A Scalable, Deterministic Approach to Stochastic Computing

Yadu Kiran
kiran013@umn.edu
University of Minnesota
Twin Cities, Minnesota, USA

Marc Riedel
mriedel@umn.edu
University of Minnesota
Twin Cities, Minnesota, USA

ABSTRACT

Stochastic computing is a paradigm in which logical operations are performed on randomly generated bit streams. Complex arithmetic operations can be performed by simple logic circuits, with a much smaller area footprint than conventional binary counterparts. However, the random or pseudorandom sources required to generate the bit streams are costly in terms of area and offset the gains. Also, due to randomness, the computation is not precise, which limits the applicability of the paradigm. Most importantly, to achieve reasonable accuracy, high latency is necessitated. Recently, deterministic approaches to stochastic computing have been proposed. They demonstrated that randomness is *not* a requirement. By structuring the computation deterministically, the result is exact and the latency is greatly reduced. However, despite being an improvement over conventional stochastic techniques, the latency increases quadratically with each level of logic. Beyond a few levels of logic, it becomes unmanageable. In this paper, we present a method for *approximating* the results of their deterministic method, with latency that only increases linearly with each level. The improvement comes at the cost of additional logic, but we demonstrate that the increase in area scales with $\sqrt{n}$, where n is the equivalent number of binary bits of precision. The new approach is general, efficient, composable, and applicable to all arithmetic operations performed with stochastic logic.

CCS CONCEPTS

• **Hardware → Analysis and design of emerging devices and systems**.

KEYWORDS

Stocastic Computing, Bitstreams, Unary, Thermometer Encoding, Deterministic Stochastic Computing, Sobol Sequences, Clock-division, Fixed-length

ACM Reference Format:
Yadu Kiran and Marc Riedel. 2022. A Scalable, Deterministic Approach to Stochastic Computing. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22), June 6–8, 2022, Irvine, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3526241.3530344

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GLSVLSI '22, June 6–8, 2022, Irvine, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9322-5/22/06…$15.00
https://doi.org/10.1145/3526241.3530344

1 INTRODUCTION

In stochastic computing, randomly generated streams of 0's and 1's are used to represent fractional numbers. The number represented by a bit stream corresponds to the probability of observing a 1 in the bit-stream at any given point in time. The benefit of this representation is that complex operations can be performed with simple logic. For instance, multiplication can be performed with a single AND gate and scaled addition can be performed with a single multiplexer. The drawbacks of the conventional stochastic model are 1) the latency is high, and; 2) due to randomness, the accuracy is low. Latency and accuracy are related parameters: to achieve acceptable accuracy, high latency is required [1].

Recently, a "deterministic" approach to stochastic computing has been proposed [5], that uses all the same structures as stochastic logic, but on deterministically generated bit streams. Deterministic approaches incur lower area costs since they generate bit streams with counters instead of expensive pseudo-random sources such as linear feedback shift registers (LFSRs). Most importantly, the latency is reduced by a factor of approximately $\frac{1}{2^n}$, where n is the equivalent number of bits of precision. However, the latency is still an issue, as it increases quadratically for each level of logic. Any operation involving two 2^n-bit input bit streams will produce a resulting bit stream of length 2^{2n} bits. This is a mathematical requirement: for an operation such as multiplication, the range of values of the product scales with the range of values of the operands. However, most computing systems operate on constant precision operands and products. Since this is not sufficient to represent the 2^{2n} output in full precision, we will have approximation errors. Our primary goal is to minimize this error.

Recent papers have discussed techniques for approximating the deterministic computation with quasirandom bit streams, such as Sobol sequences [2, 7, 8, 15]. Unfortunately, the area cost of these implementation is high: the logic to generate the quasirandom bit steams is complex and grows quickly as the number of bit streams increases, in most cases, completely offsetting the benefits.

In this paper, we present a scalable deterministic approach that maintains constant bit stream lengths, and so approximates the results, but with much lower area cost than the quasirandom sequence approach. We structure the computation by *directly* pairing up corresponding bits from the input bit streams, using only simple structures such as counters. Not only does our approach achieve a high degree of accuracy for the given bits of precision, but also maintains the length of bit streams. This property lends *Composability* to our technique, allowing multiple operations to be chained together. Maintaining constant bit stream length comes at the cost of additional logic, but we demonstrate the increase in area scales with $\sqrt{n}$, where n is the number of binary bits of precision. The new approach is general and efficient, applicable to all arithmetic operations performed with stochastic logic. We validate our results on a variety of benchmark circuits.

This paper is structured as follows: Section 2 provides a brief overview and background of stochastic computing. Section 3, and Section 4 presents our new approach. Section 5 evaluates our method on benchmarks, comparing and contrasting it with prior methods. Finally, Section 6 outlines the implications of this work.

2 BACKGROUND INFORMATION

2.1 Introduction to Stochastic Computation

The paradigm of stochastic logic (sometimes called stochastic "computing") operates on non-positional representations of numbers [3]. Bit streams represent fractional numbers: 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 $t = 1, 2, ..., L$. If the bit stream is randomized, then for precision equivalent to conventional binary with precision n, the length of the bit stream L must be 2^{2n}[11]. The probability that each bit in the stream is 1 is denoted by $P(X = 1) = x$. Fig. 1 illustrates how the value $\frac{5}{8}$ can be represented with bit streams. Note that the representation is not unique, as demonstrated by the four possibilities in the figure. In general, with a stochastic representation, the position of the 1's and 0's do not matter. Since we will be dealing with non-randomly generated bit streams, we will refer to this representation as *unary* rather than stochastic.

$$\frac{5}{8} \Rightarrow \begin{array}{c} 1\,0\,1\,1\,0\,1\,0\,1 \\ 0\,1\,0\,1\,1\,1\,0\,1 \\ 1\,1\,0\,1\,0\,0\,1\,1 \\ 0\,0\,1\,1\,1\,1\,0\,1 \end{array}$$

Figure 1: Representation of values with stochastic logic.

Common arithmetic operations that operate on probabilities can be mapped efficiently to logical operations on unary bit-streams.

- **Multiplication**. Consider a two-input AND gate whose inputs are two independent bit streams $X_1(t)$ and $X_2(t)$, as shown in Fig. 2(a). The output bit stream Y, is given by
$$\begin{aligned} y &= P(Y = 1) = P(X_1 = 1 \text{ and } X_2 = 1) \\ &= P(X_1 = 1)P(X_2 = 1) = x_1 x_2. \end{aligned}$$
- **Scaled Addition**. Consider a two-input multiplexer whose inputs are two independent stochastic bit streams X_1 and X_2, and its selecting input is a stochastic bit stream S, as shown in Fig. 2(b). The output bit stream Y, is given by
$$\begin{aligned} y &= P(Y = 1) \\ &= P(S = 1)P(X_1 = 1) + P(S = 0)P(X_2 = 1) \\ &= sx_1 + (1 - s)x_2. \end{aligned}$$

Complex functions such as exponentiation, absolute value, square roots, and hyperbolic tangent can each be computed with a small number of gates [9, 18].

2.2 The Deterministic Approach to Stochastic Computing

In conventional stochastic logic, the bit streams are generated from a random source such as a linear feedback shift register (LFSR). The

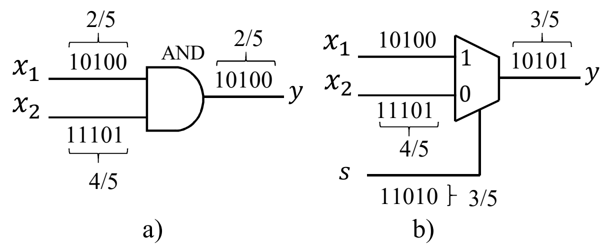

Figure 2: Stochastic implementation of common arithmetic operations: (a) Multiplication; (b) Scaled addition.

computations performed on these randomly generated bit streams are not always accurate. Fig. 3 demonstrates a worst case scenario where multiplying two input bit-streams corresponding to probabilities $\frac{3}{5}$ and $\frac{2}{5}$, results in an output of probability $\frac{0}{5}$.

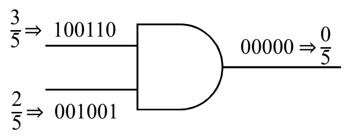
$\frac{3}{5} \Rightarrow$ 100110
$\frac{2}{5} \Rightarrow$ 001001
00000 $\Rightarrow \frac{0}{5}$

Figure 3: Errors with conventional stochastic logic.

Consider instead a *thermometer encoding*. This is just a unary encoding, but one in which all the 1's appear consecutively at the start, followed by all the 0's (or vice-versa), as shown in Fig. 4.

$$\frac{3}{4} \Rightarrow 1110 \quad \frac{5}{8} \Rightarrow 11111000$$

Figure 4: Thermometer Encoding

This encoding is not a requirement, but rather a consequence of the circuit used to generate deterministic bit streams, shown in Fig. 5. For a computation involving n-bit precision operands, the setup involves an n-bit register, counter and comparator. The register stores the corresponding binary value of the input operand. The bit stream is generated by comparing the value of the counter to the value stored in the register. The counter runs from 0 to $2^n - 1$ sequentially, so the resulting bit-stream inherits a thermometer encoding.

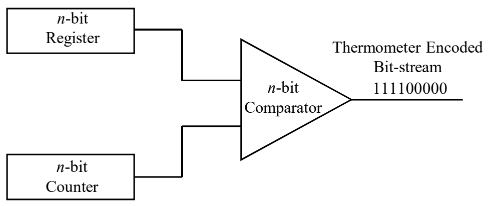

Figure 5: Thermometer Code Generator

Recently, a "deterministic" approach to stochastic computation was proposed, where the computation is performed on bit-streams which are generated deterministically based on a thermometer encoding [5]. By deterministically generating bit streams, all stochastic operations can be implemented efficiently based on the following scheme: *every bit of one operand must be matched up against every bit of the other operand(s) exactly once.*

Performing a multiply operation on unary bit-streams using the deterministic approach involves convolving one input operand with another, as illustrated in Fig 6. Holding a bit of one input operand constant, the operation is repeated for each of the bits of the other input operand. The particular approach is known as *clock-division*, due to the division of the clock signal in the circuit for generating the input bit streams.

$$A = a_0a_1a_2a_3 \qquad B = b_0b_1b_2b_3$$

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

Figure 6: Multiplication using Clock-Division

Fig. 7 illustrates the Multiply operation on two operands ($\frac{3}{4}$ and $\frac{1}{4}$) performed stochastically and deterministically. It is evident that the deterministic method achieves perfect accuracy. However, for each level of logic, the bit stream lengths increase. For a multiply operation involving two streams of 2^n bits each, the output bit stream is 2^{2n} bits. This is a mathematical requirement in order to represent the full range of values. However, for large values of n, the bit stream lengths become prohibitive. For most applications, one has to maintain a constant bit stream length across all the levels of logic, and hence, approximation is inevitable [4]. We discuss how to do this in Section 3.

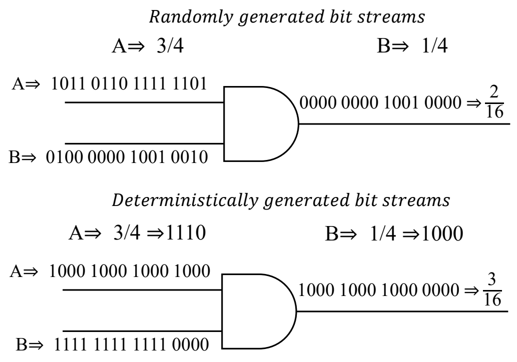

Figure 7: Multiplication - Conventional Stochastic vs Deterministic

For an operation such as multiplication, two copies of the circuit in Fig. 5 are used for generating the bit streams of the input operands. As shown in Fig. 8, the counter of the second input operand counts up only when the counter of the first input operand rolls over $2^n - 1$. This can be achieved by connecting the AND of all the output lines of the first counter to the clock input of the second counter.

3 SCALABLE DETERMINISTIC APPROACH

In the deterministic approach discussed in Section 2.2, the bit stream lengths grow quadratically with each level of logic [5]. However, this drawback becomes unsustainable for larger circuits. Our goal is to keep the length constant across multiple levels of logic.

3.1 Downscaling

The low-hanging fruit for approximating is to simply *downscale* the input operands, i.e, generate bit streams of smaller length as

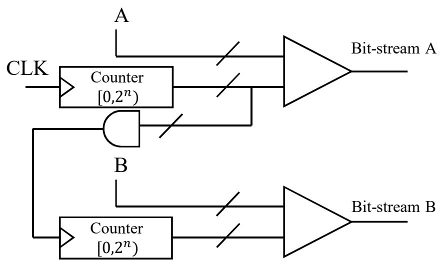

Figure 8: Circuit implementation of the clock division method.

shown in Fig. 9. Consider a input operand that would correspond to a bit stream of length L. We want to reduce the length of the bit stream generated, by downscaling or approximating the input operand itself. Down-scaling is ideally performed by reducing the bit-stream by powers of 2, i.e, divide L by $d = 2^i$, where d is the degree of downscaling. In other words, every set of d bits in the original bit stream would correspond to one bit in the downscaled bit stream. The deterministic multiplication operation restores the target length.

Downscaling is easily achieved by right-shifting the value stored in the register in Fig. 5. For example, for an input operand with $2^4 = 16$ bits of precision and probability value $\frac{12}{16}$, we would store the binary equivalent of 12, i.e 1100_2, in the register. To downscale the value by a factor of 4, we would right shift the value of the register by 2 bits to obtain the binary value 11_2 (that corresponds to the probability value $\frac{3}{4}$). In general, to downscale a value by a factor of $d = 2^i$, we would right shift by i bits. Consequently, this would also reduce the size of the counters used for bit generation.

In Fig. 7, we showed that deterministically multiplying two input bit-streams of length 2^n bits each, results in an output bit stream of length 2^{2n}. But if we were to approximate the input operands to bit streams of length $2^{\frac{n}{2}}$, then our output bit-stream would be limited to 2^n bits. If the target value of a bit-stream can be accurately represented with fewer bits then there will be no errors. For example, the probability $\frac{20}{32}$ can also be represented as $\frac{10}{16}$ or $\frac{5}{8}$. However, in general, the process of downscaling will introduce errors. We want to minimize the error. In a mathematical sense, we want a scheme that always generates the *optimal approximation*.

In the context of this paper, the *error* is the difference between the result and the optimal approximation, given a target bit stream length. For example, the probability $\frac{11}{16}$, when downscaled to 4 bits, can be optimally approximated as $\frac{3}{4}$ (but not as $\frac{1}{4}$, $\frac{2}{4}$, or $\frac{4}{4}$).

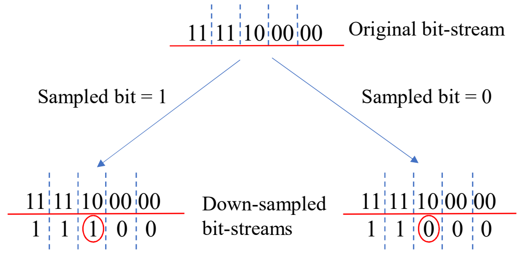

Figure 9: Downscaling - Two Possible Cases

When downscaling a thermometer encoding, there are only two possible scenarios that can occur, irrespective of the length of the input bit-streams. These are illustrated in Fig. 9 where we try to approximate $\frac{5}{10}$ to be represented with just 5 bits. In both cases, a single bit conveys the wrong information. Using either one of the down-scaled bit-streams as an input to an arithmetic operation results in an error. The method that we will present in this paper always opts for the right-hand side case in Fig. 9, where the down-scaled bit stream is an under-approximation of the actual value. The reasoning behind this will be evident in Section 3.3.

For an operation involving two downscaled input operands of 2^n bits each, it can be mathematically deduced that the error that can occur in the output bit stream is at most $(2^n - 1)$ bits out of 2^{2n} bits. It's worse than it appears as it grows as a function of the bit-stream length of the inputs, as well as the number of logic levels. We can do better.

3.2 Error Compensation

The basic idea of our approach is to systematically compensate for the error that we introduce when down-scaling. We do so during the clock division process, shown in Fig. 6.

We illustrate with an example. Consider the multiply operation of two input operands, each of length 16 bits. To restrict the length of the output bit stream to just 16 bits, we will downscale the input operands that corresponds to a bit stream of 4 bits, a downscaling factor of $\frac{16}{4} = 4$. In general, if the input-operands are *p bits in length*, we down-scale them to length *q bits*, such that q=$\sqrt{p}$ and that the length of the output bit stream remains the same as the input bit steams. Consider the case where $A = \frac{5}{16}$ and $B = \frac{15}{16}$ as shown in Fig. 10. Neither of the two input operands can be downscaled to 4 bits without introducing errors. For each input operand, we round down, shifting the value stored in the register by 2 bits. So $A = \frac{5}{16}$ gets down-scaled to $A' = \frac{1}{4}$, which is equivalent to $\frac{4}{16}$. $B = \frac{15}{16}$ gets down-scaled to $B' = \frac{3}{4}$, which is equivalent to $\frac{12}{16}$. We underestimate the value of A' by $\frac{1}{16}$, and B' by $\frac{3}{16}$.

Fig. 9 shows that only one bit in a downscaled bit-stream(s) is erroneous. And this erroneous bit is carrying *partially incorrect* information. In our example shown in Fig. 10, for A', we can interpret the second bit which is highlighted in blue, as having $1/4$th of its information "incorrect". Likewise, for B', $3/4$th of its last bit (highlighted in orange) can be considered "incorrect" information.

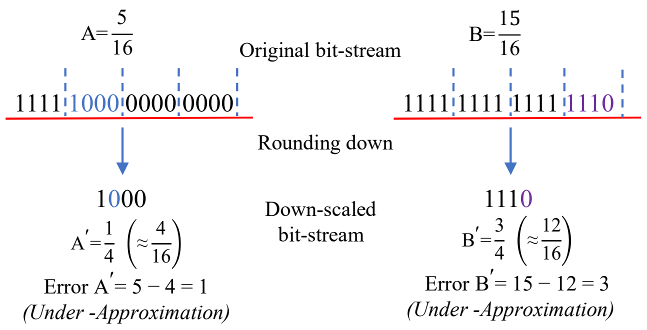

Figure 10: Example – Error Compensation

When performing the clock division operation which was discussed 2.2 in, each bit is repeated multiple times (in this example, four times). This is shown in Fig. 11. This provides the opportunity to compensate for the error incurred during downscaling. For A', we know that the second bit is erroneous, and that 1/4th of this bit is "incorrect". This bit is also repeated four times during the clock division operation. So our instinct would be to "correct" this error by inverting that bit once, out of the four times it is repeated. Similarly, for B', we know that 3/4th of its last bit is "incorrect". Naturally, we would want to invert this bit three out of the four times it is repeated.

We mentioned earlier in Section 3.1, that out of the two possible cases when downscaling (over-approximation and under-approximation), we would always under-approximate the value. By restricting ourselves to this case, we know that the erroneous bit in our downscaled bit-stream is always the first 0 we encounter in our thermometer encoded bit-stream; and to compensate for this error, we would always have to invert this 0 to 1, a certain number of times during our clock division operation.

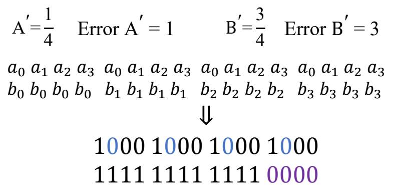
$A'=\frac{1}{4}$ Error $A' = 1$
$B'=\frac{3}{4}$ Error $B' = 3$
$a_0\,a_1\,a_2\,a_3\;a_0\,a_1\,a_2\,a_3\;a_0\,a_1\,a_2\,a_3\;a_0\,a_1\,a_2\,a_3$
$b_0\,b_0\,b_0\,b_0\;b_1\,b_1\,b_1\,b_1\;b_2\,b_2\,b_2\,b_2\;b_3\,b_3\,b_3\,b_3$
$\Downarrow$
1000 1000 1000 1000
1111 1111 1111 0000

Figure 11: Example – Candidate Bits to Invert

If the input-operands are *p bits in length*, we down-scale them to length *q bits*, such that q=$\sqrt{p}$. The down-scaled bit stream has an error of *e*, implying a portion $\frac{e}{q}$, of a 0, is incorrect. In the clock-division operation, each bit is repeated *q* times. To compensate for the error, we invert the 0 to 1, *e* out of the *q* times that it is repeated.

We know how many bits we need to invert, but we now face the challenge of determining which position of the bits to invert. The erroneous bit is repeated *q* times, and there are *q* candidate positions to perform the *e* (i.e Error magnitude) bit flips. It turns out that we can decide these positions in a deterministic fashion, by performing another multiply operation.

3.3 Multiplication within an Operation

The bit flips need to occur in the right proportion. In other words, each bit flip of the first operand should be distributed equally among all the bits of the second operand.

Take the example in Fig. 10. A' (the downscaled bit stream of A) has an error of 1, or in other words, one of the 0s should be flipped to 1 and this needs to be distributed among the bits of B'. Since B' represents $\frac{3}{4}$, it makes sense for that bit flip to align with a 1 in the bit stream for B'. On the same line B' has an error of 3, and needs to be distributed among the bits of A'. With A representing $\frac{1}{4}$, in order to distribute those bit flips uniformly, we would align only one of those bit flips with a 1 in the bit stream of A, and the remaining with 0s.

Trying to figure this distribution out off the top of one's head is easy, but we need a way to compute this deterministically using digital logic. We can do this with a multiply operation. In our example for A', we can compute $3 \times \frac{1}{4} = \frac{3}{4} \approx 1$. In fact, we can do so with another *stochastic multiply operation*.

In the example shown in Fig. 11, based on the error, we would need to invert one bit of A and three bits of B. Since we are always under-approximating our input operands (and consequently, the result), we will always be changing 0's to 1's. For a bit stream X, let $Error(X)$ be the number of bits we need to invert, and $Inv(X)$ be the number of inverted bits that need to align with a 1 from the other operand. The error compensation is illustrated below.

$$\begin{aligned}\text{Inv}(A') &= \text{Error}\,A' \times B' \\ &= 1 \times \frac{3}{4} = 1\end{aligned} \tag{1}$$

$$\begin{aligned}\text{Inv}(B') &= \text{Error}\,B' \times A' \\ &= 3 \times \frac{1}{4} = 1.\end{aligned} \tag{2}$$

Now that we know where to align those bit flips , we perform the multiply operation with error compensation (bit-flips) as shown below in Fig. 12.

```
1100 1000 1000 1000
1111 1111 1111 1000
-------------------
1100 1000 1000 1000
```

Figure 12: Inverting the Erroneous Bits

The result of this operation is an output bit-stream corresponding to the value $\frac{5}{16}$. This is our desired result, as $\frac{5}{16} \times \frac{15}{16} = \frac{4.6875}{16}$ which is optimally represented as $\frac{5}{16}$.

It is important to note that even with error compensation, it is still possible for our output bit-stream to not be an optimal approximation. This is because the multiply operations performed in Eq. (1) and Eq. (2) are carried out with the downscaled values of our original input operands A and B, and hence, there is an approximation involved. However, *the error is bounded to be* at most 2 bits*, regardless of the length of the input operands.* This is because, in Eq. (1) and Eq. (2), A' and B' can have an error of at most 1 bit (out of $2^{n/2}$ bits) from the original values of A and B, as shown in Figure 9. Consequently, the values obtained for Inv(A') and Inv(B') can also differ by at most 1 from their optimal values. Stated differently, when performing the inversion, the maximum error that can be introduced is two bits (one for A, and one for B). This would translate to a maximum error of only two bits at the output.

Initially, we set out with the goal to deterministically compute the multiplication of two 2^n length input bit streams. We then downscale them to $2^{n/2}$ length input bit streams, to produce an output bit-stream of length 2^n. This introduces errors in the resultant bit stream since we are dealing with approximations, and we want the optimal approximation for our result. This error can be deterministically quantified (and compensated) by two other multiply operations, which also involve $2^{n/2}$ bit-stream. These operations can happen in parallel. Therefore, to produce the desired output bit-stream of length 2^n bits, the latency is $2^n + 2^n = 2^{n+1}$. However, there is another optimization that can be implemented. The *two stages* of this operation, i.e determining the error and multiplication with error compensation, can be pipelined to maintain the throughput of one computation every 2^n bits.

The method we propose shares a lot of similarities with multiplication using partial products in the binary domain. We divide the bits of the operands (A and B) into higher-order (A_h and B_H), and lower-order (A_L and B_L) bits. The higher-order bits constitute the down-scaled input operands, while the lower order bits represent the error. The error is compensated by inverting bits, and where we invert those bits is determined by two multiplications: $A_L \times B_H$, and $B_L \times A_H$. We use these results to correct for the error in our main multiplication of our downscaled operands ($A_H \times B_H$). The one divergence is that we're not performing the multiplication of the lower-order bits, i.e $A_L \times B_L$. This aspect was initially part of our design, and in fact, eliminates the minute error (max bound of 2 bits) discussed earlier. But this minor improvement in accuracy is accompanied by a 30% increase in area cost. We believe that the trade-off is not worth it.

In our example, we have illustrated how to perform multiplication using 16-bit length streams, which conveniently has a square root. However, the proposed technique can still be applied to bit streams of all lengths that are powers of 2, with the caveat that we would need to sacrifice pipelining due to the difference in latencies of the two stages of the operation.

4 HARDWARE IMPLEMENTATION

The complete circuit for our method is shown in Fig. 13a and Fig. 13b. By downscaling the input length to the square root of its original value, the binary values of A and B can be partitioned in half, as shown in the figure. The higher-order bits represent our downscaled operands, while the lower-order bits represent the error.

Fig. 13a represents the first stage of our operation, responsible for computing Eq. (1) and Eq. (2). It employs two deterministic stochastic multiplier circuits, each with two unary bit stream generators. The generated bit streams are fed to an AND gate which performs the multiplication, and the result is accumulated using a counter.

The results from Fig. 13a are used in Fig. 13b, which carries out the second stage of the operation, i.e the main multiply operation. Fig. 13b features two unary bit stream generators for our downscaled input operands, which are then fed to an Error Compensation Module that performs the bit flips, and is then fed to a AND gate.

The Error Compensation Module consists of logic that computes the input to the selector line for two multiplexers: one that chooses between A and NOT(A), and the other between B and NOT(B). The outputs of these multipliexers serve as the final input to an AND. The output of the AND gate is accumulated into a n-bit counter and would be the final result of our multiply operation.

5 SIMULATION AND RESULTS

We evaluated our approach across relevant benchmarks. We compare it to prior stochastic implementations which rely on pseudorandom or quasi-random generation of bit-streams, such as LFSRs and Sobol sequences [8]. We can consider the Sobol sequence implementation to be representative of all approaches that rely on quasirandom sequences called low-discrepancy sequences, as they all showcase similar accuracy and area cost. We also compare our method against conventional binary implementation.

Table 1 shows the Mean Absolute Error (MAE) Percentage and Gate Cost of various implementations for the stochastic multiplication of two inputs. We set the area of the Sobol-Sequences approach

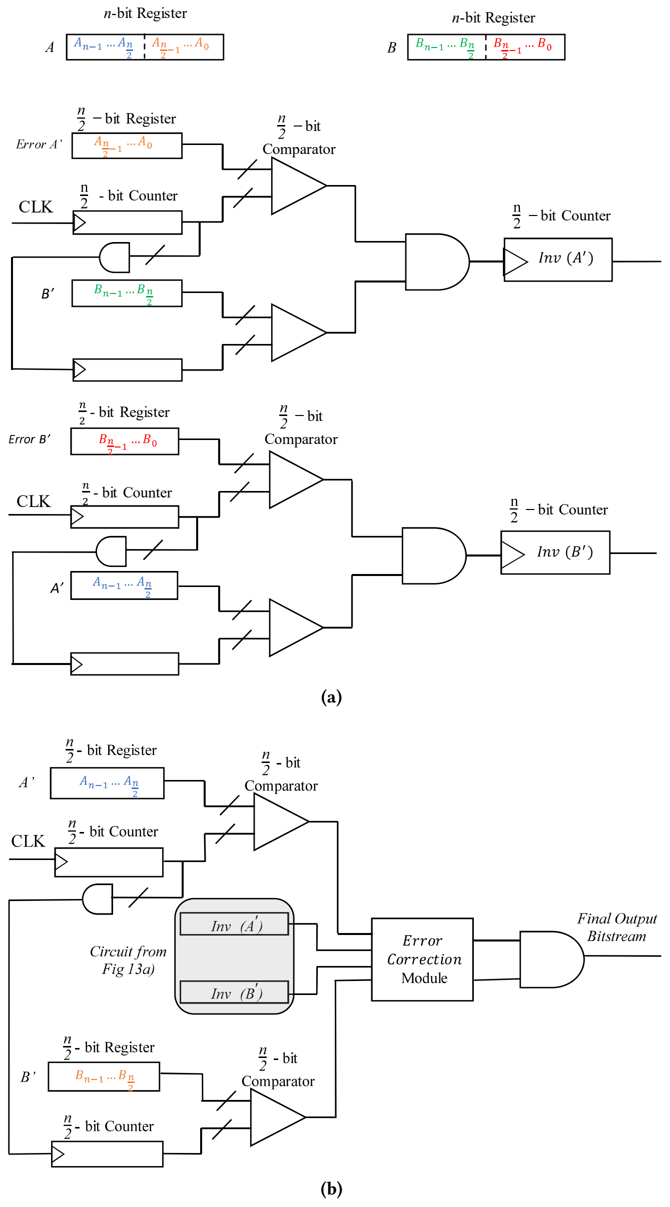

Figure 13: Circuit — Scalable Deterministic Approach: (a) Determining Error (b) Main Multiply Operation

as our reference for comparisons. The output bit streams were computed for *all possible values* of input operands of length 2^n. Note that the absolute error in this case, can never be 0. Mathematically, we would need to observe the output from 2^{2n} cycles to obtain no error at all.

Our approach offers significant improvements in accuracy over conventional stochastic implementations that use LFSRs, while trailing behind the Sobol sequence generator by only a small amount. However, when we examine the gate cost, shown in Fig. 14, we see that the increase in cost over LFSR implementations is minor. The minor improvement in accuracy provided by low-discrepancy

Table 1:
MAE and Gate Cost % for the multiply operation of various stochastic implementations.

Bitstream Length	LFSR		Sobol Sequence		Our Approach	
	MAE	Gate Cost	MAE	Gate Cost	MAE	Gate Cost
2^6	48.64%	53.29%	20.29%	100%	26.08%	87.95%
2^8	28.26%	47.28%	9.15%	100%	10.86%	66.2%
2^{10}	8.39%	43.05%	1.34%	100%	1.58%	57.73%

sequences such as the Sobol sequence is accompanied by a large increase in area cost. The gate cost for such implementations scale quadratically as the precision of the input operands increase, as evident in Fig. 14. This is due to the fact that such low-discrepancy sequences incorporate a Directional Vector Array in their circuit, whose gate cost scale by a factor of n^2 [7].

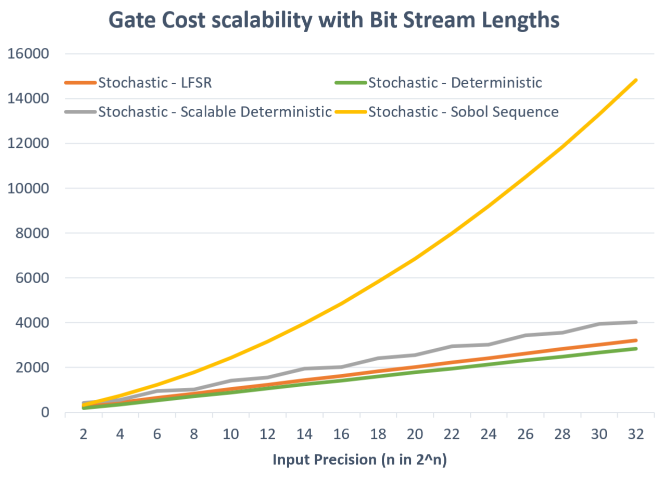

Figure 14: Relative gate cost for different implementations of a multiply circuit.

Although we only discussed Multiplication in our examples, the proposed method can be applied to many stochastic operations.[9, 18] demonstrates how to perform operations such as exponent, sin, log in the stochastic domain using NAND gates to implement the Maclaurin series expansion of these function. For these tests, we settled on bit-streams of length 2^8 bits, as it provides a good balance of accuracy, precision and latency. We do make some minor adjustments such that the coefficients in polynomial are approximated such that the denominator's precision is $\frac{1}{2^8}$. The increase in error due to this change is offset by increasing the degree of the polynomial, which translates to more levels of logic.

The Mean Absolute Error (MAE) and gate cost are shown in Table 2. The general trend continues; our technique offers almost identical accuracy as the state-of-the-art Sobol-Sequences, while offering significant reductions in area. In some cases, the gate cost of Sobol-Sequences is over twice our proposed circuit. And the gap only widens as we scale the length of the bit-streams.

Table 2:
MAE and Gate Cost % for functions implemented using Maclaurin Expansion.

Operation	LFSR		Sobol Sequence		Our Approach	
	MAE	Gate Cost	MAE	Gate Cost	MAE	Gate Cost
e^{-x}	33.2%	44.27%	12.3%	100%	14.20%	53.32%
$sin\ x$	31.3%	48.67%	13.1%	100%	12.9%	57.20%
$log(1+x)$	34.5%	45.31%	14.6%	100%	17.3%	48.02%
$sigmoid\ x$	32.7%	52.76%	10.9%	100%	12.7%	60.61%

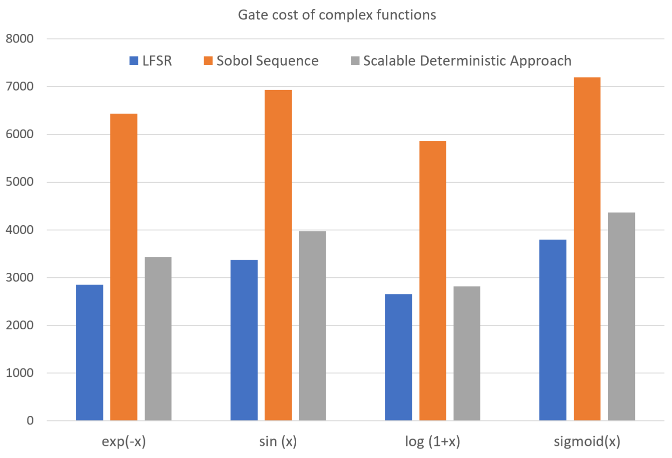

Figure 15: Relative gate cost for different implementations of a multiply circuit.

6 CONCLUSION

Recent work has demonstrated that randomness is not a requirement for "stochastic" computing. The deterministic approach in [5] mitigates most of the drawbacks typically associated with the paradigm, including the long latency. However, the method in these papers does not allow for graceful approximations when constant bit-stream lengths are required.

In this paper, we presented an approach that builds upon this foundation. By deterministically downscaling the inputs and compensating for approximation errors during the clock division operation, we demonstrate that it is possible to produce accurate results, while also preserving the bit stream lengths. This makes our approach *composable*, allowing operations to be chained together. Our simulations show that our approach can achieve very accurate results, with the maximum error bounded as two bits for each level of logic, irrespective of the bit stream length. The area cost compares favorably with conventional binary counterparts in many applications. It offers significant advantages over other stochastic approaches that rely on random or quasi-random bit streams.

REFERENCES

[1] Armin Alaghi and John P. Hayes. 2013. Survey of Stochastic Computing. *ACM Transactions on Embedded Computing*. 12, 2s, Article 92 (May 2013), 19 pages.

[2] Armin Alaghi and John P. Hayes. 2014. Fast and accurate computation using stochastic circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–4. https://doi.org/10.7873/DATE.2014.089

[3] B. R. Gaines. 1967. Stochastic Computing *(AFIPS '67 (Spring))*. Association for Computing Machinery, 149–156.

[4] Raffaele Giordano and Alberto Aloisio. 2011. Fixed-Latency, Multi-Gigabit Serial Links With Xilinx FPGAs. *IEEE Transactions on Nuclear Science* 58, 1 (2011), 194–201.

[5] Devon Jenson and Marc Riedel. 2016. A deterministic approach to stochastic computation. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/2966986.2966988

[6] Peng Li, Weikang Qian, and David J. Lilja. 2012. A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 303–308. https://doi.org/10.1109/ICCD.2012.6378656

[7] Siting Liu and Jie Han. 2018. Toward Energy-Efficient Stochastic Circuits Using Parallel Sobol Sequences. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 7 (July 2018), 1326–1339.

[8] M. Hassan Najafi, David J. Lilja, and Marc Riedel. 2018. Deterministic Methods for Stochastic Computing using Low-Discrepancy Sequences. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240797

[9] Keshab K. Parhi and Yin Liu. 2019. Computing Arithmetic Functions Using Stochastic Logic by Series Expansion. *IEEE Transactions on Emerging Topics in Computing* 7, 1 (2019), 44–59. https://doi.org/10.1109/TETC.2016.2618750

[10] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. 2018. A Survey on Deep Learning: Algorithms, Techniques, and Applications. *ACM Comput. Surv.* 51, 5 (Sept. 2018), 92:1–92:36.

[11] Weikang Qian. 2011. *Digital yet Deliberately Random: Synthesizing Logical Computation on Stochastic Bit Streams.* Ph. D. Dissertation. University of Minnesota.

[12] Weikang Qian, Xin Li, Marc D. Riedel, Kia Bazargan, and David J. Lilja. 2011. An Architecture for Fault-Tolerant Computation with Stochastic Logic. *IEEE Trans. Comput.* 60, 1 (2011), 93–105. https://doi.org/10.1109/TC.2010.202

[13] Weikang Qian and Marc D. Riedel. 2008. The synthesis of robust polynomial arithmetic with stochastic logic. In *2008 45th ACM/IEEE Design Automation Conference*. 648–653.

[14] W. Qian and M. D. Riedel. 2008. The Synthesis of Robust Polynomial Arithmetic with Stochastic Logic. In *Design Automation Conference (DAC)*. 648–653.

[15] Weikang Qian and Marc D. Riedel. 2009. Synthesizing Logical Computation on Stochastic Bit Streams. *Proceedings of the Design Automation Conference* (2009), 480–487.

[16] Weikang Qian, Marc D. Riedel, Hongchao Zhou, and Jehoshua Bruck. 2011. Transforming Probabilities with Combinational Logic. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 30, 9 (2011), 1279–1292.

[17] Weikang Qian, Chen Wang, Peng Li, David J. Lilja, Kia Bazargan, and Marc D. Riedel. 2012. An Efficient Implementation of Numerical Integration Using Logical Computation on Stochastic Bit Streams. *IEEE/ACM International conference on computer-aided design* (2012), 156–162.

[18] Sayed Ahmad Salehi, Yin Liu, Marc D. Riedel, and Keshab K. Parhi. 2017. Computing Polynomials with Positive Coefficients Using Stochastic Logic by Double-NAND Expansion. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*. Association for Computing Machinery, 471–474.