Synthesizing Logical Computation on Stochastic Bit Streams

Weikang Qian and Marc D. Riedel
Department of Electrical and Computer Engineering,
University of Minnesota, Twin Cities
{qianx030, mriedel}@umn.edu

ABSTRACT

Most digital systems operate on a positional representation of data, such as binary radix. A positional representation is a compact way to encode signal values: in binary radix, 2^n distinct values can be represented with n bits. However, operating on it requires complex logic: in each operation such as addition or multiplication, the signal must be "decoded," with the higher order bits weighted more than the lower order bits. We advocate an alternative representation: random bit streams where the signal value is encoded by the probability of obtaining a one versus a zero. This representation is much less compact than binary radix. However, complex operations can be performed with very simple logic. For instance, multiplication can be performed with a single AND gate. Also, because the representation is uniform, with all bits weighted equally, it is highly tolerant of soft errors (i.e., bit flips). In this paper, we present a general method for synthesizing digital circuitry that computes on such stochastic bit streams. Our method can be used to synthesize arbitrary polynomial functions. Through polynomial approximations, it can also be used to synthesize non-polynomial functions. Experiments on functions used in image processing show that our method produces circuits that are highly tolerant of input errors. The accuracy degrades gracefully with the error rate. For applications that mandate simple hardware, producing relatively low precision computation very reliably, our method is a winning proposition.

1. INTRODUCTION

Consider digital computation that is based on a *stochastic representation* of data: each real-valued number x ($0 \leq x \leq 1$) is represented by a sequence of random bits, each of which has probability x of being one and probability $1 - x$ of being zero. These bits can either be serial streaming on a single wire or in parallel on a bundle of wires. When serially streaming, the signals are probabilistic in *time*, as illustrated in Figure 1(a); when in parallel, they are probabilistic in *space*, as illustrated in Figure 1(b). Throughout this paper, we frame the discussion in terms of serial bit streams. However, our approach is equally applicable to parallel wire bundles. Indeed, we have advocated this sort of stochastic representation for technologies such as nanowire crossbar arrays [1].

Consider the problem of designing digital circuits that operate on stochastic bit streams. We focus on combinational circuits, that is to say, memoryless digital circuits built with logic gates such as AND, OR, and NOT. For such circuits, suppose that we supply stochastic bit streams as the inputs; we will observe stochastic bit streams at the outputs. Accordingly, combinational circuits can be viewed as constructs that accept real-valued probabilities as inputs and compute real-valued probabilities as outputs.[1] An illustration of this is shown in Figure 2. The circuit, consisting of an

[1]Throughout the paper, when we say "logical computation" or just "computation" on stochastic bit streams we mean combinational logic operating on such bit streams.

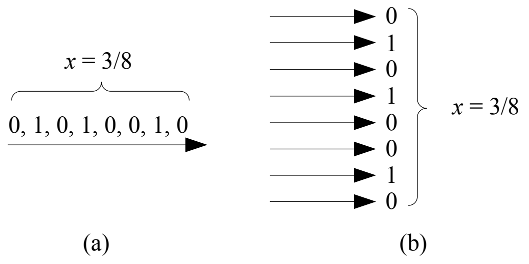

Figure 1: Stochastic representation: (a) A stochastic bit stream; (b) A stochastic wire bundle. A real value x in the unit interval $[0, 1]$ is represented as a bit stream or a bundle. For each bit in the bit stream or the bundle, the probability that it is one is x.

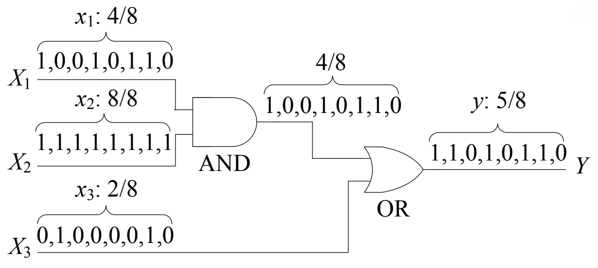

Figure 2: An example of logical computation on stochastic bit streams, implementing the arithmetic function $y = x_1x_2 + x_3 - x_1x_2x_3$. We see that, with inputs $x_1 = 1/2$, $x_2 = 1$ and $x_3 = 1/4$, the output is $5/8$, as expected.

AND gate and an OR gate, accepts ones and zeros and produces ones and zeros, as any digital circuit does. If we set the input bits x_1, x_2 and x_3 to be one randomly and independently with specific probabilities, then we will get an output y that is one with a specific probability. For instance, given input probabilities $x_1 = 1/2$, $x_2 = 1$ and $x_3 = 1/4$, the circuit in Figure 2 produces an output y with probability $5/8$.[2] The figure illustrates computations with bit lengths of 8.

Compared to a binary radix representation, a stochastic representation is 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, a stochastic representation requires a stream of 2^M bits. The two representations are at opposite ends of the spectrum: conventional binary radix is a maximally compressed, positional encoding; a stochastic representation is an uncompressed, uniform encoding.

A stochastic representation, although not very compact, has an advantage over binary radix in terms of error tolerance. Suppose that the environment is noisy: bit flips occur and these afflict all the

[2]When we say "probability" without further qualification, we mean the probability of obtaining a one.

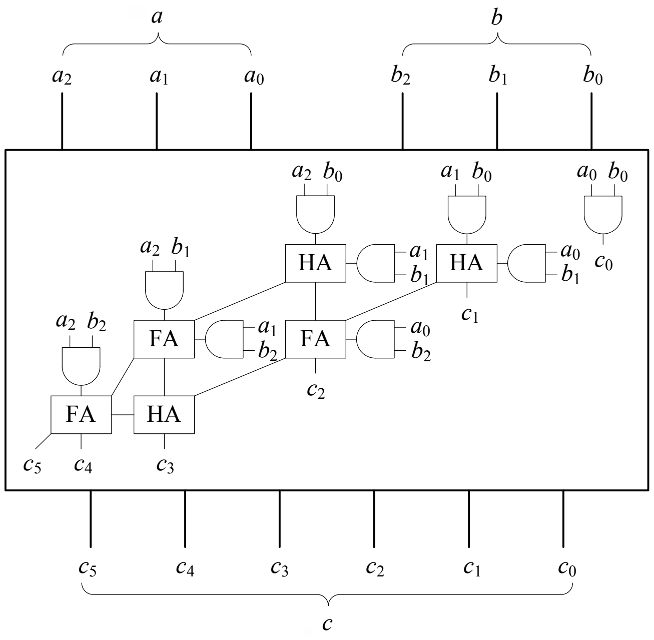

Figure 3: Multiplication on a conventional representation: a carry-save multiplier, operating on 3-bit binary radix encoded inputs A and B. "FA" refers to a full adder and "HA" refers to a half adder.

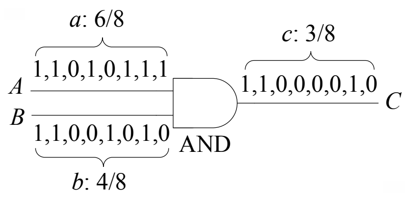

Figure 4: Multiplication on stochastic bit streams with an AND gate. Here the inputs are $6/8$ and $4/8$. The output is $6/8 \times 4/8 = 3/8$, as expected.

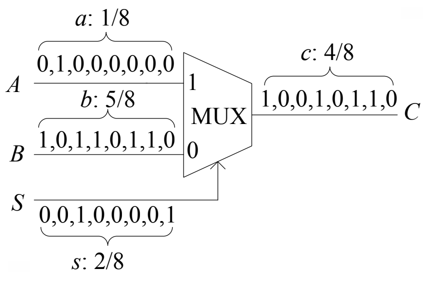

Figure 5: Scaled addition on stochastic bit streams, with a multiplexer (MUX). Here the inputs are $1/8, 5/8$, and $2/8$. The output is $2/8 \times 1/8 + (1 - 2/8) \times 5/8 = 4/8$, as expected.

bits with equal probability. Compare the two representations for a fractional number of the form $\frac{a}{2^M}$ for integers a between 0 and 2^M. With a binary radix representation, in the worst case, the most significant bit gets flipped, resulting in a change of $\frac{2^{M-1}}{2^M} = \frac{1}{2}$. In contrast, with a stochastic representation, all the bits in a stream of length 2^M have equal weight. Thus, a single bit flip always results in a change of $\frac{1}{2^M}$, which is small in comparison.

With the stochastic representation, noise does not introduce more randomness. The bit streams are random to begin with, biased to specific probability values. Rather, noise distorts the bias, producing streams with different probabilities than intended. However, this change is small. With a bit flip rate of ϵ, the change is bounded by ϵ.[3]

A stochastic representation also has the advantage over binary radix in the amount of hardware needed for arithmetic computation. Consider multiplication. Figure 3 shows a conventional design for a 3-bit carry-save multiplier, operating on binary radix-encoded numbers. It consists of 9 AND gates, 3 half adders and 3 full adders, for a total of 30 gates.[4]

In contrast, with a stochastic representation, multiplication can be implemented with much less hardware: we only need one AND gate. Figure 4 illustrates the multiplication of values that are represented by stochastic bit streams. Assuming that the two input stochastic bit streams A and B are independent, the number represented by the output stochastic bit stream C is

$$\begin{aligned} c &= P(C=1) = P(A=1 \text{ and } B=1) \\ &= P(A=1)P(B=1) = a \cdot b. \end{aligned} \tag{1}$$

So the AND gate multiplies the two values represented by the stochastic bit streams. In the figure, with bit streams of length 8, the values have a resolution of $1/8$.

Why is multiplication so simple with a stochastic representation and so complex with a conventional positional representation? Although compact, a positional representation imposes a computational burden for arithmetic: for each operation we must, in essence, "decode" the operands, weighting the higher order bits more and the lower order bits less; then we must "re-encode" the result in weighted form. Since the stochastic representation is uniform, no decoding and no re-encoding are required to operate on the values.

We can perform operations other than multiplication with the stochastic representation. Consider addition. It is not feasible to add two probability values directly; this could result in a value greater than one, which cannot be represented as a probability value. However, we can perform *scaled* addition. Figure 5 shows a scaled adder operating on real numbers in the stochastic representation. It consists of a multiplexer (MUX), a digital construct that selects one of its two input values to be the output value, based on a third "selecting" input value. For the multiplexer shown in Figure 5, S is the selecting input. When $S = 1$, the output $C = A$. Otherwise, when $S = 0$, the output $C = B$. The Boolean function implemented by the multiplexer is $C = (A \wedge S) \vee (B \wedge \neg S)$.[5] With the assumption that the three input stochastic bit streams A, B, and S are independent, the number represented by the output stochastic bit stream C is

$$\begin{aligned} c &= P(C=1) \\ &= P(S=1 \text{ and } A=1) + P(S=0 \text{ and } B=1) \\ &= P(S=1)P(A=1) + P(S=0)P(B=1) \\ &= s \cdot a + (1-s) \cdot b. \end{aligned} \tag{2}$$

Thus, with this stochastic representation, the computation performed by a multiplexer is the scaled addition of the two input values a and b, with a scaling factor of s for a and $1 - s$ for b.

The task of *analyzing* combinational circuitry operating on stochastic bit streams is well understood [2]. For instance, it can be shown that, given an input x, an inverter (i.e., a NOT gate) implements the operation $1 - x$. Given inputs x and y, an OR gate implements the operation $x + y - xy$. Analyzing the circuit in Figure 2, we see that it implements the function $x_1x_2 + x_3 - x_1x_2x_3$. Aspects such

[3]In fact, with a bit flip rate of ϵ, a number p in the stochastic representation is biased to a number $p(1-\epsilon) + (1-p)\epsilon$, a change of $(1-2p)\epsilon$ in the value.

[4]A half adder can be implemented with one XOR gate and one AND gate. A full adder can be implemented with two XOR gates, two AND gates, and one OR gate.

[5]When discussing Boolean functions, we will use $\wedge$, $\vee$, and $\neg$ to represent logical AND, OR, and negation, respectively. We adopt this convention since we use $+$ and $\cdot$ to represent *arithmetic* addition and multiplication, respectively.

as signal correlations of reconvergent paths must be taken into account. Algorithmic details for such analysis were first fleshed out by the testing community [3]. They have also found mainstream application for tasks such as timing and power analysis [4, 5].

In this paper, we will explore the more challenging task of *synthesizing* logical computation on stochastic bit streams that implements the functionality that we want. Naturally, since we are mapping probabilities to probabilities, we can only implement functions that map the unit interval $[0, 1]$ onto the unit interval $[0, 1]$. Based on the constructs for multiplication and scaled addition shown in Figures 4 and 5, we can readily implement polynomial functions of a specific form, namely polynomials with non-negative coefficients that sum up to a value no more than one:

$$g(t) = \sum_{i=0}^{n} a_i t^i$$

where, for all $i = 0, \ldots, n$, $a_i \geq 0$ and $\sum_{i=0}^{n} a_i \leq 1$.

For example, suppose that we want to implement the polynomial $g(t) = 0.3t^2 + 0.3t + 0.2$ through logical computation on stochastic bit streams. We first decompose it in terms of multiplications of the form $a \cdot b$ and scaled additions of the form $sa + (1 - s)b$, where s is a constant:

$$g(t) = 0.8(0.75(0.5t^2 + 0.5t) + 0.25 \cdot 1).$$

Then, we reconstruct it with the following sequence of multiplications and scaled additions:

$$\begin{aligned} w_1 &= t \cdot t, \\ w_2 &= 0.5w_1 + (1 - 0.5)t, \\ w_3 &= 0.75w_2 + (1 - 0.75) \cdot 1, \\ w_4 &= 0.8 \cdot w_3. \end{aligned}$$

The circuit implementing this sequence of operations is shown in Figure 6. In the figure, the inputs are labeled with the probabilities of the bits of the corresponding stochastic streams. Some of the inputs have fixed probabilities and the others have variable probabilities t. Note that the different lines with the input t are each fed with *independent* stochastic streams with bits that have probability t.

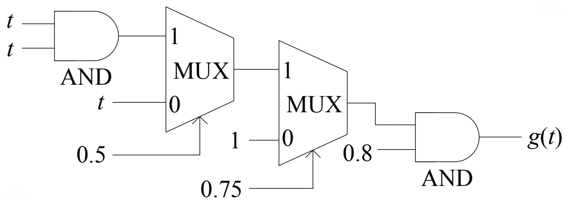

Figure 6: Computation on stochastic bit streams implementing the polynomial $g(t) = 0.3t^2 + 0.3t + 0.2$.

What if the target function is a polynomial that is not decomposable this way? Suppose that it maps the unit interval onto the unit interval but it has some coefficients less than zero or some greater than one. For instance, consider the polynomial $g(t) = \frac{3}{4} - t + \frac{3}{4}t^2$. It is not apparent how to construct a network of stochastic multipliers and adders to implement it.

In this paper, we propose a general method for synthesizing arbitrary univariate polynomial functions on stochastic bit streams. A necessary condition is that the target polynomial maps the unit interval onto the unit interval. Our major contribution is to show that this condition is also sufficient: we provide a constructive method for implementing any polynomial that satisfies this condition. Our method is based on some novel mathematics for manipulating polynomials in a special form called a Bernstein polynomial. In [6] we showed how to convert a general power-form polynomial into a Bernstein polynomial with coefficients in the unit interval. In [7] we showed how to realize such a polynomial with a form of "generalized multiplexing."

We illustrate the basic steps of our synthesis method with the example of $g(t) = \frac{3}{4} - t + \frac{3}{4}t^2$. (We define Bernstein polynomials in Section 2. We provide further details regarding the synthesis method in Section 3.)

1. Convert the polynomial into a Bernstein polynomial with all coefficients in the unit interval:
$$g(t) = \frac{3}{4} \cdot [(1 - t)^2] + \frac{1}{4} \cdot [2t(1 - t)] + \frac{1}{2} \cdot [t^2].$$
Note that the coefficients of the Bernstein polynomial are $\frac{3}{4}$, $\frac{1}{4}$ and $\frac{1}{2}$, all of which are in the unit interval.

2. Implement the Bernstein polynomial with a multiplexing circuit, as shown in Figure 7. The block labeled "+" counts the number of ones among its two inputs; this is either 0, 1, or 2. The multiplexer selects one of its three inputs as its output according to this value. Note that the inputs with probability t are each fed with *independent* stochastic streams with bits that have probability t.

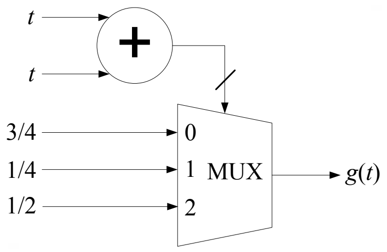

Figure 7: A generalized multiplexing circuit implementing the polynomial $g(t) = \frac{3}{4} - t + \frac{3}{4}t^2$.

1.1 Input/Output Interface

The premise for the synthesis method in this paper is that the inputs and outputs to combinational circuitry consist of stochastic bit streams (or, equivalently, of stochastic bits on parallel wire bundles). Either the inputs are presented in this form or else they must be encoded this way, say from binary radix. Either the outputs are usable in this form or they must be decoded, say back into binary radix. This input/output encoding and decoding is not the focus of this paper; rather, our contributions are at a conceptual level in terms of the logic design and the mathematics. Here we briefly comment the input/output interfacing.

For a variety of circuit applications, for instance in sensors and embedded systems, the inputs are obtained from physical measurements in analog form, so as real-valued numbers. In the analog to digital (A/D) conversion process, these real-valued numbers are converted to binary radix. However, many A/D converters, such as sigma-delta converters, naturally produce pulse streams of ones and zeros as an intermediate form [8]. Such converters could easily be adapted to produce stochastic bit streams, exploiting white noise for the encoding. Similarly, at the outputs, digital to analog (D/A) conversion could produce analog values directly from stochastic bit streams.

For applications that can exploit physical sources of randomness to generate bit streams, but can only generate *fixed* probabilities, we have developed a methodology for *transforming* the probabilities of bit streams through combinational logic [9, 10]. In particular, given unbiased bit streams – so streams with probability 0.5 – we can generate stochastic bit streams with arbitrary binary fractional probabilities.

In prior work [11], we validated our synthesis methodology using digital constructs to generate pseudorandom bit streams at the inputs and to transform these streams back to numerical values at the outputs; specifically, we used linear feedback shift registers (LFSRs) and counters for these tasks. In a sense, we are handicapping ourselves by using such digital constructs, since the cost of converting to and from pseudo-randomness dominates our designs. Still, as we show, our designs compete with conventional designs very favorably in terms of cost. They perform much better in terms of error tolerance.

1.2 Related Work

A sequence of early papers established the concept of logical computation on stochastic bit streams [12, 13]. These papers discussed basic operations such as multiplication and addition. Later papers delved into more complex operations, including exponential functions and square roots [14, 15]. In [16], the authors discuss the implementation of basic arithmetic operations as well as complex ones, including hyperbolic functions, with stochastic bit streams. They also discuss different forms of stochastic representation, including a "bipolar" representation for negative values. Much of the interest in computing with stochastic bit streams stems from the field of neural networks, where the concept is known as "pulsed" or "pulse-coded" computation [17, 18].

In fact, the general concept of stochastic computing dates back even earlier, to work by J. von Neumann in the 1950's [19]. He applied probabilistic logic to the study of thresholding and multiplexing operations on bundles of wires with stochastic signals. As he eloquently states in the introduction to his seminal paper, "Error is viewed not as an extraneous and misdirected or misdirecting accident, but as an essential part of the [design]." We find this view, that randomness and noise are integral to computation, to be compelling in the modern era of nanoscale electronics.

We point to two recent research efforts that embrace randomness in circuit and system design. In [20], the authors propose a construct that they call probabilistic CMOS (PCMOS) that generates random bits from intrinsic sources of noise. In [21], PCMOS switches are applied to form a probabilistic system-on-a-chip (PSOC); this system provides intrinsic randomness to the application layer, so that it can be exploited by probabilistic algorithms. In [22] and [23], the authors propose a methodology for designing stochastic processors, that is to say, processors that can tolerate computational errors caused by hardware uncertainties. They strive for a favorable trade-off between reliability and power consumption.

1.3 Paper Organization

In Section 2, we provide some mathematical preliminaries on Bernstein polynomials. In Section 3, we present our general method for synthesizing arbitrary univariate polynomial functions on stochastic bit streams. In Section 4, we generalize the method to arbitrary non-polynomial functions through polynomial approximations. In Section 5, we present experimental results on hardware cost, performance and error tolerance for stochastic implementations of polynomials and non-polynomial functions. In Section 6, we conclude with some thoughts on the potential impact and future directions of this work.

2. BERNSTEIN POLYNOMIALS

In this section, we discuss some of the mathematical properties of a specific type of polynomial that we use, namely Bernstein polynomials [24].

Definition 1
The family of $n+1$ polynomials of the form

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n$$

are called Bernstein basis polynomials *of degree n.*[6] □

Definition 2
A linear combination of Bernstein basis polynomials of degree n

$$B_n(t) = \sum_{i=0}^{n} b_{i,n} B_{i,n}(t) \qquad (3)$$

is a Bernstein polynomial of degree n. The $b_{i,n}$'s are called Bernstein coefficients. □

Polynomials are usually represented in power form. We can convert a power-form polynomial of degree n, $g(t) = \sum_{i=0}^{n} a_{i,n} t^i$, into a Bernstein polynomial of degree n as $g(t) = \sum_{i=0}^{n} b_{i,n} B_{i,n}(t)$. The conversion from the power-form coefficients $a_{i,n}$ to the Bernstein coefficients $b_{i,n}$ is a closed-form expression:

$$b_{i,n} = \sum_{j=0}^{i} \frac{\binom{i}{j}}{\binom{n}{j}} a_{j,n}, \quad 0 \le i \le n. \qquad (4)$$

The reader is referred to [25] for a proof of this.

Example 1
The polynomial $g_1(t) = \frac{1}{4} + \frac{9}{8}t - \frac{15}{8}t^2 + \frac{5}{4}t^3$ can be converted into a Bernstein polynomial of degree 3:

$$g_1(t) = \frac{2}{8}B_{0,3}(t) + \frac{5}{8}B_{1,3}(t) + \frac{3}{8}B_{2,3}(t) + \frac{6}{8}B_{3,3}(t). \qquad \square$$

Generally, a power-form polynomial of degree n can be converted into an equivalent Bernstein polynomial of degree greater than or equal to n. The coefficients of a Bernstein polynomial of degree $m+1$ ($m \ge n$) can be derived from the Bernstein coefficients of an equivalent Bernstein polynomial of degree m. We refer to this as *degree elevation*.

$$b_{i,m+1} = \begin{cases} b_{0,m} & i = 0 \\ (1 - \frac{i}{m+1})b_{i,m} + \frac{i}{m+1}b_{i-1,m} & 1 \le i \le m \\ b_{m,m} & i = m+1. \end{cases} \qquad (5)$$

Again, the reader is referred to [25] for a proof of this.

3. SYNTHESIZING POLYNOMIAL ARITHMETIC

As we show in [7], computation on stochastic bit streams generally implements a multivariate polynomial $F(x_1, \ldots, x_n)$ with integer coefficients. The degree of each variable is at most one, i.e., there are no terms with variables raised to the power of two, three or higher. If we associate some of the x_i's of the polynomial $F(x_1, \ldots, x_n)$ with real constants in the unit interval and the others with a common variable t, then the function F becomes a real-coefficient *univariate* polynomial $g(t)$. With different choices of the original Boolean function f and different settings of the probabilities of the x_i's, we get different polynomials $g(t)$.

Example 2
Consider the function implemented by a multiplexer operating on stochastic bit streams, Equation (2). It is a multivariate polynomial, $g(a,b,s) = b + sa - sb$. The polynomial has integer coefficients. The degree of each variable is at most one. If we set $s = a = t$ and $b = 0.8$ in the polynomial, then we get a univariate polynomial $g(t) = 0.8 - 0.8t + t^2$. □

The first question that arises is: what kind of univariate polynomials can be implemented by computation on stochastic bit streams?

[6]Here $\binom{n}{k}$ denotes the binomial coefficient "n choose k."

In [6], we prove the following theorem stating a necessary condition on the polynomials. The theorem essentially says that, given inputs that are probability values – that is to say, real values in the unit interval – the polynomial must also evaluate to a probability value. There is a caveat here: if the polynomial is not identically equal to 0 or 1, then it must evaluate to a value in the open interval $(0, 1)$ when the input is also in the open interval $(0, 1)$.

Theorem 1
If a polynomial $g(t)$ can be implemented by logical computation on stochastic bit streams, then

1. *$g(t)$ is identically equal to 0 or 1 ($g(t) \equiv 0$ or 1), or*
2. *$g(t)$ maps the open interval $(0, 1)$ to itself ($g(t) \in (0, 1)$, for all $t \in (0, 1)$) and $0 \le g(0), g(1) \le 1$.* □

For instance, as shown in Example 2, the polynomial $g(t) = 0.8 - 0.8t + t^2$ can be implemented by logical computation on stochastic bit streams. It is not hard to see that $g(t)$ satisfies the necessary condition. In fact, $g(0) = 0.8$, $g(1) = 1$ and $0 < g(t) < 1$, for all $0 < t < 1$.

The next question that arises is: can *any* polynomial satisfying the necessary condition be implemented by logical computation on stochastic bit streams? If so, how? We propose a synthesis method that solves this problem; constructively, we show that, provided that a polynomial satisfies the necessary condition, we can implement it. First, in Section 3.1, we show how to implement a Bernstein polynomial with coefficients in the unit interval. Then, in Section 3.2, we describe how to convert a general power-form representation into such a polynomial.

3.1 Synthesizing Bernstein Polynomials with Coefficients in the Unit Interval

If all the coefficients of a Bernstein polynomial are in the unit interval, i.e., $0 \le b_{i,n} \le 1$, for all $0 \le i \le n$, then we can implement it with the construct shown in Figure 8.

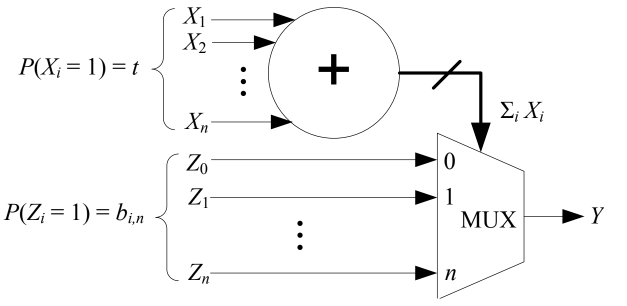

Figure 8: Combinational logic that implements a Bernstein polynomial $B_n(t) = \sum_{i=0}^{n} b_{i,n} B_{i,n}(t)$ with all coefficients in the unit interval.

The block labeled "+" in Figure 8 has n inputs $X_1, \ldots, X_n$ and $\lceil \log_2(n+1) \rceil$ outputs. It consists of combinational logic that computes the weight of the inputs, that is to say, it counts the number of ones in the n Boolean inputs $X_1, \ldots, X_n$, producing a *binary radix encoding* of this count. We will call this an n-bit Boolean "weight counter." The multiplexer (MUX) shown in the figure has "data" inputs $Z_0, \ldots, Z_n$ and the $\lceil \log_2(n+1) \rceil$ outputs of the weight counter as the selecting inputs. If the binary radix encoding of the outputs of the weight counter is k $(0 \le k \le n)$, then the output Y of the multiplexer is set to Z_k.

Figure 9 gives a simple design for an 8-bit Boolean weight counter based on a tree of adders. An n-bit Boolean weight counter can be implemented in a similar way.

In order to implement the Bernstein polynomial

$$B_n(t) = \sum_{i=0}^{n} b_{i,n} B_{i,n}(t),$$

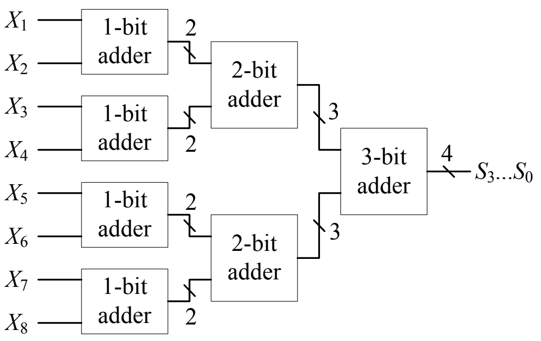

Figure 9: The implementation of an 8-bit Boolean weight counter.

we set the inputs $X_1, \ldots, X_n$ to be independent stochastic bit streams with probability t. Equivalently, $X_1, \ldots, X_n$ can be viewed as independent random Boolean variables that have the same probability t of being one. The probability that the count of ones among the X_i's is k $(0 \le k \le n)$ is given by the binomial distribution:

$$P\left(\sum_{i=1}^{n} X_i = k\right) = \binom{n}{k} t^k (1-t)^{n-k} = B_{k,n}(t). \quad (6)$$

We set the inputs $Z_0, \ldots, Z_n$ to be independent stochastic bit streams with probability equal to the Bernstein coefficients $b_{0,n}, \ldots, b_{n,n}$, respectively. Notice that we can represent $b_{i,n}$ with stochastic bit streams because we assume that $0 \le b_{i,n} \le 1$. Equivalently, we can view $Z_0, \ldots, Z_n$ as $n+1$ independent random Boolean variables that are one with probabilities $b_{0,n}, \ldots, b_{n,n}$, respectively.

The probability that the output Y is one is

$$\begin{aligned} y &= P(Y = 1) \\ &= \sum_{k=0}^{n} \left(P\left(Y = 1 | \sum_{i=1}^{n} X_i = k\right) P\left(\sum_{i=1}^{n} X_i = k\right) \right). \end{aligned} \quad (7)$$

Since the multiplexer sets Y equal to Z_k, when $\sum_{i=1}^{n} X_i = k$, we have

$$P\left(Y = 1 | \sum_{i=1}^{n} X_i = k\right) = P(Z_k = 1) = b_{k,n}. \quad (8)$$

Thus, from Equations (3), (6), (7), and (8), we have

$$y = \sum_{k=0}^{n} b_{k,n} B_{k,n}(t) = B_n(t). \quad (9)$$

We conclude that the circuit in Figure 8 implements the given Bernstein polynomial with all coefficients in the unit interval. We have the following theorem.

Theorem 2
If all the coefficients of a Bernstein polynomial are in the unit interval, i.e., $0 \le b_{i,n} \le 1$, for $0 \le i \le n$, then we can synthesize logical computation on stochastic bit streams to implement it. □

Example 3
Figure 10 shows a circuit that implements the Bernstein polynomial

$$g_1(t) = \frac{2}{8} B_{0,3}(t) + \frac{5}{8} B_{1,3}(t) + \frac{3}{8} B_{2,3}(t) + \frac{6}{8} B_{3,3}(t),$$

converted from the power-form polynomial $g_1(t)$ in Example 1. The function is evaluated at $t = 0.5$. The stochastic bit streams X_1, X_2 and X_3 are independent, each with probability $t = 0.5$. The stochastic bit streams $Z_0, \ldots, Z_3$ have probabilities $\frac{2}{8}$, $\frac{5}{8}$, $\frac{3}{8}$, and $\frac{6}{8}$, respectively. As expected, the computation produces the correct output value: $g_1(0.5) = 0.5$. □

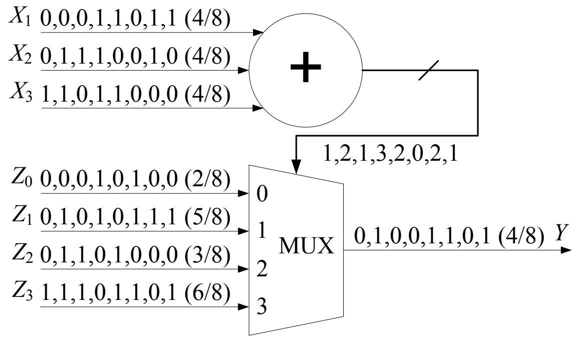
X_1 0,0,0,1,1,0,1,1 (4/8)
X_2 0,1,1,1,0,0,1,0 (4/8)
X_3 1,1,0,1,1,0,0,0 (4/8)
+
1,2,1,3,2,0,2,1
Z_0 0,0,0,1,0,1,0,0 (2/8)
Z_1 0,1,0,1,0,1,1,1 (5/8)
Z_2 0,1,1,0,1,0,0,0 (3/8)
Z_3 1,1,1,0,1,1,0,1 (6/8)
0
1
2
3
MUX
0,1,0,0,1,1,0,1 (4/8) Y

Figure 10: Computation on stochastic bit streams that implements the Bernstein polynomial $g_1(t) = \frac{2}{8}B_{0,3}(t) + \frac{5}{8}B_{1,3}(t) + \frac{3}{8}B_{2,3}(t) + \frac{6}{8}B_{3,3}(t)$ at $t = 0.5$.

3.2 Synthesis of Power-Form Polynomials

In the previous section, we saw that we can implement a polynomial through logical computation on stochastic bit streams if the polynomial can be represented as a Bernstein polynomial with coefficients in the unit interval. A question that arises is: what kind of polynomials can be represented in this form? Generally, we seek to implement polynomials given to us in power form. In [6], we proved that any polynomial that satisfies Theorem 1 – so essentially any polynomial that maps the unit interval onto the unit interval – can be converted into a Bernstein polynomial with all coefficients in the unit interval.[7] Based on this result and Theorem 2, we can see that the *necessary* condition shown in Theorem 1 is also a *sufficient* condition for a polynomial to be implemented by logical computation on stochastic bit streams.

Example 4
Consider the polynomial $g_2(t) = 3t - 8t^2 + 6t^3$ of degree 3, Since $g_2(t) \in (0,1)$, for all $t \in (0,1)$ and $g_2(0) = 0, g_2(1) = 1$, it satisfies the necessary condition shown in Theorem 1. Note that

$$\begin{aligned} g_2(t) &= B_{1,3}(t) - \frac{2}{3}B_{2,3}(t) + B_{3,3}(t) \\ &= \frac{3}{4}B_{1,4}(t) + \frac{1}{6}B_{2,4}(t) - \frac{1}{4}B_{3,4}(t) + B_{4,4}(t) \\ &= \frac{3}{5}B_{1,5}(t) + \frac{2}{5}B_{2,5}(t) + B_{5,5}(t). \end{aligned}$$

Thus, the polynomial $g_2(t)$ can be converted into a Bernstein polynomial with coefficients in the unit interval. The degree of such a Bernstein polynomial is 5, greater than that of the original power form polynomial. □

Given a power-form polynomial $g(t) = \sum_{i=0}^{n} a_{i,n}t^i$ that satisfies the condition of Theorem 1, we can synthesize it in the following steps:

1. Let $m = n$. Obtain $b_{0,m}, b_{1,m}, \dots, b_{m,m}$ from $a_{0,n}, a_{1,n}, \dots, a_{n,n}$ by Equation (4).
2. Check to see if $0 \le b_{i,m} \le 1$, for all $i = 0, 1, \dots, m$. If so, go to step 4.
3. Let $m = m + 1$. Calculate $b_{0,m}, b_{1,m}, \dots, b_{m,m}$ from $b_{0,m-1}, b_{1,m-1}, \dots, b_{m-1,m-1}$ based on Equation (5). Go to step 2.
4. Synthesize the Bernstein polynomial
$$B_m(t) = \sum_{i=0}^{m} b_{i,m}B_{i,m}(t).$$
with the generalized multiplexing construct in Figure 8.

[7]The degree of the equivalent Bernstein polynomial with coefficients in the unit interval may be greater than the degree of the original polynomial.

4. SYNTHESIZING NON-POLYNOMIAL FUNCTIONS

In real applications, we often encounter non-polynomial functions, such as trigonometric functions. In this section, we discuss the implementation of such functions; further details are given in [11]. Our strategy is to approximate them by Bernstein polynomials with coefficients in the unit interval. In the previous section, we saw how to implement such Bernstein polynomials.

We formulate the problem of implementing an arbitrary function $g(t)$ as follows. Given $g(t)$, a continuous function on the unit interval, and n, the degree of a Bernstein polynomial, find real numbers $b_{i,n}$, $i = 0, \dots, n$, that minimize

$$\int_0^1 (g(t) - \sum_{i=0}^{n} b_{i,n}B_{i,n}(t))^2 \, \mathrm{d}t, \tag{10}$$

subject to

$$0 \le b_{i,n} \le 1, \text{ for all } i = 0, 1, \dots, n. \tag{11}$$

Here we try to find the optimal approximation by minimizing an objective function, Equation (10), that measures the approximation error. This is the square of the L^2 norm on the difference between the original function $g(t)$ and the Bernstein polynomial $B_n(t) = \sum_{i=0}^{n} b_{i,n}B_{i,n}(t)$. The integral is on the unit interval because t, representing a probability value, is always in the unit interval. The constraints in Equation (11) guarantee that the Bernstein coefficients are all in the unit interval. With such coefficients, the construct in Figure 8 computes an optimal approximation of the function.

The optimization problem is a constrained quadratic programming problem [11]. Its solution can be obtained using standard techniques.

Example 5
Consider the non-polynomial function $g_3(t) = t^{0.45}$. We approximate this function by a Bernstein polynomial of degree 6. By solving the constrained quadratic optimization problem, we obtain the Bernstein coefficients:

$$b_{0,6} = 0.0955, b_{1,6} = 0.7207, b_{2,6} = 0.3476, b_{3,6} = 0.9988,$$
$$b_{4,6} = 0.7017, b_{5,6} = 0.9695, b_{6,6} = 0.9939. \quad \square$$

5. EXPERIMENTS

We present experimental results analyzing the hardware cost, performance, and error tolerance for implementations of polynomial and non-polynomial functions. Specifically, we compare the *area-delay product* of stochastic versus conventional implementations of arbitrary polynomials. Also, we compare the tolerance to soft errors, i.e., bit flips, in the input data for both implementations of a common non-polynomial function in image processing, namely *gamma correction*.

5.1 Hardware Comparison

Suppose that we want to compute a polynomial $g(t) = \sum_{i=0}^{n} a_{i,n}t^i$ with a resolution 2^{-M}. To achieve this resolution, a conventional implementation based on binary radix requires M bits. A stochastic implementation requires bit streams of length 2^M.

For our conventional implementation, we first factorize the polynomial as $g(t) = a_{0,n}+t(a_{1,n}+t(a_{2,n}+\cdots+t(a_{n-1,n}+ta_{n,n})))$. From this form, we implement it with a sequence of multiplications and additions. Since it is of degree n, we evaluate the polynomial with n iterations through a circuit consisting of a multiplier and an adder. For the hardware, we use standard benchmark circuits.[8] For M bits, the circuit consists of $10M^2 - 4M - 9$ logic gates.

[8]Specifically, we use circuit C6288 from the ISCAS'85 collection for the multiplier [26]. It is typical of the genre, built with carry-save adders.

The critical path passes through $12M - 11$ gates. We assume that the cost of each logic gate is *unit area* and the time for each logic gate to compute a value is *unit delay*. Accordingly, the area-delay product for this conventional implementation of a polynomial of degree n is

$$(10M^2 - 4M - 9)(12M - 11)n,$$

where the factor n accounts for the n iterations.

For our stochastic implementation, we first convert a polynomial from a power form into a Bernstein form. Then we build the circuit structure shown in Figure 8 to compute the function. Table 1 shows the area $A(n)$ and delay $D(n)$ of the circuits to compute Bernstein polynomials of degree $n = 3, 4, 5,$ and 6. The area-delay product for a stochastic implementation of a polynomial of degree n is

$$A(n)D(n)2^M,$$

where the factor 2^M accounts for the length of the bit streams. (Note that if we were using parallel wire bundles instead of serial bit streams, then we would need 2^M *copies* instead of 2^M *cycles*; the area-delay product would be the same.)

Table 1: The area and delay of the circuits to compute Bernstein polynomials of degree 3, 4, 5, and 6.

degree n of Bernstein polynomial	area $A(n)$	delay $D(n)$
3	22	10
4	40	17
5	49	20
6	58	20

Table 2: A comparison of the area-delay product for conventional and stochastic implementations of polynomials with different degree n and resolution 2^{-M}.

n	M	area-delay product		$\frac{\text{stochastic product}}{\text{conventional product}}$
		conventional	stochastic	
3	7	99207	28160	0.284
	8	152745	56320	0.369
	9	222615	112640	0.506
	10	310977	225280	0.724
4	7	132276	87040	0.658
	8	203660	174080	0.855
	9	296820	348160	1.173
	10	414636	696320	1.679
5	7	165345	125440	0.759
	8	254575	250880	0.986
	9	371025	501760	1.352
	10	518295	1003520	1.936
6	7	198414	148480	0.748
	8	305490	296960	0.972
	9	445230	593920	1.334
	10	621954	1187840	1.910

Table 2 shows the area-delay product for conventional and stochastic implementations for polynomials of degrees $n = 3, 4, 5, 6$ and resolutions 2^{-M}, $M = 7, 8, 9, 10$. The last column shows the ratio of the two. We can see that for $M \leq 8$, the area-delay product of the stochastic implementation is always less than that of the conventional implementation. Indeed, for small values of n and M, it is much less.

5.2 Comparison of Error Tolerance for Gamma Correction Function

In this section, we compare the error tolerance of our stochastic implementation to that of conventional implementation of a non-polynomial function commonly used in image processing: gamma correction. The gamma correction function is a nonlinear operation used to code and decode luminance and tri-stimulus values in video and still-image systems. It is defined by a power-law expression

$$V_{\text{out}} = V_{\text{in}}^{\gamma},$$

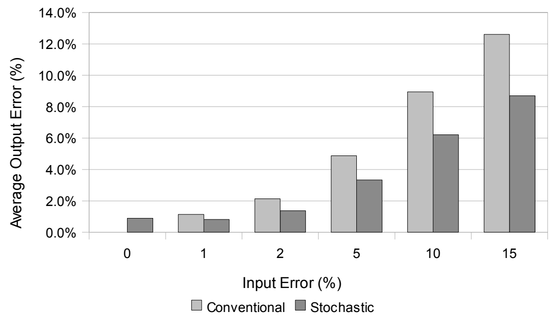

Figure 11: The average output errors of conventional and stochastic implementations under different input error ratios.

where V_{in} is normalized between zero and one [27]. We apply a value of $\gamma = 0.45$, which is the value used in most video cameras.

For a conventional implementation, we implement the gamma correction function as it is most commonly done: based on direct table lookup. For example, for a display system that supports 8 bits of color depth per pixel, we use an 8-bit input / 8-bit output table of its values.

For a stochastic implementation, we use the techniques shown in Section 4 to approximate the function by a Bernstein polynomial. With $\gamma = 0.45$, the gamma correction function is just the function shown in Example 5. We choose the degree of the Bernstein polynomial to be 6. For this value, the Bernstein coefficients are those in that example. The precision of the computation is 2^{-10}. Accordingly, the length of the stochastic bit streams is fixed at $2^{10} = 1024$ bits.

We compare the tolerances of the two implementations to soft errors in the input data. These are simulated by independently flipping a given percentage of the input bits. For example, if the input error ratio is 5%, this implies that 5% of the total number of input bits are randomly chosen and flipped. The input image without error is shown in the left column of Figure 12.

We measure the output error. Figure 11 plots the average percentage of output error in the images generated by the two implementations for five different input error ratios. For any given error rate larger than zero, the stochastic implementation generates smaller output errors than the conventional implementation does. The rate of the output errors grows more slowly as the input error rate increases. This is as expected: with the stochastic implementation, all bits have equal weight; each bit flip does little damage. With the conventional implementation, bit flips afflict each bit of the binary radix representation with equal probability. If the most significant bit gets flipped, the error that occurs is large.

The effect of bit flips is more clearly demonstrated in Figure 12, which shows the images generated by the two implementations under different input error ratios. When the input error ratio is 10%, the image generated by the conventional method is full of noisy pixels, while the image generated by the stochastic method is still recognizable.

6. DISCUSSION

The computation that we are advocating in this paper has a pseudo *analog* character, reminiscent of computations performed by physical systems such as electronics on continuously variable signals such as voltage. In our case, the variable signal is the probability of obtaining a one in a stochastic yet *digital* bit stream. Indeed, our system is built from ordinary, cheap digital electronics such as CMOS. Digital constructs in CMOS operate on physical signals such as voltage, of course. However, they are designed with the premise that these signals can always be unequivocally interpreted as zero or as one.

This is certainly counterintuitive: why impose an analog view on digital values? As we have outlined in this paper, it might often be advantageous to do so, both from the standpoint of the hardware re-

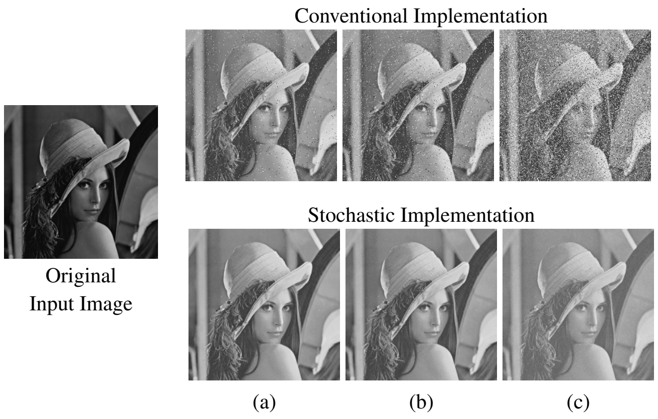

Figure 12: Error tolerance for the gamma correction function. The images in the top row are generated by a conventional implementation. The images in the bottom row are generated by our stochastic implementation. Input error ratios are (a) 1%; (b) 2%; (c) 10%.

sources required as well as the error tolerance of the computation. Many of the functions that we seek to implement for computational systems such as signal processing are *arithmetic* functions, consisting of operations like addition and multiplication. Complex functions, such as exponentials and trigonometric functions, are generally computed through polynomial approximations, so through multiplications and additions. As we have argued, these operations are very natural and efficient when performed on stochastic bit streams.

We are the first to tackle the problem of synthesizing arbitrary polynomial functions through logical computation on stochastic bit streams. The synthesis results for our stochastic implementations of a variety of functions are convincing. The area-delay product is comparable to that of conventional implementations with adders and multipliers. Since stochastic bit streams are uniform, with no bit privileged above any other, the computation is highly error tolerant. As higher and higher rates of bit flips occur, the accuracy degrades gracefully.

Indeed, computation on stochastic bit streams could offer tunable precision: as the length of the stochastic bit stream increases, the precision of the value represented by it also increases. Thus, without hardware redesign, we have the flexibility to tradeoff precision and computation time. In contrast, with a conventional binary-radix implementation, when a higher precision is required, the underlying hardware has to be redesigned. Note that we have been evaluating our stochastic implementations under the conservative assumption that the clock rate will be the same as that of a conventional implementation. However, with much simpler hardware – for instance a single AND gate performing a complex task like multiplication – we could potentially implement computation with much higher clock rates, particularly if it is pipelined.

In this paper, we have discussed *combinational* logic operating on stochastic bit streams. Such combinational circuitry often forms the bulk of the computation in datapaths. In future work, we will attempt to generalize the paradigm to *sequential* logic, i.e., finite-state machines, operating on stochastic bit streams.

7. ACKNOWLEDGMENTS

This work is supported by a grant from the Semiconductor Research Corporation's Focus Center Research Program on Functional Engineered Nano-Architectonics and by a CAREER Award, #0845650, from the National Science Foundation. The authors thank Xin Li for his help with the experiments and Kia Bazargan, David Lilja, and Ivo Rosenberg for discussions.

8. REFERENCES

[1] W. Qian, J. Backes, and M. D. Riedel, "The synthesis of stochastic circuits for nanoscale computation," *International Journal of Nanotechnology and Molecular Computation*, vol. 1, no. 4, pp. 39–57, 2010.

[2] K. P. Parker and E. J. McCluskey, "Probabilistic treatment of general combinational networks," *IEEE Transactions on Computers*, vol. 24, no. 6, pp. 668–670, 1975.

[3] J. Savir, G. Ditlow, and P. H. Bardell, "Random pattern testability," *IEEE Transactions on Computers*, vol. 33, pp. 79–90, 1984.

[4] J.-J. Liou, K.-T. Cheng, S. Kundu, and A. Krstic, "Fast statistical timing analysis by probabilistic event propagation," in *Design Automation Conference*, 2001, pp. 661–666.

[5] R. Marculescu, D. Marculescu, and M. Pedram, "Logic level power estimation considering spatiotemporal correlations," in *International Conference on Computer-Aided Design*, 1994, pp. 294–299.

[6] W. Qian, M. D. Riedel, and I. Rosenberg, "Uniform approximation and Bernstein polynomials with coefficients in the unit interval," University of Minnesota, Tech. Rep., 2010, submitted to *European Journal of Combinatorics*. [Online]. Available: http://cctbio.ece.umn.edu/wiki/index.php/Research

[7] W. Qian and M. D. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," in *Design Automation Conference*, 2008, pp. 648–653.

[8] B. Dufort and G. W. Roberts, *Analog Test Signal Generation Using Periodic $\Sigma\Delta$-Encoded Data Streams*. Kluwer Academic, 2000.

[9] W. Qian, M. D. Riedel, K. Barzagan, and D. Lilja, "The synthesis of combinational logic to generate probabilities," in *International Conference on Computer-Aided Design*, 2009, pp. 367–374.

[10] W. Qian and M. D. Riedel, "Two-level logic synthesis for probabilistic computation," in *International Workshop on Logic and Synthesis*, 2010, pp. 95–102.

[11] 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 (to appear)*, 2010.

[12] S. T. Ribeiro, "Random-pulse machines," *IEEE Transactions on Electronic Computers*, vol. 16, no. 3, pp. 261–276, 1967.

[13] B. Gaines, "Stochastic computing systems," in *Advances in Information Systems Science*. Plenum, 1969, vol. 2, ch. 2, pp. 37–172.

[14] C. Janer, J. Quero, J. Ortega, and L. Franquelo, "Fully parallel stochastic computation architecture," *IEEE Transactions on Signal Processing*, vol. 44, no. 8, pp. 2110–2117, 1996.

[15] S. Toral, J. Quero, and L. Franquelo, "Stochastic pulse coded arithmetic," in *International Symposium on Circuits and Systems*, vol. 1, 2000, pp. 599–602.

[16] B. Brown and H. Card, "Stochastic neural computation I: Computational elements," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 891–905, 2001.

[17] C. Bishop, *Neural Networks for Patten Recognition*. Clarendon Press, 1995.

[18] S. Deiss, R. Douglas, and A. Whatley, "A pulse coded communications infrastructure for neuromorphic systems," in *Pulsed Neural Networks*. MIT Press, 1999, ch. 6.

[19] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*. Princeton University Press, 1956, pp. 43–98.

[20] S. Cheemalavagu, P. Korkmaz, K. Palem, B. Akgul, and L. Chakrapani, "A probabilistic CMOS switch and its realization by exploiting noise," in *IFIP International Conference on VLSI*, 2005, pp. 535–541.

[21] L. Chakrapani, P. Korkmaz, B. Akgul, and K. Palem, "Probabilistic system-on-a-chip architecture," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, pp. 1–28, 2007.

[22] S. Narayanan, J. Sartori, R. Kumar, and D. Jones, "Scalable stochastic processors," in *Design, Automation and Test in Europe*, 2010, pp. 335–338.

[23] P. S. Duggirala, S. Mitra, R. Kumar, and D. Glazeski, "On the theory of stochastic processors," in *International Conference on Quantitative Evaluation of Systems*, 2010.

[24] G. Lorentz, *Bernstein Polynomials*. University of Toronto Press, 1953.

[25] R. Farouki and V. Rajan, "On the numerical condition of polynomials in Bernstein form," *Computer Aided Geometric Design*, vol. 4, no. 3, pp. 191–216, 1987.

[26] "ISCAS-85 C6288 16x16 multiplier." [Online]. Available: http://www.eecs.umich.edu/ jhayes/iscas/c6288.html

[27] D. Lee, R. Cheung, and J. Villasenor, "A flexible architecture for precise gamma correction," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 4, pp. 474–478, 2007.