Design of Sequential Stochastic Computing Systems

by

Paishun Ting

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2019

Doctoral Committee:

Professor John P. Hayes, Chair
Professor Scott Mahlke
Professor Karem A. Sakallah
Professor Zhengya Zhang

Paishun Ting

paishun@umich.edu

ORCID iD: 0000-0002-5675-3816

DEDICATION

This dissertation is dedicated to my family and my friends.

ACKNOWLEDGEMENTS

I am truly grateful for all the help and advice I have received during my pursuit of the Ph.D. degree. I am specially thankful to my Ph.D. advisor Professor John P. Hayes for his guidance and kindness. I was able to learn so much from him, including but not limited to, identifying and solving critical research problems, writing technical documents and presenting scientific results. It has been a pleasure and an honor to be able to work with him for the past few years. I would also like to thank my Ph.D. committee members Professor Scott Mahlke, Professor Karem A. Sakallah and Professor Zhengya Zhang for their constructive suggestions and their tremendous help that made my research and dissertation better.

I am thankful to my former and current colleagues at the University of Michigan, including Dr. Armin Alaghi, Dr. Te-Hsuan Chen, I-Che Chen, William Sullivan and Timothy Baker, for their helpful discussions on research problems and for their useful advice on my career path. I would also like to express my gratefulness to my friends who supported me through these challenging but fruitful years. The good and tough moments we had together are something that will never be forgotten.

Finally, I would like to thank my family, especially my parents and my brother, for their company and their encouragement. Their unconditional support has been my emotional and spiritual anchor during the toughest times in my life. I must have been the luckiest person in the world to be a member of this wonderful family. Thank you Dad, Mom, and my younger brother.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADDIE	Adaptive Digital Element
BSC	Binary-to-stochastic Converter
CEASE	Constant Elimination Algorithm for Suppression of Errors
DFF	D Flip-Flop
DNN	Deep Neural Network
FSM	Finite-State Machine
LDPC	Low-Density Parity-Check
LFSR	Linear Feedback Shift Register
MIND	Algorithm for Minimizing Delay in Maxflow
MOUSE	Monte-Carlo Optimization Using Stochastic Equivalence
MSD	Mean Squared Deviation
MSE	Mean Squared Error
NN	Neural Network
RFE	Random Fluctuation Error
RIC	Randomness Injection Circuit
RNS	Random Number Source
SBC	Stochastic-to-binary Converter
SC	Stochastic Computing
SCC	Stochastic Computing Correlation

SN	Stochastic Number
ReSC	Reconfigurable Stochastic Computing Architecture
VAIL	Valid Isolator Placement Algorithm Based on Integer Linear Programming
VIP	Valid Isolator Placement

LIST OF SYMBOLS

Symbol	Meaning
$+$	The plus sign denotes OR in logic expressions and add in arithmetic expressions.
$\cdot$	The dot sign or juxtaposition denotes AND in logic expressions, and multiply or dot product in arithmetic expressions.
$\mathscr{D}$	An isolator placement.
N	The length of a stochastic number.
N_1	The number of 1s in a stochastic number.
$\mathbf{X}$	An uppercase boldface letter denotes a stochastic number represented by a randomized bit-stream.
$\mathscr{X}$	A set of stochastic numbers
$\mathbf{X}(k)$	A *k*-cycle delayed version of the stochastic number **X**.
$\mathbf{X}^{(t)}$	The *i*-th bit of the stochastic number **X**.
$\mathbf{X}^{(i:j)}$	The segment of the stochastic number **X** containing bits from the *i*-th bit to the *j*-th of **X**.
X	An uppercase italic letter denotes the exact value of the stochastic number **X**.
$\hat{X}^{(N)}$	The fraction of 1s in the first *N* bits of the stochastic number **X**.
x	A lowercase letter denotes a line in a circuit or a Boolean variable.
$Var(Z)$	Variance of a random variable *Z*.

ABSTRACT

Stochastic computing (SC) is an unconventional technique that has recently re-emerged as an attractive design alternative to conventional binary computing (BC). SC represents data using probabilistic bit-streams, whose value is associated with their frequency of 1s and 0s. Such a highly unusual data format allows arithmetic operations, including multiplication and addition, to be implemented with extremely simple circuits, hinting at exceptionally low power and small size. These SC features show great promise in tasks like image processing and machine learning that can benefit from massive parallelism. However, SC's inherently random nature also presents several design challenges, such as long computation time, unpredictable SC behavior, and errors induced by random fluctuation and correlation. These issues have limited the use of SC to applications that can tolerate errors or only require low precision. This dissertation begins by investigating SC's major error sources. We observe that many SC designs use internal constant signals that we show to be a major contributor to random fluctuation errors. We then devise a new design method that eliminates these errors by transferring the role of constant signals to memory components within sequential stochastic circuits. We also examine methods to remove correlation errors in SC, focusing on a decorrelation method called isolation that works by inserting sequential delay elements. As isolation has been used in non-optimal ways previously, we propose a novel isolation-based method that introduces minimal hardware overhead. Noting that both of the foregoing error-elimination methods rely on sequential elements, we next study the overall role of sequential components in SC. We identify several new classes of sequential stochastic circuits, and analyze their properties in depth. This leads to a circuit optimization method that exploits stochastically equivalent state-transitions in sequential circuits. Many SC systems are actually a hybrid of BC and SC features. Connecting these two types of computation technologies usually introduces considerable hardware and latency overhead. We examine the BC-SC interface, and propose a new hybrid architecture that can replace some such interfaces without introducing more latency than necessary. Finally, we explore a new opportunity in SC by viewing SC-style randomness as a design resource instead of an error source. We further propose a sequential element that can inject a precise amount of randomness into SC computation, which benefits applications that need randomness, such as image dithering and neural-network hardening. The dissertation concludes with a discussion of some promising directions for future SC research.

CHAPTER 1
Introduction

Stochastic computing (SC) is an unconventional computational approach whose origins date back to the 1960s [29][32]. It computes with pseudo-random bit-streams called stochastic numbers (SNs) that have numerical values defined by the probability of 1 appearing anywhere in them. The trademark features of SC are extremely low power, low area cost, and high tolerance of soft errors, all of which are enabled by its uncommon probabilistic data representation. SC has received substantial attention recently, as many rapidly growing applications, including machine learning, wearable devices and biomedical implants, can benefit from SC's distinct advantages. However, SC also presents many design challenges, most of which stem from its poorly understood theory. This chapter begins by introducing SC and its potential applications. It then provides an overview of the major research problems in SC that motivate our work.

1.1 Motivation

As factors like clock speed and voltage scaling approach their limits, recent trends in integrated circuit design have been to exploit parallelism, improve energy efficiency, and accommodate soft errors by means of heterogeneous architectures, dedicated accelerator units, fault-tolerant devices, among others. Stochastic computing (SC) is a highly unconventional design technique that stands out as very promising for those specialized tasks that have stringent power and area constraints, or require massively parallel operations but can tolerate low precision and accuracy.

In contrast to conventional binary computing (BC) that processes radix-2 numbers, SC computes with (clocked) probabilistic bit-streams called stochastic numbers (SNs), whose numerical value is determined by the fraction of 1s they contain. For example, **X** = 1001, **X** = 0011, and **X** = 10101011 all represent a (unipolar) value of 0.5, as half of their bits are 1s. This highly unusual way of representing data enables several key arithmetic operations to be realized by extremely small circuits. For example, as we will see shortly, stochastic multiplication can be realized with just a single standard AND gate.

Although it originated and underwent early development in the 1960s, SC has until recently received very little attention due to the dominance of conventional BC technology. A major revival of SC occurred around 1990 when SC was successfully applied to implement FGPA-accelerated neural networks (NNs) [14][76], which implement machine-learning (ML) algorithms that require massive but parallelizable arithmetic operations. The application of SC to NNs has continued as new challenges arise, including the introduction of deep neural networks (DNNs), and the trend to providing on-device ML services to mobile devices like smartphones [35]. The past decade has witnessed the success of DNNs as they achieved performance better than humans in a variety of ML tasks. However, DNNs are also notorious for their hunger for resources. On-device ML services must operate at low power to avoid draining a device’s battery. Executing ML algorithms like DNNs on wearable devices is also challenging, even with the aid of dedicated neural units. SC has continued attracting the eye of the research community as it can perform several key tasks for DNNs without consuming excessive power or area [15][62]. Further, as we will explore in this dissertation, SC-based DNNs are automatically more resilient against adversarial attacks, a type of malicious data that can deceive ML algorithms.

Biomedical implants are another area where SC looks very promising. For instance, implantable vision chips for the visually impaired should operate fast enough to provide a patient with real-time vision. This is best achieved with computationally massive but physically tiny per-pixel vision processors that can operate in parallel. On the

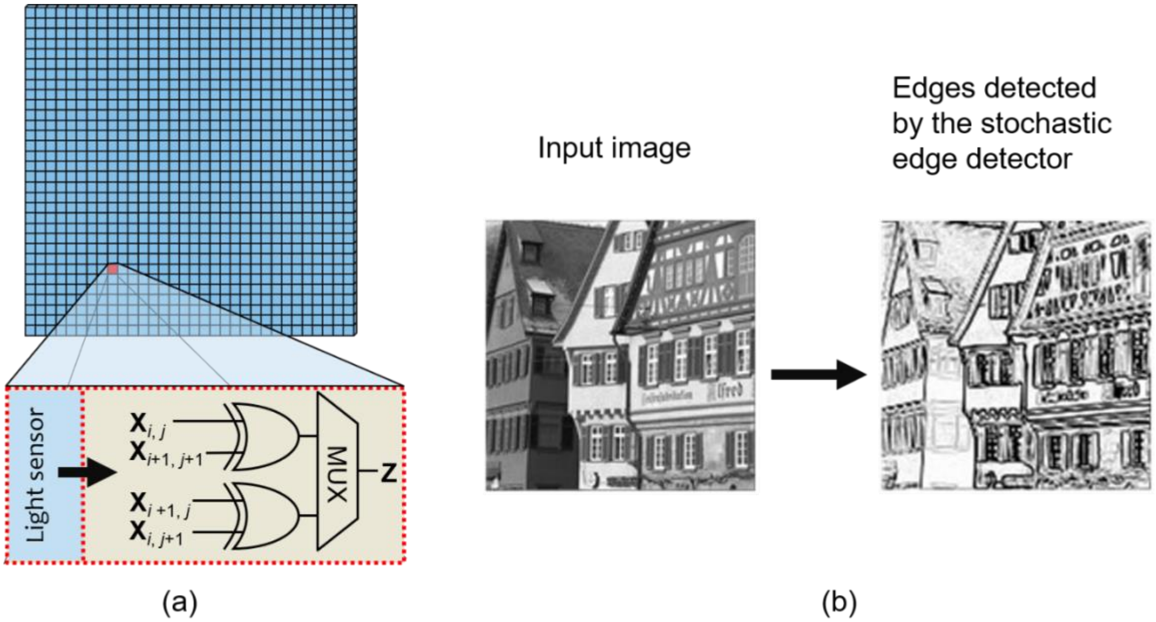


Figure 1.1 (a) Stochastic sensor array for edge detection [8]. (b) An edge image generated by the stochastic edge detector [8].

other hand, vision chips have very strict temperature limits that must be met to prevent human issues from being damaged by excessive heat. Building a BC chip that meets all the preceding requirements is very challenging. On the contrary, an SC-based implementation can easily do so. Figure 1.1*a* shows an SC-based vision chip for edge detection [8], which features per-pixel processors that enable real-time image processing. Each processor takes signals sensed by four adjacent light sensors, and implements the Roberts cross edge detection formula $Z = |X_{i,\ j} - X_{i+1,\ j+1}| + |X_{\ i+1,\ j} - X_{i,\ j+1}|$. As shown in Figure 1.1, the SC-based vision chip can perform high-quality edge detection with each SC processor containing fewer than ten gates, in contrast to a BC implementation which can easily require several hundreds of gates per pixel. This allows SC-based vision chips to operate with low heat output and low power consumption.

Besides the foregoing examples, SC has also been successfully applied to other applications, including digital filters [11][19], error-correcting code decoding [26][77], control systems [56], etc. For instance, SC has been used for decoding state-of-the-art error-correcting codes like low-density parity-check (LDPC) codes [33][68], and has been demonstrated to attain a higher throughput than a comparable BC implementation [49].

Despite the significant progress that has been made in the past few decades, SC still faces several major challenges that limit its practical use. These include immature design methodologies, poorly understood randomness behavior, and complex trade-offs between accuracy, run time, and hardware cost. This dissertation aims to solve some of the preceding problems, with the goal of paving the way for future SC research and applications.

1.2 Stochastic Computing

As noted previously, SC computes with randomized bit-streams called SNs using standard versions of logic circuits, which we refer to as *stochastic circuits*. An SN $\mathbf{X}$ applied to a logic wire x is a (clocked) sequence of N bits $\mathbf{X} = \mathbf{X}^{(1)}\mathbf{X}^{(2)}\mathbf{X}^{(3)}\ldots\ \mathbf{X}^{(N)}$, where $\mathbf{X}^{(t)}$ is the t-th random bit. In *unipolar* SC, $\mathbf{X}$'s value X is the probability of a 1 appearing in $\mathbf{X}$, which we denote as $X = p(\mathbf{X}^{(t)} = 1) = p_{\mathrm{X}}(1)$. This value is fixed for all t and is usually approximated by measuring the fraction of 1s in $\mathbf{X}$, i.e., $\hat{X} = N_1 / N$, where N_1 is the number of 1s in $\mathbf{X}$. The quality with which the measured value $\hat{X}$ approximates the exact value X depends on the bit-stream length N. Roughly speaking, the larger N, the better $\hat{X}$ approximates X.

In SC, an individual numerical value does not have a unique SN representation. For example, $\mathbf{X}$ = 001, $\mathbf{X}$ = 010 and $\mathbf{X}$ = 101000 all represent a (unipolar) SN with a measured value 1/3, but their bit patterns and lengths are different. While probability values are in the range [0, 1], SC can process any real numbers by mapping them to the probability range. In particular, *bipolar* SC handles negative numbers through the mapping $X = 2p_{\mathrm{X}}(1) - 1$, so the SN $\mathbf{X}$ = 001 has a bipolar value $2 \times 1/3 - 1 = -1/3$.

SC's highly unusual data representation enables it to implement numerous key

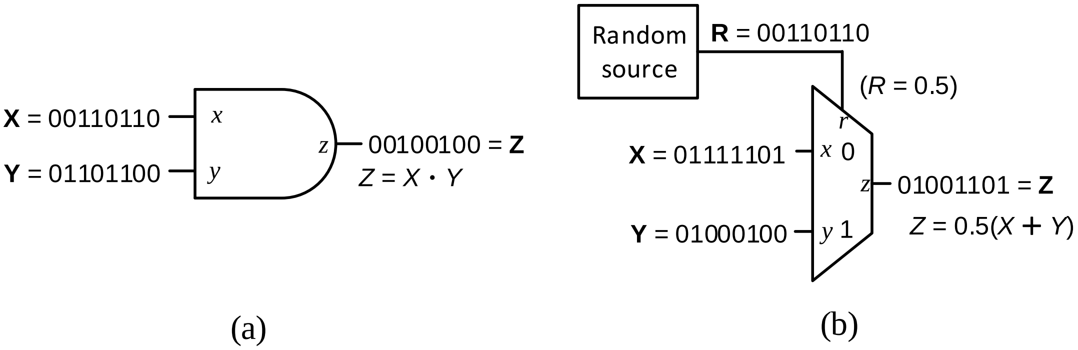


Figure 1.2 Key SC arithmetic circuits: (a) unipolar stochastic multiplier, and (b) stochastic scaled adder.

arithmetic functions, notably multiplication and addition, with very small circuits. For example, the AND gate shown in Figure 1.2*a* performs unipolar stochastic multiplication. This is because the output probability $Z = p_{\mathbf{Z}}(1)$ is equal to the probability that both input bits are 1, which defines the arithmetic operation $p_{\mathbf{X}}(1)p_{\mathbf{Y}}(1) = XY$. However, for a stochastic multiplier to perform accurately, its two inputs must be statistically independent or uncorrelated; this is an SC-specific design problem that will be examined in depth later. The standard implementation of a stochastic scaled adder is given in Figure 1.2*b*; it is built around a two-way multiplexer (MUX) that computes the Boolean operation $z = xr' + yr$. It is noteworthy that the adder has an internal random source that generates an SN **R** with a constant value $R = 0.5$. In each clock cycle, this constant SN selects a bit from **X** and **Y** to send to the output line. Consequently, the probability value of **Z** is the average, or the scaled sum, of X and Y, i.e., $Z = 0.5(X + Y)$. This special internal SN **R** effectively scales down the output value so that Z is in the valid probability range [0, 1].

One very appealing feature of SC is that, while it computes with analog probabilities, it is compatible with standard digital logic circuits. SC-formatted data can be readily ported to and from BC circuits via stochastic-to-binary converters (SBCs) and binary-to-stochastic converters (BSCs), respectively. Figure 1.3*a* shows a BSC built around a comparator that transforms a binary number B_X to the corresponding SN. In each

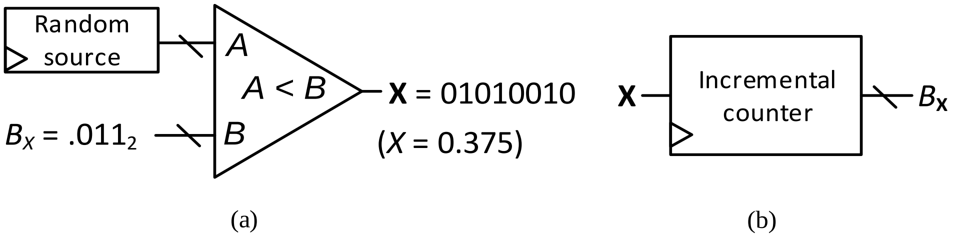


Figure 1.3 Data-format converters: (a) binary-to-stochastic converter (BSC), and (b) stochastic-to-binary converter (SBC).

clock cycle, the internal random number source (RNS) generates a uniformly distributed random number, against which B_X is compared. The probability of a 1 appearing in the output line is therefore equal to B_X. To convert an SN to a binary number, it suffices to count the number of 1s in that bit-stream. Hence an incremental counter can implement an SBC; see Figure 1.3*b*.

SC is also highly tolerant of soft errors. An occasion bit flip in an SN does not have a catastrophic impact on that SN's value. For instance, consider a 16-bit SN **X** = 0110011100101100, which has the measured value 8/16 = 0.5. Suppose a bit-flip error occurs in **X**'s first bit, changing it to **X**' = 1110011100101100. The measured value of *X*' is 9/16 = 0.5625, which is still a good approximation to its original value 0.5. Further, 0-to-1 bit flips and 1-to-0 bit flips tend to cancel out each other. SC's fault tolerance makes it an appealing design candidate for devices that are error prone and applications that operate in extreme environments [21].

SC also has a property referred to as progressive precision, meaning any initial segment of an SN provides a rough approximation of that SN's value. For example, consider again the 16-bit 0.5-valued SN **X** = 0110011100101100. The first 8-bit segment of **X** is 01100111 with a measured value of 5/8 = 0.625, which is a reasonably good approximation to the original value 0.5. Progressive precision allows an SC computation to dynamically trade accuracy for shorter latency. This is particularly helpful in applications like neural networks, which can stop computation as soon as a satisfactory result becomes available.

1.3 Research Challenges

While SC has many appealing features as discussed earlier, it also presents many challenges, most of which stem from SC's inherent randomness. It computes with probabilities, so randomness is required to drive the operation of most stochastic circuits. Insufficient randomness can easily lead to correlation between SNs, a major accuracy-reducing factor in most stochastic circuits. On the other hand, too much randomness can result in random fluctuation errors (RFEs), which can render an SC computation noisy and hence less useful. Complex trade-offs between accuracy and compute time also make it hard to design efficient SC systems. Sequential elements are usually needed to tackle the preceding problems. However, their stochastic behavior is poorly understood; this adds extra design complexity and results in non-optimal ad-hoc circuits. Last but not least, applications that explicitly need randomness can take advantage of SC's intrinsic random nature. However, SC randomness is hard to control, making it difficult to exploit as a practical design resource.

Many stochastic circuits are designed to work with independent or uncorrelated input SNs. For example, the multiplier in Figure 1.2*a* requires its two inputs to be independent in order to function properly. In such a case, correlation can directly alter the function of a stochastic circuit, leading to undesirable correlation errors. Consider, for instance, implementing a stochastic squarer that computes $Z = X^2$. It may be tempting to do so by feeding the stochastic multiplier with two identical, and therefore maximally correlated, SNs. This, however, leads to an output bit-stream that is also identical to the input bit-streams, so $Z = X \neq X^2$, as shown in Figure 1.4*a*. Correlation of this kind must usually be removed through a process termed *decorrelation* in this dissertation. An efficient but ad hoc decorrelation method proposed by Gaines, one of the inventors of SC, is to use delay elements he called isolators to intentionally introduce delays into SNs, so that the correlated bits in these SNs do not interact with each other [29]. Applying this decorrelation method to the squarer design results in the circuit in Figure 1.4*b*, which is a multiplier with

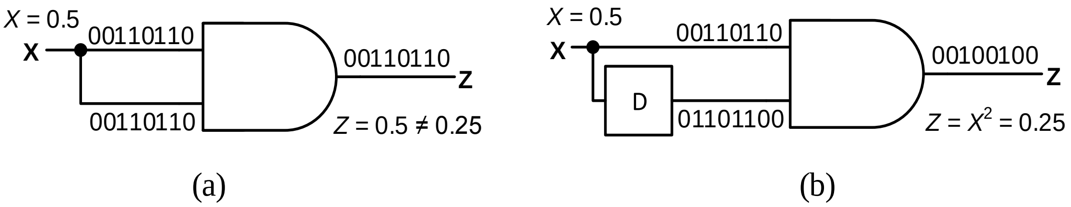


Figure 1.4 Stochastic squarer: (a) incorrect implementation due to input correlation, and (b) correct implementation enabled by an isolator *D*.

an isolator (implemented by a D flip-flop) inserted in one of its input lines. The foregoing discussion suggests that correlation control is essential to the accuracy of stochastic circuits. However, correlation problems can be complex, and decorrelation methods of the above kind must be used correctly and efficiently to remove undesired correlations without introducing excessive hardware overhead, a problem that will be examined in depth in Chapter 3.

Another major accuracy-reducing factor in SC is random fluctuation errors (RFEs) which occur due to SC's randomized and probabilistic data representation. For example, an 8-bit SN **X** having a probability value $X = 0.5$ may very likely have exactly four 1s and four 0s in it, such as **X** = 01001101. However, SNs that only approximate the exact value, e.g., **X** = 01100010 or **X** = 10111100, may also occur due to random fluctuation. In applications like neural networks, RFEs can easily accumulate from layer to layer, rendering the final outputs unacceptably noisy. RFEs can be effectively reduced by lengthening SNs. However, this comes at the cost of long computation time. A recent trend to tackle random fluctuation is to reduce randomness or introduce some amount of determinism into SC. An example of this approach is the removal of SNs with constant probability values. Consider the scaled adder in Figure 1.2*b*, which uses an internal constant SN **R** to randomly select one of its input bits to send to the output. Figure 1.5*a* shows a recent ad-hoc design for scaled addition that removes the need for the extra internal SN. This scaled adder computes $Z = 0.5(X + Y)$ by releasing exactly a single 1 to the output line whenever it receives two 1s, and therefore this adder fluctuates much less than the

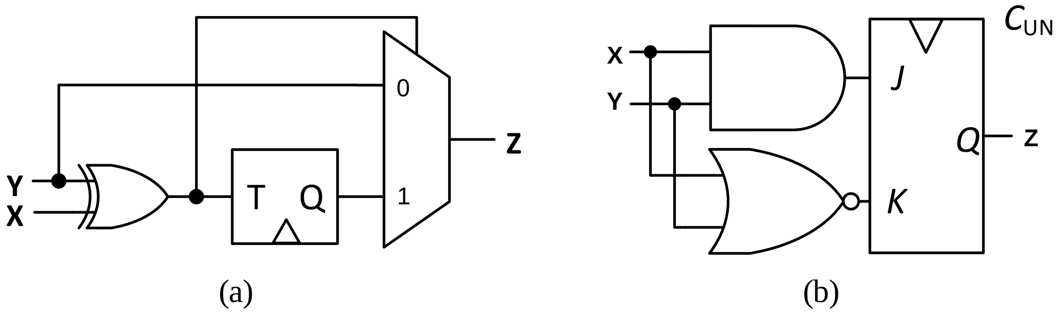


Figure 1.5 (a) A constant-free stochastic scaled adder [48]. (b) A design for the update node used in the stochastic LDPC code decoder [33].

standard design of Figure 1.2*b*. This constant-free adder is also a major factor in the success of the NN implementation in [48]. Extending constant removal to other SC designs, however, is a non-trivial problem which we will study in Chapter 2.

Sequential elements play an indispensable part in SC, as suggested by the fact that both decorrelation and constant removal call for sequential solutions. Sequential components are also needed to design *special* arithmetic functions. For example, Figure 1.5*b* shows an SC implementation for the update node used in LDPC code decoders [33][68]. It is a sequential design built around a JK flip-flop, and computes the function $F(X,Y) = \frac{XY}{XY + (1-X)(1-Y)}$ which cannot be realized exactly by combinational methods. Despite the prominence of sequential elements in SC, the theory underlying sequential SC designs is still poorly understood, restricting their use to relatively few circuit types, notably up/down counters [29][52]. Further, it is noteworthy that interfaces (e.g., SBCs and BSCs) connecting SC and BC components in a hybrid system are also sequential, and are often the most expensive part of a system [65]. However, their cost implications are frequently overlooked, leading to unoptimized interface designs with excessive latency or hardware overhead. These problems will be examined in Chapters 4 and 5.

Finally, carefully controlled SC randomness can be very useful in applications that need randomness. However, most previous work has focused on reducing or removing SC-style randomness for accuracy considerations. Consequently, methods for precisely managing or controlling SC randomness in a way that benefits applications requiring randomness are greatly lacking. This problem is investigated in Chapter 6.

1.4 Dissertation Outline

SC has emerged as a very promising computing technology as demonstrated recently in several SC-based applications. However, many SC-specific design issues including random fluctuation and correlation errors, along with complex trade-off among area cost, latency and accuracy, make designing efficient SC circuits that meet practical constraints very difficult. This dissertation investigates and analyzes the preceding problems in depth, and proposes several practical solutions, with a focus on sequential design issues. It is divided into three main parts: (1) treatment of random fluctuation and correlation, (2) analysis of sequential stochastic circuits and SC-BC hybrid systems, and (3) exploiting SC randomness as a resource. Specifically, Chapter 2 and 3 propose sequential methods to alleviate random fluctuation and correlation errors. Chapters 4 and 5 examine the role of sequential elements in SC and design considerations for complex SC-BC hybrid systems. Chapter 6 explores a new design opportunity enabled by viewing SC's randomness as a resource.

Chapter 2 discusses the impact of random fluctuation errors (RFEs) on stochastic circuits, and identifies stochastic constants as a major cause for RFEs that has been completely overlooked in the previous literature. A constant-elimination method CEASE is proposed to remove all the constant-induced random fluctuation by transferring their role to memory elements in a way that results in maximal reduction of RFEs. Chapter 3 analyzes undesired correlation in SC designs, and reviews ways to quantify and manage correlation. It then proposes an isolation-based decorrelation method called VAIL that can eliminate undesired correlation in a stochastic circuit. It does so by systematically inserting sequential delay elements or isolators in a way that minimizes hardware overhead.

In Chapter 4, we review the main approaches to sequential SC design , and identify several key classes of sequential designs. We analyze the properties of these classes and provide a randomized algorithm MOUSE for optimizing sequential stochastic circuits. Because most so-called “SC-based” circuits are in fact a hybrid design that combine SC and BC components, we dedicate Chapter 5 to analyzing several key design aspects of hybrid systems, focusing on the area and latency penalty induced by SC-BC interfaces. This chapter also presents a hybrid design that can significantly reduce the latency penalty without compromising on other design metrics like accuracy and area cost.

Chapter 6, in contrast to previous negative views on SC randomness as an accuracy-reducing factor, shows that carefully controlled randomness can play a positive role in applications that need randomness. The chapter then proposes a method to precisely control the amount of such randomness to fit the needs of specific applications. It also demonstrates the usefulness of SC randomness in two applications, namely image dithering and neural-network hardening. Finally, Chapter 7 concludes the dissertation and discusses some directions for future research.

CHAPTER 2
Random Fluctuation

As discussed in Chapter 1, SC tends to have low accuracy due to factors like random fluctuation and correlation. This chapter focuses on random fluctuation errors (RFEs), a major accuracy-reducing factor introduced by SC's inherent randomness. RFEs can be mitigated by increasing the length of SNs. This, however, often leads to excessive computation time and energy consumption. In this chapter, we first observe that many SC designs heavily rely on constant SNs, which contribute significantly to RFEs. We then investigate the role of constant inputs in SC, and propose a systematic algorithm CEASE to eliminate constant-induced RFEs by inserting memory elements into the target circuits. The resulting sequential designs are optimal in terms of the amount of RFE reduction. This chapter also presents case studies involving several representative stochastic circuits. The material in this chapter has been published in [72] (which received a Best Paper award) and [74].

2.1 Errors Induced by Randomness

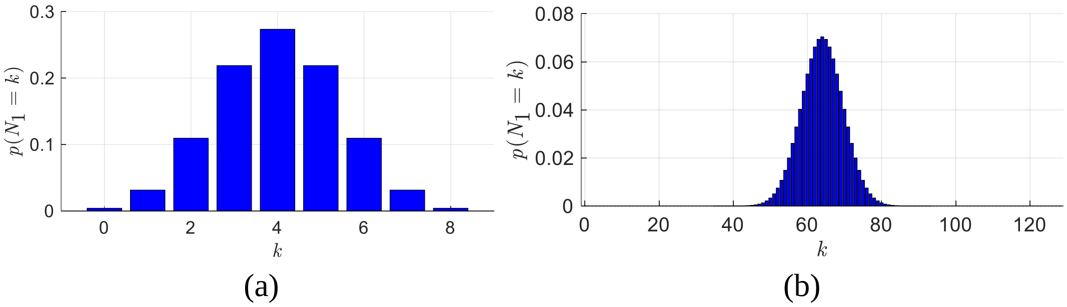
Randomness plays an essential role in SC, as it is used to generate the SNs that drive the operation of stochastic circuits. Random fluctuation thus occurs naturally when dealing with SNs. Consider the analogy of tossing a fair coin 8 times. While the outcome is most likely to be 4 heads and 4 tails, other outcomes like 5 heads and 3 tails are also likely. This is exactly what happens when we generate an 8-bit SN **X** with a desired value of $X = 0.5$. The measured value of **X** is most likely to be 4/8, but 3/8 or 5/8 are also quite likely. In fact, given a Bernoulli SN **X** whose bits are independently generated, N_1, the number of 1s in **X**, follows a binomial distribution thus:

$$p(N_1 = k) = \binom{N}{k} X^k (1 - X)^{(N-k)} \tag{2.1}$$

Figure 2.1*a* depicts this distribution with $N = 8$ and $X = 0.5$. It plots $p(N_1 = k)$, the probability of an SN having k 1s, against k. As expected, the event $N_1 = 4$ is the most probable. However, $N_1 = 3$ and $N_1 = 5$ are also likely.

Random fluctuation errors (RFEs) in SC are caused by randomness of the foregoing type. An intuitive but effective way to reduce RFEs is to increase the SN length N. Figure 2.1*b* shows the distribution of N_1 in a 128-bit SN with value 0.5. The most likely value for N_1 is 64, implying that a measured value 0.5 is the most probable. Further, compared to the distribution of an 8-bit SN in Figure 2.1*a*, Figure 2.1*b* is dispersed much less, meaning that with an 128-bit SN, it is far less likely for a measured value to deviate from the exact value by a large margin. This suggests that increasing SN length indeed reduces RFEs. However, long SNs lead to long run time and hence high energy consumption. To mitigate such problems, SC usually trades accuracy for computation time, thereby narrowing the range of applications to which it can be successfully applied.

This chapter describes a novel method to reduce RFEs in SC while maintaining desirable SC features such as error tolerance. It is based on the observation that most SC designs, from the simple scaled adder in Figure 1.2*b* to complex stochastic circuits generated by major synthesis methods like STRAUSS [6] and ReSC [65], heavily rely on



(a) (b)

Figure 2.1 Distribution of the number of 1s in SNs with value 0.5 for (a) $N = 8$, and (b) $N = 128$, respectively.

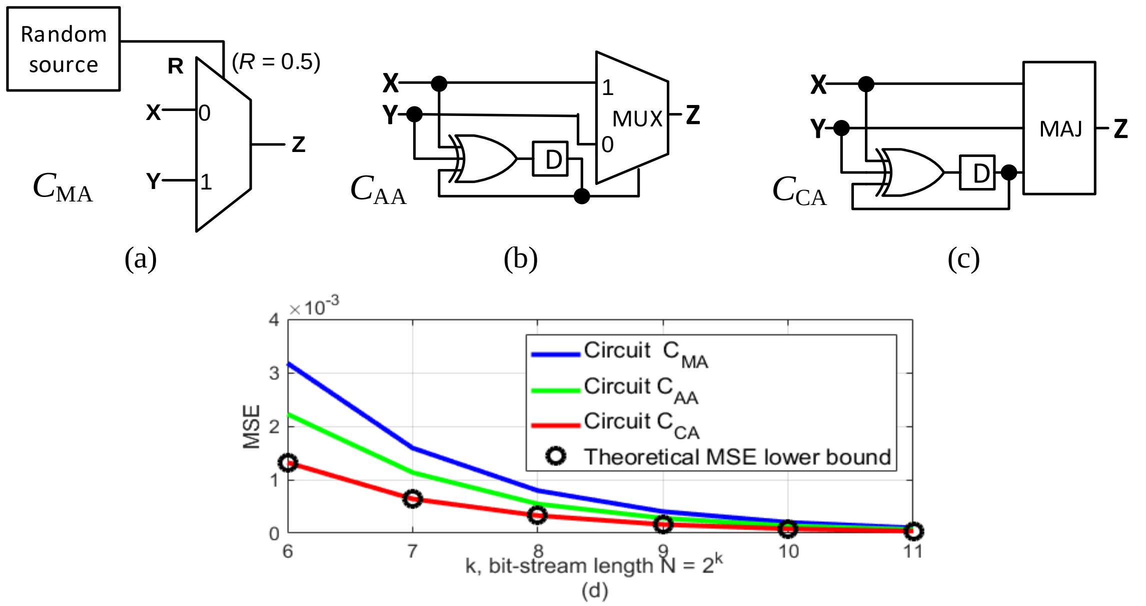


Figure 2.2 Three stochastic implementations of scaled addition: (a) conventional MUX-based design C_{MA} with a constant input **R** = 0.5; (b) ad hoc sequential design C_{AA} with no constant input [72]; (c) sequential design C_{CA} produced by CEASE; (d) error comparison of the three designs.

the use of constant SNs to achieve good function approximations. These constant SNs not only increase hardware overhead due to their need for random sources but, as will be clear shortly, also turn out to be a significant source of RFEs.

In this chapter, we show for the first time that it is possible to remove all RFE-inducing input constants by resorting to a new class of sequential SC designs. We devise a systematic method, which we call ***C**onstant **E**limination **A**lgorithm for **S**uppression of **E**rrors* (*CEASE*) [72][74] for constant removal. While a function may have various constant-free circuit implementations, CEASE-generated circuits provide a guarantee of optimality on RFE reduction. Figure 2.2*a* repeats the conventional scaled adder previously shown in Figure 1.2*b*. It uses a single constant **R** to scale the output back to the probability range. Figure 2.2*b-c* depict two sequential scaled adder designs C_{AA} and C_{CA} after eliminating **R**; the former was designed in ad hoc fashion [72], while the latter is based on

CEASE. Figure 2.2*d* plots the RFEs of all three scaled adders in Figure 2.2*a-c* against bit-stream length N, where RFEs are measured by the (sampled) mean squared error (MSE), which will be defined shortly. It can be seen that the CEASE-based design C_{CA} is the most accurate. Furthermore, as we will demonstrate, C_{CA} meets the theoretical bound of the lowest RFE level indicated by the small circles.

2.2 The Role of Constants

This section begins by reviewing relevant concepts of stochastic circuits, focusing on the functions they implement. It then investigates the role of constants used in SC designs. As briefly discussed in Chapter 1, a combinational stochastic circuit C computes by transforming a set of input SNs based on the SNs' probabilities and f, the Boolean function C implements.

Example 2.1: Consider again the AND-gate multiplier given in Figure 1.2*a*. This AND gate implements the Boolean function $z = f_{\mathrm{AND}}(x, y) = xy$, meaning that $f_{\mathrm{AND}}(x, y)$ always outputs a 0 bit except for the input $xy = 11$, in which case $f_{\mathrm{AND}}(x, y) = 1$. The probability that the AND gate receives the input pattern $xy = 11$ is obviously $p_{\mathbf{XY}}(1, 1)$. Hence, the probability that a 1 appears in the output line z is $Z = f_{\mathrm{AND}}(1, 1)p_{\mathbf{XY}}(1, 1) = p_{\mathbf{XY}}(1, 1)$. For independent bit-streams **X** and **Y**, $Z = p_{\mathbf{XY}}(1, 1) = p_{\mathbf{X}}(1)p_{\mathbf{Y}}(1) = XY$, which is indeed the arithmetic or stochastic product of the two input values X and Y. □

In the preceding example, we can write the stochastic function computed by the AND gate as $Z = f_{\mathrm{AND}}(0, 0)p_{\mathbf{XY}}(0, 0) + f_{\mathrm{AND}}(0, 1)p_{\mathbf{XY}}(0, 1) + f_{\mathrm{AND}}(1, 0)p_{\mathbf{XY}}(1, 0) + f_{\mathrm{AND}}(1, 1)p_{\mathbf{XY}}(1, 1) = \sum_b f(b)p_{\mathcal{X}}(b)$, where b denotes the value of a 2-bit input vector. In general, for an n-input, single-output combinational circuit C that implements the Boolean function $z = f(x_1, x_2, \ldots, x_n)$, the stochastic function F that C implements depends on both f and the joint probability distribution $p_{\mathbf{X}_1 \cdots \mathbf{X}_n}$ of input SNs $\mathcal{X} = \{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n\}$, where $\mathbf{X}_i$ is the SN applied to C's input line x_i. Specifically, $F = F(f, p_{\mathcal{X}})$ takes the following form:

$$p_{\mathbf{Z}} = F(f, p_{\mathcal{X}}) = \sum_b f(b)p_{\mathcal{X}}(b) \tag{2.2}$$

where $p_{\mathcal{X}}(b)$ is the joint probability distribution of the input SNs, and the summation is over all combinations of the n-bit input vector b. Equation (2.2) indicates that a stochastic function is a linear combination of the probability terms $p_{\mathcal{X}}(b)$ with coefficients $f(b)$ that take 0-1 values.

Example 2.2: The two-way multiplexer in Figure 2.2*a* has three input SNs $\mathcal{X} = \{\mathbf{X}, \mathbf{Y}, \mathbf{R}\}$ applied to its inputs x, y, r. It implements the Boolean function $f_{\mathrm{MUX}}(x, y, r)$, which outputs a 0 bit except when $f_{\mathrm{MUX}}(1, 0, 0) = f_{\mathrm{MUX}}(0, 1, 1) = f_{\mathrm{MUX}}(1, 1 ,0) = f_{\mathrm{MUX}}(1, 1, 1) = 1$. The probability that the circuit's output is 1 is the probability that one of the input patterns 100, 110, 011 or 111 occurs. Applying Equation (2.2), we obtain

$$Z = \textstyle\sum_b f_{\mathrm{MUX}}(b)p_{\mathcal{X}}(b) = p_{\mathcal{X}}(1, 0, 0) + p_{\mathcal{X}}(1, 1, 0) + p_{\mathcal{X}}(0, 1, 1) + p_{\mathcal{X}}(1, 1, 1) \qquad (2.3)$$

which is a sum of binary-weighted probability terms that defines the stochastic function of a multiplexer. □

Many stochastic circuits, including those synthesized by STRAUSS and ReSC, heavily use constant SNs to define both the target function's value and its precision. Constants in a stochastic circuit refer to internal ancillary SNs that have a fixed value not controllable by the circuit's user. For instance, the scaled adder in Example **2.2** uses an internal SN **R** with value 0.5 to scale the output down to the probability range [0, 1]. Here **R** is a constant generated by a random source not controllable by the user of the circuit; the user can only control the values of the variable SNs **X** and **Y**. In such cases, we can separate the input SNs $\mathcal{X}$ into two disjoint subsets $\mathcal{X} = \{\mathcal{X}_{\mathrm{V}}, \mathcal{X}_{\mathrm{C}}\}$, where $\mathcal{X}_{\mathrm{V}}$ and $\mathcal{X}_{\mathrm{C}}$ denote variables and constants, respectively [23]. A stochastic function of $p_{\mathcal{X}_{\mathrm{V}}}$ can then be derived from Equation (2.2) by replacing the $\mathcal{X}_{\mathrm{C}}$ with appropriate constant values, as the following example illustrates.

Example 2.2 (contd.): Returning to the MUX-based adder C_{MA} in Figure 2.2*a*, here we demonstrate how to describe the stochastic behavior of the adder as a function of user-supplied variable SNs. Let $\mathcal{X} = \{\mathcal{X}_{\mathrm{V}}, \mathcal{X}_{\mathrm{C}}\}$, where $\mathcal{X}_{\mathrm{V}} = \{\mathbf{X}, \mathbf{Y}\}$ is the set of variable SNs and $\mathcal{X}_\text{C} = \{\mathbf{R}\}$ with $R = 0.5$ is a singleton constant SN. If **R** is independent of $\mathcal{X}_\text{V}$, then on substituting $p_\mathbf{R}(1) = R = 0.5$ into Equation (2.3), we get

$$\begin{aligned} Z &= 0.5p_{\mathcal{X}_\text{V}}(1,0) + 0.5p_{\mathcal{X}_\text{V}}(0,1) + 0.5p_{\mathcal{X}_\text{V}}(1,1) + 0.5p_{\mathcal{X}_\text{V}}(1,1) \\ &= 0.5[p_{\mathcal{X}_\text{V}}(1,0) + p_{\mathcal{X}_\text{V}}(0,1) + p_{\mathcal{X}_\text{V}}(1,1) + p_{\mathcal{X}_\text{V}}(1,1)] \\ &= 0.5[p_\mathbf{X}(1) + p_\mathbf{Y}(1)] = 0.5(X + Y) \end{aligned} \tag{2.4}$$

which is indeed the expected scaled addition function of the variables X and Y. □

Looking more closely at Equation (2.4), we can see that it is a linear combination of probability terms with *non-binary coefficients*, in contrast to Equation (2.3) and the more general Equation (2.2) which allow only the binary coefficients 0 and 1 for their probability terms. This suggests that constant SNs enable a stochastic function to have coefficients that are any rational numbers in the range [0, 1]. The following theorem generalizes this observation.

Theorem 2.1: The stochastic function realized by a combinational circuit with input SNs $\mathcal{X} = \{\mathcal{X}_\text{V}, \mathcal{X}_\text{C}\}$ is

$$Z = F(p_{\mathcal{X}_\text{V}}) = \sum\nolimits_{b_\text{V}}\left[g(b_\text{V})p_{\mathcal{X}_\text{V}}(b_\text{V})\right] \tag{2.5}$$

where the $g(b_\text{V})$s are rational constants in the range [0, 1] that depend implicitly on f and $p_{\mathcal{X}_\text{C}}$. □

A proof of this theorem can be found in Appendix A.1. Theorem **2.1** reveals some interesting facts about the impact of constant inputs on stochastic functions. It implies that, at the expense of extra constant inputs, the class of implementable functions can be greatly broadened. For example, a combinational circuit cannot implement the stochastic function $Z = 0.5(X + Y)$ with just two variable inputs **X** and **Y**, since this function also requires the non-binary coefficient 0.5. This scaled add function is combinationally implementable,

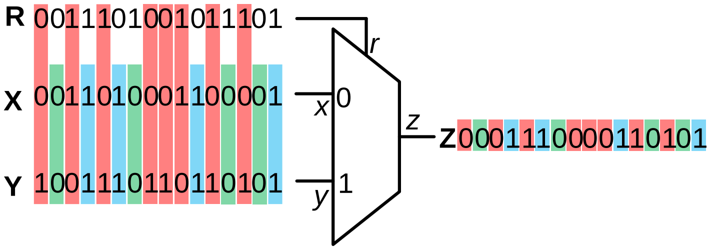


Figure 2.3 MUX-based stochastic scaled adder C_{MA}. On receiving 11 (blue) or 00 (green), the output is 1 or 0, respectively. On receiving 01 or 10 (red), the output is 1 with probability 0.5.

however, as demonstrated in Example **2.2**, by supplying an extra constant input SN **R** of value 0.5. More generally, when a target function is not already in the form of Equation (2.5), it has to be converted to that form by introducing suitable constants. The accuracy with which the resulting function approximates the target function F highly depends on the number of constants used, as can be seen in SC synthesizers like STRAUSS [6] and ReSC [65]. A circuit with good approximation accuracy typically comes with many constant inputs. For instance, the STRAUSS implementation of $Z = \frac{7}{16} - \frac{1}{4}X - \frac{9}{16}X^2$ derived in [6] employs four constant inputs $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{R}_3$ and $\mathbf{R}_4$, each of value 0.5. These constants require costly randomness sources to generate them, and are also a significant source of RFEs that has been long overlooked.

Example 2.2 (contd.): We re-examine the MUX-based adder C_{MA} of Figure 2.2*a*, shown again in Figure 2.3. To implement the scaled addition

$$Z = 0.5(X + Y) \tag{2.6}$$

accurately, the expected number of 1s in **Z** should be half the number of 1s in the input SNs **X** and **Y**. For a given cycle t, when both $\mathbf{X}^{(t)}$ and $\mathbf{Y}^{(t)}$ are 1, the corresponding output bit $\mathbf{Z}^{(t)}$ will be 1; this is exact since a single 1 should be produced in **Z** whenever the circuit receives two 1s. Similarly, when both $\mathbf{X}^{(t)}$ and $\mathbf{Y}^{(t)}$ are 0, $\mathbf{Z}^{(t)}$ will have the exact value 0. When only one of the two inputs is 1, i.e., $\mathbf{X}^{(t)}\mathbf{Y}^{(t)}$ = 10 or 01, Equation (2.6) implies that ideally $\mathbf{Z}^{(t)}$ should be 0.5. However, $\mathbf{Z}^{(t)}$ obviously cannot directly output "0.5" from a logic circuit that computes with 0-1 values. This representational dilemma is effectively resolved by the extra constant SN **R** whose stochastic value 0.5 ensures that, on average, a single 1 is generated in **Z** whenever two copies of 10 or 01 are received. In other words, a 1 and a 0 are expected to be produced in response to every two applications of 10 or 01. This single 1 spread over two cycles thus contributes 1/2 to **Z**. □

The fact that constants produce extra RFEs can be seen from Figure 2.3, where the constant **R** is used to select inputs whenever $\mathbf{X}^{(t)}\mathbf{Y}^{(t)}$ = 10 or 01 (marked in red). Notice here that in the red cycles, four 1s should appear in every eight output bits. However, since this is only true on average, there may be variations due to the probabilistic nature of **R**. In this example, there are only three 1s instead of four in the eight output bits selected by **R**, producing a 1/16 deviation in the output value, and implying that **R** indeed introduces considerable RFEs. The key to eliminating **R** (and the RFEs it introduces) is to enable the circuit to remember every two applications of 10 or 01, which implies changing it from combinational to sequential. This makes it possible for the circuit to output exactly one 1 and one 0 for every two applications of input patterns 01 or 10.

2.3 Constant Elimination

To eliminate constant inputs and their associated errors while keeping their functional benefits, we propose a systematic algorithm called ***C**onstant **E**limination **A**lgorithm for **S**uppression of **E**rrors* (*CEASE*) [72][74]. CEASE transforms a target combinational function into an equivalent stochastic finite-state machine (FSM) with no constant inputs and with reduced RFEs. CEASE also offers a guarantee of optimality on RFE reduction, thereby providing a maximum improvement in accuracy. The idea behind CEASE is to introduce memory elements to count and remember non-binary values; the resulting FSM accumulates such values to be output later. When an accumulated value exceeds one, a 1 is sent to the output.

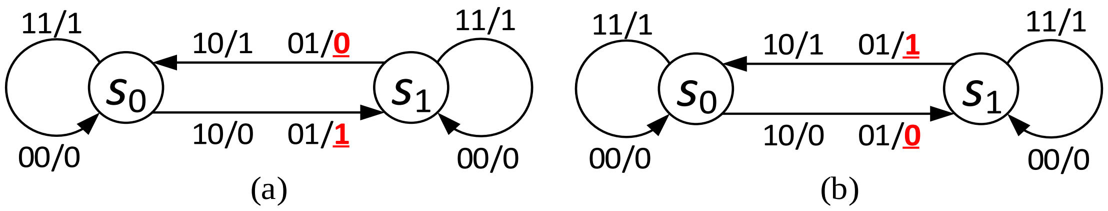


Figure 2.4 STGs for the sequential scaled adders corresponding to (a) ad hoc design C_{AA} of Figure 2.2*b*, and (b) CEASE-generated design C_{CA} of Figure 2.2*c*. The differences between the two STGs are marked in red.

Input: F^*: Target stochastic function

Output: C_{OMC}: A constant-free OMC circuit approximating F^*

Step 1 Approximate F^* by F using Equation (2.5) with rational coefficients $g = \{g(b_0), g(b_1), \ldots, g(b_{m-1})\}$ in [0, 1].

Step 2 Find the least common multiple q of the denominators of g. Let $a = \{a(b_0), a(b_1), \ldots, a(b_{m-1})\} = q \cdot g$.

Step 3 Construct a modulo-q counter M_C with states $s_0, s_1, \ldots, s_{q-1}$.

Step 4 Modify M_C so that on receiving the pattern b_i, it jumps forward $a(b_i)$ states. At each clock cycle, it outputs 1 if M_C overflows, otherwise it outputs 0.

Step 5 Synthesize M_C using any suitable conventional synthesis technique. Output the synthesized circuit C_{OMC}.

Figure 2.5: Algorithm CEASE for constant elimination.

Example 2.3: The state-transition graph (STG) of the CEASE-generated scaled adder C_{CA} (Figure 2.2*c*) is shown in Figure 2.4*b*. Like the combinational adder in Figure 2.3, C_{CA} immediately outputs a 1 when 11 is received, and a 0 when 00 is received. The difference is that when a 10 or 01 is received, C_{CA} remembers this information by going from state s_0 to state s_1, and outputting a 0. When another 10 or 01 is received, the circuit's implicit counter will, in effect, overflow by returning to state s_0 and outputting a 1. In this way, it is guaranteed that *exactly* one 1 is generated whenever *exactly* two copies of 10 or 01 are received. Hence, the RFEs introduced by constant SNs are completely removed. □

Figure 2.5 gives a pseudo-code summary of CEASE. In general, CEASE takes as its input a target arithmetic function F in the form of Equation (2.5), and generates an FSM M_C realizing F without constant inputs. If F is not already in the form of Equation (2.5), it must be approximated by using an SC synthesizer such as ReSC. M_C can be realized in various ways. We refer to any circuit that implements M_C as an *optimal modulo counting* (*OMC*) circuit C_{OMC}. An OMC circuit implements M_C by keeping a running sum of each non-binary input value of interest. Whenever the accumulated sum exceeds one, the circuit outputs a 1 and stores the overflow amount.

Example 2.3 (contd.): Consider applying CEASE as given in Figure 2.5 to synthesize an OMC circuit for the scaled addition function $Z = 0.5(X + Y)$. Equation (2.4) implies that $Z = 0.5p_{\mathcal{X}_V}(1,0) + 0.5p_{\mathcal{X}_V}(0,1) + p_{\mathcal{X}_V}(1,1)$. Therefore, the coefficient set g is $\{g(0,0), g(0,1), g(1,0), g(1,1)\} = \{0/2, 1/2, 1/2, 2/2\}$. The least common multiple q of the denominators in g is the number of count states needed. Since $q = 2$ here, we need a two-state counter. Furthermore, $a = q \cdot g = \{0, 1, 1, 2\}$. Therefore, the counter is designed such that every time the pattern $\mathbf{X}^{(t)}\mathbf{Y}^{(t)} = 10$ or 01 is applied, the counter adds one to its state. The pattern $\mathbf{X}^{(t)}\mathbf{Y}^{(t)} = 11$ adds two to the counter's state. When the counter overflows, a 1 is sent to the output; otherwise the output is set to 0. This confirms that Figure 2.4*b* is indeed the STG for the FSM of an exact scaled adder, with the circuit C_{CA} in Figure 2.2*c* being one of its many possible OMC implementations. □

Another viewpoint on the validity of the scaled adder in Figure 2.4*b* is its behavior under steady-state probability distribution. It is easy to see that the long-term probabilities of staying in state s_0 and s_1 are equal, i.e., $p_S(s_0) = p_S(s_1) = 1/2$, since the state-transition behavior of this circuit is symmetric. The probability of outputting a 1 when $S = s_0$ is $p_{\mathcal{X}_V}(1,1)$, and that probability is $p_{\mathcal{X}_V}(1,1) + p_{\mathcal{X}_V}(0,1) + p_{\mathcal{X}_V}(1,0)$ when $S = s_1$. Hence, the overall probability of outputting a 1 is $Z = p_S(s_0)p_{\mathcal{X}_V}(1,1) + p_S(s_1)[p_{\mathcal{X}_V}(1,1) +$

$p_{\mathcal{X}_\mathrm{V}}(0,1) + p_{\mathcal{X}_\mathrm{V}}(1,0)] = 0.5[p_\mathbf{X}(1) + p_\mathbf{Y}(1)] = 0.5(X + Y)$, which is indeed the scaled addition function.

We now consider the optimality of CEASE in terms of the amount of RFEs it removes. A common way to quantify SC errors is *mean squared error* (*MSE*), which is the accuracy metric we will be using in this chapter. The MSE of an *N*-bit SN **Z** is defined as:

$$\mathrm{MSE}(\mathbf{Z}, N) = \mathbb{E}[(\hat{Z}^{(N)} - Z)^2] \tag{2.7}$$

where $\mathbb{E}(\cdot)$ denotes the expectation function, and $\hat{Z}^{(N)}$ denotes the measured value of the *N*-bit SN **Z** generated by a stochastic circuit. Equation (2.7) computes the average squared deviation of an SN's measured value from its exact or desired value. The next theorem says that among all possible FSMs implementing a stochastic function *F*, CEASE produces a result with the smallest MSE, i.e., the output SN produced by a CEASE-derived OMC circuit fluctuates the least.

Theorem 2.2: Given a stochastic function $F(p_{\mathcal{X}_\mathrm{V}})$ in the form of Equation (2.5), suppose the members of $\mathcal{X}_\mathrm{V}$ are Bernoulli bit-streams with unknown correlations. Then for all positive integers *N*:

$$\mathrm{MSE}(\mathbf{Z}_\mathrm{C}, N) \lesssim \mathrm{MSE}(\mathbf{Z}, N) \tag{2.8}$$

where $\mathbf{Z}_\mathrm{C}$ is the SN produced by a CEASE-generated OMC circuit that implements *F*, while **Z** is the SN produced by any other design that implements *F*. □

The notation $\lesssim$ in Equation (2.8) indicates that inequality holds up to a rounding error. A proof of Theorem **2.2** is given in Appendix A.2. Theorem **2.2** can be understood intuitively from the fact that CEASE's precise counting process guarantees exactness, and hence minimizes MSEs. For comparison, consider the ad hoc design C_AA in Figure 2.2*b*, whose STG is given in Figure 2.4*a*. One can easily see that C_AA also computes scaled addition like the OMC circuit C_CA in Figure 2.2*c* whose STG is in Figure 2.4*b*. Suppose the following artificially constructed SNs are applied to both C_AA and C_CA:

$$\mathbf{X}_{art} = 010101010101 \ (X = 6/12 = 0.5)$$

$$\mathbf{Y}_{art} = 101010101010 \ (Y = 6/12 = 0.5)$$

The expected output value should be $0.5(X_{art} + Y_{art}) = 0.5$, which is exactly what C_{CA} will give. However, feeding these two input SNs to C_{AA} initialized to state s_0 will produce the output $\mathbf{Z} = 111111111111$ ($Z = 12/12 = 1$) — a 100% error! The accuracy difference between the two designs is due to the fact that the OMC circuit *guarantees* to output a 1 whenever two copies of 10 or 01 are received, whereas the ad hoc design does not. CEASE-generated designs also retain the high tolerance of stochastic circuits to transient errors (bit-flips) affecting the variable inputs. An occasional transient or soft error causes a relatively small miscount of the applied input patterns, which can then result in a similarly small output error. For instance, if $\mathbf{X}_{art}$ is changed to 010101010000 due to two 1-to-0 bit-flips, the output value produced by C_{CA} will become 5/12, which is a good approximation of the exact output value 0.5 = 6/12.

A scaled sequential adder constructed in ad hoc fashion around a T flip-flop is given in [48] and shown by simulation to be more accurate than the standard combinational design. The STG of that adder turns out to be exactly the same as the CEASE-generated one shown in Figure 2.4*b*, implying that this T-flip-flop-based design is also an OMC circuit. This confirms the high accuracy claimed for the T-flip-flop-based adder, a key factor in the success of the neural network implementation in [48].

2.4 Case Studies

This section applies CEASE to some representative circuits and assesses the corresponding accuracy improvements. It also examines the accuracy of CEASE using randomly generated stochastic circuits.

Typical Application: CEASE can be applied to any combinational circuits with constant inputs, including those that use SN formats other than unipolar, since it deals

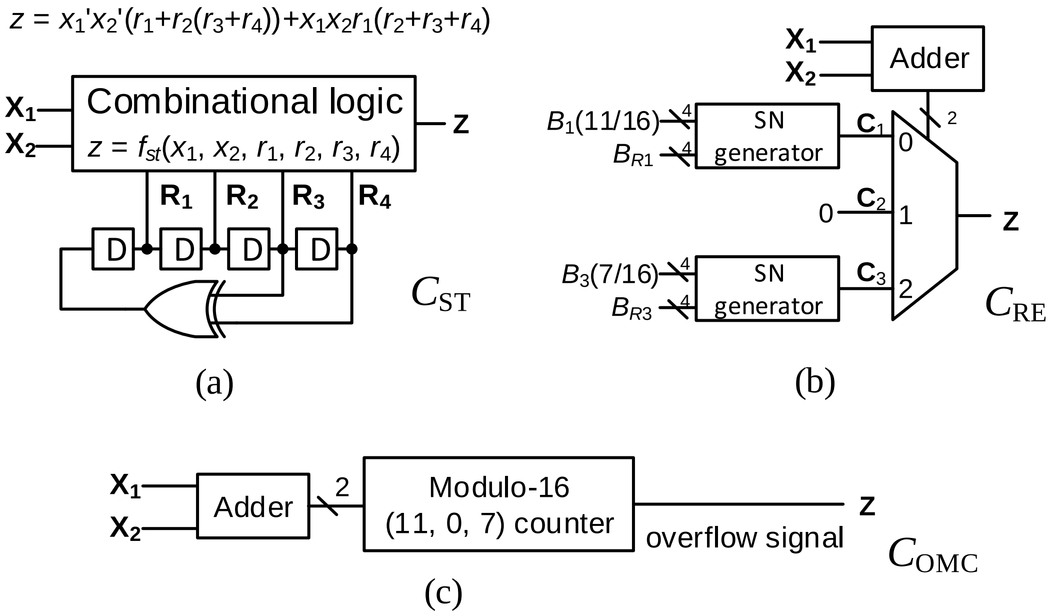


Figure 2.6 Three implementations of Equation (2.9): (a) STRAUSS design C_{ST}, (b) ReSC design C_{RE}, and (c) CEASE design C_{OMC}.

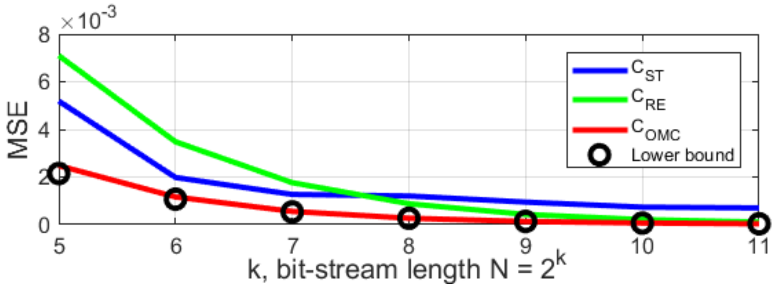


Figure 2.7 MSE comparison for the circuits in Figure 2.6.

directly with probabilities rather than their interpretation or format. Suppose, for example, that CEASE is applied to the circuit C_{ST} synthesized by STRAUSS [6] and outlined in Figure 2.6*a*. C_{ST} uses the inverted bipolar (IBP) SN format with the mapping $X = 1 - 2p\mathrm{x}(1)$ to handle negative values, and realizes the following stochastic function:

$$\tilde{Z} = \frac{7}{16} - \frac{1}{8}\left(\tilde{X}_1 + \tilde{X}_2\right) - \frac{9}{16}\tilde{X}_1\tilde{X}_2 \tag{2.9}$$

where $\tilde{\mathbf{X}}_1$ and $\tilde{\mathbf{X}}_2$ are independent IBP SNs with the same value. This STRAUSS design relies heavily on constant SNs, as it employs four constants $\mathbf{R}_1$–$\mathbf{R}_4$, each of value 0.5. Another implementation C_{RE} of the same function $\tilde{Z}$ synthesized by ReSC [65] is given in Figure 2.6*b*; it relies on three constants $\mathbf{C}_1$–$\mathbf{C}_3$ to provide the same level of accuracy. To implement Equation (2.9) using CEASE, we first derive the corresponding unipolar stochastic function from the relation $\tilde{X} = 1 - 2X$, where $X = p_{\mathbf{X}}(1)$ is the unipolar SN value corresponding to the IBP value $\tilde{X}$. On replacing $\tilde{Z}$, $\tilde{X}_1$ and $\tilde{X}_2$ by their unipolar counterparts in Equation (2.9) and re-arranging, we obtain

$$Z = \frac{11}{16} - \frac{11}{16}X_1 - \frac{11}{16}X_2 + \frac{18}{16}X_1X_2 \tag{2.10}$$

Since X_1 and X_2 are independent, the term X_1X_2 can be written as $p_{\mathbf{X}_1}(1)\, p_{\mathbf{X}_2}(1) = p_{\mathcal{X}_\text{V}}(1,1)$, where $\mathcal{X}_\text{V} = \{\mathbf{X}_1, \mathbf{X}_2\}$. Furthermore, we can “de-marginalize” the marginal probabilities by using $X_1 = p_{\mathcal{X}_\text{V}}(1,0) + p_{\mathcal{X}_\text{V}}(1,1)$ and $X_2 = p_{\mathcal{X}_\text{V}}(0,1) + p_{\mathcal{X}_\text{V}}(1,1)$. Replacing X_1, X_2 and X_1X_2 in Equation (2.10) with these probabilities yields a unipolar stochastic function to which we can apply CEASE.

$$Z = F(f, p_{\mathcal{X}_\text{V}}) = \frac{11}{16}p_{\mathcal{X}_\text{V}}(0,0) + \frac{7}{16}p_{\mathcal{X}_\text{V}}(1,1) \tag{2.11}$$

Equation (2.11) is the unipolar or probability interpretation of Equation (2.9) with coefficients in [0, 1]. This fact can also be directly seen from the ReSC design C_{RE} in Figure 2.6*b*, which outputs 11/16 and 7/16 when the input pattern is 00 and 11, respectively.

A CEASE-generated OMC design C_{OMC} implementing Equation (2.9) in the IBP domain and Equation (2.11) in the unipolar domain is given in Figure 2.6*c*. This is a constant-free sequential circuit built around a modulo-16 counter, which adds 11 or 7 to its count state on receiving a 00 or 11, respectively; it remains in the same state on receiving a 01 or 10. Whenever the counter overflows, a 1 is produced at the output and the counter is reset to the amount of the overflow. C_{OMC} requires four flip-flops for its 16-state counter. C_{ST} shown in Figure 2.6*a*, requires four constant SNs that are generated by a 4-tap LFSR, which also needs four flip-flops. However, C_{ST} has the limitation that each tap of the LFSR does not produce a constant with value exactly 0.5, because it does not loop through the all-0 state, resulting in the constant 8/15 instead of 0.5. To eliminate this small error, C_{ST} would require random sources that are more accurate and probably a little costlier than a 4-bit LFSR. C_{RE}, besides its expensive SN generators, also needs two high-quality 4-bit random sources (omitted in Figure 2.6*b*) for B_{R1} and B_{R3}.

An MSE comparison of the above three circuits appears in Figure 2.7. Here we use MATLAB's *rand* function to generate high-quality random numbers for the ReSC design C_{RE}. The STRAUSS design C_{ST} does not converge to the correct value due to the error introduced by the LFSR's missing all-0 state; this error may be removed by replacing the LFSR with a better random number source. The OMC circuit C_{OMC}, on the other hand, consistently provides the best accuracy among all the designs, and its MSEs match the theoretical lower bound predicted by Theorem **2.2**. This implies that C_{OMC} can compute in far less time, and hence with better energy efficiency, than the other designs. For example, C_{OMC} achieves an MSE of 0.002 with N = 32 bits, while the ReSC design C_{RE} needs approximately 128 bits for the same accuracy.

Complex Matrix Multiplication: Figure 2.8*a* shows a combinational stochastic circuit with 12 constant inputs implementing complex matrix multiplication [59]. It has four outputs, each of which depends on three constant inputs, all of which can be eliminated by CEASE. Here we examine the accuracy improvement after applying CEASE to the sub-circuit spanned by $\mathbf{Z}_1^i$, one of the circuit's four primary outputs. The resulting STG has four states which require two flip-flops to implement. The CEASE-generated OMC circuit is similar in structure to that in Figure 2.6*c*. An MSE comparison between the circuit in Figure 2.8*a* and the OMC circuit appears in Figure 2.8*b*, which again shows that CEASE improves accuracy and, at the same time, matches the lower bound on MSE.

Random Circuits: In the absence of benchmark stochastic circuits, we use

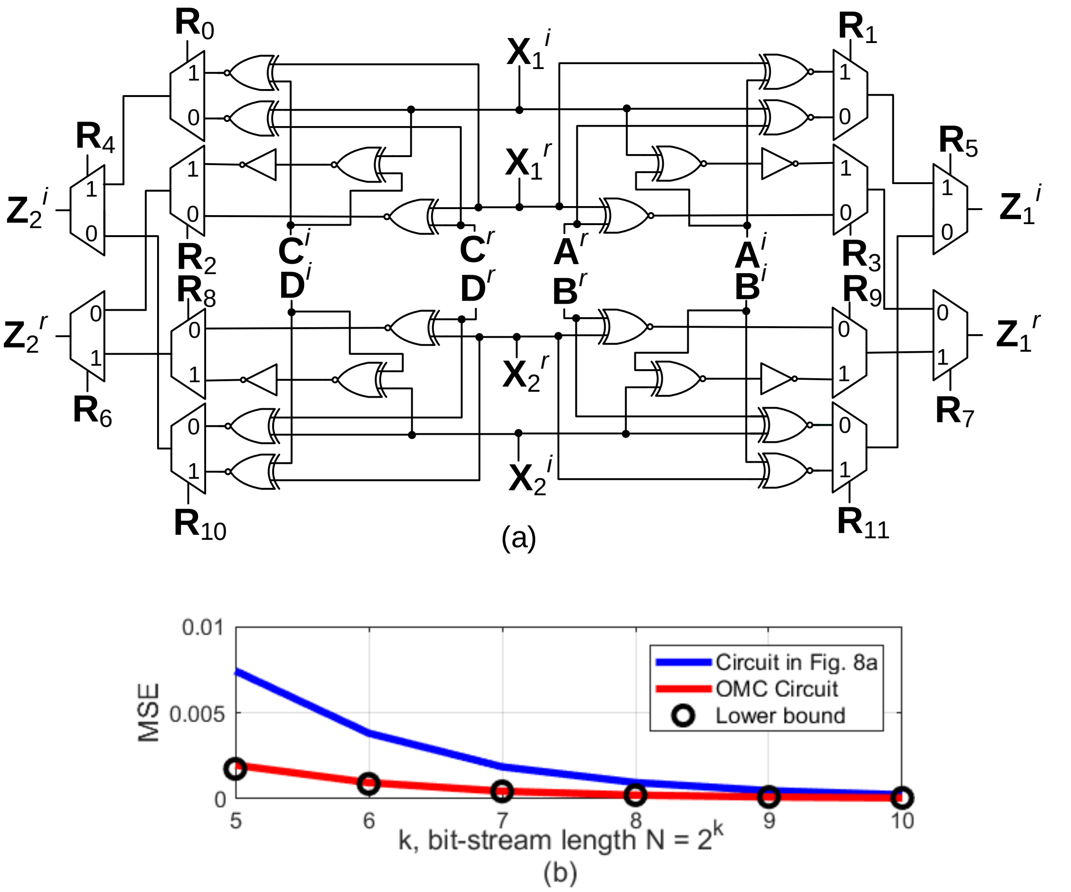


Figure 2.8 (a) Stochastic circuit implementing complex matrix multiplication [59]. (b) MSE comparison with an OMC design.

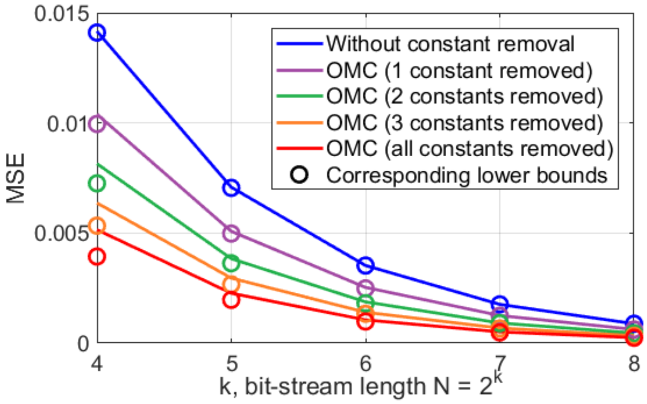


Figure 2.9 MSE comparison of random circuits with four constant- and two variable-input SNs. The lower bounds treat the unremoved constants as variables.

randomly generated ones to further estimate the performance of CEASE. Specifically, we first generate 100,000 random stochastic functions in the form of Equation (2.5) that are implementable using four-constant, two-variable stochastic circuits, where the constants all have value 0.5 and the variable inputs are fed with arbitrary random values. We then apply CEASE and synthesize OMC circuits implementing these random functions. Figure 2.9 plots the average MSEs of these circuits against bit-stream length. We also allow CEASE to remove some or all the constants. As can be seen in Figure 2.9, the MSEs depend on the number of constants removed, with the lowest MSEs achieved by removing all the constants. The results match the theoretical lower bounds, with slight deviations caused by rounding very short SNs.

Sequential Stochastic Circuits: CEASE can be used to remove some RFEs in sequential stochastic circuits as well. We illustrate this via the circuit C_{SEQ} in Figure 2.10*a*, which is a sequential realization of the stochastic function $Z = (X - 2X^2 + 1.5X^3) / (1 - 2X + 2X^2)$ built around an up/down counter, and is one of the most common and best understood sequential SC design styles [52]. C_{SEQ} is based on a 4-state FSM M that realizes a saturating up/down counter (Figure 2.10*c*), and a 4-way multiplexer Q that uses S, the state of M, to select one of its four input SNs to send to the output. Three of these SNs are **0** or **1**, whose bit-streams can be produced by static sources that do not induce RFEs. The remaining constant SN **R** has value 0.5 which is the usual RFE-inducing constant. **R** is fed to the combinational multiplexer Q but not to the sequential part M. Hence, we can incorporate CEASE into C_{SEQ} to remove **R** and improve accuracy. This is done by directly applying the CEASE algorithm to Q to obtain the new FSM $C_{\text{SEQ-CEASE}}$ shown in Figure 2.10*b*. $C_{\text{SEQ-CEASE}}$ transfers the role of all constant inputs **R**, **0** and **1** to its sequential OMC part, and eliminates RFEs due to **R**. Figure 2.10*d* compares the MSEs of C_{SEQ} and $\text{C}_{\text{SEQ-CEASE}}$. It shows clearly that $C_{\text{SEQ-CEASE}}$ has less error than C_{SEQ} for any given bit-stream length. Note, however, that $C_{\text{SEQ-CEASE}}$ does not guarantee the minimum possible error, since CEASE does not consider RFEs produced by the sequential component M whose behavior on random fluctuations is very different from that of a combinational circuit.

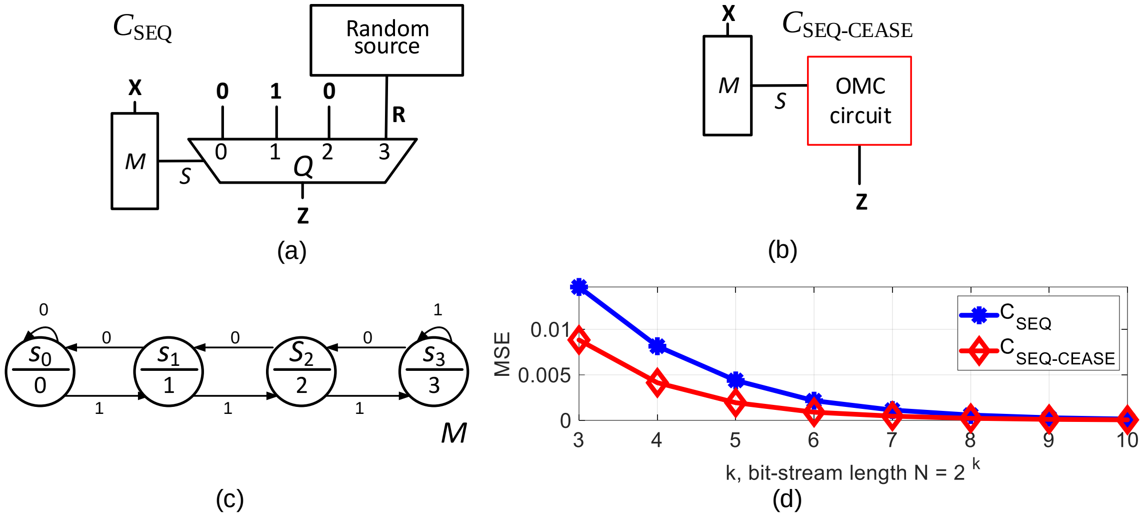


Figure 2.10 (a) Sequential circuit C_{SEQ} implementing $Z = (X - 2X^2 + 1.5X^3) / (1 - 2X + 2X^2)$ [52]. (b) Sequential circuit $C_{SEQ\text{-}CEASE}$ obtained by applying CEASE to the combinational part Q of C_{SEQ}. (c) State-transition graph for the sequential part M of C_{SEQ}. (d) MSE comparison between C_{SEQ} and $C_{SEQ\text{-}CEASE}$.

As we have demonstrated, CEASE can achieve significantly higher accuracy than conventional SC designs with very little increase in hardware cost. In particular, a CEASE design never uses more memory elements than the STRAUSS and ReSC synthesizers. For example, all three designs in Figure 2.6 require just four D flip-flops (DFFs). Thus, CEASE provides an attractive design alternative with high accuracy that may enable a circuit to achieve a satisfactory level of performance with shorter bit-streams. For example, the DNN presented in [48] uses a set of highly accurate stochastic adders that are functionally identical to the OMC circuit in Figure 2.2*c* to strike a balance between low hardware cost and high accuracy requirements.

2.5 Summary

This section has analyzed random fluctuation errors, focusing on those induced by constant SNs. Specifically, we have clarified the role of constant inputs and shown that they are an unexpected source of significant amounts of RFEs despite the fact that such constant inputs are essential in practical SC design. We further demonstrated that constant inputs can be completely eliminated by employing suitably designed sequential stochastic circuits. A systematic algorithm CEASE was devised for efficiently removing constants in this way. The resulting FSMs can be implemented as optimal modulo-counting (OMC) circuits. We further proved analytically that CEASE is optimal in terms of its ability to eliminate RFEs. Case studies were presented which confirm that with fixed computation time (and hence fixed energy consumption), constant-free sequential designs of the kind generated by CEASE can greatly improve the accuracy of SC.

CHAPTER 3
Correlation

Unlike random fluctuation discussed in Chapter 2, accuracy reduction caused by correlation cannot be mitigated by simply lengthening SNs. Special decorrelation circuitry must be used to remove undesired correlation. Regeneration, which is a common decorrelation method, entails huge area and delay overhead. An attractive alternative is isolation-based decorrelation, which is the focus of this chapter. Isolation works by inserting delays (isolators) to eliminate undesirable interactions among SNs. Although it has far lower cost than regeneration, isolation has only been used in an ad hoc fashion, which usually results in non-optimal designs. This chapter begins by examining the characteristics of SC isolation. We show that unless carefully used, it can result in excessive isolator numbers or unexpected functional corruption. We therefore derive the conditions for correct isolator deployment. We also describe the first isolator placement algorithm designed to minimize isolator numbers. Finally, some case studies on decorrelating representative circuits are presented. The material in this chapter has been published in [69] and [9]; the latter work was co-authored with Armin Alaghi and Vincent Lee.

3.1 Correlation in Stochastic Circuits

Correlation in SC refers to similarity between stochastic signals, and is usually caused by insufficient randomness. Correlation control has long been a vexing problem in SC designs, because accuracy-reducing correlation can occur frequently and unexpectedly.

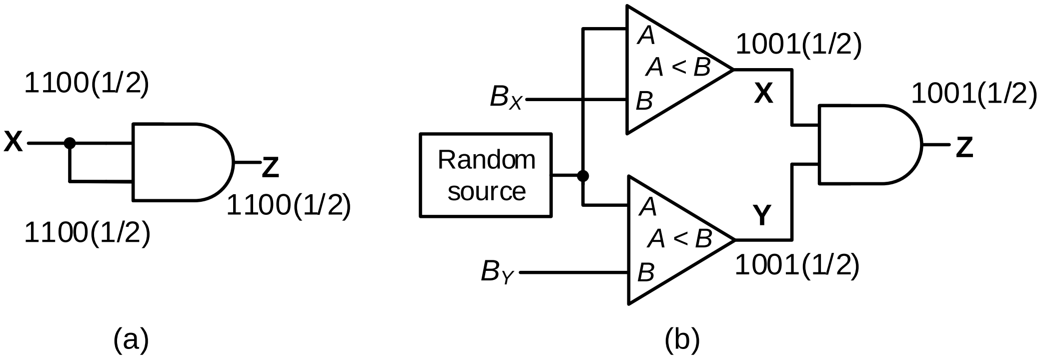


Figure 3.1 SC multiplier with maximally correlated inputs (a) due to signal reuse, and (b) due to a common random source.

This chapter focuses on correlation (or more specifically cross-correlation) among multiple SNs. Autocorrelation, a related topic that concerns the correlation between the bits of a single SN in different clock cycles, will be discussed in Chapter 4.

Correlation can be introduced in many ways, such as signal reuse due to re-convergent fanout, common-mode noise, and sharing random sources. Figure 3.1*a* illustrates correlation between the inputs of an AND-gate multiplier that, at first sight, seems configured to realize X^2. A single bit-stream representing **X** is applied to both inputs of the AND gate. As briefly discussed in Chapter 1, this produces exactly the same bit-stream at the output line *z*, implying that the gate computes *X* instead of X^2. Such a major error is attributable to maximal (positive) correlation between the AND's input signals.

Correlation can be viewed as either an error source that corrupts a stochastic function, or as a resource that changes the function to a potentially more useful one. An example for the latter use is the stochastic edge detector [8] mentioned in Chapter 1, which requires maximally correlated input SNs to function. To leverage correlation as a resource, the amount of correlation injection must be carefully controlled. However, quantifying and managing correlation in SC is surprisingly challenging. Among the 76 correlation metrics summarized in the 2010 survey by Choi et al. [24], none can be directly applied to designing stochastic circuits! For this reason, only a few stochastic circuits have been designed to work with intentional correlation, despite SC tools that were recently developed like stochastic cross correlation (SCC) [3] and correlation manipulation circuits [47]. Consequently, most stochastic circuits, including those produced by major synthesizers like STRAUSS [6] and ReSC [65], require uncorrelated inputs. Ensuring independence among inputs is hence a very important task in SC.

Independence, or the complete absence of correlation, among SNs is relatively easy to define [38]. We say that two SNs **X** and **Y** are *independent*, if the following holds for all *t*:

$$p(\mathbf{X}^{(t)} = 1, \mathbf{Y}^{(t)} = 1) = p(\mathbf{X}^{(t)} = 1)p(\mathbf{Y}^{(t)} = 1) \tag{3.1}$$

The definition can be easily generalized to multiple SNs. For input SNs that are mutually independent, their joint probability distribution can be written as the product of their individual probabilities. This allows us to write the stochastic function defined in Equation (2.2) as

$$p_{\mathbf{Z}} = \sum_{l_1 \cdots l_n = \mathbf{0}}^{\mathbf{1}} \left[f(l_1, \cdots, l_n) \prod_{j=1}^{n} p_{\mathbf{X}_j}(l_j) \right] \tag{3.2}$$

Most SC designs assume independent input SNs, and hence implement a stochastic function in the form of Equation (3.2).

Example 3.1: As described in Example **2.1**, an AND gate implements the stochastic function $Z = p_{\mathbf{XY}}(1, 1)$, which depends on the joint probability of **X** and **Y**. It is usually used with independent inputs, in which case it implements the function in the form of Equation (3.2), which is stochastic multiplication $Z = p_{\mathbf{XY}}(1, 1) = p_{\mathbf{X}}(1)p_{\mathbf{Y}}(1) = XY$. It can be shown that this AND gate performs other functions with other types of correlation [3]. For instance, consider the scenario of Figure 3.1*b* where **X** and **Y** are generated from the same randomness source, and are thus maximally correlated, i.e., 1s in **X** maximally overlap with 1s in **Y**. If **X** contains more 1s than **Y**, then, by conditioning on **Y** = 1, we have $Z = p_{\mathbf{XY}}(1, 1) = p_{\mathbf{X|Y}}(1 \mid 1)p_{\mathbf{Y}}(1) = p_{\mathbf{Y}}(1) = Y$. This is because whenever $\mathbf{Y}^{(t)} = 1$, $\mathbf{X}^{(t)} = 1$, and thus $p_{\mathbf{X|Y}}(1 \mid 1) = 1$. Similarly, if **Y** contains more 1s than **X**, then $Z = X$. Summarizing, $Z = min(X, Y) \neq XY$ when **X** and **Y** are maximally correlated. □

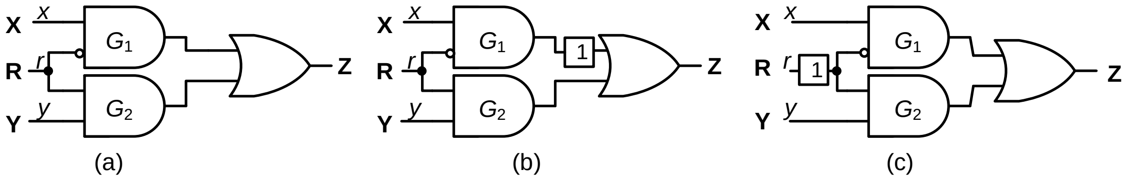


Figure 3.2 (a) MUX circuit, (b) an invalid isolator placement for the MUX, and (c) a valid isolator placement for the MUX.

The foregoing discussion suggests that undesired correlation among input SNs can drastically change the functionality of a circuit in an unexpected way. Furthermore, if internal lines are required to be independent (e.g., the two input lines of the AND gate in Figure 3.1*a*), re-convergent fanout can quickly introduce correlation that also severely degrades accuracy or functionality even with independent primary inputs. Such behaviors call for correlation removal, a process termed *decorrelation* in this dissertation. However, decorrelation introduces sequential components that usually come with considerable latency and hardware overhead. Thus, eliminating the correlation while minimizing the usage of decorrelation circuitry is of crucial importance.

This raises an interesting but difficult question: Which SNs in a circuit should be decorrelated? Generally, the correlated primary inputs of a circuit or its major components (i.e., some internal lines of a circuit) need decorrelation. However, in some cases, correlation among inputs does not affect the functionality of a stochastic circuit at all, e.g., if signals from these inputs cannot reach the output simultaneously. Intuitively speaking, this means that two correlated SNs do not need decorrelation, if they never interact with each other from the viewpoint of the output. This property is called *correlation insensitivity* (*CI*) [2] and eliminates the need for decorrelation.

Example 3.2: Consider the multiplexer (MUX) circuit in Figure 3.2*a* which realizes $z = f(x, r, y) = xr' + yr$. Recall that a MUX implements stochastic scaled addition when the constant **R** is independent with both **X** and **Y**. Here we will show that the lines *x* and *y* in a MUX form a CI pair, meaning that this stochastic scaled adder can function correctly even if **X** and **Y** are correlated. Intuitively speaking, x and y are a CI pair because they cannot both affect the output line z at the same time. Specifically, if $r = 0$, then $z = f(x, 0, y) = x$, and the output depends on x only. Also, if $r = 1$, then $z = f(x, 1, y) = y$, which depends only on y. This implies that under no circumstances will **X** and **Y** interact with each other. Their correlation thus has no effect on the circuit. Indeed, the stochastic function of the MUX is $F = p_{\mathbf{XR}}(1, 0) + p_{\mathbf{YR}}(1, 1)$. This does not depend on the joint probability distribution of **X** and **Y**, regardless of the value of **R**. This shows that it is unnecessary (but unharmful from a functional viewpoint) to decorrelate SNs applied to CI inputs, even if the SNs are correlated. □

We refer interested readers to [2] for further discussion of CI. Here, we provide a way to verify if two inputs of a circuit form a CI pair. For a combinational circuit C implementing the Boolean function $z = f$, the inputs x_i and x_j form a CI pair, if and only if $(dz/dx_i) \wedge (dz/dx_j) = 0$. Here dz/dx_i is the Boolean difference of z with respect to x_i, i.e. $dz/dx_i = f(x_1, \ldots, x_{i-1}, 0,\ldots, x_n) \oplus f(x_1, \ldots, x_{i-1}, 1, \ldots, x_n)$ [2]. This verification method essentially identifies CI pairs by checking if there is any way for a pair of inputs to affect the primary output at the same time. Although correlation can take complicated forms, this chapter will assume that the pairs of SNs that need decorrelation are given. Such a pair of SNs contains correlated SNs that are fed to non-CI ports of a stochastic circuit.

Next, we will describe methods to decorrelate input SNs, focusing on a special approach called *isolation-based decorrelation* (IBD). We will demonstrate in Section 3.5 that the IBD method proposed here can be extended to handle cases where internal lines also call for decorrelation, i.e., an SC system or circuit composed of multiple interconnected SC components, each of which requires independent inputs.

3.2 Mitigating Correlation Errors

As discussed previously, undesired correlation can severely impact the accuracy of a stochastic circuit. Special decorrelation circuitry must be used to cope with undesired correlation. This section describes and analyzes several commonly used decorrelation methods, focusing on isolation-based decorrelation (IBD).

Regeneration decorrelates by reproducing correlated SNs using new and uncorrelated randomness sources. This is done by using SBCs to transform the SNs back to a binary form, and then using independent RNSs to produce a new set of SNs. An example that decorrelates the circuit of Figure 3.1*a* is shown in Figure 3.3*a*. Here, a counter serving as an SBC first transforms **X** to a binary number, which is then used to produce a new and uncorrelated SN by the BSC comprising a comparator and an independent randomness source. Regeneration is arguably the most powerful method of decorrelating SNs, since it can be applied to almost all scenarios. However, regeneration is extremely expensive in hardware overhead due to the need for RNSs, comparators, and counters, all of which have very high area cost [65]. The use of SBCs also implies long latency that is equal to the bit-stream length *N*, because new SNs cannot be reproduced until the stochastic-to-binary conversion has been completed. This latency problem can interfere with SN flow, and desirable properties such as progressive precision [8] may be lost. Regeneration can also introduce new errors during the format conversion process, as the accuracy of regenerated SNs highly depends on the precision of the intermediate binary numbers as well as the quality of the random numbers used by the BSC.

Shuffling is another decorrelation method that also tackles correlation errors by randomizing an SN [28][47][68]. It differs from regeneration in that a shuffler does not completely regenerate an SN, but instead it remembers some bits it has received so far, and randomly permutes them for later release. Figure 3.3*b* shows how a shuffler with depth *D* = 1 bit is used to decorrelate the squarer in Figure 3.1*a*. The depth parameter *D* is the number of flip flops for storing input bits. Here in each clock cycle, the shuffler uses the extra random number **R** to determine whether to store the new bit in its flip flop, or directly release this bit to the output line. If it chooses to store the new bit, the bit previously held by the flip flop will be released to the output line. This way, the number of 1s in the newly

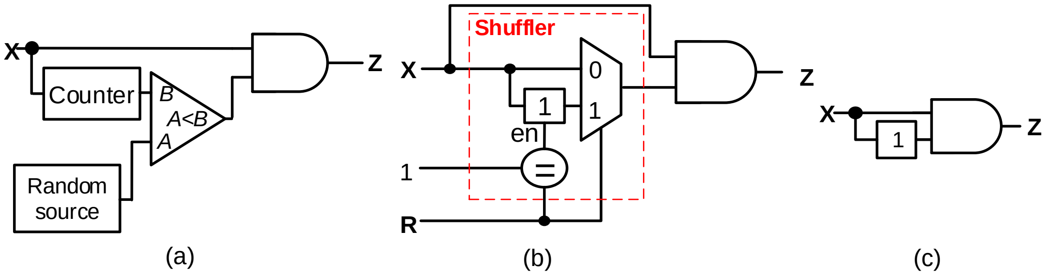


Figure 3.3 (a) Regeneration-based, (b) shuffle-based, and (c) isolation-based decorrelation. The box marked “1” denotes an isolator flip-flop.

generated SN and that of the original SN can be made exactly the same. However, their location can be very different, since an early bit can potentially be released much later. The performance of a shuffler depends on its depth *D*, which determines the number of bits it can hold. With a larger *D*, the shuffler can randomize the bit locations in the SN more uniformly, but at the cost of higher hardware overhead and longer latency to initialize its flip flops. In other words, unless $D = N - 1$, shuffling does not completely eliminate correlation, but only reduces it. To see this, consider permuting the SN **X** = 000000111111 with a shuffler having a small depth $D = 1$. It is not hard to see that an early 1 received by the shuffler is much more likely to be released earlier than later, resulting in a new SN that still has most of its 1s in its initial segment. Another major disadvantage of a shuffler is that it also requires extra (and expensive) random sources, whose quality has a direct impact on the shuffler’s performance.

Isolation is a decorrelation method proposed (but not explored much) by Gaines [29] in the 1960s, which uses delay elements called isolators. An *isolator* is intended to produce a 1-cycle time delay in an SN **X** and is easily implemented by inserting a DFF into line *x*. We use a boxed number to represent the number of consecutive isolators inserted in a line. For instance, Figure 3.3*c* shows an AND gate with one isolator, i.e., a single DFF, inserted into its lower input line. Isolator insertion, which we will refer to as *isolation-based decorrelation* (IBD), decorrelates a stochastic circuit by adjusting the delays of correlated SNs. If the bits of **X** are independently generated, as is normally the case, then

X and a *k*-cycle delayed version **X**(*k*) of **X** are independent at any given clock period. Thus, the IBD shown in Figure 3.3*c* for the squarer circuit of Figure 3.1*a* changes the circuit's output in clock period *t* to $Z = p(\mathbf{X}(0)^{(t)} = 1, \mathbf{X}(1)^{(t)} = 1) = p_{\mathbf{X}(0)}p_{\mathbf{X}(1)} = X^2 = 0.25$, as desired. In general, IBD has much lower hardware cost than both regeneration and shuffling, and is therefore the preferred method among the three. However, IBD has received little research attention [22], despite the fact that it was proposed long ago [29]. In particular, no systematic study of IBD has been reported, and many SC designs employed IBD in an ad hoc fashion. As the next example shows, carelessly inserting many isolators into a circuit may not only lead to overuse of isolators, but also to incorrect decorrelation. In other words, isolator numbers and positions must be carefully chosen.

Example 3.3: The circuit in Figure 3.4*a* realizes the Boolean function $z = f(x_1, x_2, x_3) = (x_1 + x_2)x_3$. When it is designed to work with independent inputs, it implements the stochastic function $Z = (X_1 + X_2 - X_1X_2)X_3$. It does so by first computing $G = X_1 + X_2 - X_1X_2$ via the OR gate. It then uses an AND gate to produce the final result $(X_1 + X_2 - X_1X_2)X_3$. This can also be seen by directly applying Equation (2.2)

$$\begin{aligned} Z &= p_{\mathbf{X}_1\mathbf{X}_2\mathbf{X}_3}(1,0,1) + p_{\mathbf{X}_1\mathbf{X}_2\mathbf{X}_3}(1,1,1) + p_{\mathbf{X}_1\mathbf{X}_2\mathbf{X}_3}(0,1,1) \\ &= X_1(1 - X_2)X_3 + X_1X_2X_3 + (1 - X_1)X_2X_3 \\ &= (X_1 + X_2 - X_1X_2)X_3 \end{aligned} \qquad (3.3)$$

This design does *not* work as expected, however, if $\mathbf{X}_1$, $\mathbf{X}_2$ and $\mathbf{X}_3$ are correlated. Suppose $\mathbf{X}_1$, $\mathbf{X}_2$ and $\mathbf{X}_3$ all have value 0.5. The correct output should be $Z = (0.5 + 0.5 - 0.25)0.5 = 0.375$. However, if $\mathbf{X}_1$, $\mathbf{X}_2$ and $\mathbf{X}_3$ are identical (maximally correlated), the OR gate outputs exactly the same bit-stream as $\mathbf{X}_1$, and so does the AND gate. The value of **Z** is estimated as 0.5 instead of the desired 0.375. It is essential to eliminate such correlation among SNs for accurate computation. Figure 3.4*b* shows a typical isolator placement eliminating the correlation among the three primary inputs; it uses three isolators to create delays of 0, 1 and 2 in the three primary input lines. This turns out to be non-optimal in terms of isolator

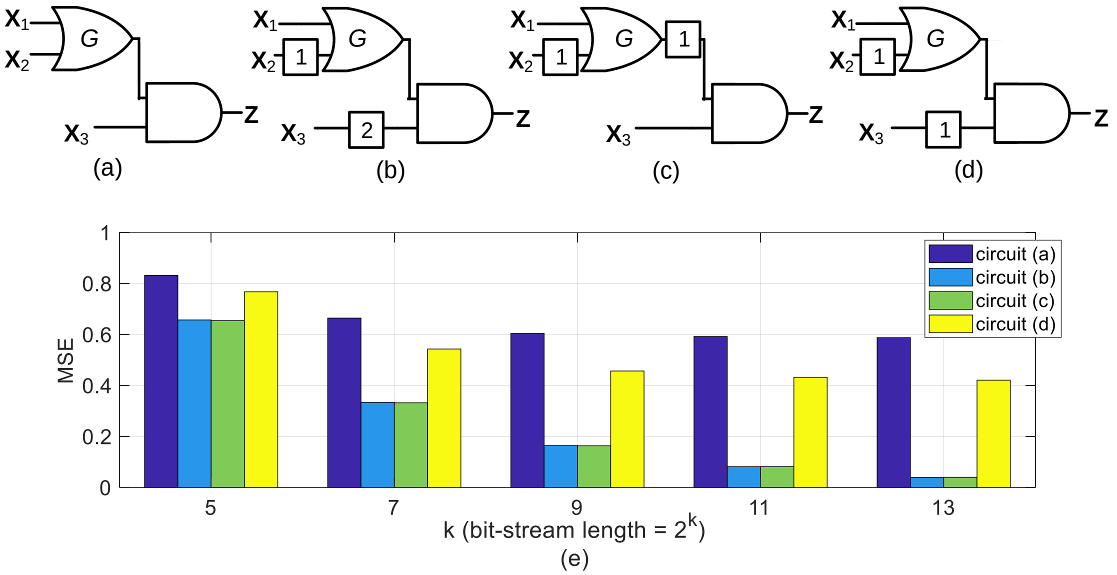


Figure 3.4 SC circuits intended to implement $Z = (X_1 + X_2 – X_1X_2)X_3$ (a) with no decorrelation, (b) with decorrelation using 3 isolators, (c) with decorrelation using 2 isolators, (d) with incomplete and hence invalid decorrelation. (e) Error comparison of the three circuits.

count, however. An optimal placement with only two isolators appears in Figure 3.4*c*, where $G = X_1(1) + X_2(2) – X_1(1)X_2(2)$ and $X_3 = X_3(0)$ are independent, and therefore $Z = (X_1 + X_2 – X_1X_2)X_3$. Isolators must also be carefully placed to ensure functional correctness. Figure 3.4*d* shows a placement that also uses two isolators. However, it is incorrect as $\mathbf{X}_2$ and $\mathbf{X}_3$ are both delayed by one clock cycle, so they are still correlated as they interact with each other. The accuracy of the circuits in Figure 3.4 is compared via simulation in Figure 3.4*e*, which plots the MSEs in **Z** against bit-stream length for 10,000 randomly generated input values. The original circuit in Figure 3.4*a* and the incorrectly decorrelated circuit in Figure 3.4*d* both output erroneous values, regardless of bit-stream length. The two isolation-based decorrelated circuits shown in Figure 3.4*b*-*c* show no correlation error, and their random-fluctuation errors go rapidly toward zero as the bit-streams lengthen. □

Though IBD has vastly lower area cost than regeneration and shuffling, it still can impose considerable area overhead. Hence, reducing isolator count while delaying the SNs correctly is important. As we will see, correlation can take much subtler forms than those in the preceding examples, and finding a good isolator placement is a tricky problem. This is mainly due to our limited understanding of how to decorrelate stochastic circuits, a topic that has been largely ignored in the literature. The remainder of this chapter investigates the foundations of IBD, with a focus on optimizing isolator placement.

3.3 Isolation-Based Decorrelation

IBD works by inserting isolators (DFFs) into a circuit to delay selected SNs by selected numbers of clock cycles. It exploits the inherent temporal independence among successive bits generated by a Bernoulli RNS [38]. In particular, IBD assumes that a set of SNs become mutually independent, if the correlated SNs in this set are delayed by different clock cycles. The intuitive idea behind IBD is to insert isolators so that the target circuit only "sees" independent SNs.

It is worth mentioning that linear-feedback shift registers (LFSRs) can be viewed as a special example of IBD use in SC. Jeavons et al. observe that the $(2^m - 1)$-bit sequences obtained by tapping each stage of an m-bit LFSR with maximum period, which is the most common RNS used in SC, are mutually independent [38]. This special case of isolator usage delays a single SN with value 0.5 multiple times to generate a set of mutually independent SNs, all with value 0.5.

To more generally analyze IBD, we need to first express its behavior mathematically. An IBD $\mathfrak{D}$ transforms a target circuit C into a circuit $C_{\mathfrak{D}}$ containing isolators which is no longer combinational. This prohibits the use of Equation (2.2) to describe $C_{\mathfrak{D}}$, because it is for combinational circuits only. However, $C_{\mathfrak{D}}$ can be described by a Boolean function $f_{\mathfrak{D}}$ of delayed primary inputs. This concept, sometimes called a *clocked Boolean function* (*CBF*), was developed for some distantly related problems such as sequential circuit verification [66]. For example, the decorrelated circuit of Figure 3.4*b* has the CBF:

$$z = f_{\mathcal{D}}(x_1(0), x_2(1), x_3(2)) = x_1(0)x_3(2) + x_2(1)x_3(2) \tag{3.4}$$

Here, we explicitly express Equation (3.4) using delayed Boolean variables to emphasize the fact that the circuit, albeit sequential, implements a Boolean function $f_{\mathcal{D}}$ that operates on delayed inputs. The stochastic behavior of $C_{\mathcal{D}}$ therefore takes the form $Z = F(f_{\mathcal{D}}, p_{\mathcal{X}_{\mathcal{D}}})$, where $\mathcal{X}_{\mathcal{D}} = \{\mathbf{X}_1(0), \mathbf{X}_2(1), \mathbf{X}_3(2)\}$ is the set of SNs applied to x_1, x_2, x_3, respectively. Observe that now the output value Z depends on the delayed versions of the input SNs.

The CBF concept enables us to describe the stochastic behavior of any circuit in which isolators are placed. Again, careless isolator placement can lead to invalid decorrelation. In Example **3.3**, the circuit $C_{\mathcal{D}}$ shown in Figure 3.4*d* has an invalid isolator placement $\mathcal{D}$ for the circuit in Figure 3.4*a*. Its stochastic function $Z = F(f_{\mathcal{D}}, p_{\mathcal{X}_{\mathcal{D}}})$ has $\mathcal{X}_{\mathcal{D}} = \{\mathbf{X}_1(0), \mathbf{X}_2(1), \mathbf{X}_3(1)\}$, with $\mathbf{X}_2(1)$, $\mathbf{X}_3(1)$ still being correlated. Another example of an invalid placement is given by the circuit in Figure 3.2*b*, which is an attempt to decorrelate the circuit in Figure 3.2*a*. In this case, $\mathcal{X}_{\mathcal{D}} = \{\mathbf{X}(1), \mathbf{Y}(0), \mathbf{R}(0), \mathbf{R}(1)\}$, and the circuit depends on four SNs instead of three, which is obviously invalid.

How do we determine the validity of an IBD? Suppose we are interested in using IBD to decorrelate a set of possibly correlated SNs $\mathcal{X}$ for a target circuit C. Let $C_{\mathcal{D}}$ denote the circuit obtained from C by an isolator placement $\mathcal{D}$. Further, let $\mathcal{X}_{\mathcal{D}}$ be an SN set containing possibly delayed versions of the SNs in $\mathcal{X}$, and $\mathcal{X}_{\text{IND}}$ be a set of SNs that are independent, but have the same values as the SNs in $\mathcal{X}$. The placement $\mathcal{D}$ is called a *valid isolator placement* (VIP) if $F(f_{\mathcal{D}}, p_{\mathcal{X}_{\mathcal{D}}}) = F(f, p_{\mathcal{X}_{\text{IND}}})$ for all values of the input SNs. Otherwise, $\mathcal{D}$ is an *invalid placement*. Intuitively, this means that $\mathcal{D}$ is a VIP, if $C_{\mathcal{D}}$ performs that same function as C does with independent inputs. The following theorem, proven in Appendix A.3, gives sufficient conditions for an isolator placement to be a VIP.

Theorem 3.1: $\mathcal{D}$ is a valid isolator placement (VIP) for C if the following two conditions are satisfied:

$$\mathcal{X}_{\mathcal{D}} = \{\mathbf{X}_1(t_1), \mathbf{X}_2(t_2), \ldots, \mathbf{X}_n(t_n)\} \tag{3.5}$$

$$\text{For every pair } \mathbf{X}_i, \mathbf{X}_j \text{ that needs decorrelation, } t_i \neq t_j \tag{3.6}$$

Condition (3.5) means that the output of $C_{\mathcal{D}}$ "sees" only a single delayed version $\mathbf{X}_i(t_i)$ of each SN $\mathbf{X}_i$. Condition (3.6) implies that all the SNs that need decorrelation are delayed without conflicting time overlap. Together, the two conditions ensure that the SNs reaching the output are independent, delayed versions of the input SNs, and hence $\mathcal{D}$ is a VIP.

Using Theorem **3.1**, we can quickly verify if a circuit is correctly decorrelated. For instance, the circuit shown in Figure 3.2*c* is a VIP for the SC circuit of Figure 3.2*a*, since $\mathcal{X}_{\mathcal{D}} = \{\mathbf{X}(0), \mathbf{Y}(0), \mathbf{R}(1)\}$, and *x*, *y* form a CI pair. For the same reason, the circuit $C_{\mathcal{D}}$ of Figure 3.4*b* is a VIP for that in Figure 3.4*a*, since $\mathcal{X}_{\mathcal{D}} = \{\mathbf{X}_1(0), \mathbf{X}_2(1), \mathbf{X}_3(2)\}$ and these SNs are delayed by different numbers of cycles. A stochastic circuit may have many VIPs. For instance, the circuit of Figure 3.4*c* is also valid but contains only two isolators. We are particularly interested in finding VIPs that contain as few isolators as possible. This problem can be difficult, even for circuits of moderate size, mainly because SNs derived from a source **X** delayed by the same amount are correlated, and their uncontrolled interaction can lead to circuit malfunction.

Theorem **3.1** provides functional conditions (3.5) and (3.6) that guarantee a VIP. Next, we identify structural (path) properties of a circuit that meet these conditions, and suggest how VIPs can be constructed. The starting point is a stochastic circuit *C* without isolators. The goal is to place a set of isolators on the lines of *C* so that the resulting $C_{\mathcal{D}}$ is validly decorrelated.

Let $\mathcal{X} = \{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n\}$ be a set of input SNs for *C*. To construct a VIP $\mathcal{D}$ for *C* using Theorem **3.1** requires $\mathcal{X}_{\mathcal{D}} = \{\mathbf{X}_1(t_1), \mathbf{X}_2(t_2), \ldots, \mathbf{X}_n(t_n)\}$, and $t_i \neq t_j$ if the pair $\mathbf{X}_i$, $\mathbf{X}_j$ needs decorrelation. The next theorem places conditions on the input-output paths of $C_{\mathcal{D}}$ which ensure the conditions of Theorem **3.1** are met.

Theorem 3.2: $\mathscr{D}$ is a VIP for C if the following two conditions are satisfied:

All paths from a primary input x_i to the primary output z of $C_{\mathscr{D}}$ contain the same number of isolators. (3.7)

For each primary input pair x_i, x_j with $\{\mathbf{X}_i, \mathbf{X}_j\}$ needing decorrelation, at least one path from x_i to z contains a different number of isolators than at least one path from x_j to z. (3.8)

Proof of Theorem 3.2: Assume all paths from x_i to z contain the same number of isolators, say t_i isolators. Then $f_{\mathscr{D}}$ depends on x_i only through $x_i(t_i)$, the t_i-cycle-delayed version of x_i. Thus $f_{\mathscr{D}} = f_{\mathscr{D}}(x_1(t_1), x_2(t_2), \ldots, x_n(t_n))$, implying $\mathscr{X}_{\mathscr{D}} = \{\mathbf{X}_1(t_1), \mathbf{X}_2(t_2), \ldots, \mathbf{X}_n(t_n)\}$ and (3.5) holds. Similarly, (3.8) implies that whenever the pair $\mathbf{X}_i$, $\mathbf{X}_j$ needs decorrelation, $t_i \neq t_j$, and hence (3.6) holds. By Theorem **3.1**, $\mathscr{D}$ must be a VIP for C.

Again, we use the circuit in Figure 3.2*c* as an example to illustrate Theorem **3.2**. Condition (3.7) needs to be fulfilled by all inputs. For instance, input r has two paths to z, and they all contain one isolator. Also, inputs needing decorrelation have to fulfill Condition (3.8). For example, input pair x, r is not CI, and $\mathbf{X}$, $\mathbf{R}$ are correlated. Therefore the path from x to z and the path from r to z have to contain different numbers of isolators (in this case, zero and one isolator, respectively). The circuit shown in Figure 3.2*c* is thus correctly decorrelated.

3.4 Optimizing Isolator Placement

Meeting the conditions we have previously established guarantees a valid isolator placement (VIP). In practice, we are also interested in minimizing hardware overhead induced by decorrelation circuits. In other words, VIPs that use a minimum number of isolators are highly preferred. In this section, we show that such a VIP can be systematically constructed by solving an integer linear programming (ILP) problem that aims to optimize isolator usage under the set of constraints stated in Theorem **3.2**.

Let $C_{\mathfrak{D}}$ be the circuit resulting from employing an IBD $\mathfrak{D}$ in a target circuit *C*. Let **E** = {e_1, e_2,…, e_q} be the set of lines in *C,* and let **W** = {w_1, w_2,…, w_q}, where w_i is the number of isolators placed on line e_i. The ILP's objective function is to minimize $\sum_{i=1}^{q} w_i$ while meeting the two conditions in Theorem **3.2**. These conditions must be converted into a suitable set of ILP constraints. Condition (3.7) requires equal numbers of isolators on pairs of paths. For a path-pair P_1, P_2, this can be described by an *equality constraint* (EC):

$$\sum_{i:e_i \in P_1} w_i = \sum_{i:e_i \in P_2} w_i \tag{3.9}$$

Condition (3.8) requires distinct numbers of isolators in path-pairs P_1, P_2, and can be described by an *inequality constraint* (IEC) set:

$$\begin{aligned} &\sum_{i:e_i \in P_1} w_i = \sum_j j\, d_{1,j}, && \sum_{i:e_i \in P_2} w_i = \sum_j j\, d_{2,j} \\ &\sum_j d_{1,j} = 1, && \sum_j d_{2,j} = 1 \\ &d_{1,j} + d_{2,j} \leq 1 && \end{aligned} \tag{3.10}$$

where $d_{k,j}$ is a decision variable that will be 1 if *j* isolators are placed on P_k, and 0 otherwise. The summation variable *j* ranges from 0 to the maximum number of isolators permitted in a line. The IECs ensure that a unique number of isolators is placed on each of P_1 and P_2, and that these numbers are different.

In addition to ECs and IECs, we also need *non-negativity and integer constraints* (NICs) to ensure that w_i and $d_{k,j}$ are non-negative integers for all *i, j* and *k.* The resulting ILP algorithm for constructing a valid isolator placement while minimizing isolator usage is summarized in Figure 3.5; we refer to it as VAIL (**V**IP **A**lgorithm based on **IL**P).

Example 3.4: Consider the circuit in Figure 3.6 which realizes the Boolean function $z = (x'y)'u + (x'y)v$, where *u*, *v* form a CI pair. Assume the input SNs **U**, **V**, **X** and **Y** are correlated. A straightforward ad hoc VIP is obtained by delaying the SNs at the

```
Input:    C: target circuit implementing $z = f(x_1, x_2, \ldots, x_n)$
          I: list of SN pairs that need decorrelation
Output:   $C_{\mathfrak{D}}$: decorrelated circuit
Step 1    Solve integer program $min \sum_{i=1}^{q} w_i$ with the constraints:
            (1) ECs for all path-pairs from each $x_i$ to $z$
            (2) IECs for a path-pair from each $\{x_i, x_j\}$ to $z$ if $\{\mathbf{X}_i, \mathbf{X}_j\} \in I$.
            (3) NICs for all $w_i$ and all decision variables
Step 2    Map $\{w_1, w_2, \ldots, w_q\}$ to $C_{\mathfrak{D}}$
```

Figure 3.5 The VAIL algorithm for constructing a VIP with minimal isolator count.

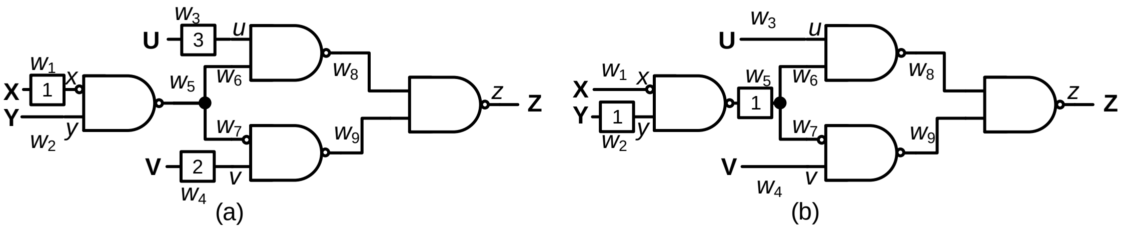


Figure 3.6 Example SC circuit decorrelated using (a) a simple ad hoc IBD method, and (b) the proposed VAIL method.

primary inputs. As shown in Figure 3.6*a*, 0, 1, 2 and 3 isolators are inserted in inputs *y, x, v* and *u*, respectively, for a total of 6 isolators. To decorrelate the circuit using VAIL, we formulate the ILP $min \sum_{i=1}^{9} w_i$ with the following constraints:

I. $w_1 + w_5 + w_6 + w_8 = w_1 + w_5 + w_7 + w_9$

II. $w_2 + w_5 + w_6 + w_8 = w_2 + w_5 + w_7 + w_9$

III. $w_1 + w_5 + w_6 + w_8 \neq w_2 + w_5 + w_6 + w_8$

IV. $w_3 + w_8 \neq w_1 + w_5 + w_6 + w_8$

V. $w_3 + w_8 \neq w_2 + w_5 + w_6 + w_8$

VI. $w_4 + w_9 \neq w_1 + w_5 + w_6 + w_8$

VII. $w_4 + w_9 \neq w_2 + w_5 + w_6 + w_8$

where w_i is a non-negative integer for all i. Constraints I and II are ECs, and constraints III-VII can readily be converted to IECs. A solution to this ILP has just two isolators, whose placement is depicted in Figure 3.6*b*. □

3.5 Case Studies

This section evaluates VAIL's ability to remove correlation errors as well as its hardware overhead. It does so by comparing VAIL against a naïve isolator placement on random circuits and some other representative circuits.

Random Circuits: First, we use randomly generated 4-input and 6-input circuits with gate counts ranging from 10 to 20, which is a typical size range for performance assessment of stochastic circuits. We generate the input SNs with three different levels of correlation: high, moderate and low, corresponding to generating 100%, 70% and 30% of the SN bits, respectively, from the same RNS. The resulting errors are averaged over 500 randomly generated SNs for each circuit, and then averaged across 100 randomly generated circuits. Similar conclusions can be drawn when averaged over more SNs and more random circuits. Here, the output errors are measured by the absolute difference in value between the circuits' outputs and the desired outputs, and are plotted against output bit-stream length in Figure 3.7. Recall that making the input SNs longer can only reduce errors caused by random fluctuation, while errors induced by correlation remain. Therefore, as clearly seen from Figure 3.7, the errors of circuits without decorrelation do not converge to zero as the input SNs lengthen, since they have correlation errors. The magnitude of these errors depends on the level of correlation. Circuits with highly correlated inputs have large correlation errors, as expected. The random fluctuation errors of circuits decorrelated by VAIL also decrease to zero as the input SNs lengthen, but the circuits exhibit no correlation errors whatsoever. The results in Figure 3.7 show that isolator placement by VAIL eliminates all correlation errors from the chosen SC circuits, regardless of the correlation level among their primary inputs.

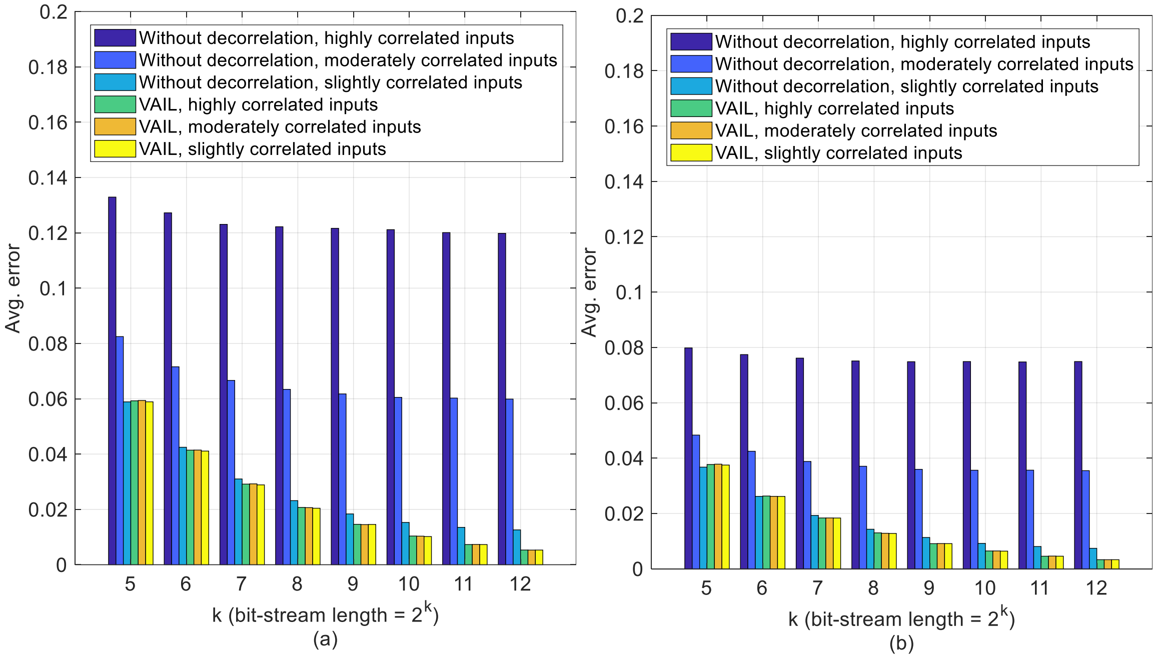


Figure 3.7 Accuracy assessments of VAIL with various correlation levels for (a) 4-input and (b) 6-input random circuits.

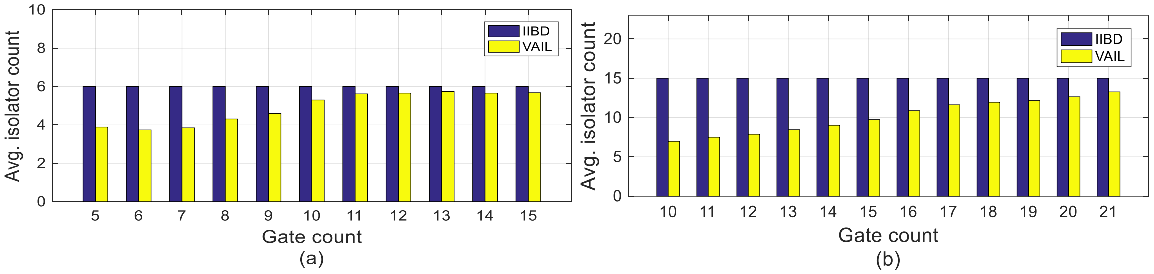


Figure 3.8 Isolator usage comparison between VAIL and IIBD for (a) 4-input and (b) 6-input random circuits of various sizes.

We also compared the performance of VAIL against a basic isolator placement scheme in terms of the number of isolators required to correctly decorrelate a random circuit. We assume that all the input SNs of a target SC circuit are to be pairwise decorrelated. For comparison with VAIL, we use the intuitively clear IBD approach in Figure 3.6*a*, namely, delaying the primary input SNs by different amounts 0,1,2,3,…; we refer to this as *input IBD* or IIBD. The number of isolators needed for IIBD is $n(n-1)/2$, where *n* is the number of inputs, whereas the number of isolators used by VAIL varies with the structure and the size of the target circuit. Figure 3.8 compares the isolator usage of VAIL and IIBD by plotting isolator usage against some typical SC circuit sizes measured by gate count. For each size, we generate 100 randomly structured circuits and compute their average isolator count.

As Figure 3.8 shows, VAIL uses on average 35.2% and 53.5% fewer isolators for 5-gate 4-input, and 10-gate 6-input circuits, respectively. This implies that for SC circuits of moderate size, VAIL can greatly reduce the hardware overhead of isolators required for IBD. Also note an interesting phenomenon: the average isolator number gradually grows as circuit size increases. This is because in large randomly generated circuits, the interconnections among gates tend to be complex. The path constraints such as ECs and IECs required for placing isolators internally are rather stringent for large circuits. Hence, an optimal solution contains relatively few internal isolators, resulting in more isolators being placed in the primary input lines. Many real-world circuits, however, are highly structured, and thus require far fewer isolators for decorrelation using VAIL than random circuits of similar size, as will be shown shortly. The number of isolators used by IIBD gives a useful upper bound on the number used by VAIL, since the solution space of VAIL contains that of IIBD. Note too that IIBD and VAIL lead to circuits that are very similar in terms of accuracy, as both of them can completely remove correlation errors if employed correctly.

Complex Matrix Multiplier: Figure 3.9 shows a relatively large stochastic circuit intended for simulating a quantum circuit [59], which we also used for evaluating CEASE in Chapter 2. It implements the following matrix multiplication:

$$\begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} = \frac{1}{4}\begin{bmatrix} A & B \\ C & D \end{bmatrix}\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \tag{3.11}$$

where each variable is a complex number with its real and imaginary parts denoted by

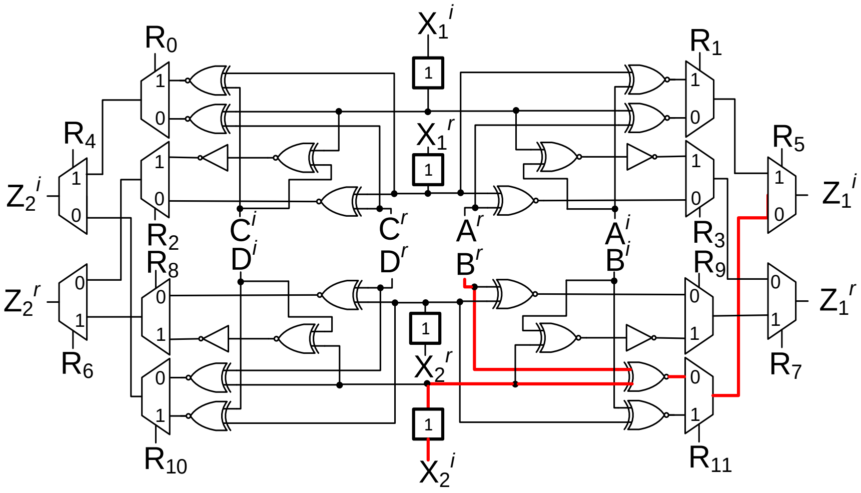


Figure 3.9 SC circuit implementing complex matrix multiplication [59]; an optimal placement of four isolators is shown.

superscripts *r* and *i*, respectively, in the figure. The circuit has six variable input SN pairs **A**, **B**, **C**, **D**, $\mathbf{X}_1$, $\mathbf{X}_2$ and two output pairs $\mathbf{Z}_1$, $\mathbf{Z}_2$, all defined by Equation (3.11). It also has 12 constant inputs $\mathbf{R}_0$-$\mathbf{R}_{11}$, all of which carry a fixed value 0.5. As suggested in [59], the accuracy of simulation is sensitive to signal correlation, and ensuring independence across the SNs, both variable and constant, is important. Here, we will assume that the constant sources $\mathbf{R}_0$-$\mathbf{R}_{11}$ are independent, which is usually the case when they are generated from multiple LFSRs with distinct initial states. However, the variable inputs are potentially correlated, which is the case when they originate from shared upstream SC circuits. The goal here is to construct a VIP with VAIL that, under the stated assumptions, guarantees the four output SNs are free of correlation errors.

The four outputs of the circuit are driven by different, but overlapping, sub-circuits, each with eight primary inputs. To construct a VIP using VAIL, we first list the required constraints, including ECs, IECs and NICs, for each of the sub-circuits. Then we form an integer linear programing problem subject to all the listed constraints. For illustration purposes, we only show how to construct ECs and IECs for the sub-circuit with output $\mathbf{Z}_1^i$, which depends on the value of $\mathbf{X}_1^i$, $\mathbf{X}_1^r$, $\mathbf{X}_2^i$, $\mathbf{X}_2^r$, $\mathbf{A}^i$, $\mathbf{A}^r$, $\mathbf{B}^i$, $\mathbf{B}^r$. These inputs all have only one path to $\mathbf{Z}_1{}^i$, so no ECs are needed. IECs are required for SN pairs that need decorrelation. For example, $\mathbf{B}^r$ and $\mathbf{X}_2{}^i$ are non-CI and correlated, and therefore the path from $\mathbf{B}^r$ to $\mathbf{Z}_1{}^i$ and the path from $\mathbf{X}_2{}^i$ to $\mathbf{Z}_1{}^i$, as marked in red in Figure 3.9, need to contain different numbers of isolators. For each sub-circuit, there are four input pairs requiring IECs.

Combining all IECs and NICs for the four sub-circuits and solving the resulting ILP, we get a VIP that inserts a single isolator into each of the input lines $\mathbf{X}_1{}^i$, $\mathbf{X}_1{}^r$, $\mathbf{X}_2{}^i$, $\mathbf{X}_2{}^r$, as depicted in Figure 3.9. It may seem surprising that four isolators suffice to decorrelate what is a big circuit by SC standards. The reason for this is that the circuit has many CI pairs due to its many multiplexers (which have CI inputs, as discussed in Example **3.2**); hence many correlated inputs do not need decorrelation. The optimality of this solution can be easily confirmed by noting that we need at least four isolators, one for each of the SN pairs {$\mathbf{A}^i$, $\mathbf{X}_1{}^i$}, {$\mathbf{A}^r$, $\mathbf{X}_1{}^r$}, {$\mathbf{B}^i$, $\mathbf{X}_2{}^i$}, {$\mathbf{B}^r$, $\mathbf{X}_2{}^r$}, to decorrelate the circuit, and therefore four is a lower bound for isolator counts. It turns out that IIBD requires 34 isolators to validly decorrelate the entire circuit. Also, if we ignore the presence of CI, then VAIL requires 30 isolators for decorrelation.

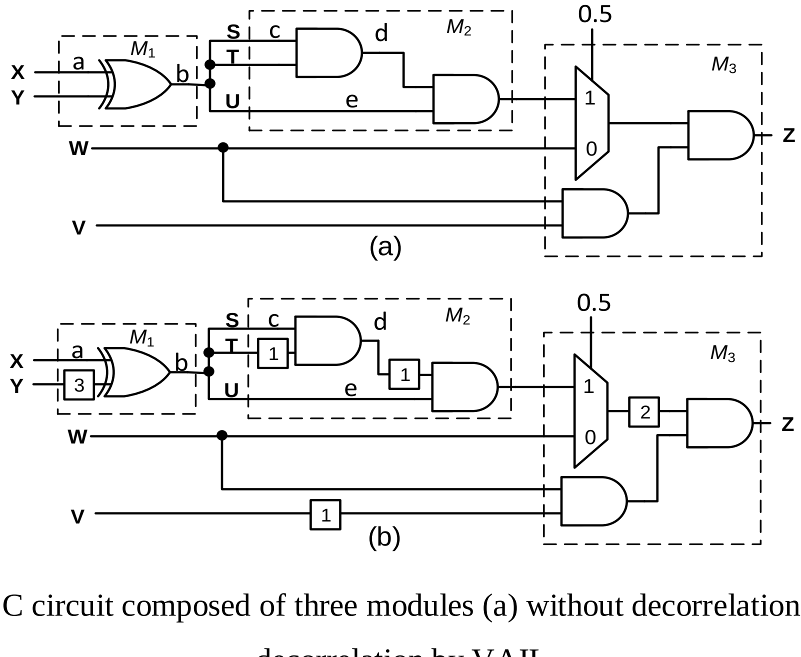


Figure 3.10 SC circuit composed of three modules (a) without decorrelation, and (b) after decorrelation by VAIL

SC Systems: So far, we have discussed the case where only primary input SNs need decorrelation. In practice, however, an SC system or circuit may be constructed from interconnected modules, each of which is a stochastic component requiring its own inputs to be independent. This occurs when designing using a library of SC modules. In such cases, simply decorrelating the primary inputs of the overall system cannot guarantee correct functioning, since additional independence constraints are imposed on the internal lines supplying the modules. A simple special case is seen in single-input circuits like the squarer of Figure 3.1*a*. Obviously, inserting an isolator in the primary input *x* does not provide decorrelation. The squarer is best viewed as a system comprising a fanout network and an internal AND module whose two inputs must be decorrelated.

Here, we show how VAIL can be used to decorrelate a more complex system using the circuit in Figure 3.10*a* as an example. This circuit is intended to compute the multivariate polynomial $Z = 0.5WV(X + Y - 2XY)^3 + 0.5W^2V$. Assume the circuit is composed of three modules M_1, M_2 and M_3. Module M_1 consists of an XOR gate that computes $F_{M_1}(A, B) = A + B - 2AB$. Module M_2 uses two AND gates to compute $F_{M_2}(A, B, C) = ABC$, and when its three inputs are independent copies of a single SN, module M_2 computes the cube of that SN. Module M_3 contains a MUX and two AND gates, and computes $F_{M_3}(A, B, C, D) = 0.5(A + B)CD$. All these modules (except the MUX of module M_3 which has CI inputs) need independent inputs to function correctly. We will only consider decorrelation of module M_2 for illustration purposes. The decorrelation constraints for the other modules can be deduced similarly.

Suppose the four input SNs of the system **X**, **Y**, **V**, **W** are correlated, and let **S**, **T**, **U** be the three SN inputs of M_2. To ensure that M_2 works as if its three inputs were independent, we need to impose ECs and IECs on the inputs of M_2 with respect to the output of M_2. While ECs can be imposed to M_2 directly, IECs cannot, since M_2 is not a standalone SC circuit and we do not have the correlation information about **S**, **T**, **U**. We cope with this problem by extending the path constraints from the inputs of M_2 to the primary inputs of the system. The IECs for M_2 can then be formulated as follows. Let P_i

denote a path from an input i of M_2 to the output of M_2, and let R_i denote a path from a primary input of the system to the input i of M_2. For any paths R_i and R_j originating from the same or correlated primary inputs, paths R_i-P_i and R_j-P_j need to contain different numbers of isolators. For example, paths *abcd* and *abe* must contain different numbers of isolators, since they originate from the same primary input *a*. The IECs force correlated SNs to arrive at the output of module M_2 without correlation.

Solving the ILP with constraints for each of the modules results in the decorrelated system shown in Figure 3.10*b*. VAIL decorrelates the system with a total of eight isolators, which is the optimal isolator count in this case.

3.6 Summary

We have examined in depth the behavior of general stochastic circuits under the influence of correlation. We developed a basic theory for isolator placement and obtained conditions for a placement to be valid. The theoretical analysis reveals that isolator placement is a complex problem with subtle features, such as changing the function of the target circuit in unexpected ways. We introduced the problem of optimal isolator placement, and presented VAIL, an ILP-based algorithm to solve it. Our experiments show that VAIL can find valid, low-cost isolator placements for relatively large stochastic circuits.

CHAPTER 4

Sequential Stochastic Circuits

While combinational stochastic circuits are relatively easy to design, their limitations such as having few implementable functions and issues related to random fluctuation and correlation call for sequential methods. The theory underlying sequential stochastic circuits is not fully understood, however, limiting the use of sequential circuits to a few classes, notably the up/down-counter-based (UCB) circuits. This chapter identifies and investigates two new sequential circuit classes of particular interest to SC, namely shift-register-based (SRB) and optimal-modulo-counting (OMC) circuits. The properties of these two circuit classes are studied in depth, and their central role in SC is demonstrated. This leads to several important applications, such as an algorithm MOUSE for SRB design optimization, and a systematic method for controlling rounding policy in OMC designs. Finally, the chapter discusses the impact of autocorrelation on independently-designed sequential circuits, and describes a mitigation method for it. The material in this chapter has been published in [70] and [74].

4.1 Role of Sequential Elements in SC

Much previous SC work has focused on combinational designs, partly because they are the easiest to deal with. However, it is increasingly clear that sequential elements play an indispensable role in many aspects of SC. For example, the RFE removal method described in Chapter 2 and the decorrelation method discussed in Chapter 3 both rely on sequential components. The random number sources (RNSs) driving the input bit sequences of stochastic circuits are also inherently sequential. Further, SC synthesizers like

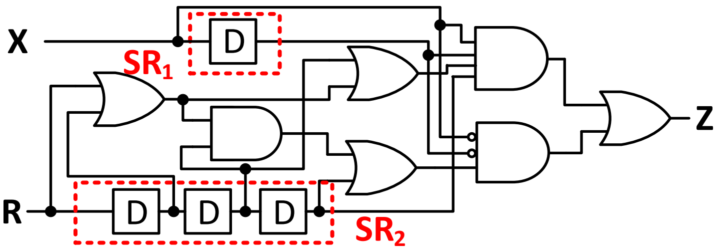


Figure 4.1 Gate-level design of the STRAUSS circuit in Figure 2.6*a*. It requires two copies of variable SN **X** and four independent constant SNs generated by shift registers SR1 and SR2.

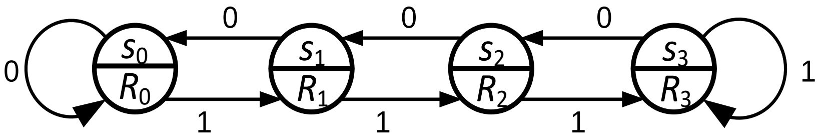


Figure 4.2 State transition graph (STG) for a 4-state Moore-type UCB circuit.

STRAUSS and ReSC require internal constant SNs and independent copies of variable SNs, which also call for sequential components to generate.

Example 4.1: Figure 4.1*a* shows a gate-level design of the STRAUSS circuit in Figure 2.6*a* that computes the (inverted) bipolar stochastic function $F(X) = 0.4375 – 0.25X – 0.5625X^2$. Although $F(X)$ is a single-input function, its combinational part requires two independent copies of **X** and four independent constant SNs $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{R}_3$, $\mathbf{R}_4$, each of value 0.5. The constant SNs can be generated by a 4-bit LFSR, as is done in [6], or they can be produced by a chain of three isolators if a random source **R** is available, as in Figure 4.1*a*. Generation of all six independent inputs can thus be done by two shift registers as indicated in the figure. Taken together, they make the final circuit highly sequential. □

Besides their foregoing ancillary use, sequential components are also explicitly used to implement arithmetic functions that are hard to realize combinationally, such as the update node for LDPC code decoders [33][68], an ad hoc design that we briefly discussed in Chapter 1. Much of the literature on systematic sequential designs, on the other hand, goes back to early ADDIE designs [29] that involve *up/down-counter-based* (UCB) circuits. As depicted in Figure 4.2, the STG of a single-input UCB circuit has a chain-like structure where each state can only be reached from two adjacent states. Prior work has led to several variants of this basic structure. ADDIE [29] developed by Gaines is a sequential design built around a UCB circuit, which can be used to approximate complex functions like square-rooting and division by carefully constructing a global feedback loop according to the function being implemented. Brown and Card use UCB circuits to implement activation functions for their early artificial neural network design [16]. They were able to approximate useful functions like exponentiation and hyperbolic tangent by manually setting the output values associated with each state. More recently, Li et al. developed a sequential SC synthesizer based on the UCB structure which they refer to as a “linear FSM” [52]. This method significantly extends the applicability of [16] to a wide range of functions by using an optimization algorithm to automatically determine the best output values associated with each state.

It is noteworthy that none of the above UCB models applies to circuits decorrelated by isolation developed in Chapter 3, or circuits generated by CEASE discussed in Chapter 2. Isolation works by using DFFs to intentionally delay signals. These DFFs create feed-forward shift registers, so we refer to the resulting sequential stochastic circuits as *shift-register-based* (SRB). On the other hand, CEASE removes constant-induced RFEs by maintaining an accurate running sum of the output values. It does so by using a modulo counter that keeps track of the unreleased output values in its states, so we call CEASE-generated circuits *optimal-modulo-counting* (OMC) circuits. While these two new classes of sequential stochastic designs are very useful as we have seen in previous chapters, their sequential properties have not yet been thoroughly studied and consequently remain poorly understood. This chapter first reviews the basics of sequential stochastic circuits. It then points out some limitations of the UCB approach and explores the SRB and OMC classes. Analysis of these two new types of circuits reveals that they have some distinct advantages, which lead to new opportunities for improving SC designs.

4.2 Behavior of Sequential Stochastic Circuits

This section reviews some analysis tools for sequential stochastic circuits, which will serve as the foundation for analyzing SRB and OMC circuits.

To characterize the behavior of a sequential circuit, it is convenient to model the circuit using a state-transition graph (STG) that describes the finite-state machine (FSM) implemented by the circuit. The state-transition behavior is probabilistic due to the randomness of the input SNs. Sequential stochastic circuits implement complex functions by providing multiple states that are visited in different frequencies. Each state can realize a different combinational stochastic function. Consequently, the overall function depends on two factors: (1) the frequency or probability $\pi_i = p(S = s_i)$ for the circuit to visit a state s_i, and (2) the corresponding output function F_i, which is the combinational stochastic function implemented in state s_i. Specifically, suppose an n-input, single-output sequential circuit C realizes an FSM M with q states s_0, s_1, …, $s_{q\text{-}1}$. The stochastic function C implements is:

$$p\mathrm{z} = F_M\,(M, p\mathcal{X}) = \boldsymbol{\pi}\mathbf{F}^{\mathrm{T}} = \textstyle\sum_i \pi_i F_i \tag{4.1}$$

where $\boldsymbol{\pi} = [\pi_0, \pi_1, \ldots, \pi_{q-1}]$ is the row vector of state probabilities which sum to 1, while $\mathbf{F} = [F_0, F_1, \ldots, F_{q-1}]$. Here $F_i = F(f_i, p\mathcal{X})$ and f_i is the Boolean function M implements when $S = s_i$. Intuitively, Equation (4.1) can be understood as a weighted sum of π_i's, with the corresponding weights being F_i's. Depending on the structure of the STG, π_i can take a very complex form, allowing sequential circuits to implement or approximate arithmetic functions that are otherwise hard to realize with combinational stochastic circuits.

Example 4.2: Consider the circuit C_{UN} in Figure 1.5*b*, which is repeated in Figure 4.3*a*. It is the update node used in LDPC code decoders [33]. The STG of C_{UN} is shown in Figure 4.3*b*. Clearly, C_{UN} implements an FSM of the Moore type, as its output depends solely on the current state. When $S = s_0$, the output is a constant 0, and thus $f_0(x, y) = 0$ and

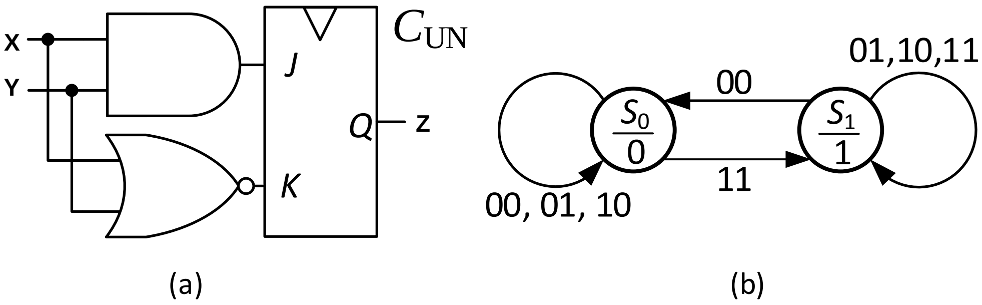


Figure 4.3 (a) Update node C_{UN} used in stochastic LDPC decoders [33], and (b) its state-transition graph.

$F_0 = F(f_0, p_{\mathcal{X}}) = 0$. Similarly, $f_1(x, y) = 1$ and $F_1 = F(f_1, p_{\mathcal{X}}) = 1$. Furthermore, as we will see later, the probabilities of C_{UN} being in states s_0 and s_1 are $\frac{(1-X)(1-Y)}{XY+(1-X)(1-Y)}$ and $\frac{XY}{XY+(1-X)(1-Y)}$, respectively. Plugging these derived values into Equation (4.1) yields C_{UN}'s stochastic function $Z = \frac{XY}{XY+(1-X)(1-Y)}$. Note that this function cannot be implemented exactly by a combinational stochastic circuit. □

Next, we discuss how to compute state probabilities π_i's. State transitions of a sequential stochastic circuit C are probabilistic, and their behavior can be viewed as a Markov chain. This means that given the current state of C, the next state that C visits will be independent of all the previous state visits that C has made [30]. For example, if the current state of the circuit C_{UN} in Figure 4.3 is s_0, the probability of the next state being s_1 is equal to $p_{\mathcal{X}}(1, 1)$, the probability that both the inputs are 1, regardless of what states C has visited previously. Viewing C as a Markov chain enables us to obtain C's state probabilities as a steady-state distribution. Specifically, when C has finite states with each of them being reachable from all other states, then the state probabilities π_i's under steady-state distribution are the solution to following equation

$$\boldsymbol{\pi}\mathrm{P} = \boldsymbol{\pi} \tag{4.2}$$

where and P is a $q \times q$ state-transition probability matrix, whose (i, j)-th element $\mathrm{P}_{i,j}$ denotes the probability of transitioning from state i to state j. This equation is called an *equilibrium equation*, as it describes the state probabilities when a circuit reaches its equilibrium.

Example 4.2 (contd.): Returning to the LDPC update node example: to compute the state probabilities for C_{UN} in Figure 4.3*a*, we first write C_{UN}'s equilibrium equation using Equation (4.2):

$$\boldsymbol{\pi}\begin{bmatrix} 1-p_{\mathcal{X}}(1,1) & p_{\mathcal{X}}(1,1) \\ p_{\mathcal{X}}(0,0) & 1-p_{\mathcal{X}}(0,0) \end{bmatrix} = \boldsymbol{\pi} \tag{4.3}$$

Here the state-transition probability matrix can be obtained directly from the STG given in Figure 4.3*b*. For example, $\mathrm{P}_{0,1} = p_{\mathcal{X}}(1, 1)$, because the probability for the circuit to transition from state s_0 to state s_1 is equal to $p_{\mathcal{X}}(1, 1)$, the probability that both inputs are 1. Solving Equation (4.3) for $\boldsymbol{\pi}$ with the constraint $\pi_0 + \pi_1 = 1$ leads to $\boldsymbol{\pi} = \left[\frac{p_{\mathcal{X}}(0,0)}{p_{\mathcal{X}}(0,0)+p_{\mathcal{X}}(1,1)}, \frac{p_{\mathcal{X}}(1,1)}{p_{\mathcal{X}}(0,0)+p_{\mathcal{X}}(1,1)}\right]$. When $\mathbf{X}_1$ and $\mathbf{X}_2$ are independent, the state probabilities become $\boldsymbol{\pi} = \left[\frac{(1-X)(1-Y)}{XY+(1-X)(1-Y)}, \frac{XY}{XY+(1-X)(1-Y)}\right]$, as expected. □

Equation (4.3) provides a way to compute state probabilities and serves as an important tool in designing and analyzing sequential stochastic circuits. For instance, the sequential synthesizer proposed in [52] uses a UCB circuit to approximate a target function. It does so by first computing the state probabilities $\boldsymbol{\pi}$, which has the form $\pi_i = \frac{r(X)^i - r(X)^{i+1}}{1-r(X)^n}$ for any single-input q-state UCB circuits. Here $r(X) = \frac{1-X}{X}$ and $X \neq 0.5$. The output values associated with each state are then determined in a way that results in the UCB circuit best approximating the target function.

While prior work has focused mainly on UCB circuits, we recently discovered that other types of circuits, such as SRB and OMC, are also very useful in SC design. This is mainly due to their distinct STG structures, which lead to several highly desirable features.

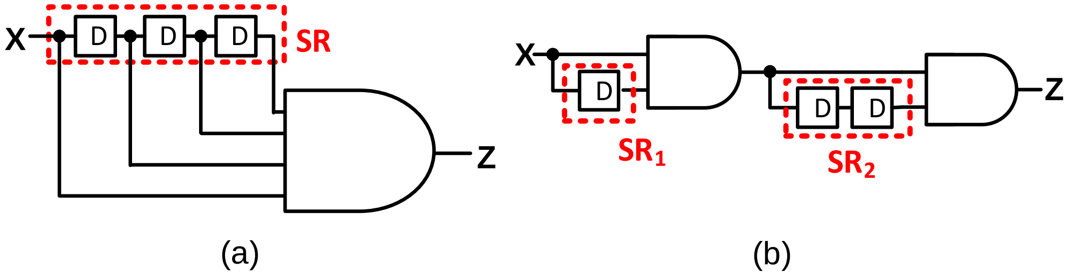


Figure 4.4 Implementations of $F(X) = X^4$: (a) canonical SRB design, and (b) non-canonical SRB design.

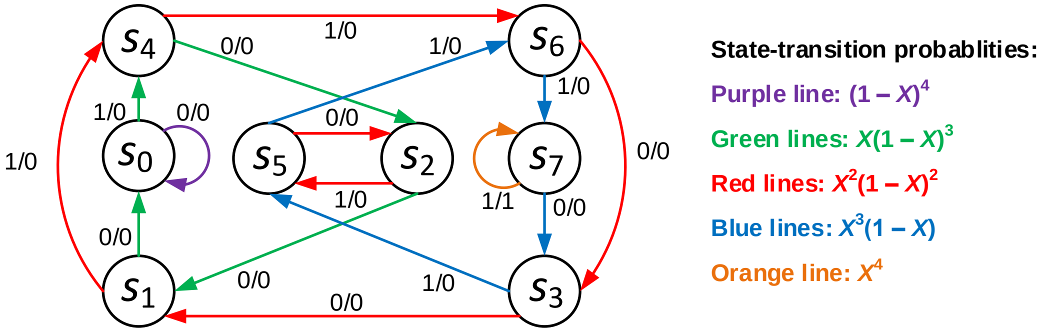


Figure 4.5 STG for the SRB circuit of Figure 4.4*a*. State transitions occurring with the same probability are marked in the same color.

4.3 Shift-Register-Based Circuits

This section describes the functions and some useful properties of shift-register-based (SRB) circuits. It also discusses an optimization method enabled by SRB circuits' special state-transition features.

SRB circuits are a class of sequential stochastic circuits formed by inserting feed-forward DFFs into a combinational circuit. This type of circuit includes those decorrelated by isolation. An SRB circuit implements a classical form of FSM called a *definite* or *finite-input-memory machine* [43]. The name reflects the fact that the current state of an SRB circuit is determined by the most recent *N* inputs for some fixed *N*. In other words, for an

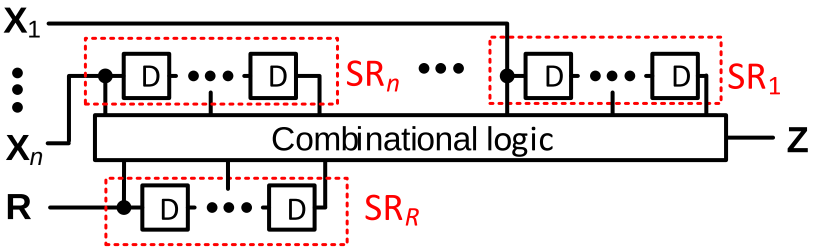


Figure 4.6 A general canonical single-output SRB circuit taking $\mathbf{X}_1$, $\mathbf{X}_2$, …, $\mathbf{X}_n$ as variable inputs and **R** as a constant input.

SRB circuit, every length-*N* input sequence is a synchronizing sequence that takes the circuit to a known state, regardless which state it originally was in. For example, the SRB circuit with three DFFs in Figure 4.4*a* has an input memory of $N = 3$, because its current state can be determined by three most recent inputs. The circuit's STG is depicted in Figure 4.5, which demonstrates its definiteness. For instance, applying a 3-bit sequence like 101 leads it to s_5, regardless of the initial state.

While a definite FSM can have many implementations, it has a canonical form which is an SRB circuit with each shift register taking its input directly from a primary input [43]. Figure 4.6 shows a canonical SRB circuit for SC purposes, where the shift registers generate multiple independent copies of the variable and constant inputs. The circuits in Figure 4.4*a* and Figure 4.1*a* are in canonical form. The SRB circuit in Figure 4.4*b* is a non-canonical definite circuit since the internal shift register SR_2 has an internal signal as its input. We will focus on the canonical SRB forms of definite machines, as all non-canonical SRB circuits can be made canonical without modifying their stochastic functionality.

We first examine the stochastic functions implemented by SRB circuits. Recall that in Chapter 3 we described the functionality of an arbitrary SRB circuit using a clocked Boolean function (CBF) [66], which allows us to model the circuit as if it were combinational. This section, on the other hand, focuses on analyzing canonical SRB circuits using Equation (4.1), which was developed specifically for sequential circuits.

While these two different analysis approaches are both valid, the latter one reveals some optimization opportunities for SRB circuits that are otherwise hard to see.

We consider circuits containing a single shift register; the multiple shift-register case is similar. A shift register SR composed of N DFFs has 2^N states. Let s_i denote the state when SR stores d_i, the radix-2 form of i. For example, when SR is composed of three DFFs and is in state s_6, the DFFs are storing $d_6 = 110$. SR's STG structure is a binary, directed de Bruijn graph, where two nodes s_i and s_j are connected if d_j is a shifted value of d_i, with the empty position filled with 0 or 1. For example, the 8-node de Bruijn graph in Figure 4.5 has an outgoing edge from s_7 to s_3, because $d_3 = 011$ is obtained by shifting a 0 into $d_7 = 111$.

The first step towards analyzing SRB circuits is to compute the state probabilities. While Equation (4.2) can be used to obtain the steady-state distribution of an SRB circuit, here we use an intuitive approach based on the properties of SRB circuits. With a shift register composed of N DFFs, associate a set of state groups $G_0, G_1, \ldots, G_N$, where state s_i belongs to group G_j, if the number of 1s in d_i is j. For example, s_3, s_5 and s_6 in the STG of Figure 4.5 belong to G_2 because $d_3 = 011$, $d_5 = 101$, and $d_6 = 110$ all contain two 1s. Since the FSM has an input memory of $N = 3$, the current state is determined by the most recent three inputs, which are stored in the DFFs. This implies that π_i, the probability of visiting s_i, depends only on the most recent three inputs. Since the probability of seeing a 1 in the input SN **X** is X, the probability of receiving a 3-bit pattern with j 1s is $X^j(1 - X)^{3-j}$. For example, the probability of receiving the pattern 101 is $X^2(1 - X)$, which is π_5, the probability of visiting s_5. It follows that in a shift register with N DFFs, the steady-state probability distribution of $s_i \in G_j$ is $\pi_i = X^j(1 - X)^{N-j}$. This leads immediately to the next result.

Theorem 4.1. A single-input SRB circuit C having N DFFs implements the stochastic function $F(X) = \sum_{j=0}^{N} \sum_{i:s_i \in G_j} F_i \, X^j (1 - X)^{N-j}$.

Proof: From Equation (4.1), we know that $F(X) = \Sigma_i F_i\pi_i$, which sums over all state probabilities weighted by the corresponding output functions. Also recall that an SRB circuit allows dividing its states into different state groups G_j's. This enables us to reform the single summation in Equation (4.1) into a two-step summation: first summing over all state groups and then summing over all state probabilities in each state group. Consequently, $\Sigma_i F_i\pi_i = \sum_{j=0}^{N} \sum_{i:s_i \in G_j} F_i \pi_i$. Finally, using the fact that $\pi_i = X^j(1 - X)^{N-j}$ when $s_i \in G_j$, we obtain $F(X) = \sum_{j=0}^{N} \sum_{i:s_i \in G_j} F_i X^j (1 - X)^{N-j}$. □

Using Theorem **4.1**, we can quickly compute or verify the stochastic function an SRB circuit implements. For example, the circuit in Figure 4.4*a* has $F_0 = F_1 = \ldots = F_6 = 0$, $F_7 = X$ and $n = 3$. Further, $\pi_7 = X^3$ since $s_7 \in G_3$. From Theorem **4.1**, we know immediately that it computes $F(X) = F_7 \cdot \pi_7 = F_7 \cdot X^3 = X^4$, as expected.

Having discussed the functions of SRB circuits, we next focus on their definiteness property. Definiteness has a useful role in SC design. As discussed previously, a sequential stochastic circuit's functionality depends on the state probabilities in the circuit's steady state, which can be achieved after sufficient "warmup" time has elapsed. In other words, when its input SNs change value, a sequential circuit cannot respond completely until steady state is reached. Such warmup delays can be indefinite in length for general circuits. However, the warmup delay of an SRB circuit is bounded by its finite input memory—a definite advantage.

Figure 4.7 shows two SC designs realizing a complex single-input function *tanh*(2*X*). One is a UCB design from [16]; the other is a new SRB design of similar accuracy. Figure 4.7*c* compares the errors of the two circuits as the input changes. Both are fed with a bipolar SN of value $X = 0.9$ for a warmup period, after which X is changed to 0.3. The figure plots the average errors against the number of clock cycles after the value transition takes place. The SRB design reaches steady state 4 cycles after the input transition, while the UCB design takes about 12 cycles. In other words, this SRB design is

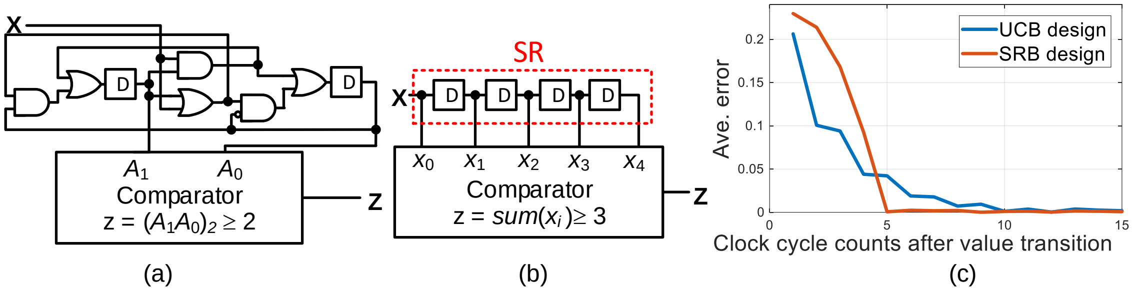


Figure 4.7 SC implementations of $Z = tanh(2X)$ using (a) a UCB design [16], and (b) an SRB design. (c) Error comparison between the two designs when input X changes from 0.9 to 0.3.

guaranteed to completely adjust to its new input value after 4 clock cycles. The short response time of the SRB design is due to its definiteness property, which is important in delay-sensitive applications like real-time object detectors. The above observations lead directly to the following result.

Theorem 4.2. The number of clock cycles required by an SRB circuit to reach a steady state is at most N, the input memory length.

We will see later in this chapter that such finite warmup time of SRB circuits is also highly desirable when dealing with autocorrelation, an accuracy-reducing factor for sequential stochastic circuits.

So far, the chapter has been analyzing SRB circuits using their *state* probabilities. However, as we will see shortly, it is sometimes helpful to examine sequential circuits from the viewpoint of their *state-transition* probabilities. To obtain state-transition probabilities, we expand F_i's in Equation (4.1) using $F_i = F(f_i, p_{\mathcal{X}}) = \sum_b f_i(b)p_{\mathcal{X}}(b)$. This leads to $Z = \sum_i \sum_b f_i(b)\pi_i p_{\mathcal{X}}(b)$. Notice here that the term $\pi_i p_{\mathcal{X}}(b)$ is the probability that the machine is in state s_i and the current input is b. This is the probability of a state transition from s_i given input b, which we denote by $p(T_i^{(b)})$. For example, in the STG of Figure 4.5, $p(T_2^{(0)}) = \pi_2 p_{\mathcal{X}}(0)$ is the probability of being in s_2 with input 0; this is also the probability of a transition from s_2 to s_1. We can now describe the stochastic function of a (Mealy) sequential circuit using $p(T_i^{(b)})$ and the corresponding output value $f_i(b)$ as follows:

$$Z = \sum_i \sum_b f_i(b)p(T_i^{(b)}) \tag{4.4}$$

An interesting and useful fact is that many state transitions of an SRB circuit occur with the same probability, a phenomenon seldom seen in UCB and other types of circuits. For example, in Figure 4.5, $p(T_7^{(0)}) = p(T_5^{(1)})$, i.e., the transition from s_7 to s_3 occurs equally frequently as that from s_5 to s_6. We define state transitions as *stochastically equivalent* if they have the same probability. Equation (4.4) implies that the output values $f_i(a)$ and $f_j(b)$ of two stochastically equivalent transitions $T_i^{(a)}$ and $T_j^{(b)}$ can be switched without changing the overall stochastic function. This directly leads to the following result.

Theorem 4.3. Suppose a sequential stochastic circuit has state transitions with output values $f_i(a) \neq f_j(b)$. These values can be switched without changing the underlying stochastic function if and only if $p(T_i^{(a)}) = p(T_j^{(b)})$.

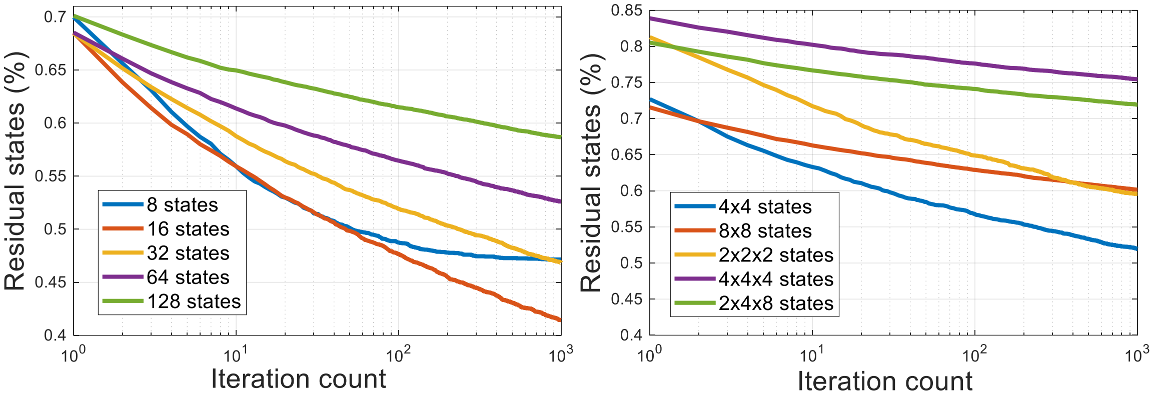
As Theorem **4.3** suggests, in sequential stochastic circuit design, especially in the SRB case, a set of distinct FSMs $\mathscr{F}$ can implement the same stochastic function, but with potentially different costs. This raises an interesting new design question: Find an FSM in $\mathscr{F}$ that, when standard optimization techniques are applied, results in the greatest hardware cost reduction. Next, we describe a Monte-Carlo-based algorithm *MOUSE* (**M**onte-Carlo **O**ptimization **U**sing **S**tochastic **E**quivalence) for optimizing SRB circuits. Given an SRB design, MOUSE randomly searches for a better design by switching state function values based on Theorem **4.3** using a user-defined cost metric. For illustration purposes, the cost metric used here is state count. In each iteration, MOUSE explores the space of stochastically equivalent transitions of a target circuit, and generates a new circuit implementing the same stochastic function by randomly switching equivalent state function values. A fast state reduction procedure STAMINA [67] assesses the new circuit's cost. MOUSE keeps the best design it finds until a stop criterion is met. The algorithm is

```
Input:    $C$: Target SRB circuit
          $iter\_lim$: Iteration limit
Output:   $C_{OUT}$: Result SRB circuit with reduced cost
Step 1    Set $Best\_Cost$ = state count of $C$
          Set $iter\_count = 0$
          Set $f_i$ to be the Boolean function $C$ implements when in state $s_i$
Step 2    For each stochastically equivalent transition class S in $C$
              Set $w$ = sum over all $f_i(b)$'s associated with S
              Randomly pick $w$ $f_i(b)$'s and assign them output value 1
              For the unpicked $f_i(b)$'s, assign them output value 0
Step 3    $iter\_count = iter\_count + 1$
          Set $C_{CUR}$ = circuit obtained by applying state reduction to $C$
          Set $Cost$ = state count of $C_{CUR}$
Step 4    If $Cost < Best\_Cost$, then Set $Best\_Cost = Cost$, $C_{OUT} = C_{CUR}$
Step 5    If $iter\_count < iter\_lim$, then Go to Step 2
          Else output $C_{OUT}$
```

Figure 4.8 Pseudo-code for optimization algorithm MOUSE.

summarized in Figure 4.8. The final circuit, which may implement a very different FSM, preserves all the stochastic properties of the original SRB design, including its output function and the property of finite input memory.

We assess MOUSE's performance using random circuits. Figure 4.9*a* plots the percentage of residual states against iteration count for single-input SRB circuits produced by averaging over 500 random circuits. The state count is frequently reduced by more than 50% within 1,000 iterations. For small FSMs such as SRB circuits with three DFFs, the state reduction process slows down and reaches its optimal value within 1,000 iterations. This is because with a small FSM, few possibilities exist for function value switches, so MOUSE can usually find a good solution after just a few iterations. Similar phenomena are observed for multi-input SRB circuits, as shown in Figure 4.9*b*.



(a) (b)

Figure 4.9 Percentage of residual states vs. iteration count for (a) single-input SRB circuits, and (b) multi-input SRB circuits.

MOUSE can also be applied to existing SRB designs for hardware cost reduction. For example, the multi-input SRB in Figure 4.1*a* has 16 states. Applying STAMINA directly to this circuit results in 13 states—a 19% decrease in state count. MOUSE, on the other hand, reduces the number of states from 16 to 7 after about 90 iterations. Hence, in this particular case, MOUSE enables one of the original circuit's four DFFs to be eliminated.

4.4 Optimal-Modulo-Counting Circuits

Having discussed SRB circuits, we introduce another new class of circuits called *optimal-modulo-counting* (OMC) circuits. This section begins by examining the functions of OMC circuits. It then discusses some desirable properties of OMC circuits, including high accuracy and controllable rounding policies.

An OMC circuit, as its name suggests, implements a special type of modulo counter. The FSM of an OMC circuit consists of mutually reachable states that keep a running sum of input values. All the states of an OMC circuit have the same state-transition behavior, which can be characterized by a_i, the number of states the circuit advances on

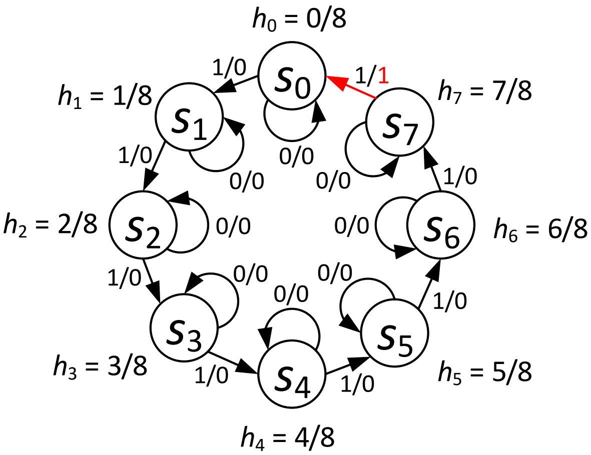


Figure 4.10 State-transition graph for an OMC circuit implementing the scaling function $F(X) = X/8$.

receiving the input pattern b_i. However, the output values are state dependent. Specifically, for a q-state OMC circuit, each state s_i holds a residual accumulated value $h_i = i/q$. Each state transition adds a value of a_i/q to h_i. Whenever h_i becomes larger or equal to 1, it overflows by setting the output to 1.

Example 4.3: Figure 4.10 shows a simple example of an 8-state OMC circuit C for illustration purposes. It implements the scaling function $F(X) = X/8$. Here, C is a single-input single-output circuit, with all its states having exactly the same state-transition behavior: they stay in the same state when $x = b_0 = 0$, but move one state forward and at the same time add 1/8 to h_i when $x = b_1 = 1$. In other words, the number of states C advances on receiving a 0 and 1 are $a_0 = 0$ and $a_1 = 1$, respectively. The output value does depend on the state and the residual value, which only overflows when $S = s_7$ and $x = 1$, as highlighted in red in the figure. The function $F(X)$ that C implements can be easily derived. It is not hard to see that whenever eight 1s are received, the residual value will definitely overflow with a 1 released to the output line. The probability that a 1 appears in C's input line is X, and thus $F(X) = X/8$, which is the desired function. □

As discussed in Chapter 2, combinational stochastic circuits with constants, which can be converted to OMC circuits using CEASE, implement functions in the form of

Equation (2.5). Next, we will show that all OMC circuits, whether generated by CEASE or not, implement a function in the form of Equation (2.5) accurately up to an unavoidable rounding error. Depending on the rounding policy, rounding may produce a 1-bit error when the SN length is unable to represent certain values exactly. For example, an SN of odd length N cannot represent 1/2 exactly, but one of length $N \pm 1$ can.

Theorem 4.4. For an n-input, q-state OMC circuit C with N-bit SNs, the number of 1s in its output SN **Z** is

$$N_{1,\mathbf{Z}} = \sum_{i=0}^{m-1} \frac{a_i}{q} k_i + \tilde{\epsilon} \tag{4.5}$$

where k_i is the number of bit-patterns b_i received by C, while a_i is the number of states C advances when a b_i pattern is received. The residual (non-outputted) error due to rounding is $\tilde{\epsilon} = \frac{a_{-1}}{q} - \epsilon$, where a_{-1} is such that $s_{a_{-1}}$ is the initial state of C and $\epsilon \in [0, 1)$. Further, **Z**'s value is

$$Z = \sum_{i=0}^{m-1} \frac{a_i}{q} p_{\mathcal{X}_\mathrm{V}}(b_i) + \frac{1}{N} \mathbb{E}(\tilde{\epsilon}) \tag{4.6}$$

where $p_{\mathcal{X}_\mathrm{V}}(b_i)$ is the probability of C receiving the input pattern b_i. □

The proof appears in Appendix A.4. Here, we give an intuitive explanation of Equations (4.5) and (4.6). In Equation (4.5), the number of 1s in the output $N_{1,\mathbf{Z}}$ reflects the number of times that the modulo counter goes beyond 1 and overflows. The modulo counter accumulates the value $\frac{a_i}{q}$ for each pattern b_i it receives, and there are k_i occurrences of the pattern b_i. Therefore, $\frac{a_{-1}}{q} + \sum_{i=0}^{m-1} \frac{a_i}{q} k_i$ is the total value accumulated in the modulo counter at the end, where $\frac{a_{-1}}{q}$ accounts for the value initially stored in the counter and can be adjusted by configuring the counter's initial state. Since the number of 1s in **Z** must be an integer, $N_{1,\mathbf{Z}}$ must be $\left\lfloor \sum_{i=0}^{m-1} \frac{a_i}{q} k_i + \frac{a_{-1}}{q} \right\rfloor$. In Equation (4.5), this floor operation is captured by the term $\tilde{\epsilon}$, which depends on the initial state of the counter and must be less than one, because the residual (non-overflow) value stored in the counter must be less than one. To go from Equation (4.5) to Equation (4.6), first note that the value of **Z** is the expected number of 1s in **Z** divided by *N*. Consequently, $Z = \mathbb{E}\left(\sum_{i=0}^{m-1}\frac{a_i}{q}\frac{k_i}{N}+\frac{\tilde{\epsilon}}{N}\right) = \sum_{i=0}^{m-1}\frac{a_i}{q}\mathbb{E}\left(\frac{k_i}{N}\right)+\frac{\mathbb{E}(\tilde{\epsilon})}{N} = \sum_{i=0}^{m-1}\frac{a_i}{q}p_{\mathcal{X}_{\mathrm{V}}}(b_i)+\frac{1}{N}\mathbb{E}(\tilde{\epsilon})$, since $\mathbb{E}\left(\frac{k_i}{N}\right)$, the expected fraction of the pattern b_i in the input lines, is the probability that b_i occurs.

Recall that in Theorem **4.4**, a_i is the number of states *C* advances on receiving b_i, and q is the total states in *C*. This clearly implies that $a_i \leq q$ for all *i*, so in Equation (4.6) the coefficients $\frac{a_i}{q} \in [0,1]$ are indeed rational numbers in the unit interval. Comparing Equation (4.6) with Equation (2.5), we see that *C* indeed implements a stochastic function in the form of Equation (2.5), the class of functions combinationally implementable with constant inputs. These functions are exact up to an unavoidable rounding error $\frac{1}{N}\mathbb{E}(\tilde{\epsilon})$, which, in the worst case, can only cause a 1-bit difference in $N_{1,\mathbf{Z}}$.

The foregoing discussion leads to the conclusion that OMC circuits can implement combinationally implementable functions accurately (up to only a rounding error), as long as the inputs applied to them are accurate. This property is due to the specially designed modulo-counting STG which allows OMC circuits to keep an accurate running sum of values for later release.

Next, we discuss another special property of OMC circuits: controllable rounding policy. Like any other circuits that compute with finite precision, an OMC circuit can have a rounding error due to the representational limitation of finite-length SNs, as briefly described earlier. For example, a 4-bit SN can only contain 0, 1, 2, 3 or 4 logical 1s. It cannot contain, say, 2.5 logical 1s to represent the value 2.5/4 = 0.625 exactly. If the exact value is 0.625, the number of 1s in the SN must be rounded to an integer closest to 2.5, such as 2 or 3. In the case of OMC circuits, this rounding error is reflected by the term $\tilde{\epsilon}$ in Equation (4.5). While rounding policy is difficult to manage for general sequential

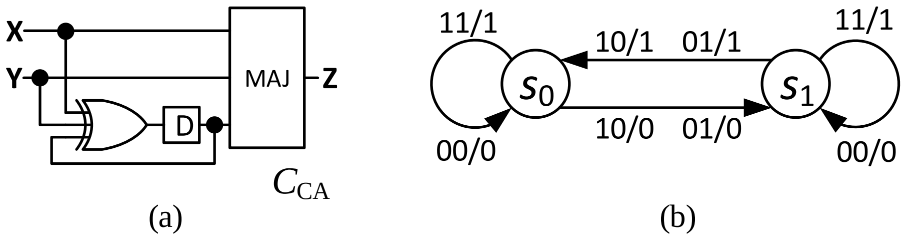


Figure 4.11 (a) OMC scaled adder C_{CA}, and (b) its STG.

stochastic circuits, it is easily controllable for OMC circuits by adjusting their initial states. In [48], the authors show that the final value their ad-hoc OMC adder rounds to depends on the adder's initial state and the rounding direction: truncation (rounding down) or rounding up. Here, we generalize their observation to OMC circuits beyond stochastic adders, and demonstrate analytically that for an arbitrary OMC circuit, different and desirable rounding policies can be achieved by carefully adjusting the circuit's initial state.

Consider an OMC circuit C with q states. Initializing C to state s_0, the first state of the associated modulo counter, is equivalent to truncating the fraction part of the expected number of 1s in the output bit-stream **Z**. This is because C only produces a 1 in the output line whenever its modulo counter overflows. Therefore, when C finishes a computation in a state $s \neq s_0$, the residual value stored in the modulo counter is discarded. Consider again the OMC scaled adder C_{CA} given in Figure 2.2*c*, which is repeated in Figure 4.11. Here we initialize C_{CA} to its first state s_0, and apply the following two inputs; **X** = 00011010 and **Y** = 01100110. Since $X = 3/8$, and $Y = 4/8 = 1/2$, the expected value for the output **Z** is 0.5(X + Y) = 0.5 · (7/8) = 3.5/8, implying that ideally the number of 1s in **Z** should be $N_{1,\mathbf{Z}} = 3.5$, which is obviously not achievable with digital logic circuits. If we work out the stochastic computation, it is not hard to see that in this case, **Z** = 00101010, which contains three 1s, and the machine terminates at state s_1. The residual value 0.5 stored in the modulo counter state s_1 is consequently discarded, and the actual output value is thus truncated to 3/8.

Suppose we now change the adder C_{CA}'s initial state to s_1, and leave the other conditions unchanged. In this case, **Z** = 01010110, which contains four 1s, and C_{CA}

terminates at state s_0. Clearly, C_{CA} now rounds the expected value 3.5/8 to 4/8. This is because initializing C_{CA} to state s_1 implicitly introduces an extra value of 0.5 into the circuit. This circuit will then produce a 1 whenever the residual value is 0.5.

In general, initializing an OMC circuit C to the state $S^{(0)} = s_{a_{-1}}$ implements the following rounding policy with respect to $N_{1,\mathbf{z}}$: “Discard a residual value if it is less than 1 – $\frac{a_{-1}}{q}$, and add a carry of 1 otherwise.” This follows from the fact that $N_{1,\mathbf{z}}$ = $\left\lfloor \sum_{i=1}^{m} \frac{a_i}{q} k_i + \frac{a_{-1}}{q} \right\rfloor$; see Equation (4.5). If the residual value of $\sum_{i=1}^{m} \frac{a_i}{q} k_i$ is less than 1 – $\frac{a_{-1}}{q}$, then even after adding the contribution from the initial state $\frac{a_{-1}}{q}$, the overall residual value will still be discarded by the floor operation.

Summarizing, due to the fact that the precision of a digital system is finite, rounding errors are unavoidable. The rounding policy of an OMC circuit can be adjusted by configuring its initial state. For an OMC circuit with q states, initializing it to state s_0 is equivalent to the rounding policy of truncating the fraction part of $N_{1,\mathbf{z}}$. On the other hand, initializing the circuit to a “middle state,” i.e., to $s_{\lfloor q/2 \rfloor}$, is equivalent to rounding $N_{1,\mathbf{z}}$ to the closest integer.

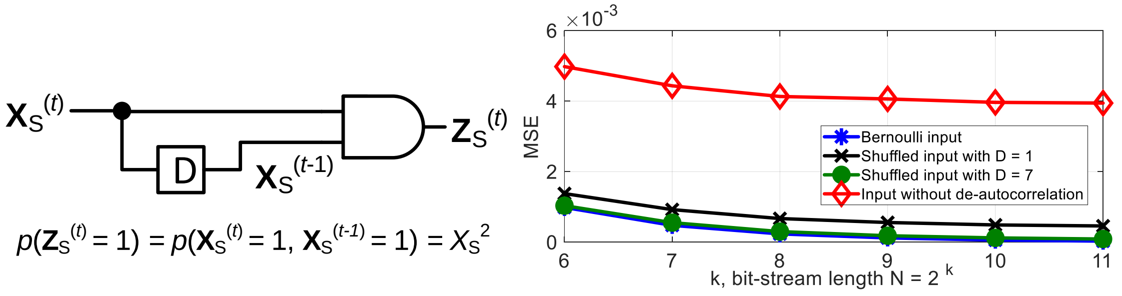
4.5 Autocorrelation

So far, the chapter has focused on individual sequential designs. However, when cascading multiple independent sequential modules, one must exercise extra care to prevent accuracy loss caused by autocorrelation, an accuracy-reducing factor when bits of SNs are not independently generated. This section first briefly introduces autocorrelation in SC. It then discusses some desirable autocorrelation properties that SRB and OMC circuits have. Finally, the section describes a method to mitigate autocorrelation errors in SC designs containing multiple sequential circuits.

Autocorrelation quantifies the correlation between two elements in a single data sequence. The *autocorrelation coefficient* AC(t, s) for an SN **X** describes the similarity between $\mathbf{X}^{(t)}$ and $\mathbf{X}^{(s)}$. For example, applying the definition of Pearson correlation coefficient [34] to the autocorrelation coefficient in the SC context, we obtain AC(t, s) = $\frac{\mathbb{E}[(\mathbf{X}^{(t)}-X)(\mathbf{X}^{(s)}-X)]}{\sigma^2}$, where $\sigma^2 = \mathbb{E}[(\mathbf{X}^{(i)} - X)^2]$ is the variance of $\mathbf{X}^{(i)}$. Thus AC(t, s) measures the similarity between the probabilities of the bits at two time steps t and s. If $\mathbf{X}$ is Bernoulli, i.e., if each bit of $\mathbf{X}$ is independently generated, then $\mathbf{X}$'s autocorrelation coefficient is two-valued with AC(t, t) = 1, while AC(t, s) = 0 for all $t \neq s$ [31].

Most sequential stochastic circuits are designed to work with input SNs that satisfy certain temporal independence constraints such as Bernoulli properties. Undesired autocorrelations can sometimes degrade the accuracy of such circuits dramatically. On the other hand, sequential stochastic circuits themselves can introduce autocorrelations into their output SNs, making them non-Bernoulli [16]. The preceding facts imply that if two independently-designed sequential stochastic circuits are cascaded, the downstream circuit's accuracy may drastically degrade due to autocorrelations introduced by the predecessor circuit. Since both SRB and OMC circuits are sequential designs, we are interested in the impact of autocorrelation on these two types of circuits. It turns out that, both SRB and OMC circuits have some desirable properties related to autocorrelation, which we examine in this section.

First, we illustrate via an example how undesired autocorrelation affects the accuracy of a sequential stochastic circuit. Figure 4.12*a* shows the stochastic squarer C_S we have seen in Chapter 3, which is the standard sequential circuit for computing $Z_S = X_S^2$. To compute the value of $\mathbf{Z}_S$ at any cycle t, this squarer uses its DFF to hold $\mathbf{X}_S^{(t-1)}$, which is then multiplied with $\mathbf{X}_S^{(t)}$ via the AND gate. The AND gate thus computes $p(\mathbf{X}_S^{(t-1)} = 1, \mathbf{X}_S^{(t)} = 1) = p_{\mathbf{X}_S^{(t)}}(1)p_{\mathbf{X}_S^{(t-1)}}(1) = X_S^2$, which implicitly assumes independence between $\mathbf{X}_S^{(t)}$ and $\mathbf{X}_S^{(t-1)}$. Since the output probability should be correct for all t, it is not hard to see that C_S requires all the adjacent bits in its input $\mathbf{X}_S$ to be independent. In other words, it requires $\mathbf{X}_S^{(i)}$ and $\mathbf{X}_S^{(j)}$ to be uncorrelated, for all i, j, where $i = j + 1$. Figure 4.12*b* plots mean squared



(a) (b)

Figure 4.12 (a) Conventional sequential stochastic squarer. (b) MSE vs. bit-stream length for the squarer with inputs that have different autocorrelation levels.

error (MSE) against bit-stream length for C_S using 0.5-valued inputs with different autocorrelation levels. Specifically, the blue line shows the error rate for a Bernoulli input, which converges towards zero quickly as bit-stream length increases. The red line, on the other hand, shows the error rate when the input is produced by the sequential OMC adder C_{CA} in Figure 4.11*a*. This input SN thus exhibits undesired autocorrelation, resulting in C_S having obvious errors. The black line and the green line each indicates an error rate for an autocorrelated input that is de-autocorrelated using a shuffler [47][68], which we will discuss shortly.

Many existing sequential stochastic circuits, including the preceding squarer example, can suffer from accuracy degradation if inputs are autocorrelated [16]. However, this is not true for an OMC circuit *C*, which can therefore be called *autocorrelation-insensitive*. This means that *C* is immune to autocorrelation-induced accuracy degradation. To see this, recall from Theorem **4.4** that *C* implements a stochastic function of the form $\sum_{i=1}^{m} \frac{a_i}{q} p_{\mathcal{X}_V}(b_i) + \frac{1}{N}\mathbb{E}(\tilde{\epsilon})$, which depends only on the joint probability of all the inputs at a given cycle, but not on the joint probability of any individual inputs across different cycles. Theorem **4.5** summarizes the above discussion:

Theorem 4.5. For an OMC circuit *C*, autocorrelation in any of its input SNs does not reduce *C*’s accuracy. □

Having discussed the behavior of OMC circuits with autocorrelation in their inputs, we next investigate the effect these circuits have on autocorrelation in their outputs. This issue is worth looking into, especially when an SN **Z** generated by an OMC circuit is to be processed by some downstream sequential stochastic circuits that are sensitive to autocorrelation. Indeed, directly feeding **Z** to any sequential circuit may degrade accuracy. The red line in Figure 4.12*b* shows the error rate plotted against bit-stream length for the squarer of Figure 4.12*a* using a bit-stream generated by the OMC adder C_{CA} in Figure 4.11*a* as the squarer's input. Here the scaled adder C_{CA} adds the two input SNs of value 0.5, and produces its output SN, also of value 0.5. We see that the MSE does not converge to zero, regardless of the bit-stream length. This clearly shows that the autocorrelation injected by C_{CA} degrades the accuracy of the downstream squarer.

While it is hard to analyze the impact of undesired autocorrelations on an arbitrary sequential stochastic circuit, analyzing SRB circuits defined in Section 4.3 is relatively easy due to their definiteness property. Here, we analyze quantitatively how an autocorrelated SN produced by an upstream OMC circuit affects a downstream SRB circuit. Recall that an SRB circuit can be described by a CBF f_{S}, from which its stochastic function F_{S} can be constructed; see Chapter 3. The joint probability terms in F_{S} can then be analyzed using the Law of Total Probability (LTP) [34], which allows a probability term to be decomposed into several sub-terms, each involving a distinct event. For example, suppose A and B_1, B_2, … B_k are events where the B_i's are disjoint and their union is the entire sample space. Then LTP says

$$p(\mathrm{A}) = \sum_{i=1}^{k} p(\mathrm{A}, \mathrm{B}_i) = \sum_{i=1}^{k} p(\mathrm{A}\ |\mathrm{B}_i)p(\mathrm{B}_i) \tag{4.7}$$

By applying LTP and conditioning on the states of the preceding OMC circuit, F_{S} can be analyzed. We illustrate this via the squarer example.

Example 4.4: Here we use the OMC adder C_{CA} in Figure 4.11*a* as the upstream circuit, followed by the squarer C_{S} of Figure 4.12*a* to illustrate the impact of autocorrelation. Specifically, we feed $\mathcal{X}_{\mathrm{V}} = \{\mathbf{X}, \mathbf{Y}\}$, the two inputs of C_{CA}, with independent Bernoulli SNs, each of value 0.5. Thus $\mathbf{Z}$, the output of C_{CA}, should have a value of 0.5. Since $\mathbf{Z}$ is produced by the sequential circuit C_{CA}, extra autocorrelation is injected into it. We then use $\mathbf{Z}$ as the input SN $\mathbf{X}_{\mathrm{S}}$ to the squarer C_{S}. Obviously, the exact value for C_{S}'s output $\mathbf{Z}_{\mathrm{S}}$ should then be $0.5^2 = 0.25$. To compute the actual value for $\mathbf{Z}_{\mathrm{S}}$ with $\mathbf{X}_{\mathrm{S}}$ as the input, we first construct the stochastic function for C_{S} using its CBF $z_{\mathrm{S}}^{(t)} = x_{\mathrm{S}}^{(t)}x_{\mathrm{S}}^{(t-1)}$, which is:

$$\begin{aligned} Z_{\mathrm{S}} &= p(\mathbf{Z}_{\mathrm{S}}^{(t)} = 1) = \textstyle\sum_b \left[f_{\mathrm{S}}(b)p_{\mathbf{X}_{\mathrm{S}}^{(t)}\mathbf{X}_{\mathrm{S}}^{(t-1)}}(b)\right] \\ &= p_{\mathbf{X}_{\mathrm{S}}^{(t)}\mathbf{X}_{\mathrm{S}}^{(t-1)}}(1,1) = p(\mathbf{X}_{\mathrm{S}}^{(t)} = 1, \mathbf{X}_{\mathrm{S}}^{(t-1)} = 1). \end{aligned} \tag{4.8}$$

We then apply LTP by conditioning on the adder C_{CA}'s state at time $t - 1$:

$$\begin{aligned} Z_{\mathrm{S}} &= p(\mathbf{X}_{\mathrm{S}}^{(t)} = 1, \mathbf{X}_{\mathrm{S}}^{(t-1)} = 1 \mid S^{(t-1)} = s_0)p(S^{(t-1)} = s_0) \\ &+ p(\mathbf{X}_{\mathrm{S}}^{(t)} = 1, \mathbf{X}_{\mathrm{S}}^{(t-1)} = 1 \mid S^{(t-1)} = s_1)p(S^{(t-1)} = s_1) \end{aligned} \tag{4.9}$$

We then use the fact that with Bernoulli inputs, the steady-state distribution of C_{CA}'s states is uniform. This implies that $p(S^{(t-1)} = s_0) = p(S^{(t-1)} = s_1) = 0.5$. Furthermore, $p(\mathbf{X}_{\mathrm{S}}^{(t)} = 1, \mathbf{X}_{\mathrm{S}}^{(t-1)} = 1 \mid S^{(t-1)} = s_0) = p(\mathcal{X}_{\mathrm{V}}^{(t)} = 11, \mathcal{X}_{\mathrm{V}}^{(t-1)} = 11)$. This is because in state s_0, C_{CA} can only output two consecutive 1s when it receives inputs 11, 11. On the other hand, there are five cases where C_{CA} can add two consecutive 1s in state s_1; see Figure 4.13. Each of these cases occurs with probability 1/16, so $p(\mathbf{X}_{\mathrm{S}}^{(t)} = 1, \mathbf{X}_{\mathrm{S}}^{(t-1)} = 1 \mid S^{(t-1)} = s_1) = 5/16$. Summarizing, $Z_{\mathrm{S}} = 0.5 \cdot \frac{1}{16} + 0.5 \cdot \frac{5}{16} = \frac{3}{16} \neq 0.25$, which is a 25% error! In terms of MSE, this error is $(3/16 - 0.25)^2 = 0.004$, which is the value to which the red line in Figure 4.12*b* converges. □

	$\mathcal{X}_{\mathrm{V}}^{(t)}$	$\mathcal{X}_{\mathrm{V}}^{(t-1)}$	$p(\mathcal{X}_{\mathrm{V}}^{(t)}, \mathcal{X}_{\mathrm{V}}^{(t-1)})$
Case 1	11	10	1/16
Case 2	11	01	1/16
Case 3	10	11	1/16
Case 4	01	11	1/16
Case 5	11	11	1/16

Figure 4.13 Five input cases in which the OMC adder C_{CA} in Figure 4.11*a* outputs two consecutive 1s when in state s_1.

Input: C_{OMC}: An OMC circuit

C_{D}: A single-input SRB circuit downstream from C_{OMC}

$\mathcal{X}$: Bernoulli input SNs of C_{OMC}

Output: Z_{D}: C_{D}'s output value when used with C_{OMC}

Step 1 Express C_{D}'s logic function through a CBF

$$z_{\text{D}}^{(t_k)} = f_{\text{D}}(b) = f_{\text{D}}(x_{\text{D}}^{(t_k)}, x_{\text{D}}^{(t_{k-1})}, \ldots, x_{\text{D}}^{(t_1)})$$

where $t_k > t_{k\text{-}1} > \ldots > t_1$.

Step 2 Construct C_{D}'s stochastic function as

$$Z_{\text{D}} = \textstyle\sum_b \left[f_{\text{D}}(b)\, p_{\mathbf{X}_{\text{D}}^{(t_k)}\mathbf{X}_{\text{D}}^{(t_{k-1})}\ldots\mathbf{X}_{\text{D}}^{(t_1)}}(b)\right]$$

Step 3 For all b's such that $f_{\text{D}}(b) = 1$, compute the joint probability $p_{\mathbf{X}_{\text{D}}^{(t_k)}\mathbf{X}_{\text{D}}^{(t_{k-1})}\ldots\mathbf{X}_{\text{D}}^{(t_1)}}(b)$ using LTP by conditioning on the state variable $S^{(t_1)}$ of C_{OMC}. Specifically,

$$p_{\mathbf{X}_{\text{D}}^{(t_k)}\mathbf{X}_{\text{D}}^{(t_{k-1})}\ldots\mathbf{X}_{\text{D}}^{(t_1)}}(b) = \tfrac{1}{q}\textstyle\sum_s \left[p_{\mathbf{X}_{\text{D}}^{(t_k)}\mathbf{X}_{\text{D}}^{(t_{k-1})}\ldots\mathbf{X}_{\text{D}}^{(t_1)}\,|\,S^{(t_1)}}(b\,|\,s)\right]$$

where q is the number of states in C_{OMC}.

Step 4 Use $\mathcal{X}$'s values to compute $\tfrac{1}{q}\sum_s \left[p_{\mathbf{X}_{\text{D}}^{(t_k)}\mathbf{X}_{\text{D}}^{(t_{k-1})}\ldots\mathbf{X}_{\text{D}}^{(t_1)}\,|\,S^{(t_1)}}(b\,|\,s)\right]$ for all s's and all b's such that $f_{\text{D}}(b) = 1$. Output the resulting Z_{D}.

Figure 4.14 Algorithm to compute the output value of an SRB circuit used with OMC-supplied (and therefore autocorrelated) inputs.

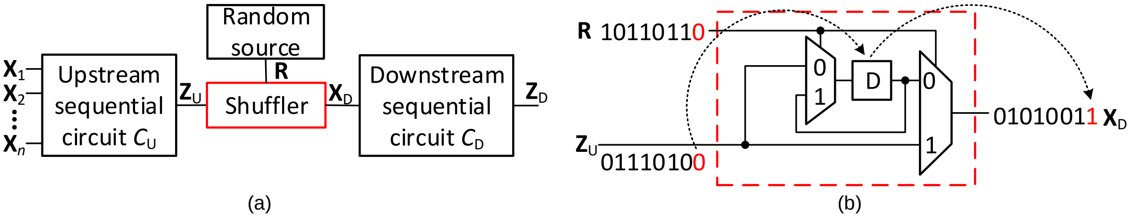


Figure 4.15 (a) Shuffler to mitigate autocorrelation in a downstream sequential circuit. (b) Shuffler of depth $D = 1$.

Example **4.4** shows how to analyze the autocorrelation error introduced by an OMC circuit for given input values. One can also go one step further to compute the expected error by averaging all possible input values. Figure 4.14 provides a pseudo-code algorithm summarizing the preceding analysis.

Finally, we consider how to mitigate autocorrelation induced by an upstream circuit so that it can be coupled efficiently with other sequential stochastic circuits. For this, we adopt a de-autocorrelation method based on shuffling, which has been used for correlation management in prior work [47][68]. In Chapter 3, we showed how shufflers mitigate errors introduced by cross-correlation between SNs. It turns out that shufflers can also mitigate autocorrelation errors. Figure 4.15*a* shows how a shuffler is inserted between an upstream sequential circuit C_U and a downstream circuit C_D, while Figure 4.15*b* depicts the implementation of a shuffler of depth D = 1, i.e., it contains a single DFF. At each clock cycle, the shuffler randomly decides either to output the received bit, or store that bit in a DFF. If the shuffler chooses to store the received bit in a DFF, then the previous bit value stored in the DFF will be released to the output line. As we have seen in Chapter 2, since a shuffler only outputs what it has received, the values of its input and output SNs are the same. However, the location of the 1s can be very different, hence achieving the goal of de-autocorrelation.

A shuffler of depth D can be built with D DFFs. To illustrate the de-autocorrelation capability of shufflers, we use the ongoing example where an OMC adder generates a 0.5-value SN that is consumed by a downstream squarer in Figure 4.12*a*. The black and green lines in Figure 4.12*b* show the accuracy of the squarer with its OMC-circuit-supplied inputs de-autocorrelated by shufflers of depth D = 1 and D = 7, respectively. Obviously, with a depth-7 shuffler, the accuracy of the squarer can be improved to a level similar to a squarer having a Bernoulli input. As a final note, with more DFFs, a shuffler can regenerate a more Bernoulli-like SN, but this comes with the drawbacks of higher hardware cost and longer warmup time to initialize the shuffler.

4.6 Summary

This chapter examined the role of sequential components in SC, and identified two key classes of sequential stochastic circuits, namely SRB and OMC circuits. The chapter also investigated some key properties of SRB and OMC circuits, such as SRB circuits' definiteness and stochastic-equivalent state-transition behavior, and OMC circuits' insensitivity to autocorrelation. These properties lead to several useful applications, including an optimization method MOUSE for SRB circuits, and a systematic way to configure an OMC circuit for a desired rounding policy. Finally, autocorrelation's impact on sequential circuits and its mitigation were demonstrated.

CHAPTER 5
Stochastic-Binary Hybrid Systems

Our discussions of SC designs has so far focused their stochastic aspects. However, many so-called “SC-based” systems like neural networks or vision systems are, in fact, a hybrid of SC and BC features, where BC is used, either explicitly or implicitly, for tasks requiring higher accuracy, such as data storage, control functions, or complex arithmetic operations. The resulting hybrid SC-BC designs often incorporate unsatisfactory tradeoffs between system latency and accuracy. For example, they may require many costly SBCs and BSCs that interrupt bit-stream flow and cause significant delay overhead. While improving accuracy has been a major research goal in SC, less attention has been paid to reducing latency. In particular, latency induced by SC-BC interfaces has been largely overlooked in the prior SC literature. This chapter presents a novel design methodology called Maxflow that minimizes the latency of hybrid SC-BC operations without reducing accuracy and without interrupting data flow more than necessary. Further, Maxflow supports efficient latency-accuracy tradeoffs with little hardware modification. Its effectiveness is demonstrated for an artificial neural network (NN) trained for an image classification task. The material in this chapter has been published in [73].

5.1 Integration of SC and Binary Components

As discussed in previous chapters, SC computes with probabilistic data using digital logic circuits. This means that SC is technologically compatible with circuits employing conventional binary computing (BC). In other words, while their data formats are very different, SC and BC can readily be combined into a single hybrid system. In fact,

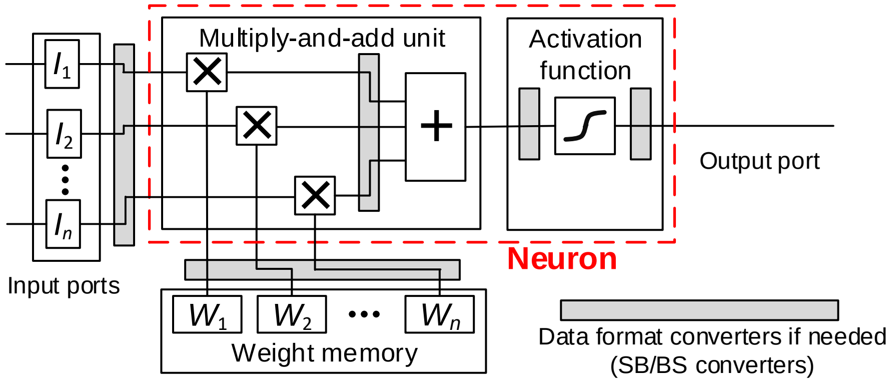


Figure 5.1 Hybrid neuron circuitry employing SC and/or BC components that are integrated via data format converters.

most so-called “SC-based” systems in the literature are actually a hybrid combination of SC and BC features, with SC parts that are only employed for specialized tasks. In particular, SC is well suited to low-precision arithmetic tasks that can benefit from SC’s low cost and error tolerance. It is less well suited to data storage, control functions, and some complex arithmetic operations that are often better implemented with BC [54]. For instance, many NN designs use BC for implementing activation functions [15][48][62] and for weight storage [17][37]; see Figure 5.1. Hybrid SC-BC systems usually need many costly SBCs and BSCs to integrate their SC and BC parts. The data format converters themselves can be seen as hybrid sequential circuits [70]. Data converters are also used extensively to restore accuracy by regenerating correlation-corrupted SNs [22]. This is done by stochastic-to-binary conversion followed by binary-to-stochastic conversion with a new RNS; see Chapter 3.

Figure 5.1 shows the block structure of a generic neuron of the kind found in most NNs [17]. The figure emphasizes the hybrid nature of such systems. Each block processes SC- and/or BC-formatted data, so data converters (shaded) must be placed at the SC-BC interfaces. For example, the NN architecture proposed by Braendler et al. [15] is mainly based on SC, but its activation functions are implemented with BC via lookup tables (LUTs) for accuracy reasons. Data converters are thus placed before and after each activation function block to allow communication with other SC units. Lee et al. propose a hybrid NN [48] that only utilizes SC in the input layer; the remaining layers use BC. Sim and Lee describe an SC-based multiplication accelerator for deep NNs, which require data converters at its inputs and outputs [62].

The impact of data converters on system latency is substantial, as Figure 5.1 suggests. For example, a typical SBC produces its BC output B_X by counting the 1s in the N-bit input SN **X**. This requires observing and accumulating all bits of **X**, a task that imposes a delay overhead of about N clock cycles. In a system with many processing stages, this kind of overhead contributes significantly to the system latency. Sometimes the BC parts of a hybrid design are implicit, so their impact on latency may be overlooked, as the following example illustrates.

Example 5.1: The usual stochastic circuit to compute $Z(X) = \sqrt{X}$ was first described in the 1960s [29] based on the ADDIE structure discussed in Chapter 4. It is a multicycle sequential design built around an up/down counter with a global feedback loop. Figure 5.2*a* shows a recent incarnation found in a power-supply monitoring system [10]. The output SN **Z** is fed back in such a way that the counter's state S converges toward $\sqrt{X}$. This is an implicit hybrid structure, since S is a binary number B_Z, and the BSC with an LFSR as its random number source is required to convert B_Z to **Z**. The overall latency and accuracy of this circuit are hard to predict as they depend on complex circuit dynamics like warmup time, convergence rate, LFSR quality, etc., but could be several times N. A significant factor contributing to this design's low accuracy and long latency is that, while it explicitly uses BC format for internal data representation, the main processing circuitry (a squarer comprising an isolator and an AND gate) uses SC. While this hybrid implementation is low-cost in hardware and power usage, its SC-based feedback loop leads to a low-accuracy signal, resulting in excessive random fluctuation that makes the up/down counter converge slowly to its steady state. □

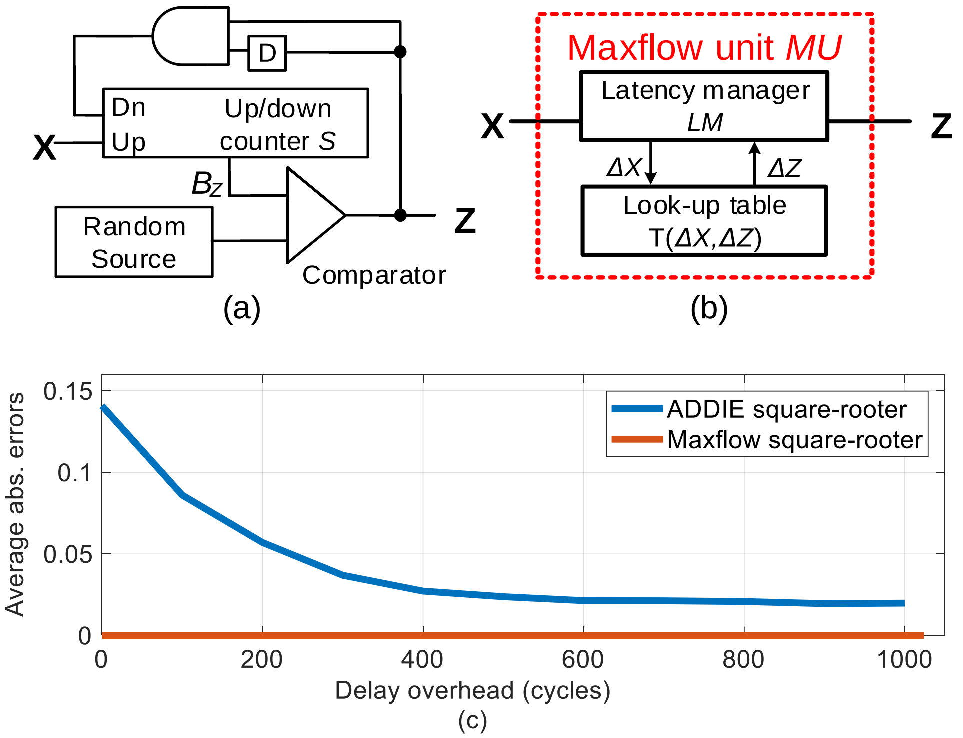


Figure 5.2 (a) Conventional square-rooter design [10] and (b) Maxflow design where **X** and **Z** are SNs. (c) Comparison of errors against delay overhead for N = 256 bits.

This chapter introduces a new design methodology, which we refer to as Maxflow, that can overcome many of the accuracy and latency limitations of previous hybrid designs. Maxflow takes SC-formatted inputs, and produces SC-formatted outputs. However, it also uses BC for internal data representation, and thus Maxflow itself is also a hybrid method. Maxflow's defining feature is that it computes accurately and at the same time minimizes the processing latency. It does so with a special preprocessing step we call *Algorithm for Minimizing Delay in Maxflow* (MIND) that can determine the delay constraint required for implementing a specific function. It then uses a special mechanism to generate output SNs in a way that guarantees accurate computation while meeting the delay constraint.

Example 5.1 (contd.): Applying Maxflow to implement a square-rooter yields a design with very low latency—just 2 clock cycles for $N = 8$—combined with high accuracy. Figure 5.2*c* compares the error rates of the ADDIE square-rooter (Figure 5.2*a*) and a

Maxflow square-rooter for N = 256 bits. The ground truth is the best possible SC approximation of $\sqrt{X}$ with N = 256. The delay overhead shown in Figure 5.2*c* is the number of cycles that must elapse before the full output SN value is measured. The conventional ADDIE square-rooter requires a long warmup time to produce a reasonably accurate output, while the Maxflow design attains an accurate result after just 64 cycles. Further, with a delay less than 64 cycles, Maxflow still does not present significant errors. This shows another advantage of Maxflow: it can trade accuracy for shorter latency. □

The rest of the chapter describes the mechanism and hardware implementation of Maxflow in detail and applies it to the design of an NN.

5.2 Latency Minimization for Complex Functions

This section provides a high-level description of Maxflow and describes in detail MIND, the preprocessing stage of Maxflow that minimizes latency overhead. Hardware implementation of Maxflow will be discussed in Section 5.3.

In a nutshell, Maxflow exploits the fact that each bit of an input bit-stream **X** provides partial information about a result **Z**, making it possible to start generating **Z** using early bits of **X**. Maxflow does this in a way that minimizes the latency of computing $Z(X)$ without compromising accuracy. This in turn tends to maximize data flow in complex all-SC or hybrid structures. Maxflow replaces a design like that of Figure 5.2*a* along with any associated data converters by the generic *MU* structure of Figure 5.2*b*. *MU* has two parts: a LUT storing a table T(ΔX,ΔZ) that indirectly defines Z, and a small controller called the latency manager *LM*. Here ΔX denotes a parameter combining the current bit of **X** and the number of 1s and 0s from **X** received so far. From ΔX, *LM* incrementally computes a lower bound on the number of 1s in **Z**, and ΔZ symbolizes a change in this lower bound. Using ΔZ, *LM* identifies and buffers bits of **Z** which it outputs at the earliest cycle possible. Roughly speaking, as *LM* receives sufficient bits from **X**, it accumulates a precise number of 1s (0s) for later release into **Z**. Note that, as mentioned earlier, Maxflow can be seen as

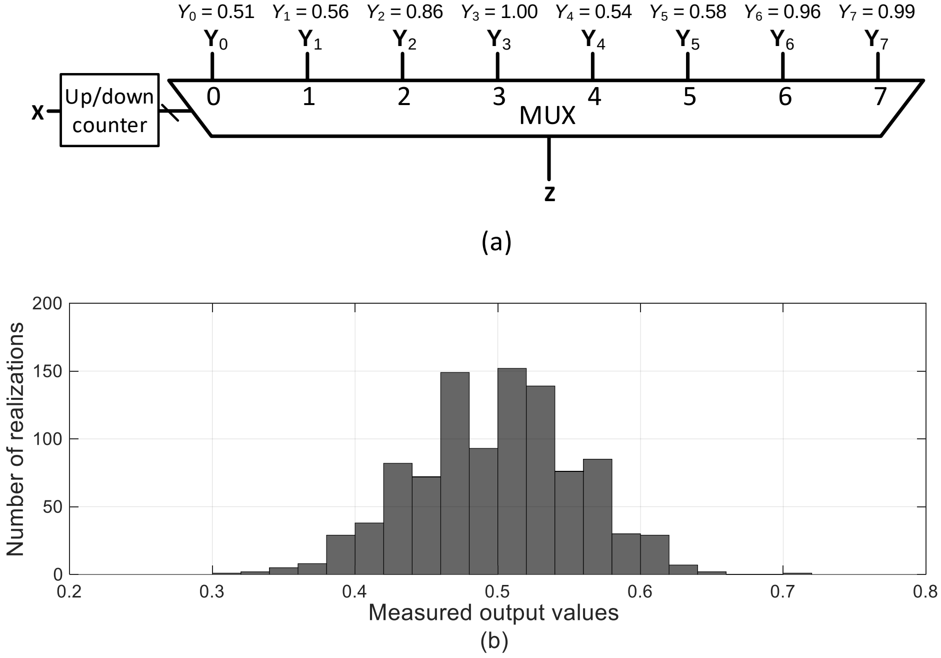


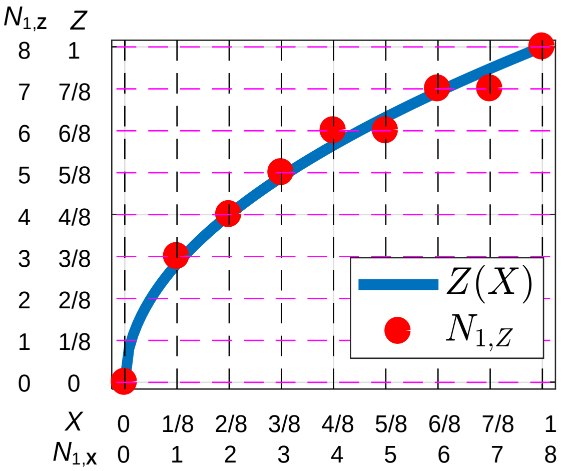
Figure 5.3 (a) Stochastic circuit C_{SIG} for $Z = sigmoid(4X)$ generated using the sequential SC synthesizer in [52] (b) Histogram of Z for 1,000 bipolar input SNs of value $X = 0.0$.

a type of hybrid design. It only stores the function definition in binary form, but it does not explicitly process **X** and **Z** in binary format. The hardware needed by Maxflow depends on $Z(X)$ and N. In Figure 5.2, the ADDIE square-rooter design contains about $2\lceil\log_2 N\rceil$ flip-flops. The Maxflow design has a similar number of flip-flops in LM, while its LUT is of size $\lceil\log_2 N\rceil k$, where $k \leq N$. As N increases, k grows at a rate of about $\log_2\sqrt{N}$. For example, with $N = 8$, $k = 2$, while with $N = 4{,}096$, $k = 6$.

Maxflow is especially suited to complex functions that are hard to implement accurately and efficiently by conventional SC approaches. Improving accuracy without using long SNs or incurring significant latency overhead has long been at the center of SC research. Prior work in this direction has focused on combinational stochastic circuits using such techniques as deterministic bit-streams [15][39][62][75], low-discrepancy codes [4], non-standard SC designs [48][72], etc. However, little is known about how to design complex stochastic circuits with guaranteed levels of accuracy, especially when they employ sequential components which have complex behavior and high inherent latency [16][70].

Figure 5.3*a* shows a UCB circuit C_{SIG} produced by the sequential synthesizer proposed by Li et al. [52] to implement the sigmoid function $sigmoid(4X) = 1/(1 + e^{-4X})$. The input **X** updates the state of the counter which indirectly determines **Z** at each clock cycle. C_{SIG}'s accuracy is affected by several factors like the bit-pattern of **X** and the warmup time required for the counter to reach steady state [70]. These factors cause **Z** to fluctuate a lot, as Figure 5.3*b* shows. This is a histogram of the output values $Z(X)$ obtained by feeding C_{SIG} 1,000 independently generated bipolar SNs, each 256 bits long and containing exactly 128 1s, hence encoding the bipolar number $X = 0.0$ exactly. The bit-patterns of the SNs, i.e., the locations of their 1s, vary widely. As Figure 5.3*b* indicates, while the output values concentrate around the expected value $sigmoid(0) = 0.5$, they are widely dispersed. This highlights the design's lack of accuracy control.

Previous SC systems with stringent accuracy constraints alleviate the preceding problem by resorting to hybrid SC-BC designs that convert SC signals to BC signals, and process them with highly accurate BC components [15][48]. We will henceforth refer to these designs collectively as "Hybrid." The major drawback of Hybrid is the large delay overhead for data conversion. The Maxflow approach, on the other hand, uses **X**'s initial bits to provide fast estimates of **Z**'s value, thereby allowing early but accurate generation of the output bit-stream. It does this in a manner reminiscent of interval arithmetic [40] by placing $N_{1,\mathbf{Z}}$, the number of 1s in **Z**, in an integer interval $I^{(i)} = [L_{1,\mathbf{Z}}^{(i)}, U_{1,\mathbf{Z}}^{(i)}]$, where $L_{1,\mathbf{Z}}^{(i)}$ and $U_{1,\mathbf{Z}}^{(i)}$ are lower and upper bounds, respectively, on $N_{1,\mathbf{Z}}$. The interval $I^{(i)}$ represents the uncertainty about **Z**'s value, and reflects the number of 1s and 0s that are known or identified in **Z** at cycle i. Specifically, the least number of 1s and 0s that **Z** must contain at cycle i are $L_{1,\mathbf{Z}}^{(i)}$ and $N - U_{1,\mathbf{Z}}^{(i)}$, respectively. A sequence of gradually shrinking intervals $I^{(0)}, I^{(1)}, \ldots, I^{(N)}$ is generated as **X** streams into MU until eventually $L_{1,\mathbf{Z}}^{(i)}$ and $U_{1,\mathbf{Z}}^{(i)}$ coincide. Maxflow's accuracy can be attributed to the fact that it only includes identified bits in the output.



X	$Z(X)$
0	0
1/8	0.35
2/8	0.50
3/8	0.61
4/8	0.71
5/8	0.79
6/8	0.87
7/8	0.94
1	1

$N_{1,\mathrm{x}}$	$N_{1,\mathrm{z}}$
0	0
1	3
2	4
3	5
4	6
5	6
6	7
7	7
8	8

(a) (b) (c)

Figure 5.4 Computation of $Z(X) = \sqrt{X}$. (a) Plot of $Z(X) = \sqrt{X}$ for $N = 8$; red dots mark $N_{1,Z}$. (b) Tabulated $Z(X)$ values. (c) N-table mapping $N_{1,X}$ to $N_{1,Z}$.

Example 5.1 (contd.): For illustration purposes, consider implementing $Z = \sqrt{X}$ with $N = 8$ using Maxflow. (In practice, $N = 2^8$ is more likely.) Here X can only have nine values of the form $N_{1,\mathbf{X}}/N$, where $N_{1,\mathbf{X}} \in \{0, 1, \ldots, 8\}$. The blue curve in Figure 5.4*a* shows $Z(X)$, and the red dots mark the values of $N_{1,\mathbf{Z}}$ for the nine possible values of $N_{1,\mathbf{X}}$. For example, when $N_{1,\mathbf{X}} = 6$, we get $Z(X) = 0.87$ and $N_{1,\mathbf{Z}} = 7$. Figure 5.4*b* tabulates $Z(X)$, while Figure 5.4*c* shows an "N-table" $\mathrm{T}(N_{1,\mathbf{X}},N_{1,\mathbf{Z}})$ that maps $N_{1,\mathbf{X}}$ to $N_{1,\mathbf{Z}}$. The values of $N_{1,\mathbf{X}}$ and $N_{1,\mathbf{Z}}$ in Figure 5.4*c* are exact in that they represent the best possible approximations to $\sqrt{X}$ given the unavoidable rounding errors of 8-bit SNs. Maxflow accurately predicts $N_{1,\mathbf{Z}}$**,** the number of 1s in **Z**, by inspecting **X** bit by bit. Before computation starts, nothing is known about X and Z, and we can only say that $N_{1,\mathbf{Z}}$ lies in the interval $I^{(0)} = [0, 8]$. Suppose the first bit of **X** is $\mathbf{X}^{(1)} = 1$. Figure 5.4*c* indicates that with $N_{1,\mathbf{X}} \geq 1$, $N_{1,\mathbf{Z}}$ must be larger or equal to 3, i.e., **Z** must contain at least three 1s, so now we can place $N_{1,\mathbf{Z}}$ in the interval $I^{(1)} = [3, 8]$. If $\mathbf{X}^{(2)}$ is also 1, the interval containing $N_{1,\mathbf{Z}}$ shrinks to $I^{(2)} = [4, 8]$. On the other hand, if $\mathbf{X}^{(2)} = 0$, then $I^{(2)} = [3, 7]$, meaning that **Z** must have at least one 0. Continuing in

Clock cycle *t*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
$\mathbf{X}^{(t)}$	0	0	0	0	1	0	0	1		
Q = Number of identified 1s/0s	0/1	0/1	0/2	0/2	3/2	3/3	3/4	4/4	4/4	4/4
S = Number of outputted 1s/0s	0/0	0/0	0/1	0/2	1/2	2/2	2/3	3/3	3/4	4/4
$Q - S$	0/1	0/1	0/1	0/0	2/0	1/1	1/1	1/1	1/0	0/0
$\mathbf{Z}^{(t)}$	$\leftarrow d_{\text{MIN}} \rightarrow$		0	0	1	1	0	1	0	1

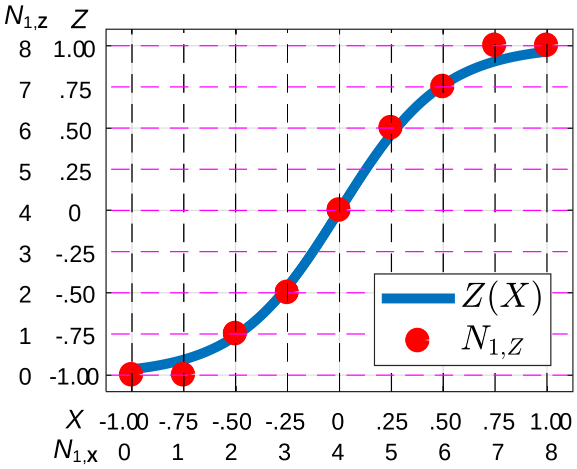
Figure 5.5 Computation of $Z(X) = \sqrt{X}$ by a Maxflow square-rooter when **X** = 00001001.

this fashion, the interval shrinks monotonically until it reaches $I^{(N)} = [N_{1,\mathbf{Z}}, N_{1,\mathbf{Z}}]$ and the exact value Z is known.

Suppose now **X** = 00001001 with time flowing from left to right. The latency manager *LM* computes the sequence of 9 intervals $I^{(0)}, I^{(1)}, \ldots\ I^{(8)}$:

$$[0,8], [0,7], [0,7], [0,6], [0,6], [3,6], [3,5], [3,4], [4,4] \tag{5.1}$$

where $I^{(i)} = [L_{1,\mathbf{Z}}{}^{(i)}, U_{1,\mathbf{Z}}{}^{(i)}]$, and the numbers of identified 1s and 0s are $L_{1,\mathbf{Z}}{}^{(i)}$ and $N - U_{1,\mathbf{Z}}{}^{(i)}$, respectively. *LM* uses the interval sequence to gradually identify bits in **Z**. Figure 5.5 tabulates the numbers of identified bits for this particular **X**. It shows that if we insert a delay of $d_{\text{MIN}} = 2$ cycles and start generating **Z** at cycle 3, enough identified output bits are always available and *LM* can output one of them. The SN **Z** in Figure 5.5 contains the expected four 1s and four 0s, and its bit-pattern reflects the randomness of **X**'s bit-pattern. Thus, the Maxflow design computes accurately with a minimal delay $d_{\text{MIN}} = 2$ for the given **X**. In general, d_{MIN} must be set to cover all possible 8-bit input SNs. An algorithm MIND to determine d_{MIN} is given below. It turns out that $d_{\text{MIN}} = 2$ works for all possible **X**'s in this example. Therefore, Maxflow reduces the latency to 2 cycles; in contrast, the minimum delay overhead of a hybrid design is at least 7 (and probably much more.) As we show later, d_{MIN} scales linearly with N for a target function. Hence, Maxflow can reduce the latency to a fixed fraction of N. □



X	$Z(X)$
–1.00	–0.96
–0.75	–0.91
–0.50	–0.76
–0.25	–0.46
0.00	0
0.25	0.46
0.50	0.76
0.75	0.91
1.00	0.96

$N_{1,\mathbf{x}}$	$N_{1,\mathbf{z}}$
0	0
1	0
2	1
3	2
4	4
5	6
6	7
7	8
8	8

(a) (b) (c)

Figure 5.6 Computation of $Z(X) = tanh(2X)$. (a) Plot of $Z(X)$ with red dots marking $N_{1,Z}$. (b) SN values for $Z(X)$, and (c) the corresponding N-table T($N_{1,X}$, $N_{1,Z}$).

	$\mathbf{X}^{(1:3)}$	$N_{1,\mathbf{x}}{}^{(1:3)}$	$I^{(3)}$	$q(3, N_{1,\mathbf{x}}{}^{(1:3)})$
Case I	000	0	[0,6]	2
Case II	001, 010, 100	1	[0,7]	1
Case III	011, 101, 110	2	[1,8]	1
Case IV	111	3	[2,8]	2

Figure 5.7 Initial 3-bit segments of **X** considered by MIND.

Now, we present the algorithm MIND that computes d_{MIN} for a given Maxflow design. The hardware implementation of Maxflow is discussed in Section 5.3. To illustrate how MIND works, consider finding d_{MIN} for a hyperbolic tangent function, a common activation function in NNs. Figure 5.6 plots $Z = tanh(2X)$ with $N = 8$, where **X** and **Z** are in bipolar format. The value of d_{MIN} must be determined such that the Maxflow unit always has enough identified bits to output, regardless of **X**'s pattern. Suppose the initial 3-bit segment $\mathbf{X}^{(1:3)}$ of **X** has been received. There are four possible cases defined by $N_{1,\mathbf{x}}{}^{(1:3)}$, the number of 1s in $\mathbf{X}^{(1:3)}$; see Figure 5.7. For example, in Case III, $\mathbf{X}^{(1:3)}$ has two 1s and the corresponding output interval is $I^{(3)} = [1,8]$, meaning that **Z** contains at least one 1. Let

```
Input:   $T(N_{1,x}, N_{1,z})$: $N$-table defining the target function $Z(X)$
         $N$: Bit-stream length
Output:  $d_{MIN}$: Minimal delay overhead
Step 1   For $i = 1, 2, \ldots, N$
             For $N_{1,x}^{(1:i)} = 1, 2, \ldots, i$:
                 Compute $I^{(i)} = [L_{1,z}^{(i)}, U_{1,z}^{(i)}]$ associated with $N_{1,x}^{(1:i)}$ via $T(N_{1,x}, N_{1,z})$
                 Set $q(i, N_{1,x}^{(1:i)}) = L_{1,z}(i) + N - U_{1,z}(i)$.
             Set $u^{(i)} = min\ q(i\ , N_{1,x}^{(1:i)})$.
Step 2   For $d = 0, 1, \ldots, N - 1$:
             If $u^{(i)} \geq i - d$,  for $i = 1 + d, 2 + d, \ldots, N$, then
                 set $d_{MIN} = d$, and output $d_{MIN}$
```

Figure 5.8 Pseudo-code of MIND to calculate the minimum latency overhead d_{MIN} for a Maxflow design.

$q(i, N_{1,x}^{(1:3)}) = L_{1,z}^{(i)} + (N - U_{1,z}^{(i)})$ denote the number of identified output bits on receiving an $\mathbf{X}^{(1:3)}$ that has $N_{1,x}^{(1:3)}$ 1s. For instance, $q(3, 2) = 1 + (8 - 8) = 1$, since $I^{(3)} = [1,8]$ when $N_{1,x}^{(1:3)} = 2$. From Figure 5.7, it is clear that regardless of $\mathbf{X}^{(1:3)}$'s bit-pattern, there is always at least $u^{(3)} = 1$ identified bit at clock cycle 3, where

$$u^{(i)} = min\ q(i\ , N_{1,x}^{(1:3)}) \tag{5.2}$$

and the *min* operation is over all possible $N_{1,x}^{(1:3)}$'s for each cycle i. Note that $u^{(i)}$ increases monotonically with i, because we can identify additional bits in **Z** as we receive more bits of **X**. Recall that Maxflow guarantees accuracy by not outputting more bits than it has identified at each cycle. That implies that $u^{(i)}$ must be greater than or equal to the number of outputted bits for all i's under consideration. This translates to the set of inequalities:

$$u^{(i)} \geq i - d, \text{ for } i = 1 + d, 2 + d, \ldots, N \tag{5.3}$$

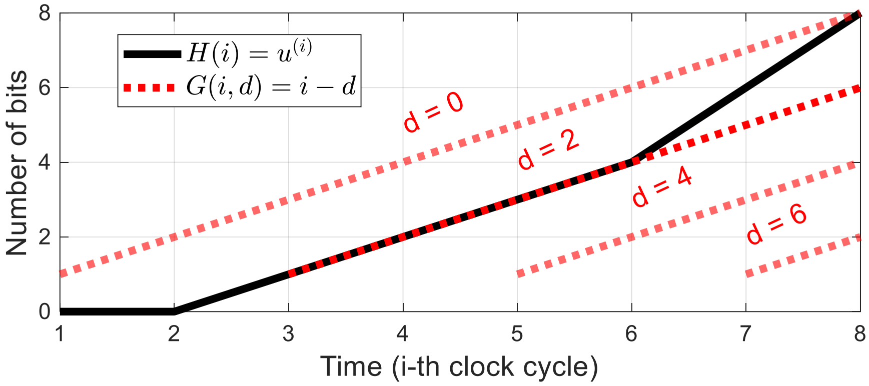


Figure 5.9 The black line plots the number of bits identified against time for $Z(X) = tanh(2X)$. The red dashed lines plot the number of outputted bits against time for various delay values d.

To gain further insight into the constraint set (5.3), we define two functions $H(i) = u^{(i)}$ and $G(i, d) = i - d$. $H(i)$ is the least number of bits that must have been identified at cycle i, while $G(i, d)$ is the number of bits that have been outputted at cycle i, which can be adjusted by controlling the delay overhead d. When satisfied, the constraint set (5.3) guarantees that never will fewer bits be identified than are outputted at any clock cycle. MIND finds d_{MIN} by scanning for the smallest feasible d that satisfies the constraint set (5.3). Figure 5.8 summarizes MIND.

Example 5.2: To find d_{MIN} for $tanh(2X)$ with $N = 8$, MIND first computes $I^{(i)}$ for $\mathbf{X}^{(1:i)}$ and $N_{1,\mathrm{x}^{(1:i)}}$ for $i = 1, 2, \ldots, 8$. For instance, with $\mathbf{X}^{(1:5)} = 01001$, $N_{1,\mathrm{x}^{(1:5)}} = 2$, and the corresponding $I^{(5)} = [1, 6]$, as can be inferred from Figure 5.6c; $q(i, N_{1,\mathrm{x}^{(1:i)}})$ can then be computed from the corresponding $I^{(i)}$. For instance, with $I^{(5)} = [L_{1,\mathrm{z}^{(5)}}, U_{1,\mathrm{z}^{(5)}}] = [1, 6]$, $q(5, 2) = L_{1,\mathrm{z}^{(5)}} + (N - U_{1,\mathrm{z}^{(5)}}) = 3$. Next, MIND calculates $u^{(i)}$ for $i = 1, 2, \ldots, 8$ via Equation (5.2), resulting in:

$$\boldsymbol{u} = [u^{(1)}\ u^{(2)}\ \ldots\ u^{(8)}] = [0\ 0\ 1\ 2\ 3\ 4\ 6\ 8]$$

To visualize this, the black line in Figure 5.9 plots $H(i) = u^{(i)}$ against time $i = 1, 2, \ldots, 8$. The red (dashed) lines plot $G(i, d) = i - d$ against time for various d's. Satisfying the constraint set (5.3) requires that the red lines not be above the black line. Finding d_{MIN} can thus be seen as finding the red line that touches, but does not lie above, the black line. This process is reflected in Step 2 of MIND (Figure 5.8). In the present example, d_{MIN} is 2, implying that a delay of 2 cycles is needed to ensure accurate computation for $Z(X) = tanh(2X)$. □

5.3 Implementation of Maxflow

Rather than storing the N-table for the target function, it is more efficient to store a Δ-table which provides the incremental change of $N_{1,\mathbf{Z}}$, and can be easily pre-computed from the corresponding N-table. In general, a Δ-table has an address consisting of $N_{0,\mathbf{X}}^{(1:i-1)}$, $N_{1,\mathbf{X}}^{(1:i-1)}$ and $\mathbf{X}(i)$. However, for monotonically increasing functions, including square-root, *tanh* and *sigmoid*, $N_{1,\mathbf{X}}^{(1:i-1)}$ and $\mathbf{X}(i)$ suffice as the address of the Δ-table, as we illustrate in this section.

$N_{1,\mathbf{X}}$	0	1	2	3	4	5	6	7	8
$N_{1,\mathbf{Z}}$	0	0	1	2	4	6	7	8	8

(a)

$N_{1,\mathbf{X}}^{(1:i-1)}$	0	1	2	3	4	5	6	7
ΔZ	0	1	1	2	2	1	1	0

(b)

Figure 5.10 (a) N-table for $Z(X) = tanh(2X)$. (b) The corresponding Δ-table with $\mathbf{X}^{(i)} = 1$.

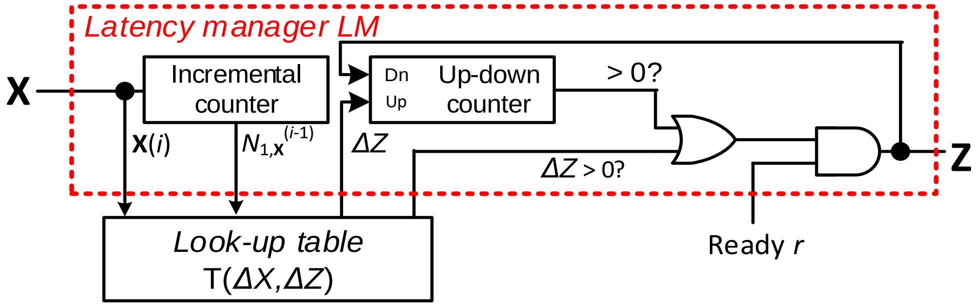


Figure 5.11 Block diagram of a Maxflow design

Figure 5.10 shows both the N-table and the Δ-table (for the part with $\mathbf{X}^{(i)} = 1$ only) for $tanh(2X)$ with $N = 8$. Recall that an N-table maps the number of 1s in $\mathbf{X}$ to the number of 1s in $\mathbf{Z}$. A Δ-table, on the other hand, stores the change in the number of 1s of $\mathbf{Z}$ on receiving a new bit from $\mathbf{X}$. For example, in Figure 5.10*a*, when $N_{1,\mathbf{X}} = 3$, $N_{1,\mathbf{Z}} = 2$. Also, when $N_{1,\mathbf{X}} = 4$, $N_{1,\mathbf{Z}} = 4$. This implies that if $N_{1,\mathbf{X}}$ changes from 3 to 4, $N_{1,\mathrm{Z}}$ will increase by 2, since it changes from 2 to 4. This is exactly the information stored in the Δ-table. Figure 5.10*b* shows the Δ-table when the current bit $\mathbf{X}^{(i)} = 1$. We omit the part for $\mathbf{X}^{(i)} = 0$, since for a monotonically increasing function, $\mathbf{X}^{(i)} = 0$ implies no change in $N_{1,\mathbf{Z}}$. Observe that when $N_{1,\mathbf{X}}^{(1:i-1)} = 3$, $\Delta Z = 2$. This is because given that $\mathbf{X}^{(i)} = 1$, and that we previously received three 1s from $\mathbf{X}$, i.e., $N_{1,\mathbf{X}} = 3$, the increment of $N_{1,\mathbf{Z}}$ is 2, as discussed previously.

Figure 5.11 shows a Maxflow unit (Figure 5.2*b*) in more detail. The Δ-table maps $\mathbf{X}^{(i)}$ and $N_{1,\mathbf{X}}^{(1:i-1)}$ to ΔZ, the increment of newly identified 1s, which is then buffered by the up/down counter. The output is set to 1, if the buffer is larger than 0 or if there are newly identified 1s ($\Delta Z > 0$). Further, if the current output is 1, the buffer will decrease by 1. Note that the buffer is initialized to the minimum value of $N_{1,\mathbf{Z}}$ for the target function, which is the least number of 1s $\mathbf{Z}$ must contain. The ready bit r is set to 1 after d_{MIN} cycles, where d_{MIN} has been computed in advance using MIND.

Comparing the Maxflow and Hybrid designs, we see that they are similar in terms of memory size. Maxflow has two counters requiring approximately $\lceil \log_2(N) \rceil$ flip flops each. Hybrid uses $\lceil \log_2(N) \rceil$ flip flops for its SBC, and another $\lceil \log_2(N) \rceil$ flip flops for an LFSR random source. Both Maxflow and Hybrid may use LUTs of a comparable size to store the target function's definition. However, the Maxflow design guarantees minimum delay, while Hybrid imposes the worst-case delay of approximately N cycles.

Figure 5.12 compares the delay overhead of Hybrid and Maxflow designs for some representative functions. Hybrid always imposes the worst-case delay overhead, since it requires about N clock cycles for data conversion, regardless of the function being

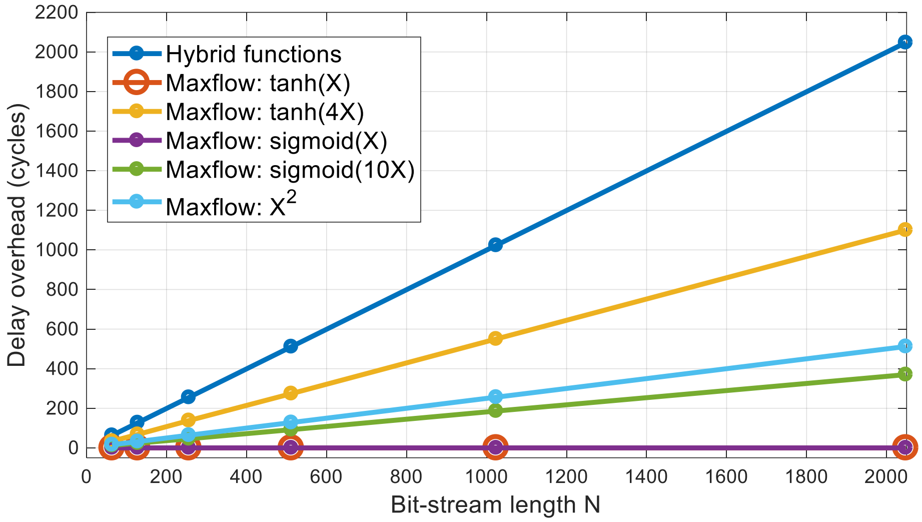


Figure 5.12 Delay overhead d_{MIN} for various Hybrid and Maxflow designs for some representative functions.

implemented. Maxflow usually has much smaller delay than the worst case. The delay overhead d_{MIN} for all cases increases linearly with respect to the bit-stream length N. Note that for functions like *tanh*(X) and *sigmoid*(X), Maxflow can always compute accurately with no added delay. This is because these functions have sufficiently steep slopes when the input value is near –1 and 1, implying that a few initial bits from the input can quickly identify many bits for the output SN. Also, the minimum value of *sigmoid*(X) is 0, implying that there must be at least half bits in the output SN are 1. This further allows Maxflow to identify some output bits even before the computation starts.

Maxflow can easily trade accuracy for shorter delay by setting the ready bit to 1 earlier. Ignoring the d_{MIN} constraint may result in insufficient identified bits of **Z**, in which case Maxflow must randomly output a 1 or a 0. In this experiment, we let Maxflow always output a 0 when there are not enough identified bits. Surprisingly, many stochastic functions seem very tolerant of aggressive delay-trading when they are implemented with Maxflow. Figure 5.13 plots the absolute errors against delay overhead for a Maxflow implementation of *sigmoid*(10X), where the results are averaged over 10,000 random inputs

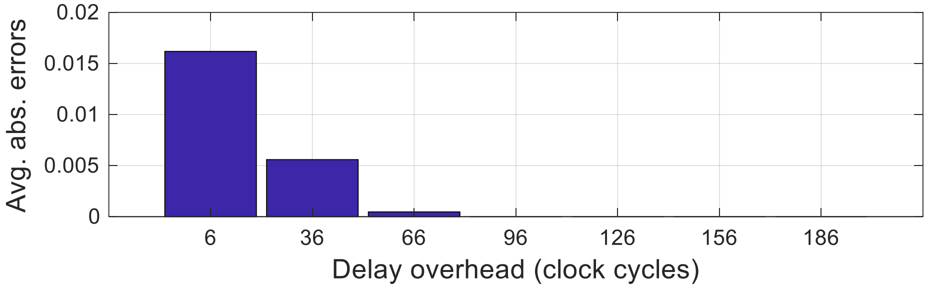


Figure 5.13 Error comparison under different delay conditions for $Z = sigmoid(10X)$ implemented by Maxflow.

of 1,024 bits. The required d_{MIN} in this case is 186 bits. However, even aggressively halving d_{MIN} does not produce significant errors. This favorable accuracy-delay tradeoff can be explained as follows. The delay constraint guarantees that the output will always be accurate, regardless of the input bit-patterns. However, our experiments show that for many functions, only a small fraction of the input patterns are "difficult" in the sense that they result in slow identification of the output bits. The probability of encountering difficult patterns is low, so even aggressively shortening the delay should not introduce significant errors.

5.4 Case Study: Artificial Neural Network

We demonstrate Maxflow's effectiveness for an artificial NN that achieves a very low misclassification rate for the well-known MNIST [46] dataset of handwritten digits. The topology of the NN is shown in Figure 5.14, and is architecturally similar to the LeNet-5 family [46][48]. It has two convolution layers, two average-pooling layers, a densely connected layer, a soft-max layer for training (omitted in Figure 5.14), and achieves a software-based test accuracy of around 99%. For hardware-based NNs, Maxflow can be extended and applied to several SC variants like those using accumulative parallel counters (APCs) [71] to improve multiply-and-add accuracy [12][50]. For clarity, we demonstrate Maxflow here for an NN employing conventional single-line SC with bipolar SNs to

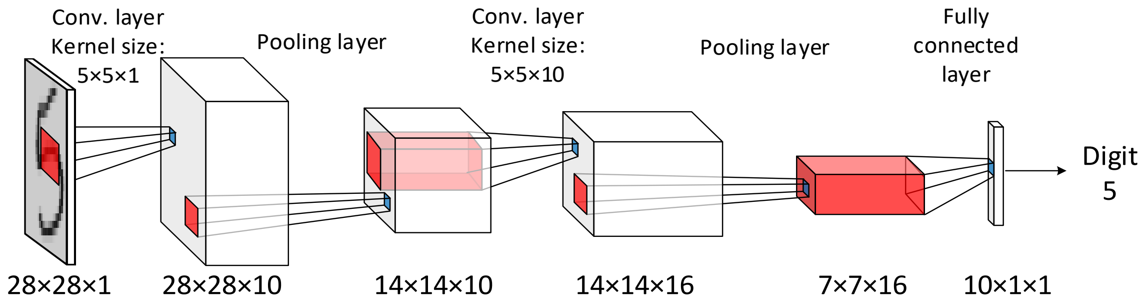


Figure 5.14 NN trained with the MNIST dataset [46].

account for negative numerical weights. Multiplication of two SNs is thus implemented by a single XNOR gate. Scaled addition is realized by a highly accurate CEASE design that was first used in the NN design reported in [48]. Recall that the CEASE adder has been demonstrated to have optimal accuracy [72]. This CEASE adder replaces the standard MUX-based adder by a sequential circuit that accurately averages all input patterns; see Chapter 2 for more details.

The synaptic weights in our NN are represented by evenly distributed low-discrepancy SNs, as used in [15][62]. We tested the NN with various choices of activation functions, including *sigmoid* and ReLU. We define ReLU(X) = $min(max(0, X), 1)$, with the value capped at 1 due to SN range limitations. All the NN designs presented in this section were trained offline on images with pixels in the range [−1 1], and simulated at the gate level to test their inference performance using C++ with GPU acceleration based on Nvidia's CUDA framework [58].

We compared the performance of three NN types with activation functions implemented by (1) Maxflow, (2) a Hybrid design with an accurate binary LUT like that in [15], and (3) a sequential SC design built around a UCB circuit [52]. Increasing the number of states for the UCB design generally improves its ability to approximate a function, but at the cost of a long warmup period during which the accuracy can be very low [70]. We chose 128-state UCB circuits to strike a balance between the number of states

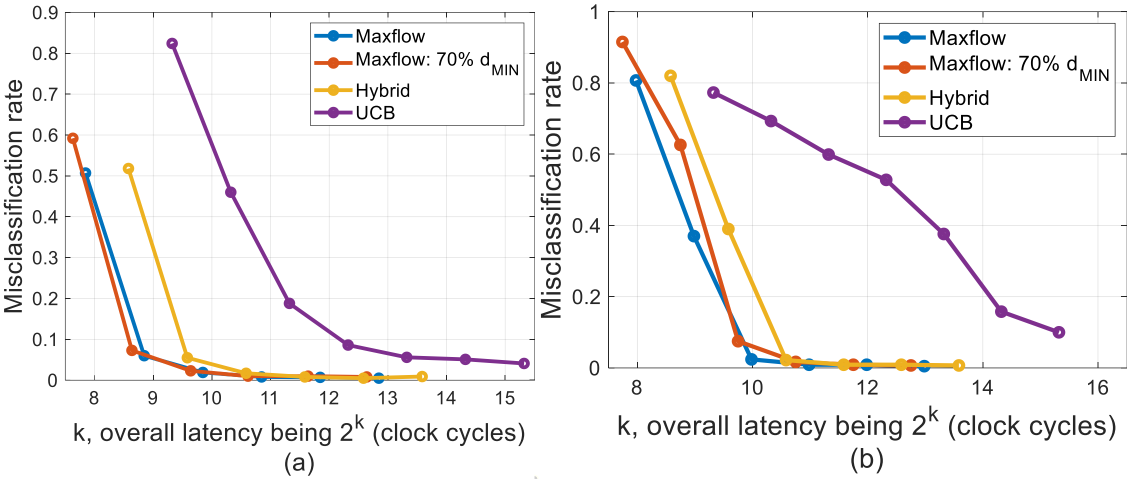


Figure 5.15 Overall latency of the neural network in Figure 5.14 with several versions of (a) *sigmoid*, and (b) ReLU as the activation function.

and accuracy. Further, to provide the randomness required by the UCB circuits, we place a shuffler before each of them for de-autocorrelation. The shuffler used has depth $D \approx N/8$, where N is the SN length. We also configured Maxflow to trade accuracy for speed by reducing d_{MIN} by 30%. Figure 5.15 plots the resulting misclassification rate against latency for the foregoing designs, where latency is defined as the cycle count from the beginning of the computation until the classification result appears. Figure 5.15*a* shows the results for *sigmoid* functions., The aggressive Maxflow design reduces the latency to about half of that of Hybrid with the same accuracy. Specifically, for a misclassification rate of about 2%, Hybrid has a latency of about 1,530 cycles, while the aggressive Maxflow design requires about 780 cycles, which is around 49% reduction in latency. The UCB NN, on the other hand, requires about 2^{13} cycles to achieve an error rate of about 5% due to its lack of accuracy guarantees. As shown in Figure 5.15*b*, ReLU functions yield similar results to those for *sigmoid*.

5.5 Summary

This chapter investigated the problem of excessively long delay in complex SC systems, noting that much of this delay is attributable to hybrid SC-BC features. The chapter then introduced Maxflow, a design methodology to minimize latency overhead in a hybrid system without compromising accuracy. Several complex functions can be implemented using Maxflow with hardware cost comparable to that of their hybrid SC-BC implementations. Experimental results for an NN trained for image classification demonstrate that Maxflow can significantly reduce the NN’s overall latency without reducing its classification accuracy.

CHAPTER 6
Exploiting Randomness in SC

Like most previous research on SC, the dissertation has so far focused on solving well-defined challenges such as long run times and insufficient accuracy attributable to SC's inherently random behavior. This chapter, on the other hand, shows that taking advantage of, or even adding to, a stochastic circuit's randomness can provide big benefits in applications like image processing and machine learning. The amount of such randomness, must however, be carefully controlled to achieve a beneficial effect without corrupting an application's functionality. The chapter describes a low-cost element to control the randomness levels of stochastic signals. It also discusses two applications where SC can provide performance-enhancing randomness at very low cost, while retaining all the other benefits of SC. Specifically, we show how to improve the visual quality of black-and-white images via stochastic dithering, a technique that leverages randomness to enhance image details. Further, we demonstrate how the randomness of an SC-based layer makes a neural network (NN) more resilient against attacks aimed at making the NN misbehave, than an NN realized entirely by conventional, non-stochastic designs.

6.1 Randomness in Stochastic Numbers

Low accuracy and randomness-induced errors have long been viewed as the main shortcomings of SC [54]. In contrast to this pessimistic view, we show next that SC's inherent randomness is actually a useful resource in some image-processing and machine-learning applications. Furthermore, we also describe a novel way to increase SC's

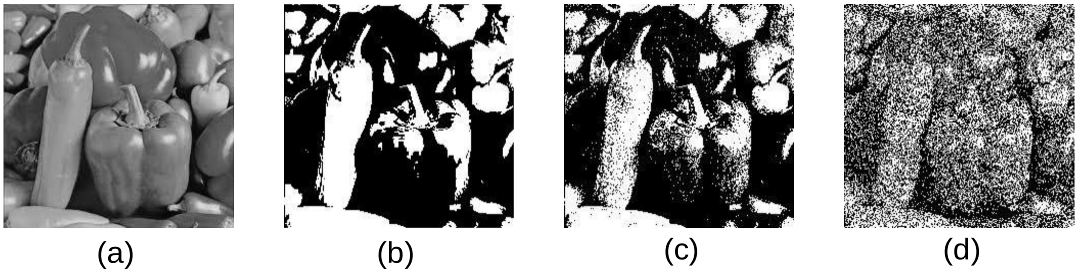


Figure 6.1 Images of peppers: (a) original grayscale image, (b) binarized image, (c) binarized image with suitable dithering, (d) binarized image with excessive dithering.

randomness levels precisely and cheaply to meet the needs of such applications, and so enhance their performance. In image processing, for instance, artificially injected randomness can preserve visual details in black-and-white images. Figure 6.1*b* shows a "binarized" image obtained by hard-thresholding the pixel intensities of the grayscale image in Figure 6.1*a*, i.e., by limiting the intensity values to black and white (0 and 1) . Figure 6.1*c* shows the same binarized image, but with randomness injected before thresholding via the well-known method of stochastic dithering [78]. Such randomness smooths the otherwise sharp boundaries between black and white regions, and effectively restores visual details.

The central idea here is that SC can be efficiently applied not only in areas that tolerate random fluctuating signals, but also in those that *need* randomness, as Figure 6.1 suggests. Such applications recognize a basic advantage of SC: it provides beneficial randomness for a very low cost, in contrast to conventional binary computing where injecting randomness requires considerable extra circuitry [42]. However, the randomness in stochastic signals is hard to control. Insufficient randomness may not satisfy a specific application's needs, while too much added randomness may produce excessive noise that corrupts SC computation unacceptably. Figure 6.1*d* shows a binarized image with too much dithering, rendering the final image excessively noisy. This implies that judicious selection of the amount of randomness used in SC designs is essential.

Next, we examine and quantify the randomness in SNs. As discussed in Chapter 1, an N-bit SN **X**'s probability value is usually estimated as $\hat{X}^{(N)} = N_1 / N$, the fraction of 1s in **X**. Here, $N_1 = \mathbf{X}^{(1)} + \mathbf{X}^{(2)} + \dots + \mathbf{X}^{(N)}$ is the sum of all bit values $\mathbf{X}^{(t)}$'s in **X**, i.e., the number of 1s in **X**. A common way to quantify the randomness level of an SN is MSE (mean squared error), which is defined by Equation (2.7). MSE measures the average error in the estimated value $\hat{X}^{(N)}$, and can also be viewed as measuring the average squared deviation of $\hat{X}^{(N)}$ from the exact value X. Since this chapter treats such deviation as a randomness parameter rather than an error, we use the term *mean squared deviation* (MSD) instead of MSE to emphasize this fact. In other words, the MSD will serve as the metric for randomness levels in this chapter. It takes the same form as MSE, that is,

$$\mathrm{MSD}(\mathbf{X}, N) = \mathbb{E}[(\hat{X}^{(N)} - X)^2] \tag{6.1}$$

In the context of SC, MSD is the variance $Var(\hat{X}^{(N)})$ of $\hat{X}^{(N)}$. Broadly speaking, an SN's MSD decreases with its length N. In other words, one can expect a more accurate computation at the cost of longer compute time. How fast MSD decreases with N depends on the SN's randomness properties which, in turn, depend on factors like the quality of the random source used to generate that SN. Consequently, providing a generic MSD convergence rate that fits all types of SNs is difficult.

While it is hard to draw firm conclusions about the MSD-N relationship for a general SN, this relation can be quantified analytically and succinctly for a Bernoulli SN, in which each bit is independently generated. Most SC designs implicitly assume that their input SNs are Bernoulli. Further, for any combinational circuit that has Bernoulli input SNs, its output will also be Bernoulli. The MSD of an N-bit Bernoulli SN **X** then takes the form [15].

$$\mathrm{MSD}(\mathbf{X}, N) = X(1 - X) / N \tag{6.2}$$

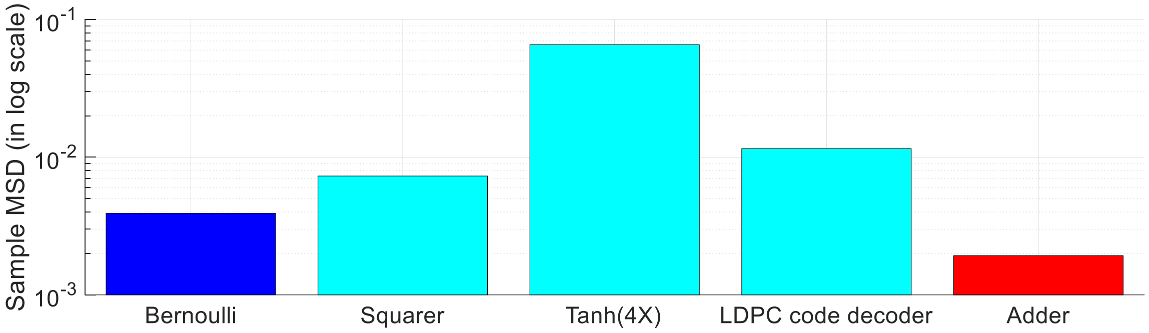


Figure 6.2 MSD comparison for 0.5-valued 64-bit SNs produced by various representative sequential stochastic circuits.

which shows that the MSD or the variance of a Bernoulli SN **X** decreases with bit-stream length at the rate of $1/N$. Further, **X**'s MSD also depends on **X**'s value X. Specifically, the MSD is a maximum when $X = 0.5$, while it is zero when X is 0.0 or 1.0.

In contrast to Bernoulli SNs, the bits of a non-Bernoulli SN are not fully independent and so are correlated with each other in some way; this is referred to as autocorrelation. As discussed in Chapter 4, autocorrelation is unavoidable in sequential SC designs [16][70], which introduce dependency among bits via their built-in memory. Quantifying the MSD-N relationship for autocorrelated SNs is considerably more challenging than the Bernoulli case, since autocorrelation can take many different and often subtle forms [70]. Further, the correlation or the covariance among bits can contribute to the MSD level of an autocorrelated SN, leading to an MSD that can be greater than, equal to, or less than, that of a Bernoulli SN with the same length and the same probability value.

Example 6.1: Randomness Levels. Figure 6.2 compares the MSDs of some representative single-output sequential circuits. These MSDs are obtained by averaging the squared deviations of each circuit's output over 10,000 simulated trials. For a fair comparison, all the circuits use 64-bit SNs, and are made to produce an output SN of value 0.5. The squarer is that of Figure 1.4*b* with input value $X = 1/\sqrt{2}$, so the expected output value is $Z = X^2 = 0.5$. Tanh(4X) computes a hyperbolic tangent function used in the neural network of [16]. The LDPC code decoder is a two-input sequential design for the update node [33]. Its two inputs are fed with SNs of value 0.5, so the output is also 0.5. Finally, the adder is an OMC design developed for the NN implementation in [48]. Here, the adder averages two 0.5-valued SNs, so the output value is also 0.5. Figure 6.2 shows the (sample) output MSDs for all four designs. While these SNs all have a value of 0.5, their MSD levels are quite different. This reflects the fact that these designs have widely varying structures, leading to output bit-streams with very different autocorrelation patterns. □

Example **6.1** reveals some interesting facts about randomness in sequential stochastic circuits. All the designs except the adder produce an output SN with a larger MSD than the Bernoulli SN. Autocorrelation injected by sequential elements tends to reduce the amount of information carried by the output SN, as each bit in the SN provides overlapping, and hence dependent, information about that SN's value. This usually leads to a relatively high MSD. The exception here is the (scaled) adder, which takes two independent SNs with the same value 0.5 and produces a single output SN which also has a value of 0.5. The adder's low MSD can be attributed to the fact that it is designed to utilize information from both inputs. The output SN thus achieves a lower MSD than the Bernoulli SN. Note, however, that more inputs do not always lead to less randomness. The LDPC code decoder is also a two-input circuit, but its output has a much higher MSD than the Bernoulli case.

Example **6.1** also suggests that sequential elements can change an SN's MSD level, but that the amount of randomness inserted or removed by sequential elements is hard to predict. Next, we will describe a novel method to solve the preceding problem.

6.2 Controlling Randomness in SC

Prior research has considered SC's inherent randomness and randomness-induced errors as a price paid to achieve very low area and power. Consequently, much effort has been put into reducing SC-style randomness. This typically involves either eliminating some stochasticity or else injecting more determinism into SC designs. For instance,

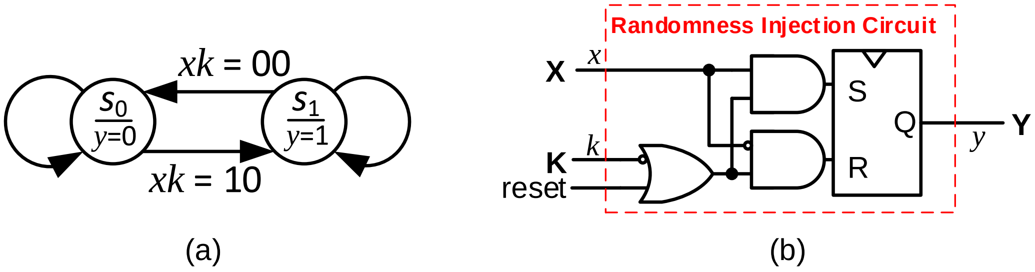


Figure 6.3 (a) STG and (b) logic design of the proposed randomness injection circuit (RIC). The SN **K** controls the randomness level of **Y**, while the reset line allows the first bit of **X** to initialize the RIC's state.

CEASE (see Chapter 2) can reduce random fluctuation errors by eliminating unnecessary constant SNs. Also, conventional unipolar or bipolar SN formats can be replaced by various deterministic formats, such as low-discrepancy sequences [4][39].These formats have been shown to significantly improve accuracy and energy efficiency in applications like deep neural networks (DNNs) [27][48]. Besides resorting to deterministic formats, most previous work tries to control SC randomness indirectly by adjusting SN length N. Its goal, is either to improve accuracy and reduce MSD by increasing N, or else to reduce run-time by reducing N, usually at the cost of higher MSD. To halve the MDS level, N has to be doubled, as implied by Equation (6.2). For those applications that need dynamic accuracy or MSD levels, considerable design effort or hardware overhead is required. Moreover, this length-adjustment method excludes fine-tuning of MSD levels, as most SC designs only work well with bit-streams of length $N = 2^i$, where i is a positive integer. In other words, the set of available MSD levels is limited to those attainable by an SN length of $N = 2^i$ for some i. On the other hand, this chapter's main goal is to increase MDS levels precisely for applications that can exploit extra randomness.

We now present a novel way to precisely increase MSD levels for SC. It employs a *randomness injection circuit* (RIC), which is a small circuit built around an SR flip-flop that adds randomness to an SN **X** without the need to modify **X**'s length; see Figure 6.3. A

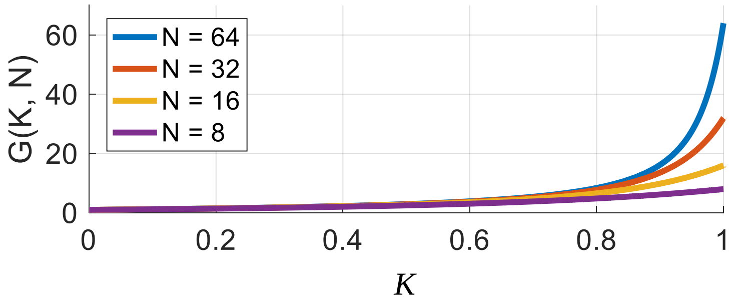


Figure 6.4 Plot of the randomness increment $G(K, N)$ against K for various SN lengths.

RIC works by transforming **X** to another SN **Y** with a higher MSD, which is determined by a user-supplied SN **K**. Intuitively, the RIC uses **K**'s value K to specify a portion of **X** to be duplicated in **Y**. If **Y** copies a large portion of **X**, then **Y** will have a similar MSD level to **X**. On the other hand, if **Y** preserves very little information from **X**, then **Y** will have a much higher MSD than **X**. Loosely speaking, the larger K becomes, the more randomness is added to **Y**. The following theorem formally quantifies the above discussion.

Theorem 6.1. Let $Var(\hat{X}^{(N)}) = X(1 - X) / N$ be the variance or MSD for an N-bit Bernoulli SN **X**. A RIC converts **X** to another N-bit SN **Y** with $Y = X$ and MSD level

$$Var(\hat{Y}^{(N)}) = [1 + \frac{2K[(1-K)N+K^N-1]}{(1-K)^2N}]Var(\hat{X}^{(N)}) \tag{6.3}$$

where $K \in [0,1]$ is the value of a user-supplied SN **K** that controls **Y**'s MSD levels.

Theorem **6.1** is proven in Appendix A.5. Here, we give an intuitive explanation of Equation (6.3). Let $G(K, N) = 1 + \frac{2K[(1-K)N+K^N-1]}{(1-K)^2N}$ denote the randomness increment produced by the RIC, which is always positive for $K \in [0, 1]$. Obviously, $G(K, N)$ depends on both K and N. Figure 6.4 plots $G(K, N)$ against K for different values of N. We see that if $K \to 0$, then $G(K, N) \to 1$, and **Y**'s MSD $Var(\hat{Y}^{(N)}) \to Var(\hat{X}^{(N)})$. In other words, **Y**'s MSD approaches the Bernoulli input **X**'s MSD. Also, as $K \to 1$, $G(K, N) \to N$, and $Var(\hat{Y}^{(N)}) \to N \cdot Var(\hat{X}^{(N)}) = X(1 - X) = Var(\hat{X}^{(1)})$, which is the largest possible MSD level attained by truncating **X** to a single bit. For $K \in (0, 1)$, $Var(\hat{Y}^{(N)})$ will take a value in the range

[$Var(\hat{X}^{(N)})$, $Var(\hat{X}^{(1)})$]. In other words, by simply adjusting K, the MSD level of **Y** can be set to any value in [$Var(\hat{X}^{(N)})$, $Var(\hat{X}^{(1)})$]. This range covers all possible MSD levels achievable by adjusting or truncating **X**'s length to fewer than N bits. Further, as the next example illustrates, a RIC can also achieve precise MSD levels that are not attainable simply by adjusting **X**'s length.

Example 6.2: Consider a Bernoulli SN **X** whose length is $N = 32$ bits. Let $Var(\hat{X}^{(32)})$ denote **X**'s current MSD level. Suppose we want to increase **X**'s MSD level by a factor of 5 to $5Var(\hat{X}^{(32)})$. Using the conventional method of length adjustment, **X** must be truncated to 32/5 = 6.4 bits, which is obviously not possible, as SN length must be a positive integer. However, according to Theorem **6.1**, RIC can easily achieve this MSD level by setting $G(K, 32)$ to 5. The equation $G(K, 32) = 5$ can be easily solved by scanning through K's values in the range [0, 1]. This leads to $K = 0.69$. In other words, a RIC design with $K = 0.69$ increases a 32-bit Bernoulli SN's MSD by a factor of 5. Furthermore, RIC can work simultaneously with SN length adjustment. Consider reducing **X**'s bit-stream length to 8 bits, and at the same time using a RIC to achieve the same MSD level of $5Var(\hat{X}^{(32)})$. First note that $5Var(\hat{X}^{(32)}) = 1.25Var(\hat{X}^{(8)})$; see Equation (6.2). Then solving $G(K, 8)= 1.25$, gives $K = 0.1273$. In other words, a RIC with $K = 0.1273$ and with output length truncated to 8 bits also achieves the MSD level $5Var(\hat{X}^{(32)})$. □

Example **6.2** shows that a RIC can add randomness at a granularity finer than SN length adjustment. Further, the RIC is quite flexible in that, with no hardware modification, it can work simultaneously with SN length adjustment, or with SC systems that have dynamically changing SN lengths. This is because the RIC's STG does not depend on N. Next, we will investigate two applications, namely image dithering and neural-network hardening, whose performance can be enhanced by RIC-controlled SC randomness.

6.3 Case Study: Image Dithering

Binarization in image processing converts a (grayscale) image to a black-and-white

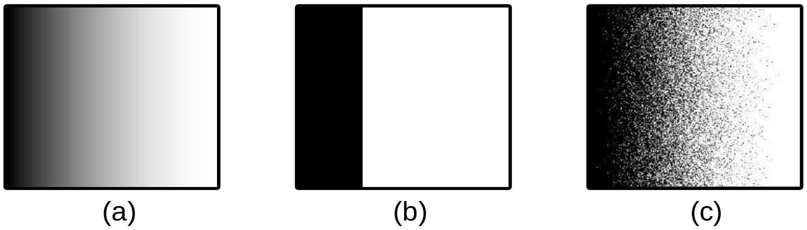


Figure 6.5 Images with gradient intensities: (a) original grayscale image, (b) binarized image, and (c) binarized image with stochastic dithering.

one [64]. It is intended to reduce the image's representational cost at the price of information loss, since pixel intensities in a binarized image are reduced to a single bit. Thresholding is a common way to binarize images: pixels above and below an intensity threshold *thr* are set to 1 and 0, respectively. However, thresholding does not distinguish intensities that are very close to *thr* from those that are far away from *thr*. Figure 6.5*a* depicts a grayscale image in which the pixel intensities gradually change in the horizontal direction. Figure 6.5*b* shows the same image binarized by hard-thresholding. This image draws a sharp vertical boundary between black and white regions; within each region the pixel intensities are indistinguishable, and visual details are lost.

Stochastic dithering restores some of an image's visual details by injecting random perturbations, which convert grayscale to densities of white and black pixels that can in turn convey intensity information. Figure 6.5*c* is obtained by binarizing Figure 6.5*a* with stochastic dithering. Compared to Figure 6.5*b*, Figure 6.5*c* obviously provides more visual detail, including a sense of gradient. The key to this intensity-density conversion is to add randomness that allows some near-threshold intensity values to cross the threshold. The probability of a successful crossing depends on the original pixel intensity. For example, assume that $thr = 0.5$. Let I_1 and I_2 denote the regions whose intensities are 0.49 and 0.05, respectively. Without dithering, pixels in I_1 or I_2 will all be thresholded to 0. On the other hand, with random perturbations, pixels in I_1 are very likely to exceed *thr*, while those in I_2 are not. This is because pixels in I_1 have an intensity that is very close to *thr*, so a small

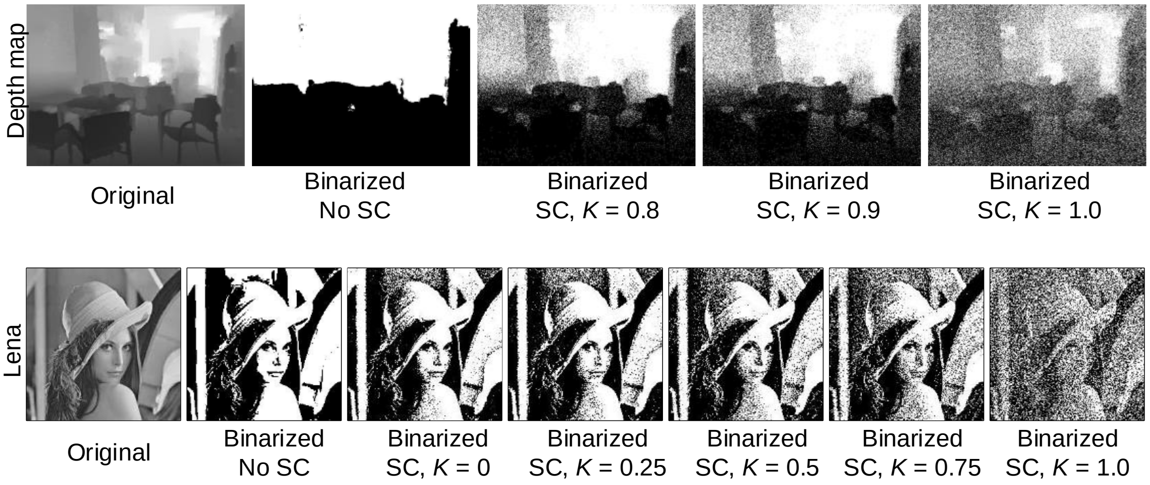


Figure 6.6 Images of a depth map [61] and Lena with stochastic dithering at different randomness levels enabled by a RIC.

perturbation can easily cause them to cross the threshold. After binarization, pixels in I_1 are more likely to be set to 1 than pixels in I_2, leading to I_1's higher white-pixel density.

SC has been applied to image-processing tasks that require low power and low precision, such as a retinal implant chip [8]. It is especially useful for near-sensor preprocessing like edge detection, where analog images are continuously sensed by light sensors, since analog data can be easily converted to SNs. As noted already, SC provides automatic stochastic dithering for SC-based image-processing tasks. Here, we focus on using a RIC to precisely control the level of dither perturbations. We compare the visual quality of binarized images processed by conventional binary computing (BC) and by SC with N = 128 bits. Figure 6.6 shows images of a depth map and Lena in grayscale and binarized forms. The depth map, which encodes objects' physical distance as intensities, is from the NYU depth dataset [61] obtained using depth sensors in Microsoft Kinect. For those images processed by SC, RICs are used to control randomness levels for stochastic dithering before binarization. As the figure shows, the images with dithering present more visual details than hard-thresholded images. Further, as expected, for RICs using a larger K, the added randomness increases. When K = 1.0, SNs carry the largest possible MSD, which is equivalent to that of an SN truncated to a single bit.

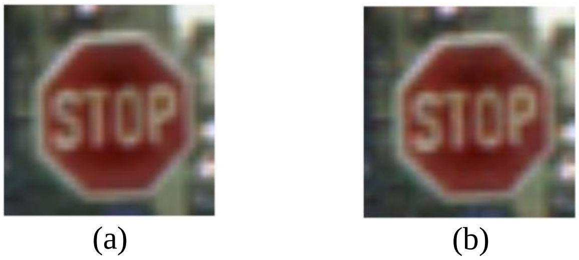


Figure 6.7 (a) Image of a stop sign; (b) the same image modified by an attack that causes it to be misclassified as a yield sign by a DNN [60]

The major advantage of RICs is that they allow a user to control MSD levels. This is important because different images may require different MSD levels for the best dithering results, as the depth map and Lena images in Figure 6.6 suggest. Also, image quality is a somewhat subjective concept, so different users may have different preferences for MSD levels, which is also enabled by RICs.

6.4 Case Study: Neural-Network Hardening

A major emerging application that can also benefit from SC's randomness is hardening deep neural networks (DNNs) against malicious attacks. DNNs have achieved unprecedented success with human-level or better performance in applications such as image classification. Despite their dominance in the field of artificial intelligence (AI), research has shown recently that DNNs are highly vulnerable to adversarial attacks (AAs) [18][20], raising severe security concerns in areas like autonomous vehicle control and face recognition systems. In image classification, an AA employs artificial images that are real images disguised in a way that is imperceptible to humans, but causes a DNN to misclassify them. Figure 6.7*a* shows an image of a stop sign, while Figure 6.7*b* shows the same image after being deliberately altered to make a DNN classify it as a yield sign [60]; this is obviously dangerous for a DNN-controlled autonomous vehicle. Such deception also exposes applications using cloud-based ML services like Google Cloud Vision and Amazon Rekognition to security threats. To defend against AAs, implementation details of DNNs employed in safety-critical applications are usually deliberately concealed, as it is considered significantly harder to attack such DNNs. However, recent work [20] suggests that even in a "black-box" setting, where the target DNN's details (e.g., the number of layers, the weight values, whether hardware-based or not, etc.) are unknown, AAs can still be easily generated by sending test inputs (queries) to the DNN that leverage the corresponding output responses. To date, there appears to be no general way to make a DNN completely immune to AAs [13].

We now show that adding certain SC features to black-box DNNs makes them much less vulnerable to attack, in addition to providing other SC benefits such as low power and small area. The goal of (untargeted) AAs is to generate an artificial image $\mathbf{x}$ that looks like a normal image $\mathbf{x}_0$, but is wrongly classified by the target classifier $Net(\mathbf{x})$. Thus, AA generation may be formulated as the optimization problem [18]

$$\text{Minimize}_{\mathbf{x}\in[0,1]^d}\ f(\mathbf{x}) = \text{Dist}(\mathbf{x}, \mathbf{x}_0) + \lambda \cdot \text{Loss}(\mathbf{x}, Net(\mathbf{x}), t) \tag{6.4}$$

where f is the objective function, which depends on $\mathbf{x}$, $\mathbf{x}_0$, Net and t. When clear from the context, we omit such dependencies, and simply write $f(\mathbf{x}, \mathbf{x}_0, Net, t)$ as $f(\mathbf{x})$. $\text{Dist}(\mathbf{x}, \mathbf{x}_0)$ is a distance metric that measures the visual dissimilarity between the attack image $\mathbf{x}$ and the original image $\mathbf{x}_0$. The attacker aims to make $\mathbf{x}$ appear like a normal image $\mathbf{x}_0$. Here, we use squared Euclidean distance, so $\text{Dist}(\mathbf{x}, \mathbf{x}_0) = |\mathbf{x} - \mathbf{x}_0|^2$. The term $\text{Loss}(\mathbf{x}, Net(\mathbf{x}), t)$ in Equation (6.4) forces the attack image $\mathbf{x}$ to be wrongly classified, and is defined as

$$\text{Loss}(\mathbf{x}, Net(\mathbf{x}), t) = \max[Net(\mathbf{x})_t - \max_{i\neq t}[Net(\mathbf{x})_i] + \kappa, 0] \tag{6.5}$$

where $Net(\mathbf{x})_t$ is $\mathbf{x}$'s score for class t, while $\max_{i\neq t}[Net(\mathbf{x})_i]$ is $\mathbf{x}$'s largest score for all classes other than t. κ is a non-negative parameter defining a required confidence level for attack success. When $\kappa = 0$, the loss function in Equation (6.5) will incur a non-zero penalty, unless the attack image $\mathbf{x}$'s largest score is not the correct class t, i.e., it incurs a penalty unless $Net(\mathbf{x})_t \leq \max_{i\neq t}[Net(\mathbf{x})_i]$. When $\kappa > 0$, Equation (6.5) will introduce a penalty, unless $\mathbf{x}$'s largest score is larger than that of t by at least κ. In other words, the attack confidence κ controls the amount of score value by which the mispredicted label should be higher than the correct label. Finally, the parameter λ in Equation (6.4) balances the attack's success and its visual similarity to the normal image. Summarizing, Equation (6.4) can be understood as aiming to generate an attack image **x** that is visually similar to a normal image $\mathbf{x}_0$, but is classified by the targeted DNN differently from $\mathbf{x}_0$.

In a black-box setting, Equation (6.4) cannot be solved by simply using the standard gradient-descent-based method, as the analytical form of the term *Net*(**x**) is concealed. This renders the true gradients of *f*(*x*) unobtainable, which gives a false sense of security that black-box DNNs are safe against AAs. It is known [20], however, that black-box NNs can still be attacked by approximating *f*(*x*)'s true gradients as

$$g_i := \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i} \approx \frac{f(\mathbf{x}+\epsilon\mathbf{e}_i)-f(\mathbf{x}-\epsilon\mathbf{e}_i)}{2\epsilon} \tag{6.6}$$

where $\mathbf{x}_i$ is the *i*-th pixel of **x**, $\mathbf{e}_i$ is a one-hot coded vector with the only 1 appearing in its *i*-th element, and ϵ is a small constant. The significance of Equation (6.6) is that it approximates *f*(**x**)'s true gradient with respect to $\mathbf{x}_i$ without knowing the target DNN's details. It simply sends test images with pixel $\mathbf{x}_i$ slightly perturbed, observes the objective's responses $f(\mathbf{x} \pm \epsilon\mathbf{e}_i)$, and then approximates *f*(**x**)'s gradients, which can be subsequently used to optimize Equation (6.4) for AA generation. This is called a *zeroth-order method*, which generates AAs by optimizing Equation (6.4) with approximate gradients obtained by querying the target DNN [20].

Equation (6.6) relies on accurate input-output responses for reasonably good gradient approximation. Randomization can thus harden a DNN by blurring the DNN's input-output responses [53]. This leads to corrupted gradient approximation, making zeroth-order attacks costlier as more DNN queries are needed to average out the blurring effect. Next, we show that DNNs with an SC layer automatically have such a defensive randomization effect.

To assess the preceding idea, we applied a black-box AA generation method called zeroth-order optimization attack (ZOO) [20] to a DNN whose last fully-connected layer is replaced with an SC implementation. Specifically, we used a VGG-19 DNN [63] trained on the CIFAR10 dataset [44], which we simulated using the Tensorflow machine learning framework [1] with a customized SC fully-connected layer, which we denote as VGG-19-SC. Note that adding SC randomness can degrade classification accuracy. The key to minimizing this degradation is to inject suitable randomness when training the DNN, so it learns how to operate under a rather noisy environment, as suggested in [53]. We did so by adding a suitable amount of Gaussian perturbations to the fully-connected layer's inputs during training. We also constrained all numerical values of the last layer to the range [−1, 1] to simulate bipolar SC operations. The resulting testing accuracy for the trained VGG-19 network is around 86%. Finally, we note that while SC can be used to implement other layers of a DNN as well, we found experimentally that using SC for the last layer affects classification accuracy the least. Intuitively, perturbations introduced in early layers tend to be magnified undesirably when they propagate through the DNN.

For the VGG-19-SC DNNs, we built RICs into the SC layer to control the amount of injected randomness. Specifically, the SN length used in the SC layer was 32 bits, and RICs were inserted on the output lines of the SC layer for precise MSD-level incrementing. We assessed the performance of the VGG-19-SC DNNs by measuring the attack success rate given a query count budget. We did so by constraining the maximum number of gradient steps to be 5,000, and each step was processed in a batch of 32 gradient estimates. The balance parameter λ in Equation (6.4) was initialized to 1, and was dynamically adjusted at run-time, as in [20]. We defined an attack to be successful if, within the query count budget, it is wrongly classified with confidence $\kappa = 0.2$, and if its squared Euclidean distortion is less than 10. For each DNN setting, 100 attacks were attempted.

Figure 6.8 shows both the classification accuracy and the attack success rates for the various DNNs. Despite the added SC randomness, the classification accuracy does not suffer from noticeable degradation, as these DNNs were trained to operate with randomness present. The attack success rates for VGG-19-SCs decrease with *K*, a RIC

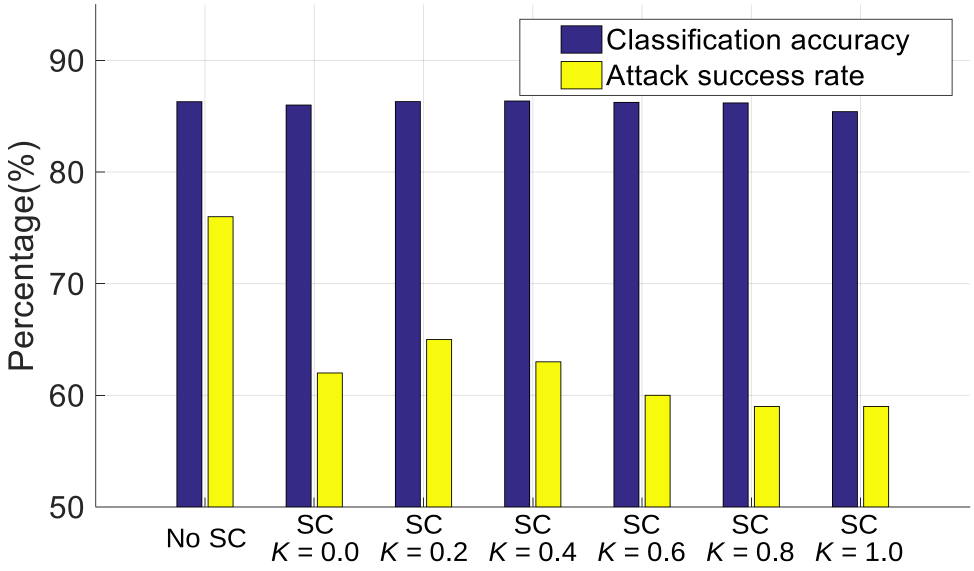


Figure 6.8 (a) Classification accuracy and attack success rates for VGG-19 and VGG-19-SC with various randomness levels.

parameter that enables balancing between classification accuracy and attack success rates. In this case, *K* = 1 appears to be a good choice, because it reduces VGG-19-SC's classification accuracy only by about 1%, compared to that of VGG-19. However, the success rate of attacks on VGG-19-SC decreases from 76% down to 59%. For those DNNs trained without noise present, *K* must be carefully chosen so that the classification accuracy is not compromised. Note that while SC has been successfully applied to NNs to reduce power and area costs previously, here we have shown for the first time that NNs with SC features automatically gain an extra benefit—they are costlier to attack.

Finally, it is noteworthy that, even an attack has been successfully generated by ZOO, applying this same attack to a VGG-19-SC DNN's may not always make the DNN misbehave. This is because the DNN's SC layer automatically perturbs the attack images randomly. We observed that such perturbations can often corrupt, and consequently void, low-confidence attack attempts—another side benefit enabled by SC's randomness.

6.5 Summary

This chapter has demonstrated that SC's randomness is a potential design resource that can greatly enhance the performance of certain tasks in image processing and machine learning. We first observed that autocorrelation introduced by sequential elements can change the randomness levels in stochastic numbers. We then devised a novel sequential circuit called a RIC that can introduce precise levels of randomness into stochastic circuits. The value of this technique was shown for two applications. Specifically, SC randomness automatically enables stochastic dithering that preserves visual details in binarized images by converting pixel intensities to densities of bright pixels. It also increases the resilience of black-box DNNs against adversarial attacks, a type of malicious data that are used to deceive a machine-learning model.

CHAPTER 7
Conclusions

This dissertation addressed several central problems encountered in SC, focusing on randomness management, error elimination, and design methodology for sequential circuits. This chapter first summarizes the major contributions of our research. It then provides pointers to promising future research areas in SC.

7.1 Summary of Contributions

Our early research on implementing SC-based matrix operations [71] revealed several SC's major challenges, including complex accuracy-latency trade-offs, lack of general design methodologies and poorly understood randomness behavior. We therefore decided to concentrate our doctoral research on some of these problems.

Chapter 2 began with an in-depth investigation of random fluctuation errors (RFEs) in SC. RFEs can be solved by increasing bit-stream length. This, however, entails significant latency overhead. We observed that many SC designs use constant SNs as ancillary signals and discovered that they are, in fact, a major contributor to RFEs. We then devised an algorithm CEASE (Constant Elimination Algorithm for Suppression of Errors) that can transfer the role of constants to memory elements, and at the same time eliminate the RFEs induced by these constants. A CEASE-designed circuit uses its memory elements to maintain an accurate count of input values, which are later converted to bit-streams for release to the output. Further, we were able to show that CEASE designs achieve the lowest RFE level among all other designs that implement the same function. The effectiveness of

CEASE was demonstrated for several representative stochastic circuits, which displayed considerable accuracy improvement without incurring any latency overhead.

Chapter 3 addressed correlation, another major error source in SC. Correlation errors occur when SNs with insufficient randomness interact with each other undesirably. They must usually be eliminated by a process called decorrelation. Our research focused on a specific decorrelation method called isolation, which works by inserting delay elements termed isolators to misalign the interacting correlated SNs. While isolation has far less hardware overhead than other decorrelation methods, it has only been used in the past in an ad hoc fashion, resulting in non-optimal isolator numbers or unexpected functional corruption. We thus developed the first systematic isolation-decorrelation algorithm VAIL (Valid Isolator Placement Algorithm Based on Integer Linear Programming), which guarantees a correct decorrelation using a minimum isolator number. VAIL does so by first forming a set of constraints that must be met for correct decorrelation. It then minimizes the use of isolators while meeting these constraints. Our experimental results on representative circuits showed that VAIL can correctly remove correlation using much fewer isolators than a naïve or ad hoc isolation method.

Noting that both CEASE and VAIL produce sequential designs, we went on to investigate the behavior of sequential elements in SC. Although sequential components have long been known to play an important role in SC, their theory has been poorly understood. This has limited sequential SC designs to a very few types, most of which are built around up/down counters. Chapter 4 identified two new classes of sequential stochastic circuits, namely optimal-modulo-counting (OMC) circuits and shift-register-based (SRB) circuits, both having some special and desirable properties. OMC circuits include those generated by our CEASE algorithm. They are insensitive to input autocorrelation, meaning that autocorrelation in the input SNs does not affect the accuracy of OMC circuits. We showed that an OMC circuit's rounding policy can be easily managed by adjusting its initial states. SRB circuits are built around feed-forward structures resembling shift registers and can reach a steady state within a fixed amount of time. For

SRB circuits, we also defined stochastically equivalent state transitions, whose associated output values can be switched without changing the circuits' stochastic behavior. This led to a sequential optimization algorithm MOUSE (Monte-Carlo Optimization Using Stochastic Equivalence), which reduces a circuit's hardware cost by switching stochastically equivalent output values using a Monte-Carlo method.

Chapter 5 is concerned with design aspects of SC-BC hybrid systems. We observed that many so-called "SC-based" designs in the literature resort to BC components for arithmetic operations that require high accuracy. Connecting SC and BC parts, usually however, poses significant latency overhead. To address this problem, we devised a new hybrid architecture called Maxflow. It has SC-based input-output interfaces, but its internal structure uses BC. Nevertheless, Maxflow computes accurately without introducing more delay than necessary, and hence tends to maximize signal flow. Maxflow's high accuracy can be attributed to the fact that it only includes in its output SN bits that can be correctly inferred from the input bits received so far. The effectiveness of Maxflow was demonstrated for a neural network trained to classify handwritten digits. Experimental results showed that, using Maxflow to implement the NN's activation functions can significantly reduce the NN's latency without sacrificing its classification accuracy.

Finally, Chapter 6 discussed the potential of using SC's intrinsic randomness as a resource instead of treating it as an undesirable error source. It first noted that a significant amount of previous research effort has been devoted to reducing randomness with the goal of improving SC accuracy, usually at the cost of considerable hardware overhead. In contrast, we showed that SC can provide beneficial randomness to certain applications at very low cost. However, the amount of randomness must be carefully controlled so that it does not corrupt the functionality of the applications. To address this problem, we proposed a tiny new element called a RIC (Randomness Injection Circuit) that can increase a stochastic circuit's randomness level in a precise manner. RIC does so by using an extra SN whose value controls the amount of variance added to a stochastic number. Our idea of exploiting SC randomness was demonstrated for two applications. First, we showed that

SC's intrinsic randomness provides automatic image dithering, a technique that preserves visual details in a binarized image by random perturbations. Second, we demonstrated that a black-box NN with an SC layer is more resilient against adversarial attacks. This is because SC randomness blurs the NN's output responses to malicious test inputs, making attack generation much costlier.

7.2 Directions for Future Work

Finally, we discuss some potential directions of future SC research, as well as some possible extensions of the work described in this dissertation.

Although significant progress has been made on eliminating randomness-induced errors in SC, the theory for controlling SC randomness is still not fully understood. Most prior work in this direction aimed at reducing or completely removing randomness to improve SC accuracy by introducing deterministic elements. In contrast, this dissertation introduced the RIC (Randomness Injection Circuit) which can add precise amounts of randomness to SC in a way that is beneficial to several applications. A possible extension of RIC is a randomness *reduction* circuit that can decrease SC randomness levels precisely. As Example **6.1** shows, an OMC scaled adder can combine two or more Bernoulli SNs of the same value, and produce another SN with less MSD (mean squared deviation). In fact, the MSD level can be halved (up to a rounding effect) by combining two independent SNs of the same value. Further, as we show next, RICs can be used with OMC adders to reduce randomness levels quantitatively and precisely.

Consider, for example, using two independently generated Bernoulli SNs $\mathbf{X}_1$ and $\mathbf{X}_2$ with $X_1 = X_2$ to produce another SN $\mathbf{Z}$ of the same value, but with the MSD level reduced by 25%. In other words, the goal here is to generate $\mathbf{Z}$ such that $Var(\hat{Z}^{(N)}) = 0.75Var(\hat{X}_1^{(N)})$, and $Z = X_1$. One way to do so, is first using RICs to produce two SNs $\mathbf{Y}_1$ and $\mathbf{Y}_2$ from $\mathbf{X}_1$ and $\mathbf{X}_2$, respectively, each increasing the MSD level by 50%, so $Var(\hat{Y}_1^{(N)}) = 1.5Var(\hat{X}_1^{(N)})$ and $Var(\hat{Y}_2^{(N)}) = 1.5Var(\hat{X}_2^{(N)})$. Then, combine $\mathbf{Y}_1$ and $\mathbf{Y}_2$ using an OMC adder to get $\mathbf{Z}$, which halves the MSD level of $\mathbf{Y}_1$ (and $\mathbf{Y}_2$), i.e., $Var(\hat{Z}^{(N)}) = 0.5Var(\hat{Y}_1^{(N)}) = 0.75Var(\hat{X}_1^{(N)})$.

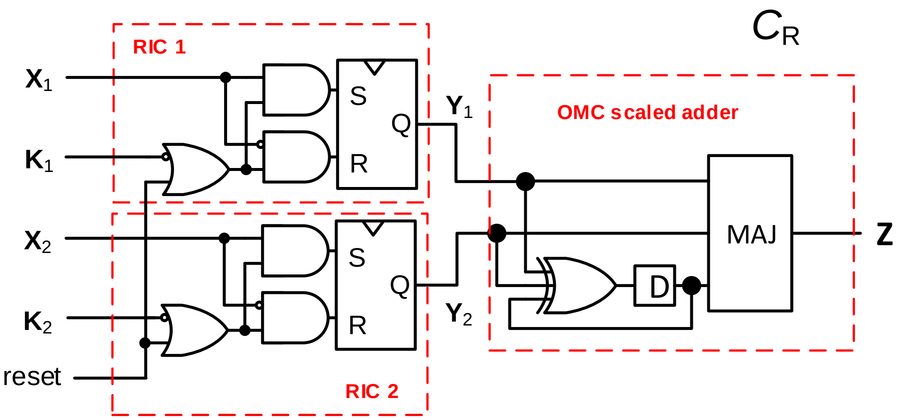


Figure 7.1 SC design for MSD reduction using two RICs and an OMC scaled adder.

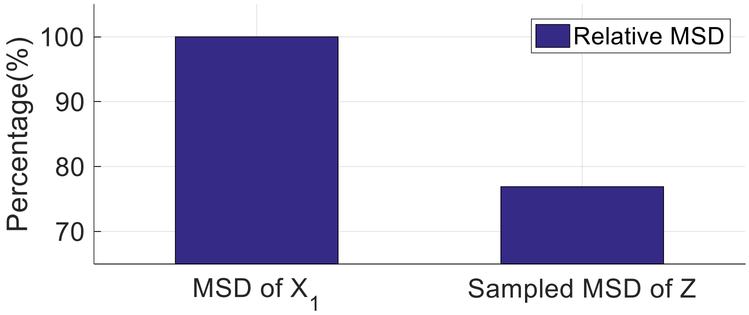


Figure 7.2 Relative (sampled) MSD of **Z**, the output of the circuit C_R in Figure 7.1, compared to the MSD of $\mathbf{X}_1$, one of C_R's inputs.

Figure 7.1 depicts a circuit C_R that contains the needed RICs and the OMC adder. We experimentally verified the preceding idea by setting $X_1 = X_2 = 0.5$ and $N = 32$ for C_R, and measuring the output MSD relative to the input MSD by averaging over 100,000 trials. The result is shown in Figure 7.2, which confirms that the MSD of **Z** is indeed about 25% less than the MSD of $\mathbf{X}_1$ (and $\mathbf{X}_2$).

Another topic worth further investigation is characterizing SC's intrinsic randomness. In this dissertation, we quantified an SN's randomness using MSD, which is the second central moment or variance of the SN. It is clear, however, that an SN's randomness level can depend on other factors like the SN's value, as suggested by

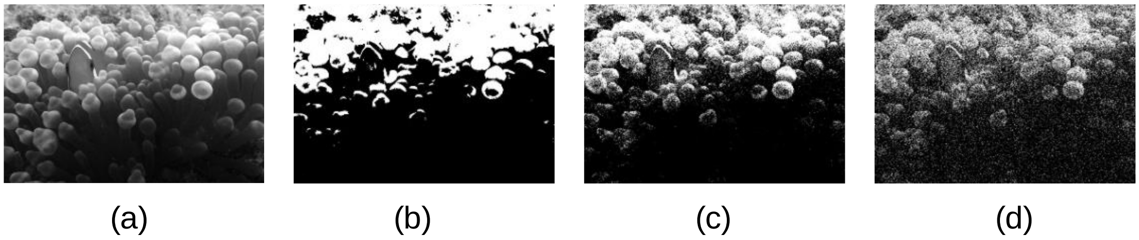
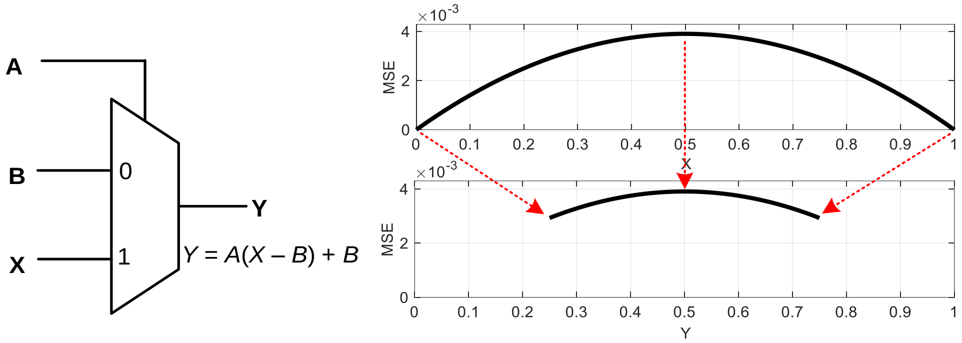


Figure 7.3 Images from the ImageNet dataset [25]: (a) Original grayscale image, (b) binarized image by hard-thresholding, (c) binarized image with SC-based stochastic dithering, and (d) binarized image with SC-based stochastic dithering and an affine transformation

Equation (6.2). Specifically, an SN attains a maximum randomness level when its value is 0.5, and has no randomness when its value is 1 or 0. This is very different from noise models like additive white Gaussian noise (AWGN) commonly used in information theory and communication systems, where the amount of randomness is assumed to be uncorrelated with the signal's value. The fact that there is almost no randomness when the SN's value is close to 1 or 0 can sometimes degrade the performance of applications like image dithering. Figure 7.3 shows images from the ImageNet dataset [25]. In particular, Figure 7.3*c* shows a binarized image with SC-based stochastic dithering. As can be seen, while this image preserves more visual details than the image in Figure 7.3*b*, it still suffers from considerable loss of details in the dark and bright regions. This is because in those regions, the SNs' value is either close to 1 or close to 0, and hence carries almost no randomness, thereby voiding the advantage of stochastic dithering.

One possible way to solve the preceding problem is to shrink the value of the input SNs by applying an affine transform. Figure 7.4*a* shows a stochastic circuit that takes **X** as input, and produces **Y** as output according to the affine transformation $Y = A(X - B) + B$. This transformation can be intuitively understood as shrinking X by a factor A around the center B. In other words, **X**'s value region will decrease from [0, 1] to [$B(1 - A)$, $A + B(1 - A)$]. For instance, when $A = B = 0.5$, **Y**'s value region will be [0.25, 0.75]; see



(a) (b)

Figure 7.4 (a) Stochastic circuit that performs the affine transformation $Y = A(X - B) + B$. (b) Values of X and the corresponding values of Y when $A = B = 0.5$.

Figure 7.4*b*. The significance of this affine transform is that it can avoid an SN from taking values close to 0 or 1, and hence forces the SN to always have sufficient randomness. For example, applying the affine transformation with $A = B = 0.5$ before stochastic dithering and binarization leads to the image of Figure 7.3*d*. Obviously, this image preserves details for the dark and bright regions that are lost in the image of Figure 7.3*c*.

Redesigning SC-based NNs to take into account SC's intrinsic randomness appears to be a promising research direction as well, as it improves the NNs' performance by allowing them to learn how to work in an SC environment. Many existing SC-based NNs are designed for inference tasks only, and their numerical weights are learned offline by training on software-based or BC NNs. Consequently, the learned weights are not tolerant of SC-style randomness, which in turn significantly degrades the performance of the NNs. To cope with this problem, previous work has focused on reducing SC's randomness and introducing deterministic elements to restore NNs' performance, usually at the cost of considerable design efforts and hardware overhead. It is clear, as we have demonstrated in this dissertation, that training an NN with intentionally injected randomness makes the NN more robust of SC-style randomness in the inference phase. To take full advantage of this idea, however, requires more research efforts. Specifically, design aspects like the amount and the type of the injected randomness, as well as the location to inject such randomness, all call for in-depth investigation.

In summary, SC is a rapidly growing, promising but relatively immature computing methodology that still offers many interesting research opportunities to explore. We hope the material presented in this dissertation will contribute to the future research and development of SC and its applications.

APPENDIX

A.1 Proof of Theorem 2.1

By classifying SN inputs into variable and constant parts as in $\mathcal{X} = \{\mathcal{X}_V, \mathcal{X}_C\}$, we can re-write $p_Z = F(f, p_{\mathcal{X}}) = \sum_b f(b)p_{\mathcal{X}}(b)$ (Equation (2.2)) as:

$$p_Z = F(f, p_{\mathcal{X}_V, \mathcal{X}_C}) = \sum_{b_V, b_C}\left[f(b_V, b_C)p_{\mathcal{X}_V, \mathcal{X}_C}(b_V, b_C)\right] \tag{A.1}$$

Further, using the properties of conditional probability, we can re-write $p_{\mathcal{X}_V, \mathcal{X}_C}(b_V, b_C)$ as $p_{\mathcal{X}_C|\mathcal{X}_V}(b_C|b_V) \cdot p_{\mathcal{X}_V}(b_V)$, where the term $p_{\mathcal{X}_C|\mathcal{X}_V}(b_C|b_V)$ is a function of b_C and b_V. Equation (A.1) then becomes

$$\begin{aligned} p_Z &= \sum_{b_V, b_C}\left[f(b_V, b_C)p_{\mathcal{X}_C|\mathcal{X}_V}(b_C|b_V) \cdot p_{\mathcal{X}_V}(b_V)\right] \\ &= \sum_{b_V}\left[p_{\mathcal{X}_V}(b_V) \cdot \sum_{b_C}\left[f(b_V, b_C) \cdot p_{\mathcal{X}_C|\mathcal{X}_V}(b_C|b_V)\right]\right] \end{aligned} \tag{A.2}$$

The summation $g(b_V) = \sum_{b_C}\left[f(b_V, b_C) \cdot p_{\mathcal{X}_C|\mathcal{X}_V}(b_C|b_V)\right]$ is over all combinations of b_C, and hence does not depend on b_C, so we can re-write Equation (A.2) as $p_Z = F(p_{\mathcal{X}_V}) = \sum_{b_V}\left[g(b_V) \cdot p_{\mathcal{X}_V}(b_V)\right]$ which is linear in $p_{\mathcal{X}_V}(b_V)$ with all coefficients $g(b_V)$ in the range [0,1]. The dependency of $F(p_{\mathcal{X}_V})$ on f and $p_{\mathcal{X}_C}$ is implicit via $g(b_V)$ only.

A.2 Proof of Theorem 2.2

Here, the target function is assumed to be in the form of $Z = F(p_{\mathcal{X}_V}) = \sum_{b_V}\left[g(b_V) \cdot p_{\mathcal{X}_V}(b_V)\right]$ (Equation (2.5)). For brevity, we use g_i and p_i to denote $g(b_i)$ and $p_{\mathcal{X}_V}(b_i)$, respectively. The goal is to show that an OMC circuit implements this type of function with minimum MSE, up to a rounding error. The approach taken here is to first construct a circuit C_I that can be shown to achieve a minimum MSE. Then, we show that an OMC circuit achieves the same MSE level as C_I, up to a rounding error.

Consider a circuit C_I with a set of n variable inputs. The number of possible bit patterns that C_I can receive is thus $m = 2^n$; denote these bit patterns by $b_0, b_1, \ldots, b_{m-1}$. Let $B^{(t)}$ be the random variable denoting the bit pattern that C_I receives at clock cycle t, so $p(B^{(t)} = b_i) = p_i$. Suppose C_I is run for k clock cycles. Let k_i be the number of patterns b_i that C_I receives during these k clock cycles. Next, we define the function that C_I implements. Specifically, C_I does the following: after receiving the input patterns $B = B^{(1)}, B^{(2)}, \ldots, B^{(k)}$ in the k clock cycles, the output value it produces is $\hat{Z} = \sum_{i=0}^{m-1} g_i\hat{p}_i$, where $\hat{p}_i = k_i / k$, the proportion of pattern b_i received in these k clock cycles. Notice that both $p = [p_0, p_1, \ldots, p_{m-1}]$ and $\hat{p} = [\hat{p}_0, \hat{p}_1, \ldots, \hat{p}_{m-1}]$ are a probability vector, so their elements sum to 1, i.e. , $\sum_{i=0}^{m-1} p_i = 1$, $\sum_{i=0}^{m-1} \hat{p}_i = 1$. Next, we show that this C_I approximates the exact value $Z = \sum_{i=1}^{m} g_i p_i$ with the minimum possible MSE.

Recall that the output of C_I is $\hat{Z} = \sum_{i=0}^{m-1} g_i\hat{p}_i$, which can be viewed as an estimator for the true value $Z = \sum_{i=1}^{m} g_i p_i$. In SC, we are particularly interested in unbiased estimators whose expected value is the same as the true value. Obviously, C_I produces an unbiased estimator, because $\mathbb{E}(\hat{Z}) = \mathbb{E}(\sum_{i=0}^{m-1} g_i \frac{k_i}{k}) = [\sum_{i=0}^{m-1} g_i \mathbb{E}(\frac{k_i}{k})] = \sum_{i=0}^{m-1} g_i p_i = Z$. We then apply the Cramér–Rao bound (CRB) [41], a well-known bound for the MSE of estimators in the field of statistics, and show that $\hat{Z}$ actually achieves this bound.

The estimator $\hat{Z} = \sum_{i=1}^{m} g_i\hat{p}_i$ is a function of $\hat{p}_i$, implying that C_I first estimates p_i as $\hat{p}_i$, and then uses these estimates to compute $\hat{Z}$, an estimator for Z. Next, we apply the CRB to $\hat{Z}$. The CRB's form for unbiased estimators is:

$$Var_{\boldsymbol{p}}(\hat{Z}) \geq \nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p) \qquad \text{(A.3)}$$

where $Var_{\boldsymbol{p}}(\hat{Z}) = \mathbb{E}(\hat{Z} - Z)^2$ is the MSE for $\hat{Z}$. ∇ is the gradient operator, and, the superscripts T and -1 denotes transpose and inversion. $I(p)$ is the Fisher information matrix of B [41], whose (i, j)-th entry is $\mathbb{E}\left[\frac{\partial \log(p(B;p))}{\partial p_i} \frac{\partial \log(p(B;p))}{\partial p_j}\right]$. In other words, if the MSE of $\hat{Z}$ attains $\nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p)$, then it is optimal in terms of MSE. In the following, we will compute both $Var_{\boldsymbol{p}}(\hat{Z})$ and $\nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p)$, and show that they are equal.

Case I: $k = 1$

First, consider the case when $k = 1$, i.e., C_I is run for only one cycle. We will extend the result for $k = 1$ to the cases for $k \geq 2$.

Since we only consider a single clock cycle, $B = B^{(1)}$. To ease the derivation, we define a set of ancillary Bernoulli (binary 0-1) random variables $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_{m-1}$ where $p(\alpha_i = 1) = p_i$ and $\sum_{i=0}^{m-1} \alpha_i = 1$. Recall that $p(B = b_i) = p_i$, so α_i and B are linked through

$$B = b_i \Leftrightarrow \alpha_i = 1 \tag{A.4}$$

The way α is defined implies that only a single member of α is 1 at any time, and all other members are 0. To begin, we write

$$Var_{\boldsymbol{p}}(\hat{Z}) = \mathbb{E}(\hat{Z} - Z)^2 = \mathbb{E}(\hat{Z}^2) - Z^2 \tag{A.5}$$

where

$$\mathbb{E}(\hat{Z}^2) = \mathbb{E}[(\textstyle\sum_{i=0}^{m-1} g_i \frac{k_i}{k})^2] = \mathbb{E}[(\sum_{i=0}^{m-1} g_i k_i)^2] \tag{A.6}$$

$$= \mathbb{E}[(\textstyle\sum_{i=0}^{m-1} g_i k_i)(\sum_{i=0}^{m-1} g_i k_i)] \tag{A.7}$$

$$= \mathbb{E}[(\textstyle\sum_{i=0}^{m-1} g_i^2 k_i^2)] + \mathbb{E}[(\sum_{i \neq j} g_i g_j k_i k_j)] \tag{A.8}$$

$$= \textstyle\sum_{i=0}^{m-1} g_i^2 \mathbb{E}(k_i^2) + \sum_{i \neq j} g_i g_j \mathbb{E}(k_i k_j) \tag{A.9}$$

$$= \textstyle\sum_{i=0}^{m-1} g_i^2 \mathbb{E}(k_i) + \sum_{i \neq j} g_i g_j \mathbb{E}(k_i k_j) \tag{A.10}$$

$$= \textstyle\sum_{i=0}^{m-1} g_i^2 p_i \tag{A.11}$$

Here, Equation (A.10) is due to the fact that $k_i^2 = k_i$ when $k = 1$; Equation (A.11) is because $\mathbb{E}(k_i k_j) = 0$. Therefore,

$$Var_{\boldsymbol{p}}(\hat{Z}) = \sum_{i=0}^{m-1} g_i^2 p_i - Z^2 \tag{A.12}$$

Next, we compute $\nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p)$. Recall that $\sum_{i=0}^{m-1} p_i = 1$, so the degree of freedom (DoF) of p is $m - 1$. We let $p_{m-1} = 1 - \sum_{i=0}^{m-2} p_i$ to implicitly enforce this constraint. Further, $\nabla F(p) \in \mathbb{R}^{(m-1)\times 1}$ and $I(p) \in \mathbb{R}^{(m-1)\times(m-1)}$. We can write

$$\begin{aligned}\nabla F(p) &= \left[\frac{\partial \sum_{i=0}^{m-1} g_i p_i}{\partial p_1}, \frac{\partial \sum_{i=0}^{m-1} g_i p_i}{\partial p_2}, \dots, \frac{\partial \sum_{i=0}^{m-1} g_i p_i}{\partial p_{m-1}}\right]^{\mathrm{T}} \\ &= [g_0 - g_{m-1}, g_1 - g_{m-1}, .., g_{m-2} - g_{m-1}]^{\mathrm{T}}\end{aligned} \tag{A.13}$$

To compute $I(p)$, consider the (i, j)-th element of $I(p)$

$$I_{(i,j)}(p) = \mathbb{E}\left[\frac{\partial \log(p(B;p))}{\partial p_i} \frac{\partial \log(p(B;p))}{\partial p_j}\right] \tag{A.14}$$

Where $p(B;\ p) = \prod_{i=0}^{m-1} p_i^{\alpha_i}$; recall that B is linked to α through Equation (A.4). Thus, $\log(p(B;\ p)) = \log(\prod_{i=0}^{m-1} p_i^{\alpha_i}) = \sum_{i=0}^{m-1} \alpha_i \log(p_i)$. Hence, $\frac{\partial \log(p(B;p))}{\partial p_i} = \frac{\partial \sum_{i=0}^{m-1} \alpha_i \log(p_i)}{\partial p_i} = \frac{\alpha_i}{p_i} - \frac{\alpha_{m-1}}{p_{m-1}}$. We have

$$\begin{aligned}I_{(i,j)}(p) &= \mathbb{E}\left[\frac{\partial \log(p(B;\boldsymbol{p}))}{\partial p_i} \frac{\partial \log(p(B;\boldsymbol{p}))}{\partial p_j}\right] = \mathbb{E}\left[\left(\frac{\alpha_i}{p_i} - \frac{\alpha_{m-1}}{p_{m-1}}\right)\left(\frac{\alpha_j}{p_j} - \frac{\alpha_{m-1}}{p_{m-1}}\right)\right] \\ &= \mathbb{E}\left[\frac{\alpha_i \alpha_j}{p_i p_j}\right] - \mathbb{E}\left[\frac{\alpha_i \alpha_{m-1}}{p_i p_{m-1}}\right] - \mathbb{E}\left[\frac{\alpha_j \alpha_{m-1}}{p_j p_{m-1}}\right] + \mathbb{E}\left[\frac{\alpha_{m-1}^2}{p_{m-1}^2}\right]\end{aligned} \tag{A.15}$$

Note that since α_i is binary, when $i = j$, $\mathbb{E}\left[\frac{\alpha_i \alpha_j}{p_i p_j}\right] = \mathbb{E}\left[\frac{\alpha_i^2}{p_i^2}\right] = \frac{\mathbb{E}[\alpha_i^2]}{p_i^2} = \frac{\mathbb{E}[\alpha_i]}{p_i^2} = \frac{p_i}{p_i^2} = \frac{1}{p_i}$. When $i \neq j$, $\mathbb{E}\left[\frac{\alpha_i \alpha_j}{p_i p_j}\right] = 0$, because either $\alpha_i = 0$ or $\alpha_j = 0$. We conclude that

$$I_{(i,j)}(p) = \begin{cases} \frac{1}{p_i} + \frac{1}{p_{m-1}} & \text{if } i = j \\ \frac{1}{p_{m-1}} & \text{if } i \neq j \end{cases} \tag{A.16}$$

and

$$I(p) = \begin{bmatrix} \frac{1}{p_0} + \frac{1}{p_{m-1}} & \frac{1}{p_{m-1}} & \cdots & \frac{1}{p_{m-1}} \\ \frac{1}{p_{m-1}} & \frac{1}{p_1} + \frac{1}{p_{m-1}} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \frac{1}{p_{m-1}} & \cdots & \cdots & \frac{1}{p_{m-2}} + \frac{1}{p_{m-1}} \end{bmatrix} \tag{A.17}$$

To compute the inverse of $I(p)$, observe that $I(p)$ can be decomposed as follows:

$$I(p) = \frac{1}{p_{m-1}} \mathbf{1}\mathbf{1}^{\mathrm{T}} + Q$$

where $\mathbf{1} \in \mathbb{R}^{(m-1) \times 1}$ is an all-1 column vector. $Q \in \mathbb{R}^{(m-1) \times (m-1)}$ is a diagonal matrix

$$Q = \begin{bmatrix} \frac{1}{p_0} & 0 & \cdots & 0 \\ 0 & \frac{1}{p_1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \frac{1}{p_{m-2}} \end{bmatrix}$$

whose inverse is easy to compute

$$Q^{-1} = \begin{bmatrix} p_0 & 0 & \cdots & 0 \\ 0 & p_1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & p_{m-2} \end{bmatrix}$$

Applying the Sherman–Morrison formula [45] to $I(p) = \frac{1}{p_{m-1}} \mathbf{1}\mathbf{1}^{\mathrm{T}} + Q$ yields

$$I(p)^{-1} = Q^{-1} - \frac{\frac{1}{p_{m-1}}Q^{-1}\mathbf{1}\mathbf{1}^{\mathrm{T}}Q^{-1}}{1+\frac{1}{p_{m-1}}\mathbf{1}^{\mathrm{T}}Q^{-1}\mathbf{1}} = Q^{-1} - \frac{Q^{-1}\mathbf{1}\mathbf{1}^{\mathrm{T}}Q^{-1}}{p_{m-1}+\mathbf{1}^{\mathrm{T}}Q^{-1}\mathbf{1}}$$

where

$$Q^{-1}\mathbf{1}\mathbf{1}^{\mathrm{T}}Q^{-1} = \begin{bmatrix} p_0^2 & p_0p_1 & \cdots & p_0p_{m-2} \\ p_0p_1 & p_1^2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ p_0p_{m-2} & \cdots & \cdots & p_{m-2}^2 \end{bmatrix}$$

and

$$p_{m-1} + \mathbf{1}^{\mathrm{T}}Q^{-1}\mathbf{1} = p_{m-1} + \sum_{i=0}^{m-2} p_i = 1$$

Hence

$$I(p)^{-1} = Q^{-1} - Q^{-1}\mathbf{1}\mathbf{1}^{\mathrm{T}}Q^{-1} = \begin{bmatrix} p_0 - p_0^2 & -p_0p_1 & \cdots & -p_0p_{m-2} \\ -p_0p_1 & p_1 - p_1^2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ -p_0p_{m-2} & \cdots & \cdots & p_{m-2} - p_{m-2}^2 \end{bmatrix} \qquad \text{(A.18)}$$

Therefore, after simplification, we get

$$\nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p) = [p_0(g_0 - Z), p_1(g_1 - Z), \ldots, p_{m-2}(g_{m-2} - Z)]\nabla F(p)$$

$$= \Sigma_{i=0}^{m-2} p_i g_i^2 - g_{m-1}\Sigma_{i=0}^{m-2} p_i g_i - Z\Sigma_{i=0}^{m-2} p_i g_i + Z g_{m-1}\Sigma_{i=0}^{m-2} p_i \qquad \text{(A.19)}$$

$$= \Sigma_{i=0}^{m-2} p_i g_i^2 + p_{m-1} g_{m-1}^2 - Z^2 = \Sigma_{i=0}^{m-1} g_i^2 p_i - Z^2$$

Comparing Equations (A.12) and (A.19), we see that in this case $Var_{\boldsymbol{p}}(\hat{Z}) = \nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p)$. Since $\hat{Z}$, the estimator of Z constructed by C_I, achieves the CRB, $\hat{Z}$ must has the minimum MSE among all unbiased estimators.

Case 2: $k \geq 2$

The result for $k = 1$ developed above can be extended to the case for $k \geq 2$. Denote the k independent input bit patterns C_I receives as $B^{(1)}$, $B^{(2)}$, …, $B^{(k)}$, each of which gives a single-sample estimate of Z, which we write as $\hat{Z}^{(1)}$, $\hat{Z}^{(2)}$, …, $\hat{Z}^{(k)}$. Noting the fact that $\hat{Z} = \frac{1}{k}(\hat{Z}^{(1)} + \hat{Z}^{(2)} + \ldots + \hat{Z}^{(k)})$, we have

$$Var_{\boldsymbol{p}}(\hat{Z}) = Var_{\boldsymbol{p}}[\tfrac{1}{k}(\hat{Z}^{(1)} + \hat{Z}^{(2)} + \ldots + \hat{Z}^{(k)})] = \tfrac{1}{k}Var_{\boldsymbol{p}}(\hat{Z}^{(1)}) \quad \text{(A.20)}$$

which is $\frac{1}{k}$ times the variance of the case for $k = 1$. $\nabla F(p)$ remains unchanged for $k \geq 2$, as it does not depend on k. Consider the (i-j)-th entry for the Fisher information matrix [41] $I_{(i,j)}(p) = \mathbb{E}\left[\frac{\partial \log(p(B;p))}{\partial p_i}\frac{\partial \log(p(B;p))}{\partial p_j}\right]$. With $k \geq 2$, we have $p(B; p) = \Pi_{r=1}^{k} p(B^{(r)}|p)$ and,

$$\log(p(B;p)) = \log\left(\prod_{r=1}^{k}\prod_{i=0}^{m-1} p_i^{\alpha_i^{(r)}}\right) = \sum_{r=1}^{k}\sum_{i=0}^{m-1}\alpha_i^{(r)}\log(p_i) \quad \text{(A.21)}$$

Hence,

$$\frac{\partial \log(p(B;p))}{\partial p_i} = \frac{\partial \sum_{r=1}^{k}\sum_{i=0}^{m-1}\alpha_i^{(r)}\log(p_i)}{\partial p_i} = \frac{\sum_{r=1}^{k}\alpha_i^{(r)}}{p_i} - \frac{\sum_{r=1}^{k}\alpha_{m-1}^{(r)}}{p_{m-1}} \quad \text{(A.22)}$$

We then get

$$\begin{aligned} I_{(i,j)}(p) &= \mathbb{E}\left[\left(\frac{\sum_{r=1}^{k}\alpha_i^{(r)}}{p_i} - \frac{\sum_{r=1}^{k}\alpha_{m-1}^{(r)}}{p_{m-1}}\right)\left(\frac{\sum_{s=1}^{k}\alpha_j^{(s)}}{p_j} - \frac{\sum_{s=1}^{k}\alpha_{m-1}^{(s)}}{p_{m-1}}\right)\right] \\ &= \mathbb{E}\left[\frac{\sum_{r=1}^{k}\alpha_i^{(r)}\sum_{s=1}^{k}\alpha_j^{(s)}}{p_ip_j}\right] - \mathbb{E}\left[\frac{\sum_{r=1}^{k}\alpha_i^{(r)}\sum_{s=1}^{k}\alpha_{m-1}^{(s)}}{p_ip_{m-1}}\right] \quad \text{(A.23)} \\ &\quad -\mathbb{E}\left[\frac{\sum_{r=1}^{k}\alpha_{m-1}^{(r)}\sum_{s=1}^{k}\alpha_j^{(s)}}{p_jp_{m-1}}\right] + \mathbb{E}\left[\frac{\sum_{r=1}^{k}\alpha_{m-1}^{(r)}\sum_{s=1}^{k}\alpha_{m-1}^{(s)}}{p_{m-1}^2}\right] \end{aligned}$$

Note that when $i = j$,

$$\mathbb{E}\left[\frac{\sum_{r=1}^{k}\alpha_i{}^{(r)}\sum_{s=1}^{k}\alpha_j{}^{(s)}}{p_i p_j}\right] = \mathbb{E}\left[\frac{\sum_{r=1}^{k}\alpha_i{}^{(r)}\sum_{s=1}^{k}\alpha_i{}^{(r)}}{p_i^2}\right] \tag{A.24}$$

$$= \frac{1}{p_i^2}\,\mathbb{E}\left[\sum_{r=1}^{k}\alpha_i^{(r)^2} + \sum_{r,s:r\neq s}\alpha_i^{(r)}\alpha_i^{(s)}\right] \tag{A.25}$$

$$= \frac{1}{p_i^2}\sum_{r=1}^{k}\mathbb{E}[\alpha_i^{(r)^2}] + \frac{1}{p_i^2}\sum_{r,s:r\neq s}\mathbb{E}[\alpha_i^{(r)}]\mathbb{E}[\alpha_i^{(s)}] \tag{A.26}$$

$$= \frac{k}{p_i} + \sum_{r,s:r\neq s} 1 \tag{A.27}$$

because $\alpha_i{}^{(r)} \perp \alpha_i{}^{(s)}$ if $r \neq s$. If $i \neq j$,

$$\mathbb{E}\left[\frac{\sum_{r=1}^{k}\alpha_i^{(r)}\sum_{s=1}^{k}\alpha_j^{(r)}}{p_i p_j}\right] = \frac{1}{p_i p_j}\,\mathbb{E}\left[\sum_{r=1}^{k}\alpha_i^{(r)}\alpha_j^{(r)} + \sum_{r,s:r\neq s}\alpha_i^{(r)}\alpha_j^{(s)}\right] \tag{A.28}$$

$$= \frac{1}{p_i p_j}\sum_{r,s:r\neq s}\mathbb{E}[\alpha_i^{(r)}]\mathbb{E}[\alpha_j^{(s)}] \tag{A.29}$$

$$= \sum_{r,s:r\neq s} 1 \tag{A.30}$$

This leads to

$$I_{(i,\,j)}(p) = \begin{cases} \dfrac{k}{p_i} + \dfrac{k}{p_{m-1}} & \text{if } i = j \\ \dfrac{k}{p_{m-1}} & \text{if } i \neq j \end{cases} \tag{A.31}$$

and

$$I(p) = k\begin{bmatrix} \frac{1}{p_0} + \frac{1}{p_{m-1}} & \frac{1}{p_{m-1}} & \cdots & \frac{1}{p_{m-1}} \\ \frac{1}{p_{m-1}} & \frac{1}{p_1} + \frac{1}{p_{m-1}} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \frac{1}{p_{m-1}} & \cdots & \cdots & \frac{1}{p_{m-2}} + \frac{1}{p_{m-1}} \end{bmatrix} \tag{A.32}$$

Comparing Equations (A.17) and (A.32), we see that $I(p)$ becomes k times larger, compared to $I(p)$ when k = 1. Therefore $\nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p)$ becomes K times smaller, compared to the case of $k = 1$. Since both $Var_{\boldsymbol{p}}(\hat{Z})$ and $\nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p)$ are shrunk k times compared to the case of $k = 1$, we conclude that $Var_{\mathbf{p}}(\hat{Z}) = \nabla F(p)^{\mathrm{T}} I(p)^{-1} \nabla F(p)$ also holds for all $k \geq 2$. This implies that for all input lengths, C_I achieves the lowest possible MSE.

Generalization to OMC Circuits

We have shown that the circuit C_I that computes $\hat{Z} = \sum_{i=0}^{m-1} g_i \hat{p}_i$ achieves an optimal result in terms of MSE reduction. Further, from Theorem **4.4**, we also know that the measured output value of an OMC circuit C_{OMC} is $\hat{Z}_{\mathrm{C}} = \sum_{i=0}^{m-1} \frac{a_i}{q} \frac{k_i}{N} + \frac{\tilde{\varepsilon}}{N} = \sum_{i=0}^{m-1} g_i \hat{p}_i + \frac{\tilde{\varepsilon}}{N}$. Comparing $\hat{Z}$ and $\hat{Z}_{\mathrm{C}}$, we see that C_{OMC} produces the same output value as C_I, except for the term $\frac{\tilde{\varepsilon}}{N}$ induced by the rounding error. Therefore, we conclude that an OMC circuit achieves minimum possible MSE up to a rounding error.

A.3 Proof of Theorem 3.1

Let $C_{\mathfrak{D}}$ denote a stochastic circuit C realizing the Boolean function $f(x_1, x_2, \ldots, x_n)$ with an isolator placement $\mathfrak{D}$. Let $F_{\mathfrak{D}}(f_{\mathfrak{D}}, p_{\mathcal{X}_{\mathfrak{D}}})$ be the stochastic function $C_{\mathfrak{D}}$ implements, and let $\mathcal{X} = \{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n\}$ be a set of input SNs for C.

Lemma A.1: If $\mathcal{X}_{\mathfrak{D}} = \{\mathbf{X}_1(t_1), \mathbf{X}_2(t_2), \ldots, \mathbf{X}_n(t_n)\}$, then $f_{\mathfrak{D}} = f$.

Proof of Lemma A.1: Since $\mathcal{X}_{\mathfrak{D}} = \{\mathbf{X}_1(t_1), \mathbf{X}_2(t_2), \ldots, \mathbf{X}_n(t_n)\}$, $f_{\mathfrak{D}} = f_{\mathfrak{D}}(x_1(t_1), x_2(t_2), \ldots, x_n(t_n))$ for some $t_1, t_2, \ldots, t_n$. To prove that $f_{\mathfrak{D}} = f$, it is sufficient to show that $f_{\mathfrak{D}}(l_1, l_2, \ldots, l_n) = f(l_1, l_2, \ldots, l_n)$ for all possible n-bit vectors $b = \{l_1, l_2, \ldots, l_n\}$. Let $t_0 = \max(t_1, t_2, \ldots, t_n)$. Consider a set of bit-streams of values $l_1, l_2, \ldots, l_n$ that are fed into the input lines of C and $C_{\mathfrak{D}}$ for t clock cycles, where $t > t_0$. Obviously, $t > t_i$ for $i = 1, 2,.., n$. Then, at clock period t, $x_i(t_i) = l_i$ for all i. Thus, $C_{\mathfrak{D}}$ must output the same value as C, since $C_{\mathfrak{D}}$ is simply a circuit with isolators. In other words, at clock period t, we must have $x_i(t_i) = x_i = l_i$ for all i, and $f_{\mathfrak{D}}(l_1, l_2, \ldots, l_n) = f(l_1, l_2, \ldots, l_n)$. Since the above argument holds for any n-bit vector $b = \{l_1, l_2, \ldots, l_n\}$, we conclude that $f_{\mathfrak{D}} = f$.

We now use Lemma A.1 to prove Theorem **3.1**. Since $\mathcal{X}_{\mathfrak{D}} = \{\mathbf{X}_1(t_1), \mathbf{X}_2(t_2), \ldots, \mathbf{X}_n(t_n)\}$, Lemma A.1 implies that $f_{\mathfrak{D}} = f$. The stochastic function realized by $C_{\mathfrak{D}}$ is thus $F_{\mathfrak{D}}(f_{\mathfrak{D}}, p_{\mathcal{X}_{\mathfrak{D}}}) = F_{\mathfrak{D}}(f, p_{\mathcal{X}_{\mathfrak{D}}})$ taking the following form:

$$F_{\mathfrak{D}}(f, p_{\mathcal{X}_{\mathfrak{D}}}) = \sum_{l_1,\cdots,l_n}\left[p_{\mathbf{X}_1(t_1)\cdots\mathbf{X}_n(t_\mathrm{n})}(l_1,\cdots,l_n)f(l_1,\cdots,l_n)\right] \tag{A.33}$$

Furthermore, since $t_i \neq t_j$ for every $\mathbf{X}_i$, $\mathbf{X}_j$ pair that needs decorrelation, we know that $p_{\mathbf{X}_1(t_1)\cdots\mathbf{X}_n(t_\mathrm{n})} = p_{\mathbf{X}_1(t_1)} \cdot p_{\mathbf{X}_2(t_2)} \cdots p_{\mathbf{X}_n(t_n)}$. This is because we assume that a set of SNs become mutually independent, if the correlated SNs in this set are delayed by different clock cycles. Also notice that $p_{\mathbf{X}_{j(t_j)}}(b_j) = p_{\mathbf{X}_j}(b_j)$ for all j. We then have

$$F_{\mathfrak{D}}(f_{\mathfrak{D}}, p_{\mathcal{X}_{\mathfrak{D}}}) = \sum_{l_1,\cdots,l_n}\left[\prod_{j=1}^{n} p_{\mathbf{X}_j}(l_j)\, f(l_1,\cdots,l_n)\right] = F(f, p_{\mathcal{X}_{\mathrm{IND}}}) \tag{A.34}$$

which completes the proof.

A.4 Proof of Theorem 4.4

To get the number of 1s in the output **Z**, we first compute the total number of states that C will advance after receiving the N-bit SNs, which is $\sum_{i=0}^{m-1} a_i k_i + a_{-1}$. The number of 1s in **Z** is therefore $\left\lfloor \frac{1}{q}\left(\sum_{i=0}^{m-1} a_i k_i + a_{-1}\right)\right\rfloor = \left\lfloor \sum_{i=0}^{m-1} \frac{a_i}{q} k_i + \frac{a_{-1}}{q}\right\rfloor = \sum_{i=0}^{m-1} \frac{a_i}{q} k_i + \frac{a_{-1}}{q} - \epsilon$, where $\epsilon \in [0, 1]$ is an offset term that takes into account the floor operation. Setting $\tilde{\epsilon} = \frac{a_{-1}}{q} - \epsilon$, we obtain the number of 1s in **Z** as $N_{1,\mathrm{Z}} = \sum_{i=0}^{m-1} \frac{a_i}{q} N_i + \tilde{\epsilon}$, which completes the first part of the proof. The value of **Z** is thus the expectation of $\frac{1}{N}\sum_{i=0}^{m-1} \frac{a_i}{q} k_i + \frac{\tilde{\epsilon}}{N}$, which is $Z = \mathbb{E}\left(\frac{1}{N}\sum_{i=0}^{m-1} \frac{a_i}{q} k_i + \frac{\tilde{\epsilon}}{N}\right) = \sum_{i=0}^{m-1} \frac{a_i}{q}\mathbb{E}\left(\frac{k_i}{N}\right) + \frac{\mathbb{E}(\tilde{\epsilon})}{N} = \sum_{i=0}^{m-1} \frac{a_i}{q} p_{\mathcal{X}_\mathrm{V}}(b_i) + \frac{1}{N}\mathbb{E}(\tilde{\epsilon})$. This completes the second part of the proof.

A.5 Proof of Theorem 6.1

Let C_{RIC} be a RIC as depicted in Figure 6.3. The STG of C_{RIC} is a two-state Markov chain, whose t-step state-transition probability matrix is $\mathrm{P}^t = \begin{bmatrix} 1-X & X \\ 1-X & X \end{bmatrix} + K^t\begin{bmatrix} X & -X \\ X-1 & 1-X \end{bmatrix}$ [57]. In other words, the (i, j)-th element of P^t denotes the probability of C_{RIC} being in state s_j after t clock cycles, given that C_{RIC} is currently in state s_i. To show that the output value Y of C_{RIC} is the same as C_{RIC}'s input value X, suppose that C_{RIC} is initialized to state s_1 with probability X, and hence to state s_0 with probability $1 - X$, i.e., the initial state probability vector is $\boldsymbol{\pi} = [1 - X, X]$. Then, for all t,

$$\begin{aligned} \boldsymbol{\pi}\mathrm{P}^t &= [1-X \quad X]\left(\begin{bmatrix} 1-X & X \\ 1-X & X \end{bmatrix} + K^t\begin{bmatrix} X & -X \\ X-1 & 1-X \end{bmatrix}\right) \\ &= [1-X \quad X] = \boldsymbol{\pi} \end{aligned} \tag{A.35}$$

which implies that the probability of C_{RIC} being in state s_1 in any clock cycle is X. Since the output value associated with state s_1 is 1, the probability of a 1 appearing on the output line in any clock cycle is therefore X, i.e. $Y = X$. This completes the first part of the proof.

Next, we show that $Var(\hat{Y}^{(N)}) = Var(\hat{X}^{(N)}) \cdot G(K, N)$.

Lemma A.2: Let $\mathbf{Y}$ be the output SN of a RIC. The covariance between its i-th bit and its j-th bit is $Cov(\mathbf{Y}^{(i)}, \mathbf{Y}^{(j)}) = K^t(1 - X)X$, where $t = |i - j|$.

Proof of Lemma A.2:

$$Cov(\mathbf{Y}^{(i)}, \mathbf{Y}^{(i+t)}) = \mathbb{E}[(\mathbf{Y}^{(i)} - Y)(\mathbf{Y}^{(i+t)} - Y)] \tag{A.36}$$

$$= \mathbb{E}(\mathbf{Y}^{(i)}\mathbf{Y}^{(i+t)}) - Y^2 \tag{A.37}$$

$$= p(\mathbf{Y}^{(i)} = 1, \mathbf{Y}^{(i+t)} = 1) - X^2 \tag{A.38}$$

$$= p(\mathbf{Y}^{(i+t)} = 1 \mid \mathbf{Y}^{(i)} = 1)X - X^2 \tag{A.39}$$

where Equation (A.38) is due to the fact that $\mathbf{Y}^{(i)} \cdot \mathbf{Y}^{(i\,+\,t)} = 1$ if and only if both $\mathbf{Y}^{(i)}$ and $\mathbf{Y}^{(i\,+\,t)}$ are 1, and the fact that $Y = X$. Equation (A.39) is obtained by conditioning on the event $\mathbf{Y}^{(i)} = 1$. Note that the conditional probability $p(\mathbf{Y}^{(i\,+\,t)} = 1 \mid \mathbf{Y}^{(i)} = 1)$ is equal to the probability that the state $S = s_1$ at clock cycle $i + t$, given that the state $S = s_1$ at clock cycle i. This conditional probability can thus be computed by using the t-step state-transition probability defined previously. Specifically, let the state probability vector be $\boldsymbol{\pi} = [0, 1]$ at clock cycle i. Then $\boldsymbol{\pi}\mathrm{P}^t = [0, 1]\mathrm{P}^t = [(1 - X)(1 - K^t), X + K^t(1 - X)]$, meaning that the probability of $S = s_1$ at clock cycle $i + t$ is $X + K^t(1 - X)$. This further implies that $p(\mathbf{Y}^{(i\,+\,t)} = 1 \mid \mathbf{Y}^{(i)} = 1) = X + K^t(1 - X)$. Plugging this probability back to Equation (A.39), we have

$$Cov(\mathbf{Y}^{(i)}, \mathbf{Y}^{(i\,+\,t)}) = [X + K^t(1 - X)]X - X^2 \tag{A.40}$$

$$= K^t(1 - X)X \tag{A.41}$$

which completes the proof of Lemma A.2.

Lemma A.3: $\Sigma_{j=1}^{N-1}\Sigma_{i=j+1}^{N} Cov(\mathbf{Y}^{(i)}, \mathbf{Y}^{(j)}) = N^2 \cdot Var(\hat{X}^{(N)}) \cdot [G(K, N) - 1] / 2$

Proof of Lemma A.3: From Lemma A.2, we have

$$\Sigma_{j=1}^{N-1}\Sigma_{i=j+1}^{N} Cov(\mathbf{Y}^{(i)}, \mathbf{Y}^{(j)}) = \Sigma_{j=1}^{N-1}\Sigma_{i=j+1}^{N} K^{i\text{-}j}(1 - X)X \tag{A.42}$$

$$= (1 - X)X \cdot \Sigma_{j=1}^{N-1}\Sigma_{t=1}^{N-j} K^t \tag{A.43}$$

$$= (1 - X)X \cdot T \tag{A.44}$$

where Equation (A.43) is obtained via change of variable $t = i - j$. Also, we set $T = \Sigma_{j=1}^{N-1}\Sigma_{t=1}^{N-j} K^t$ here for brevity. On expanding the summation of T, we obtain

$$T = (N - 1)K + (N - 2)K^2 + \ldots + 2K^{N-2} + K^{N-1} \tag{A.45}$$

This leads to $KT - T = -NK + \Sigma_{t=1}^{N} K^t = -NK + \frac{K(1-K^N)}{1-K} = \frac{K(1-K^N)-NK(1-K)}{1-K}$. Solving for T yields

$$T = \frac{K[(1-K)N+K^N-1]}{(1-K)^2} = N[G(K, N) - 1] / 2 \tag{A.46}$$

On plugging Equation (A.46) back to Equation (A.44), we get

$$\begin{aligned}\Sigma_{j=1}^{N-1}\Sigma_{i=j+1}^{N}Cov(\mathbf{Y}^{(i)}, \mathbf{Y}^{(j)}) &= (1 - X)X \cdot T = N \cdot Var(\hat{X}^{(N)}) \cdot T \\ &= N^2 \cdot Var(\hat{X}^{(N)}) \cdot [G(K, N) - 1] / 2\end{aligned} \tag{A.47}$$

which completes the proof of Lemma A.3.

Finally, we use Lemma A.3 to prove $Var(\hat{Y}^{(N)}) = Var(\hat{X}^{(N)}) \cdot G(K, N)$. We write

$$Var(\hat{Y}^{(N)}) = Var(\tfrac{1}{N}\Sigma_{i=1}^{N}\mathbf{Y}^{(i)}) \tag{A.48}$$

$$= \frac{1}{N^2}\Sigma_{i=1}^{N}Var(\mathbf{Y}^{(i)}) + \frac{2}{N^2}\Sigma_{j=1}^{N-1}\Sigma_{i=j+1}^{N}Cov(\mathbf{Y}^{(i)}, \mathbf{Y}^{(j)}) \tag{A.49}$$

$$= Var(\hat{X}^{(N)}) + Var(\hat{X}^{(N)}) \cdot [G(K, N) - 1] \tag{A.50}$$

$$= Var(\hat{X}^{(N)}) \cdot G(K, N) \tag{A.51}$$

where Equation (A.50) is obtained by plugging Equation (A.47) into Equation (A.49). This completes the second part of the proof.

BIBLIOGRAPHY

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M. and Kudlur, M., "Tensorflow: A system for large-scale machine learning," *Proc. Symp. on Operating Sys. Design and Implementation*, 16, pp. 265-283, 2016.

[2] Alaghi, A. and Hayes, J. P., "Dimension reduction in statistical simulation of digital circuits," *Proc. on Theory of Modeling and Simulation*, pp. 1-8, 2015.

[3] Alaghi, A. and Hayes, J. P., "Exploiting correlation in stochastic circuit design," *Proc. Int. Conf. Computer Design*, pp. 39-46, 2013.

[4] Alaghi, A. and Hayes, J. P., "Fast and accurate computation using stochastic circuits," *Proc. Conf. on Design, Automation & Test in Europe*, pp. 76:1-76:4, 2014.

[5] Alaghi, A. and Hayes, J. P., "On the functions realized by stochastic computing circuits," *Proc. Great Lakes Symp. on VLSI*, pp. 331-336, 2015.

[6] Alaghi, A. and Hayes, J. P., "STRAUSS: spectral transform use in stochastic circuit synthesis," *IEEE Trans. CAD,* 34, 1770-1783, 2015.

[7] Alaghi, A. and Hayes, J. P., "Survey of stochastic computing," *ACM Trans. Embedded Computing Systems*, 12, 2s, pp. 92:11-92:19, 2013.

[8] Alaghi, A., Li, C. and Hayes, J. P., "Stochastic circuits for real-time image-processing applications," *Proc. Design Automation Conf.*, pp. 1-6, 2013.

[9] Alaghi, A., Ting, P., Lee, V.T. and Hayes, J. P., "Accuracy and correlation in stochastic computing." In Gross, W.J. and V.C. Gaudet (eds.). *Stochastic Computing: Techniques and Applications,* Springer Nature, pp 77-102, 2019.

[10] Ananth, R. S., "Programmable supervisory circuit and applications thereof," US Patent no. 6,618,711, 2003.

[11] Ardakani, A., Leduc-Primeau, F. and Gross, W.J., "Hardware implementation of FIR/IIR digital filters using integral stochastic computation," *Proc. Int. Conf. on Acoustics, Speech and Signal Processing*, pp. 6540-6544, 2016.

[12] Ardakani, A., Leduc-Primeau, F., Onizawa, N., Hanyu, T. and Gross, W.J., "VLSI implementation of deep neural networks using integral stochastic computing," *Proc. Int. Symp. on Turbo Codes and Iterative Information Processing,* pp. 216-220, 2016.

[13] Athalye, A., Carlini, N., and Wagner, D., “Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples,” *Proc. Int. Conf. on Machine Learning*, pp. 274-283, 2018.

[14] Bade, S.L. and Hutchings, B.L., “FPGA-based stochastic neural networks-implementation,” *Proc. Workshop on FPGAs for Custom Computing* Machines, pp. 189-198, 1994.

[15] Braendler, D., Hendtlass, T. and O'Donoghue, P., “Deterministic bit-stream digital neurons,” *IEEE Trans. Neural Nets.*, 13, pp. 1514-1525, 2002.

[16] Brown, B.D. and Card, H.C., “Stochastic neural computation I: computational elements,” *IEEE Trans. Computers*, 50, pp. 891-905, 2001.

[17] Canals, V., Morro, A., Oliver, A., Alomar, M.L. and Rosselló, J.L., “A new stochastic computing methodology for efficient neural network implementation,” *IEEE Trans. Neural Nets. & Learning Sys*, 27(3), pp. 551-564, 2016.

[18] Carlini, N. and Wagner, D. “Towards evaluating the robustness of neural networks,” *Proc. Symp. on Security and Privacy*, pp. 39-57, 2017.

[19] Chang, Y.N. and Parhi, K.K., “Architectures for digital filters using stochastic computing,” *Proc. Int. Conf. on Acoustics, Speech and Signal Processing* pp. 2697-2701, 2013.

[20] Chen, P.-Y., Zhang, H., Sharma, Y., Yi, J., and Hsieh, C.-J., “ZOO: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models,” *Proc. Workshop on Artificial Intelligence and Security* , pp.15-26, 2017.

[21] Chen, T.H., Alaghi, A. and Hayes, J.P., “Behavior of stochastic circuits under severe error conditions,” *Info. Tech.*, 56, 4, pp. 182-191, 2014.

[22] Chen, T.-H. and Hayes, J. P., “Analyzing and controlling accuracy in stochastic circuits,” *Proc. Int. Conf. on Computer Design*, 367-373, 2014.

[23] Chen, T.-H. and Hayes, J. P., “Equivalence among stochastic logic circuits and its application,” *Proc. Design Automation Conf.*, pp. 131:1-131:6, 2015.

[24] Choi, S. S., Chia, S. H. and Tappert, C., “A survey of binary similarity and distance measures,” *Journ. Systemics, Cybernetics and Informatics*, 8, pp. 43-48, 2010.

[25] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K. and Fei-Fei, L., “ImageNet: A large-scale hierarchical image database,” *Proc. Conf. on Comp. Vision and Pattern Recognition*, pp. 248-255, 2009.

[26] Dong, Q.T., Arzel, M., Jego, C. and Gross, W.J., “Stochastic decoding of turbo codes,” *IEEE Trans. Signal Processing*, 58(12), pp.6421-6425, 2010.

[27] Faraji, S.R., Najafi, M.H., Li, B., Bazargan, K. and Lilja, D.J., “Energy-efficient convolutional neural networks with deterministic bit-stream processing,” *Proc. Conf. on Design, Automation & Test in Europe*, pp. 1736-1741, 2019.

[28] Friedman, J.S., Droulez, J., Bessière, P., Lobo, J. and Querlioz, D., "Approximation enhancement for stochastic Bayesian inference," *Int. Journal of Approximate Reasoning*, 85, pp.139-158, 2017

[29] Gaines, B. R., "Stochastic computing systems," *Advances in Inform. Systems Science*, 2, pp. 37-172, 1969.

[30] Gallager, R.G. *Discrete Stochastic Processes*. Springer, 1996.

[31] Golomb, S. W. and Gong, G. *Signal Design for Good Correlation*. Cambridge University Press, 2005.

[32] Gross, W.J. and V.C. Gaudet (eds.). *Stochastic Computing*: *Techniques and Applications*. Springer Nature, 2019.

[33] Gross, W.J., Gaudet, V.C. and Milner, A., "Stochastic implementation of LDPC decoders," *Proc. Asilomar Conf. on Signals, Systems & Comp.*, pp. 713-717, 2005.

[34] Gubner, J. A. *Probability and Random Processes for Electrical and Computer Engineers.* Cambridge University Press, 2006.

[35] Guo, T., "Cloud-based or on-device: An empirical study of mobile deep inference," *Proc. Int. Conf. on Cloud Engineering*, pp. 184-190, 2018.

[36] He, K., Zhang, X., Ren, S. and Sun, J., "Deep residual learning for image recognition," *Proc. Conf. on Comp. Vision and Pattern Recognition*, pp. 770-778, 2016.

[37] Hikawa, H., "A digital hardware pulse-mode neuron with piecewise linear activation function," *IEEE Trans. Neural Nets.*, 14, pp. 1028-1037, 2003.

[38] Jeavons, P., Cohen, D. A. and Shawe-Taylor, J., "Generating binary sequences for stochastic computing," *IEEE Trans. Information Theory*, 40, pp. 716-720, 1994.

[39] Jenson, D. and Riedel, M., "A deterministic approach to stochastic computation," *Proc. Proc. Int. Conf. on Computer-Aided Design*, pp. 1-8, 2016.

[40] Kearfott, R.B., "Interval computations: Introduction, uses, and resources," *Euromath Bulletin*, 2(1), pp. 95-112, 1996.

[41] Keener, R.W. *Theoretical Statistics: Topics for a Core Course*. Springer, 2010.

[42] Knag, P., Lu, W. and Zhang, Z., "A native stochastic computing architecture enabled by memristors," *IEEE Trans. Nanotech.*, 13(2), pp. 283-293, 2014.

[43] Kohavi, Z. and Jha, N.K. *Switching and Finite Automata Theory*, 3rd ed. Cambridge Univ. Press, 2010.

[44] Krizhevsky, A. and Hinton, G. "Learning multiple layers of features from tiny images," *University of Toronto Tech. Report*. 1(4), 2009.

[45] Laub, A.J. *Matrix Analysis for Scientists and Engineers*. SIAM, 2005.

[46] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., "Gradient-based learning applied to document recognition," *Proc. IEEE*, 86(11), pp. 2278-2324, 1998.

[47] Lee, V.T., Alaghi, A. and Ceze, L., "Correlation manipulating circuits for stochastic computing," *Proc. Conf. on Design, Automation & Test in Europe*, pp. 1417-1422, 2018.

[48] Lee, V.T., Alaghi, A., Hayes, J.P., Sathe, V. and Ceze, L., "Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing." *Proc. Conf. on Design, Automation & Test in Europe*, pp. 13-18, 2017.

[49] Lee, X.R., Chen, C.L., Chang, H.C. and Lee, C.Y., "A 7.92 Gb/s 437.2 mW stochastic LDPC decoder chip for IEEE 802.15. 3c applications," *IEEE Trans. Circuits and Systems I*, 62(2), pp.507-516, 2015.

[50] Li, J., Yuan, Z., Li, Z., Ding, C., Ren, A., Qiu, Q., Draper, J. and Wang, Y., "Hardware-driven nonlinear activation for stochastic computing based deep convolutional neural networks," *Proc. Int. Joint Conf. on Neural Nets.*, pp. 1230-1236, 2017.

[51] Li, P. and Lilja, D.J., "Using stochastic computing to implement digital image processing algorithms," *Proc. Int. Conf. Computer Design*, pp. 154-161, 2011.

[52] Li, P., Lilja, D. J., Qian, W., Bazargan, K. and Riedel, M. D., "The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic," *Proc. Int. Conf. on Computer-Aided Design*, pp. 480-487, 2012.

[53] Liu, X., Cheng, M., Zhang, H. and Hsieh, C.J., "Towards robust neural networks via random self-ensemble," *Proc. European Conf. on Comp. Vision*, pp. 369-385, 2018.

[54] Manohar, R., "Comparing stochastic and deterministic computing," *IEEE Comp. Arch. Letters*, 14(2), pp. 119-122, 2015.

[55] Marconi, T. and Sorin C., "Dynamic bitstream length scaling energy effective stochastic LDPC decoding," *Proc. Great Lakes Symp. on VLSI*, pp. 245-248, 2015.

[56] Marin, S.T., Reboul, J.Q. and Franquelo, L.G., "Digital stochastic realization of complex analog controllers," *IEEE Trans. Industrial Electronics*, 49(5), pp.1101-1109, 2002.

[57] Mizera, A., Pang, J. and Yuan, Q., "Reviving the two-state Markov chain approach," *IEEE/ACM Trans. Computational Biology and Bioinformatics*, 15(5) pp. 1525-1537, 2018.

[58] Nickolls, J., Buck, I. and Garland, M., "Scalable parallel programming," *Proc. Hot Chips 20 Symp.*, pp. 40-53, 2008.

[59] Paler, A., Kinseher, J., Polian, I. and Hayes, J.P., "Approximate simulation of circuits with probabilistic behavior." *Proc. Int. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Sys.*, pp. 95-100, 2013.

[60] Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z.B. and Swami, A., "Practical black-box attacks against machine learning," *Proc. Asia Conf. on Comp. and Communications Security*, pp. 506-519, 2017.

[61] Silberman, N., Hoiem, D., Kohli, P. and Fergus, R., "Indoor segmentation and support inference from rgbd images," *Proc. European Conf. on Comp. Vision*, pp. 746-760, 2012.

[62] Sim, H. and Lee, J., "A new stochastic computing multiplier with application to deep convolutional neural networks," *Proc. Design Automation Conf.*, pp. 1-6, 2017.

[63] Simonyan, K. and Zisserman, A. "Very deep convolutional networks for large-scale image recognition," *Proc. Int. Conf. on Learning Representations*, 2015.

[64] Stathis, P. et al. "An evaluation survey of binarization algorithms on historical documents," *Proc. Int. Conf. on Learning Representations*, pp. 1-4, 2008.

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

[66] Ranjan, R. K., Singhal, V., Somenzi, F., Brayton, R.K., "Using combinational verification for sequential circuits." *Proc. Conf. on Design, Automation & Test in Europe*, pp.138-144, 1999.

[67] Rho, J.K., Hachtel, G.D., Somenzi, F. and Jacoby, R.M., "Exact and heuristic algorithms for the minimization of incompletely specified state machines," *IEEE Trans. CAD*, 13, pp. 167-177, 1994.

[68] Tehrani, S.S., Gross, W.J. and Mannor, S. "Stochastic decoding of LDPC codes." *IEEE Communications Letters*, 10(10), pp. 716-718, 2006.

[69] Ting, P.-S. and Hayes, J. P., "Isolation-based decorrelation of stochastic circuits," *Proc. Int. Conf. Computer Design*, pp. 88-95, 2016.

[70] Ting, P.-S. and Hayes, J. P., "On the role of sequential circuits in stochastic computing," *Proc. Great Lakes Symp. on VLSI*, pp. 475-478, 2017.

[71] Ting, P.-S. and Hayes, J. P., "Stochastic logic realization of matrix operations," *Proc. Euromicro Conf. on Digital Sys. Design*, pp. 356-364, 2014.

[72] Ting, P. and Hayes, J. P., "Eliminating a hidden error source in stochastic circuits," *Proc. Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pp. 1-6, 2017. Received the conference's **Best Paper** award.

[73] Ting, P. and Hayes, J. P., "Maxflow: minimizing latency in hybrid stochastic-binary systems," *Proc. Great Lakes Symp. on VLSI*, pp. 21-26, 2018.

[74] Ting, P. and Hayes, J. P., "Removing constant-induced errors in stochastic circuits," *To appear in IET Comp. & Digital Techniques*, 2018.

[75] Vahapoglu, E. and Altun, M., "Accurate synthesis of arithmetic operations with stochastic logic," *Proc. ISVLSI*, pp.415-420, 2016.

[76] Van Daalen, M., Jeavons, P. and Shawe-Taylor, J., “A stochastic neural architecture that exploits dynamically reconfigurable FPGAs,” *Proc. Workshop on FPGAs for Custom Computing Machines*, pp. 202-211, 1993.

[77] Yuan, B. and Parhi, K.K., “Belief propagation decoding of polar codes using stochastic computing,” *Proc. Int. Symp. on Circuits and Systems*, pp. 157-160, 2016.

[78] Yue, L., Ganesan, P., Sathish, B.S., Manikandan, C., Niranjan, A., Elamaran, V. and Hussein, A.F., “The importance of dithering technique revisited with biomedical images—a survey,” *IEEE Access*, 7, pp. 3627-3634, 2019.