Spring 2018

Generalized adaptive variable bit truncation model for approximate stochastic computing

Keerthana Pamidimukkala

GENERALIZED ADAPTIVE VARIABLE BIT TRUNCATION MODEL FOR APPROXIMATE STOCHASTIC COMPUTING

by

KEERTHANA PAMIDIMUKKALA

A THESIS

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

2018

Approved by

Dr. Minsu Choi, Advisor
Dr. R. Joe Stanley
Dr. Kurt Kosbar

ABSTRACT

Stochastic computing as a computing paradigm is currently undergoing revival as the advancements in technology make it applicable especially in the wake of the need for higher computing power for emerging applications. Recent research in stochastic computing exploits the benefits of approximate computing, called Approximate Stochastic Computing (ASC), which further reduces the operational overhead in implementing stochastic circuits. A mathematical model is proposed to analyze the efficiency and error involved in ASC. Using this mathematical model, a new generalized adaptive method improving on ASC is proposed in the current thesis. The proposed method has been discussed with two possible implementation variants - Area efficient and Time efficient. The proposed method has also been implemented in Matlab to compare against ASC and is shown to perform better than previous approaches for error-tolerant applications.

ACKNOWLEDGMENTS

I would like to thank NSF for sponsoring my research through Dr. Minsu Choi. I would like to thank Dr. Minsu Choi for his continued support and encouragement and for advising me through the research in a short span of time. I would also like to thank Dr. Kosbar and Dr. Stanley for accepting my request to be on the thesis committee in such short notice.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

1. INTRODUCTION

The modern history of computing architectures is comprised of multiple approaches and refinements to the field of computing since its inception ranging from Analog architectures in 1822 [5] to modern architectures of the current day. Most successful and prominent ones that are in use at present are based on Arithmetic and Logic Units (ALUs) working on the binary representation of numbers. Due to advancements in the field of VLSI, higher computational efficiency was made possible which paved the way for microprocessor-based designs to dominate the field of computing.

Further improvements in CMOS technology among other technological advancements have made very high-speed operations in the range of Trillion Floating Point Operations Per Second (TFLOPS) [12]. Other methods of computing were ahead of time in technology and had difficulties in being realized during initial stages of VLSI improvements; for example - Quantum computing, Optical computing, and Stochastic computing among others. The current document focuses on Stochastic Computing (SC) since the enhancements in technology enable the use of SC in realistic applications.

1.1. STOCHASTIC COMPUTING

Stochastic computing refers to the method of using probabilistic representations of numbers to perform numerical operations. Stochastic computing was introduced as a concept of computing in 1956 [13]. This concept was further improved with implementations throughout the 1960s [7, 9]. To explain the concept of stochastic computing, it is necessary to understand stochastic representation of data.

1.1.1. Stochastic Representation of Information. Consider binary representation. The binary representation of numbers depends on a set number of bits to represent each number. Each position corresponds to a different weight in the representation. The weighted sum of all the bits with appropriate weights will reproduce the number stored. All arithmetic operations can be done on the Binary information similar to decimal information. In contrast with that, Stochastic computing depends on stream sof data to represent one single number. Said stream is a random variable with a mean corresponding to the number being represented. This is called Stochastic stream. There are different methods of achieving stochastic representation:

1. Analog representation - Where analog values are used in the Stochastic stream

2. Bipolar representation - where the values $+1$ and -1 are used to represent the Stochastic stream

3. Unipolar representation - where the values 0 and 1 are used to represent the Stochastic stream

This document uses unipolar representation throughout since this is more compatible with the current ecosystem of technologies and the data format used is the same. Such a Stochastic stream is known as Stochastic bit stream since it consists of binary bits. Using this representation can be beneficial as outlined in some of the basic concepts of stochastic computing below.

1.1.2. Basic Stochastic Circuits. To convert any number into a stochastic bit stream, a comparator and a Random Number Generator (RNG) can be used as shown in Figure 1.1. It is to be noted that the input number and the RNG output need to be on the same scale for the conversion to be appropriate. The RNG can also be reused to represent multiple inputs in a stochastic system; this results in correlation between the stochastic bit streams which can be detrimental to the operation of certain stochastic circuits but it can also be exploited for benefits as detailed in [2].

Using a stochastic bit stream, it is possible to use the probabilistic nature of logic gates to perform arithmetic operations. Consider a simple AND logical gate as shown below in Figure 1.2. The truth table for the AND gate is given in Table 1.1. If the inputs

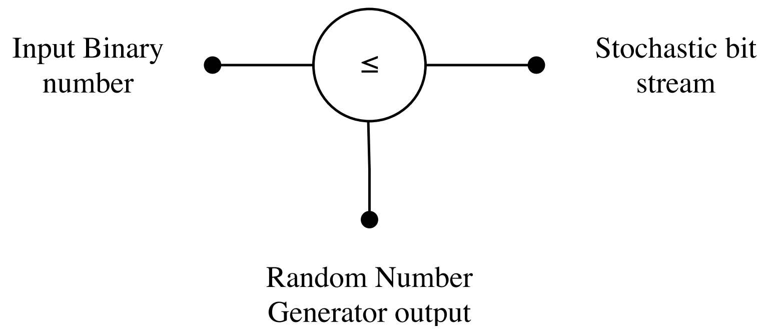


Figure 1.1. Bit stream generation using comparator and RNG

are considered as streams of bits, It can be observed from this truth table that the inputs are of probability 0.5 each (denoting that the probability of finding bit '1'in the stream is 0.5). Similarly, the output stream has a probability of 0.25. Here the input probabilities seem to have been multiplied.

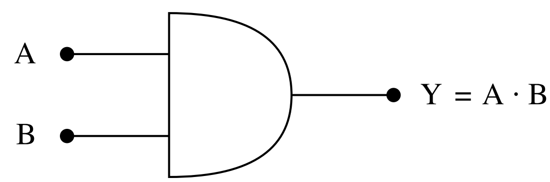
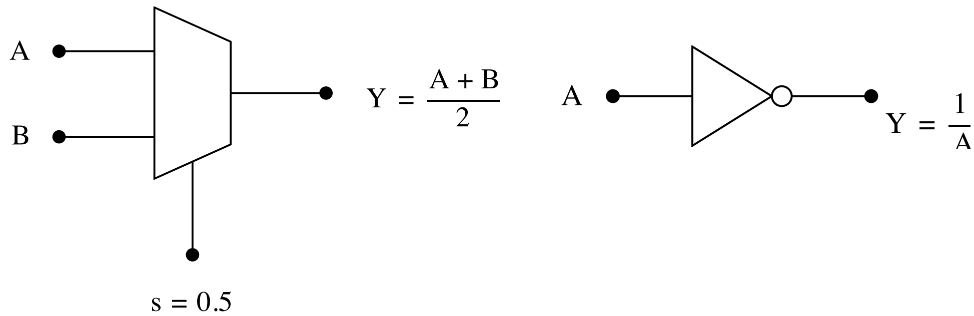


Figure 1.2. AND logical gate

Table 1.1. AND logical truth table

A	B	$Y = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

And as long as the logic behind the gate is the same, the same effect can be observed for input streams of any probability at the input. Similarly, other logic gates can be used to perform different operations as outlined in [3]. Some of the circuits are shown below in Figure 1.3 along with their equivalent stochastic operation. Some of the more complex mathematical operations are outlined in [8].



(a) Scaled adder using 2 × 1 multiplexer (b) Inverter using NOT logical gate

Figure 1.3. Basic stochastic circuits

To consume the result of these stochastic circuits, the stochastic bit stream can be converted back using a counter to calculate the probability of 1 in the output bit stream.

1.2. EXISTING RESEARCH AND APPLICATIONS OF STOCHASTIC COMPUTING

Stochastic circuits are most viable for applications with inherent fault tolerance. A few examples are Image processing, Signal processing, Neural networks among others. These applications can be considered sufficiently fault tolerant since error percentage or peak signal to noise ratio (PSNR) seem to not affect the intended outcome drastically from the perspective of human observation.

As an example, consider the process of edge detection from image processing domain. Edge detection refers to the process of highlighting intensity variations in an image while suppressing constant intensity areas. There are multiple edge detection mechanisms available [6]. One of the simpler yet efficient edge detection mechanism is Robert-Cross edge detection. This algorithm works on a 2×2 pixel neighborhood to detect edges in an intensity image. The following images in Figure 1.4 show the result of Robert-Cross edge detection algorithm using stochastic computing with various bit stream duration/length.

(a) Original input image

(b) 8 bit stochastic length

(c) 64 bit stochastic length

(d) 128 bit stochastic length

(e) 256 bit stochastic length

Figure 1.4. Comparing edge detection outputs for different bit stream lengths

Figure 1.4 shows that although statistically insignificant lengths were chosen for stochastic bit streams, the output is not affected drastically. Further, [4] presents more information on stochastic computing applied to image processing techniques.

1.2.1. Approximate Computing. Figure 1.4 also shows that stochastic computing is an approximate method of computation. The result gets closer to the intended original value as the length of stochastic bit stream increases; just as the expected value (mean) of a sample set reaches the true mean as the sample size increases. To achieve better efficiency along with allowing error to increase slightly, lower stochastic bit lengths can be favored for faster execution. Another way of doing the same would be to intentionally truncate lower significant bits from the input binary numbers. This is known as Approximate Stochastic Computing (ASC) [10].

1.3. APPROXIMATE STOCHASTIC COMPUTING

In stochastic computing, as a rule of thumb, 2^n bits are used in the stochastic domain to represent an n bit binary number. However, evidently from Figure 1.4, it may not be necessary. This involves analysis of lowest energy point as described in [1]. Applying approximate computing at the input level by discarding bits of lower significance form the input will further reduce the number of stochastic bits required since the number of bits required to represent the input is now lowered. If m bits are truncated from the input numbers, the required number of bits in the stochastic stream would approximately be 2^{n-m}. A complete analysis of this concept is presented in [10]. Here, the author uses the values of $n = 4$ and $m = 4$ for implementation.

Additionally, to increase the efficiency especially for the Robert-Cross edge detection implementation, an adaptive mechanism has been discussed in [11]. This adaptive mechanism suggests adding an extra count of 1 to the output bit stream while converting back to the binary domain. This extra count is conditional and is only applied when the majority of the inputs have the bit 1 in their 5$^{\text{th}}$ most significant bit position. The current document aims to propose an improvement over the concept of ASC.

1.4. ORGANIZATION OF THE CURRENT DOCUMENT

The following sections present a method proposed as an analysis and improvement over ASC. Section 2 shows a mathematical model for error involved in a stochastic process and uses the model to analyze ASC and design goals for the proposed method. Section 3 discusses the proposed method in detail and provides a few ways of implementation. Section 4 shows the performance of the proposed method using the Robert-Cross edge detection application and compares the proposed design to ASC and adaptive ASC. Section 5 provides a summary of the current document. Appendices A and B provide implementation code and supporting information for simulations used for analysis.

2. MODELLING STOCHASTIC ERROR

2.1. ANALYSIS

The nature of stochastic computing is to embrace error in randomness to simplify operation. Analysis of stochastic error due to randomness is necessary to find the minimum energy point of operation in stochastic circuits [1]. To find the most optimum stochastic bit length, s, for use in any stochastic system design, probabilistic error in representation of a binary number in stochastic domain needs to be examined. To perform this analysis, Matlab scripts were employed as an experiment to evaluate the absolute error and percentage error in this process. The Matlab script used here is given in Appendix A.

2.2. IMPLEMENTATION

To evaluate stochastic process error, consider an experiment where an n bit number, N, is converted into a stochastic bit stream of length s, and then the original number N is approximated as $\hat{N}$, using this bit stream. This experiment when performed on a single value of N with a specific bit stream length allows us to calculate the process error involved by computing the difference between N and $\hat{N}$. This error is representative of stochastic process error involved for the chosen values of N, n and s. To generalize this error, this experiment was performed $L = 5 \times 10^6$ times, for statistical significance, with random values of N, and all values of n and s within the bounds as specified, and then the results were summarized. The sample set of values chosen for n was (4, 6, 8, 10, 12, 14, 16). Accordingly, the value of N was uniformly varied between 0 and $2^n - 1$. The value of s for each N was varied between 1 and $2^n - 1$.

To compute the error in this process, equations (2.1) and (2.2) were used, where $\epsilon_{sc,a}$ is the absolute error and $\epsilon_{sc,p}$ is the percentage error involved.

$$\epsilon_{sc,a} = \left|N - \hat{N}\right| \tag{2.1}$$

$$\epsilon_{sc,p} = \frac{\epsilon_{sc,a}}{N} \times 100 \tag{2.2}$$

To aggregate the error for a uniformly distributed input, average error ϵ_{sc} is considered, given by (2.3)

$$\epsilon_{sc} = \frac{1}{L}\sum_{i=0}^{L} \epsilon_{sc,a}[i] \tag{2.3}$$

This experiment yields a plot as shown in Figure 2.1. The plot shows how ϵ_{sc} varies as s varies. Also, Figure 2.2 shows how $\epsilon_{sc,p}$ varies as s varies.

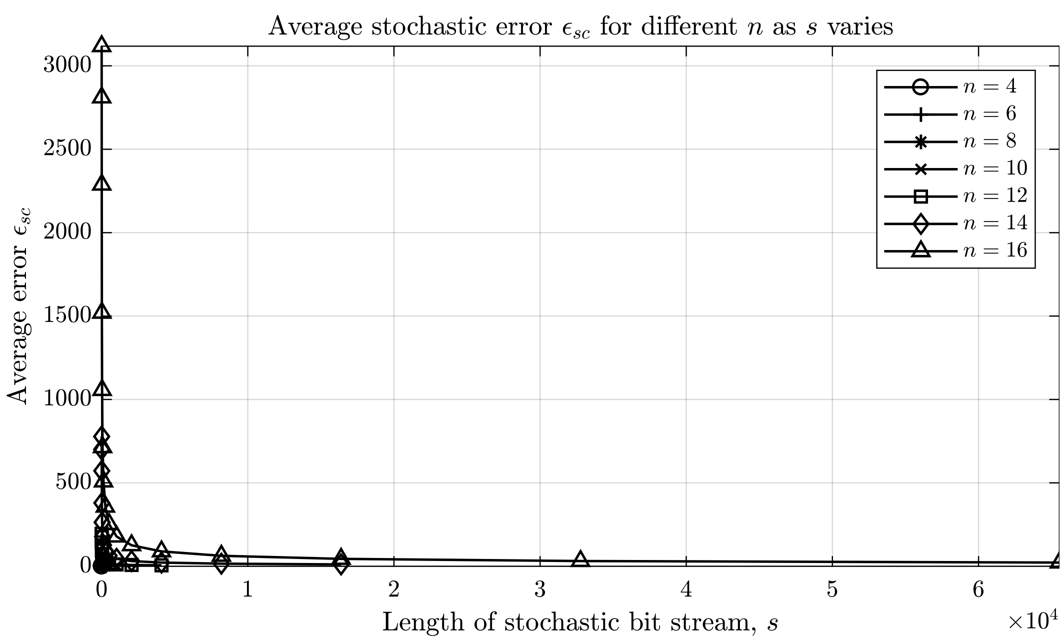


Figure 2.1. Variation in absolute error $\epsilon_{sc,a}$ for different n, as s varies

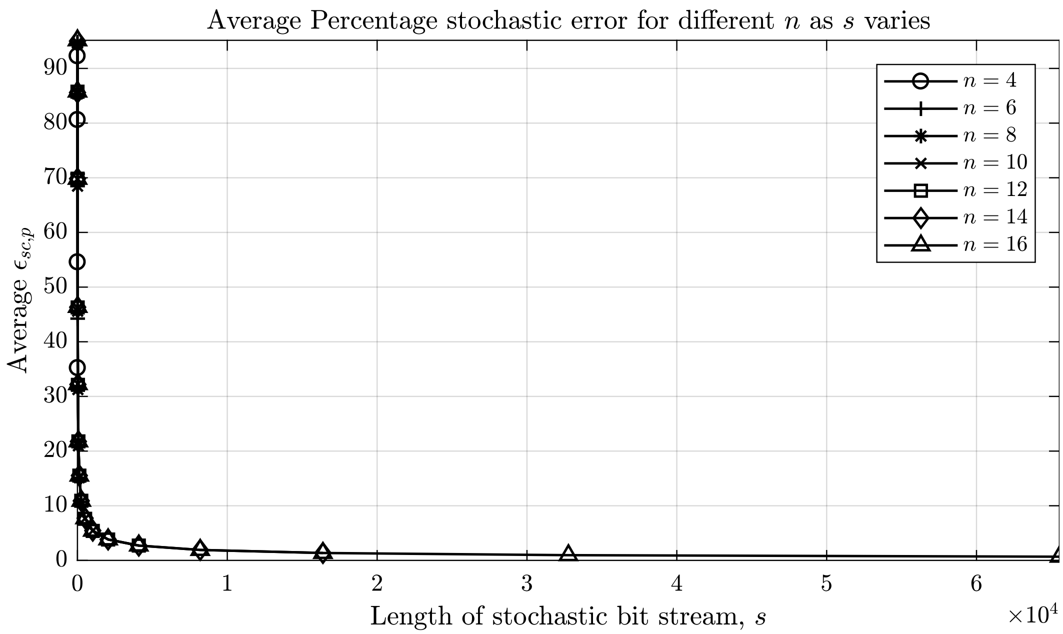


Figure 2.2. Variation in percentage error $\epsilon_{sc,p}$ for different n, as s varies

Figure 2.1 does not reveal much information in the state presented and zooming in reveals that the data for any value of n is in a much different range than the data for other values of n. This is shown in Figure 2.3.

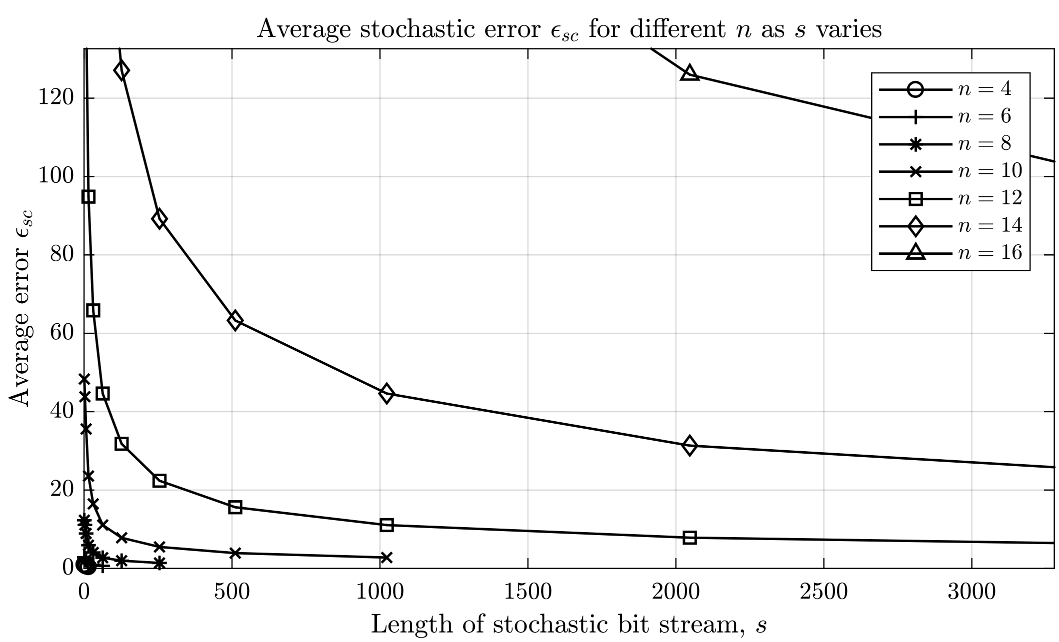


Figure 2.3. Zoomed in view of the variation in absolute error $\epsilon_{sc,a}$ for different n, as s varies

This also suggests that for fair comparison the horizontal axis needs to be normalized since for different n, the range of possible numbers varies exponentially. Normalizing the horizontal axis, the plot takes the form as shown in Figure 2.4

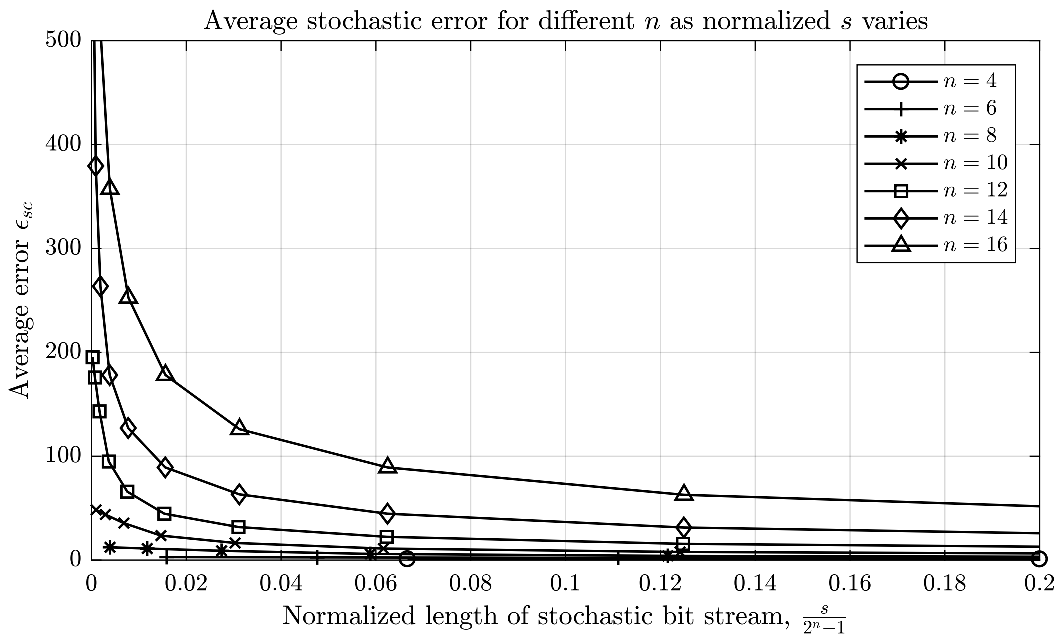


Figure 2.4. Variation in percentage error $\epsilon_{sc,p}$ for different n, as normalized s varies

The curves represented in Figure 2.4 resembles an exponential decay function as given by $y = ae^{-bx}$. Hence plotting this in a natural logarithmic domain reveals more information than what is visible in above plots. Therefore a plot for ϵ_{sc} was generated with normalized axis variables x and y, as given in equations (2.4) and (2.5) respectively.

$$x = \ln\left(\frac{s}{2^n - 1}\right) \tag{2.4}$$

$$y = \ln\left(\frac{\epsilon_{sc}}{2^{\frac{n}{2}}}\right) \tag{2.5}$$

It can be noted that the equation (2.5) incorporates a scaling factor of $2^{\frac{n}{2}}$ before the logarithm operation. This scaling factor was experimentally determined and also corresponds to the average of a uniform distribution in the range $(0, 2^n)$. This is appropriate since the error being shown is the average error of applying stochastic process to any n bit number chosen randomly using a uniform distribution of the same range. The plot of y versus x is shown in Figure 2.5.

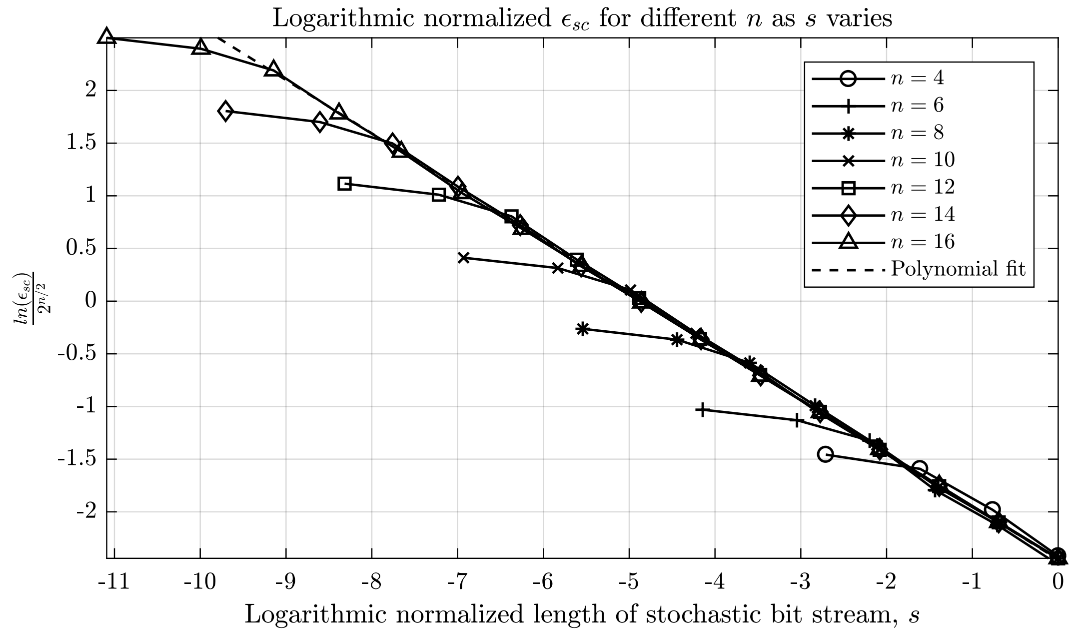


Figure 2.5. Logarithmic plot of ϵ_{sc}

Figure 2.5 shows that the normalized average error can be modelled as a straight line in logarithmic domain.

2.3. MATHEMATICAL MODEL

Observing the Figure 2.5, it can be estimated that the equation of the curve is of the form shown in equation (2.6)

$$y = mx + c \tag{2.6}$$

This estimation can be realized using polynomial fitting function, `polyfit()`, in Matlab. Matlab code for this estimation is presented in Appendix B.

The first two entries for each n are discarded since they evidently do not contribute to the polynomial fit. This polynomial fitting results in the equation (2.7).

$$y = -0.5057x - 2.4523 \tag{2.7}$$

The coefficients -0.5057 and -2.4523 are approximated to -0.5 and -2 respectively to simplify the equation. To justify this approximation, consider the value of ϵ_{sc} at $x = 0$ from equations (2.7) and (2.5), which evaluates to 4.3775×10^{-223}. The same evaluation yields 1.2020×10^{-268} for the approximated co-efficient values of -0.5 and -2. The values themselves and also their difference is very low since they represent average absolute error in stochastic process. Hence, equation (2.7) can be approximated as equation (2.8).

$$y = -0.5x - 2 \tag{2.8}$$

Substituting (2.4) and (2.5) in (2.8),

$$\begin{aligned}
\ln\left(\frac{\epsilon_{sc}}{2^{\frac{n}{2}}}\right) &= -0.5 \times \ln\left(\frac{s}{2^n - 1}\right) - 2 \\
&= -\left[\ln\left(\sqrt{\frac{s}{2^n - 1}}\right) + 2 \times \ln(e)\right] \\
&= -\left[\ln\left(\sqrt{\frac{s}{2^n - 1}}\right) + \ln(e^2)\right] \\
&= -\left[\ln\left(\sqrt{\frac{s}{2^n - 1}} \times (e^2)\right)\right]
\end{aligned}$$

$$= \ln\left(\frac{1}{e^2}\sqrt{\frac{2^n - 1}{s}}\right)$$

$$\Rightarrow \frac{\epsilon_{sc}}{2^{\frac{n}{2}}} = \frac{1}{e^2}\sqrt{\frac{2^n - 1}{s}}$$

which gives

$$\epsilon_{sc} = \frac{1}{e^2}\sqrt{\frac{2^n\left(2^n - 1\right)}{s}} \tag{2.9}$$

Equation (2.9) quantifies the error built into the stochastic process being employed. Furthermore, as [10, 11] describe, additionally truncating bits from the given input operands is time efficient and approximates the output close to the actual value but then introduces an error. Mathematically this truncation error for any number N while truncating m bits is given by (2.10).

$$\epsilon_t = N \ mod \ 2^m \tag{2.10}$$

Hence, on an average, the combined error for any number from 0 through $2^n - 1$ in total can be shown in (2.11).

$$\epsilon = \frac{1}{e^2}\sqrt{\frac{2^k\left(2^k - 1\right)}{s}} + N \ mod \ 2^m \tag{2.11}$$

where k is $n - m$, the number of bits chosen to be *kept* after the truncation.

2.4. INFERENCE

In [10], for a fixed truncation with n = 8 and m = 4, where the least four significant bits are truncated, the truncation error is in the range (0, 15). Hence, the average truncation error is the mean of a uniform distribution in this range, which is ≈ 7.5. Additionally, the average stochastic error for [10] where the remaining k bits after truncation are represented using $2^4 = 16$ clock cycles, i.e $k = 4$ and $s = 16$, is $\epsilon_{sc} = 0.5$ using equation (2.9). Thus the combined error for the process described in [10] is $\epsilon \approx 7.5 + 0.5 = 8$.

For the same value of $n = 8$ if the stochastic bit length s is capped at 16 instead of $2^n - 1 = 255$, with no intentional loss of any information due to truncation (i.e, truncation error = 0), equation (2.11) approximates the total process error to 8.6445. This is fairly comparable to the value obtained above for [10], while still being equally time efficient (i.e, $s = 16$).

This also shows that the equation (2.11) can be used as a tool to estimate the total approximate process error due to factors such as input bit length (n), stochastic bit length (s), number of bits truncated (m), and/or to estimate the optimal stochastic bit length when a predefined average process error (ϵ) can be tolerated.

2.5. CONSTRAINTS FOR DESIGN

With the above analysis, any method that may be proposed to improve upon [10] needs to satisfy the following constraints to be more generic and adaptive.

1. *Avoid deterministic components* in procedure to preserve the random nature of Stochastic Computing.

2. *Adaptive truncation based on input number* since all numerical values are not affected by optimization mechanism the same way.

3. *Allow variable input bit length* to remain applicable for any application that may use different bit length *n*.

4. *Allow pre-calculation of tolerable error* in stochastic process so that the procedure incorporates determining the error margin in design phase for any implementation.

5. *Flexibility to choose truncation length* and stochastic bit length based on tolerable error.

6. *Improve overall accuracy* while optimizing area and/or execution time.

Following the above mentioned constraints as closely as possible, a generalized adaptive truncation methodology has been proposed in the current document.

3. PROPOSED DESIGN

3.1. INFORMATION IN BINARY FORMAT

Consider numbers stored in binary format (integers for simplicity). Information stored in such a format lies in between the most significant high bit and the least significant high bit in any number's representation. This can be represented as shown in Figure 3.1.

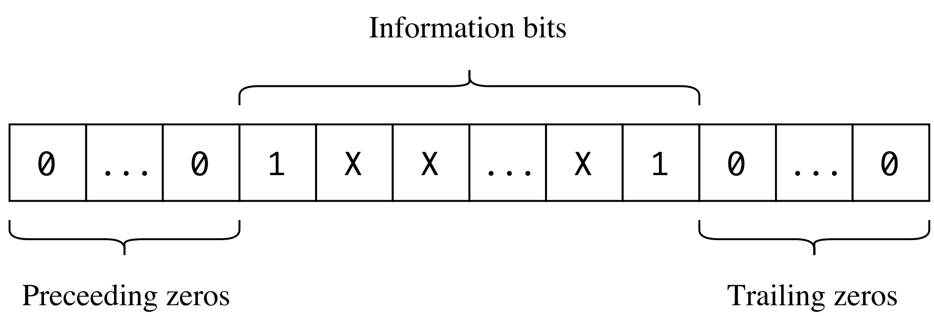


Figure 3.1. Information in Binary format

It is imperative to perform mathematical operations on information stored in above format by using the bits shown in the information range. This also means that truncating certain bits based on position, in any number, may result in lost information. The proposed method aims to perform any operation in the information range of its operands instead of a fixed position range. Consider the following examples for $n = 8$ bit operands.

$$\begin{array}{rl} A = & \texttt{000000} \\[6pt] B = & \texttt{000000} \\[6pt] \dfrac{A+B}{2} = & \texttt{000000} \end{array} \left[\begin{array}{c} \texttt{01} \\[6pt] \texttt{11} \\[6pt] \texttt{01} \end{array} \right] \begin{array}{l} = (1)_{base\ 10} \\[6pt] = (3)_{base\ 10} \\[6pt] = (1)_{base\ 10} \end{array} \tag{3.1}$$

$$
\begin{array}{rl}
A = & \mathtt{00} \\
B = & \mathtt{00} \\
\dfrac{A+B}{2} = & \mathtt{00}
\end{array}
\left[\begin{array}{c}
\mathtt{0101} \\
\mathtt{1110} \\
\mathtt{1001}
\end{array}\right]
\begin{array}{ll}
\mathtt{00} & = (20)_{base\ 10} \\
\mathtt{00} & = (56)_{base\ 10} \\
\mathtt{10} & = (38)_{base\ 10}
\end{array}
\tag{3.2}
$$

The example shown in (3.1) shows that the operation is dominant in the range of two lowest significant bits (highlighted with brackets). Similarly the example in (3.2) shows that the operation is dominant in the range of 6^{th} lowest significant bit to 3^{rd} lowest significant bit. The dominant information among the operands takes precedence, which means the information bit range varies based on the operands for a given operation.

3.2. COMPUTING USING INFORMATION BITS

The generic process of working with specific information bit ranges in any given set of operands could be outlined as follows

1. Extract information bits from all operands
2. Perform the operation on extracted information bits
3. Modify the result to match original scale of the operands

Although there may be various methods to achieve the above steps, two methods are proposed below. Both the methods are designed to be efficient in different ways while realizing the same process above.

3.3. METHOD 1: AREA EFFICIENT

The current section provides the algorithm and an example to help explain the area efficient implementation of proposed method. It is followed by the block diagram and analysis on the proposed implementation.

3.3.1. Algorithm. This method is designed to keep area of the implementation fairly low by trading off number of clock cycles. This method involves shifting each operand to the left until at least one of the operands has a high bit in the most significant position, and then performing the operation on as as many bits as necessary. The following steps show the algorithm.

1. Left shift all operands until there is at least one high bit at most significant position among the inputs; say η number of shifts
2. Consider first k MSBs of the resulting set; k is number of bits kept after truncation
3. Perform the required operation
4. Store the result as first k MSBs of an n bit number
5. Right shift the result as many times as the inputs were left shifted in first step, i.e., by the amount of η

3.3.2. Example. To illustrate this, consider the following example. The operation to be performed is $\frac{A+B+C+D}{4}$ as given in equation (3.3) below, and the operands are $n = 8$ bit integers.

$$\begin{aligned} A &= \mathtt{00} \\ B &= \mathtt{00} \\ C &= \mathtt{00} \\ D &= \mathtt{00} \\ \frac{A+B+C+D}{4} &= \mathtt{00} \end{aligned} \begin{bmatrix} \mathtt{00\ \ 1011} \\ \mathtt{10\ \ 0101} \\ \mathtt{11\ \ 1000} \\ \mathtt{01\ \ 0101} \\ \mathtt{01\ \ 1111} \end{bmatrix} \tag{3.3}$$

In the above equation, it can be observed that the required operation has to be performed only in the bit range from position 1 to position 6, right to left.

Step 1 - According to the algorithm, the operands have to be left shifted by a factor of $\eta = 2$. Thus the operands can be rewritten as shown in equation (3.4).

$$
\begin{aligned}
A' &= \mathtt{0010\ 1100} \\
B' &= \mathtt{1001\ 0100} \\
C' &= \mathtt{1110\ 0000} \\
D' &= \mathtt{0111\ 1100}
\end{aligned}
\tag{3.4}
$$

Step 2 - Now these operands can be truncated by keeping k most significant bits in each operand. Consider the case where $k = 3$. The operands will now be as shown in (3.5).

$$
\begin{aligned}
A_k &= \mathtt{001} \\
B_k &= \mathtt{100} \\
C_k &= \mathtt{111} \\
D_k &= \mathtt{011}
\end{aligned}
\tag{3.5}
$$

Step 3 - Operating on these operands, the operation $\frac{A+B+C+D}{4}$ results in a value around $O_k = \mathtt{111}$ when constrained to $k = 3$ bits.

Step 4 - Storing the above output as first $k = 3$ MSBs of an $n = 8$ bit number gives the output $O' = \mathtt{11100000}$

Step 5 - This intermediate result has to be shifted to the right by a value of $\eta = 2$ to obtain the intended output $O_\eta = \mathtt{00011000}$

The output calculated can be represented in decimal as the number 24 and the original result shown in (3.3) can be represented in decimal as 31. This difference is due to the bits truncated in step 2.

3.3.3. Block Diagram. Figure 3.2 shows the block diagram for Method 1. Here, all the steps before performing the operation are represented as *Pre-condiddtioning* steps, since the input operands are *conditioned* to be sent to the operation block. Similarly, all the steps performed after the operation are referred to as *Post-conditioning* steps since these steps *condition* the intermediate result of the operation, O_k, to get an estimation of the desired output, O_η.

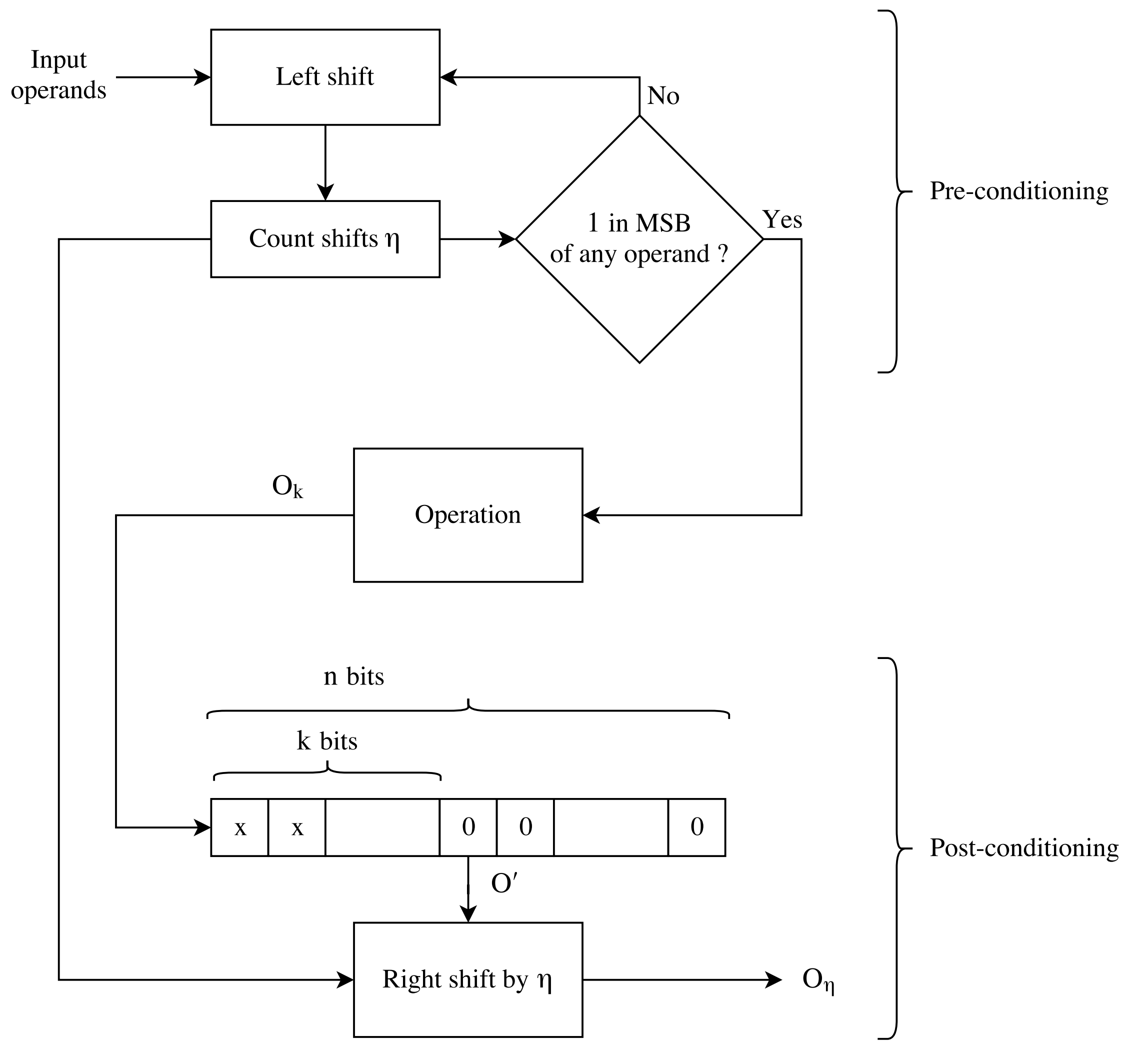


Figure 3.2. Block diagram of Method 1

3.3.4. Analysis. This method allows for a simple algorithm to eliminate truncation error in lower values of input, compared to the fixed truncation method [10]. However, it adds significant clock cycle overhead before execution of the operation. This overhead is not fixed and gets more significant as the input values get lower. Exact implementation of the method may vary the actual amount of overhead but the fact remains that there is more sequential procedure in the above method than parallel procedure.

To mitigate some of the overhead issue, another method is proposed where parallel execution is favored.

3.4. METHOD 2: TIME EFFICIENT

For the time efficient method, the concept of Information Look-up Table is introduced so that converting operands to information bits and then adjusting the output to the right scale is done in parallel in this process.

3.4.1. Condensing Information: Information Look-up Table (ILUT). To be able to represent all the information in a number, the following method proposes usage of specific details of the inputs as listed below. This allows for mathematical operations to be performed on different set of bit positions for the given inputs rather than fixed set of bit positions for any set of inputs.

1. I - Information between the most and least significant 1s in the number
2. α - The number of trailing zeros after the least significant 1
3. β - The number of bits in I

The above parameters were chosen based on the two following factors -

1. Ability to recreate the original number from the condensed information.
2. Ability to variably shift or scale a given number for any operation based on other operands required in that operation.

The above process produces the following look up table for $n = 4$ bit numbers, for example as given in Table 3.1.

Table 3.1. ILUT for an n = 4 bit number

Number	I	α	β
0000	0	0	0
0001	1	0	1
0010	1	1	1
0011	11	0	2
0100	1	2	1
0101	101	0	3
0110	11	1	2
0111	111	0	3
1000	1	3	1
1001	1001	0	4
1010	101	1	3
1011	1011	0	4
1100	11	2	2
1101	1101	0	4
1110	111	1	3
1111	1111	0	4

3.4.2. Operating on Information Bits: Usage of the ILUT. Since the information range of any operation depends on the operands, the current method requires preconditioning the operands to be used in the operation. Consider an example below in (3.6).

$$\begin{aligned} A &= \mathtt{0}\,\left[\begin{matrix}\mathtt{01}\\[1em] \mathtt{10}\\[1em] \mathtt{01}\end{matrix}\right]\begin{matrix}\mathtt{0}\\[1em] \mathtt{0}\\[1em] \mathtt{1}\end{matrix} \\ B &= \mathtt{0} \\ \frac{A+B}{2} &= \mathtt{0} \end{aligned} \tag{3.6}$$

To achieve the result given, consider the values of I and α for the operands from Table 3.1. The values are as given in Table 3.2 below.

Table 3.2. ILUT for n = 4 bit operands in example

Number	I	α	β
0100	1	1	1
1000	1	2	1

For the current operation, note that the operation is most dominant in the range of 3rd least significant bit to the 2nd least significant bit. To use the information given in Table 3.2 to perform the operation in given range, it can be observed that the operation starts at an index equal to lowest α, denoted by α_{min}, and ends at an index equal to highest $\alpha + \beta$, denoted by μ_{max}.

We can calculate the amount of shift, $\hat{\alpha}$, to be applied to each I as $\alpha - \alpha_{min}$ if α is represented as a matrix as shown in the equation (3.7)

$$\hat{\alpha} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} - \alpha_{min} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.7}$$

applying the corresponding calculated shifts to each I, we obtain I' as shown in (3.8). These values can now be used to perform the operation.

$$I' = \begin{bmatrix} \mathtt{01} \\ \mathtt{10} \end{bmatrix} \tag{3.8}$$

The result of the operation for I' will now be $\frac{A+B}{2} = [\mathtt{01}]$. This result needs to be shifted appropriately by a factor of $\alpha_{min} = 1$ to arrive at the final output as shown in equation (3.9)

$$\frac{A + B}{2} = \mathtt{0010} \tag{3.9}$$

3.4.3. Algorithm. The current method involves creating an ILUT for a specific n value chosen for any operation. The example above can be further extended to have a k bit constraint on the length of bits in I. In which case, the lower bound of the operation is not α_{min} as given in the example above. The lower bound now is $max\,(\alpha + \beta - k)$ among the operands. This means that the ILUT can directly store α and $\mu = \alpha + \beta - k$ for each input instead of just α and β. Information from this ILUT can then be used to perform the operation as outlined in the algorithm below.

1. Fetch information bits, I, number of trailing zeros, α, and μ for each of the operands
2. Conditionally shift each I by a factor of $\alpha - \mu_{max}$, giving I', where μ_{max} is the maximum value for μ among the operands
3. Perform the operation on I'
4. Shift the result back by a factor of μ_{max}

3.4.4. Example. Consider the same example as given in section 3.3.2 of Method 1, reproduced in equation (3.10), where the operation to be performed is $O = \frac{A+B+C+D}{4}$.

$$\begin{aligned} A &= \mathtt{00}\left[\begin{array}{l}\mathtt{00\ \ 1011} \\ \mathtt{10\ \ 0101} \\ \mathtt{11\ \ 1000} \\ \mathtt{01\ \ 0101} \\ \mathtt{01\ \ 1111}\end{array}\right] \\ B &= \mathtt{00} \\ C &= \mathtt{00} \\ D &= \mathtt{00} \\ \frac{A+B+C+D}{4} &= \mathtt{00} \end{aligned} \tag{3.10}$$

The operation is still significant only in the bounds of bit positions 1 through 6, right to left. The following steps illustrate the algorithm in detail.

Step 1 - Fetch I, α and μ. For this set of operands the ILUT can be given as shown in Table 3.3 for $k = 3$.

Table 3.3. ILUT for $n = 8$ bit operands, keeping $k = 3$ bits in I for current example

Number	I	α	$\mu = \alpha + \beta - k$
0000 1011	101	0	2
0010 0101	100	0	3
0011 1000	111	3	3
0001 0101	101	0	2

Step 2 - Conditionally shift the operands by a factor of $\alpha - \mu_{max}$. Conditional shift means to shift the I bits to the left if the value of $\alpha - \mu_{max}$ is positive, and shift it to the right when negative. The operands are now modified as shown in equation (3.11)

$$I'_A = \mathtt{001}$$
$$I'_B = \mathtt{100}$$
$$I'_C = \mathtt{111} \qquad (3.11)$$
$$I'_D = \mathtt{010}$$

Step 3 - Perform the operation on I' values. The operation will result in a number around $O' = \mathtt{011}$.

Step 4 - Shift by a factor of μ_{max}. This will result in the output $O_\mu = \mathtt{00011000}$, which is the same as the output obtained by Method 1.

3.4.5. Block Diagram. The Figure 3.3 shows the block diagram for this method. This is comparable to the block diagram of Method 1 from Figure 3.2. It can be observed that the sequential Pre-conditioning is replaced with a parallel procedure in Method 2.

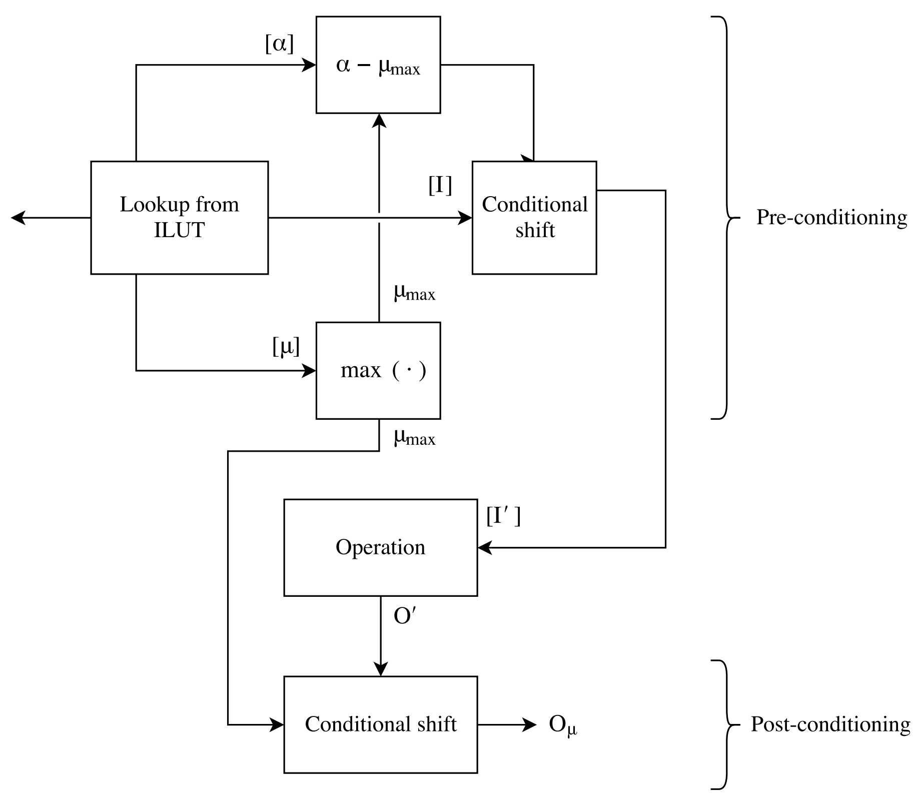


Figure 3.3. Block diagram of Method 2

3.4.6. Analysis. This method proposes an algorithm while fetching information bits in parallel and executing the operation directly on the information bits results in the desired output in a more time efficient manner. However this results in larger area since complete ILUT storage is necessary. Additionally, the size of ILUT increases exponentially as n increases linearly, thus requiring additional space for memory.

3.5. COMPARISON: METHOD 1 VS METHOD 2

The Table 3.4 gives a brief summary and comparison between the methods proposed above. This comparison can help choose between the methods for any particular implementation as necessary.

Table 3.4. Comparison of Method 1 and Method 2

Method 1	**Method 2**
Area efficient	Time efficient
No need of ILUT, thus reducing area	ILUT increases exponentially as n increases linearly, thus increasing area required
Number of clock cycles vary based on the input operands	Same number of clock cycles for any input with fixed n

3.6. COMPARISON WITH ASC

The Table 3.5 shows how the proposed method compares with ASC and/or Adaptive ASC [10, 11].

Table 3.5. Comparison of proposed method and ASC

Proposed method	**ASC/Adaptive ASC**
Better performance for operations involving smaller numbers. In image processing, this means better performance in low intensity images/areas	Higher percentage error in lower numerical ranges
Number of kept bits, k, can be calculated (hence, also truncation length m) using ϵ_{sc} from equation (2.9)	Fixed truncation length $m = 4$ for $n = 8$ bit numbers
Number of clock cycles can also be a choice based on equation (2.9)	Fixed number of clock cycles: $s = 16$ in ASC and $s = 16$ or $s = 17$ for adaptive ASC
Can be extended to different ranges of n	Specific for $n = 8$ bit operations

4. PERFORMANCE

To evaluate the proposed method in comparison with [10], a Robert Cross filter was implemented Matlab to simulate this process in order to benchmark its performance. The implementation uses the values of $n = 8$ and $m = 4$ (truncated bits) resulting in $k = 4$ for fairness in comparison with approximate stochastic processing as defined in [10].

4.1. ROBERT CROSS EDGE DETECTION

Robert Cross edge detection is an algorithm applied to gray scale images over a neighborhood of 2×2 pixels highlighting the changes in intensities which results in an image that highlights only the edges in the input image. The approximate equation for this process is given in (4.1), where X is the input image, Y is the output image, (i, j) are pixel indices in vertical and horizontal direction respectively.

$$Y(i, j) = 0.5 \times (|X(i+1, j+1) - X(i, j)| + |X(i, j+1) - X(i+1, j)|) \tag{4.1}$$

4.1.1. Robert Cross Algorithm in Stochastic Computing. The stochastic equivalent for the operation given in (4.1) is as shown Figure 4.1.

For this implementation, correlated random numbers are assumed at all four inputs when converting the inputs from Binary to Stochastic bit streams. This means that all the four inputs are converted to stochastic bit streams using the same random number at their comparators. The block diagram for Adaptive Variable-bit truncation ASC as proposed in [11] is given in Figure 4.2.

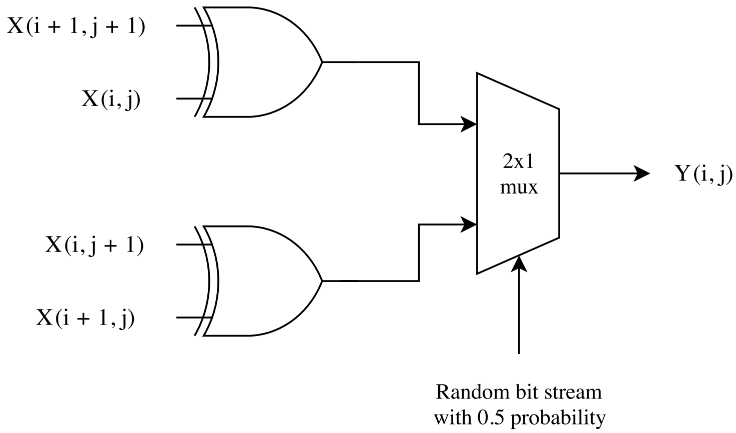


Figure 4.1. Robert Cross algorithm in stochastic domain

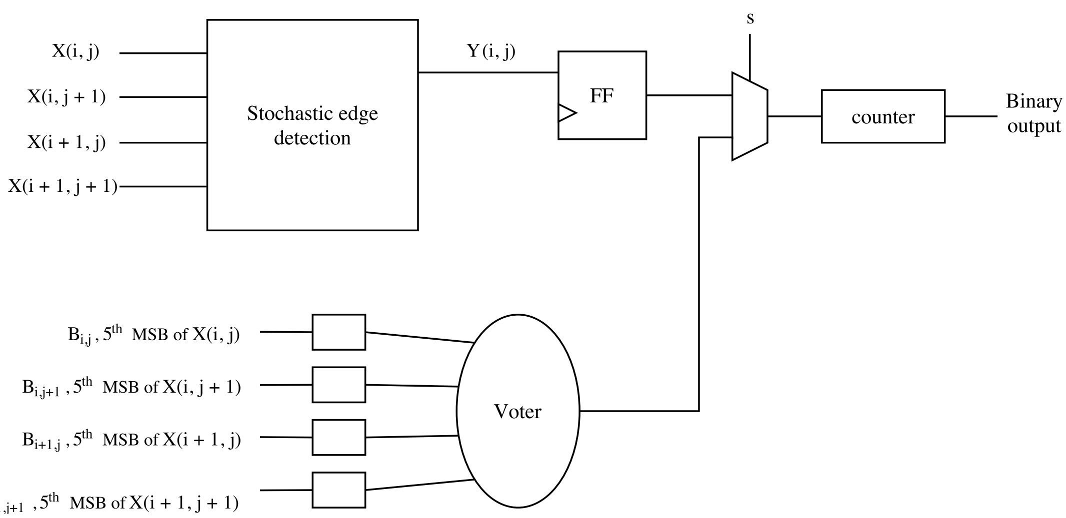


Figure 4.2. Block diagram of Robert Cross algorithm using proposed method from [11]

4.2. RESULTS

In order to better analyze the performance of the proposed design as compared with previous research, the same metrics as used in [10, 11] are used in the current document. They are explained as follows.

4.2.1. Result Metrics. Two metrics were employed to measure the performance of the proposed design -

1. Mean Squared Error (MSE) - This metric is chosen to show how different the output of proposed method (or any method being compared) is, as compared to the theoretical output of Robert-Cross algorithm.

2. Peak Signal-to-Noise Ratio (PSNR) - This metric depicts the amount of noise present in the output as compared to the signal, due to MSE.

These metrics are calculated using the equations shown in equations (4.2) and (4.3) below.

$$MSE = \sum_{i=0}^{r} \sum_{j=0}^{c} |O(i,j) - O_t(i,j)|^2 \tag{4.2}$$

$$PSNR = 10 \times log_{10} \left(\frac{255^2}{MSE} \right) \tag{4.3}$$

Where r is the number of rows in the input image, c is the number of columns in the input image, O is the output being compared, and O_t is the theoretical Robert-Cross edge detection output. The number 255 in (4.3) refers to the peak possible signal for an 8 bit image.

The following figures in the next section contain sub-figures labelled (a) through (d) alphabetically, which are structured in this manner -

(a) Original image used as the input for the edge detection algorithm

(b) Output obtained using Binary operation as given in Robert-Cross equation 4.1

(c) Output obtained using the proposed method.

(d) Output obtained using ASC with 16 clock cycles (truncation only)

(e) Output obtained using ASC with 17 clock cycles (truncation and compensation with extra clock cycle)

The Table 4.1 shows a metric comparison of results for proposed method, and the methods from [10] and [11]. Input images for these results are presented in detail below. It can be observed that the proposed method almost always has lower MSE and higher PSNR compared to [11].

Table 4.1. Result table

Section	Picture	MSE			PSNR (dB)		
		New design	Old design		New design	Old design	
			16 clock cycles	17 clock cycles		16 clock cycles	17 clock cycles
Initial	Cameraman	147.8104	183.7811	267.2218	26.4	25.4	23.8
	Lena	255.4039	331.9134	358.7511	24.0585	22.9206	22.5829
	Generated	23.1125	28.8353	164.9034	34.4	33.5	25.9585
Intensity	Low	39.6976	134.8726	204.7925	32.1432	26.8316	25.0177
	Medium	48.8106	156.4889	222.9728	31.2457	26.186	24.6483
	High	0.9016	6.828	127.6069	48.5806	39.7878	127.6069
Misc	Banana	173.332	240.05	350.87	25.742	24.3276	22.6793
	Horizontal stripes	141.134	191.07	266	26.6345	25.3187	23.8819
	Skull MRI	406.6909	506.7403	453.6058	22.0382	21.0829	21.564

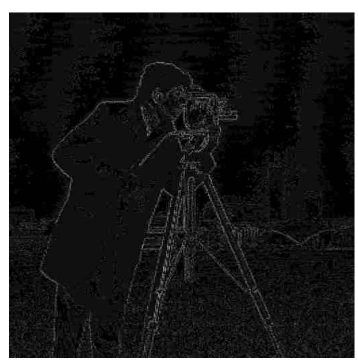
4.2.2. Standard Image Result: Cameraman. Figure 4.3 is a standard test image used in image processing applications to benchmark any image processing algorithm. It can be observed in Figure 4.3 that output in Sub-Figure 4.3c contains visibly lesser noise in low intensity areas such as the coat and the sky, when compared to both sub-figures 4.3d and 4.3e. This is also evident in the MSE and PSNR values from Table 4.1.

(a) Original input image

(b) Theoretical output

(c) Output using proposed method

(d) ASC output with 16 cycles

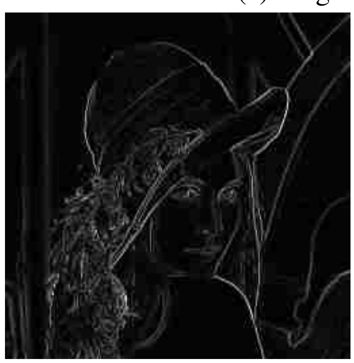


(e) ASC output uing 17 cycles

Figure 4.3. Comparing edge detection outputs for a standard image: Camera man

4.2.3. Standard Image Result: Lena. Similar to Figure 4.3, the Figure 4.4 also shows a standard image used for benchmarking image processing algorithms. Sub-Figure 4.4c shows visibly lesser noise in low intensity areas (such as hat and shoulder areas) as compared to sub-figures 4.4d and 4.4e.

(a) Original input image



(b) Theoretical output

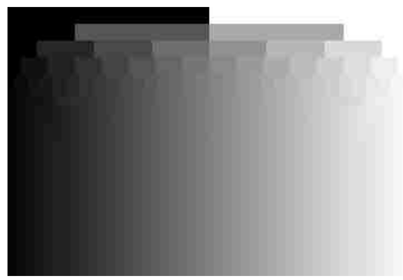
(c) Output using proposed method

(d) ASC output with 16 cycles

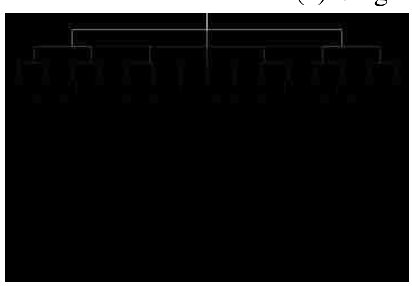
(e) ASC output with 17 cycles

Figure 4.4. Comparing edge detection outputs for a standard image: Lena

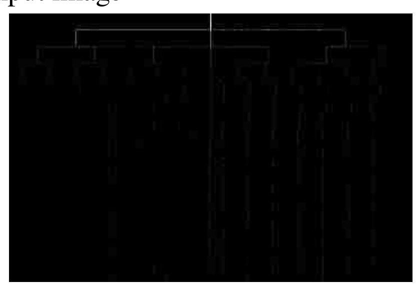
4.2.4. Generated Image Result. Figure 4.5 shows a generated image which consists of varying frequency and intensity components. The image mainly contains a slow-varying intensity gradient from left to right changing from black to white. This will facilitate inspection of the algorithm's performance at various intensity ranges. Sub-Figure 4.5c shows lower intensity performance in the proposed method is better than that of the result shown in Sub-Figure 4.5d and 4.5e.



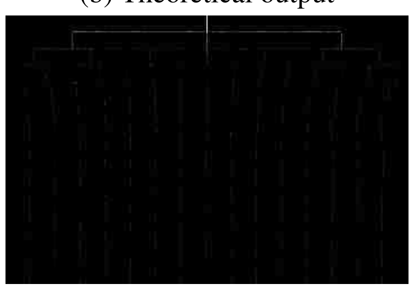
(a) Original input image



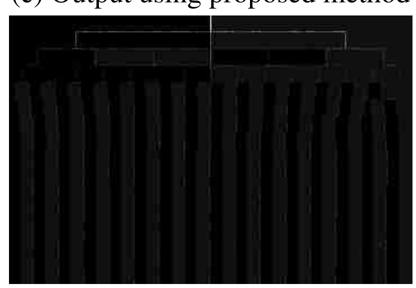
(b) Theoretical output



(c) Output using proposed method



(d) ASC output with 16 cycles

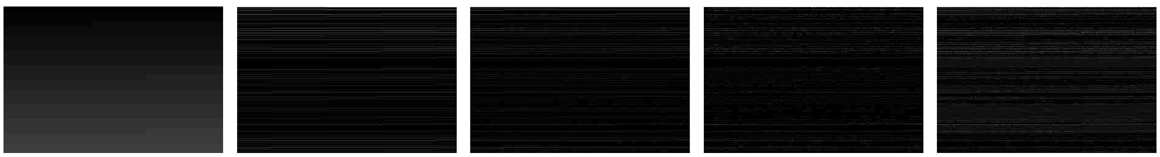


(e) ASC output with 17 cycles

Figure 4.5. Comparing edge detection outputs for a generated image with varying intensity and frequency

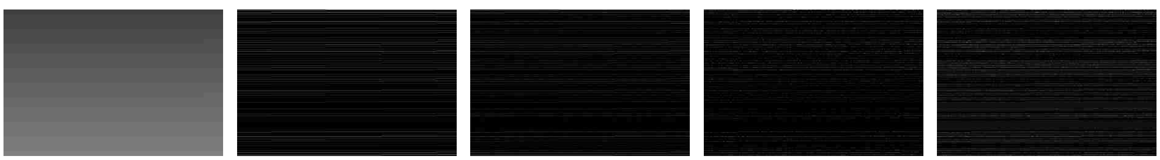
4.3. INTENSITY BASED ANALYSIS

The following input figures were generated using a slight gradient with three different intensity ranges. This helps analyze the performance of each algorithm being compared specifically in terms of intensity. Each of the Figures 4.6, 4.7 and 4.8 below show (a) Original input, (b) Theoretical output using Robert-Cross equation, (c) ASC output with 16 clock cycles, and (d) ASC output with 17 clock cycles.



(a) (b) (c) (d) (e)

Figure 4.6. Low intensity image



(a) (b) (c) (d) (e)

Figure 4.7. Moderate intensity image

(a) (b) (c) (d) (e)

Figure 4.8. High intensity image

To help explain these results, consider the percentage error involved in stochastic process when using truncation method as in [10]. Truncating bits in a number introduces peaks of percentage error as shown in the Figure 4.9. The proposed method performs the same as ASC as defined in [10] in higher numerical ranges, but in lower ranges the percentage error spikes are reduced significantly and the percentage error does not increase beyond 10%.

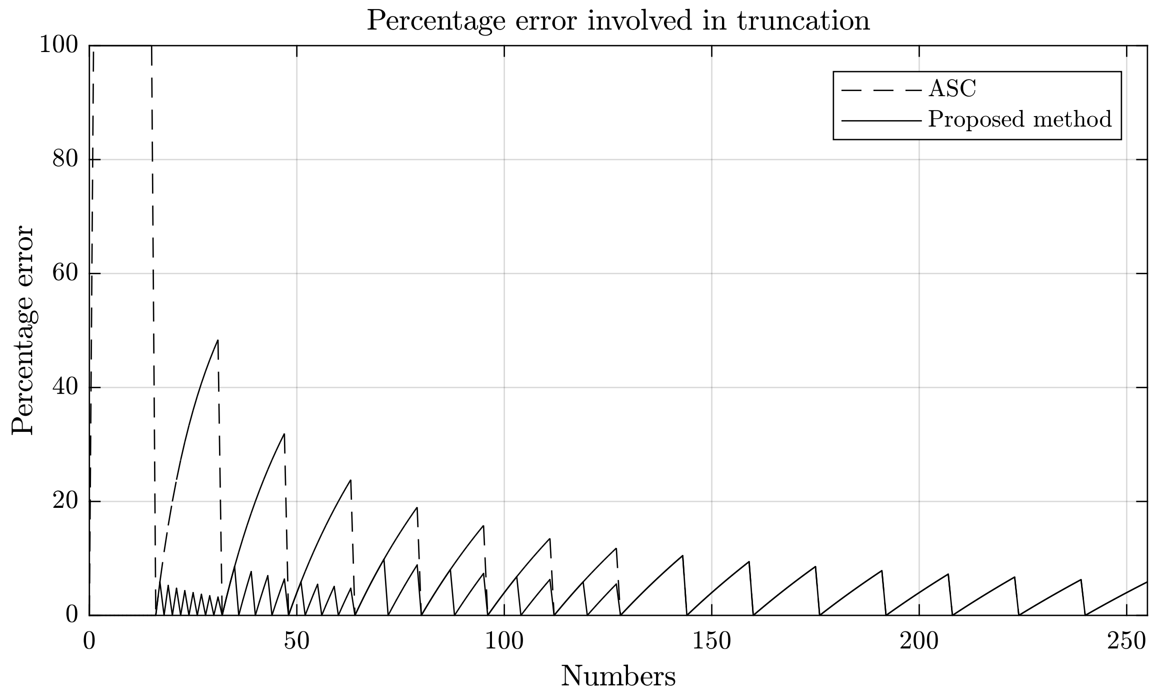
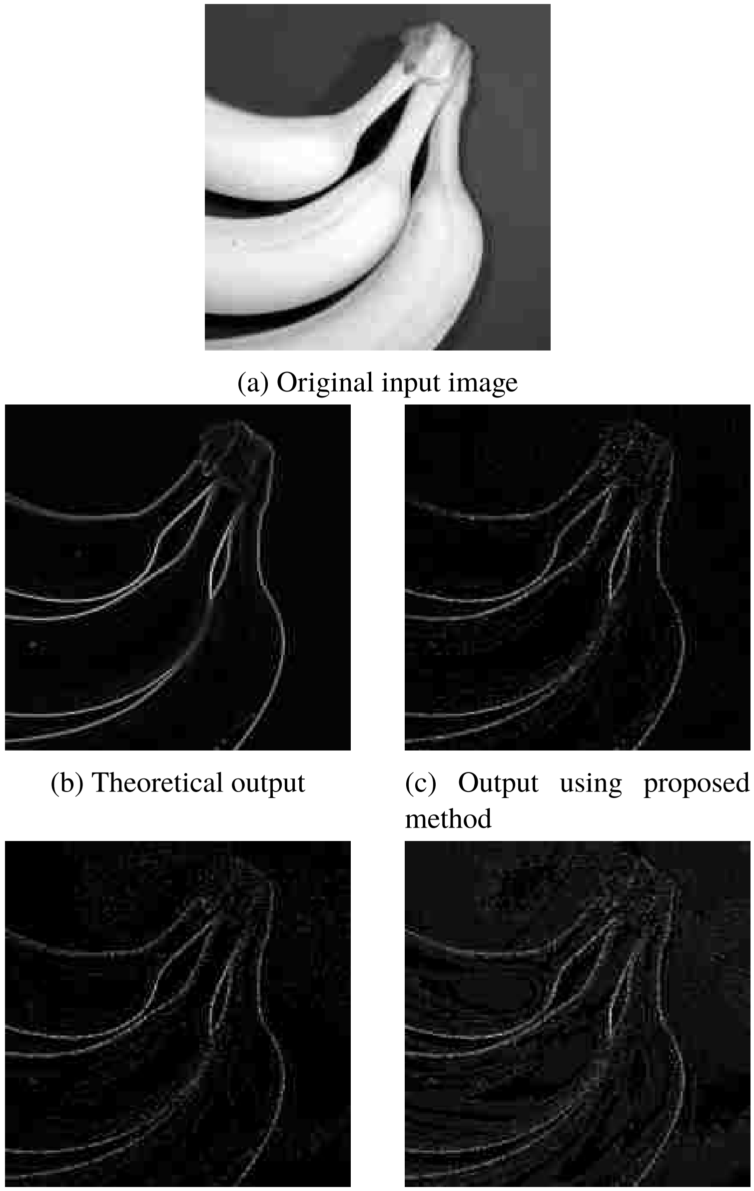


Figure 4.9. Error introduced during truncation in [10] as compared to proposed method

4.4. RESULTS FOR REAL WORLD IMAGES

The following Figures 4.10, 4.11 and 4.12 show some real world examples following similar theme as discussed above. The MSE and PSNR values for these images are also presented in Table 4.1.



(a) Original input image

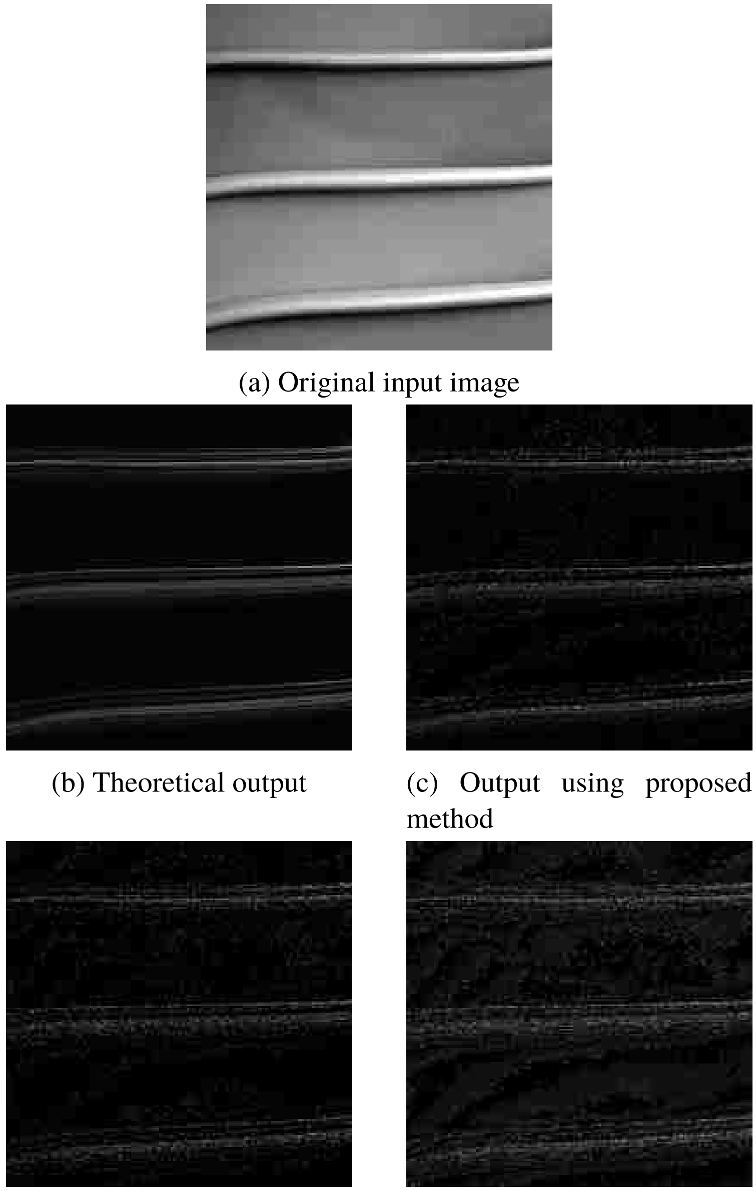
(b) Theoretical output

(c) Output using proposed method

(d) ASC output with 16 cycles

(e) ASC output with 17 cycles

Figure 4.10. Banana image



(a) Original input image

(b) Theoretical output

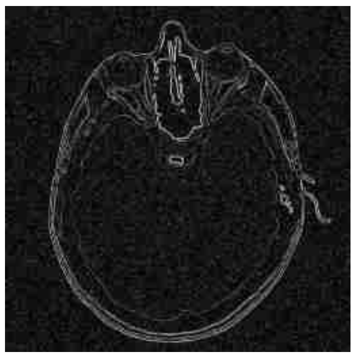
(c) Output using proposed method

(d) ASC output with 16 cycles

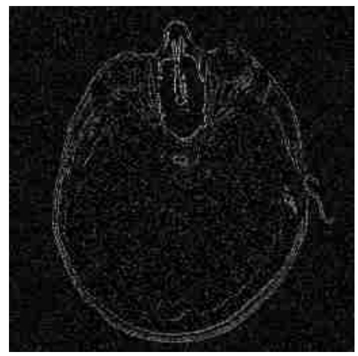
(e) ASC output with 17 cycles

Figure 4.11. Horizontal stripes

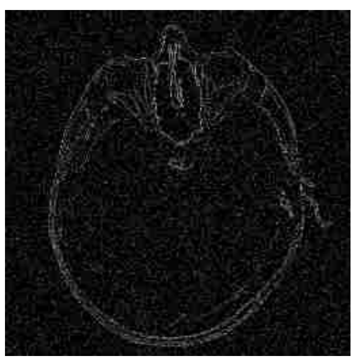
(a) Original input image



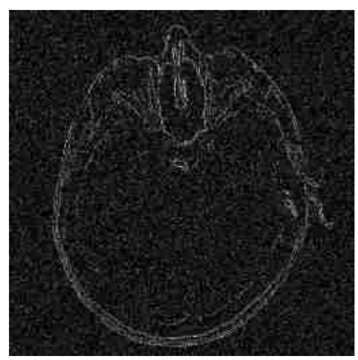
(b) Theoretical output



(c) Output using proposed method



(d) ASC output with 16 cycles



(e) ASC output with 17 cycles

Figure 4.12. MRI of a human skull

5. CONCLUSION

The current document describes an approximate stochastic computing methodology and generalizes the error involved in the process. A mathematical model for the error involved is presented and is used to build constraints to propose a new methodology. This mathematical model can be used as a tool to estimate expected error in any stochastic system where truncation is involved at the input, to any degree.

Using these constraints, the generalized error, and a common understanding of binary information in general, a new method for performing operations on information using approximate stochastic computing techniques is proposed. This new method is compared against simple approximate stochastic computing and adaptive approximate stochastic computing methodologies. The comparison shows that the method proposed performs better than ASC and adaptive ASC in most respects.

Future work involves hardware implementation of the method for computationally challenging applications such as neural networks and artificial intelligence to reduce computation times and area of implementation.

APPENDIX A

GENERATE ILUT

```matlab
%% Housekeeping
clc;clearvars;close all;
set(0, 'defaulttextinterpreter', 'latex');
set(0, 'defaultlegendinterpreter', 'latex');
set(groot, 'defaultAxesTickLabelInterpreter', 'latex');
set(groot, 'defaultLegendInterpreter', 'latex');
set(gcf, 'PaperSize', [6 3.6]);

%% n-bit look up table

% Configure here

n = 8;          % Bit length
k = 4;          % Kept bits after truncation

%% Initialization

% Number range
x = 0:(2^n-1);
% Empty array for error storage
init_empty = x'*0;
% Index of first high bit
x_first_1_index = init_empty;
% Absolute error for proposed method
error = init_empty;
% Percentage error for proposed method
error_percentage = init_empty;
% Absolute error with simple truncation
truncation_error = init_empty;
% Percentage error with simple truncation
truncation_percent = init_empty;
% Condensed information metrics
alpha = init_empty;
beta = init_empty;
mu = init_empty;
% Information bits binary array
I_bin = de2bi((1:(2^n - 1))*0, k, 'left-msb');

% Other temporary variables
x_lmsb = de2bi(x, n, 'left-msb');

% Process
for i = 2:2^n
    x_first_1_index(i) = find(x_lmsb(i,:), 1);

        trunc_index = x_first_1_index(i) + k;

    x_lmsb(i,trunc_index:n) = 0;
    x_decimal_after_trunc = bi2de(x_lmsb(i,:), 'left-msb');

    error(i) = abs(x(i) - x_decimal_after_trunc);
    error_percentage(i) = error(i)*100/x(i);

    truncation_error(i) = mod(x(i),2^(n - k));
    truncation_percent(i) = truncation_error(i)*100/x(i);
```

```

    alpha(i) = find(de2bi(x_decimal_after_trunc, n), 1) - 1;
        beta(i) = length(x_first_1_index(i):n-alpha(i));
        mu(i) = alpha(i) + beta(i) - k;

    I_bin(i,:) = de2bi(bi2de(x_lmsb(i,x_first_1_index(i):n-
       alpha(i)), 'left-msb'), k, 'left-msb');
end

%information bits integers
I_int = bi2de(I_bin, 'left-msb');

save('lut', 'I_int', 'alpha', 'mu', 'k');

%% Plot a few results
plot(x, truncation_percent, '--k');
hold on;
grid on;
plot(x, error_percentage, '-k');
title('Percentage error involved in truncation');
legend('$4$ bit Truncation method', 'Poposed method with $k
   = 4$ kept bits');
xlabel('Numbers');
ylabel('Percentage error');
set(gcf, 'PaperSize', [6 3.6]);
pbaspect([5/3 1 1]);
ylim([0 100]);
xlim([0 max(x)])
```

APPENDIX B

TO CALCULATE STOCHASTIC ERROR ϵ_{SC}

```
%% Housekeeping
clc;close all;clear all;

%% Configuration
n_max = 16;
n_array = [4 6 8 10 12 14 16];

%% Intermediate values
s_array = zeros(1,n_max);
for x = 1:n_max
    s_array(x) = 2^x-1;
end

numberOfIntegers = 5e6;
error = NaN(length(n_array), length(s_array), numberOfIntegers);
inputIntegers = error;
averageError = NaN(length(n_array), length(s_array));
averagePercentError = averageError;

%% Process
for k = 1:length(n_array)
    n = n_array(k);
    for l = 1:length(s_array)
        s = s_array(l);
        inputIntegers(k,l,:) = uint16(ones(1,1, numberOfIntegers)*0.05*(2^n-1));
        for i = 1:numberOfIntegers
            stream = rand(1,s) <= double(inputIntegers(k,l,i))/(2^n-1);
            outputNumber = uint16((mean(stream))*(2^n-1));
            error(k,l,i) = outputNumber - inputIntegers(k,l,i);
        end
        averageError(k,l) = mean(abs(error(k,l,:)));
        averagePercentError(k,l) = mean(100 * abs(error(k,l,:)) ./  (inputIntegers(k,l,:) + 0.01));
    end
end

% save('aa_abs_percent.mat');

%% Set markers

markers = {'o', '+', '*', 'x', 's', 'd', '^'};
set(0, 'DefaultTextInterpreter', 'latex');
set(gcf, 'PaperSize', [6 3.6]);

%% s versus averageError for different n

figure;
hold on;
for i = 1:length(n_array)
    n = n_array(i);
```

```
    s = s_array(s_array < 2^n);
        x = s;
        y = averageError(i,1:length(s));
    plot(x,y, ['-k' markers{i}], 'LineWidth', 1);
end
xlim([0 max(x)]);
ylim([0 max(y)]);
legend('$n = 4$','$n = 6$', '$n = 8$', '$n = 10$', '$n = 12$', '$n = 14$', '$n = 16$')
grid on;
box on;
pbaspect([5/3 1 1]);
title('Average stochastic error for different $n$ as $s$ varies', 'Interpreter', 'latex');
xlabel('Length of stochastic bit stream, $s$','Interpreter', 'latex');
ylabel('Average error $\epsilon_{sc}$','Interpreter', 'latex');

%% s versus averagePercentError for different n

figure;
hold on;
for i = 1:length(n_array)
    n = n_array(i);
    s = s_array(s_array < 2^n);
        x = s;
        y = averagePercentError(i,1:length(s));
    plot(x,y, ['-k' markers{i}], 'LineWidth', 1);
end
xlim([0 max(x)]);
ylim([0 max(y)]);
legend('$n = 4$','$n = 6$', '$n = 8$', '$n = 10$', '$n = 12$', '$n = 14$', '$n = 16$')
grid on;
box on;
pbaspect([5/3 1 1]);
title('Average Percentage stochastic error for different $n$ as $s$ varies', 'Interpreter', 'latex');
xlabel('Length of stochastic bit stream, $s$','Interpreter', 'latex');
ylabel('Average $\epsilon_{sc,p}$','Interpreter', 'latex');

%% Normalized s versus averageError for different n

figure;
for i = 1:length(n_array)
    n = n_array(i);
    s = s_array(s_array < 2^n);
        x = s/(2^n - 1);
        y = averageError(i,1:length(s));
    plot(x,y, ['-k' markers{i}], 'LineWidth', 1);
    hold on;
end
xlim([0 0.2]);
```

```
ylim([0 500]);
legend('$n = 4$','$n = 6$', '$n = 8$', '$n = 10$', '$n = 12$', '$n = 14$', '$n = 16$')
grid on;
box on;
pbaspect([5/3 1 1]);
title('Average stochastic error for different $n$ as normalized $s$ varies', 'Interpreter', 'latex');
xlabel('Normalized length of stochastic bit stream, $\frac{s}{2^n - 1}$','Interpreter', 'latex');
ylabel('Average error $\epsilon_{sc}$','Interpreter', 'latex');

%% Logarithmic normalized s versus averageError for different n

figure;
data_x = [];
data_y = [];
for i = 1:length(n_array)
    n = n_array(i);
    s = s_array(s_array < 2^n);
        x = log(s./max(s));
        y = log(shiftdim(averageError(i,1:length(s)))./(2^(n/2)));
    plot(x,y, ['-k' markers{i}], 'LineWidth', 1);
        data_x = [data_x x(3:end)];
        data_y = [data_y y(3:end)'];
    hold on;
end

combined_data = sortrows([data_x' data_y']);
[P, S] = polyfit(combined_data(:,1), combined_data(:,2), 1);

plot(x, P(1).*x + P(2), '--k', 'LineWidth', 1);

xlim([min(x) max(x)]);
ylim([min(y) max(y)]);
legend('$n = 4$','$n = 6$', '$n = 8$', '$n = 10$', '$n = 12$', '$n = 14$', '$n = 16$', 'Polynomial fit')
grid on;
box on;
pbaspect([5/3 1 1]);
title('Logarithmic normalized $\epsilon_{sc}$ for different $n$ as $s$ varies', 'Interpreter', 'latex');
xlabel('Logarithmic normalized length of stochastic bit stream, $s$','Interpreter', 'latex');
ylabel('$\frac{ln\left(\epsilon_{sc}\right)}{2^{n/2}}$','Interpreter', 'latex');

%% Logarithmic normalized s versus averagePercentageError for different n

figure;
for i = 1:length(n_array)
```

```
    n = n_array(i);
    s = s_array(s_array < 2^n);
        x = log(s./max(s));
        y = log(shiftdim(averagePercentError(i,1:length(s))).*(2^(n/2)));
    plot(x,y, ['-k' markers{i}], 'LineWidth', 1);
    hold on;
end
xlim([min(x) max(x)]);
ylim([min(y) max(y)]);
legend('$n = 4$','$n = 6$', '$n = 8$', '$n = 10$', '$n = 12$', '$n = 14$', '$n = 16$')
grid on;
box on;
pbaspect([5/3 1 1]);
title('Average stochastic error $\epsilon_{sc}$ for different $n$ as $s$ varies', 'Interpreter', 'latex');
xlabel('Length of stochastic bit stream, $s$','Interpreter', 'latex');
ylabel('Average error $\epsilon_{sc}$','Interpreter', 'latex');
```

REFERENCES

[1] Alaghi, A., Chan, W. T. J., Hayes, J. P., Kahng, A. B., and Li, J., 'Optimizing stochastic circuits for accuracy-energy tradeoffs,' in '2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD),' 2015 pp. 178–185, doi: 10.1109/ICCAD.2015.7372568.

[2] Alaghi, A. and Hayes, J. P., 'Exploiting correlation in stochastic circuit design,' in 'Computer Design (ICCD), 2013 IEEE 31st International Conference on,' IEEE, 2013 pp. 39–46.

[3] Alaghi, A. and Hayes, J. P., 'Survey of stochastic computing,' ACM Transactions on Embedded computing systems (TECS), 2013, **12**(2s), p. 92.

[4] Alaghi, A., Li, C., and Hayes, J. P., 'Stochastic circuits for real-time image-processing applications,' in 'Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE,' IEEE, 2013 pp. 1–6.

[5] Copeland, B. J., 'The modern history of computing,' in E. N. Zalta, editor, 'The Stanford Encyclopedia of Philosophy,' Metaphysics Research Lab, Stanford University, winter 2017 edition, 2017.

[6] Davis, L. S., 'A survey of edge detection techniques,' Computer graphics and image processing, 1975, **4**(3), pp. 248–270.

[7] Gaines, B. R., 'Stochastic computing,' in 'Proceedings of the April 18-20, 1967, Spring Joint Computer Conference,' AFIPS '67 (Spring), ACM, New York, NY, USA, 1967 pp. 149–156, doi:10.1145/1465482.1465505.

[8] Gaines, B. R. *et al.*, 'Stochastic computing systems,' Advances in information systems science, 1969, **2**(2), pp. 37–172.

[9] Mars, P. and Poppelbaum, W. J., *Stochastic and deterministic averaging processors*, 1, Peter Peregrinus Press, 1981.

[10] Seva, R., Metku, P., Kim, K. K., Kim, Y.-B., and Choi, M., 'Approximate stochastic computing (asc) for image processing applications,' in 'SoC Design Conference (ISOCC), 2016 International,' IEEE, 2016 pp. 31–32.

[11] Seva, R., Metku, P., Kim, K. K., Kim, Y. B., and Choi, M., 'Approximate stochastic computing (asc) for image processing applications,' in '2016 International SoC Design Conference (ISOCC),' 2016 pp. 31–32, doi:10.1109/ISOCC.2016.7799758.

[12] Vestias, M. and Neto, H., 'Trends of cpu, gpu and fpga for high-performance computing,' in 'Field Programmable Logic and Applications (FPL), 2014 24th International Conference on,' IEEE, 2014 pp. 1–6.

[13] Von Neumann, J., 'Probabilistic logics and the synthesis of reliable organisms from unreliable components,' Automata studies, 1956, **34**, pp. 43–98.

VITA

Keerthana Pamidimukkala received her undergraduate degree in Electronics and Communication Engineering from Jawaharlal Nehru Technological University (JNTU), Hyderabad, India, in the year 2015 during which she briefly worked with Research Centre Imarat (RCI), a division of Defense Research and Development Organization (DRDO) of India. She has worked as a graduate research assistant at the Electromagnetic Compatibility (EMC) laboratory at Missouri S&T in the field of IC design studying the effects of EMC and ESD on ICs. She has also worked as a Hardware Design Engineering Intern in the EMC team at Apple Inc, Cupertino, CA. She received her masters degree in Electrical Engineering from Missouri University of Science and Technology in May, 2018.