation 6 is maintained for subsequent levels. This allows for the same arithmetic logic as a stochastic representation.

A circuit implementation of the relatively prime method is shown in Figure 9. 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.

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 frequencies. An analog implementation can also benefit from a simplified clock distribution circuit [8].
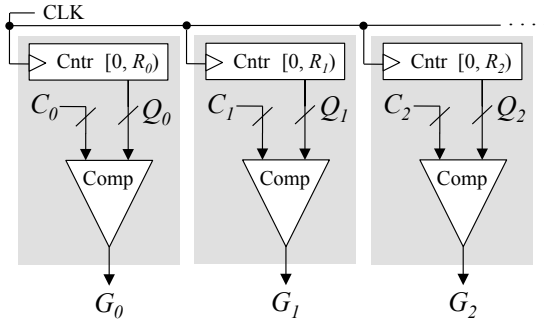
Fig. 9: Circuit implementation of the relatively prime method

Fig. 12: Circuit implementation of the rotation method

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). Figure 10 demonstrates this method with two bit streams, both with proportions of length four.
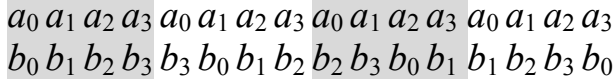
$$a_0\, a_1\, a_2\, a_3\ a_0\, a_1\, a_2\, a_3\ a_0\, a_1\, a_2\, a_3\ a_0\, a_1\, a_2\, a_3$$
$$b_0\, b_1\, b_2\, b_3\ b_3\, b_0\, b_1\, b_2\ b_2\, b_3\, b_0\, b_1\ b_1\, b_2\, b_3\, b_0$$

Fig. 10: Two 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. Assuming all proportions have the same length, we can extend the two bit stream example to work with multiple bit streams by inhibiting counters at powers of the operand length. This allows the operands to rotate relative to longer bit streams. For example, consider the circuit in Figure 11. 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.
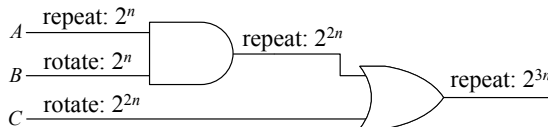

Fig. 11: Arbitrary multi-level circuit with bit streams generated by the rotation method

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 modules. Using n-bit binary counters and comparators, the circuit requires i n-bit comparators and $2i - 1$ n-bit counters.
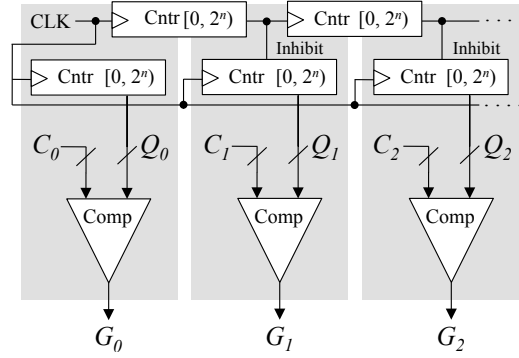
The advantage of using rotation as a method for generating independent bit streams is that we can use operands with the same resolution, but this requires more basic components than the relatively prime method.

C. Clock Division

The clock division method works by clock dividing operands. Similar to the rotation method, it also allows proportions to have arbitrary lengths. This method was first seen in Example 5. Figure 13 demonstrates this method with two bit streams, both with proportions of length four. Bit stream B is clock divided by the length of bit stream A's proportion.

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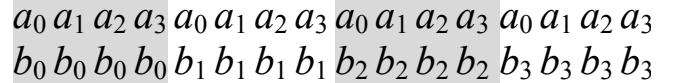$$b_0\, b_0\, b_0\, b_0\ b_1\, b_1\, b_1\, b_1\ b_2\, b_2\, b_2\, b_2\ b_3\, b_3\, b_3\, b_3$$

Fig. 13: Two 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 counter of every ith converter module increments every 2^{ni} clock cycles. This can be implemented in circuit form by simply chaining the converter module counters together, as shown in Figure 14. 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.
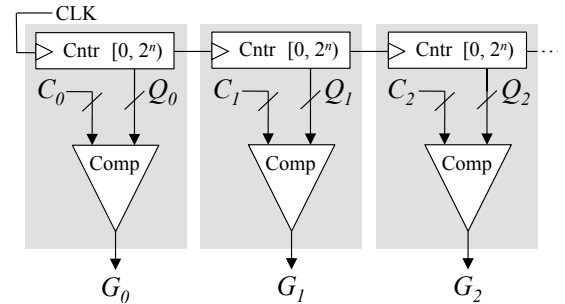

Fig. 14: Circuit implementation of the clock division method

VI. Experiments

In this section we compare the hardware complexity and latency of the deterministic methods with conventional stochastic methods. We also compare implementations of Bernstein polynomials, a general method for synthesizing arbitrary functions, for both precise and approximate computations.

A. Generator Comparison

Perfectly precise computations require the output resolution to be at least equal to the product of the independent 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, a stochastic representation requires bit streams that are 2^{2n} bits long to represent a value with $1/2^n$ precision. Also, to ensure the generated bit streams are sufficiently random and independent, each LFSR must have at least as many states as the required output bit stream. Therefore, to compute with perfect precision each LFSR must have at least length $2n_{in}i$. In this way, the precision of the computation is determined by LFSR length.

With the deterministic methods, the resolution n of each input i is determined by the length of its converter module counter. The output resolution is simply the product of the counter ranges. For example, with the clock division method, each converter module counter is connected in series. With i inputs each with resolution n, the series connection forms a large counter 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 stochastic methods.

To compare the area of the circuits in terms of gates, we assume three gates for every cell of a comparator and six gates for each flip-flop of a counter or LFSR (this is similar to the hardware complexity used in [9] in terms of fanin-two NAND gates). For i inputs with n-bit binary resolution, the gate count for each basic component is given by:

TABLE III
Gate count for basic components

Component	Gate Count
Comparator	$3n$
Counter	$6n$
LFSR	$12ni$

Using the basic component totals for each deterministic method from Section V and the fact that each "Randomizer Unit" needs one comparator and one LFSR per input, the total gate count and bit stream length for precise computations in terms of independent inputs i with resolution n is given by Table IV.

TABLE IV
Gate count and latency of stochastic and deterministic bit stream generators

Representation	Method	Gate Count	Latency
Stochastic	Randomizer	$12ni^2 + 3ni$	2^{2ni}
Deterministic	Rel. Prime Rotation	$9ni$ $15ni - 6n$	2^{ni}
	Clock Div.	$9ni$	

The equations of Table IV show that the deterministic methods use less area and compute to the same precision in exponentially less time. In addition, because the computations do not suffer from correlation, they are completely accurate. Table V compares the gate count and latency of the conventional stochastic method with the clock division method using numerical values of i and n.

TABLE V
Numerical comparison of Randomizer and clock division stream generators

i	n	Randomizer		Clock Div.		$\frac{determ.product}{stoch.product}$
		Gates	Latency	Gates	Latency	
2	4-bit	216	2^{16}	72	2^8	1.30×10^{-3}
	8-bit	432	2^{32}	144	2^{16}	5.09×10^{-6}
3	4-bit	468	2^{24}	108	2^{12}	5.63×10^{-5}
	8-bit	936	2^{48}	216	2^{24}	1.38×10^{-8}
4	4-bit	816	2^{32}	144	2^{16}	2.69×10^{-6}
	8-bit	1632	2^{64}	288	2^{32}	4.11×10^{-11}
5	4-bit	1260	2^{40}	180	2^{20}	1.36×10^{-7}
	8-bit	2520	2^{80}	360	2^{40}	1.30×10^{-13}

For the given number of inputs i and resolution n, the clock divide method has a 66-85% reduction in area and an exponential decrease in computation time. The deterministic area-delay product is orders of magnitude smaller than the stochastic area-delay product.

B. Bernstein Polynomial Implementation

In this subsection we compare the implementation of Bernstein polynomials using stochastic and deterministic methods. A Bernstein polynomial can be used to synthesize power-form polynomial functions or approximate non-polynomial functions that map values from the unit interval to values in the unit interval, examples include $g(x) = 6x^3 - 8x^2 + 3x$ and $f(x) = x^{0.45}$. This arithmetic circuit can be implemented using an adder block and multiplexer block, as shown in Figure 15. Bit streams z_i form the coefficients of the Bernstein polynomial and bit streams x_i form the input x. Additional details can be found in [10].
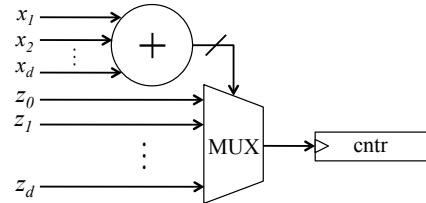
Fig. 15: Circuit implementation of Bernstein polynomial with output counter

For both representations, the adder block is formed by a d-bit adder made up of $2d$ gates and a $(d+1)$-channel multiplexer made with $3d$ gates. Each method needs the same number of bit streams and therefore requires $2d+1$ comparators. Each z bit stream must be independent from the input x bit streams, but they do not have to be independent from each other. This is because only one z bit stream is selected at a time. Therefore, Equation 6 does not have to be maintained between the coefficient bit streams and the same number source can be used for all z bit streams. Both methods require $d+1$ number sources.

Using the above basic component counts and including the output counter, Table VI characterizes the hardware complexity and latency for implementing a Bernstein polynomial of degree d with n-bit binary input resolution.

Table VII compares the clock division method to the stochastic method with numerical values of degree d and binary input resolution n.

TABLE VII
Numerical comparison of Randomizer and clock division implementation of Bernstein polynomials with degree d and input resolution n

d	n	Randomizer		Clock Div.		$\frac{determ.product}{stoch.product}$
		Gates	Latency	Gates	Latency	
2	4-bit	646	2^{24}	214	2^{12}	8.09×10^{-5}
3	4-bit	1059	2^{32}	291	2^{16}	4.19×10^{-6}
4	4-bit	1568	2^{40}	368	2^{20}	2.24×10^{-7}
5	4-bit	2173	2^{48}	445	2^{24}	1.22×10^{-8}

With 4-bit binary inputs, the clock divide method provides 66-79% reduction in circuit area over the given range of polynomial degrees. Again, the latency of the deterministic representation is exponentially less than the stochastic representation. The area-delay product of the clock divide method for computing Bernstein polynomials is orders of magnitude less than the stochastic method.

If a lower output resolution is desired to reduce the bit stream lengths, the resolution of the inputs can be relaxed:

$$n_{in} = \lceil \frac{n_{out}}{i} \rceil = \lceil \frac{n_{out}}{d+1} \rceil \qquad (10)$$

In general, stochastic computation uses imprecise output resolutions to avoid long delays and reduce the size of the LFSRs. By keeping the output resolution fixed, the LFSR lengths are linearly proportional to d.

Using an n_{out}-bit output resolution and relaxing the inputs, the hardware complexity for the stochastic method with a constant output resolution is given by $12n_{out}d + 24n_{out} + 6\lceil \frac{n_{out}}{d+1} \rceil d + 3\lceil \frac{n_{out}}{d+1} \rceil + 5d$. Table VIII compares the methods with constant output resolution (where the inputs of the deterministic methods are relaxed according to Equation 10).

TABLE VIII
Numerical comparison of Randomizer and clock division implementation of Bernstein polynomials with degree d and constant output resolution n_{out}

d	n_{out}	Randomizer		Clock Div.		$\frac{determ.product}{stoch.product}$
		Gates	Latency	Gates	Latency	
3	8-bit	537	2^{16}	153	2^{8}	1.11×10^{-3}
4	10-bit	794	2^{20}	194	2^{10}	2.39×10^{-4}
5	12-bit	1099	2^{24}	235	2^{12}	5.22×10^{-5}
6	14-bit	1452	2^{28}	276	2^{14}	1.16×10^{-5}
7	16-bit	1853	2^{32}	317	2^{16}	2.61×10^{-6}

The results of Table VIII show the clock divide method provides a 71 to 82% reduction in area for practical implementations of a Bernstein polynomial with constant output resolution.

VII. CONCLUSION

There has been widespread interest in the idea of stochastic logic in recent years. We point to [5] for a survey of work in the area. While numerous papers have advocated the advantages, the narrative has never been compelling. Yes, the paradigm permits complex arithmetic operations to be performed with remarkably simple logic, but the logic to generate pseudorandom 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 stochastic computation 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 generate pseudorandom bit streams. More importantly, we can use much shorter bit streams to achieve the same accuracy as with statistical sampling through randomness. Indeed, the results of our computation are predictable and completely accurate for all input values.

Of course, compared to a binary radix representation, our deterministic representation is still not very compact. With M bits, a binary radix representation can represent 2^M distinct numbers. To represent real numbers with a resolution of 2^{-M}, i.e., numbers of the form $\frac{a}{2^M}$ for integers a between 0 and 2^M, we require a stream of 2^M bits. In contrast, a stochastic representation requires 2^{2M} bits to achieve the same precision!

We conclude that there is no clear reason to compute on stochastic bit streams. Even when randomness is free, say harvested from thermal noise or some other physical source, stochastic computing entails very high latency. In contrast, computation on deterministic uniform bit streams is less costly, has much lower latency, and is completely accurate.

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," *Proceedings of the Design Automation Conference*, 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] W. Qian, "Digital yet deliberately random: Synthesizing logical computation on stochastic bit streams," Ph.D. dissertation, University of Minnesota, 2011.

[8] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan, "Polysynchronous stochastic circuits," *IEEE/ACM Asia and South Pacific Design Automation Conference*, 2016.

[9] P. Li, W. Qian, and D. J. Lilja, "A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic," *IEEE 30th International Conference on Computer Design (ICCD)*, 2012.

[10] W. Qian and M. D. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," *ACM Design Automation Conference*, 2008.