UNIVERSITY OF CALIFORNIA,
IRVINE

# Cost-Efficient Approximate Log Multipliers for Convolutional Neural Networks

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Min Soo Kim

2020

# DEDICATION

To my lovely wife and daughter,
Bora and Grace

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ALGORITHMS

# ACKNOWLEDGMENTS

I would like to express my sincere gratitude to those who have helped me complete this dissertation. First and foremost, I would like to thank my advisor, Professor Nader Bagherzadeh, and my partner Alberto Del Barrio who has closely worked with me throughout this project. Also, it has been a pleasure to work with Professor Hyunjin Kim, Professor Román Hermida, Leonardo Oliveira and the colleagues in Advanced Computer Architecture Group (ACAG). I have learned much by working with the great minds, and this dissertation was simply not possible without their guidance, support, and contributions.

I am also grateful to my committee members, Professor Jean-Luc Gaudiot and Professor Rainer Dömer, for their valuable comments, kind support and encouragements.

I express special thanks to Vladimir Alves and the people at NGD Systems, Inc. for offering me an internship position at the company. Working there for many quarters has been a great learning experience, and the position has been the primary source of funding for the Ph.D. study. I also thank the Center for Pervasive Communications and Computing at UCI for the funding assistance.

I thank my parents and my family for their endless support. I also thank my friends at UCI and Bethel Church for the priceless memories together. Lastly and above all, I thank and praise my God for I can do nothing without Him.

# VITA

## Min Soo Kim

### EDUCATION

**Doctor of Philosophy in Electrical and Computer Engineering** **2020**
University of California, Irvine *Irvine, California*

**Master of Science in Electrical and Computer Engineering** **2011**
University of California, Irvine *Irvine, California*

**Bachelor of Applied Science in Engineering Science** **2008**
University of Toronto *Toronto, Canada*

### RESEARCH EXPERIENCE

**Graduate Student Researcher** **2015–2020**
University of California, Irvine *Irvine, California*

**Internship in Engineering** **2017 - 2020**
NGD Systems, Inc. *Irvine, California*

**Research Intern** **Summer 2018**
Microsoft Research *Redmond, Washington*

**Research Engineer** **2011–2014**
Silicon Works Co., Ltd. *Daejeon, South Korea*

**Graduate Student Researcher** **2009–2011**
University of California, Irvine *Irvine, California*

### TEACHING EXPERIENCE

**Teaching Assistant** **2017 - 2018**
**Introduction to Digital Systems**,
**Introduction to Digital Logic Laboratory**,
**Oragnization of Digital Computers Laboratory**,
University of California, Irvine *Irvine, California*

## REFEREED JOURNAL PUBLICATIONS

**Min Soo Kim**, Alberto A. Del Bario, Leonardo Tavares Oliveira, Román Hermida and Nader Bagherzadeh, **"Efficient Mitchell's Approximate Log Multipliers for Convolutional Neural Networks"** **May 2019**
IEEE Transactions on Computers

## REFEREED CONFERENCE PUBLICATIONS

HyunJin Kim, **Min Soo Kim**, Alberto A. Del Bario and Nader Bagherzadeh, **"A Cost-Efficient Iterative Truncated Logarithmic Multiplication for Convolutional Neural Networks"** **June 2019**
IEEE Symposium on Computer Arithmetic

Leonardo Tavares Oliveira, **Min Soo Kim**, Alberto A. Del Barrio, Nader Bagherzadeh and Ricardo Menotti, **"Design of Power-Efficient FPGA Convolutional Cores with Approximate Log Multiplier"** **April 2019**
European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning

**Min Soo Kim**, Alberto A. Del Barrio, Román Hermida and Nader Bagherzadeh, **"Low-power Implementation of Mitchell's Approximate Logarithmic Multiplication for Convolutional Neural Networks"** **January 2018**
Asia and South Pacific Design Automation Conference

## AWARDS

| | |
|---|---|
| **Hyundai Motor Group Scholarship** | 2019 |
| **The Governor Generals Academic Medal, Bronze** | 2004 |
| **Da Vinci Scholar from Leonardo da Vinci National Competition** | 2004 |

# ABSTRACT OF THE DISSERTATION

Cost-Efficient Approximate Log Multipliers for Convolutional Neural Networks

by

Min Soo Kim

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2020

Professor Nader Bagherzadeh, Chair

The breakthroughs in multi-layer convolutional neural networks (CNNs) have caused significant progress in the applications of image classification and recognition. The size of CNNs has continuously increased to improve their prediction capabilities on various applications, and it has become increasingly costly to perform the required computations. In particular, CNNs involve a large number of multiply-accumulate (MAC) operations, and it is important to minimize the cost of multiplication as it requires most computational resources.

This dissertation proposes cost-efficient approximate log multipliers, optimized for performing CNN inferences. Approximate multipliers have reduced hardware costs compared to the conventional multipliers but produce products that are not exact. The proposed multipliers are based on Mitchell's Log Multiplication that converts multiplications to additions by taking approximate logarithm. Various design techniques are applied to Mitchell Log Multiplier, including fully-parallel LOD, efficient shift amount calculation, and exact zero computation. Additionally, the truncation of the operands is studied to create the customizable log multiplier that further reduces energy consumption. This dissertation also proposes using the one's complements to handle negative numbers to significantly reduce the associated costs while having minimal impact on CNN performances. The viability of the proposed designs is supported by the detailed formal analysis as well as the experimental results on CNNs.

The proposed customizable design at $w$=8 saves up to 88% energy compared to the exact fixed-point multiplier at 32 bits with just a performance degradation of 0.2% on AlexNet for the ImageNet ILSVRC2012 dataset.

The effects of approximate multiplication are analyzed when performing inferences on deep CNNs, to provide a deeper understanding of why CNN inferences are resilient against the errors in multiplication. The analysis identifies the critical factors in the convolution, fully-connected, and batch normalization layers that allow more accurate CNN predictions despite the errors from approximate multiplication. The same factors also provide an arithmetic explanation of why bfloat16 multiplication performs well on CNNs. The experiments with deep network architectures, such as ResNet and Inception-v4, show that the approximate multipliers can produce predictions that are nearly as accurate as the FP32 references, while saving significant amount of energy compared to the bfloat16 arithmetic.

Lastly, a convolution core that utilizes the approximate log multiplier is designed to significantly reduce the power consumption of FPGA accelerators. The core also exploits FPGA reconfigurability as well as the parallelism and input sharing opportunities in convolution to minimize the hardware costs. The simulation results show reductions up to 78.19% of LUT usage and 60.54% of power consumption compared to the core that uses exact fixed-point multipliers, while maintaining comparable accuracy on the LeNet for MNIST dataset.

# Chapter 1

# Introduction

For the past several years, the techniques employing Machine Learning allowed significant breakthroughs in many applications, and convolutional neural networks (CNNs) have been widely successful in advancing the field of computer vision. From the milestone network LeNet for the handwritten digit recognition [38], CNNs have been continually studied and improved to perform well even for the large-scale image classification. The CNNs developed in this progress, such as AlexNet and GoogLeNet [33, 63], showed the trend where the amount of computations augmented as the number of layers increased for better accuracy. With such a large and growing number of computations, as well as the rising application of Machine Learning techniques to many areas, it is vital to develop efficient processing hardware units for CNNs. For deep learning to have revolutionary impacts on the real-world applications, their computational costs must meet the timing, energy, monetary, and other design constraints of the deployed services. Many approaches have been studied to reduce the computational costs at all levels of software and hardware, from advances in network architectures [63, 7] down to electronics where even memory devices are extensively researched [62, 60].

One distinction to make in neural network computation is the training versus inference computations. Training teaches neural networks to develop the classification capabilities through the gradient-based backpropagation algorithms performed on a large amount of supplied data. These algorithms involve delicate computations of gradient values and are performed with floating-point units. The DaDianNao project [5] studied applying fixed point arithmetic to training and found that training required much more precision than inference. Hence, it is difficult to apply approximate computing to the training of CNNs. Although training requires more computations when compared to inference, it is still important to reduce the cost of inference as much as possible, because it is the inference that is usually subject to more strict real-world design constraints. The training phase may be performed offline in advance, and the trained network models may be approximated when they are deployed to perform inferences, to meet the design constraints and reduce the cost of services.

Many hardware-based approaches have shown significant improvements for the computational costs of CNN inferences, but there are two limitations commonly found in these works. Some techniques are computationally expensive in order to optimize their methods for each network model, or to retrain networks to compensate the performance degradation from their methods [35, 70]. Also, many techniques such as [55] are only effective for small networks and cannot scale to deeper CNNs, as they report much worse performance results when tested for deeper networks. In the past it had been shown that small number of bits are sufficient for small CNNs and many techniques relied on aggressive quantization, but more complex networks require more bits to properly represent the amount of information [37].

One promising hardware-based approach is the application of approximate multiplication to CNN inferences. The convolution layers in CNNs consist of a large number of multiply-accumulate (MAC) operations, and they take up the majority of computations for CNN inferences [54]. The MAC operations are ultimately performed in the hardware circuits, and it is important to minimize the cost of these circuits to perform more computations with the same amount of resources. For MAC operations, multiplications are more complex than additions and consume most resources. The proposed methodology is to minimize the cost of multiplication by replacing the conventional multipliers with approximate multipliers. Unlike aggressive quantization that trades off numeric precision, approximate multipliers trade off arithmetic accuracy that is less dependent on the network models, making them better suited for deeper CNNs. The approach does not involve any optimization to a target network model or require additional processing of the network models, allowing easy adaptation into the ASIC and FPGA accelerators.

Approximate multipliers are significantly cheaper compared to the exact multipliers, but they introduce errors in the results. There are many different types of approximate multipliers, with various costs and error characteristics. Some designs use the electronic properties [6] and some approximate by intentionally flipping bits in the logic [13], while others use algorithms to approximate multiplication [58]. Several previous research papers have shown that it is possible to apply approximate multiplication to the inference stage of neural networks after training with exact arithmetic [45, 58, 50, 13]. Such techniques usually demonstrated small drops in performance but had significant reduction in resources. The resource reduction, especially the power savings, would be beneficial for embedded systems and datacenters, as emphasized by the efforts from Google to create a custom TPU processor for Machine Learning on its datacenters [24].

The logarithmic multiplier based on Mitchell's Algorithm [47] is a promising approximate multiplier for neural network computation. Figure 1.1 shows the difference between the conventional fixed-point multiplier and the log multiplier. This multiplier converts multiplications into additions by taking approximate logarithm. This algorithm is well known to have a significant benefit in area and power savings while maintaining a reasonably low error rate. Many research such as [4, 46, 1, 42] have recognized its benefits and attempted to improve it since the original proposal [47]. The original algorithm has a worst case relative error that is proven to be as low as 11.1%, and this property is potentially important in

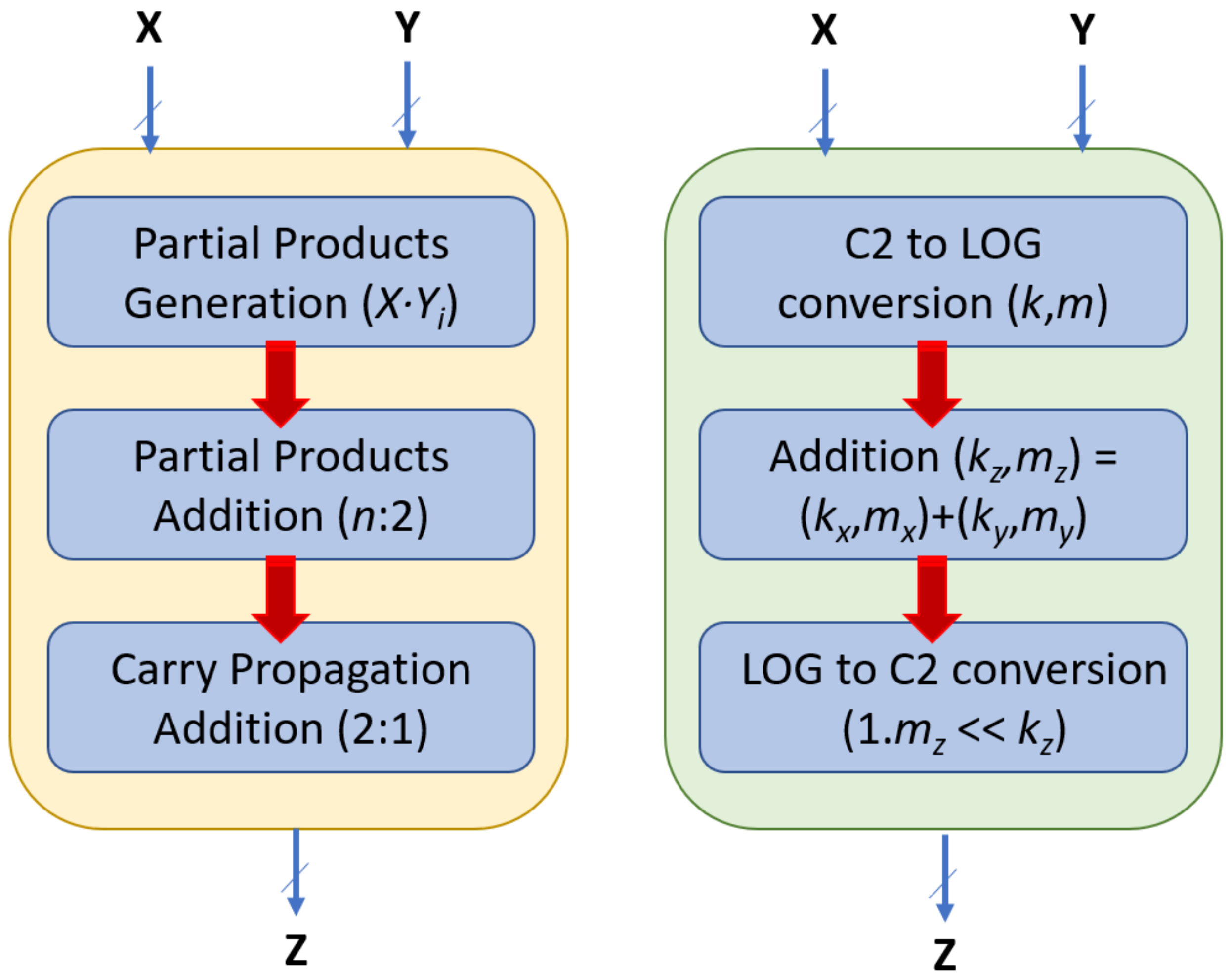

X
Y
Partial Products Generation ($X\cdot Y_i$)
Partial Products Addition ($n$:2)
Carry Propagation Addition (2:1)
Z
X
Y
C2 to LOG conversion ($k$,$m$)
Addition ($k_z$,$m_z$) = ($k_x$,$m_x$)+($k_y$,$m_y$)
LOG to C2 conversion ($1.m_z << k_z$)
Z

Figure 1.1: Difference between (a) the conventional fixed-point multiplication and (b) the approximate log multiplication.

neural networks that emulate firing of neurons. A large worst-case error would have a greater chance of incorrectly firing a neuron, which would lead to a larger probability of incorrect classification. Another important benefit of the log multiplication is the consistent error characteristics, which allows for consistent observation of the effects across various CNN instances. In this dissertation, the consistent observation enabled the in-depth analysis of the effects of approximate multiplication on CNN inferences.

The following contributions are made in this dissertation toward the study of approximate multiplication for CNN inferences. Firstly, cost-efficient approximate multipliers based on

Mitchell's Algorithm are presented in Chapter 2. We present an efficient implementation of Mitchell's Algorithm and apply various techniques to create customizable designs that can reduce significant amount of resources compared to exact multiplication. The effects of the proposed log multipliers on CNN inferences are studied in Chapter 3. Various experiments on CNN models support the conclusion that the proposed designs are suitable for CNN inferences, and the analysis also provide the theoretical understanding of the effects of approximate multiplication and why CNNs are resilient against the errors in multiplication. A convolution core design that utilizes the proposed multipliers is presented in Chapter 4. The convolution core design provides the opportunities for resource sharing between multiple instances of the log multipliers, and consumes significantly less power and LUT resources for a CNN accelerator on FPGA. Lastly, the conclusion of this dissertation is presented in Chapter 5.

# Chapter 2

# Design of Cost-Efficient Approximate Log Multipliers

This chapter describes the proposed approximate log multipliers optimized for CNN inferences. The multipliers are based on Mitchell's Algorithm [47], and various techniques are applied to minimize their hardware costs while maintaining reasonable accuracies on CNN inferences.

## 2.1 Mitchell Log Multiplier

The Mitchell's log multiplication algorithm and the proposed implementation is described in this section. The original paper on Mitchell's Algorithm [47] does not specify any digital circuit implementation, so a low-cost implementation is created and optimized through various techniques. The implementation is the basis from which the other proposed designs are derived, and its development as well as the performance on CNNs were published in [29].

### 2.1.1 Mitchell's Algorithm

The first approach to logarithmic multipliers was presented by J. Mitchell in 1962 [47]. The logarithmic multiplication of two numbers $A \cdot B$ requires converting them to logarithm, then adding both logarithms and finally computing the antilogarithm of the result.

Equation 2.1 represents Z, an n-bit number.

$$Z = \sum_{i=0}^{n-1} 2^i z_i = 2^k \left(1 + \sum_{i=j}^{k-1} 2^{i-k} z_i\right), \qquad k \geq j \geq 0. \tag{2.1}$$

where $k$ is the position corresponding to the leading one, $z_i$ is a bit value at the $i^{th}$ position, and $j$ depends on the number's precision. Then, the logarithm with the basis 2 of $Z$ is expressed by Equation 2.2.

$$log_2(Z) = log_2\left(2^k\left(1 + \sum_{i=j}^{k-1} 2^{i-k} z_i\right)\right) = log_2\left(2^k(1+x)\right) = k + log_2(1+x) \tag{2.2}$$

Then, the expression $\log_2(1+x)$ is approximated with $x$, as $\forall x \in [0,1]$ both expressions provide close results. Figure 2.1 shows the resulting approximate logarithm compared against exact logarithm. The original work provided the mathematical analysis of the error and proved that the maximum error was $-11.1\%$ relative to the exact product [47]. In other words, the magnitude of the product from Mitchell's Algorithm will be relatively smaller than that of the exact product, from 0% up to 11.1%. The Mitchell Log Multiplier cannot produce a product that has a bigger magnitude than the exact product.

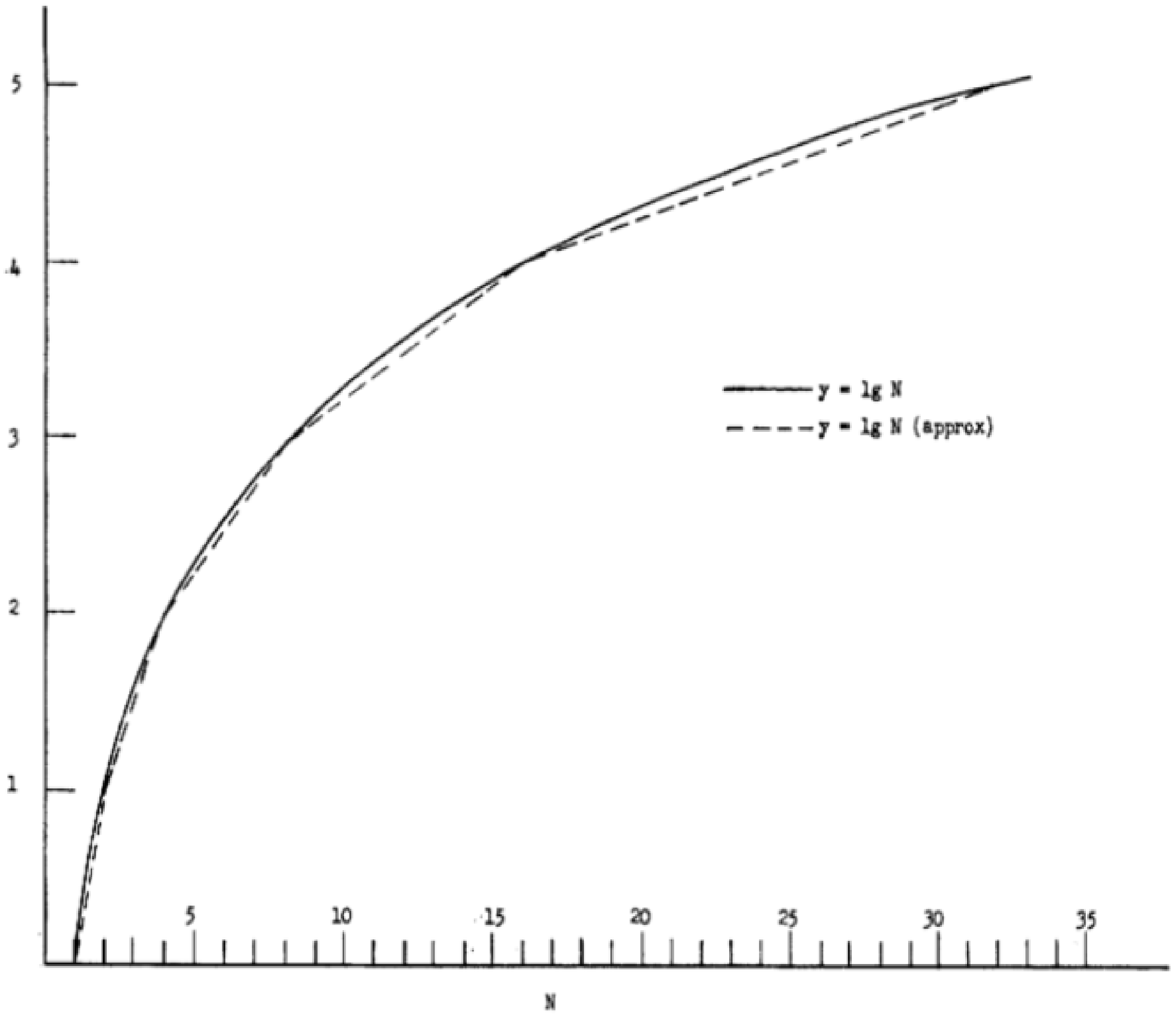


Figure 2.1: Approximate logarithm from Mitchell's Algorithm, compared against logarithm

**Algorithm 1** Digital Logic Implementation of Mitchell's Algorithm

```
Require: A, B: $n$-bits
Ensure: P: 2n-bits                                  ▷ P is an approximate product
                                                                 ▷ Logarithm
  $h_A \leftarrow LOD(A), h_B \leftarrow LOD(B)$
  $k_A \leftarrow ENC(h_A), k_B \leftarrow ENC(h_B)$
  $x_A \leftarrow A << (n - k_A - 1), x_B \leftarrow B << (n - k_B - 1)$
                                                ▷ Addition in the LNS domain
  $op1 \leftarrow' 0' \,\&\, k_A \,\&\, x_A[n-2..0]$
  $op2 \leftarrow' 0' \,\&\, k_B \,\&\, x_B[n-2..0]$
  $L \leftarrow op1 + op2$
                                                             ▷ Antilogarithm
  $charac \leftarrow L[n + \log_2(n) - 1..n - 1]$
  $lr \leftarrow charac[\log_2(n)]$
  $m \leftarrow' 1' \,\&\, L[n-2..0]$
  if $lr =' 1'$ then                                  ▷ Large characteristic
      $shamtL \leftarrow ('0' \,\&\, charac[\log_2(n) - 1..0]) + 1$
      $D \leftarrow m << shamtL$
  else                                                ▷ Small characteristic
      $shamtR \leftarrow n - charac[\log_2(n) - 1..0] - 1$
      $D \leftarrow m >> shamtR$
  end if
                                           ▷ Check if the result should be zero
  if $A = 0 \vee B = 0$ then
      $P \leftarrow 0$
  else
      $P \leftarrow D$
  end if
```

### 2.1.2 Proposed Design

The proposed implementation is detailed by Algorithm 1. Figure 2.2 shows the design of our logarithmic multiplier. It must be noted that & stands for the concatenation symbol and x[b..a] represents the bits that range from positions b to a belonging to signal x.

The first step is implemented through the Leading One Detectors (LOD) and the Encoders (ENC). It must be noted that the LOD module produces a one-hot representation of the leading one. Hence, the encoder is composed of just a set of OR gates, instead of the priority encoder employed in [4, 45]. Our LOD module leverages the implementation provided in

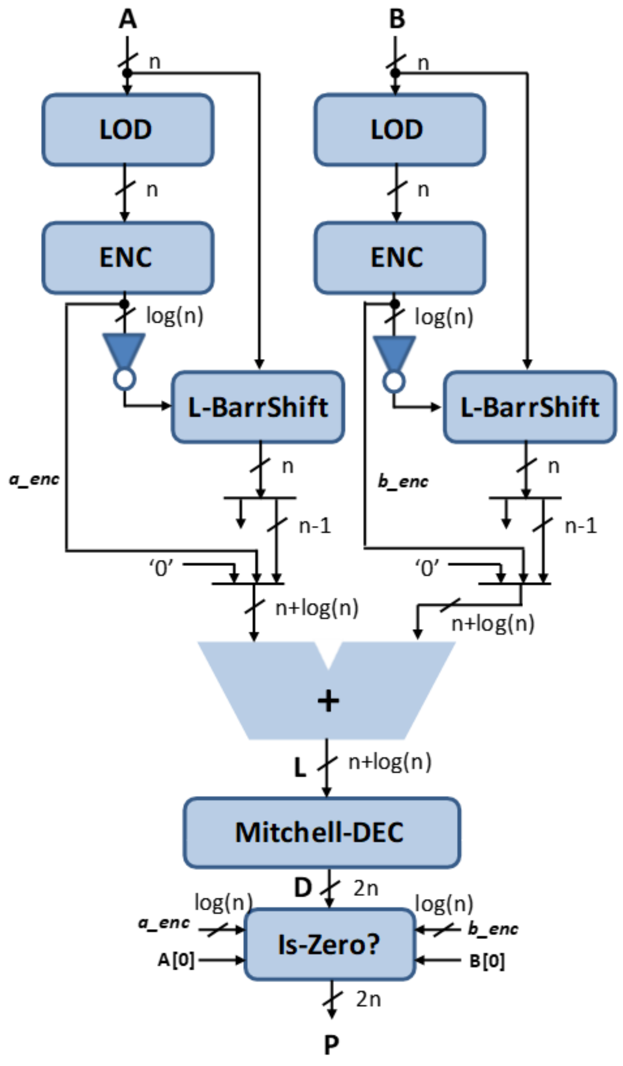


Figure 2.2: Mitchell Logarithmic Multiplier according to Algorithm 1

[4, 45], where the proposed 4-bits LOD blocks can be modeled with Equations 2.3 and 2.4.

$$h_j = z_j \cdot m_j, \qquad 0 \leq j \leq 4 \tag{2.3}$$

$$m_j = \begin{cases} 1, & j = 3 \\ \overline{z_{j+1}} \cdot m_{j+1}, & 0 \leq j \leq 3 \end{cases} \tag{2.4}$$

where $h_j$ is the $j^th$ bit belonging to the hot one representation (H) of the leading one in Z. Expanding Equation 2.4 for a generic bitwidth n, it would be possible to obtain the expression given in Equation 2.5.

$$m_j = z_j \cdot \overline{\sum_{i=j+1}^{n-1} z_i} \tag{2.5}$$

Using Equation 2.5 and constructing an or-tree in the same fashion as the Kogge-Stone adder [31] calculates the carry-in signals of an addition, it is possible to obtain Equations 2.6 and 2.7, which model our fully parallel LOD.

$$m_{i,j} = \begin{cases} z_j, & i = 0 \\ m_{i-1,j}, & i > 0, (n-1-j) < 2^{i-1} \\ m_{i-1,j} + m_{i-1,j+2^{i-1}}, & i > 0, (n-1-j) \geq 2^{i-1} \end{cases} \tag{2.6}$$

$$\forall i, 0 \leq i \leq log(n), \forall j, 0 \leq j < n$$

$$h_j = \begin{cases} z_j, & j = n-1 \\ \overline{(m_{log(n),j+1})} \cdot z_j, & j < n-1 \end{cases} \tag{2.7}$$

$\overline{(m_{log(n),j+1})}$ is a signal that indicates whether or not there is a '0's chain at the left of $z_j$, $h_j$ is the $j^{th}$ bit belonging to the one-hot representation (H) of the leading one in Z. As an example, a 4-bit LOD block is shown in Figure 2.3.

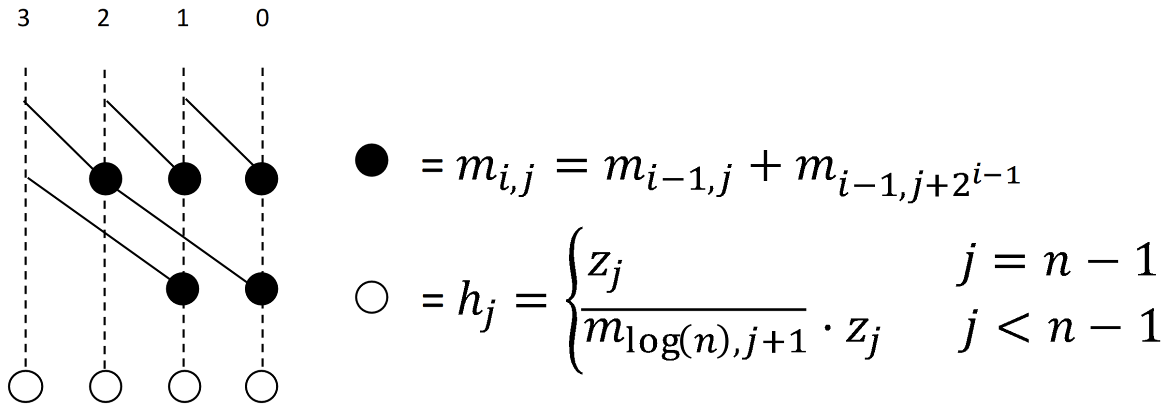

3
2
1
0
● $= m_{i,j} = m_{i-1,j} + m_{i-1,j+2^{i-1}}$
○ $= h_j = \begin{cases} z_j & j = n-1 \\ \overline{m_{\log(n),j+1}} \cdot z_j & j < n-1 \end{cases}$

Figure 2.3: 4-bit leading one detector

Finally, Equation 2.8 gives the value of $e_i$, if it is the $i^{th}$ bit of the encoded value E.

$$e_i = \sum_{j=0}^{n-1} h_j, \qquad \forall i, 0 \leq i \leq log(n) \tag{2.8}$$

In order to compute the mantissas $x_A$ and $x_B$, the shift amount is computed utilizing one's complement arithmetic, as $n - k_i - 1$ is equivalent to not$(k_i)$ when n is a power of 2. Afterwards two left barrel shifters generate the mantissas, which concatenated with the corresponding characteristics compose the two operands added to compute L.

Afterwards, the result L needs to be decoded. Figure 2.4 depicts the structure of our Mitchell Decoder. In Antilogarithm of Algorithm 1, two cases are distinguished depending on the most significant bit of L (labelled as $lr$), which is also the most significant bit of the characteristic of L. If $lr$='1', the mantissa of L must be shifted at least n positions to the left. That is why a 2n-bit left barrel shifter is employed for such purpose. It must be noted that when this left shifter is being used, the shift amount $shamtL$ is always increased by 1. Hence, this shifter has been customized to always shift one extra position to the left. In this way an addition is avoided at zero cost. On the other hand, when $lr$='0', its mantissa must be shifted to the

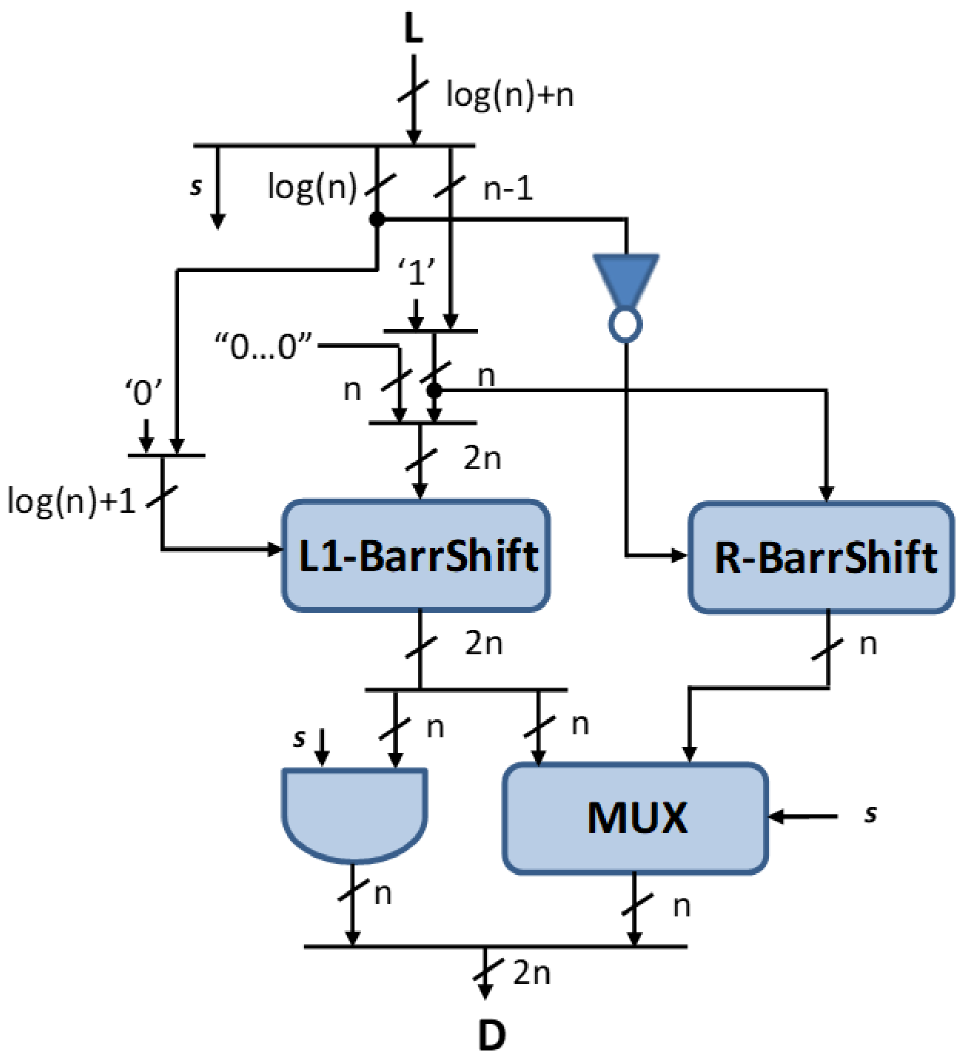


Figure 2.4: Mitchell Decoder

Table 2.1: Complexity Comparison

| | LOD | ENC | SHT | ADD | Zero |
|---|---|---|---|---|---|
| [4, 45] | 4-bit blocks [1] | Priority | 2 Left (n) | 1 (2n), 2 (log(n)) | – |
| Ours | Parallel | Or-tree | 2 Left (n), 1 Right (n), 1 Left(2n) | 1 (log(n) + n) | Or-tree |

left by n-1 positions at most, which is equivalent to shift to the right by *shamtR*. *shamtR* is also efficiently calculated by employing one's complement arithmetic. As can be observed, the right barrel shifter bitwidth is just n-bits. Thus, while the least significant bits of D must be selected using a multiplexer (MUX), the most significant ones can be obtained through an AND gate with the most significant bit of L.

When any of the operands is zero, the result P must be zero. It has been demonstrated by Mrazek et al. in [50] that it is important to produce the accurate zero result when one of the operands is zero, and we implemented a zero detection unit to correctly handle it. In order to implement it, we leverage the following property: if an operand is zero, the value provided by ENC is zero and the least significant bit of the operand is '0'. This is shown in Figure 2.5 and the logic within the darkened area produces valid results just after the encoded values of the one-hot representations are computed.

### 2.1.3 Implementation Complexity

Table 2.1 contains a detailed summary of the complexity between our proposal and the one presented in [4, 45]. It must be noted that the latter is an iterative design that combines several basic blocks (BBs) to obtain different accuracies. Thus, the data shown in the table corresponds with just one BB, while our Mitchell's multiplier data correspond with the whole multiplier.

As shown in Table 2.1, the leading one detection(LOD) stage is different. While the LOD

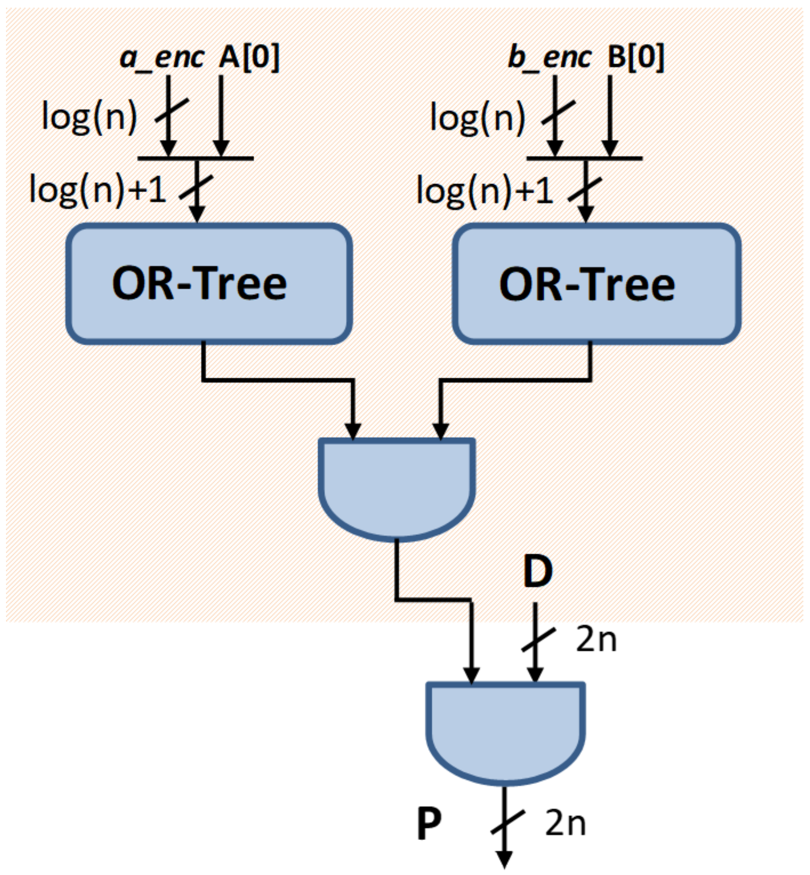


Figure 2.5: Is-Zero block structure: performs both zero detection and correct result generation

implementations in [4, 45, 1] are based on 4-bit blocks interconnected through carry signals, our implementation is fully parallel and therefore faster. Second, as our LOD produces a one-hot representation, the encoding phase (ENC) is realized through OR-trees instead of complex priority encoders. In terms of shifters (SHT) our proposal requires more instances of these resources, but less adders (ADD). It must be noted that in both SHT and ADD columns the size of every resource is specified among parentheses, as they vary depending on the multiplier implementation. Finally, our proposal includes a zero detection unit based on the encoded signals, whose bit width is log(n), which is simpler than employing two n-bit comparators with zero.

### 2.1.4 Synthesis Results

To evaluate the power and area benefits of the proposed design, we performed synthesis using Synopsys Design Compiler and compared it against the synthesis results of the exact fixed-point multiplier and the 2-stage iterative logarithmic multiplier. The exact multiplier is automatically synthesized by Design Compiler from the simple Verilog multiplication, and the 2-stage iterative logarithmic multiplier presented in [3] was modified to remove all pipeline registers and add the zero detection unit. We used a 32nm digital standard cell library from Synopsys, and repeated the synthesis for 8, 16, and 32 bits. The synthesis was performed with "Ultra" effort at the clock frequency of 250 MHz, because we wanted to see the maximum power savings without being constrained by the timing. The 2-stage iterative multiplier at 32 bits was time constrained and had an increased area, but we did not adjust the clock speed of the experiment because of the inefficiency of the design. We also compared the referenced error rates of these multipliers.

Table 2.2 shows the synthesis results of the multipliers to compare their power and area. Although our optimized design has the worst error rate, it is significantly smaller and consumes

Table 2.2: Comparison of power and area after synthesis

| | **N=8** | | | **N=16** | | | **N=32** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Exact** | **Proposed** | **Iter. Log** | **Exact** | **Proposed** | **Iter. Log** | **Exact** | **Proposed** | **Iter. Log** |
| **Mean Err. (%)** | 0 | -3.77 | -0.83 | 0 | -3.83 | -0.99 | 0 | -3.87 | N/A |
| **Worst Case Error (%)** | 0 | -11.11 | -6.25 | 0 | -11.11 | -6.25 | 0 | -11.11 | -6.25 |
| **Cell Area ($um^2$)** | 403 | 312 | 872 | 1681 | 909 | 2189 | 6409 | 2161 | 7220 |
| **Critical Path ($ns$)** | 1.07 | 1.13 | 1.75 | 2.23 | 2.31 | 3.77 | 3.78 | 3.70 | 4.00 |
| **Total Power ($mW$)** | 0.269 | 0.197 | 0.544 | 1.240 | 0.549 | 1.310 | 6.02 | 1.41 | 4.64 |
| **Power Saving (%)** | – | 26.8 | -102.2 | – | 55.7 | -5.6 | – | 76.6 | 22.9 |

less power than the other multipliers. The critical path length of the proposed design is comparable to that of the exact multiplier and clearly shorter than the critical path of the iterative design. Our logarithmic multipliers show better reduction of power and area as the number of bits increases, thus possessing better scalability. Compared to the exact fixed-point multiplier, it saves up to 76.6% of power at 32 bits and shows a clear saving of 26.8% even at 8 bits. The proposed design achieves larger power and area savings compared to the 2-stage iterative multiplier that had been applied to the neural networks in [45]. In fact, the iterative multiplier design seems inefficient and consumes more power and area than the exact multiplier at 8 and 16 bits. We divided the 2-stage iterative logarithmic multiplier to create the 1-stage multiplier without the error correction, but it was still larger than our proposed design and consumed 0.689 $mW$ at 16 bits, while having the worst case relative error of 25% [4]. It must be noted that the authors of [45] had a different aim than ours. They focused on pipelining the iterative design and compared it against the matrix multiplier in Xilinx Spartan 3 FPGA. Nevertheless, in this dark silicon era where we are limited primarily by power, our proposed logarithmic multiplier suits better than the state-of-the-art for CNN inferences.

## 2.2 Truncated Mitchell Log Multiplier, Mitch-$w$

While the Mitchell Log Multiplier is very cost-efficient, it is possible to further approximate it by truncating mantissa bits. The LODs locate the leading ones and most significant portions of the input operands so that truncating the operands after taking logarithm enables performing reasonably accurate multiplication, while significantly reducing the hardware cost of the multiplier. The design of Mitch-$w$ multipliers and their performances on CNNs were published in [30].

### 2.2.1 Proposed Design

Our proposal is detailed by Algorithm 2 and Figure 2.6. It must be noted that & stands for the concatenation symbol and $x[b..a]$ represents the bits that range from positions $b$ to $a$ belonging to signal $x$. This design will hereafter be referred to as Mitch-$w$.

The main difference with respect to the Mitchell multiplier in Section 2.1 is the introduction of the parameter $w$. This parameter indicates that only the most significant $w$ bits of the operand will be taken into account. As the leading one is encoded into the characteristic, this means that only the most significant $w - 1$ mantissa bits are considered to be added. Some approaches in literature [17, 58] have shown that under the paradigm of approximate computing, the most significant part of a value may be sufficient to provide an acceptable approach. Figure 2.7 shows the truncation schemes presented at [17] and ours. Figures 2.7a and 2.7b illustrate how [17] unbiases and truncates an operand using $w$ bits, namely: the leading one located at the $k^{th}$ position, $w - 2$ bits, and an extra '1' for approximating the least significant part. This operand will be driven to a $w$x$w$ multiplier then. On the other hand, Figures 2.7c and 2.7d describe the resulting mantissa after left-shifting the operand, and how the $w - 1$ most significant bits (excluding the leading one) are used in combination

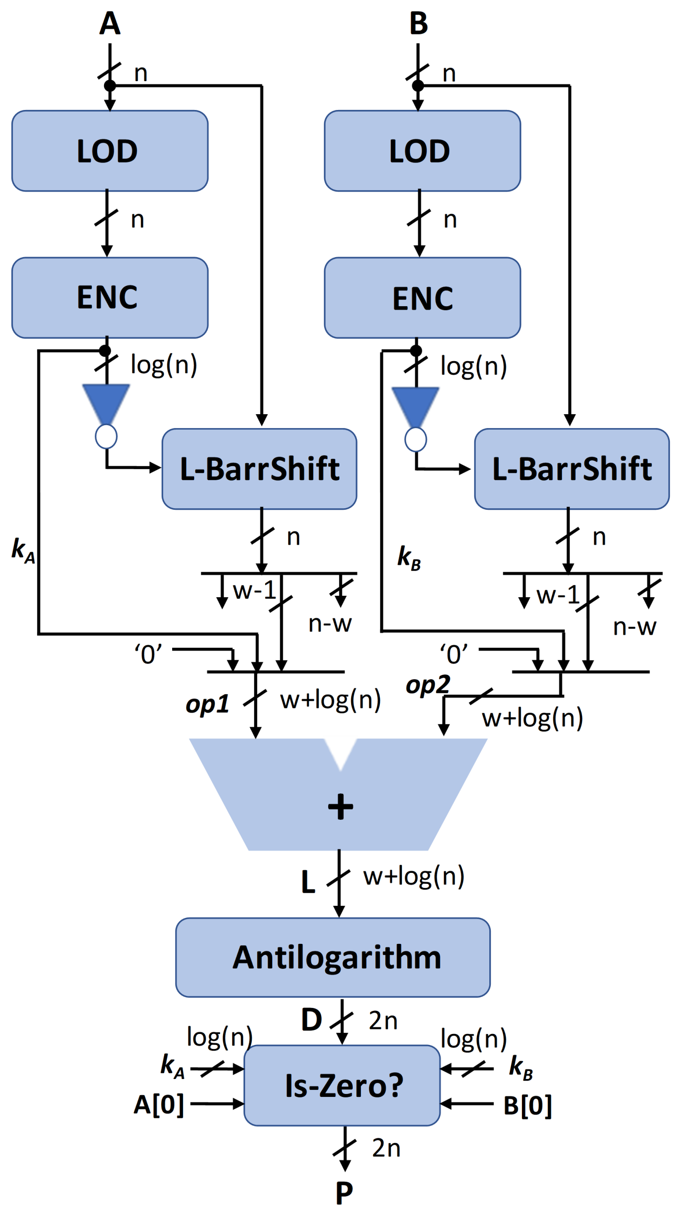


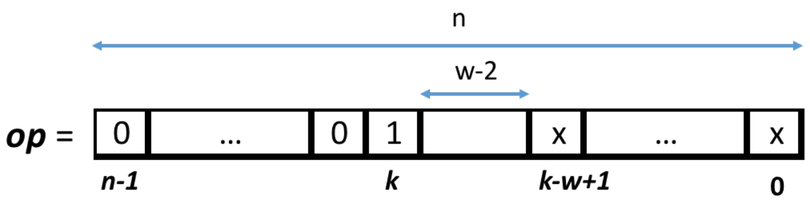
Figure 2.6: Truncated Mitchell Log Multiplier, Mitch-$w$

**Algorithm 2** Truncated Mitchell Log Multiplier (Mitch-$w$)

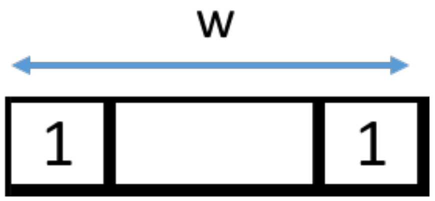
```
Require: A, B: $n$-bits, $w \in [0,n\text{-}1]$
Ensure: P: 2n-bits                                    ▷ P is an approximate product
                                                                      ▷ Logarithm
  $h_A \leftarrow LOD(A), h_B \leftarrow LOD(B)$
  $k_A \leftarrow ENC(h_A), k_B \leftarrow ENC(h_B)$
  $x_A \leftarrow A << (n - k_A - 1), x_B \leftarrow B << (n - k_B - 1)$
                                                       ▷ Addition in the LNS domain
  $op1 \leftarrow' 0' \ \& \ k_A \ \& \ x_A[n-2..n-w]$
  $op2 \leftarrow' 0' \ \& \ k_B \ \& \ x_B[n-2..n-w]$
  $L \leftarrow op1 + op2$
                                                                  ▷ Antilogarithm
  $charac \leftarrow L[w + \log_2(n) - 1..w-1]$
  $lr \leftarrow charac[\log_2(n)]$
  $m \leftarrow' 1' \ \& \ L[w-2..0]$
  if $lr =' 1'$ then                                       ▷ Large characteristic
      $shamtL \leftarrow ('0' \ \& \ charac[\log_2(n)-1..0]) + 1$
      $D \leftarrow m << shamtL$
  else                                                     ▷ Small characteristic
      $shamtR \leftarrow n - charac[\log_2(n)-1..0] - 1$
      $D \leftarrow m >> shamtR[\log_2(n)..\log_2(n) - \log_2(w)]$
  end if
                                             ▷ Check if the result should be zero
  if $A = 0 \vee B = 0$ then
      $P \leftarrow 0$
  else
      $P \leftarrow D$
  end if
```

with the characteristic $k$ to compose the operand that will be driven to the adder within the logarithmic multiplier. These differences ultimately stem from the fact that our designs perform the truncation in the approximate logarithmic domain while the authors in [17] operate in the linear domain.

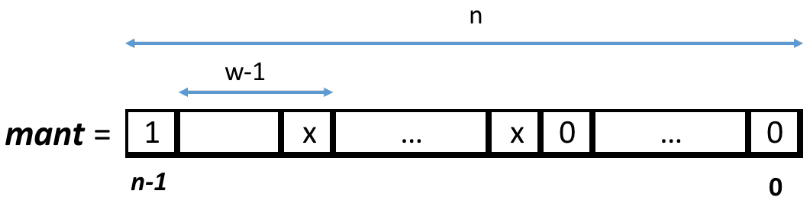
The use of the aforementioned truncated summands implies using a customized antilogarithm block. The design of this module is shown in Figure 2.8. As observed, utilization of the $w$ parameter has a large impact on power and area of this block. The original left barrel shifter shown in Section 2.1 decreases its size from $2n$ to $n + w$, while the right barrel shifter and the multiplexer decrease their sizes from $n$ to $w$. Reducing the size of these blocks has a positive



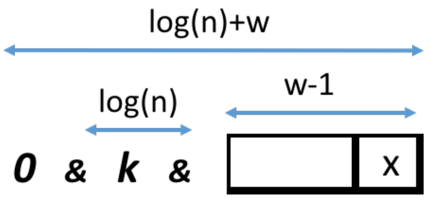
(a) Multiplier operand



(b) Unbiased and rounded operand driven to the multiplier input within DRUM [17]



(c) Mantissa truncation within Mitch-$w$



(d) Operand driven to the adder input of Mitch-$w$

Figure 2.7: Operands truncation methods

impact on power and area, but at the expense of losing some accuracy.

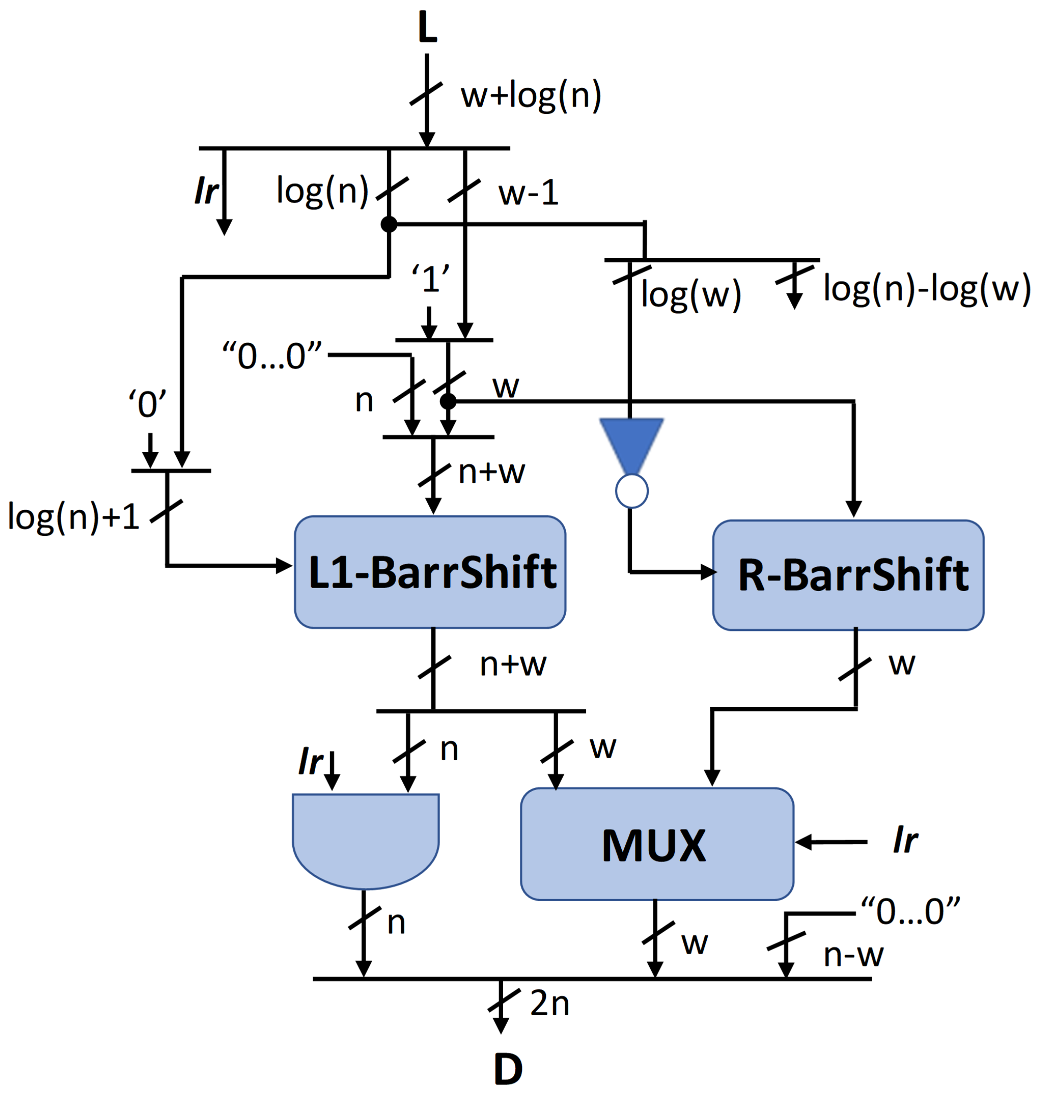


Figure 2.8: Customizable Antilogarithm block

### 2.2.2 Error Study of Truncation

In this subsection, an analysis on the use of the truncated logarithmic multiplier is presented. This error ($E_D$) is the one produced at the output of the antilogarithm block, as truncating some bits on the operand mantissas does not affect the zero-checking block. Thus, it must be noted that the error with respect to a conventional Mitchell Log Multiplier is first produced when truncating the operands.

$$\widehat{m} = m/2^{n-1} < 1 \; , \tag{2.9}$$

$$y_A = x_A \wedge (2^{n-1} - 1) \tag{2.10}$$

$$y_B = x_B \wedge (2^{n-1} - 1) \tag{2.11}$$

$$y'_A = x_A \wedge (2^{n-1} - 2^{n-w}) > y_A - 2^{n-w} \; , \tag{2.12}$$

$$y'_B = x_B \wedge (2^{n-1} - 2^{n-w}) > y_B - 2^{n-w} \; , \tag{2.13}$$

$$y_L = (y_A + y_B) \wedge (2^{n-1} - 1) \; , \tag{2.14}$$

$$y'_L = (y'_A + y'_B) \wedge (2^{n-1} - 1) \; , \tag{2.15}$$

$$k_L = k_A + k_B + (((y_A + y_B) \wedge (2^{n-1})) >> (n-1)) \; , \tag{2.16}$$

$$k'_L = k_A + k_B + (((y'_A + y'_B) \wedge (2^{n-1})) >> (n-1)) \; , \tag{2.17}$$

$$\begin{aligned} L &= k_A * 2^{n-1} + y_A + k_B * 2^{n-1} + y_B \\ &= k_L * 2^{n-1} + y_L \; , \end{aligned} \tag{2.18}$$

$$\begin{aligned} L' &= k_A * 2^{n-1} + y'_A + k_B * 2^{n-1} + y'_B \\ &= k'_L * 2^{n-1} + y'_L \; , \end{aligned} \tag{2.19}$$

$$D = antilog(L) = 2^{k_L} * (1 + \widehat{y_L}) \; , \tag{2.20}$$

$$D' = antilog(L') = 2^{k'_L} * (1 + \widehat{y'_L}) \; . \tag{2.21}$$

First, according to Equation 2.9 let us define $\widehat{m}$ as the normalized version of a generic mantissa $m$, so $\widehat{m}$ will always be $\in$ [0,1). Let $x_A$ and $x_B$ be the mantissas as defined in Algorithm 2, i.e. after left shifting. And let $y_A$ and $y_B$ be the mantissas after removing the leading one, i.e. the most significant bit, as Equations 2.10 and 2.11 indicate. And let $y'_A$ and $y'_B$ be the truncated version of those, as Equations 2.12 and 2.13 point, respectively. Let us define $L$

and $L'$ as Equations 2.18 and 2.19 indicate, respectively.

It must be noted that $k_L$ and $k'_L$ do not necessarily match, as if $y_L \neq y'_L$, the carry-out propagating towards the characteristic part of $L$ and $L'$ may be different too. Then, the error $E_D$ performed when computing $D'$ with respect to $D$ in Algorithm 2 is as pointed by Equation 2.22.

$$\begin{aligned} E_D = D - D' &= 2^{k_L} * (1 + \widehat{y_L}) - 2^{k'_L} * (1 + \widehat{y'_L}) \\ &= 2^{k_L} * ((1 + \widehat{y_L}) - 2^{k'_L - k_L} * (1 + \widehat{y'_L})) \ . \end{aligned} \tag{2.22}$$

As $k_L \geq k'_L$, because $L'$ comes from the truncated operands, two cases may happen then: either $k_L = k'_L$, or $k_L > k'_L$. If $k_L = k'_L$, then both $y_L$ and $y'_L$ propagate, or do not propagate, a carry-out to the characteristic parts $k_L$ and $k'_L$, respectively. Then, $y_L = y_A + y_B - carry$, $y'_L = y'_A + y'_B - carry'$, and Equation 2.23 holds.

$$\begin{aligned} E_D &= 2^{k_L} * ((1 + \widehat{y_L}) - (1 + \widehat{y'_L})) \\ &= 2^{k_L} * (\widehat{y_L} - \widehat{y'_L}) \\ &= 2^{k_L} * (\widehat{y_A} - \widehat{y'_A} + \widehat{y_B} - \widehat{y'_B}) \\ &< 2^{k_L} * (2^{-w+1} + 2^{-w+1}) = 2^{k_L - w + 2} \ . \end{aligned} \tag{2.23}$$

It must be noted that as Equation 2.12 indicates, $y_A - y'_A$ is lower than $2^{n-w}$, and then $\widehat{y_A} - \widehat{y'_A}$ is lower than $2^{-w+1}$. Analogously for Equation 2.13. On the other hand, if $k_L > k'_L$, then $k_L = k'_L + 1$. As the only difference between $L$ and $L'$ is a carry-out being propagated to the non-truncated part, there can only be one unit of difference in the characteristic part of $L$. Furthermore, if in $L$ such carry propagation exists, while in $L'$ does not, $\widehat{y'_L}$ will be composed of just '1's, and $\widehat{y_L}$ will be transformed into a chain of '0's. Under these conditions, Equation 2.24 holds.

$$\begin{aligned} E_D &= 2^{k_L} * ((1 + \widehat{y_L}) - 2^{k'_L - k_L} * (1 + \widehat{y'_L})) \\ &= 2^{k_L} * (1 - (1 + \widehat{y'_L})/2) \\ &= 2^{k_L} * (1 - (1 + \sum_{i=n-w+1}^{n-1} 2^{i-n})/2) \\ &= 2^{k_L} * (1 - (1 + 1 - 2^{-w+1})/2) \\ &= 2^{k_L} * (2^{-w}) = 2^{k_L - w} . \end{aligned} \tag{2.24}$$

Thus, $E_D$ is never larger than $2^{k_L-w+2}$. Or, in relative terms, it is $2^{k_L-w+2}/(2^{k_L} * (1 + \widehat{y_L})) = 2^{-w+2}/(1 + \widehat{y_L})$. In order to finally bound the error, in the worst case $\widehat{y_L} = 0$, so it can be concluded that the relative error, with respect to the conventional logarithmic multiplier, is always lower than $2^{-w+2}$. Hence, the shorter the $w$, the larger the error as expected.

### 2.2.3 Handling Negative Numbers with One's Complement (C1)

In this subsection, a proposal for handling the negative values will be described. The CNNs often involve negative weights and require the arithmetic of signed numbers. One problem with the prior design in Section 2.1 was that it did not handle signed numbers naturally, and required the 2's complements sign conversion (hereafter referred to as C2) before and after the design so that the inputs to the log multiplier are always positive. Each operand went through C2 if it was negative, and the output result also went through C2 depending on the expected output sign determined by the input signs. The inability to process signed numbers is found in many approximate multipliers, and the ones based on locating the leading one in the operands all suffer the same problem because the leading one is always placed at the leftmost bit for the negative numbers. The previous approaches either had assumed the sign-magnitude representation of signed numbers [45] or inserted C2 before and after their designs as we did [17, 58]. None of the prior works had evaluated the associated costs to fully investigate the issue.

Our proposal consists of leveraging the approximate computing scenario introduced by the CNNs. In this way, both inputs of the multiplier will be in two's complement. In this format there are several well-known facts:

- If $X \in \mathbb{Z}$, then $C2(X) = C1(X)+1$. Thus, if a number is negative, it can be approached with its one's complement (hereafter referred to as C1) at the expense of introducing an error of one *unit in the last place* (ulp).
- In C2, the sign of a number is given by the most significant bit (MSB). Thus, if $X \geq 0$ the MSB is '0', and on the contrary if $X < 0$, the MSB is '1'.

The proposed design is shown in Figure 2.9. As observed, first both operands are XOR-ed with their sign bit in order to convert the negative numbers into the positive domain by using the C1 transform. Consequently, the sign of the result will be the logical XOR between the operand signs. If such bit is '1', then the result provided by the antilogarithm block must be XOR-ed to produce a negative result. In this case, the zero handling is performed differently in comparison with the Mitchell multiplier in Section 2.1, although a similar idea is followed: the result is not zero if both operands are not zero. Let $k_{O+}$ be the characteristic of an operand $O$ after being converted to the positive domain. And let $msb_O$ (i.e. sign) and $lsb_O$ be the most and least significant bits of an operand $O$. Then, the conditions under which an operand is not zero are given by Table 2.3.

- If $k_{O+} > 0$, this means that the operand is not zero (cases 4 to 7).
- If $k_{O+} = 0$ and the operand is negative ($msb_O = 1$), the operand is not zero (cases 2 and 3).

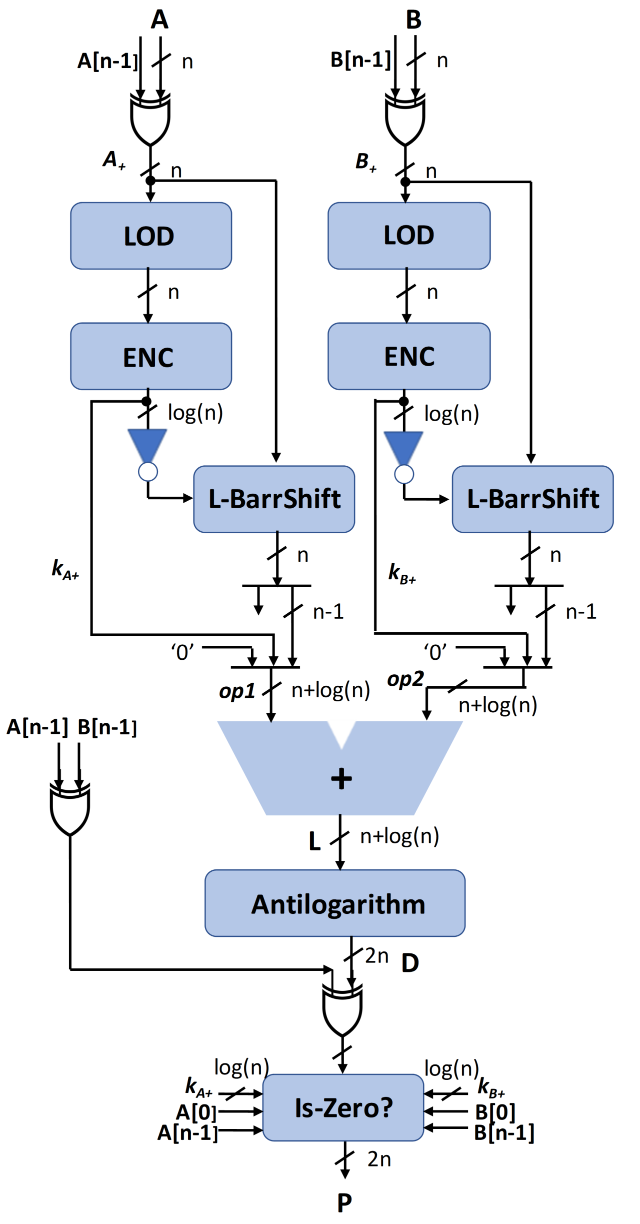


Figure 2.9: Mitch-$w$ with C1 negative number handling

Table 2.3: Truth table for determining whether an operand is zero or not

| $k_{O+} > 0$ | $msb_O$ | $lsb_O$ | $notZero_O$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

- If $k_{O+} = 0$ and the operand is positive ($msb_O = 0$), the operand is not zero if $lsb_O$ is 1 (case 1), and zero otherwise (case 0).

Therefore, the condition under which the result is not zero is as Equation 2.25 indicates.

$$\begin{aligned} &notZero_D = notZero_A \wedge notZero_B \\ &= (k_{A+} > 0 \vee msb_A \vee lsb_A) \wedge (k_{B+} > 0 \vee msb_B \vee lsb_B)\ . \end{aligned} \tag{2.25}$$

### 2.2.4 Error Study of C1 Sign Handling

The error produced by our approach when $A$ and $B$ have different signs and when $A$ and $B$ are negative is generally low. Intuitively, there is only 1 unit of difference between computing C1 or C2 of an operand. In this section, the error introduced by the proposed scheme to handle negative numbers will be studied in detail. Provided that the operands are not zero, the error occurs at the output of the antilogarithm block. Four cases may arise depending on the sign of both operands $A$ and $B$. If both are positive, there is no error in comparison to the conventional logarithmic multiplier.

**A and B possess different sign**

Without loss of generality, let us suppose that $A < 0$ and $B \geq 0$. If $A \geq 0$ and $B < 0$, the analysis is analogous. In this scenario, let us define the variables indicated by Equations 2.26 to 2.34.

$$y_L = (y_{C2(A)} + y_B) \wedge (2^{n-1} - 1) \ , \tag{2.26}$$

$$y'_L = (y_{C1(A)} + y_B) \wedge (2^{n-1} - 1) \ , \tag{2.27}$$

$$k_L = k_{C2(A)} + k_B + (((y_{C2(A)} + y_B) \wedge (2^{n-1})) >> (n-1)) \ , \tag{2.28}$$

$$k'_L = k_{C1(A)} + k_B + (((y_{C1(A)} + y_B) \wedge (2^{n-1})) >> (n-1)) \ , \tag{2.29}$$

$$\begin{aligned} L &= k_{C2(A)} * 2^{n-1} + y_{C2(A)} + k_B * 2^{n-1} + y_B \\ &= k_L * 2^{n-1} + y_L \ , \end{aligned} \tag{2.30}$$

$$\begin{aligned} L' &= k_{C1(A)} * 2^{n-1} + y_{C1(A)} + k_B * 2^{n-1} + y_B \\ &= k_L * 2^{n-1} + y'_L \ , \end{aligned} \tag{2.31}$$

$$P = C2(2^{k_L} * (1 + \widehat{y_L})) = C2(U) \ , \tag{2.32}$$

$$P'_{Neg1} = C1(2^{k'_L} * (1 + \widehat{y'_L})) = C1(V) \ , \tag{2.33}$$

$$E_{Neg1} = P - P'_{Neg1} \ . \tag{2.34}$$

$$\begin{aligned} E_{Neg1} = P' - P'_{Neg1} &= C2(U) - C1(V) \\ &= C2(U) - (C2(V) - 2^{-n+1}) = V - U + 2^{-n+1} \\ &= 2^{k'_L} * (1 + \widehat{y'_L}) - 2^{k_L} * (1 + \widehat{y_L}) + 2^{-n+1} \\ &= 2^{k_L} * (2^{k'_L - k_L} * (1 + \widehat{y'_L}) - (1 + \widehat{y_L})) + 2^{-n+1} \ . \end{aligned} \tag{2.35}$$

As in the prior theorem, two cases arise: $k_L = k'_L$ and $k_L > k'_L = k_L - 1$. If $k_L = k'_L$, then by definition of two's and one's complement $k_{C2(A)} \geq k_{C1(A)}$. If $k_{C2(A)} = k_{C1(A)}$ and $k_{C2(B)} = k_{C1(B)}$, the logical AND in the definition of $y_L$ and $y_{L'}$ has no effect and then Equation 2.36 holds. If $k_{C2(B)} > k_{C1(B)}$, the situation will be analogous as described by Equation 2.40.

$$\begin{aligned} E_{Neg1} &= 2^{k_L} * ((1 + \widehat{y'_L}) - (1 + \widehat{y_L})) + 2^{-n+1} \\ &= 2^{k_L} * (\widehat{y'_{C1(A)}} - \widehat{y'_{C2(A)}} + \widehat{y_B} - \widehat{y_B}) + 2^{-n+1} \\ &= 2^{k_L} * (\widehat{y'_{C1(A)}} - \widehat{y'_{C2(A)}}) + 2^{-n+1} \ . \end{aligned} \tag{2.36}$$

The difference between $\widehat{y_{C2(A)}}$ and $\widehat{y_{C1(A)}}$ is as studied in Equations 2.37 and 2.38, where the logical AND of Equations 2.26 and 2.27 has no effect.

$$\begin{aligned} y_{C2(A)} - y_{C1(A)} &= (C2(A) << \neg(k_{C2(A)})) \wedge (2^{n-1} - 1) \\ &\quad - (C1(A) << \neg(k_{C1(A)})) \wedge (2^{n-1} - 1) \\ &= (C2(A) - C1(A)) << n - k_{C1(A)} - 1 \\ &= 2^{n-k_{C1(A)}-1} \ , \end{aligned} \tag{2.37}$$

$$\widehat{y_{C2(A)}} - \widehat{y_{C1(A)}} = 2^{n-k_{C1(A)}-1}/2^{n-1} = 2^{-k_{C1(A)}} \ . \tag{2.38}$$

Introducing Equation 2.37 into Equation 2.36 it is possible to get Equation 2.39.

$$E_{Neg1} = 2^{k_L} * (-2^{-k_{C1(A)}}) + 2^{-n+1} \approx -2^{k_L - k_{C1(A)}} \ . \tag{2.39}$$

In this case, the relative error is $(-2^{k_L-k_{C1(A)}})/(2^{k_L}*(1+\widehat{y_L}))>-1/2^{k_{C1(A)}}$.

If $k_{C2(A)}=k_{C1(A)}+1$, then $C2(A)$ is a power of two and $y_{C2(A)}=0$, and $y_L$ does not propagate any carry. Thus the logical AND of its definition has no effect. On the other hand, if $k_L=k'_L$, then $k_{C2(B)}=k_{C1(B)}$ and $y'_L$ must propagate to compensate $k_{C2(A)}=k_{C1(A)}+1$, i.e. $y'_L=y_{C1(A)}+y_B-1$. In this scenario, Equation 2.40 holds.

$$\begin{aligned}E_{Neg1}&=2^{k_L}*((1+\widehat{y_{C1(A)}}+\widehat{y_B}-1)/2-(1+\widehat{y_B}))+2^{-n+1}\\&<2^{k_L}*((\widehat{y_{C1(A)}}+\widehat{y_B})-(1+\widehat{y_B}))+2^{-n+1}\\&\approx 2^{k_L}*(\widehat{y_{C1(A)}}-1)\ .\end{aligned}\tag{2.40}$$

Therefore, the relative error is $(2^{k_L}*(\widehat{y_{C1(A)}}-1))/(2^{k_L}*(1+\widehat{y_L}))=(\widehat{y_{C1(A)}}-1)/(1+\widehat{y_L})\rightarrow 0$. As $C2(A)$ is a power of two, $y_{C2(A)}$=0 and consequently $y_{C1(A)}$ will possess several '1's in the most significant positions. For instance, consider $n$=8 bits, $A$=-64, $C2(A)$=64 and then $k_{C2(A)}$=6, $y_{C2(A)}$=0 and $\widehat{y_{C2(A)}}$=0. On the other hand, $C1(A)$=63, $k_{C1(A)}$=5, $y_{C1(A)}$=120=11111000b and $\widehat{y_{C1(A)}}$=0.9375.

If $A\geq 0$ and $B<0$, similar expressions can be obtained.

### A and B are negative

In the scenario where $A<0$ and $B<0$, let us define the variables indicated by Equations 2.41 to 2.49.

$$y_L = (y_{C2(A)} + y_{C2(B)}) \wedge (2^{n-1} - 1) \ , \tag{2.41}$$

$$y'_L = (y_{C1(A)} + y_{C1(B)}) \wedge (2^{n-1} - 1) \ , \tag{2.42}$$

$$\begin{aligned} k_L &= k_{C2(A)} + k_{C2(B)} \\ &\quad + (((y_{C2(A)} + y_{C2(B)}) \wedge (2^{n-1})) >> (n-1)) \ , \end{aligned} \tag{2.43}$$

$$\begin{aligned} k'_L &= k_{C1(A)} + k_{C1(B)} \\ &\quad + (((y_{C1(A)} + y_{C1(B)}) \wedge (2^{n-1})) >> (n-1)) \ , \end{aligned} \tag{2.44}$$

$$\begin{aligned} L &= k_{C2(A)} * 2^{n-1} + y_{C2(A)} + k_{C2(B)} * 2^{n-1} + y_{C2(B)} \\ &= k_L * 2^{n-1} + y_L \ , \end{aligned} \tag{2.45}$$

$$\begin{aligned} L' &= k_{C1(A)} * 2^{n-1} + y_{C1(A)} + k_{C1(B)} * 2^{n-1} + y_{C1(B)} \\ &= k_L * 2^{n-1} + y'_L \ , \end{aligned} \tag{2.46}$$

$$P = 2^{k_L} * (1 + \widehat{y_L}) \ , \tag{2.47}$$

$$P'_{Neg2} = 2^{k'_L} * (1 + \widehat{y'_L}) \ , \tag{2.48}$$

$$E_{Neg2} = P - P'_{Neg2} = 2^{k_L} * ((1 + \widehat{y_L}) - 2^{k'_L - k_L} * (1 + \widehat{y'_L})) \ . \tag{2.49}$$

Again, two possible scenarios arise: either $k_L = k'_L$, or $k_L > k'_L$. If $k_L = k'_L$, then by definition of two's and one's complement $k_{C2(A)} \geq \ k_{C1(A)}$ and $k_{C2(B)} \geq \ k_{C1(B)}$. If $k_{C2(A)} = k_{C1(A)}$ and $k_{C2(B)} = k_{C1(B)}$, then the logical AND in Equations 2.41 and 2.42 have no effect and applying Equation 2.38 it is possible to prove that Equation 2.50 holds.

$$\begin{aligned} E_{Neg2} &= 2^{k_L} * (\widehat{y_{C2(A)}} - \widehat{y_{C1(A)}} + \widehat{y_{C2(B)}} - \widehat{y_{C1(B)}}) \\ &= 2^{k_L} * (2^{-k_{C1(A)}} + 2^{-k_{C1(B)}}) \ . \end{aligned} \tag{2.50}$$

In this case, the relative error would be $2^{k_L} * (2^{-k_{C1(A)}} + 2^{-k_{C1(B)}})/(2^{k_L} * (1 + \widehat{y_L})) < 2^{-k_{C1(A)}} + 2^{-k_{C1(B)}}$.

If $k_L = k'_L$ and $k_{C2(A)} > k_{C1(A)} = k_{C2(A)} - 1$, then $C2(A)$ is a power of two and then $y_{C2(A)} = 0$. In this scenario, $k_{C2(B)} = k_{C1(B)}$, as otherwise $(k_{C2(A)} + k_{C2(B)}) - (k_{C1(A)} + k_{C1(B)}) = 2$, and this cannot be compensated by a carry out propagated from $y'_L$ to $k'_L$. On the other hand, a difference of just 1 can be compensated if $y'_L$ propagates, and then $\widehat{y'_L} = \widehat{y_{C1(A)}} + \widehat{y_{C1(B)}} - 1$. Thus, Equation 2.51 holds.

$$\begin{aligned} E_{Neg2} &= 2^{k_L} * ((1 + \widehat{y_{C2(B)}}) - (1 + \widehat{y_{C1(A)}} + \widehat{y_{C1(B)}} - 1)) \\ &= 2^{k_L} * (1 - \widehat{y_{C1(A)}} + 2^{-k_{C1(B)}}) < 2^{k_L - k_{C1(B)}} \ . \end{aligned} \tag{2.51}$$

It must be noted that if $C2(A)$ is a power of 2, then $C1(A)$ and consequently $y_{C1(A)}$ will possess several '1"s in the most significant positions. Thus, $(1 - \widehat{y_{C1(A)}}) \rightarrow 0$. Hence, the relative error is around $2^{k_L - k_{C1(B)}}/(2^{k_L} * (1 + \widehat{y_L})) < 2^{-k_{C1(B)}}$. A similar expression can be found if $k_L = k'_L$ and $k_{C2(B)} > k_{C1(B)} = k_{C2(B)} - 1$.

If $k_L > k'_L$, several scenarios may happen depending on the relation between $k_{C2(A)}$ and $k_{C1(A)}$, and $k_{C2(B)}$ and $k_{C1(B)}$. If $k_{C2(A)} = k_{C1(A)}$ and $k_{C2(B)} = k_{C1(B)}$, then $y_L$ propagates a carry-out while $y'_L$ does not. Thus, $\widehat{y_L} = \widehat{y_{C2(A)}} + \widehat{y_{C2(B)}} - 1$, $k_L = k'_L + 1$ and Equation 2.52

holds.

$$\begin{aligned} E_{Neg2} &= 2^{k_L} * ((1 + \widehat{y_{C2(A)}} + \widehat{y_{C2(B)}} - 1) \\ &\quad - (1 + \widehat{y_{C1(A)}} + \widehat{y_{C1(B)}})/2)) \\ &= 2^{k_L} * (2^{-k_{C1(A)}} + 2^{-k_{C1(B)}} - 1/2) \\ &< 2^{k_L} * (2^{-k_{C1(A)}} + 2^{-k_{C1(B)}}) \ . \end{aligned} \tag{2.52}$$

In this case again the relative error is lower than $2^{-k_{C1(A)}} + 2^{-k_{C1(B)}}$. If $k_L > k'_L$, $k_{C2(A)} > k_{C1(A)} = k_{C2(A)} - 1$ and $k_{C2(B)} = k_{C1(B)}$, then $y_{C2(A)}$ is zero, so $y_L$ does not propagate a carry-out and neither does $y'_L$, otherwise $k_L = k'_L$. Thus, $k_L = k'_L + 1$. Hence, the logical AND in Equations 2.41 and 2.42 has no effect and Equation 2.53 holds.

$$\begin{aligned} E_{Neg2} &= 2^{k_L} * ((1 + \widehat{y_{C2(B)}}) - (1 + \widehat{y_{C1(A)}} + \widehat{y_{C1(B)}})/2)) \\ &= 2^{k_L - 1} * (1 + 2 * \widehat{y_{C2(B)}} - \widehat{y_{C1(B)}} - \widehat{y_{C1(A)}}) \\ &< 2^{k_L - 1} * (1 + \widehat{y_{C2(B)}} - \widehat{y_{C1(B)}} - \widehat{y_{C1(A)}}) \\ &= 2^{k_L - 1} * (1 + 2^{-k_{C1(B)}} - \widehat{y_{C1(A)}}) \\ &\approx 2^{k_L - k_{C1(B)} - 1} \ . \end{aligned} \tag{2.53}$$

In this case the relative error is lower than $(2^{k_L - k_{C1(B)} - 1})/(2^{k_L} * (1 + \widehat{y_L})) < 2^{-k_{C1(B)} - 1}$. If $k_{C2(B)} > k_{C1(B)} = k_{C2(B)} - 1$ and $k_{C2(A)} = k_{C1(A)}$, an analogous expression can be found.

The last case to be studied is $k_L > k'_L$, $k_{C2(A)} > k_{C1(A)} = k_{C2(A)} - 1$ and $k_{C2(B)} > k_{C1(B)} = k_{C2(B)} - 1$. In this scenario, $C2(A)$ and $C2(B)$ are power of two, so $y_{C2(A)}$, $y_{C2(B)}$ and $y_L$ are zero, while $y_{C1(A)}$ and $y_{C1(B)}$ will have several '1's in the most significant positions (at least one) so $y'_L$ will propagate a carry-out to $k'_L$ and then $k_L = k'_L + 1$. Thus, $\widehat{y'_L} = \widehat{y_{C1(A)}} + \widehat{y_{C1(B)}} - 1$ and Equation 2.54 holds.

$$\begin{aligned} E_{Neg2} &= 2^{k_L} * (1 - (1 + \widehat{y_{C1(A)}} + \widehat{y_{C1(B)}} - 1)/2)) \\ &= 2^{k_L} * (1 - (\widehat{y_{C1(A)}} + \widehat{y_{C1(B)}})/2)) \ . \end{aligned} \tag{2.54}$$

For the same reasons as in Section 2.2.4, $y_{C1(A)}$ and $y_{C1(B)}$ will have several '1's in the most significant positions and will be close to 1. Thus, $(\widehat{y_{C1(A)}} + \widehat{y_{C1(B)}})/2 \rightarrow\ 1$, and then the error shown in Equation 2.54 will tend to zero, and so will do the relative error.

**Corner cases**

Despite the generally low error when employing the one's complement, there are some special cases that need careful attention. In the prior proofs it has been considered that, without loss of generality, when $C2(A)$ is a power of two, then $y_{C2(A)}$ is 0 and consequently $C1(A)$ and $y_{C1(A)}$ will possess several '1's in the most significant positions. This is valid but for two special cases, namely: -1 and -2.

Following Figure 2.9, it is possible to observe that when $A = -1$, $C1(A) = A_+ = 0$, $k_{C1(A)} = k_{A_+} = 0$ and $op1 = 0$. Hence, multiplying by -1 works as adding the neutral element in the LNS domain. It must be noted, that according to Table 2.3 the result will not be converted to zero, despite the fact that $A_+ = 0$. Furthermore, if $B = -1$ and consequently $op2 = 0$, the antilogarithm will produce $D = 1$ anyway.

When $A = -2$, $C1(A) = A_+ = 1$, $k_{C1(A)} = k_{A+} = 0$ and $op1 = 0$ and $y_{C1(A)} = 0$. Hence, a similar situation arises, but with the drawback of producing a larger error, as we are actually multiplying by -2. Once again, if $B = -1$ or $B = -2$, the result given by the antilogarithm will be $D = 1$.

This situation is different when we have larger power of two numbers, because $k_{A+} > 0$ and there will be at least a MSB equal to '1' in $y_{C1(A)}$. For instance, $A = -4$, $C1(A) = A_{+} = 3$, $k_{A+} = 1$ and $y_{C1(A)} = 2^{n-2}$, i.e. $\widehat{y_{C1(A)}} = 0.5$.

**Error interpretation**

In order to summarize this section, it must be noted that all the errors depend on the one's complement of an operand, and as the operand is smaller in absolute terms, the error grows. In other words, the relative error is high for the smallest operands, but in absolute terms the difference is not high because the operands are small. In general, this will not have a large impact on the CNNs, as small results will not contribute considerably to the MAC output of each convolution and fully connected node. This claim is supported by the experiments and the analysis presented in Section 3.6.6.

### 2.2.5 Making Mitch-$w$ Unbiased

The proposed log multiplier only produces negative errors, which means that it produces approximated results that are equal to or less than exact multiplication. It never produces a result with positive error, which is larger than the correct result. Therefore, it has the biased error distribution that increases the values of the mean error and the worst case error (hereafter abbreviated as WCE). Thus, our method is to evaluate the effect of having a low mean error and the error distribution centered around zero. The $w$ truncation and approximate logarithm are the two sources of error in the proposed designs and both of them produce only negative errors. To create the unbiased designs and have the mean error closer to zero, both of the error sources must be addressed.

The entire error of DRUM multiplier comes from the truncation error and it uses rounding up to make the error unbiased as shown earlier in Figure 2.7b[17]. We adapted their technique in our Mitch-$w$ designs, so that the LSB of each adder operand is truncated and replaced with '1', which is equivalent to removing the least significant full adder and using a carry-in equal to '1'. Moreover, this technique slightly decreases the cost of other parts within the designs, as it truncates one bit and reduces the number of bits associated with the barrel shifters and the multiplexer within the antilogarithm block.

The detailed description of the approximate logarithm is presented in Section 2.1.1. The error from approximate logarithm can be unbiased by adding a constant mantissa to the added log result so that the linear approximation of the mantissa shown in Figure 2.1 is shifted up. This causes some results to have positive errors where the log curve is below the shifted linear approximation. We have empirically found with RTL simulations that adding the binary mantissa '0.0001' to the result of the addition makes the designs unbiased to have the small mean error of 0.4% for 16 and 32 bits.

### 2.2.6 Synthesis Results

To evaluate the energy and area savings of the proposed designs, we performed synthesis using Synopsys Design Compiler and compared it against the synthesis results of the exact fixed-point multiplier. The exact multiplier is automatically synthesized by Design Compiler from the simple Verilog multiplication. We used a 32nm digital standard cell library from Synopsys, and repeated the synthesis for 8, 16, and 32 bits. The synthesis was performed with "Ultra" effort at the clock frequency of 250 MHz. Critical path, area, and total power are reported by the Design Compiler, and energy is calculated from the critical path and power. The error values are obtained through RTL simulations using QuestaSim. The simulations are performed with 1,000,000 random input values and reported to the nearest 10th of a

Table 2.4: Comparison of the synthesis results of Mitch-$w$ designs

| | **N=8** | | | | | **N=16** | | | | | | **N=32** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Exact** | **Mitchell** | **Mitch-$w$** | | | **Exact** | **Mitchell** | **Mitch-$w$** | | | | **Exact** | **Mitchell** | **Mitch-$w$** | | | |
| | | | $w$**=5** | $w$**=6** | $w$**=7** | | | $w$**=5** | $w$**=6** | $w$**=7** | $w$**=8** | | | $w$**=5** | $w$**=6** | $w$**=7** | $w$**=8** |
| **Mean Err. (%)** | 0 | -3.8 | -6.5 | -4.7 | -4 | 0 | -3.8 | -7.9 | -5.9 | -4.9 | -4.4 | 0 | -3.9 | -7.9 | -5.9 | -4.9 | -4.4 |
| **PWCE (%)** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **NWCE (%)** | 0 | -11.1 | -17.3 | -13.8 | -12.0 | 0 | -11.1 | -18.0 | -14.6 | -12.9 | -12.0 | 0 | -11.1 | -18.0 | -14.7 | -12.9 | -12.0 |
| **Crit. Path** ($ns$) | 1.07 | 1.13 | 0.97 | 1.13 | 1.14 | 2.23 | 2.31 | 1.53 | 1.59 | 1.70 | 1.66 | 3.78 | 3.70 | 1.67 | 1.98 | 1.99 | 2.03 |
| **Area** ($um^2$) | 474 | 389 | 305 | 353 | 376 | 2032 | 1150 | 592 | 693 | 760 | 824 | 8627 | 2890 | 1314 | 1346 | 1496 | 1559 |
| **Tot. Pow.** ($mW$) | 0.27 | 0.20 | 0.15 | 0.17 | 0.18 | 1.24 | 0.55 | 0.26 | 0.31 | 0.34 | 0.38 | 6.02 | 1.41 | 0.47 | 0.53 | 0.58 | 0.62 |
| Energy ($pJ$) | 0.29 | 0.23 | 0.15 | 0.19 | 0.21 | 2.77 | 1.27 | 0.40 | 0.49 | 0.58 | 0.63 | 22.76 | 5.22 | 0.78 | 1.05 | 1.15 | 1.26 |
| Energy Savings | | 22% | 50% | 34% | 29% | | 54% | 86% | 82% | 79% | 77% | | 77% | 97% | 95% | 95% | 94% |

percent. All energy savings reported in the tables are in comparison to the exact fixed-point multiplier with the same number of bits.

Table 2.4 shows the comparison of the synthesis results of Mitch-$w$ designs, along with the error characteristics. The error values are relative to the results of exact multiplication, and PWCE stands for the positive worst case error while NWCE stands for the negative worst case error. The Mitchell multiplier refers to the design in Section 2.1 that did not apply the $w$ truncation. Mitch-$w$ multiplier saves significant energy and area compared to the exact fixed-point multiplier, and also improves significantly upon the Mitchell Log Multiplier in Section 2.1. Compared to the exact multiplier, our parametrized design at $w$=8 can save up to 94% of energy at 32-bits, and have potentially more cost savings with smaller $w$. It is important to note that the $w$ customizable truncation results in a lot of additional energy savings compared to the original Mitchell log multiplier without substantial increase in error.

The Mitchell multiplier had the critical path that was comparable with the exact fixed-point multiplier. With the introduction of the $w$ truncation, we reduce the critical path lengths of the designs, as the adders and the barrel shifters get simplified. We see the highest benefits in the 32-bit designs, where the critical path reduces as much as 46%, almost doubling the potential processing speed and energy reduction.

Table 2.5: Synthesis results of the unbiased designs. Energy savings are calculated with respect to the exact multiplier

| | **N=16** | | | | **N=32** | | | |
|---|---|---|---|---|---|---|---|---|
| | **Original** | | **Unbiased** | | **Original** | | **Unbiased** | |
| | $w$**=6** | $w$**=8** | $w$**=6** | $w$**=8** | $w$**=6** | $w$**=8** | $w$**=6** | $w$**=8** |
| **Mean Err. (%)** | -5.9 | -4.4 | 0.4 | 0.4 | -5.9 | -4.4 | 0.4 | 0.4 |
| **PWCE (%)** | 0 | 0 | 12.4 | 7.7 | 0 | 0 | 12.4 | 7.7 |
| **NWCE (%)** | -14.6 | -12.0 | -11.1 | -8.2 | -14.7 | -12.0 | -11.1 | -8.2 |
| **Critical Path** ($ns$) | 1.59 | 1.66 | 1.81 | 1.98 | 1.98 | 2.03 | 1.93 | 2.45 |
| **Area** ($um^2$) | 693 | 824 | 660 | 819 | 1346 | 1559 | 1394 | 1556 |
| **Tot. Power** ($mW$) | 0.31 | 0.38 | 0.30 | 0.38 | 0.53 | 0.62 | 0.51 | 0.63 |
| Energy ($pJ$) | 0.49 | 0.63 | 0.54 | 0.75 | 1.05 | 1.26 | 0.98 | 1.54 |
| Energy Savings | 82% | 77% | 80% | 73% | 95% | 94% | 96% | 93% |

Table 2.6: Synthesis results of the signed designs. Energy savings are calculated with respect to the exact multiplier

| | **N=8** | | | **N=16** | | | | | | **N=32** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Unsigned** | **C2** | **C1** | **Unsigned** | | **C2** | | **C1** | | **Unsigned** | | **C2** | | **C1** | |
| | $w$**=6** | $w$**=6** | $w$**=6** | $w$**=6** | $w$**=8** | $w$**=6** | $w$**=8** | $w$**=6** | $w$**=8** | $w$**=6** | $w$**=8** | $w$**=6** | $w$**=8** | $w$**=6** | $w$**=8** |
| **Crit. Path** ($ns$) | 1.21 | 1.99 | 1.32 | 1.81 | 1.98 | 3.84 | 3.69 | 1.75 | 1.90 | 1.93 | 2.45 | 3.86 | 3.86 | 2.19 | 2.50 |
| **Area** ($um^2$) | 380 | 624 | 483 | 660 | 819 | 1236 | 1287 | 922 | 1135 | 1394 | 1556 | 3028 | 3168 | 1815 | 2092 |
| **Tot. Power** ($mW$) | 0.18 | 0.33 | 0.26 | 0.30 | 0.38 | 0.60 | 0.70 | 0.50 | 0.61 | 0.51 | 0.63 | 1.52 | 1.64 | 0.90 | 1.08 |
| Energy ($pJ$) | 0.22 | 0.66 | 0.34 | 0.54 | 0.75 | 2.30 | 2.58 | 0.88 | 1.16 | 0.98 | 1.54 | 5.87 | 6.33 | 1.97 | 2.70 |
| Energy Savings | 25% | -127% | -19% | 80% | 73% | 17% | 7% | 68% | 58% | 96% | 93% | 74% | 72% | 91% | 88% |

Table 2.5 shows the synthesis results for the original and unbiased versions of Mitch-$w$ as described in Section 2.2.5. The unbiased designs have the positive errors as well as the negative errors, and the error ranges are slightly increased while the mean errors and the WCE are decreased. Adding the constant to the result of the adder within the log multiplier introduced the additional circuitry that increased the critical path length for all designs except $w$=6 at N=32.

Table 2.6 shows the synthesis results of the proposed designs that include the C2 and C1-based signed number handling. The error characteristics are omitted in this table because the corner cases produce large errors that make the comparison difficult. Table 2.6 shows that the power and area costs of C2 negative number handling are significant, but the C1 approximation reduces the costs considerably.

The C2 conversions also add significant delays to the critical path and make the designs slower. The C1 approximation removes the step of adding one, and the compiler performed better optimizations so that the critical path lengths of the C1-based designs are significantly shortened, even shorter than the unsigned version for 16 bits. The signed designs with C1 perform faster than the exact fixed-point multipliers, while the C2 designs do not. Combined with power, the C1-based designs are significantly more energy-efficient than the C2 counterparts.

### 2.2.7 Cost Comparison against Bfloat16

The bfloat16 format [66] reduces significant amount of hardware costs compared to the FP32 floating-point format and has been widely adopted in Machine Learning hardware accelerators. While its ease of use and the ability to perform training as well as inference are undeniably advantageous, its arithmetic units are slower and consume more energy compared to the discussed multipliers based on the fixed-point format. It is plausible to have a use-case scenario where embedded systems perform only CNN inferences under strict design constraints, while communicating to datacenters where training occurs. This section presents a brief comparison of the hardware costs against a bfloat16 MAC unit to give an idea of the potential benefits of the approximate log multiplication.

Table 2.7 shows the comparison of the costs between the MAC units of FP32, bfloat16 and the Mitch-$w$, as synthesized with a 32nm standard library from Synopsys. The Mitch-$w$6 HDL code is available in [10], the FP32 MAC design is from [11], and we modified the FP32 design to create the bfloat16 MAC. Synopsys Design Compiler automatically synthesized the fixed-point MAC, and Mitch-$w$6 is followed by an exact fixed-point adder. It is clear from

Table 2.7: Hardware costs of FP32, bfloat16, and Mitch-$w$6 MAC

| | N=16 | | | N=32 | | |
|---|---|---|---|---|---|---|
| | **bfloat16** | **Fixed** | **Mitch-$w$6** | **FP32** | **Fixed** | **Mitch-$w$6** |
| **Delay (ns)** | 4.77 | 2.07 | 2.74 | 7.52 | 4.29 | 4.39 |
| **Power (mW)** | 1.47 | 1.17 | 0.50 | 5.80 | 4.36 | 0.98 |
| **Energy (pJ)** | 7.01 | 2.42 | 1.37 | 43.62 | 18.70 | 4.30 |
| **Energy vs. bfloat16** | 100% | 35% | 20% | 622% | 267% | 61% |

Table 2.7 that applying approximate multiplication to CNNs can save significant amount of resources for inferences.

The presented figures do not consider the potential benefits when adopting multiple log multipliers, where additional optimization for resource sharing can be performed depending on the design of hardware accelerator. Chapter 4 proposes that certain parts of the log multiplier can be removed or shared between multiple instances of MAC units depending on the accelerator design.

## 2.3 Truncated Iterative Log Multiplier

While the experiments in Section 3.6 and 3.7 show that the previously proposed designs perform well for many CNN models, the CNN architectures that use depthwise separable convolution require a more accurate approximate multiplier design (see Section 3.4). The iterative logarithmic structure proposed in [4] has the capability to enhance the accuracy of Mitchell Log Multiplier, but repeating basic blocks adds significant costs to the design. There is a strong need to reduce the cost of basic blocks so that the iterative logarithmic multiplication becomes affordable. This section proposes a low-cost two-stage approximate log multiplication for CNNs, where each stage adopts Mitch-$w$. We present the detailed description of the proposed two-stage multiplier, including the techniques to convert Mitch-$w$ into an iterative design and transfer error terms between stages. The proposed design and its performance on CNNs were published in [28].

### 2.3.1 Preliminaries

In Mitchell's Algorithm presented in Section 2.1.1, an $n$-bit unsigned integer number $A$ can be expressed as:

$$A = (1 + x_A) \cdot 2^{k_A}, x_A \in [0, 1). \tag{2.55}$$

In (2.55), $k_A$ is the characteristic for indicating the location of the MSB with the value of '1', and $x_A$ is the mantissa of $A$. When $C = (1 + x_C) \cdot 2^{k_C}$ and $C = A \cdot B$, the logarithmic conversion is approximated as follows:

$$\begin{cases} k_C = k_A + k_B + 1, \\ x_C = x_A + x_B - 1, \\ \text{ if } x_A + x_B \geq 1 \\ \\ k_C = k_A + k_B, \\ x_C = x_A + x_B, \\ \text{ if } x_A + x_B < 1. \end{cases} \tag{2.56}$$

On the other hand, the theory of iterative log multiplier is presented in [4], and leverages Equation 2.57 where $P_{true}$ is the exact product and $C(1)$ is the product from the first log multiplication.

$$P_{true} = C(1) + (A - 2^{k_A}) \cdot (B - 2^{k_B}) \tag{2.57}$$

This means that the error from log multiplication is equal to the product of the operands with the leading ones removed, which corresponds to the product of the mantissas. The correction terms to improve accuracy can be iteratively generated by performing log multiplication on the mantissas from the previous stage. For the two-stage iterative log multiplier, the product from approximate multiplication is described by Equation 2.58, where $C(2)$ is the correction term from performing log multiplication on $(A - 2^{k_A}) \cdot (B - 2^{k_B})$. This correction term significantly improves the accuracy of log multiplication [4].

$$P_{approx} = C(1) + C(2) \tag{2.58}$$

Lastly, the definition of relative error, $rerr$ is described.

$$rerr = \frac{MUL_{exact} - MUL_{appr}}{MUL_{exact}}. \tag{2.59}$$

$MUL_{exact}$ denotes the product from exact multiplication and $MUL_{appr}$ is the result of approximate log multiplication, given the same input.

### 2.3.2 Proposed Design

Figure 2.10 shows the structure of the proposed $n$-bit two-stage logarithmic multiplier. The first stage works similarly to the unbiased Mitch-$w$. The characteristics of the operands are obtained by the LOD and ENC blocks, and the truncated mantissa values are shifted according to the characteristics. The product from the first stage provides $C(1)$. The difference is that another Mitch-$w$ multiplier is connected as the second stage to produce the correction term and increase accuracy.

Following Equations 2.56 and 2.57, the error terms from the first stage depend on the range of $x_{A(1)} + x_{B(1)} + 2^{-n_1}$. As shown in Figure 2.10, two *Error Term Calculator*s calculate the error terms $A(2)$ and $B(2)$, where the carry-out from the mantissa adder indicates the range

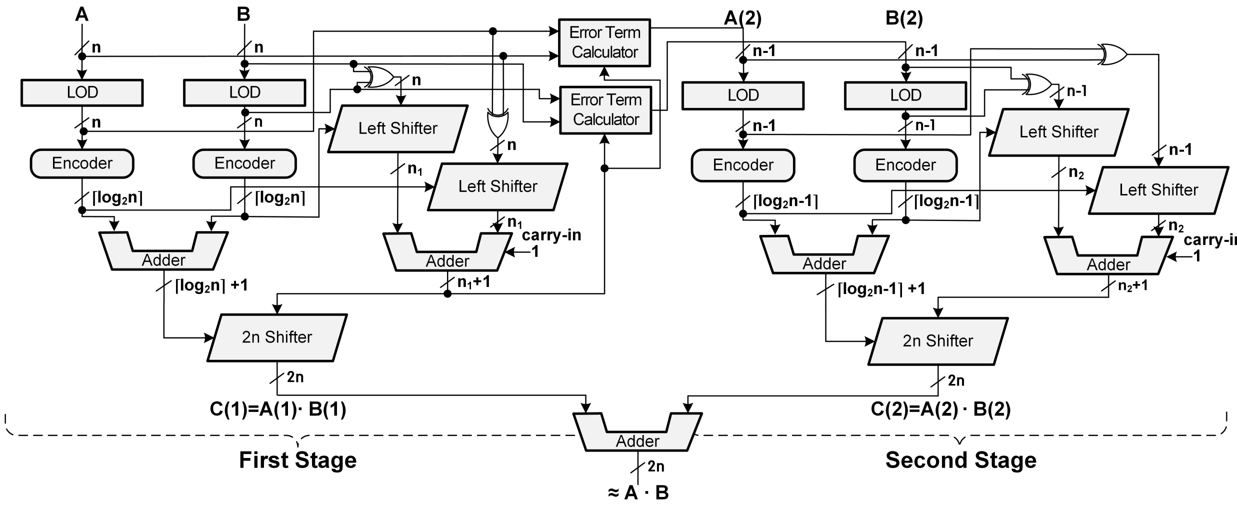


Figure 2.10: Structure of the proposed two-stage logarithmic multiplier.

of $x_{A(1)} + x_{B(1)} + 2^{-n_1}$. Hence, $A(2)$ and $B(2)$ are formulated in (2.60) as:

$$\begin{cases} A(2) = (1 - x_A) \cdot 2^{k_A} - 1, \\ B(2) = (1 - x_B) \cdot 2^{k_B} - 1, \\ \text{ if } x_{A(1)} + x_{B(1)} + 2^{-n_1} \geq 1 \\ \\ A(2) = x_A \cdot 2^{k_A}, \\ B(2) = x_B \cdot 2^{k_B}, \\ \text{ if } x_{A(1)} + x_{B(1)} + 2^{-n_1} < 1. \end{cases} \tag{2.60}$$

In the second stage, another Mitch-$w$ receives the $(n-1)$-bit error terms from the first stage to calculate $C(2)$. In the right part of Figure 2.10, two left shifters output $n_2$ bits, where $n_2 < n - 1$. Finally, $C(1)$ and $C(2)$ from the first and second stages are summed to generate the approximate product for $A \cdot B$. In addition to the zero detection on the final product, the zero detection is performed after the first stage to achieve accurate zero detection for the iterative multiplier. The zero detection units are omitted in Figure 2.10 for readability.

### 2.3.3 Error Study

Compared to the original iterative log multiplier, additional errors are introduced because of the truncation and unbiasing techniques of Mitch-$w$. The truncated mantissas of $A$ and $B$ are denoted as $x_{A_{n_1}}$ and $x_{B_{n_1}}$ when only $n_1$ high-order bits in the mantissas (equivalent to $w-1$) are used. When $n_1$ and $n_2$ high-order bits are adopted in the mantissas of the first and second stages, the product is formulated as:

$$P_{approx} = (x_{A_{n_1}} + x_{B_{n_1}} - 2^{-n_1}) \cdot 2^{k_A+k_B} + (x_{A_{n_2}} + x_{B_{n_2}} - 2^{-n_2}) \cdot 2^{k_{A(2)}+k_{B(2)}} \tag{2.61}$$

The simulations are performed with Equations 2.61 and 2.59 to study the error of this multiplier. Table 2.8 summarizes the relative errors for various $n$ and $n_1$ values, given $n_2 = 2$. The term $rerr_{min}$ stands for the minimum $rerr$, and $rerr_{max}$ stands for the maximum. Average and average absolute relative errors are denoted as $rerr_{avg}$ and $|rerr|_{avg}$ respectively, and one million pairs of inputs were randomly generated to obtain $rerr_{avg}$ and $|rerr|_{avg}$. $rerr_{min}$ can be negative because of the unbiased Mitch-$w$ is used, and $|rerr|_{avg}$ is somewhat larger than $rerr_{avg}$.

### 2.3.4 Synthesis Results

The proposed design is synthesized and its power and area are compared against other multipliers. The synthesis is performed using Synopsys Design Compiler and the 32nm standard cell library from Synopsys, where "Ultra" mode is used with the timing constraint of 100 MHz. The synthesis results as well as the relative errors are compared in TABLE 2.9. for $n = 8$, $n = 16$, and $n = 32$. A smaller value of $n_1$ is used for $n = 8$, because $n$ is already small. The multipliers used for comparison are the exact radix-4 Booth multiplier (*Booth*), Mitchell Log Multiplier [47] (*MM*), and the two-stage iterative log multiplier [3] (*IM*).

Table 2.8: Summary of relative errors

| $n$ | $n_1$ | $rerr_{max}$ | $rerr_{min}$ | $rerr_{avg}$ | $\|rerr\|_{avg}$ |
|---|---|---|---|---|---|
| 8 | 4 | 11.1% | -6.25% | -1.09% | 1.77% |
|  | 5 | 11.1% | -3.33% | -0.83% | 1.11% |
|  | 6 | 11.1% | -2.50% | -0.58% | 0.76% |
|  | 7 | 11.1% | -2.08% | -0.32% | 0.57% |
|  | 8 | 11.1% | -1.88% | -0.06% | 0.44% |
| 16 | 4 | 11.1% | -6.25% | 0.10% | 1.44% |
|  | 5 | 11.1% | -3.33% | 0.11% | 0.77% |
|  | 6 | 11.1% | -2.50% | 0.11% | 0.46% |
|  | 7 | 11.1% | -2.08% | 0.12% | 0.33% |
|  | 8 | 11.1% | -1.88% | 0.12% | 0.28% |
| 32 | 4 | 11.1% | -6.25% | 0.11% | 1.44% |
|  | 5 | 11.1% | -3.33% | 0.12% | 0.77% |
|  | 6 | 11.1% | -2.50% | 0.12% | 0.46% |
|  | 7 | 11.1% | -2.08% | 0.13% | 0.33% |
|  | 8 | 11.1% | -1.88% | 0.13% | 0.28% |

Table 2.9: Comparison of relative errors and costs

| $n$ | design | $rerr_{max}$ (%) | $rerr_{avg}$ (%) | critical path ($ns$) | area ($um^2$) | power ($uW$) |
|---|---|---|---|---|---|---|
| 8 | Booth[a] | 0 | 0 | 1.3 | 613 | 403 |
|  | MM[b] | 11.11 | 3.76 | 1.3 | 446 | 217 |
|  | IM[c] | 6.25 | 0.83 | 1.9 | 1,133 | 590 |
|  | PROP [d] | 11.11 | -1.09 | 2.6 | 786 | 370 |
| 16 | Booth[a] | 0 | 0 | 2.8 | 2,507 | 1,760 |
|  | MM[b] | 11.11 | 3.85 | 2.3 | 1,168 | 602 |
|  | IM[c] | 6.25 | 0.99 | 3.7 | 2,901 | 1,410 |
|  | PROP [e] | 11.11 | 0.11 | 5.1 | 1,638 | 739 |
| 32 | Booth[a] | 0 | 0 | 5.4 | 10,139 | 6,750 |
|  | MM[b] | 11.11 | 3.85 | 4.2 | 3,418 | 1,640 |
|  | IM[c] | 6.25 | 0.99 | 6.5 | 7,674 | 3,680 |
|  | PROP [e] | 11.11 | 0.12 | 7.9 | 3,102 | 1,370 |

[a] Radix-4 Booth multiplier
[b] Mitchell multiplier [47]
[c] Two-stage Babic's iterative multiplier [3]
[d] Proposed two-stage multiplier with $n_1 = 4, n_2 = 2$
[e] Proposed two-stage multiplier with $n_1 = 6, n_2 = 2$

The design at $n = 8$ does not show a significant improvement over the exact Booth multiplication, because Mitchell's Algorithm is more effective with a larger number of bits. For $n = 16$ and $n = 32$, the proposed design shows significant power and area savings, up to 58.0% and 79.7% respectively. The two-stage structure of the proposed design made its costs greater than those of Mitchell Log Multiplier for $n = 8$ and $n = 16$. However, the average relative error was significantly reduced. For $n = 32$, the proposed design consumes slightly less resources than Mitchell Log Multiplier. Compared to the two-stage iterative multiplier in [3], the proposed design demonstrates significant reduction in area and power. The critical path is increased in the proposed design, because the error terms from the first stage are determined by the sum of fractions. A pipeline between the first and second stages can be implemented to reduce the delay when necessary.

# Chapter 3

# Effects of Approximate Multiplication on Convolutional Neural Networks

This chapter evaluates the viability of the proposed multipliers for CNN inferences and studies the effects of approximate multiplication. While the proposed multipliers in Chapter 2 consume significantly less resources compared to exact multipliers, they produce varying amounts of errors in their products. The effects of these errors must be understood so that the proposed multipliers can be effectively used for CNN inferences.

## 3.1 Introduction to Convolutional Neural Networks

A brief introduction to CNNs is presented in this section. Figure 3.1 shows an example of a CNN. CNNs consist in many convolution layers followed by the fully connected layers that produce probabilistic predictions for the possible CNN outputs. As the study of CNNs matured, the size of CNNs increased to achieve better prediction accuracy, and showed the trend where the number of convolution layers increased while only one or two fully connected layers are used as classifiers. As an example, Figure 3.2 demonstrates the trend where the accuracy of CNNs increased with the size of CNNs for face image recognition [59]. In some networks, the fully connected layers are replaced by 1x1 convolutions which are mathematically equivalent.

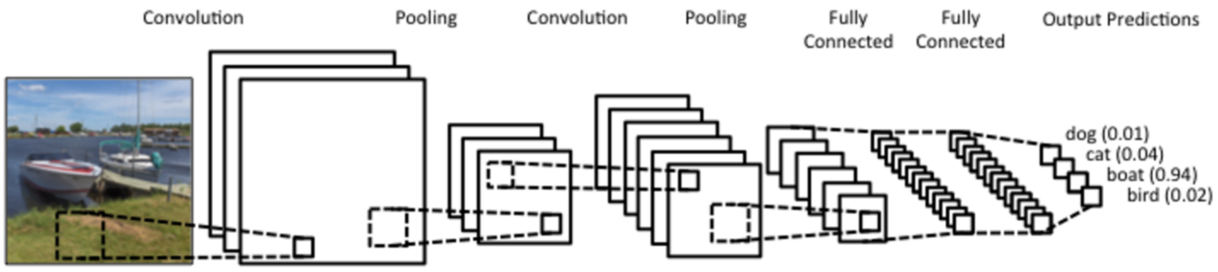


Figure 3.1: An example of a simple CNN

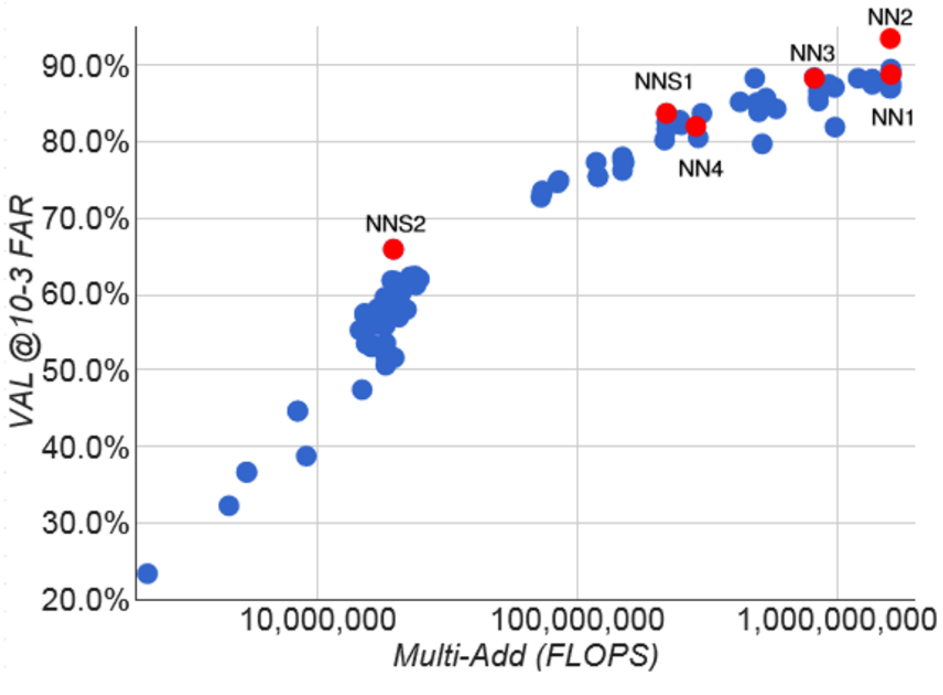


Figure 3.2: The correlation between the accuracy and size of CNNs [59]

A CNN is suitable for image recognition, because the convolution layers leverage the spatial locality of images to detect abstract features. The spatial locality means that the pixels that are closer together are more likely to form immediate features together than the pixels that are far apart, and this is leveraged by the kernels of fixed size that scan each location of input images to detect the patterns that match the kernels. A convolution layer performs convolution operations between the input channels and the corresponding kernels, where each input channel contains either the color pixels of an image or an abstract feature map from the previous convolution layer. Another dimension is created as the convolution layers have multiple output channels that have different sets of kernels to detect different abstract features. An example of convolution operation is depicted in Figure 3.3, and the mathematical description of convolution is discussed in Section 3.4. The convolution multiplies the input values with the corresponding kernel values and accumulates the products so that a location that matches the kernel pattern has a high output value that represents the detected feature. A convolution layer is commonly followed by a pooling layer where the spatial size of the feature maps is reduced to decrease the amount of computation while increasing generality. Repeating convolution and pooling in a CNN reduces input images with large dimensions to smaller abstract feature representations.

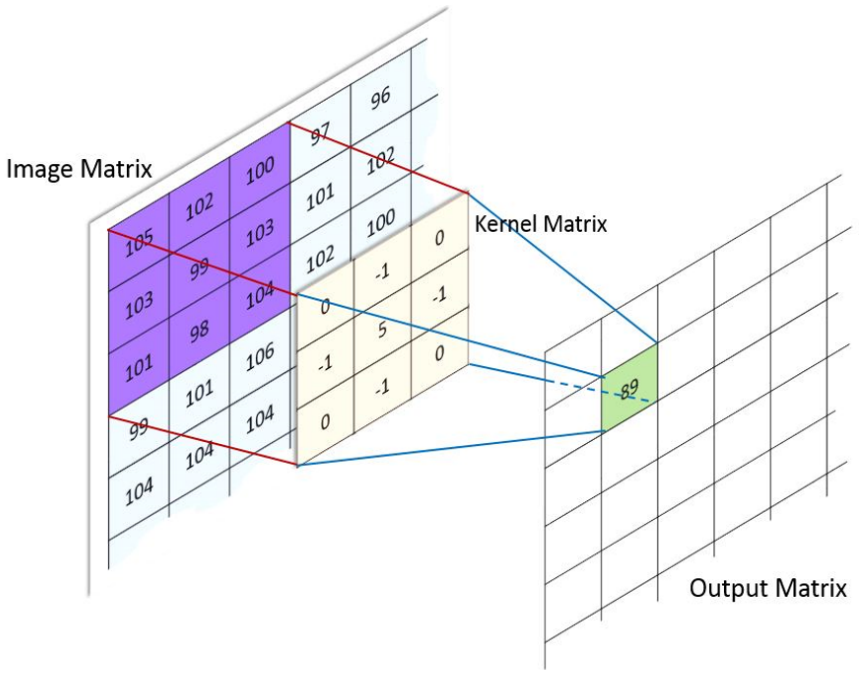


Figure 3.3: An example of convolution operation

A fully connected layer is conceptually the same as a layer of a multilayer perceptron (MLP), which is a more traditional class of feed-forward artificial neural network. A fully connected layer has one or more artificial neurons, and Figure 3.4 shows such a neuron. A fully connected layer does not have a kernel that leverages spatial locality, but instead every neuron in a layer is connected to all inputs of the layer. Each connection to an input is assigned a weight, and the products between the inputs and weights are summed with a bias value to produce an input to the activation function.

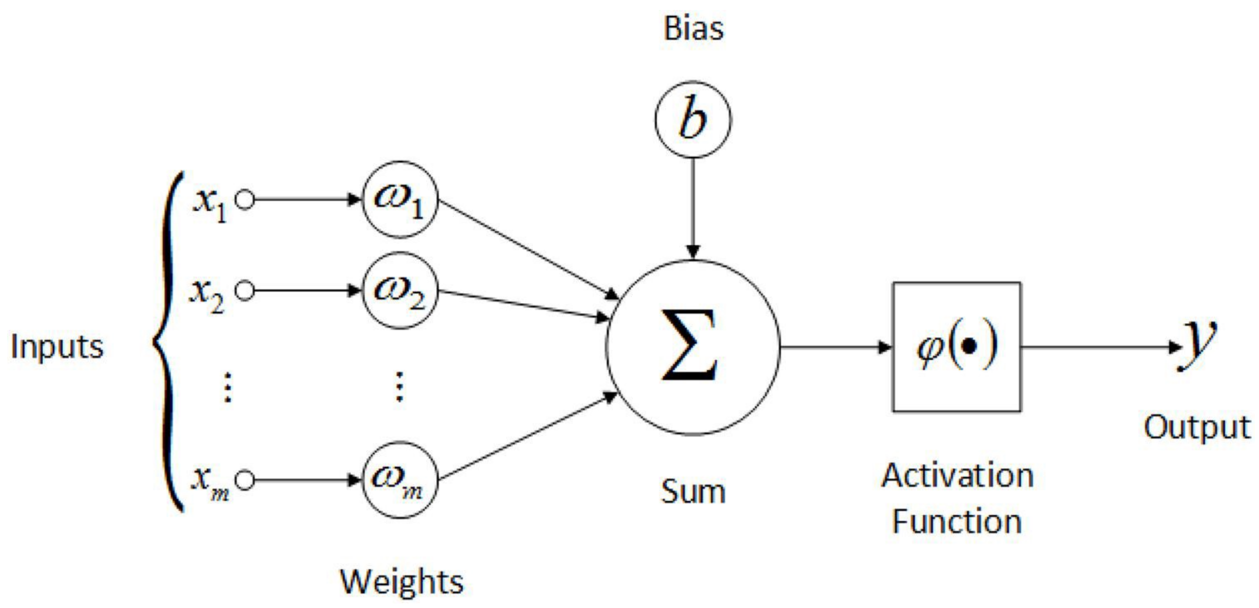


Figure 3.4: A neuron in a fully connected layer

The convolution and fully connected layers are commonly followed by the activation functions which provide nonlinear transformations. Because the convolution and fully connected layers perform linear MAC operations, simply connecting them successively results in a linear representation that can be reduced to a single matrix. The activation functions provide nonlinearity between the layers in order to prevent this reduction and expand the codomain of CNNs. The sigmoid activation functions were popular in early neural networks, but the ReLU activation function shown in Figure 3.5 became very popular in contemporary CNNs because of its low computation cost, strength against the vanishing gradient problem, and practically better convergence during training [33]. All CNNs discussed in this dissertation use the ReLU activation functions.

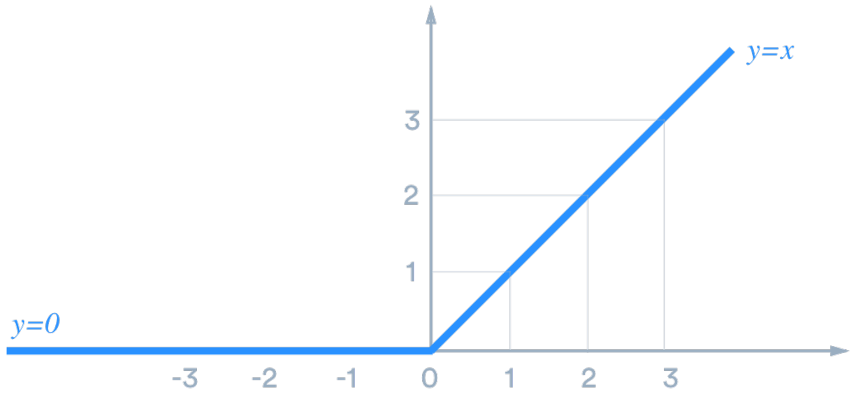


Figure 3.5: ReLU activation

There are other types of layers employed in CNNs such as normalization and loss layers, but their functions will not be discussed here because the proposed methodology is to perform the inference of convolution and fully connected layers with approximate multiplication. The only layer relevant to this dissertation is batch normalization [22], and it is briefly described in Section 3.5.

## 3.2 Approximate Multiplication for CNN Inferences

From Section 3.1, it is important to note that the convolution layers of a CNN consist in many MAC operations. In each convolution layer, the convolution operations are performed across the dimensions of width, height, input channels, and output channels, requiring large amount of computations for the given amount of input. Based on the observation that the convolution layers are the most computationally expensive layers in CNN inferences [54, 69], and that these layers perform large amount of expensive multiplications, the proposed methodology performs the multiplications in convolution with the proposed approximate multipliers. Because the optimization occurs at the circuit-level, the proposal does not require any modification of CNN architecture and can be generally applied to many network models.

The fully connected (FC) layers of CNNs also have MAC operations, but they have fewer computations compared to convolution [54]. The methodology still applies to approximate multiplication of FC layers to be consistent with the networks that use 1x1 convolution for classifiers. The effect of approximating FC layers is minimal because of the reasons discussed in Section 3.4. The operations in batch normalization are not approximated, because they can be absorbed into neighboring layers during inferences [40].

It is important to understand the difference between the method of quantization and the approximate multiplication. Quantization is the process of converting floating-point values in the CNN models to fixed-point for more cost-efficient inferences in the hardware [40]. The goal of quantization is to find the minimum number of fixed-point bits that can sufficiently represent the distribution of values, and there are some approximations with small number of fixed-point bits that cannot match the range and precision of the floating-point format. The error from this approximation depends on the network models as each has different distributions of values [25, 37]. The network dependency is the reason why more complex networks require a greater number of bits and the benefits of aggressive quantization diminish.

Approximate multiplication is less dependent on the networks because its source of error is from the approximation methods, not the lack of range and precision. Given proper quantization, approximate multiplication further minimizes the cost of multiplier for the given number of bits. Approximate multiplication is an orthogonal approach to quantization, because an approximate multiplier can be designed for any number of bits. Combining quantization with approximate multiplication may reduce significant amount of computational costs while providing better accuracy compared to the aggressive quantization that trades off CNN accuracies to work with very small number of bits.

## 3.3 Related Works

There have been many works on the approximate computing and the efficient processing of CNN inferences, and the scope of relevancy is narrowed to discuss the papers that specifically applied approximate multiplier designs to neural networks.

The works presented in [50, 13, 2, 51, 9] had used logic minimization to create the suitable approximate multipliers for each network model. They generated various approximate multipliers from an exact multiplier by intentionally flipping bits to reduce logic, and they require a large design space exploration to find the optimal solutions for each CNN model. It is difficult to create a hardware accelerator based on these approaches, because it requires to process many different instances of CNNs. On the other hand, their method is a gate-level approximation and can be considered complementary to our algorithm-level approximation. Our technique has the benefit of being independent of technology, can be easily scaled for the number of bits, and does not require searching the design space.

Sarwar et al. proposed applying the Alphabet Set Multiplier to improve energy consumption and area, at the cost of small degradation in neural network performance [58]. This multiplier design precomputed multiples of the multiplier values as alphabets and used shifting and adding of these values to approximate the product. They found that they could eliminate the pre-computing of the alphabets and use only the multiplier value as the single alphabet to eliminate costly memory accesses, at the cost of 2.83% drop in network accuracy in the MNIST dataset. This may be an acceptable drop in some cases, but it may be a significant drop for an application that requires near-perfect accuracy. They also found that the CNN accuracy drop was amplified as the applications became more complex, and the experiments performed in Section 3.6 revealed that the design was too inaccurate to handle the more complex dataset of ImageNet.

Hammad et al. [15] applied various approximate multipliers with varying accuracies to the

VGG network, and it was another evidence that approximate multiplication was compatible with CNN inferences. Their work included interesting experimental results that support our hypothesis in 3.4. They found that approximating the convolution layers with higher widths resulted in less degradation of CNN accuracy, and it agrees with our finding that variance of accumulated error decreases with more inter-channel accumulations.

The application of logarithmic multiplication to neural networks had been studied in [45, 35]. They used the 2-stage pipelined iterative logarithmic multiplication presented in [4], and the authors chose the iterative design over the original Mitchell's Algorithm because it had lower error rate and provided the opportunity for pipelining. They also claimed that the original Mitchell's Algorithm did not have large reductions of power and area compared to their design. This dissertation takes a different direction where we optimize Mitchell's Algorithm implementation and add the approximation techniques to have a significant power and area savings compared to the iterative design, and demonstrate that our design performs as well as theirs on CNNs despite the higher error. Also, the iterative log multipliers were mostly effective at performing CNN inferences, but the reason for the good performances largely remained a mystery. This chapter provides deeper understanding of the effects of approximate log multiplication on CNNs.

The log multipliers should be distinguished from the log quantization presented in [39, 48]. Unlike approximate multiplication that seeks to optimize an operation, the fundamental goal of log quantization is to optimize the quantization of CNNs. The log quantization performs all operations in the log domain and suffers from inaccurate additions, which may explain why the performances drop for more complex networks. The Mitchell's Algorithm still performs exact additions in the fixed-point format, which helps maintain the CNN performances as discussed in Section 3.4.

The most significant contributions of this dissertation compared to these previous works can be summarized as follows:

- Designing approximate multipliers that show large improvements over the state-of-the-art (See Section 3.6.7).
- Identifying the reasons why approximate multiplication is viable for CNN inferences.
- Extending the methodology of approximate multiplication to very deep CNNs with batch normalization.
- Designing a convolution core with approximate multipliers and demonstrating the large benefits of approximate multiplication on CNN accelerators.

There are many other ways of approximating multiplication that had not been applied to deep CNNs, such as [49, 36, 41, 34, 52, 68, 43, 56, 65, 21] among countless others. While we believe that the studied multiplier designs are the most promising, there are most likely other related opportunities for improving CNNs.

There are other approaches of efficient CNN processing at the cost of accuracy degradation, such as [8, 55, 27, 60] among many others. While all of them are very progressive and interesting research topics that propose different paradigms of efficient CNN processing, it is not easy to integrate them into the conventional systems and CNN architectures. There is a gap between the data scientists who seek the ease of usage and high CNN performance, and the engineering field that seeks the maximum energy efficiency. Improving the energy consumption with the approximate multipliers can potentially bridge this gap as the circuits can be easily integrated into the hardware accelerators for CNN processing.

## 3.4 Accumulated Error in Convolution and Fully Connected Layers

While optimizing CNN inference through approximate multiplication was demonstrated in several previous studies, there was limited understanding of why it worked well for CNNs. The promising results led to the general observation that CNNs were resilient against small arithmetic errors, but none of them identified the complete reason behind that resilience. Specifically, it was unclear how the CNN layers preserved their functionalities when all their multiplications have a certain amount of error. The lack of understanding made it challenging to identify the suitable approximate multiplier for each network model, leading to expensive search-based methodologies in some studies [50].

The CNN convolution layers achieve abstract feature detection by performing convolution between their input channels and kernels. They produce feature maps where the locations of abstract features are represented by high output values relative to other locations. Because the features are represented with relative values as opposed to absolute values, it is much more important to minimize the variance of error between the convolution outputs than minimizing the absolute mean of errors when applying approximate multiplication to convolution. In other words, it is acceptable to have a certain amount of error in multiplications as long as the errors affected all outputs of convolution as equally as possible. The FC layers behave in the same way that most likely nodes have relatively higher output values compared to less likely nodes, and it is important to minimize the variance of error between the nodes. These observations are clearly demonstrated by the experiments in Section 3.6.3.

The experiments also observe that the variance of accumulated error in convolution is very small when the proposed approximate multipliers are applied. The observation is counter-intuitive, because the log multipliers have significantly large range of possible error. Mitchell Log Multiplier can produce up to -11.1% relative error, and Mitch-$w$ has a bigger range of error depending on the $w$ parameter. It is intuitive to expect that the different amount of errors in multiplication affect the convolution outputs differently and disturb the feature maps, but the experimental results do not reflect this intuition.

This section provides the analytical explanation that the convolution and FC layers are resilient against the errors in multiplication, because they consist of large number of multiplications and accumulations that converge the accumulated errors to a mean value. The variance of accumulated error is minimized and all outputs of the layers are equally affected because of this convergence, and maintaining the relative magnitudes between the outputs preserves the functionality of abstract feature detection.

Equation 3.1 shows the multi-channel convolution, where feature $s$ at $(i,j)$ is the accumulation of products between kernel $w$ and input $x$ across the kernel dimensions (m,n) and the input channels (l).

$$s_{i,j} = \sum_{l} \sum_{m} \sum_{n} w_{l,m,n} \cdot x_{l,i-m,j-n} \ . \tag{3.1}$$

An approximate multiplier produces different amount of errors in the products depending on the inputs. The mean error of the multiplier is measured by repeating many multiplications with random inputs and taking the mean of errors. The inputs from CNN inference are not strictly random, but they are numerous and practically unpredictable. Approximately modeling the inputs from CNNs as uniformly random, it is statistically acceptable to expect that the accumulated error in each convolution output converges closer to the multiplier mean error when many multiplications are performed and accumulated. This convergence reduces the variance of accumulated error between the outputs and the values tend to scale by the same amount, minimizing the effect of varying error on feature detection. Figure 3.6

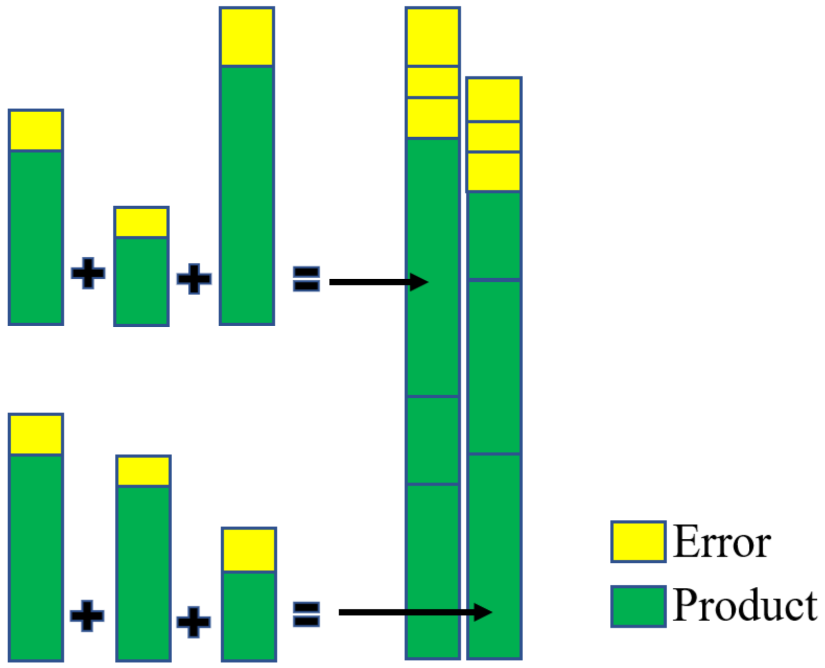


Figure 3.6: The accumulation of many products with varying amount of error converges the combined errors to a mean value.

shows the abstraction of this mechanism, and the examples on LeNet can be seen in Section 3.6.3. Equation 3.2 describes the feature $s'_{i,j}$ when multiplications are associated with the mean error of $e$.

$$s'_{i,j} = \sum_{l} \sum_{m} \sum_{n} w_{l,m,n} \cdot x_{l,i-m,j-n} \cdot (1+e) \ , \tag{3.2}$$

$$s'_{i,j} = (1+e) \cdot s_{i,j} \ . \tag{3.3}$$

Therefore, the features are simply scaled by the mean error of the approximate multiplication when a large number of products are accumulated.

It should be noted that the observations hold only for the approximate multiplications with the symmetric behavior between positive and negative results. All approximate multipliers studied in this dissertation satisfy this condition, because all of them handle signs separately from magnitudes.

The usual number of accumulations in convolution is not large enough to completely nullify the variance of accumulated error, but it is enough to sustain the functionality of abstract feature detection that needs to be robust against small variations. The CNNs typically start with smaller number of convolution channels to obtain general features, and the widths increase in the deeper layers where features become more specific. Approximate multiplication on such CNNs exhibits the desired trend of having smaller effects in the wide and deep layers as required. The larger variance of accumulated error in the shallow layers is tolerable because the feature detection needs to account for the small variations in the input images. In fact, some previous works such as [27] had claimed that earlier layers can be approximated more in neural networks.

This hypothesis implies the importance of exact additions, because it will have a more substantial impact on the variance of accumulated error. This agrees with the work by [13], where approximating the additions had a larger impact on the CNN accuracies. As multipliers in fixed-point arithmetic are much more expensive than adders, approximating only the multipliers gains the most benefit with minimal degradation in CNN inferences.

The approximate multiplication also benefits from the fact that the convolution outputs receive inputs from the same set of input channels. For each convolution output, there are two types of accumulations. One type occurs within each input channel across the kernel dimensions, while the other occurs across the input channels to produce the final output. The intra-channel accumulation combines the products from the same input channel and kernel, and therefore each channel has a specific range of values within which features are located. The inter-channel accumulation may have more varying ranges of products, because each input channel has its own kernel and input values. Different input ranges may trigger different error characteristics on the approximate multiplier, but every convolution output accumulates from all input channels so that it does not affect the variance of accumulated error between the outputs.

These observations are best understood and verified by comparing against depthwise separable convolution. Depthwise separable convolution consists in a depthwise convolution followed by a pointwise convolution [7]. Depthwise convolution is a special case of grouped convolution that eliminates the accumulation across input channels, and the reduced number of accumulations leads to an increase in the variance of accumulated error in the outputs. Figure 3.7 shows the comparison of the accumulation pattern between conventional convolution and depthwise convolution. Also, each output channel receives inputs from only one input channel, and the difference of error between output channels is subject to another approximate multiplication and variance of error before the channels are accumulated in the following pointwise convolution. More accurate approximate multipliers are required for CNNs that use depthwise separable convolution, because errors from approximate multiplication do not converge well.

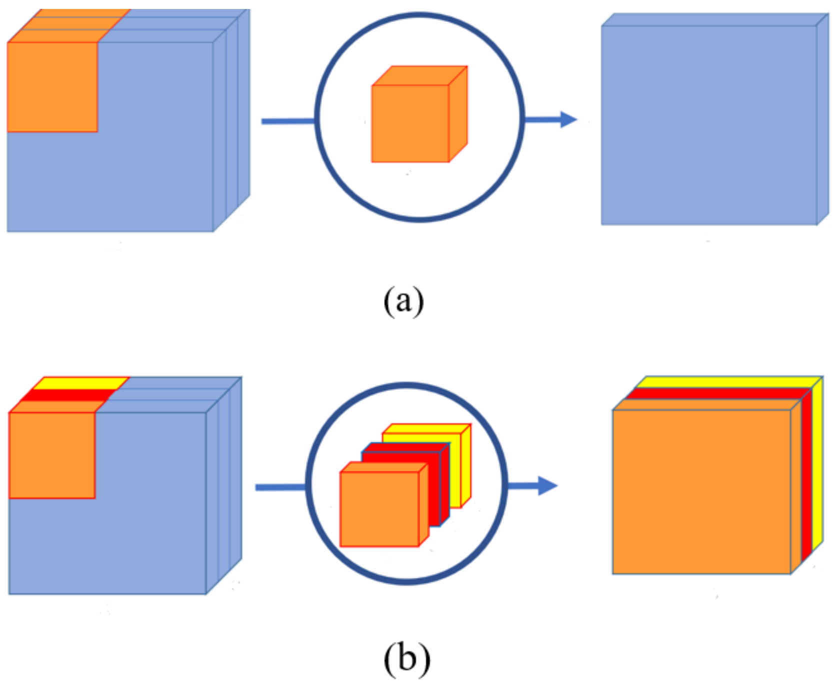


Figure 3.7: (a) Conventional Convolution (b) Depthwise Convolution

Another technique that reduces the number of accumulations is the 1x1 convolution, but it is found to be compatible with approximate multipliers. The 1x1 convolution does not have any intra-channel accumulation but accumulates the products across input channels. Because deep CNNs require large widths appropriate for their deep structures, inputs to 1x1 convolutions usually consist of many input channels, and therefore provide enough accumulations for the error convergence. Each output of 1x1 convolution also receives inputs from all input channels, which provides more consistent accumulation of error between the outputs.

The FC layers are also resilient against the effects of approximate multiplication, as the same factors help converge errors in the outputs. There is usually a large number of accumulations per each output, and all of the outputs share the same set of inputs. Thus, the FC layers show minimal differences in CNN accuracies when their multiplications are approximated.

The discoveries in this section are not limited to the error of approximate multiplication but apply to all sources of arithmetic error. They also provide deeper understanding of why bfloat16 [66] has been widely successful at accelerating CNNs despite its reduced precision. By truncating the less significant fractional bits, converting a FP32 value to bfloat16 generates the small negative error from 0% to -0.78% relative to the original FP32 value. The factors discussed in Section 3.4 also minimize the negative effects of this varying error, and they explain why using the full FP32 accumulator after bfloat16 multiplication produces the best results [19], in agreement with the observation that the accumulations need to be exact. The accumulation of mean error discussed in Section 3.5 should also be present, but the mean error of bfloat16 is too small to cause any problems for the studied CNNs.

The successful application of bfloat16 to CNNs has been explained by the high-level interpretation that the small amount of error helps the regularization of a CNN model. The interpretation is still valid and also applies to approximate multiplication, and the findings of this section provide deeper understanding with the arithmetic explanation. They also explain why the bfloat16 format has slightly degraded performances with the networks that use grouped convolution, as presented in Section 3.7.2.

## 3.5 Effects of Batch Normalization on Very Deep Neural Networks

The approximate log multiplication generates negative error in the results, meaning that the product has less magnitude compared to the exact multiplication [47]. It is evident from Equation 3.3 that the features have less magnitudes with the log multiplication, and the experiments in Section 3.6 show that this becomes a problem for deeper layers. Its negative effect on the network performance is observable in AlexNet with only 8 layers of convolution and FC, and the mean error accumulation would be problematic for much deeper networks. The approximate multiplication repeatedly reduces the magnitudes of the features, and the deeper layers receive input distributions that are difficult to distinguish. On the other hand, if an approximate multiplier has a positively biased mean error, it is possible to amplify the values beyond the range set by quantization, resulting in the arithmetic overflow. These negative effects are under the best-case scenario of ReLU activation, and the other types may suffer additional errors in activations.

Batch normalization [22], the popular technique used in most deep CNNs, can alleviate this problem and help approximate multiplication go deeper into the networks. A critical function of batch normalization is to redistribute the output feature maps to have more consistent input distributions for deeper layers. While the training process necessitates this function, the inferences on the resulting models still need to go through the normalization with the stored global parameters of expected distributions. These global parameters can be appropriately adjusted to account for the changes in the distributions due to approximate multiplication, and this can prevent the accumulation of mean error across the layers.

These global parameters are a source of error for approximate multiplication without proper adjustments, because the distribution of convolution outputs change as the result of approximate multiplication. Equations 3.6 and 3.9 show the mean ($\mu'$) and variance ($(\sigma')^2$) of the convolution output distribution, when the features $s'_{i,j}$ have the mean error $e$ from Equation 3.3.

$$\mu' = 1/m \sum_{i,j} s'_{i,j} \ , \tag{3.4}$$

$$\mu' = 1/m \sum_{i,j} (1+e) \cdot s_{i,j} \ , \tag{3.5}$$

$$\mu' = (1+e)\mu \ . \tag{3.6}$$

$$(\sigma')^2 = 1/m \sum_{i,j} (s'_{i,j} - u')^2 \ , \tag{3.7}$$

$$(\sigma')^2 = 1/m \sum_{i,j} (1+e)^2 (s_{i,j} - u)^2 \ , \tag{3.8}$$

$$(\sigma')^2 = (1+e)^2 \cdot \sigma^2 \ . \tag{3.9}$$

Therefore, the stored mean values for batch normalization are scaled by $(1+e)$, while the variance values are scaled by $(1+e)^2$. With the adjusted parameters, the batch normalization layers correctly normalize the convolution outputs and scale them back to the desired distributions. Failing to adjust these parameters results in incorrect redistribution of feature maps and worse CNN accuracies. The proposal only requires the scaling of the stored parameters and significantly improves the performance of approximate multipliers on deep neural networks. It does not introduce any new operations and does not prevent the ability of batch normalization to fold into neighboring layers.

Designing an approximate multiplier with an unbiased mean error near zero is another effective solution, and the unbiased Mitch-$w$ design is discussed in Section 2.2.5. However, the unbiased designs usually have a small amount of mean error because it is difficult to create a perfectly unbiased design, and the problem is only deferred to deeper networks.

Also, depending on the approximation method, it may take additional hardware resources to make a design unbiased as is the case with Mitch-$w$. The networks that do not use batch normalization have no choice but to use the unbiased multipliers, but otherwise the proposed adjustment is simpler, less costly, and more flexible to accommodate different approximation methods with biased mean errors.

## 3.6 Experiments with Simple Networks

In this section, the proposed multipliers in Section 2.1 and 2.2 are applied to the inferences of small CNN models. Various experiments show that the proposed log multipliers cause very little degradation in CNN inference accuracies and they also support the design decisions such as operand truncation and C1 sign handling. The observations about CNN inferences made in this section provide the basis for investigating the effects of approximate multiplication on very deep CNN models in Section 3.7.

### 3.6.1 Setup

We used the Caffe framework to evaluate the viability of our approximate multipliers on CNNs. Caffe is a well-known deep learning framework that provides the tools to train and test CNN models for the visual recognition applications [23]. We implemented various multiplier designs in C++ using a fixed-point class library, and replaced the matrix multiplication of Caffe with our own code that calls the implemented procedures. All CNN experiments in this section, except those in Section 3.6.2, were performed with 16 integer bits and 16 fractional bits, as they were sufficient for all three network instances we used. Performing careful quantization to each network model may achieve lower number of bits, but the generous quantization allows the study of approximate multiplication in isolation while also providing a more general solution to the hardware acceleration of CNNs. CNN computations involve signed numbers and the C2 conversion was used before and after the approximate multipliers to correctly handle them. The results with the C1 conversion are analyzed in Section 3.6.6.

We used three different datasets to evaluate our designs: MNIST, CIFAR-10, and ImageNet ILSVRC2012 validation data. MNIST is the handwritten digit recognition dataset, and it is composed of 60,000 training examples and 10,000 test examples of handwritten digits. LeNet was the milestone CNN that was very successful with the classification of MNIST database [38]. We used the modified version of LeNet provided in the Caffe framework that replaced the sigmoid activations with Rectified Linear Units (ReLU). CIFAR-10 [32] is the dataset for object recognition collected by the creators of AlexNet, another milestone CNN [33]. It involves recognizing the 10 objects such as airplane and automobile from 32x32 color images, and is made of 50,000 training images and 10,000 test images. The network we used for CIFAR-10 is Alex Krizhevsky's cuda-convnet [32], which was also provided with Caffe. Lastly, the AlexNet network [33] on ILSVRC2012 dataset [12] was the milestone achievement that brought the explosion of interest surrounding CNNs and image recognition applications. From ILSVRC2012, the validation set of 50,000 images is used for CNN inferences. TABLE 3.1 describes the CNNs used to perform inferences for each dataset.

The approximate multipliers were not suitable for the training of the networks, because the amount of error was too large for the gradient descent of backpropagation to converge. For the MNIST and CIFAR-10 datasets, we trained the networks first using floating-point arithmetic and applied the multiplier models to the inference stage. For these datasets, the Top-1 CNN inference accuracy has been measured across all 10,000 test images. For AlexNet, we simply used the pre-trained model that was available in Caffe without further training. Performing experiments on all 50,000 images of ILSVRC2012 validation set proved to be very time-consuming, because the C++ models of the approximate multipliers were much slower to simulate than the conventional multiplication performed in hardware. Therefore,

Table 3.1: The network descriptions of target CNNs

| **Dataset** | **CNN** | **Layers** |
|---|---|---|
| MNIST | LeNet | Conv[5x5] → Pooling[2x2, stride 2] → Conv→ Pooling→ Fully Connected (FC) [500 output] → ReLU → FC[10 output] |
| CIFAR-10 | Cuda-convnet | Conv[5x5] → Pooling[3x3, stride 2] → ReLU → LRN[3x3] → Conv → ReLU → Pooling → LRN → Conv → ReLU → Pooling → FC[10 output] |
| ImageNet | AlexNet | Conv[11x11] → ReLU → LRN → Pooling[3x3] → Conv[5x5] → ReLU → LRN → Pooling [3x3]→ Conv[3x3] → ReLU → Conv[3x3] → ReLU → Conv[3x3] → ReLU → Pooling[3x3] → FC [4096 output] → ReLU → FC [4096 output] → ReLU → FC[1000 output] |

we primarily used the first 5,000 images from the shuffled set to perform various experiments, and then used the entire set of 50,000 images in Section 3.6.8 to further prove our designs.

The performance of CNNs for visual object classification is typically measured by the Top-1 and Top-5 accuracies. Given an image, a CNN produces the inference scores of all object classes, and the Top-1 accuracy measures the rate of the object with highest score matching the correct answer. For the Top-5 accuracy, the correct answer is searched among the top five scores instead of just one.

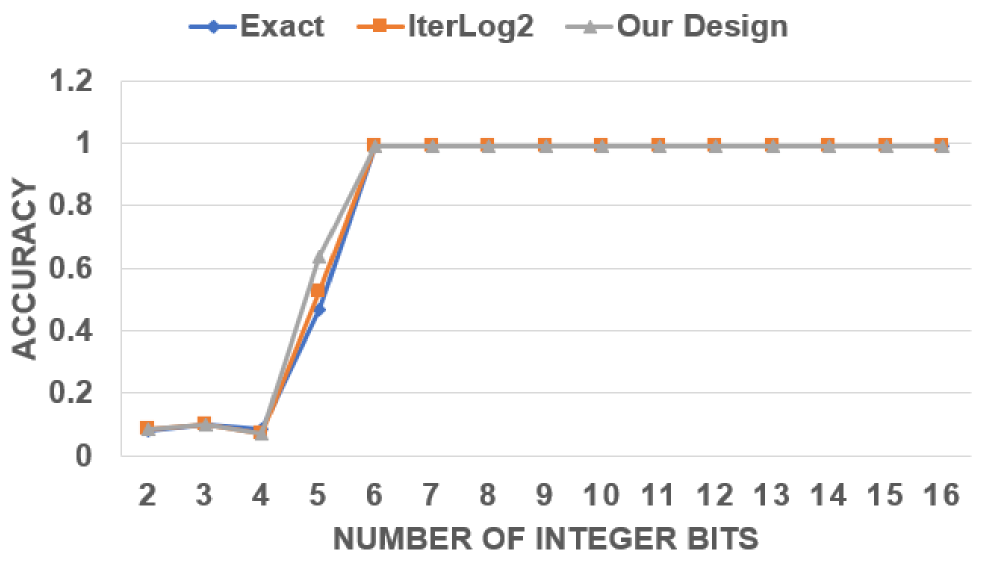
### 3.6.2 Network Dependency of Quantization

Because the multipliers perform fixed-point arithmetic, the number of bits used is important to their precision and resource usage. Hence, the numbers of integer and fractional fixed-point bits were varied to evaluate the CNNs at different numerical range and precision. It demonstrates the effects of quantization on CNN accuracies, and supports the conclusion that the number of bits required depends on each network model.

We performed our experiments on the MNIST and CIFAR-10 datasets. The Mitchell Log Multiplier was used for this evaluation as well as the exact fixed-point multiplier and the

2-stage iterative log multiplier [4]. The Top-1 CNN accuracy for each multiplier has been measured across all test images of the datasets.

Figure 3.8 and 3.9 show the comparisons of the Top-1 classification accuracies among the multipliers, with varying number of integer and fractional fixed-point bits. Figure 3.8 and 3.9 show that all the multipliers caused sharp drops in the network performance when insufficient number of bits were provided for either integer or fractional parts. We found that having insufficient number of integer bits saturated too many values and produced incorrect results, while having insufficient number of fractional bits caused the loss of precision and differentiability of the values. It was also noticed that the necessary number of bits was different for each network; our design applied on MNIST LeNet required 6 integer bits and 8 fractional bits to reach the full accuracy, while it required about 10 integer and 10 fractional bits on CIFAR-10 cuda-convnet. This supports the conclusion that different network models, datasets, quantization techniques and training methods all affect the numerical values in the network, and the number of bits must adjust to the range and precision required to represent the distribution of the values. This conclusion agrees with the findings of other research such as [25, 37, 54, 45, 5], where different numbers of bits were required for the different neural networks. Judd et al. [25] and Lai et al. [37] in particular clearly proved the network dependency and showed the trend where the most complex networks required more bits.



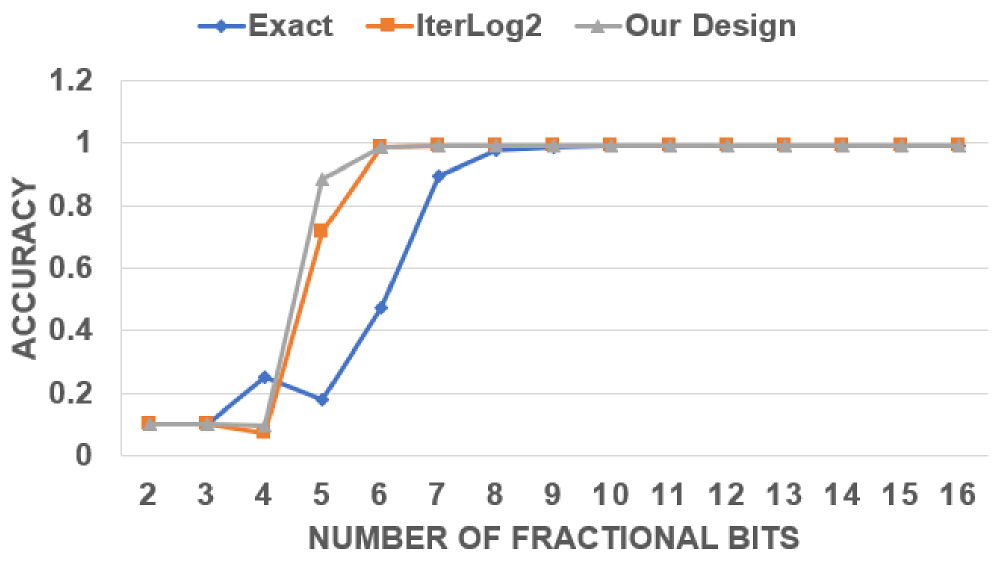
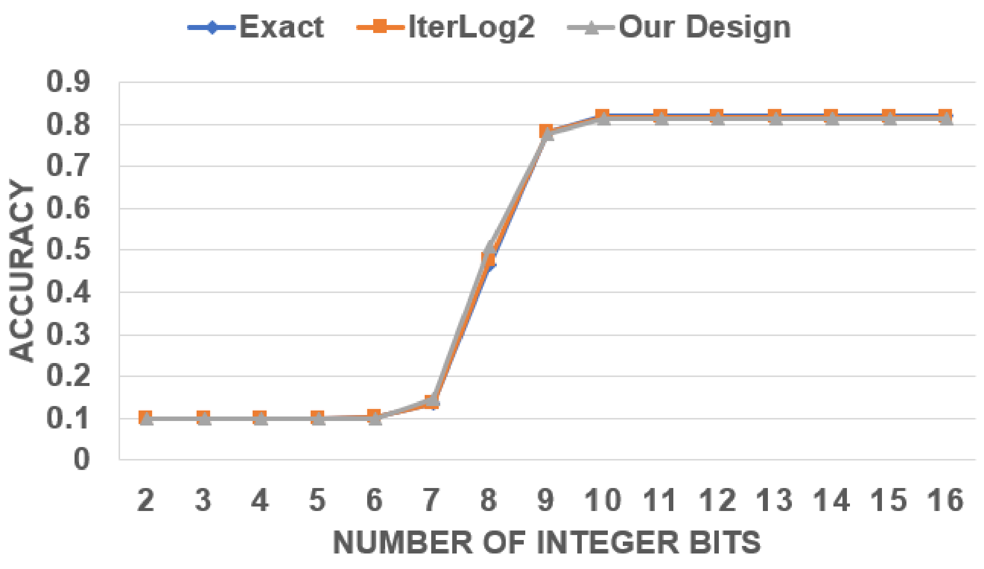


Figure 3.8: Comparison of the Top-1 classification accuracies on MNIST between the multipliers, with varying number of integer and fractional fixed-point bits.



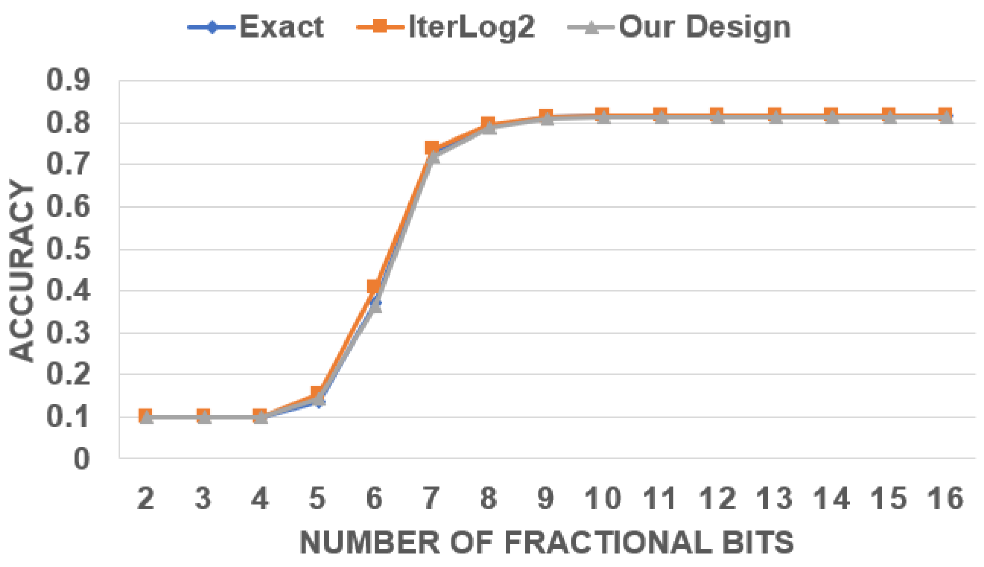


Figure 3.9: Comparison of the Top-1 classification accuracies on CIFAR-10 between the multipliers, with varying number of integer and fractional fixed-point bits.

Table 3.2: Top-1 accuracies for LeNet and cuda-convnet

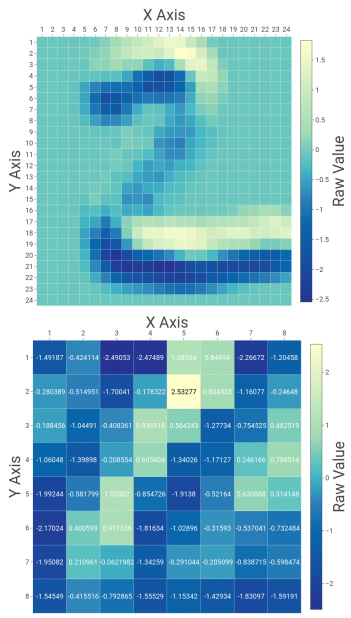
| **Network** | **Float** | **Fixed** | **Mitchell** | **Mitch-$w$** |
|---|---|---|---|---|
| LeNet | 99.0% | 99.0% | 99.0% | 99.0% |
| cuda-convnet | 81.4% | 81.9% | 81.4% | 81.3% |

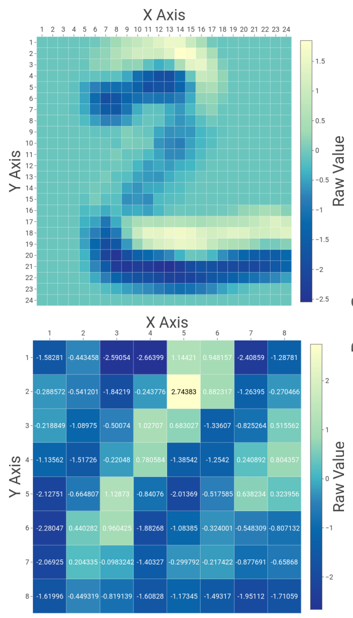
There are other research projects that reported less number of bits compared to our results. The research in [25] and [54] applied dynamic quantization to each network layer instead of using a uniform quantization to reduce the number of bits required, but it would require the cost of additional circuitries and design efforts. SqueezeNet [20] and Google Tensor Processor Unit [24] used only 8 bits and significantly reduced the hardware resource, but at the cost of small performance degradation which may not be acceptable in some applications. They have their advantages as well as disadvantages, and are not counterexamples to our experiments. After all, it is not our aim to minimize the number of bits required through various techniques. Our primary aim is to show that our logarithmic multiplier can produce substantial power reduction for the CNNs without performance degradation. Our low-power design can produce significant power savings at as little as 8 bits, and the savings will be larger when more bits are required for more complex networks.

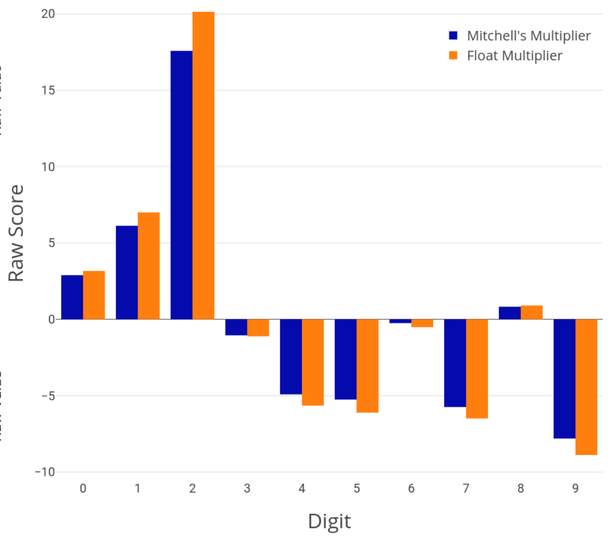
### 3.6.3 Impact on CNN Performance

Table 3.2 shows the Top-1 accuracies on MNIST and CIFAR-10 with different multipliers. Mitchell refers to the design in Section 2.1, and Mitch-$w$ is the design with $w$=6. As can be observed, our low-energy design does not cause significant performance degradation despite the error from approximation.

Figure 3.10 shows the sample convolution layer outputs and the final raw scores from one of the MNIST inferences. The Mitchell multiplier produces the convolution outputs that are very close to the floating-point multiplier in terms of the relative magnitude. Mitchell's Algorithm always produces negative errors, meaning that the magnitude of the values are always reduced, but it is applied to all values so that the convolution can still locate the abstract features in the images.


(a) Convolution by Log Mult.


(b) Convolution by Float Mult.



(c) The final scores

Figure 3.10: The convolution outputs and the final raw scores of a sample inference from LeNet

The final layer output shows the effect of approximate multiplication more clearly, as the scores are reduced in magnitude. The reductions are more than the -4% mean error of the Mitchell multiplier and indicate that the errors have been accumulated across the layers. Despite the difference in the absolute magnitude of results, the log multiplier still correctly recognizes "2", because all the outputs are reduced at the same time so that the order between them is preserved. The effect of the log multiplier is as if the image was reproduced with smaller pixel values; lighter image but still clearly recognizable. In fact, CNNs are designed to have the tolerance for such small differences in the input images to successfully classify as much images as possible. The computational error did not affect the performances of the CNNs, because they are measured by the correctness of the discrete outputs. The

Table 3.3: Top-1 and Top-5 accuracies for AlexNet

| | **Float** | **Fixed** | **Mitch-$w$6** | **Mitch-$w$8** |
|---|---|---|---|---|
| **Top-1** | 58.3% | 58.3% | 58.0% | 58.0% |
| **Top-5** | 80.2% | 80.2% | 80.0% | 80.1% |

absolute score computed for each option is not as important as the relative order of the scores. This property of the discrete classification makes CNNs resilient against approximations and reduced precisions of computation.

The exact fixed-point multiplication showed better performance in CIFAR-10 than the reference floating-point as shown in Table 3.2. Even though the Mitchell multiplier shows exactly the same accuracies as the reference in MNIST and CIFAR-10, a closer examination revealed that they did not produce the same predictions for all test images. The floating-point multiplier produced better predictions for some images while the Mitchell multiplier did better for others. This happens because inexact computations can result in correct predictions when exact computations would have resulted in the incorrect ones.

Table 3.3 shows the Top-1 and Top-5 accuracies for AlexNet on ImageNet dataset. Figure 3.11 shows the top 10 scores of a sample inference sorted by the reference floating-point score. The results from AlexNet demonstrate the same concepts presented in MNIST. The negative errors of approximate designs decrease magnitudes of the final scores, but all the output options are affected so that the relative order between them are mostly preserved.

The ImageNet [12] is more challenging than the MNIST for the approximate multiplication primarily because there are many more possible outputs and their raw scores are much closer together. The variations in error accumulation can change the order of the output more easily. We can also see in Figure 3.11 that the amount of error accumulation shown by the value reduction is higher than the MNIST example, because AlexNet has more layers and computations compared to LeNet.

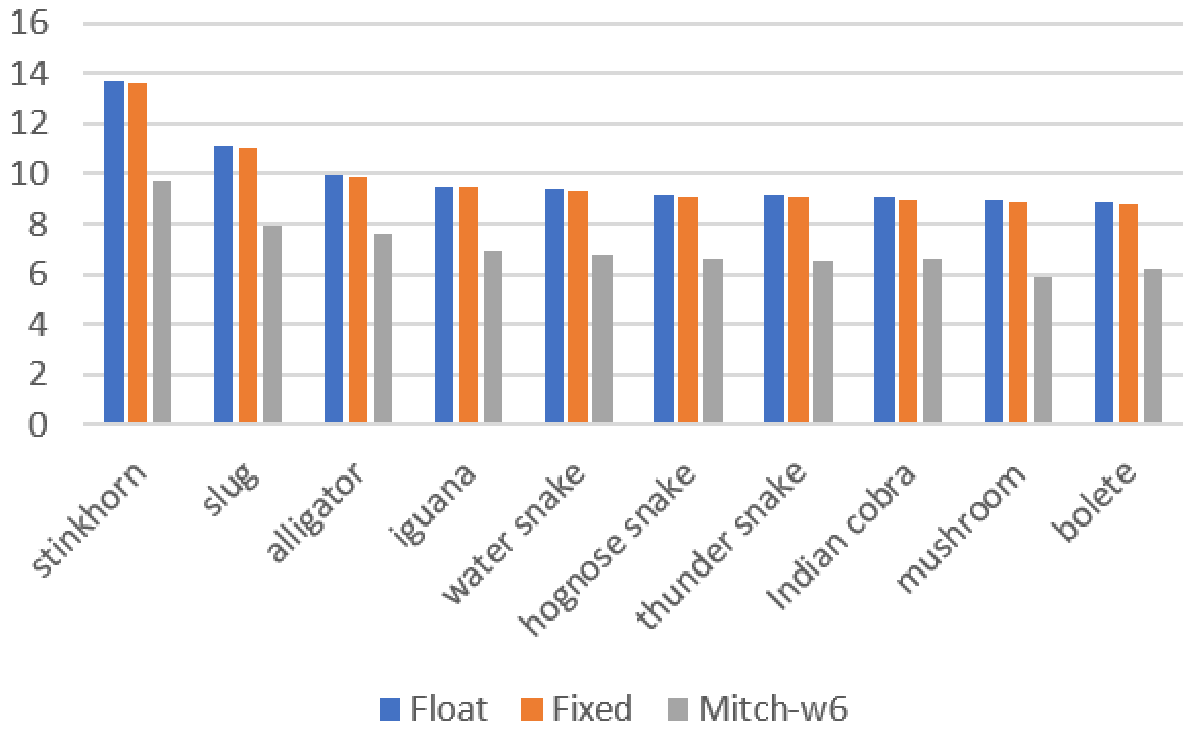


Figure 3.11: Top 10 scores of a sample AlexNet inference, sorted by the highest floating-point raw scores

Yet, despite the difficulties the proposed designs perform reasonably well even for the ImageNet. The relative order in the top 10 are slightly modified between the floating-point and Mitch-$w$ in Figure 3.11, but the example still demonstrates the same top 5 scores, with the same emphasis on the first score. Also, it should be noted that there are images that the floating-point incorrectly classifies while the Mitch-$w$ does correctly, because the inexact computations can result in correct predictions. Overall, the proposed Mitch-$w$8 design produces the Top-1 accuracy that is only 0.3% less than the floating-point multiplier, and the Top-5 accuracy is only 0.1% less.

From the presented examples, we can deduce the cause of incorrect inferences and the desirable properties of the approximate multiplication in CNNs. Reducing the mean error of the multiplier reduces the error accumulation going across the layers, but it is not the dominant factor because the same mean error accumulation is applied to all scores. On the other hand, the dispersion of the error is much more important, because an incorrect prediction occurs when the computational errors affect two scores differently, thereby reversing their relative order. Fortunately, Section 3.4 shows that this effect is minimized for the conventional convolution and FC layers because of the large amount of product accumulations.

The exact distribution of error that produces the best results for CNNs is hard to define or quantify, because it depends on each dataset and network instance. Depending on the values present in CNNs, there are error patterns that perform particularly well for a given instance. However, the range of error given by the difference between maximum and minimum possible values (or PWCE - NWCE) is proposed as a useful guideline when evaluating the approximate multipliers for CNNs. CNNs suffer performance degradation when the top choices are replaced by others due to different error accumulations, and the range of error represents the possible extent of this difference. The WCE alone is incomplete because it hides the existence of either PWCE or NWCE and underestimates the variation of error. The standard deviation of error is not as representative as the range of error, because it is the

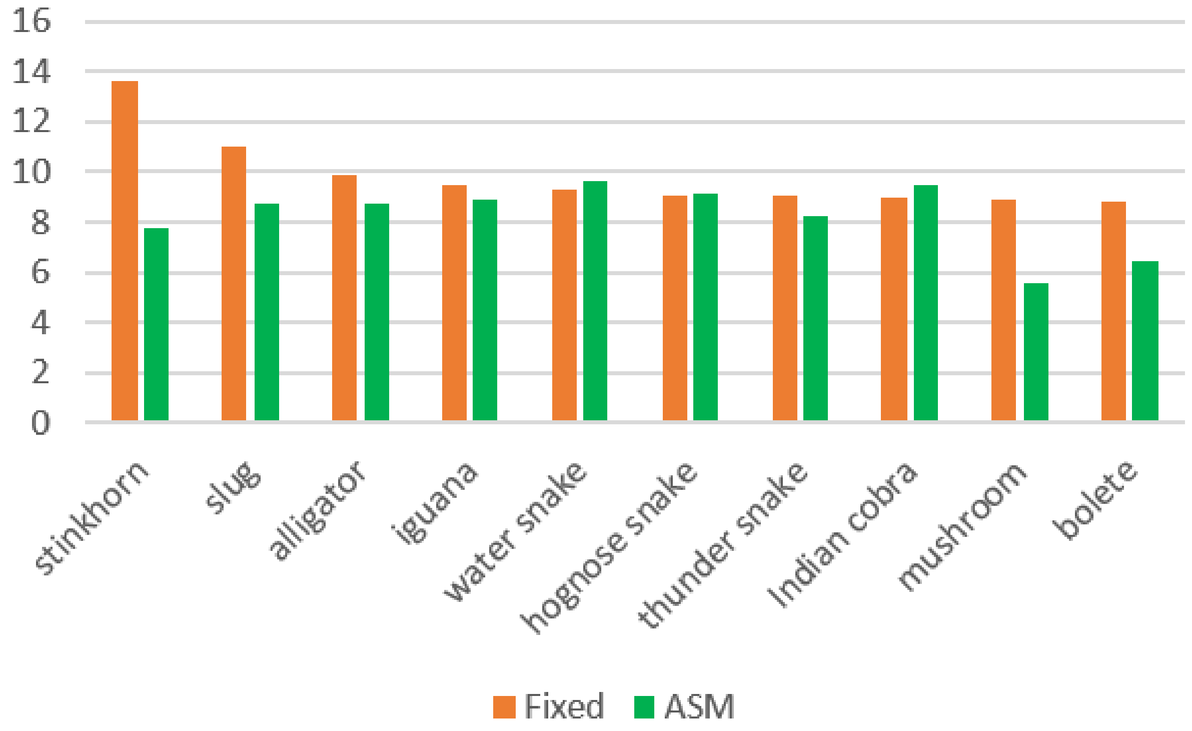


Figure 3.12: The final raw scores of the sample inference from AlexNet with the ASM multiplier

small number of large worst errors that have the significant impact on the final outputs [26].

Figure 3.12 shows the final scores of the same inference with the 1-Alphabet ASM [58]. The ASM methodology has a mean error of 1.5%, PWCE of 33% and NWCE of -47%, and authors reported performance degradation when the multiplier was applied to more complex datasets. In Figure 3.12, some of the final scores are increased in magnitude while the others are significantly decreased, as the effect of having both high positive and negative errors. The amount of error variation is so large that the order of the scores is heavily disrupted to produce incorrect inferences. The ASM performed poorly in AlexNet and we present the accuracies in Section 3.6.7 where we make the comprehensive comparisons between different approximate multipliers.

Table 3.4: Top-1 and Top-5 accuracies of AlexNet with the Mitch-$w$ design with varying $w$

| | **Top-1** | **Top-5** |
|---|---|---|
| **Mitch-$w$3** | 54.5% | 78.0% |
| **Mitch-$w$4** | 57.0% | 79.8% |
| **Mitch-$w$5** | 57.7% | 79.8% |
| **Mitch-$w$6** | 58.0% | 80.0% |
| **Mitch-$w$7** | 57.9% | 80.1% |
| **Mitch-$w$8** | 58.0% | 80.1% |

### 3.6.4 Evaluation of the Truncation Parameter

To evaluate the effect of $w$ values on CNN performance, we measured the Top-1 and Top-5 accuracies in AlexNet for various $w$ values. Table 3.4 shows the Top-1 and Top-5 accuracies for increasing $w$. The performance improves as expected when the multiplier accuracy is improved with higher $w$. The performance improvement after $w$=6 shows diminishing return so that the $w$ truncation of our designs is justified. Proper choice of $w$ depends on the application, dataset and network instance, and our parameterized designs can accommodate the requirements of different applications.

### 3.6.5 Evaluation of the Unbiased Designs

The experiments were performed with the ImageNet dataset to test the unbiased designs presented in Section 2.2.5. Table 3.5 shows the comparisons of the Top-5 accuracies for different $w$. The performances were comparable, though the unbiased designs performed slightly better than the original biased designs. Figure 3.13 shows the final outputs of the sample inference, also sorted from the left by the floating-point reference values. The mean error of the unbiased designs is a small positive value and the accumulated effect is evident in

Table 3.5: Effect of unbiased error on Top-5 accuracy for AlexNet

| | **Original** | **Unbiased** |
|---|---|---|
| **Mitch-$w$5** | 79.8% | 80.1% |
| **Mitch-$w$6** | 80.0% | 80.3% |
| **Mitch-$w$7** | 80.1% | 80.2% |
| **Mitch-$w$8** | 80.1% | 80.2% |

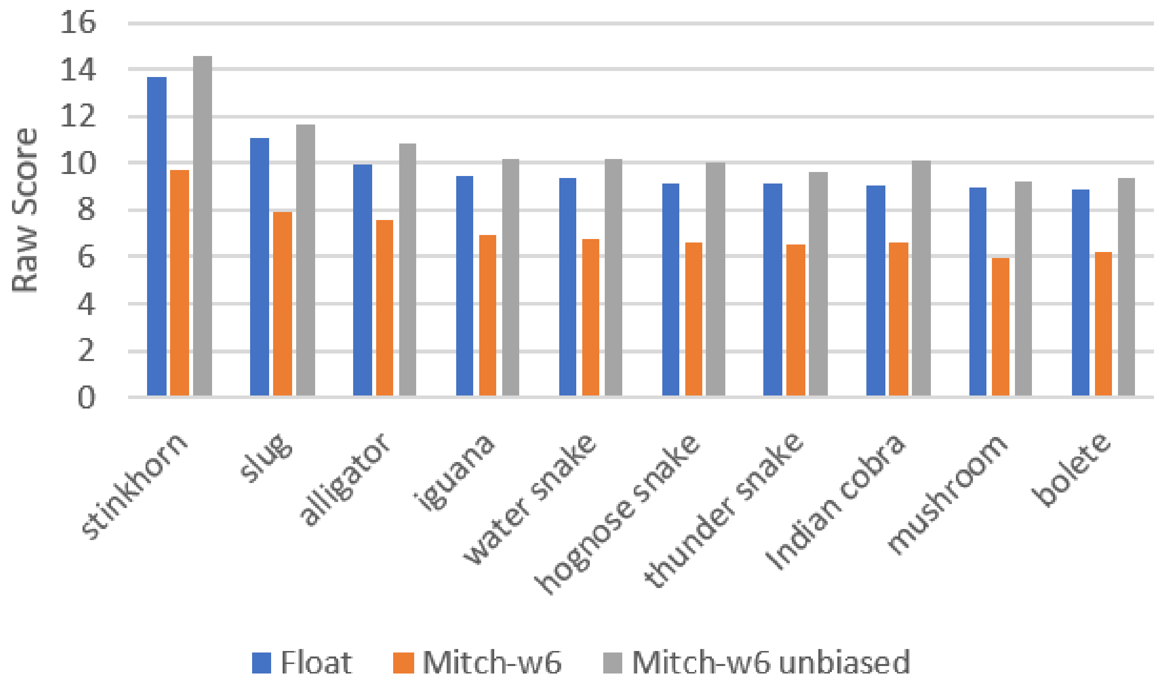


Figure 3.13: The final scores of the sample inference from AlexNet with the unbiased Mitch-$w$

Table 3.6: Effect of the C1 approximation on Top-5 accuracy for AlexNet

| | **2's comp** | **1's comp** |
|---|---|---|
| **Mitch-$w$5** | 80.1% | 80.0% |
| **Mitch-$w$6** | 80.3% | 80.2% |
| **Mitch-$w$7** | 80.2% | 80.2% |
| **Mitch-$w$8** | 80.2% | 80.2% |

the final scores. We can see the same error accumulation across the layers that is applied to all scores, and the order of Top-5 is preserved while the order of top 10 shows small variations.

The CNN accuracy improves with the unbiased designs because the small mean error prevents the compression of numerical range observed with the negative mean error. There are as many as 1000 object classes in ImageNet, and CNNs produce the final scores that are much closer together than for the simpler datasets. When the magnitudes of scores are reduced because of negative mean error, absolute differences between the scores become even smaller, and relative order between the outputs becomes more susceptible to changes. In summary, unbiasing the approximate multiplier negates the small adverse effect of negative mean errors on CNNs.

### 3.6.6 Evaluation of the C1 Sign Handling

The effect of the C1 sign handling approximation in the CNNs is discussed in this section. Table 3.6 shows the comparisons of the Top-5 accuracies using the C2 and the C1 sign conversions, for different values of $w$. We can see that both approaches produce comparable performances in AlexNet, with small drops for lower values of $w$. Nevertheless, employing the C1 transform reduces the energy, area, and critical path, as shown in Table 2.6.

Table 3.7: The average differences in the outputs of each layer for AlexNet, between C2 and C1 sign handling

| | **Conv1** | **Conv2** | **Conv3** | **Conv4** | **Conv5** | **FC1** | **FC2** | **FC3** |
|---|---|---|---|---|---|---|---|---|
| **Avg. Diff(%)** | 8.7 | 7.2 | 9.5 | 12.3 | 6.7 | 15.5 | 15.7 | 15.5 |
| **Num. Accum.** | 363 | 1200 | 2304 | 1728 | 1728 | 10816 | 4096 | 4096 |

As discussed in Section 2.2.4, the proposed designs with C1 produce very high relative errors with the small negative operands, because they are offset by the absolute value of one. The high relative error from the small operands would be unacceptable, if the application involved a chain of multiplications that makes the relative error important. However, the convolution and the fully connected layers of CNNs perform MAC operations. Each multiplication is followed by the accumulation, and the absolute error of the multiplication is accumulated instead of the relative error. This makes the corner cases of small operands have less impact on the error accumulation in the CNNs, because the results and the errors from small operands tend to be small in absolute magnitude as mentioned in Section 2.2.4.

To further support the claim, the difference in each output value from the convolution and fully connected layers is measured between the designs using the C2 and C1 conversions. The average differences for AlexNet layers at $w$=6 are displayed in Table 3.7, along with the number of accumulations for each output. Each layer has a different range of values, so the differences are taken as relative values to display the pattern more clearly. To prevent any possible confusion, this is not the same as the relative error of the multiplier, because it is normalizing the outputs of the MAC operations.

Table 3.7 clearly shows that the fully connected layers have higher differences than the convolutional layers. There is a sharp increase from the last convolutional layer to the first fully connected layer. One factor of the behavior is that the later layers take the outputs from the previous layers as inputs and therefore have more accumulated differences. The

Table 3.8: The average differences in the outputs of the layers for the CNNs, between C2 and C1 sign handling

| | **MNIST** | **CIFAR** | **ImageNet** |
|---|---|---|---|
| **Conv** | 0.6% | 1.6% | 8.6% |
| **FC** | 5.7% | 6.9% | 15.6% |

dominant factor, however, is the number of accumulations for each output. Each convolution output has a smaller number of accumulations that is the convolution kernel size multiplied by the number of input channels, and the fully connected layers have larger numbers as they receive all outputs from the previous layer as the inputs. The same pattern is also observed in the other CNNs, and the results are displayed in Table 3.8. Tables 3.7 and 3.8 not only demonstrate the process of absolute error accumulation described in the claim, but they also show that the impact of C1 is much less than what the high relative error of the corner cases suggest. They support that the high error from the C1 conversion with the small operands do not have as much impact in the CNN values.

These tables show the differences in the outputs compared to C2, but the differences do not automatically mean the degradation in CNN performance. Table 3.6 already demonstrated a comparable performance. Figure 3.14 shows the comparison of the top 10 scores from the sample inference, between the two sign handling techniques. We can see that all scores are affected at the same time so that the impact on the relative order is small. In conclusion, taking C1 for the proposed signed multiplier is a viable and cost-effective technique for the CNN computations.

### 3.6.7 Comparison against Other Approximate Multipliers on CNNs

In this section, the comparisons are made against the other approximate multipliers. We specifically compared against the 2-stage iterative log multiplier and the 1-Alphabet ASM

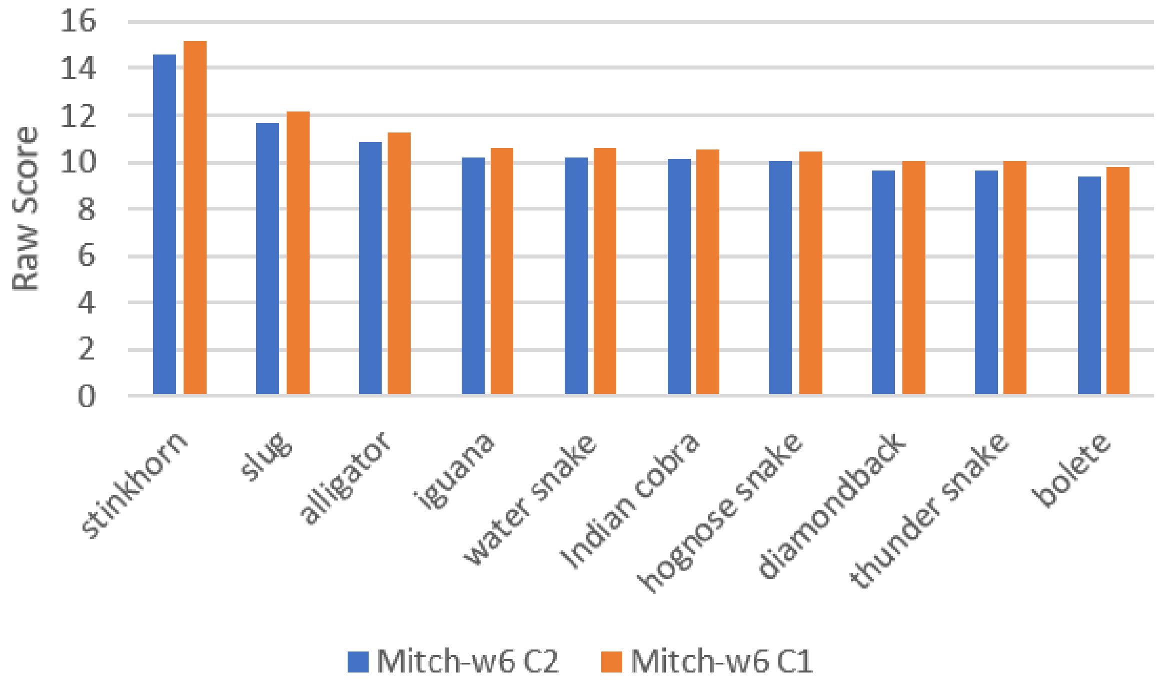


Figure 3.14: The final scores for the C2 and C1 sign handling from the sample inference of AlexNet

because they had been applied to neural networks previously, and we made the comparison to DRUM because the approximate multiplier showed very good accuracies and cost effectiveness.

Some modifications are made to the designs for the comparisons. The 2-stage iterative log multiplier presented in [45] was a pipelined design and lacked the zero detection unit so that they reported some CNN performance degradations. To compare the energy consumptions of the designs, we removed the pipeline registers and added the zero detection unit to make a fair comparison in our framework. We had observed that the addition of the zero detection unit improved the CNN performances significantly.

The 1-Alphabet ASM design presented in [58] only had 8 and 12-bit versions, but we extended the techniques to 16 and 32-bits for the comparison. Their methodology involved the retraining of the CNNs to round the unsupported values to the nearest values supported by the ASM. The retraining of the CNNs is unexplored in our work, and we instead added the rounding

Table 3.9: Comparison of the area and energy against the other approximate multipliers. Mitch-$w$ multipliers are unbiased and consider the C1 transform for negative numbers.

| | N=16 | | | | | N=32 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Fixed-point** | **Mitch-$w8$** | **2-Stage Iter. Log.** | **1-Alphabet ASM** | **DRUM6** | **Fixed-point** | **Mitch-$w8$** | **2-Stage Iter. Log.** | **1-Alphabet ASM** | **DRUM6** |
| **Crit. Path** ($ns$) | 2.23 | 1.90 | 3.88 | 2.64 | 2.64 | 3.78 | 2.50 | 4.00 | 4.00 | 3.96 |
| **Area** ($um^2$) | 2032 | 1135 | 3335 | 1543 | 1375 | 8627 | 2092 | 11218 | 7642 | 2917 |
| **Tot. Power** ($mW$) | 1.24 | 0.61 | 1.79 | 1.04 | 0.88 | 6.02 | 1.08 | 5.70 | 5.81 | 1.54 |
| Energy ($pJ$) | 2.77 | 1.16 | 6.95 | 2.75 | 2.32 | 22.76 | 2.70 | 22.80 | 23.24 | 6.10 |
| Energy Savings | 0% | 58% | -151% | 1% | 16% | 0% | 88% | 0% | -2% | 73% |

Table 3.10: Top-1 and Top-5 accuracies of the different approximate multipliers on AlexNet

| | **Fixed** | **Mitch-$w8$** | **IterLog2** | **ASM** | **DRUM6** |
|---|---|---|---|---|---|
| **Top-1** | 58.3% | 58.2% | 58.2% | 41.6% | 58.2% |
| **Top-5** | 80.2% | 80.2% | 80.2% | 67.0% | 80.2% |

logic to the C++ ASM model. We did not include the rounding logic when performing the synthesis to evaluate energy and area.

Table 3.9 shows the synthesis results of the approximate multipliers to compare their costs, especially the energy savings compared to the exact fixed-point multiplier. Table 3.10 shows the Top-1 and Top-5 accuracies in the AlexNet experiment.

The 2-stage iterative log multiplier had been proposed because it was more accurate than the original Mitchell multiplication, but Mitch-$w$ performs as well as the iterative log multiplier in our experiments. The iterative log multiplier consumes much more power compared to Mitch-$w$, and adding the sign handling for the CNNs negates the power and energy benefits against the exact fixed-point multiplication.

The 1-alphabet ASM showed 16% power savings at 16 bits and only 3% power savings at 32 bits, and worse energy savings. The power savings are larger for the smaller bits as the original work only presented 8 and 12-bit designs. However, the real problem when applying the ASM to CNNs is the high amount of CNN accuracy degradation. The work on the ASM reported the trend where the performance degradation increased for more complex applications [58], and the continuation of the trend was observed for the ImageNet with a significant performance degradation. We had demonstrated earlier that the poor performance was due to the high error dispersion. The question may be raised as to why the ASMs with more alphabets were not considered. The ASM with more alphabets requires the pre-computed alphabet values and introduces the memory accesses that add significant costs to the system.

DRUM reported good characteristics and compared well against the ESSM8 and Kulkarni multipliers [17]. It was also based on the leading one detection like the log multipliers and naturally handled zero correctly, so the design received our attention though it had never been applied to a CNN. Our experiments show that DRUM6 produces the same good inference results as the proposed design, but Mitch-$w$ is significantly smaller and consumes less energy. DRUM6 had WCE of 6.3% that is both positive and negative, and it behaved similarly to the unbiased Mitch-$w$ at $w$=8, which has slightly higher WCE of 8.1%. Despite the fact that DRUM did not require the zero detection unit, the presented log multiplication algorithm was more effective at reducing the energy consumption.

### 3.6.8 Verification with More Images

In this section, the proposed designs are used to perform the AlexNet inference for the entire ILSVRC2012 validation set, in order to confirm that the observations made about the design decisions hold true for the entire 50,000 images. TABLE 3.11 shows the Top-1 and Top-5 accuracies on AlexNet with the proposed designs. The original Mitchell's algorithm showed little performance degradation compared against the exact multipliers. The $w$ truncation was introduced and decreased the Top-1 accuracy slightly but provided significant improvement

Table 3.11: The AlexNet accuracies with the entire ILSVRC2012 validation set

| | **Top-1** | **Top-5** |
|---|---|---|
| **Float** | 56.8% | 80.0% |
| **Fixed** | 56.8% | 79.9% |
| **Mitchell's Mult** | 56.6% | 79.7% |
| **Mitch-$w$6** | 56.5% | 79.7% |
| **Mitch-$w$8** | 56.5% | 79.7% |
| **Unbiased Mitch-$w$6** | 56.5% | 79.8% |
| **Unbiased Mitch-$w$8** | 56.6% | 79.8% |
| **Unbiased Mitch-$w$6 with C1** | 56.5% | 79.8% |
| **Unbiased Mitch-$w$8 with C1** | 56.6% | 79.8% |

in the energy consumption. The unbiasing techniques are then introduced to improve the accuracies slightly as it prevented the final score values from being compressed together. We can see that using more significant bits with $w$=8 results in slightly higher accuracies, but the designs at $w$=6 still perform reasonably well. Lastly, the results show that the proposed C1 approximation of the C2 sign conversion does not degrade the CNN performances, and is a more energy-efficient alternative. TABLE 3.11 shows that the ideas presented with the subset of 5,000 images remain true for the larger number of images.

## 3.7 Experiments with Very Deep Neural Networks

This section extends the methodology of approximate multiplication to very deep CNNs that have tens and even hundreds of layers. The experiments show that the proposed multipliers may perform as accurate inferences as exact multiplication for these deep CNNs, and they also provide the experimental evidences for the theories presented in Section 3.4 and 3.5.

Table 3.12: Pre-trained CNN models used for the experiments

| Network | Model Source | BatchNorm | Grouped Conv. |
|---|---|---|---|
| **VGG16** | [23] | | |
| **GoogLeNet** | [23] | | |
| **ResNet-50** | [18] | √ | |
| **ResNet-101** | [18] | √ | |
| **ResNet-152** | [18] | √ | |
| **Inception-v4** | [44] | √ | |
| **Inception-ResNet-v2** | [61] | √ | |
| **ResNeXt-50-32x4d** | [67] | √ | √ |
| **Xception** | [44] | √ | √ |
| **MobileNetV2** | [57] | √ | √ |

### 3.7.1 Setup

Similarly to Section 3.6, the experiments are performed in the Caffe framework to evaluate the impact of approximate multipliers on deep CNN models [23]. Caffe has limited features compared to contemporary tools, but its lack of encapsulation allows easy modification of underlying matrix multiplication, making it suitable for the study. The difference from the experiments in Section 3.6 is that the C++ functions that emulate the multipliers are replaced with CUDA C++ counterparts to accelerate the simulations with a GPU. These functions are verified against RTL simulations of the HDL code of the multipliers.

The Mitch-$w$6 multiplier with the 1's complement (C1) sign handling is chosen for the experiments because the comparison against the other multipliers in Section 3.6.7 showed that it was cost-efficient while performing well on AlexNet. DRUM6 multiplier [17] is also added to the experiments, because it performed very well on AlexNet while being more costly than Mitch-$w$6. The truncated iterative log multiplier in Section 2.3 has higher accuracy than these multipliers and is tested for networks that have depthwise separable convolution. The FP32 floating-point results are included for comparison, and the bfloat16 results provide additional data points (see Section 3.4).

The target application is object classification with the ImageNet ILSVRC2012 validation dataset of 50,000 images. Only single crops are used for experiments because the C++ emulation of the approximate multipliers is very time-consuming compared to the multiplication performed in actual hardware, so the presented CNN accuracies may differ from the original literature that use 10-crops. Table 3.12 shows the list of CNN models used for the experiments, and the networks that use batch normalization and grouped convolutions are marked for comparative discussion. The pre-trained CNN models for the experiments are publicly available from online repositories, and the source is indicated with each model. Any training or retraining a network model is purposefully avoided to achieve reproducibility and to show that the proposed methodology works with many network models with only minor scaling of batch normalization parameters.

The experiments assume the quantization to 32 bits (16 integer bits and 16 fractional bits) as it is sufficient for all the tested network models. Performing careful quantization to each network model may achieve lower number of bits, but the generous quantization allows the study of approximate multiplication in isolation while also providing a more general solution to the hardware acceleration of CNNs.

### 3.7.2 Impact on CNN Performance and Variance of Accumulated Error

Figure 3.15 shows the Top-5 errors when the approximate multipliers are applied to the CNNs, compared against the FP32 reference values. The Top-1 errors shown in Figure 3.16 show the pattern very similar to Top-5 errors, so the following discussion will focus only on Top-5 errors.

For the networks with conventional convolution, the studied approximate multipliers produce predictions that are nearly as accurate as the exact FP32 floating-point references. The CNNs

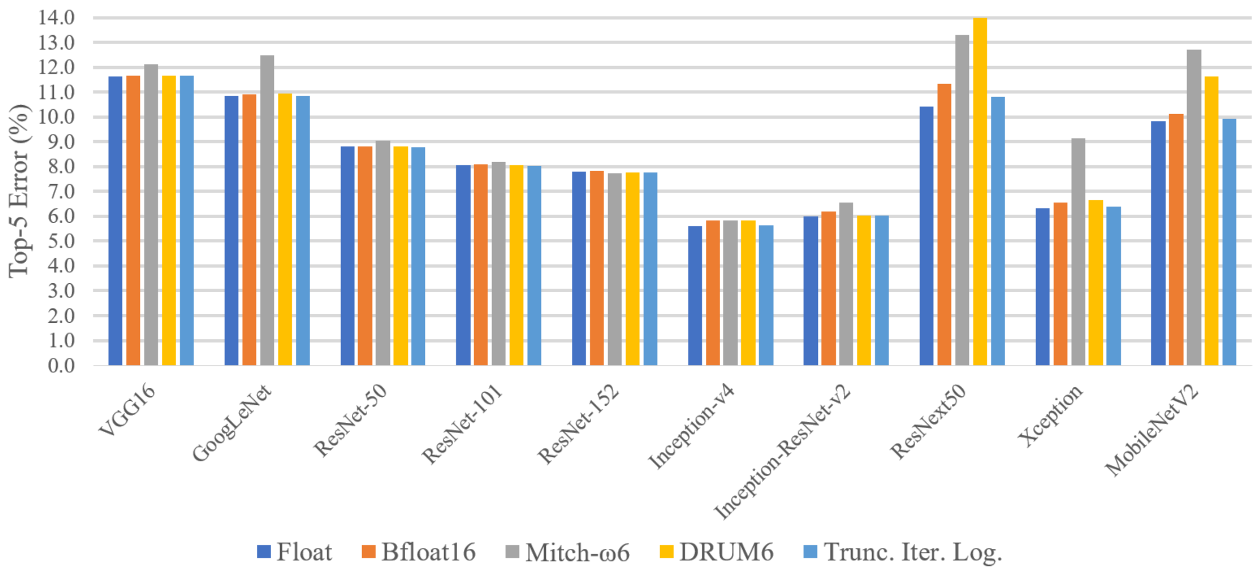


Figure 3.15: Comparison of Top-5 errors between the FP32 reference and the approximate multipliers

with grouped convolution suffer degraded accuracies when there are errors in multiplications, from approximate multiplication as well as bfloat16. The difference of CNN accuracies between different convolution types supports the hypothesis presented in Section 3.4.

For more evidence, all convolution outputs are extracted for the first five samples of ILSVRC2012 validation set, with FP32 and Mitch-$w6$ multiplications. The large size of the data makes it infeasible to perform the analysis for the entire set. The errors from approximate multiplication are measured by comparing the results. The variance of accumulated error within each channel is measured as well as the variance between the convolution outputs. The geometric means are taken across all channels as channels had wildly varying ranges of values. Table 3.13 shows the measured values for various CNNs. It demonstrates the increased variance of accumulated error for grouped and depthwise convolutions. The conventional convolution results also provide the evidence that the accumulated errors have much less variance compared to the distribution of outputs, and therefore have less impact on the functionality of feature detection.

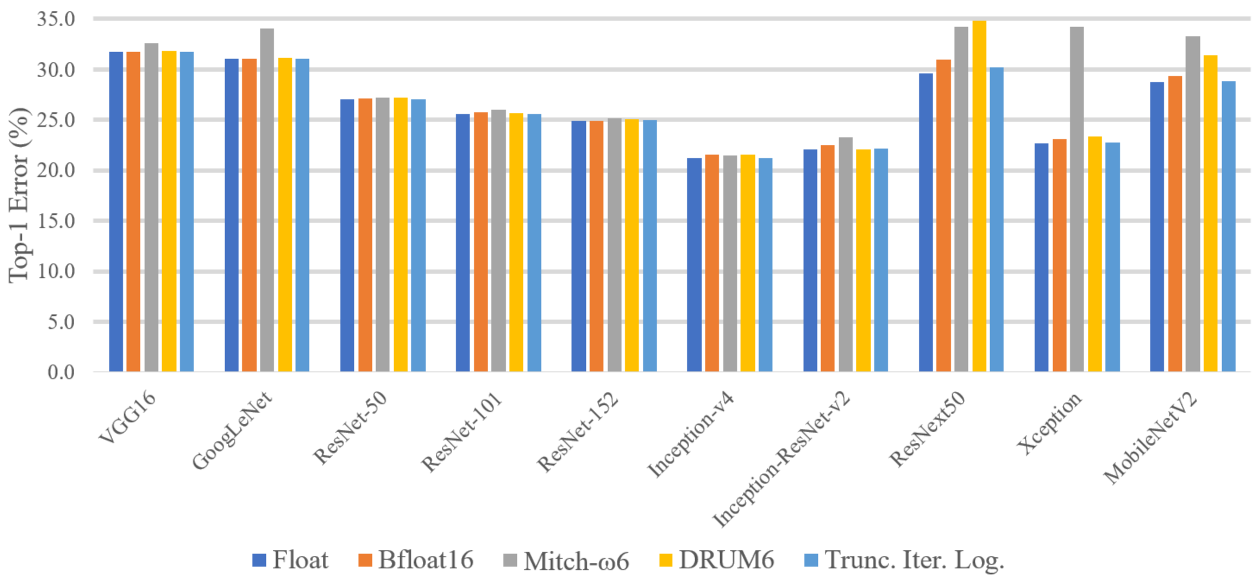


Figure 3.16: Comparison of Top-1 errors between the FP32 reference and the approximate multipliers

The measured variances in Table 3.13 do not directly correlate to the performance of Mitch-$w6$ in Figure 3.15 because Table 3.13 only shows the error variance within each channel and does not account for the error variance across channels. The approximate multiplication in ResNeXt-50-32x4d causes significant degradation in the prediction accuracy, because ResNeXt networks have many branches in their architectures where different amounts of error accumulate. The Inception networks have relatively shorter branches and show slightly

Table 3.13: Measured variance of accumulated error with Mitch-$w6$

| **Conv. Type** | **Network** | **Error Vari.** | **Output Vari.** | **Pct.** |
|---|---|---|---|---|
| **Conventional** | **ResNet-50** | 1.72E-3 | 5.64E-2 | 3.1% |
| | **ResNet-101** | 1.98E-3 | 3.94E-2 | 5.0% |
| | **ResNet-152** | 2.03E-3 | 3.26E-2 | 6.2% |
| | **Inception-v4** | 6.90E-3 | 1.11E-1 | 6.2% |
| | **Inception-ResNet-v2** | 1.19E-3 | 1.78E-2 | 6.7% |
| **Grouped** | **ResNeXt-50-32x4d** | 1.32E-4 | 1.30E-3 | 10.1% |
| **Depthwise** | **Xception** | 1.59E-2 | 7.91E-2 | 20.1% |
| | **MobileNetV2** | 1.48E-2 | 1.16E-1 | 12.7% |

more degradation compared to the ResNet models that have none. The theoretical principle discussed in Section 3.4 agrees with this analysis, though Table 3.13 could not capture these differences.

For CNNs with grouped convolutions, a sufficiently accurate approximate multiplier can still be used to perform accurate inferences, as demonstrated with the truncated iterative log multiplier in Figure 3.15. When the converging effect of accumulation is reduced, the variance of accumulated error may be reduced by producing a smaller range of errors, at the cost of more hardware resources.

### 3.7.3 Effects of Batch Normalization

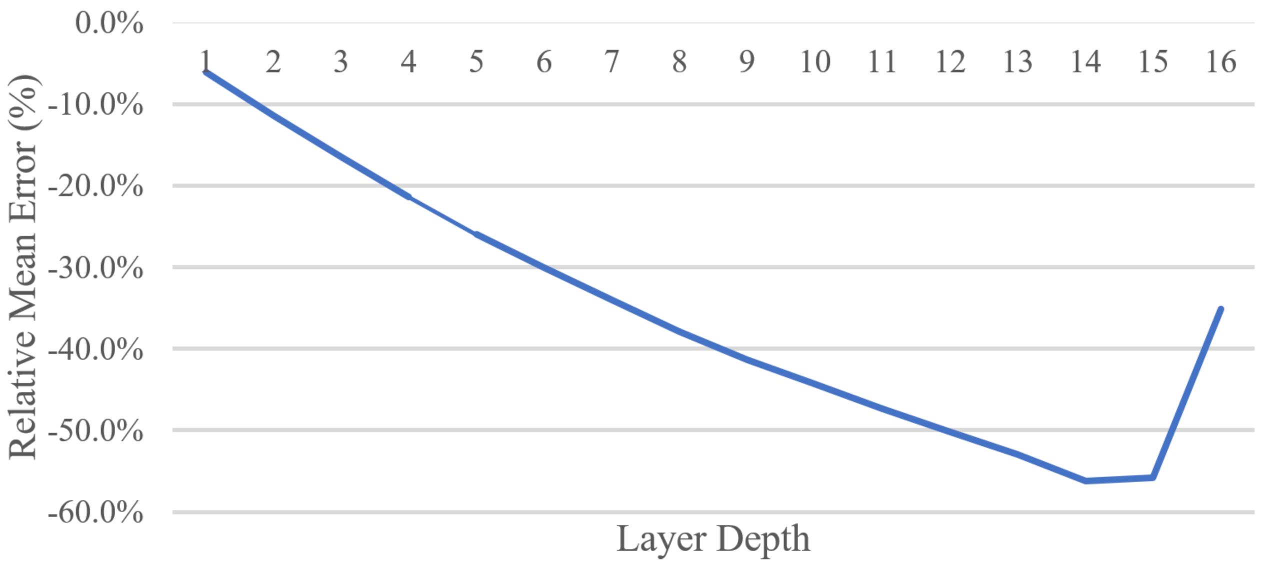


Figure 3.17: Accumulation of mean error on VGG16

Figure 3.17 demonstrates the accumulation of mean error in VGG16 with Mitch-$w6$, averaged over the first five samples. Because the network lacks batch normalization, the deeper layers receive the inputs that are repeatedly scaled down when the errors in multiplication are biased. It explains the poor performance of Mitch-$w6$ on VGG16 and GoogLeNet in Figure

Table 3.14: Impact of batch normalization adjustment with Mitch-$w6$

| | **Top-1 Error** | | **Top-5 Error** | |
|---|---|---|---|---|
| | **Original** | **Adjusted** | **Original** | **Adjusted** |
| **ResNet-50** | 31.7% | 27.2% | 10.5% | 9.0% |
| **ResNet-101** | 31.8% | 26.0% | 12.0% | 8.2% |
| **ResNet-152** | 31.2% | 25.2% | 11.5% | 7.7% |

3.15, while the unbiased DRUM6 performs well.

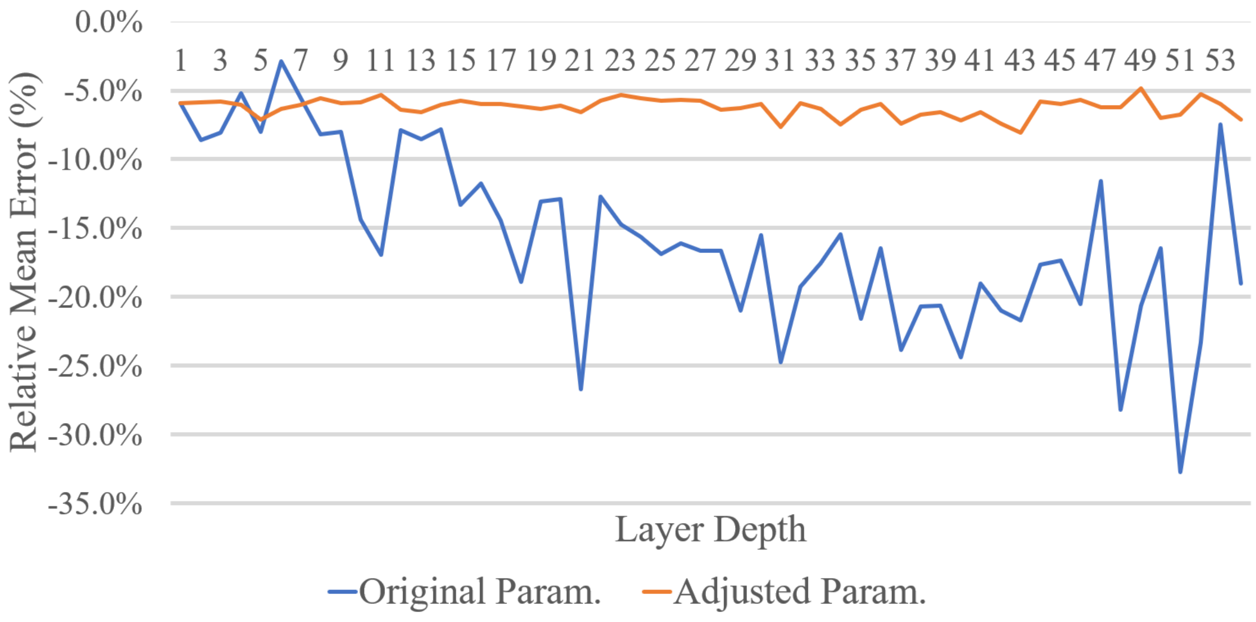


Figure 3.18: Effect of batch normalization on ResNet-50

Figure 3.18 shows the effect of batch normalization with properly adjusted parameters, on ResNet-50 with Mitch-$w6$ averaged over the first five images. For Mitch-$w6$ with the mean error of -5.9%, the mean and variance parameters in batch normalization are scaled by 0.941 and 0.885 respectively. With the proper adjustments, batch normalization eliminates the accumulation of mean error across layers and helps approximate multiplication work with deep CNNs. Figure 3.18 shows that the mean error per layer hovers around the mean error of

Mitch-$w6$, which supports the convergence of accumulated error as well as the effectiveness of the adjusted batch normalization. Failing to adjust the parameters not only accumulates error in deeper layers, but also becomes an additional source of error with incorrect redistribution of feature maps, resulting in an unstable pattern of accumulated error. Table 3.14 shows the impact on the Top-1 and Top-5 errors of the ResNet models. Incorrect batch normalization results in performance degradation, while the corrected batch normalization layers help approximate multiplication perform well for deep ResNet models.

# Chapter 4

# Integrating Approximate Multipliers into an FPGA Accelerator

This chapter presents the design of a convolution core that utilizes approximate log multipliers to significantly reduce the power consumption of the FPGA acceleration. The computations in CNN convolution layers have regular pattern and can be scheduled statically, thus presenting the opportunity for acceleration on FPGAs that have massively parallel hardware with low-throughput memory. The large amount of multiplication operations makes it rewarding to reduce the cost through approximate multiplication [13].

The convolution core not only demonstrates that approximate multipliers can be easily integrated into FPGA and ASIC accelerator designs, but also shows that optimizing the multipliers has a large impact on the resource consumption of such designs. The core also exploits FPGA reconfigurability as well as the parallelism and input sharing opportunities in convolutional layers to further minimize the costs. The contents presented in this chapter were published in [53].

There are many works such as [54, 16, 69] that presented FPGA accelerators for CNNs. Many of these works focus on quantization and reduce the precision of numerical representation to reduce the cost of implementation. As discussed in Section 3.2, the range and precision required to maintain comparable accuracy depends on each network and layer [25, 37, 54], and simply reducing the number of bits may not suffice as networks become more complex. Approximate multiplication is a potentially effective alternative for reducing the costs of FPGA/ASIC accelerators for very deep and complex CNNs.

## 4.1 Design of the Convolutional Core

Currently, the most current state-of-the-art accelerators implemented in ASICs such as Google's TPU [24] make use of systolic arrays for accelerating neural networks. However, due to the necessity of conversion from convolution to matrix multiplication and the need of a high bandwidth memory, Systolic Arrays are unfeasible for FPGA implementation, making multiple accelerators such as [64] implement highly specific convolution cores for CNN acceleration.

The convolution core proposed here is a generalization of the one proposed in [14]. This implementation of convolution core, as seen in Figure 4.1, uses registers for emulating the convolution window throughout the input feature, resulting in a convolution core that is highly specific for convolutions and scalable for different kernel filter and input feature sizes.

The weight storage depicted in Figure 4.1 is a FIFO that receives all weights from memory and saves them in registers, which are later indexed to each multiplier during convolution. The reconfiguration of FPGAs allows the weights to be stored in on-chip ROM, thus removing the overhead of communication between the host processor and FPGA for weight loading.

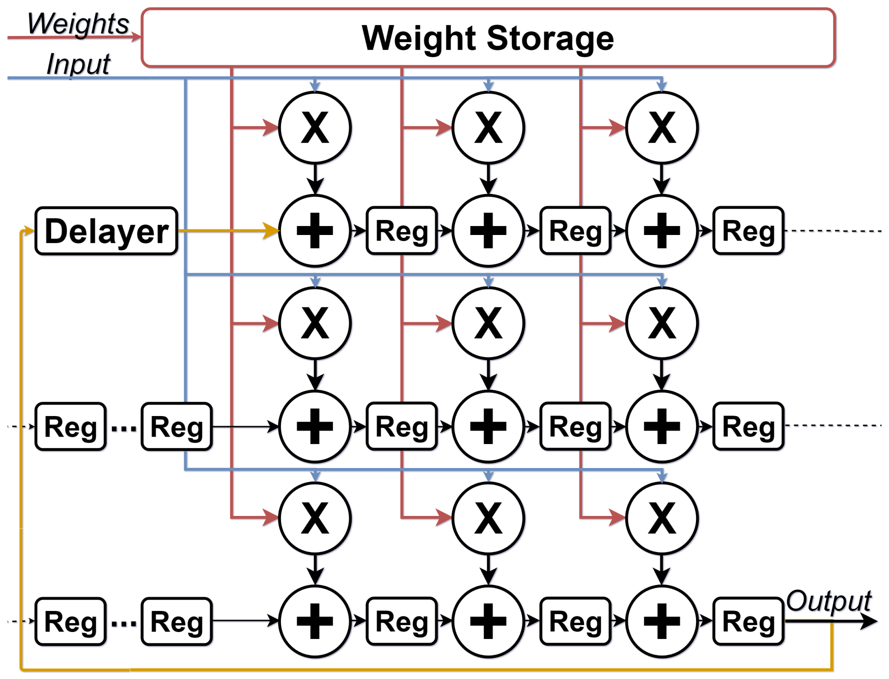


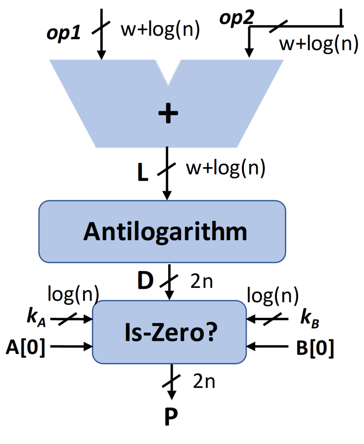
Figure 4.1: The convolution core of 3x3 Filter Size

## 4.2 Reduced Mitchell Log Multiplier

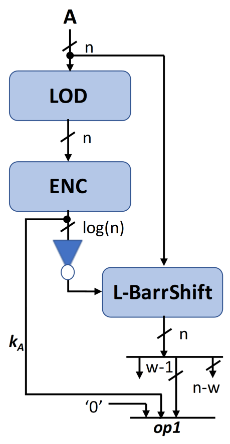
With $n_okernels$ being a divisor of the number of kernels of a convolution layer, $n_okernels$ convolution cores could be run in parallel. This allows the increase of the throughput by $n_okernels$-fold when compared to a single convolution core, thus granting the usage of the convolution core in a wide variety of FPGAs with different amount of resources.

Derived from Mitch-$w$ multiplier proposed in Section 2.2 and taking advantage of the characteristics of the proposed convolution core, a Reduced Mitchell Log Multiplier (RMitch-$w$) is designed for the convolution core. The main characteristic of RMitch-$w$ is the removal of the encoding process of Mitch-$w$, moving it to a separate module called *Feature Extractor*, as shown in Figure 4.2b.

As depicted in Figure 4.1, the convolution core structure allows the removal of redundancy in the multipliers due to input sharing, resulting in Figure 4.2a. Whereas in the Mitch-$w$ version each multiplier received the input and encoded it, RMitch-$w$ receives a tuple



(a) Reduced Multiplier



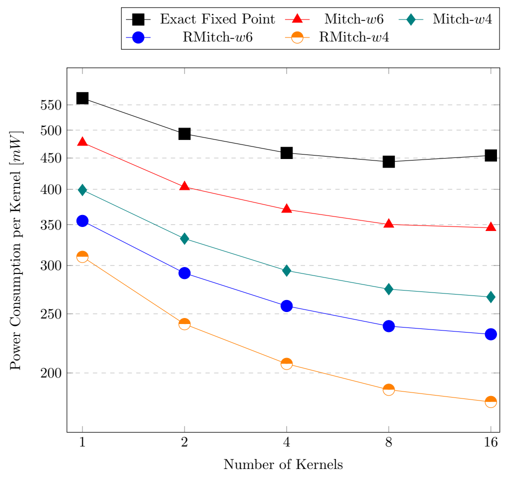
(b) Feature Extractor

Figure 4.2: Reduced Truncated Approximate Log Multiplier, RMitch-$w$

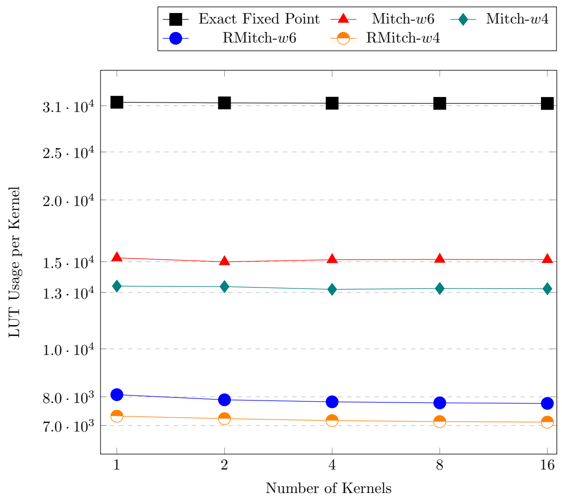
$(A[0], A[n-1], opA[\log_2 n + w - 2 : 0])$ of size $\log_2 n + w + 1$ bits from a global Feature Extractor and a previously-encoded weight from the weight storage in the same tuple structure.

Before FPGA synthesis, the network's weights are converted to the tuple structure and saved to a COE file for later use in the FPGA's ROM, resulting in a reduction of approximately 60% of LUT usage per multiplier.

Another effect of the prior encoding of the weights is a reduction of the memory footprint needed for storing the weights. For instance, when processing AlexNet [33] (approx. 3.7 million parameters in convolution layers) with RMitch-$w4$ 32 bit multiplier, the memory footprint required for the convolution layers' weights reduce from $3.7M * 4Bytes \approx 14.8$MB to $3.7M * 10bits/8 \approx 4.6$MB, a reduction of 68.92% that further enables the storage of its weights in ROMs inside the FPGA, greatly reducing the communication overhead for weight loading.



(a) Power consumption per Kernel



(b) LUT per Kernel

Figure 4.3: Scalability of the convolution core by number of kernels

## 4.3 Experimental Results

Using Vivado 2017.4, with ZYNQ-7 ZC702 Board (part xc7z020clg484-1) selected, Vivado's default settings for power estimation and 32 bits in the Q16.16 format, Figures 4.3a and 4.3b were generated, where the scalability of the number of kernels in terms of power consumption and LUT usage are compared between Exact Fixed Point, Mitch-$w$ and RMitch-$w$ multipliers. For fairness of comparison, DSP units were disabled in the convolution core, repurposing them for operations that require greater accuracy, such as accelerating the fully connected layers.

Figure 4.3a shows that the relative power consumption converges to a horizontal asymptote. With one Kernel, RMitch-$w4$ achieves a relative reduction of 45.03% and 22.31% in power consumption when compared to the Exact Fixed Point and Mitch-$w4$ multipliers. The results at 16 kernels show even better results, with relative reductions of 60.54% and 32.68% when compared to the same multipliers with 16 Kernels.

Finally, by extracting the weights from the LeNet Network available as a sample at Caffe's

|  | Exact Fixed Point | $w_1 = 6, w_2 = 6$ | $w_1 = 4, w_2 = 4$ |
|---|---|---|---|
| **LUT Usage** $[k]$ | 31.5 + 30.4 ≈ 61.9 | 8.1 + 6.9 ≈ 15.0 | 7.3 + 6.2 ≈13.5 |
| **FF Usage** $[k]$ | 3.7 + 3.1 ≈6.8 | 3.2 + 2.6 ≈5.8 | 3.1 + 2.6 ≈ 5.7 |
| **Estimated Power** $[mW]$ | 564 + 551 ≈ 1115 | 354 + 350 ≈ 704 | 310 + 300 ≈ 610 |
| **Network Accuracy** | 99.1% | 99.0% | 99.1% |

Table 4.1: Comparison of the synthesis results for the LeNet accelerator

Github repository[1], a simulated network using different $w$ parameters for each layer was executed, generating Table 4.1. Due to time constraints, the simulated network executed with a batch size of 1000 (10% of the original test set), with the simulation environment Vivado 2017.4, a single kernel for each layer for power consumption, LUT and FF usage estimation.

With reductions of respectively 75.77% and 78.19% in LUT Usage, 14.7% and 16.17% in FF Usage, 36.86% and 45.29% in Estimated Power Consumption and drops in accuracy within margin of error, the results of RMitch-$w$6 and RMitch-$w$4 from Table 4.1 further confirm the results showcased in Section 3.6, in which the Mitch-$w$ multiplier achieved an accuracy of 99.0% in LeNet.

[1]https://github.com/BVLC/caffe/tree/master/examples/mnist

# Chapter 5

# Conclusions

This dissertation proposes the approximate log multipliers based on Mitchell's Algorithm that can save significant amount of hardware costs for CNN inferences. The low-power implementation of the Mitchell Log Multiplier was created with the improved LOD block and the C1-based shift amount calculations, as well as the optimization of the decoder and the introduction of the zero detection unit to improve the CNN performances. We have introduced the additional approximating techniques of the $w$ truncation and the C1 sign handling, and provided the formal analysis of the errors as well as the experimental results to show that they are viable for CNNs. We have also shown that the multiplier may be iterated to improve the accuracy when higher accuracy is required by certain CNN architectures and applications.

While evaluating the proposed designs for CNNs, we have also made various observations that provide deeper understanding of the effect of approximate multiplication. The analysis provide a detailed explanation of why CNNs are resilient against the errors in multiplication. Specifically, we have identified that the variations in error accumulation can impact the inference accuracies of CNNs, and suggested that the approximate multipliers should seek to minimize the range of error. Approximate multiplication favors the wide convolution layers with many input channels, and batch normalization layers can be adjusted for deeper networks, making it a promising approach as the networks become wider and deeper to handle various real-world applications.

Lastly, by exploiting FPGA's reconfigurability and the characteristics of convolution, a convolution core using approximate multipliers was implemented to achieve reductions of up to 78.19% of LUT usage and 60.54% of power consumption when compared to the core that uses exact fixed-point multiplier, while maintaining comparable accuracy on a subset of MNIST dataset.

This dissertation makes significant contributions toward the state-of-the-art of approximate computing and cost-efficient CNN inference, and the most significant contributions are summarized as follows:

- Designing approximate multipliers that show large cost improvements over the state-of-the-art.
- Identifying the reasons why approximate multiplication is viable for CNN inferences.
- Extending the methodology of approximate multiplication to very deep CNNs with batch normalization.
- Designing a convolution core with approximate multipliers and demonstrating the large benefits of approximate multiplication on CNN accelerators.

Collectively, this dissertation demonstrates that approximate multiplication is a promising approach to significantly improve the efficiency of CNN inferences.

# Bibliography

[1] K. H. Abed and R. E. Siferd. Cmos vlsi implementation of a low-power logarithmic converter. *IEEE Transactions on Computers*, 52(11):1421–1433, Nov 2003.

[2] M. S. Ansari, V. Mrazek, B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han. Improving the accuracy and hardware efficiency of neural networks using approximate multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[3] Z. Babic, A. Avramovic, and P. Bulic. An iterative mitchell's algorithm based multiplier. In *Signal Processing and Information Technology, 2008. ISSPIT 2008. IEEE International Symposium on*, pages 303–308. IEEE, 2008.

[4] Z. Babić, A. Avramović, and P. Bulić. An iterative logarithmic multiplier. *Microprocess. Microsyst.*, 35(1):23–33, Feb. 2011.

[5] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[6] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Design Automation Conference*, pages 555–560. IEEE, 2010.

[7] F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[8] M. Courbariaux, Y. Bengio, and J.-P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3123–3131. 2015.

[9] S. De, J. Huisken, and H. Corporaal. Designing energy efficient approximate multipliers for neural acceleration. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 288–295. IEEE, 2018.

[10] A. Del Barrio, M. S. Kim, R. Hermida, and N. Bagherzadeh. log-arithmetic. `https://github.com/albertodbg/log-arithmetic`.

[11] A. A. Del Barrio, N. Bagherzadeh, and R. Hermida. Ultra-low-power adder stage design for exascale floating point units. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):1–24, 2014.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[13] Z. Du, K. Palem, A. Lingamneni, O. Temam, Y. Chen, and C. Wu. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 201–206, 2014.

[14] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37. IEEE, 2009.

[15] I. Hammad and K. El-Sankary. Impact of approximate multipliers on vgg deep learning network. *IEEE Access*, 6:60438–60444, 2018.

[16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

[17] S. Hashemi, R. I. Bahar, and S. Reda. Drum: A dynamic range unbiased multiplier for approximate applications. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 418–425, 2015.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[19] G. Henry, P. T. P. Tang, and A. Heinecke. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 69–76. IEEE, 2019.

[20] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016.

[21] M. Imani, M. Masich, D. Peroni, P. Wang, and T. Rosing. Canna: Neural network acceleration using configurable approximation on gpgpu. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 682–689. IEEE, 2018.

[22] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, pages 675–678, 2014.

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[25] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[26] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Quality control for approximate accelerators by error prediction. *IEEE Design & Test*, 33(1):43–50, 2016.

[27] D. Kim, J. Kung, and S. Mukhopadhyay. A power-aware digital multilayer perceptron accelerator with on-chip training based on approximate computing. *IEEE Transactions on Emerging Topics in Computing*, 5(2):164–178, 2017.

[28] H. Kim, M. S. Kim, A. A. Del Barrio, and N. Bagherzadeh. A cost-efficient iterative truncated logarithmic multiplication for convolutional neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 108–111. IEEE, 2019.

[29] M. S. Kim, A. A. D. Barrio, R. Hermida, and N. Bagherzadeh. Low-power implementation of mitchellś approximate logarithmic multiplication for convolutional neural networks. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, pages 617–622, 2018.

[30] M. S. Kim, A. A. Del Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh. Efficient mitchell's approximate log multipliers for convolutional neural networks. *IEEE Transactions on Computers*, 68(5):660–675, 2018.

[31] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, Aug 1973.

[32] A. Krizhevsky, V. Nair, and G. Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, pages 1097–1105, 2012.

[34] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pages 346–351. IEEE, 2011.

[35] J. Kung, D. Kim, and S. Mukhopadhyay. A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses. In *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*, pages 85–90. IEEE, 2015.

[36] K. Y. Kyaw, W. L. Goh, and K. S. Yeo. Low-power high-speed multiplier for error-tolerant application. In *Electron Devices and Solid-State Circuits (EDSSC), 2010 IEEE International Conference of*, pages 1–4. IEEE, 2010.

[37] L. Lai, N. Suda, and V. Chandra. Deep convolutional neural network inference with floating-point weights and fixed-point activations, 2017.

[38] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[39] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong. Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904. IEEE, 2017.

[40] D. Lin, S. Talathi, and S. Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[41] C. Liu, J. Han, and F. Lombardi. A low-power, high-performance approximate multiplier with configurable partial error recovery. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4. IEEE, 2014.

[42] W. Liu, J. Xu, D. Wang, and F. Lombardi. Design of approximate logarithmic multipliers. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 47–52, 2017.

[43] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi. Design and evaluation of approximate logarithmic multipliers for low power error-tolerant applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(9):2856–2868, 2018.

[44] Y. Liu, C. Chen, R. Zhang, T. Qin, H. Lin, and M. Yang. Enhancing the interoperability between deep learning frameworks by model conversion. Technical Report MSR-TR-2019-36, Microsoft, November 2019.

[45] U. Lotrič and P. Bulić. Applicability of approximate multipliers in hardware neural networks. *Neurocomput.*, 96:57–65, Nov. 2012.

[46] V. Mahalingam and N. Ranganathan. An efficient and accurate logarithmic multiplier based on operand decomposition. In *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06)*, 2006.

[47] J. N. Mitchell. Computer multiplication and division using binary logarithms. *IRE Transactions on Electronic Computers*, EC-11(4):512–517, Aug 1962.

[48] D. Miyashita, E. H. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[49] A. Momeni, J. Han, P. Montuschi, and F. Lombardi. Design and analysis of approximate compressors for multiplication. *IEEE Transactions on Computers*, 64(4):984–994, April 2015.

[50] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, 2016.

[51] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique. Alwann: Automatic layer-wise approximation of deep neural network accelerators without retraining. *arXiv preprint arXiv:1907.07229*, 2019.

[52] S. Narayanamoorthy, H. A. Moghaddam, Z. Liu, T. Park, and N. S. Kim. Energy-efficient approximate multiplication for digital signal processing and classification applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(6):1180–1184, 2015.

[53] L. T. Oliveira, M. S. Kim, A. A. Del Barrio, N. Bagherzadeh, and R. Menotti. Design of power-efficient fpga convolutional cores with approximate log multiplier. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 203–208, 2019.

[54] J. Qiu et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35, 2016.

[55] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[56] S. Salamat, M. Imani, S. Gupta, and T. Rosing. Rnsnet: In-memory neural network acceleration using residue number system. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–12. IEEE, 2018.

[57] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[58] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy. Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 145–150, 2016.

[59] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[60] Y. Shim, A. Sengupta, and K. Roy. Low-power approximate convolution computing unit with domain-wall motion based "spin-memristor" for image processing applications. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[61] N. Silberman. Tf-slim: A lightweight library for defining, training and evaluating complex models in tensorflow. 2017.

[62] X. Sun, X. Peng, P.-Y. Chen, R. Liu, J.-s. Seo, and S. Yu. Fully parallel rram synaptic array for implementing binary neural network with (+ 1,- 1) weights and (+ 1, 0) neurons. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 574–579. IEEE, 2018.

[63] C. Szegedy et al. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[64] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2220–2233, 2017.

[65] S. Vahdat, M. Kamal, A. Afzali-Kusha, and M. Pedram. Tosam: An energy-efficient truncation-and rounding-based scalable approximate multiplier. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(5):1161–1173, 2019.

[66] S. Wang and P. Kanwar. Bfloat16: the secret to high performance on cloud tpus. *Google Cloud Blog, August*, 2019.

[67] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. *arXiv preprint arXiv:1611.05431*, 2016.

[68] R. Zendegani, M. Kamal, M. Bahadori, A. Afzali-Kusha, and M. Pedram. Roba multiplier: A rounding-based approximate multiplier for high-speed yet energy-efficient digital signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):393–401, Feb 2017.

[69] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[70] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu. Approxann: An approximate computing framework for artificial neural network. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 701–706. IEEE, 2015.