Flexible Multiple-Precision Fused Arithmetic Units for Efficient Deep Learning Computation

A Thesis Submitted to the

College of Graduate and Postdoctoral Studies

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

in the Department of Electrical and Computer Engineering

University of Saskatchewan

Saskatoon, Saskatchewan, Canada

By

Hao Zhang

©Hao Zhang, September 2019. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Electrical and Computer Engineering
3B48 Engineering Building
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan S7N 5A9
Canada

Or

Dean of College of Graduate and Postdoctoral Studies
116 Thorvaldson Building
University of Saskatchewan
110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

Deep Learning has achieved great success in recent years. In many fields of applications, such as computer vision, biomedical analysis, and natural language processing, deep learning can achieve a performance that is even better than human-level. However, behind this superior performance is the expensive hardware cost required to implement deep learning operations. Deep learning operations are both computation intensive and memory intensive. Many research works in the literature focused on improving the efficiency of deep learning operations. In this thesis, special focus is put on improving deep learning computation and several efficient arithmetic unit architectures are proposed and optimized for deep learning computation. The contents of this thesis can be divided into three parts: (1) the optimization of general-purpose arithmetic units for deep learning computation; (2) the design of deep learning specific arithmetic units; (3) the optimization of deep learning computation using 3D memory architecture.

Deep learning models are usually trained on graphics processing unit (GPU) and the computations are done with single-precision floating-point numbers. However, recent works proved that deep learning computation can be accomplished with low precision numbers. The half-precision numbers are becoming more and more popular in deep learning computation due to their lower hardware cost compared to the single-precision numbers. In conventional floating-point arithmetic units, single-precision and beyond are well supported to achieve a better precision. However, for deep learning computation, since the computations are intensive, low precision computation is desired to achieve better throughput. As the popularity of half-precision raises, half-precision operations are also need to be supported. Moreover, the deep learning computation contains many dot-product operations and therefore, the support of mixed-precision dot-product operations can be explored in a multiple-precision architecture. In this thesis, a multiple-precision fused multiply-add (FMA) architecture is proposed. It supports half/single/double/quadruple-precision FMA operations. In addition, it also supports 2-term mixed-precision dot-product operations. Compared to the conventional multiple-precision FMA architecture, the newly added half-precision support and mixed-precision dot-product only bring minor resource overhead. The proposed FMA can be

used as general-purpose arithmetic unit. Due to the support of parallel half-precision computations and mixed-precision dot-product computations, it is especially suitable for deep learning computation.

For the design of deep learning specific computation unit, more optimizations can be performed. First, a fixed-point and floating-point merged multiply-accumulate (MAC) unit is proposed. As deep learning computation can be accomplished with low precision number formats, the support of high precision floating-point operations can be eliminated. In this design, the half-precision floating-point format is supported to provide a large dynamic range to handle small gradients for deep learning training. For deep learning inference, 8-bit fixed-point 2-term dot-product computation is supported. Second, a flexible multiple-precision MAC unit architecture is proposed. The proposed MAC unit supports both fixed-point operations and floating-point operations. For floating-point format, the proposed unit supports one 16-bit MAC operation or sum of two 8-bit multiplications plus a 16-bit addend. To make the proposed MAC unit more versatile, the bit-width of exponent and mantissa can be flexibly exchanged. By setting the bit-width of exponent to zero, the proposed MAC unit also supports fixed-point operations. For fixed-point format, the proposed unit supports one 16-bit MAC or sum of two 8-bit multiplications plus a 16-bit addend. Moreover, the proposed unit can be further divided to support sum of four 4-bit multiplications plus a 16-bit addend. At the lowest precision, the proposed MAC unit supports accumulating of eight 1-bit logic AND operations to enable the support of binary neural networks. Finally, a MAC architecture based on the posit format, a promising numerical format in deep learning computation, is proposed to facilitate the use of posit format in deep learning computation.

In addition to the above mention arithmetic units, an improved hybrid memory cube (HMC) architecture is proposed for weight-sharing deep neural network processing. By modifying the HMC instruction set and HMC logic layer, the major part of the deep learning computation can be accomplished inside memory. The proposed design reduces the memory bandwidth requirements and thus reduces the energy consumed by memory data transfer.

Acknowledgements

The research works presented in this thesis are sponsored by the Natural Sciences and Engineering Research Council (NSERC) of Canada and the Department of Electrical and Computer Engineering at the University of Saskatchewan.

I wish to express my sincere thanks to my supervisor, Dr. Seok-Bum Ko, for his guidance and support during my Ph.D. program. At the beginning of my Ph.D. program, Dr. Ko allowed me to carry on the computer arithmetic research from my master program and he gave me much freedom in specific research topic selection. I could connect computer arithmetic with deep learning computation. During the period of my study, Dr. Ko gave me much help not only in academic research but also in daily life.

I would like to place on record my gratitude to Dr. Dongdong Chen. We started working together from the beginning of my master program. Our cooperation continued during my Ph.D. program. We discussed research ideas every week. His experience in research really gave me great help and we have a great period of cooporation.

My sincere thanks also go to Dr. Li Chen, Dr. Anh V. Dinh, Dr. Francis M. Bui, and other professors at the University of Saskatchewan who had taught me. I learned many advanced knowledges from their lectures which not only broadened my horizon but also gave me many research ideas.

Thanks to my committee members Dr. Keshab K. Parhi, Dr. Raymond J. Spiteri, Dr. Li Chen, Dr. C.Y. (Tony) Chung and Dr. Anh V. Dinh for their time and effort to review my thesis and for their suggestions to imporve the quality of this thesis.

I also thank every technical staffs in the Department of Electrical and Computer Engineering who helped me setup my research environment and helped me solve the technique issues I encountered when using the development softwares.

Finally, a big thanks to my parents, my family, and my friends for their moral support throughout the years.

This is the thesis dedicated to my beloved parents.

Contents

Permission to Use . . . . **i**

Abstract . . . . **ii**

Acknowledgements . . . . **iv**

Contents . . . . **vi**

List of Tables . . . . **x**

List of Figures . . . . **xi**

List of Abbreviations . . . . **xiii**

Part I Preface **1**

Chapter 1 Introduction **2**

1.1 Deep Learning Advantages and Potentials 2

1.2 Motivation of Research Works 4

1.3 Overview of Research Works 7

1.4 Summary of Contributions 10

Chapter 2 Background **13**

2.1 Numerical Formats 13

2.1.1 Floating-Point Format 13

2.1.2 Fixed-Point Format 18

2.1.3 The Posit Format 19

2.2 Fused Arithmetic Units 21

2.3 Mixed-Precision Arithmetic Units 24

2.4 Deep Learning Computing 25

2.4.1 Overview . . . 25
2.4.2 Reduced Precision Computations . . . 27
2.4.3 Range vs Precision . . . 28
2.5 Hybrid Memory Cube . . . 30

Part II Arithmetic Unit for General Purpose Computation . . . 32

Chapter 3 Multiple-Precision Floating-Point Fused Multiply Add . . . 33
3.1 Introduction . . . 33
3.2 The Proposed Design . . . 36
3.2.1 Input Processing . . . 39
3.2.2 Mantissa Multiplier . . . 40
3.2.3 Alignment Shifter . . . 43
3.2.4 Adder . . . 50
3.2.5 Leading Zero Anticipation and Counting . . . 51
3.2.6 Normalization . . . 53
3.2.7 Rounding . . . 54
3.3 Results and Analysis . . . 54
3.4 Summary . . . 62

Part III Arithmetic Unit for Deep Learning Computation . . . 63

Chapter 4 Multiple-Precision Multiply Accumulate Unit . . . 64
4.1 Introduction . . . 64
4.2 The Proposed Design . . . 65
4.2.1 Floating-Point Mode . . . 67
4.2.2 Fixed-Point Mode . . . 68
4.2.3 Merged Multiplier Design . . . 68
4.3 Results and Analysis . . . 70
4.4 Summary . . . 72

Chapter 5 Flexible Multiple-Precision Multiply Accumulate Unit **73**
5.1 Introduction 73
5.2 Supported Numerical Formats 78
5.2.1 Numerical Format 78
5.3 The Proposed Design 79
5.3.1 Input Processing 82
5.3.2 Flexible Multiple-Precision Multiplier 85
5.3.3 Alignment Control and Shifter 89
5.3.4 Addition 91
5.3.5 Leading Zero Anticipator and Counting 92
5.3.6 Normalization Shifting 92
5.3.7 Rounding 93
5.3.8 Output Processing 94
5.4 Results and Analysis 94
5.5 Case Study 100
5.6 Discussion 103
5.7 Summary 106

Chapter 6 Posit Multiply Accumulate Unit **108**
6.1 Introduction 108
6.2 The Proposed Design 111
6.2.1 Posit Component Extraction 111
6.2.2 Mantissa Multiplier 112
6.2.3 Alignment Shifter 112
6.2.4 Adder 113
6.2.5 Leading Zero Anticipator 113
6.2.6 Normalization Shifter 113
6.2.7 Posit Output Process and Rounding 114
6.3 Results and Analysis 115
6.4 Summary 118

Part IV Deep Learning Computation using 3D Memory 119

Chapter 7 Improved Hybrid Memory Cube Architecture 120

7.1 Introduction 120

7.2 Weight-Sharing Deep Neural Network 122

7.3 The Proposed HMC Architecture 123

7.3.1 Instruction Set 124

7.3.2 HMC Request Format 125

7.3.3 HMC Memory Control 126

7.3.4 HMC Weight-Sharing Operation 127

7.4 Results and Analysis 128

7.5 Summary 130

Part V Conclusion 131

Chapter 8 Summary and Future Work 132

8.1 Summary 132

8.2 Future Work 138

References 141

List of Tables

1.1 Memory and computation requirements of various deep neural networks . . . 4

2.1 Floating-Point format defined in IEEE 754-2008 14
2.2 Results when both addition/subtraction operands are zeros 17
2.3 Comparison of the dynamic range of Posit format and floating-point format . 21
2.4 HMC Specifications 31

3.1 Synthesis results of each pipeline stage of the proposed FMA 55
3.2 Area and energy comparison of the proposed FMA with standalone FMA designs 56
3.3 Comparison of functionality with multiple-precision FMA designs 58
3.4 Comparison of the proposed FMA with previous works 59

4.1 Synthesis results of the combinational logic of each pipeline stage 70
4.2 Comparison of the proposed multiply-accumulate unit with single-mode multiply accumulate unit 71

5.1 Supported formats of the proposed MAC unit 77
5.2 Synthesis results of each pipeline stage of the proposed design 95
5.3 Comparison of the proposed MAC unit with standard arithmetic units . . . 98

6.1 Comparison of the Posit MAC with floating-point MAC 117
6.2 Delay of each pipeline stage of Posit MAC 117

7.1 Newly added instruction sets 124

8.1 Functions supported by the proposed MP-FMA and their applications 134
8.2 Functions supported by the proposed MP-MAC and their usage 135
8.3 Functions supported by the proposed Flex-MAC unit 136

List of Figures

1.1 Performance comparison of deep learning methods with previous methods . . 3

2.1 Binary floating-point format defined in IEEE 754-2008 13
2.2 Binary fixed-point format 18
2.3 Format of posit number 20
2.4 Implementing floating-point $A \times B + C$ with separate arithmetic units and fused arithmetic unit 22
2.5 Datapath of a mixed-precision FMA 24
2.6 Architecture of AlexNet 25
2.7 Process of convolution computation in deep neural network 26
2.8 Weights and activations distribution of AlexNet Convolution Layer 2 27
2.9 Normalized neural network accuracy under different exponent and mantissa bit-width 29
2.10 Basic hybrid memory cube architecture 30

3.1 Datapath of the proposed multiple-precision fused multiply-add unit 37
3.2 Unified mantissa format for different precisions 40
3.3 Partial products of the 15×15 radix-4 Booth multiplier 40
3.4 Enabled multipliers and product regions in different precision modes 41
3.5 Alignment of C operand in the proposed fused multiply-add unit 44
3.6 Unified input format for the alignment shifter of the proposed fused multiply-add unit 45
3.7 Design details of the alignment shifter of the proposed fused multiply-add unit 46
3.8 Alignment of products in mixed-precision dot-product mode 48
3.9 Unified input format of products alignment in mixed-precision operation mode 50
3.10 Diagram of the adder used in the proposed fused multiply-add 51
3.11 Unified input format for the leading zero anticipator and counting of the proposed fused multiply-add unit 52

4.1 Data-path of the proposed fixed/floating-point merged mixed-precision multiply-accumulate unit 66
4.2 Architecture of the proposed fixed/floating-point merged multiplier 68
4.3 Partial products arrangement of modified booth multiplier 69
4.4 Area-Delay and Power-Delay curve of the proposed design 72

5.1 Datapath of the proposed flexible multiple-precision multiply-accumulate unit and its usage in deep learning processor 79
5.2 Datapath of conventional multiply-accumulate unit based on standard floating-point format 81
5.3 Circuit to generate the mantissa mask mm_hi and mm_lo 83
5.4 Format of mantissa mask mm, exponent mask em, and implicit bit mask mm_pone 84
5.5 Diagram of input processing for A operand 85
5.6 Partial product generation and partial product array of the proposed flexible multiple-precision multiplier 86
5.7 Partial product array of each precision mode (white dots represent the bits not used; black dots represent partial products) 87
5.8 Alignment shifting of C with product in FLP16 mode 88
5.9 Alignment shift of C and two products in 8-bit floating-point mode 89
5.10 Addition arrangement and the leading zero anticipator and counting (LZAC) range of the proposed unit 91
5.11 Rounding method for floating-point modes 91
5.12 Rounding method for 16-bit fixed-point mode 93
5.13 Power-Delay curve and Area-Delay curve of the proposed design under STM-28nm 96
5.14 Power consumption of the proposed design in various operational modes (with timing constraint 0.27 ns) 97
5.15 Relative throughput of three implementations for six neural networks 102
5.16 Relative power efficiency of three implementations for six neural networks 102

6.1 Datapath of the proposed posit multiply-accumulate unit architecture (The proposed generator will give combinational design. Pipeline in this figure is just an example.) 110
6.2 Diagram of posit format component extraction 111
6.3 Datapath of the alignment shifter 112
6.4 Diagram of posit output processing and rounding 114
6.5 Rounding of the posit result 114
6.6 Area and power of 32-bit Posit MAC with various exponent bit-width (timing constraint is 1.6ns) 116

7.1 Process of convolution using weight sharing method 123
7.2 Architecture of the modified HMC 124
7.3 HMC memory request packet format 125
7.4 Timing of performing convolution with 3×3 kernel under different input channels 129

8.1 Organization of the thesis and the deep learning computation features supported by each proposed design 133
8.2 Using the proposed posit MAC generator to generate Verilog code 137

List of Abbreviations

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circut
CNN	Convolutional Neural Network
CONV	Convolutional Layer
CPA	Carry-Propagate Adder
CPU	Central Processing Unit
CSA	Carry-Save Adder
CT	Computer Tomography
DOT	Dot-Product
DP	Double Precision
DRAM	Dynamic Random Access Memory
FC	Fully-Connected Layer
FFT	Fast Fourier Transform
FIX	Fixed-Point Number Format
FLP	Floating-Point Number Format
FMA	Fused Multiply-Add
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HMC	Hybrid Memory Cube
HP	Half Precision
LSB	Least Significant Bit
LZA	Leading Zero Anticipator
LZAC	Leading Zero Anticipation and Counting
LZC	Leading Zero Counting
MAC	Multiply-Accumulate

MIX	Mixed Precision
MP	Multiple Precision
MSB	Most Significant Bit
PIM	Processing in Memory
QP	Quadruple Precision
SoP	Sum of Product
SP	Single Precision
TSV	Through Silicon Via
VCD	Value Change Dump
VHDL	Very High Speed Hardware Description Language

Part I

Preface

Chapter 1
Introduction

This chapter presents the performance advantage and the popularity of deep learning in many fields of applications. Due to the superior performance and the popularity, optimizing hardware deep learning operations becomes necessary. This motivates the research works to be presented in this thesis that are to design arithmetic units based on the requirements of the deep learning computation. Section 1.1 presents the performance advantage and the potential of deep learning applications. The motivation of the research works are presented in Section 1.2. Section 1.3 presents the overview of the research works. The contributions of these research works are summarized in Section 1.4.

1.1 Deep Learning Advantages and Potentials

The idea to make an intelligent system dates back to 1950s when the term artificial intelligence (AI) was coined by John McCarthy. After that, many significant algorithms were proposed by AI researchers. However, due to the limitation of the computation power in the early days, training a large scale AI model may take months, and thus AI algorithms still stayed in small scale. In early 2000s, the development of computation power made the research on large scale AI algorithms and deep learning algorithms becoming possible. Around the same time, many large database became available which boosted the development of deep learning. A major breakthrough happens in 2012 when the AlexNet [1] achieved significant improvements on image classifications over preceding methods.

After that many research works have been done on applying deep learning methods in many applications. In 2015, the ResNet [2] achieves the image classification accuracy that is even higher than human. In the field of medical imaging analysis, deep learning methods

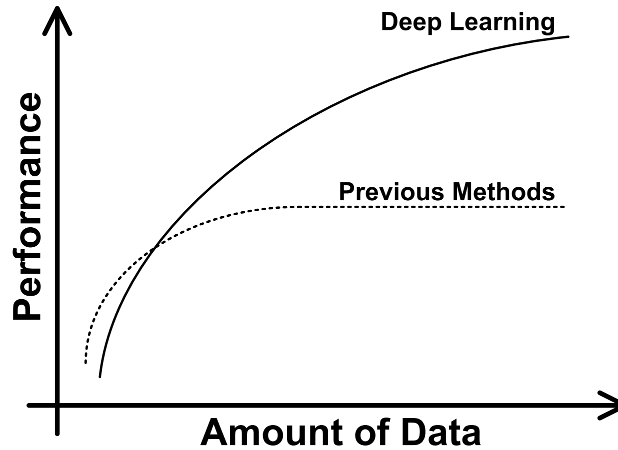


Figure 1.1: Performance comparison of deep learning methods with previous methods (Reproduced from: B. Catanzaro, "Computer Arithmetic in Deep Learning [Keynote Talk]," in ARITH26, 2016.)

are also extensively investigated. In 2017, researchers from Stanford University proposed and trained a convolutional neural network to perform skin cancer classification [3] and the network achieved a performance that was at the same level as dermatologists. In more complex tasks, such as the Go game, the deep learning method, the AlphaGo [4], can already beat the human champion. In addition to the above mentioned achievements, deep learning techniques are also widely applied in applications such as natural language processing, big-data analysis and autonomous driving.

Deep learning methods are quickly applied in many fields of applications because of two main advantages: (1) promising performance in large database, as shown in Figure 1.1. The performance of many conventional AI methods become saturated with the amount of available data goes higher. This may be limited by the computation and analysis capability of small scale models. However, in deep learning, as the model can have much larger scale, the potential of analyzing big database is also expected to be significant. (2) Reducing human workload. On one hand, deep learning is running in an end-to-end way so that the features required by performing classification or other tasks are hierarchically extracted from the input data in deep learning methods. Whereas, in conventional AI methods, features extraction and selection needs to be done manually which take a lot of time. On the other hand, in some applications, with the same level of accuracy, deep learning model can perform a task much faster than human. One example is the computer tomography (CT) image analysis.

Table 1.1: Memory and computation requirements of various deep neural networks

	LeNet-5	AlexNet	Overfeat	VGG16	GoogLeNet	ResNet-50
Top-5 Error	N/A	16.4	14.2	7.4	6.7	5.3
# of weights	60K	61M	146M	138M	7M	25.5M
# of MACs[†]	341K	724M	2.8G	15.5G	1.43G	3.9G

[†] MAC refers to the multiply-accumulate operation, and it includes one multiplication followed by one addition.

A human radiologist usually needs 5 to 10 minutes to read and analyzes a CT scan while a deep learning model can do the same task in only several seconds. Therefore, deep learning can significantly reduce the human radiologists' workload.

Deep learning can provide promising results in many fields of applications. However, the cost of implementing deep learning operations is expensive. As shown in Table 1.1, a deep neural network may have millions of parameters and a single operation may require billions of computations. As the popularity of deep learning methods raises, many recent hardware designs target to optimizing deep learning operations, both in reducing the energy consumption and in improving the speed performance. In recent years, many research works have been done on optimizing the datapath for deep learning operations [5]. As the core of computation, the arithmetic unit can determine the efficiency and functionality of the whole hardware design. In this thesis, special focuses are put on the design of arithmetic units for efficient deep learning computation. Several novel arithmetic unit architectures are proposed based on the characteristics of deep learning computation.

1.2 Motivation of Research Works

Deep neural networks are usually trained on graphics processing units (GPUs). Many deep learning frameworks, such as Tensorflow [6] and Caffe [7], perform deep neural network computations using 32-bit single-precision (SP) floating-point numbers [8] by default. However, the data-path of single-precision floating-point units is complex and the hardware cost of implementing single-precision units are expensive. These lead to a high energy consumption and a large latency when implementing deep neural networks in customized hardware. In order

to reduce the hardware cost, some research works in recent years are focused on reducing the numerical precision required by deep neural network computation. In [9], the authors successfully implemented deep neural networks with 16-bit half-precision numbers [8]. In some other works, half-precision numbers are also used in deep learning training and inference operations.

In the floating-point unit (FPU) of general-purpose processors, single-precision and higher precision formats are well supported in order to provide a better accuracy for scientific computations. However, for deep learning computation, on one hand, using low precision formats (half-precision) is able to maintain high accuracy. On the other hand, deep learning is computation intensive and thus low precision computation is desired to improve the overall speed performance. Considering these factors, the support of half-precision operations in FPU should be added to speed up deep learning computation in general-purpose processors.

In deep learning computation, the reduced precision computation is usually applied together with the mixed-precision computation in order to manage the accuracy [10]. Reduced precision method is usually applied to the multiplication because the multiplication operation is slow and the multiplier consumes large amount of resources. With reduced precision multiplication, the area of the multiplier can be reduced and the speed of the multiplication operations can be improved. When performing accumulation, a high precision adder is applied. In deep learning computation, a high precision data is kept. Before multiplication, this data is truncated to low precision in order to perform low precision multiplication. When performing accumulation, the product will be accumulated to the higher precision copy so that the accuracy can be recovered. In conventional multiple-precision FPU, both low precision multiplication data-path and high precision accumulation data-path already exist. Therefore, the support of mixed-precision computation in a multiple-precision FPU can be explored. When designing deep learning specific arithmetic unit, the mixed-precision computation feature should also be considered.

For deep neural network training, a number format with large dynamic range, such as floating-point format, is required to handle very small gradient values during the last few iterations of training. For deep neural network inference, as there is no gradient computation, the numerical precision can be further reduced. In many cases, fixed-point format can provide

enough accuracy in inference [11] [12]. When performing fixed-point computations, compared to floating-point computations, both energy consumption and speed performance can be improved. Therefore, for a deep learning processor that needs to handle both training and inference operations, both floating-point operations and fixed-point operations are required to be supported.

Reduced precision computation is feasible for deep learning computation. However, the minimum required precisions for different deep neural networks or different layers of a deep neural network are not identical [11] [13] [14]. Therefore, using a computation unit that supports only one numerical precision to perform all computations is not efficient and in this case, a multiple-precision computation unit is desired. For example, if only one 16-bit fixed-point unit is used, when one layer can be computed with 4-bit number, the other 12-bit of the computation unit is not utilized. However, if a multiple-precision 16-bit unit, that can be reconfigured to dual 8-bit units or quad 4-bit units at runtime, is applied, for 4-bit computation, four parallel computations can be performed and thus the throughput can be improved. In addition, the improvement in throughput can also be helpful to improve energy efficiency. Therefore, for a computation unit that is designed specific for general purpose deep learning computation, multiple-precision unit is preferred.

The floating-point format defined in [8] contains a sign, an exponent, and a mantissa. The bit-width of the exponent corresponds to the dynamic range of the numerical format while the bit-width of the mantissa corresponds to the representation precision of the format. In deep learning computation, the importance of exponent and mantissa to the neural network accuracy is different [13] [15]. According to the results in [15], the dynamic range is more important than representation precision for neural network accuracy. By using this finding, the deep neural network computation unit can be designed with a constant total bit-width but the bit-width of exponent and mantissa can be dynamically exchanged. So that the requirement of exponent can be met first and the remaining bits can be allocated to the mantissa. By designing with this method, more flexibility can be provided for deep learning computation.

The research on deep learning computation focuses not only on the conventional computation system such as fixed-point and floating-point computation system. In recent years,

deep learning computation using logarithmic number system and other new number formats are also investigated. The recent proposed posit number system [16] is one of them. Posit encodes numbers in a non-uniform way which fits well with the deep neural network data distribution. In addition, with the same total bit-width, the posit format can provide much larger dynamic range than floating-point format. Therefore, the use of posit in deep learning computation is promising. As posit is relatively new, there is only hardware adder and hardware multiplier available in the literature. In order to facilitate the use of posit in deep learning applications, a posit based multiply-accumulate (MAC) unit or other fused unit is required.

In recent years, the 3D memory architectures, such as hybrid memory cube (HMC) [17] and high-bandwidth memory (HBM) [18], are proposed. In 3D memory, the storage layers are stacked vertically and the communication among storage layers is achieved by the through silicon via (TSV). At the bottom of the 3D memory architecture, there is a logic layer where the memory controller and some simple logic functions are implemented. Due to the integration of the logic layer, data processing inside memory, termed processing-in-memory (PIM), becomes possible. With PIM, the bandwidth requirements of the memory interface can be significantly reduced. Memory data transfer is also reduced because some data processing can be done inside memory. Both of these reduces the energy consumed by memory data transfer. As deep learning is memory and computation intensive, using 3D memory in deep learning processing is expected to significantly improve the energy efficiency of deep learning computation.

1.3 Overview of Research Works

In this thesis, based on the computation characteristics discussed in Section 1.2, several arithmetic unit architectures optimized for deep learning computation are proposed. We provide the solutions both to optimize general-purpose FPU for deep learning computation and to design deep learning specific computation units. The arithmetic units proposed in this thesis are based on multiple-precision arithmetic unit architectures and are target for general support for as many neural network models and operations as possible. This is

different from the design of an architecture that is optimized for a specific model or operation. Therefore, the proposed arithmetic units are more useful in application specific integrated circuit (ASIC) based processor designs that are used in servers and datacenters. The whole thesis is composed of five parts with eight chapters shown as follows:

- **Part I Preface** includes:
  - Chapter 1 *Introduction*: presents the importance of deep learning and the motivations, the overview, and the contributions of the research works.
  - Chapter 2 *Background*: introduces the background information required to present the proposed research works.
- **Part II Arithmetic Unit for General Purpose Computing** includes:
  - Chapter 3 *Multiple-Precision Floating-Point Fused Multiply Add*: presents the design of a multiple-precision floating-point fused multiply add (FMA) architecture. In this chapter, the solution to optimize general purposed FPU for deep learning computation is presented. The architecture presented is designed for general purposed FPU. However, the newly added parallel half-precision supports and mixed-precision dot-product supports make the proposed design especially suitable for deep learning computation. Compared to the state-of-the-art multiple-precision floating-point FMA architectures, the newly added features make the proposed FMA supporting more functions with only minor resource overhead.
- **Part III Arithmetic Unit for Deep Learning Computing** includes:
  - Chapter 4 *Multiple-Precision Multiply Accumulate Unit*: presents an efficient fixed-point and floating-point merged MAC unit architecture. This unit is proposed for deep learning processors that need to handle both deep learning training and inference operations. The support for floating-point computations can be used in deep learning training and the support for fixed-point computations can be used in deep learning inference. In addition, the mixed-precision computation feature is introduced to this

unit to make it more suitable in deep learning computation. Compared to a floating-point MAC unit, the proposed design has negligible resource overhead but enables the deep learning processors to support both training and inference operations.

  – Chapter 5 *Flexible Multiple-Precision Multiply Accumulate Unit*: presents a flexible multiple-precision MAC unit architecture. The proposed unit also supports both floating-point operations and fixed-point operations. In floating-point mode, both 16-bit or dual 8-bit operations are supported. In fixed-point mode, 16-bit or dual 8-bit or quad 4-bit operations are supported. At the lowest precision, the proposed unit also supports logic operations for binary neural networks. In order to make the proposed unit more flexible, in floating-point mode, the bit-width of exponent and mantissa can be mutually exchanged. In fixed-point mode, the bit-width of integer and fraction can also be exchanged. Compared to a 16-bit floating-point MAC unit, the proposed unit provides more flexibility for deep learning computation with only minor resource overhead.

  – Chapter 6 *Posit Multiply Accumulate Unit*: presents an MAC architecture designed based on the posit number format. Posit number format encodes the numbers in a non-uniform way which fits well with the deep learning data distribution. In addition, with the same bit-width, posit can provide much larger dynamic range than the floating-point format. As the range is more important than precision for deep learning accuracy, posit is expected to be promising in deep learning applications. The proposed posit MAC unit in this chapter is intended to facilitate the use of posit format in deep learning applications.

- **Part IV Arithmetic Unit for 3D Memory** includes:

  – Chapter 7 *Improved Hybrid Memory Cube Architecture*: presents an improved HMC architecture for deep neural network processing. The proposed architecture is designed based on the original HMC architecture with minimal modification in order to avoid thermal problem. Two new instructions are added for weight-sharing deep learning computation. The memory vault controller is also modified to support parallel vault operations. Moreover, a simple MAC unit is added to the logic layer for

each memory vault to achieve deep learning computation. A software simulation of the proposed architecture is performed and the results show a reduced memory bandwidth requirement and thus an improved speed performance.

- **Part V Conclusion** includes:
  - Chapter 8 *Summary and Future Work*: includes the summary of all presented research works and the plan for future works.

1.4 Summary of Contributions

In this thesis, several new arithmetic unit architectures are designed and implemented for efficient deep learning computation. Some of them are the first published designs which introduced new computation features or combined multiple features into a single architecture. Those proposed arithmetic units are ready to be combined with memory module and control module into a flexible and efficient deep neural network processors. For each of the proposed arithmetic units, the hardware design merits, including timing, area, power, and energy, are analyzed in detail. The designs are also compared with related designs available in the literature to show their advantages and improvements. For the flexible MAC design, a simplified neural network case study is included to show its performance under different application scenarios. Finally, a hybrid memory cube architecture is proposed to overcome the limitations of the conventional memory interface to achieve better speed performance and energy efficiency for deep learning computation. Moreover, our proposed posit based MAC unit could be a starting point of research works that utilize the posit number system in deep learning applications.

Below is the list of publications, arranged according to the order of appearance in this thesis:

- Chapter 3 *Multiple-Precision Floating-Point Fused Multiply Add*:
  - **H. Zhang**, D. Chen and S. Ko, "Efficient Multiple-Precision Floating-Point Fused Multiply-Add with Mixed-Precision Support," in *IEEE Transactions on Computers*, vol. 68, no. 7, pp. 1035-1048, Jul 2019.

- Chapter 4 *Multiple-Precision Multiply Accumulate Unit*:
  - **H. Zhang**, H. J. Lee and S. Ko, "Efficient Fixed/Floating-Point Merged Mixed-Precision Multiply-Accumulate Unit for Deep Learning Processors," *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1-5.
- Chapter 5 *Flexible Multiple-Precision Multiply Accumulate Unit*:
  - **H. Zhang**, D. Chen and S. Ko, "New Flexible Multiple-Precision Multiply Accumulate Unit for Deep Neural Network Training and Inference," *IEEE Transactions on Computers*, accepted August 2019.
- Chapter 6 *Posit Multiply Accumulate Unit*:
  - **H. Zhang**, J. He and S. Ko, "Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications," *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2019, pp. 1-5.
- Chapter 7 *Improved Hybrid Memory Cube Architecture*:
  - **H. Zhang**, J. He and S. Ko, "Improved Hybrid Memory Cube for Weight-Sharing Deep Convolutional Neural Networks," *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, March 2019, pp. 1-5.
- Other publications that are not included in this thesis:
  - **H. Zhang**, D. Chen and S. Ko, "Area- and Power-Efficient Iterative Single/Double-Precision Merged Floating-Point Multiplier on FPGA," in *IET Computers and Digital Techniques*, vol. 11, no. 4, pp. 149-158, Jul 2017.
  - **H. Zhang**, D. Chen and S. Ko, "High Performance and Energy Efficient Single-Precision and Double-Precision Merged Floating-Point Adder on FPGA," in *IET Computers and Digital Techniques*, vol. 12, no. 1, pp. 20-29, Jan 2018.
  - Y. Wang, K. Shahbazi, **H. Zhang**, K. Oh, J. Lee and S. Ko, "Efficient Spiking Neural Network Training and Inference with Reduced Precision Memory and Computing," accepted by *IET Computers and Digital Techniques*, Jun 2019.

- K. Chae, G. Jin, S. Ko, Y. Wang, **H. Zhang**, E. Choi and H. Choi, “Deep Learning for Classification of A Small ($\leq$ 2cm) Pulmonary Nodule on CT Imaging: A Preliminary Study,” accepted by *Elsevier Academic Radiology*, May 2018.
- Z. Jiang, **H. Zhang**, Y. Wang and S. Ko, “Retinal Blood Vessel Segmentation Using Fully Convolutional Network with Transfer Learning,” in *Elsevier Computerized Medical Imaging and Graphics*, vol. 68, pp. 1-15, Sep 2018.
- L. Han, **H. Zhang** and S. Ko, “Decimal Floating-Point Fused Multiply-Add with Redundant Internal Encodings,” in *IET Computers and Digital Techniques*, vol. 10, no. 4, pp. 147-156, Jul 2016.
- L.Han, **H.Zhang** and S. Ko, “Area and Power Efficient Decimal Carry-Free Adder,” in *IET Electronics Letters*, vol. 51, no. 23, pp. 1852-1854, Nov 2015.
- Y. Wang, **H. Zhang**, K. Chae, G. Jin and S. Ko, “Novel Convolutional Neural Network Architecture for Improved Pulmonary Nodule Classification on Computed Tomography,” under review *Multidimensional Systems and Signal Processing*, submitted Mar 2019.
- Y. Wang, E. Choi, **H. Zhang**, G. Jin and S. Ko, “Breast Cancer Classification in Automated Breast Ultrasound using Multi-View CNN with Transfer Learning,” under review *Ultrasound in Medicine and Biology*, submitted Jul 2019.

Chapter 2

Background

This chapter presents the background information of the proposed works in this thesis. Various numerical formats that are used in the proposed works are presented in Section 2.1. Section 2.2 presents the basic architectures of fused arithmetic units including fused multiply add unit and multiply accumulate unit. Section 2.3 presents the concepts and advantages of mixed-precision computing. Section 2.4 presents the characteristics of deep learning computing in detail which motivate the research works in this thesis. Section 2.5 presents the architecture of a hybrid memory cube.

2.1 Numerical Formats

2.1.1 Floating-Point Format

The binary floating-point number formats defined in IEEE 754-2008 contain three components: 1-bit sign (S), w-bit biased exponent (E), and p-bit mantissa (M), as shown in Figure 2.1. The biased exponent E is obtained by $e + bias$, where e is the actual exponent value and $bias = 2^{w-1} - 1$. There is always an implicit bit in front of the mantissa.

The basic precision formats defined in IEEE 754-2008 are 16-bit half-precision (HP), 32-bit single-precision (SP), 64-bit double-precision (DP), and 128-bit quadruple-precision (QP). For these four formats, the bit-width of each component, the bias value, and the maximum and minimum exponent for normal number is summarized in Table 2.1.

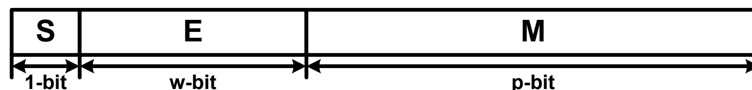


Figure 2.1: Binary floating-point format defined in IEEE 754-2008

Table 2.1: Floating-Point format defined in IEEE 754-2008

Format	Sign	Exponent	Mantissa	Bias	e_{max}	e_{min}
16-bit HP	1	5	10	15	15	-14
32-bit SP	1	8	23	127	127	-126
64-bit DP	1	11	52	1023	1023	-1022
128-bit QP	1	15	112	16383	16383	-16382

The IEEE 754-2008 floating-point format also reserves some special values for exceptional cases handling. The floating-point number T represented in IEEE 754-2008 format and its actual value v can be summarized as follows:

- If $E = 2^w - 1$ and $M \neq 0$, then T is not a number (NaN).
- If $E = 2^w - 1$ and $M = 0$, then T is infinity (Inf) and $v = (-1)^S \times (+\infty)$.
- If $1 \leq E \leq 2^w - 2$, then T is normal number and $v = (-1)^S \times (1 + M) \times 2^{E-bias}$. Normal number has an implicit bit of 1.
- If $E = 0$ and $M \neq 0$, then T is subnormal number and $v = (-1)^S \times M \times 2^{1-bias}$. Subnormal number has an implicit bit of 0.
- If $E = 0$ and $M = 0$, the T is zero and $v = (-1)^S \times (+0)$.

Rounding in IEEE 754-2008

When performing floating-point operations, such as addition and multiplication, the infinitely precise intermediate results will usually have larger bit-width than the format defined bit-width. In order to make these results be accommodated into the IEEE 754-2008 defined formats, rounding operations are required. In IEEE 754-2008, five rounding modes are defined:

- *roundTiesToEven*: the infinitely precise result should be rounded to the nearest representable floating-point number. If two numbers are equally near to the result, the one with an even least significant bit (LSB) should be used. This is the default rounding mode of IEEE 754-2008 standard.

- *roundTiesToAway*: the infinitely precise result should be rounded to the nearest representable floating-point number. If two numbers are equally near to the result, the one with a larger magnitude should be used.
- *roundTowardPositive*: the infinitely precise result should be rounded to the floating-point number that is closet to and no less than the result.
- *roundTowardNegative*: the infinitely precise result should be rounded to the floating-point number that is closet to and no greater than the result.
- *roundTowardZero*: the infinitely precise result should be rounded to the floating-point number that is closet to and no greater in magnitude than the result.

Exception Handling in IEEE 754-2008

IEEE 754-2008 standard also defines several default exception handling: invalid operation, division by zero, overflow, underflow, and inexact. When these exception cases happen, the corresponding flags are required to be raised.

- Invalid operation: this exception is signaled when there is no definable result.
  - any computational operation on a signaling NaN.
  - multiplication in a form of $0 \times \infty$ or $\infty \times 0$.
  - fused multiply-add in a form of $0 \times \infty + c$ or $\infty \times 0 + c$ unless c is a quiet NaN.
  - addition, subtraction, or fused multiply-add: magnitude subtraction of infinities.
  - division: $0 \div 0$ or $\infty \div \infty$.
  - remainder: remainder(x, y) when y is zero or x is infinite and neither is NaN.
  - square root if the operand is less than zero.
  - quantize when the result does not fit in the destination format or when one operand is finite and the other is infinite.
  - conversion of a floating-point number to an integer format, when the source is NaN, infinity, or a value that would convert to an integer outside the range of the result format under the applicable rounding attribute.

  - comparison by way of unordered-signaling predicates when operands are unordered.
  - $logB(NaN)$, $logB(\infty)$, or $logB(0)$ when the output of $logB$ is an integer format.

- Division by zero: this exception is signaled when an exact infinite result is defined for an operation on finite operands.
  - for division, when the divisor is zero and the dividend is a finite non-zero number, the sign of the infinity is the exclusive OR of the operands' signs.
  - $logB(0)$ when result of $logB$ is a floating-point format, the sign of the infinity is minus ($-\infty$).

- Overflow: this exception is signaled when the format's largest finite number is exceeded in magnitude. The default result is determined by the rounding method and the sign of the intermediate result:
  - roundTiesToEven and roundTiesToAway carry all overflow to ∞ with the sign of the intermediate result.
  - roundTowardZero carries all overflows to the format's largest finite number with the sign of the intermediate result.
  - roundTowardNegative carries positive overflows to the format's largest finite number and carries negative overflows to $-\infty$.
  - roundTowardPositive carries negative overflows to the format's most negative number and carries positive overflows to $+\infty$.

- Underflow: this exception is signaled when a tiny non-zero result is detected. A rounded result should be provided which might be zero, subnormal, or $\pm b^{emin}$.
  - after rounding: a non-zero result computed as though the exponent range were unbounded would lie strictly between $\pm b^{emin}$.
  - before rounding: a non-zero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm b^{emin}$.

- Inexact: this exception is signaled when the rounded result is different from what would have been computed if both exponent range and precision were unbounded.

Table 2.2: Results when both addition/subtraction operands are zeros

x op y	Rounding Mode	
	RTN, RTZ, RTPos†	RTNeg†
$(+0)+(+0)$	$+0$	$+0$
$(+0)+(-0)$	$+0$	-0
$(-0)+(+0)$	$+0$	-0
$(-0)+(-0)$	-0	-0
$(+0)-(+0)$	$+0$	-0
$(+0)-(-0)$	$+0$	$+0$
$(-0)-(+0)$	-0	-0
$(-0)-(-0)$	$+0$	-0

† RTN: roundTiesToEven; RTZ: roundTowardZero; RTPos: roundTowardPositive; RTNeg: roundTowardNegative.

The Sign of the Operational Result

In IEEE 754-2008 standard, the sign is not interpreted if either an input or result is NaN. If neither the inputs nor results are NaN, the sign of a product or quotient is the exclusive OR of the operands' signs.

The sign of an addition or subtraction is determined by the relationship between the two operands. When the sum of two operands with opposite signs (or the difference of two operands with the same sign) is exactly zero, the sign of the sum is positive except roundTowardNegative where the sign of an exact zero sum is negative. However, $x + x = x - (-x)$ retains the same sign as x even when x is zero. When both operands are zero, depending on the rounding methods, the results are summarized in Table 2.2.

For FMA operations, if the infinite precise result is exactly zero, the sign of the result should be determined by the rules for a sum of operands. When the exact result is non-zero but the result is zero because of rounding, the resulting zero takes the sign of the exact result.

In this thesis, the research work presented in Chapter 5 uses customized floating-point formats. Although they may have different bit-width from the standard ones, the floating-point conventions defined in IEEE 754-2008, such as the biased exponent, implicit mantissa bit, and subnormal cases handling, still apply.

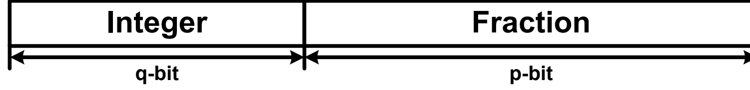


Figure 2.2: Binary fixed-point format

2.1.2 Fixed-Point Format

Fixed-point format is much simpler than the floating-point format. It does not have the exponent field and all bits belong to the significand as shown in Figure 2.2. Therefore, with the same total bit-width, the fixed-point format has smaller representation range than that of the floating-point format.

There are two categories of fixed-point format: signed and unsigned. For unsigned number, its value can be determined by

$$v = \sum_{i=0}^{q-1} I_i \times 2^i + \sum_{j=0}^{p-1} F_j \times 2^{-(p-j)} \tag{2.1}$$

where I_i represents the i-th bit of the integer and F_j represents the j-th bit of the fraction.

For signed number, the most significant bit (MSB) represents its sign. For zero and positive number, the sign is equal to 0 and for negative number, the sign is equal to 1. For signed number, its value can be determined by

$$v = (-1)^{I_{q-1}} \times 2^{q-1} + \sum_{i=0}^{q-2} I_i \times 2^i + \sum_{j=0}^{p-1} F_j \times 2^{-(p-j)} \tag{2.2}$$

The signed fixed-point formats used in this thesis use the two's complement encoding which is widely used in modern computer systems. In two's complement encoding, a non-negative number has the same notation as the number written in unsigned notation. For a negative number, the encoding can be obtained by inverting all bits of the notation of its absolute value and then adding one to the LSB.

For fixed-point numbers, extension is often required when converting from one numerical format to another format with larger bit-width. For a pure fractional number, the extension can be done by simply adding multiple zero bits to the right of the LSB. When extending the integer part, there is some differences between the unsigned number and signed number. For unsigned numbers, the extension can be done with prefixing zero bits. For signed numbers

represented in two's complement format, the extension is done with replicating the sign bit. On the other hand, when converting a number with large bit-width to a format with smaller bit-width, the rounding process that is used in floating-point operations can be used for fractional part. For integer part, usually a truncation is performed to remove the unused most significant several bits.

In deep neural networks, weights contain both positive and negative values and should be represented with signed numbers if fixed-point format is used. For activations, as most current deep neural networks use rectified linear unit (ReLU) as their activation function, the activations are always non-negative numbers and thus can be represented with unsigned fixed-point format. For hardware design, the unsigned format can always be prefixed with one more bit zero to be converted to signed format. In this thesis, the fixed-point units are designed with signed format.

2.1.3 The Posit Format

The floating-point format and fixed-point format described in the above sections use uniformed encoding for all numbers. In this way, after the format is determined (for example 32-bit floating-point numbers with 8-bit exponent and 23-bit mantissa), all the numbers within the representation range are represented using the same format. This is fine for general-purpose computing. However, for some applications with non-uniformed data distribution, such as deep learning (this will be discussed in Section 2.4), uniformed encoding become inefficient. For these applications, on one hand, data with small values are densely appeared so that a high precision (large bit-width of mantissa) is required to distinguish these small values. On the other hand, data with large values and tiny values can also appear, and thus a large dynamic range (large bit-width of exponent) is required to correctly represent these numbers. Consequently, a large floating-point format is required to meet these two requirements which leads to an expensive cost in arithmetic and memory operations. However, in these applications, the large exponent bit-width and large mantissa bit-width are not always required at the same time. For densely distributed small values, a large bit-width mantissa is needed but the exponent bit-width can be small. For sparsely distributed large or tiny values, a large bit-width exponent is needed but the mantissa bit-width can be small.

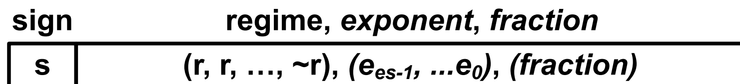


Figure 2.3: Format of posit number

Therefore, the encoding for the numerical values in these applications can be improved.

In posit number system [16], a non-uniformed encoding method is applied where the total bit-width is a constant but the bit-width of components can be changed. This characteristic fits well with the data distribution of deep learning applications. For densely distributed small values, more bits can be allocated to the mantissa. For sparsely distributed large or tiny values, the exponent can occupy more bits. Therefore, the total bit-width of posit numbers when compared to the conventional floating-point numbers can be reduced in deep learning applications.

The general format of a posit number is shown in Figure 2.3. A posit format, Posit(nb, es), is defined with the total bit-width nb and the maximum exponent bit-width es. Except the sign bit has a fixed 1-bit bit-width, the bit-width of all other components, the regime, the exponent, and the mantissa, are flexible. Exponent and fraction appear only when there is remaining bits for them. A sign bit of zero represents a positive number and a sign bit of one represents a negative number. If a number is negative, two's complement will be taken before decoding components.

The regime part is first considered. It is a series of 0s or 1s followed by a bit with an opposite value. The number of 1s or 0s is corresponding to the regime value rg. If the regime bits start with m-bit zeros followed by a one bit, then $rg = -m$. On the other hand, if the regime bits start with m-bit ones followed by a zero bit, then $rg = m - 1$. Then if the sign and regime do not occupy the whole nb-bit, the exponent is followed. Its maximum bit-width is defined with es. Exponent is an unsigned value without bias. Then the remaining bits are allocated to fraction $frac$. There is always an implicit one bit for the $frac$. The value of a number represented in posit format is:

$$v = (-1)^s \times useed^{rg} \times 2^{exp} \times (1 + frac) \tag{2.3}$$

where $useed = 2^{2^{es}}$. Zero is represented as all zero bits and infinity is represented as 1-bit one followed by all zeros. There is no subnormal in posit format.

Table 2.3: Comparison of the dynamic range of Posit format and floating-point format

Format	Min. Value†	Max. Value†	Dynamic Range*
Std FP8	2×10^{-3}	3×10^{2}	1.5×10^{5}
Posit(8,4)	1×10^{-29}	8×10^{28}	8.0×10^{57}
Std FP16	6×10^{-8}	7×10^{4}	1.2×10^{12}
Posit(16,5)	1×10^{-135}	7×10^{134}	7.0×10^{269}
Std FP32	1×10^{-45}	4×10^{38}	4.0×10^{83}
Posit(32,8)	8×10^{-2309}	8×10^{2311}	1.0×10^{4620}

† absolute value.
* dynamic range = max. value / min. value.

The comparison of the dynamic range of some posit formats with standard floating-point formats is shown in Table 2.3. Due to the use of a dynamic regime component, the representation of posit is much larger than the conventional floating-point format with the same bit-width. However, due to the bit-width of each component is dynamic during runtime, extra hardware is required to correctly extract each component from input operands and to correctly pack the resulting components into the output format. The details of the posit hardware designs will be discussed in Chapter 6.

2.2 Fused Arithmetic Units

Multiplier and adder are two common operators used in many applications. Conventionally the multiplier and adder are usually designed as separate hardware units. However, consecutive multiplication and addition operations (for example, $y = \sum x \times w + b$) are commonly used in many applications, such as signal processing and machine learning. In these applications, a fused arithmetic unit is desired to improve the computing performance.

There are three categories of fused arithmetic units that are popularly used: the FMA, MAC, and dot-product (DOT, or sum of product, SoP). FMA is usually referred to floating-point operations while MAC is usually referred to fixed-point operations. Both of them are designed to perform the operations in the form of $A \times B \pm C$. For FMA, three operands, A, B, and C, are in the same numerical precision, while in MAC, the precision of C is usually higher than A and B. DOT is common for both fixed-point and floating-point. The i-way

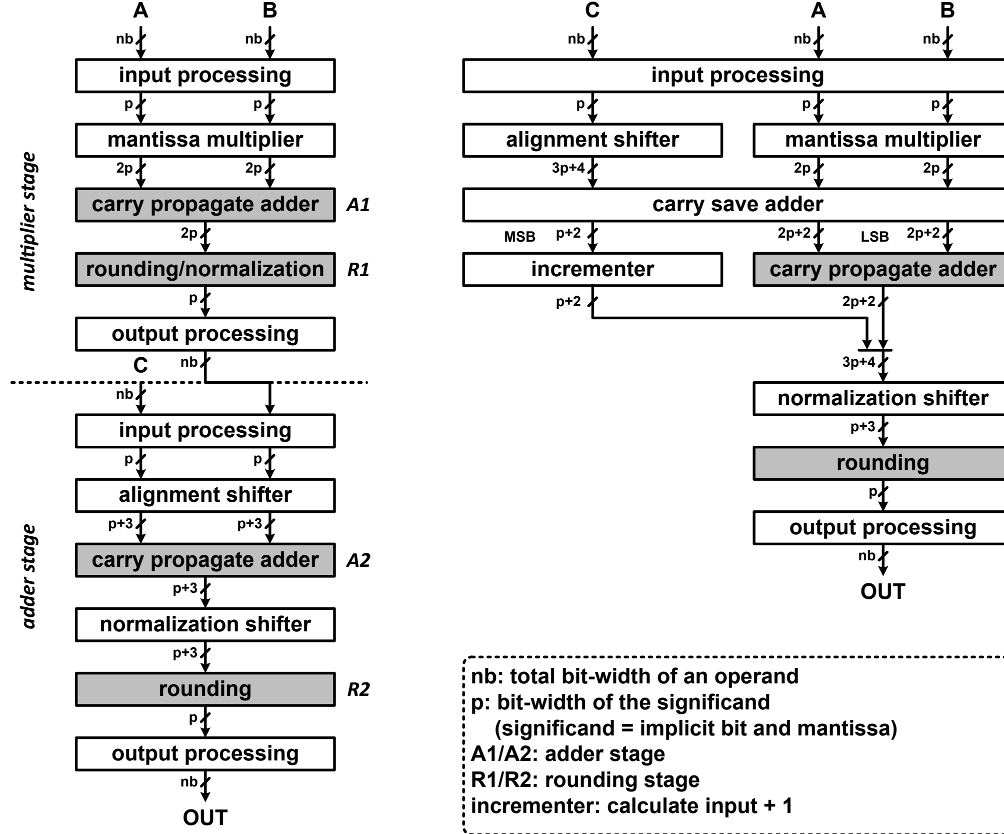


Figure 2.4: Implementing floating-point $A \times B + C$ with separate arithmetic units and fused arithmetic unit

DOT is designed to perform the operation of $\sum_i (A_i \times B_i) + C$.

The first floating-point FMA architecture was proposed and used in IBM RS/6000 processor [19]. It combined the floating-point multiplication and addition into a single operation. The right part of Figure 2.4 shows a simplified FMA data-path when computing $A \times B + C$. The p-bit mantissa of A and B are multiplied. The multiplier contains a partial product generation unit and partial product accumulate unit and a carry-save format product is generated. In parallel to the multiplication, the alignment of C is performed. After the product is generated, the aligned C should also be available. The aligned C and the product then go through one stage of (3,2) carry save adder (CSA) to be accumulated into two vectors. Then the generated two vectors are added using a carry propagate adder (CPA). Only the LSB $2p + 2$-bit vectors should go into the CPA. The MSB $p + 2$-bit only contains one vector and can be added with an incrementer. Then the addition result is normalized and rounded and the output is generated.

Figure 2.4 also compares the whole data-path when implementing $A \times B + C$ using separate multiplier and adder and FMA unit. In the separate units implementation, the multiplier and the adder use separate adders ($A1$ and $A2$) while in the FMA unit, the multiplication and addition can share one CPA. This will lead to a reduction in hardware area. Moreover, FMA can generate a result with better accuracy than separate multiplier and adder. On one hand, in the separate units implementation, rounding operations are performed twice while FMA only performs one rounding. On the other hand, the addition data-path of the FMA is larger than that of the separate adder. Therefore, more bits can be reserved to participate the future computation stages. In addition, the FMA unit can also be used to perform normal multiplication operation or addition operation: by setting $B = 1$, the FMA can compute $A + C$ and by setting the absolute value of C to 0, the FMA can compute $A \times B$. In order to ensure the multiplication is compliant with IEEE 754-2008 standard [8], the sign of C must be determined according to the sign of $A \times B$ and the rounding mode. In roundTiesToEven, roundTowardZero, or roundTowardPositive modes, if the sign of $A \times B$ is positive, then C can be set to $+0$ or -0. However, when the sign of $A \times B$ is negative, C also needs to be set to -0. Moreover, in roundTowardNegative mode, the sign of C must be the same as that of $A \times B$.

Due to the above mentioned advantages, FMA was added into the IEEE 754-2008 standard and many processors use FMA as their basic FPU architecture. The MAC unit has similar data-path to the FMA except that the bit-width of the carry propagate adder may be larger since the precision of C is larger than A and B in MAC. DOT unit can be designed by implementing multiple multipliers followed by a large carry propagate adder. When floating-point formats are used, the products are needed to be aligned with each other before the final addition.

For those arithmetic units proposed in this thesis, if only floating-point formats are supported, such as the design discussed in Chapter 3, the design is called FMA unit. When both floating-point and fixed-point are supported, such as the ones in Chapter 4 and Chapter 5, the design is termed as MAC unit. The posit based design discussed in Chapter 6 is termed as MAC unit since the format of A, B and C could be different.

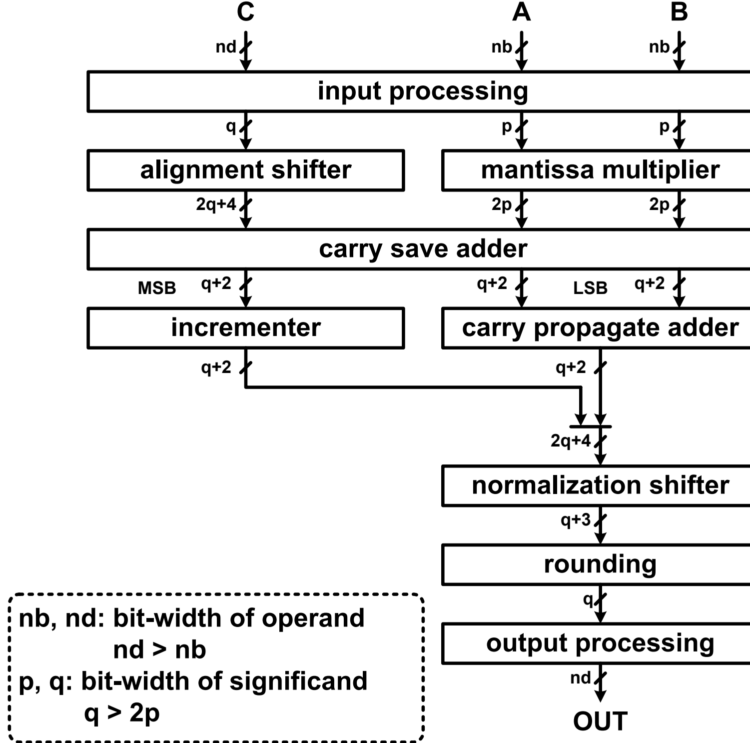


Figure 2.5: Datapath of a mixed-precision FMA

2.3 Mixed-Precision Arithmetic Units

Mixed-precision arithmetic units use different precisions at different computation phases. The conventional MAC architecture is a mixed-precision unit since C has different precision from A and B. The FMA unit can be modified by extending the precision of C and the adder data-path to realize mixed-precision operations. Mixed-precision arithmetic units are usually used in application-specific computing system designs. Due to the mixed-precision feature, it usually designed in a fused arithmetic unit. The use of mixed-precision arithmetic unit has mainly two advantages: (1) the multiplication can be done with low precision to improve the computing speed. (2) the addition is done with high precision to maintain a highly accurate result.

Mixed-precision operations are suggested to be used in deep learning together with reduced precision computing in [10]. Many reduced precision deep neural network training methods require the mixed-precision computing to maintain the accuracy [9] [20]. In the field of linear algebra, the mixed-precision method is also widely used to perform the itera-

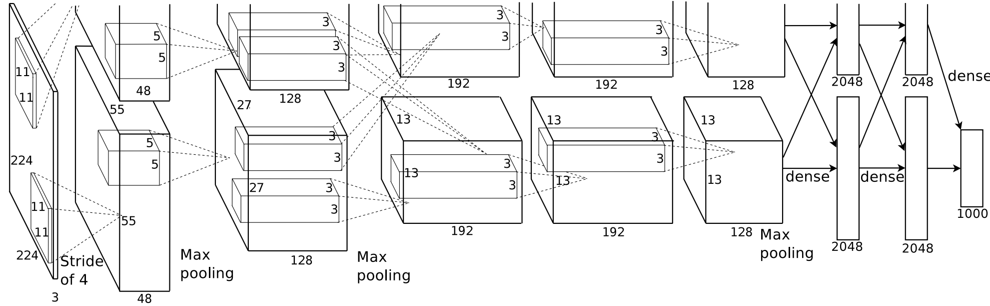


Figure 2.6: Architecture of AlexNet (From: Figure 2 of reference [1].)

tive refinement operations. In [21], a two precisions mixed-precision method is proposed and in [22], a three precisions mixed-precision method is proposed. Both methods can maintain the same level of accuracy as the one when computing with only high precision numbers while achieving significant speedup.

A simplified hardware mixed-precision FMA architecture is shown in Figure 2.5. It is also used to compute $A \times B + C$ but now C has higher numerical precision than A and B. The data-path shown in Figure 2.5 assumes the mantissa bit-width of C is larger than twice the mantissa bit-width of A and B. (For other cases, the data-path bit-width should be modified accordingly.) The multiplication is done in p precision while the alignment, addition, normalization, and rounding are done with q precision.

2.4 Deep Learning Computing

2.4.1 Overview

Deep neural networks are usually composed of convolutional layers (CONV), fully-connected layers (FC), activation layers, and pooling layers. The architecture of AlexNet, which is a popular convolutional neural network (CNN) architecture, is shown in Figure 2.6. It consists of 5 CONVs followed by 3 FCs. Activation layers are used after each FC and CONV and pooling layers are applied after the second, third and fifth CONVs.

The majority of the computations happens in convolutional layers [5]. For AlexNet, to generate one output, a total of 724 million multiply-accumulate operations are required where the five CONVs contain 665 million MAC operations.

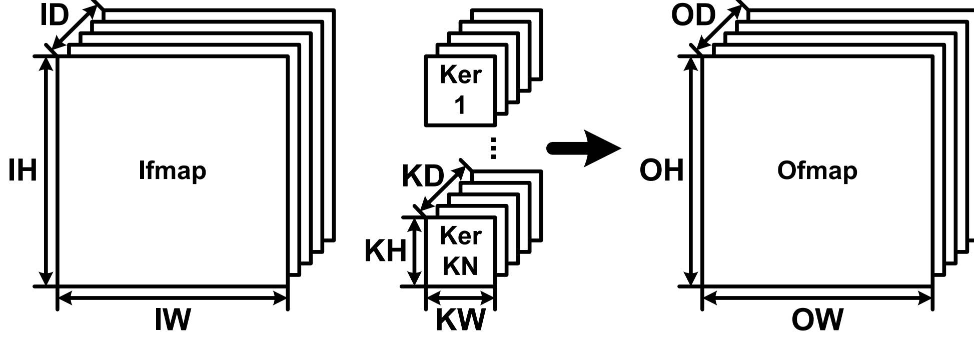


Figure 2.7: Process of convolution computation in deep neural network

The process of calculating a convolution in a deep neural network is shown in Figure 2.7. There are ID input feature maps (Ifmap) of size $IH \times IW$ and KN kernels (Ker) of size $KD \times KH \times KW$. They are used to generate OD output feature maps (Ofmap) of size $OH \times OW$. In deep neural network, $KD = ID$ and $OD = KN$. Assume the amount of zero padding is P for input feature maps and the stride of kernel, which is the amount of pixels of each kernel sliding, is represented with S, then size of the Ofmap can be generated from the size of Ifmap with equation (2.4):

$$\begin{aligned} OH &= (IH + 2P - KH)/S + 1 \\ OW &= (IW + 2P - KW)/S + 1 \end{aligned} \tag{2.4}$$

Then, the value of each pixel in Ofmaps can be calculated with equation (2.5):

$$\begin{gathered} O[m][x][y] = act(B[m] + \sum_{i=0}^{KW-1} \sum_{j=0}^{KH-1} \sum_{k=0}^{ID-1} I[k][Sx+i][Sy+j] \times W[m][k][i][j]) \\ 0 \leq m < OD,\ 0 \leq x < OW,\ 0 \leq y < OH \end{gathered} \tag{2.5}$$

where O, B, I, W represent output feature map, bias, input feature map (after padding), and kernel, respectively, and $act()$ represents the activation function. When the feature map size is large or the number of feature maps is large, the amount of required computations will be large. In addition, the majority of computations is in the form of multiply-accumulate. For FC layer computation, similar equations can be applied with $KW = IW$ and $KH = IH$.

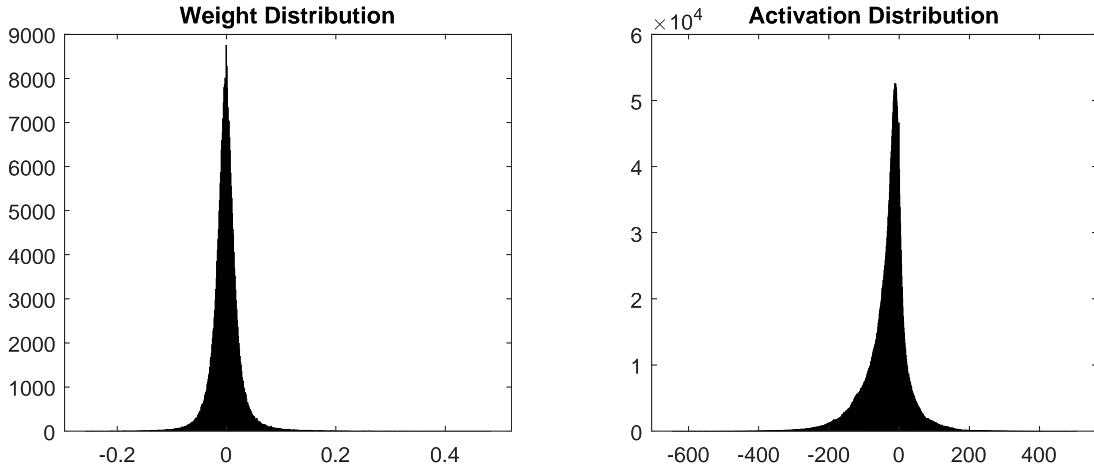


Figure 2.8: Weights and activations distribution of AlexNet Convolution Layer 2

2.4.2 Reduced Precision Computations

Deep neural networks are usually trained using GPUs where the 32-bit single-precision floating-point format is the default computing format. Therefore, many pre-trained neural network models come with single-precision floating-point parameters. However, the costs of implementing single-precision operators are expensive. For deep neural networks, they are computationally intensive. Therefore, the overall cost, including timing and energy, of implementing deep neural network with single-precision format is high. In order to reduce the implementation cost, many research works focus on reduced precision computations for deep neural networks.

The feasibility of using reduced precision computation is derived from the distributions of deep neural network data. An example is shown in Figure 2.8 where the histogram of weights and activations of CONV 2 of AlexNet [1] are plotted. As shown in Figure 2.8, both weights and activations are distributed in narrow ranges and thus fewer number of bits (than 32-bit) can be used to correctly represent and compute for weight and activation. Although only one example is shown here, this form of data distribution is not uncommon in deep neural network models [5] [13].

Many research works focused on reduced precision deep learning computing are available in the literature. In [9], the authors proposed to train deep neural network with 16-bit half-precision number format. The 12-bit floating-point format is utilized in training in [23].

In [11] and [13], 8-bit floating-point format is used.

In addition to floating-point numbers, fixed-point numbers are also used in deep neural network computing. The works in [11] and [12] show that inference can be accomplished with 8-bit fixed-point numbers. The 16-bit fixed-point format is used by some deep neural network accelerators [24] [25]. Moreover, for more efficiency, binarized neural network [26] is proposed where neural network parameters are constrained to ± 1.

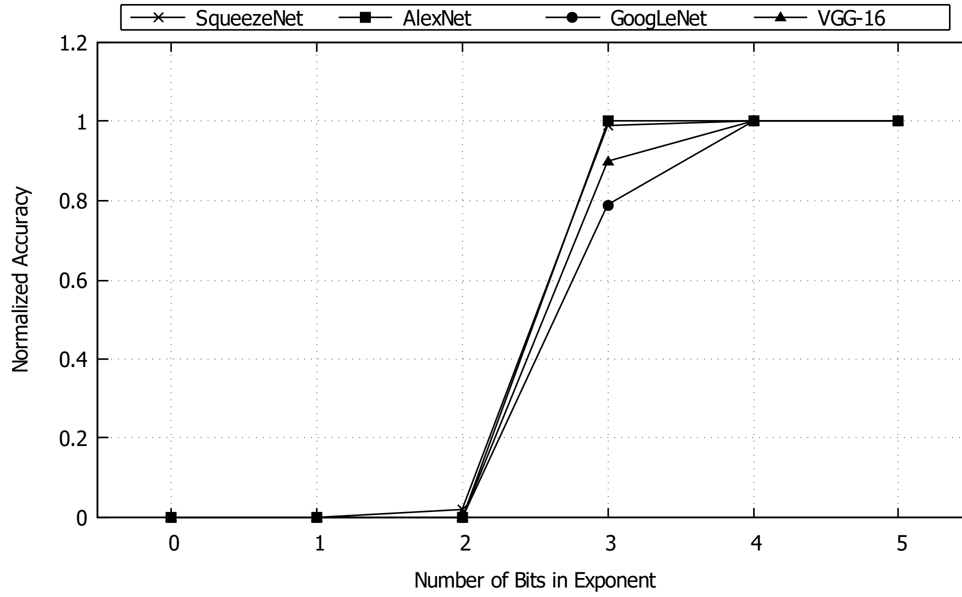
Generally, when using reduced precision computing, floating-point format is preferred in training. During training, in addition to the normal forward computations, weight and bias parameter are required to be tuned in each training iteration. At the last few training iterations, the parameter changes, which is called gradient, are usually very small. Therefore, a number format with large dynamic range is required to handle such small values. However, in inference, there is no gradient issue and therefore, smaller precision or even fixed-point format can be used to achieve more timing and resource benefit.

2.4.3 Range vs Precision

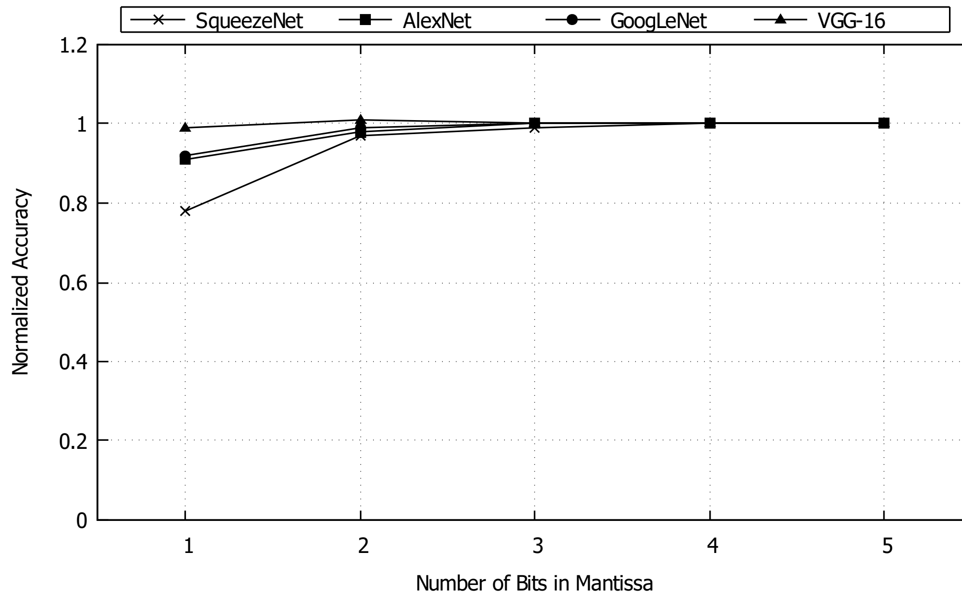
When using reduced precision floating-point, the contribution of the exponent and the mantissa on neural network accuracy should be figured out so that the number format could be more efficient. This question has been investigated in [15]. In this section, the conclusion from [15] will be discussed.

The results of range and precision in [15] are reproduced and shown in Figure 2.9. The accuracy shown is the relative accuracy that is the ratio between the accuracy of current configuration and the accuracy when using single-precision format. For exponent, as shown in Figure 2.9(a), when the bit-width is small (< 3), the neural network cannot be used. With the increase of exponent bit-width, the neural network gradually reaches the full accuracy. The results of mantissa are quite different as shown in Figure 2.9(b). Even when the mantissa only has 1-bit, these neural networks can still maintain a relatively high accuracy. When the mantissa has 2-bit or more bits, the full accuracy is achieved and more mantissa bits do not lead to the increase in accuracy.

Comparing the accuracy under different exponent and mantissa bit-width, we can find that the exponent bit-width (representation range) is more important than the mantissa



(a) Exponent bit-width



(b) Mantissa bit-width

Figure 2.9: Normalized neural network accuracy under different exponent and mantissa bit-width (Reproduced from Figure 8 and Figure 11 of [15].)

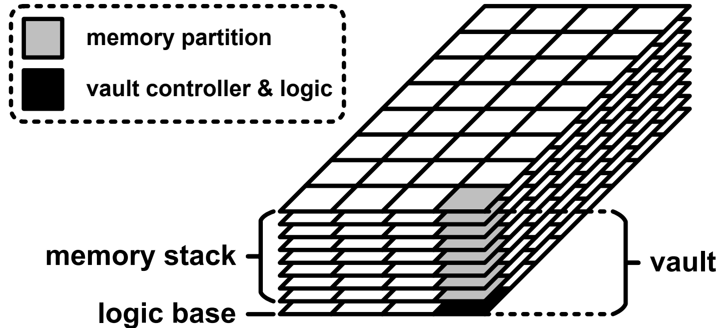


Figure 2.10: Basic hybrid memory cube architecture

bit-width (representation precision) for neural network accuracy. As long as the exponent bit-width is large enough, no matter what the bit-width of mantissa is, the neural network accuracy can always stay at high level. Therefore, when designing a numerical format for neural network computing, the requirement of exponent bit-width should be met first and the remaining bits can be allocated to the mantissa.

2.5 Hybrid Memory Cube

Dynamic random-access memory (DRAM) has been used as the main memory of modern computer systems and GPUs. The bandwidth and speed of DRAM get improved through each architecture evolution. However, in recent years, the improvement of computing power is much larger than the improvement of memory and the DRAM gradually becomes the performance bottleneck of computer systems. In addition, there is an increased demand in large memory capacity. However, as the DRAM unit is a 2D planar, the high capacity DRAM will occupy too many board area. Moreover, the energy consumed by data transfer through the memory interface is high, especially for data intensive applications. To solve these problems, the 3D memory [17] [18] architecture is proposed. In 3D memory architecture, DRAM dies are stacked vertically and thus reduces their board area consumption. In addition, a logic layer is added in the 3D memory. Many logic and arithmetic functions can be implemented in the logic layer. As a result, the memory bandwidth requirement and thus the energy consumed by data transfer can be reduced.

Hybrid memory cube (HMC) [17] is one of the available 3D memory architectures. The

Table 2.4: HMC Specifications

	Configurations
Number of links in package	2, 4
Link lane speed	12.5, 15, 25, 28, 30
Memory density	8GB
Number of vaults	32
Memory bank	512 banks
Maximum DRAM data bandwidth	320GB/s
Maximum vault data bandwidth	10GB/s

basic organization of a HMC architecture is shown in Figure 2.10. It consists of up to 8 DRAM dies. The bottom layer is a logic layer. As shown in Figure 2.10, each memory die is divided into 32 memory partitions. Each partition contains many memory banks (usually 8 or 16 depending on the memory density). The memory partitions in the same vertical location form a vault. The communication within a vault is realized by the through silicon via (TSV). The vault controller is implemented in the logic layer. In addition, some logic or arithmetic functions are also performed in the logic layer. The specifications of HMC defined in [17] are presented in Table 2.4.

Part II

Arithmetic Unit for General Purpose Computation

Chapter 3

Multiple-Precision Floating-Point Fused Multiply Add[1]

This chapter presents the design of a multiple-precision floating-point FMA unit for general-purpose computing. Compared to other multiple-precision FMA units available in the literature, the proposed FMA unit introduces the support of parallel half-precision operations and the support of mixed-precision dot-product operations. Due to these newly added computing features, the proposed unit is especially suitable for deep learning computing. Section 3.1 presents the motivations to design such an FMA unit. The proposed FMA architecture is presented in Section 3.2. Section 3.3 presents the synthesis results of the proposed design. Section 3.4 concludes this chapter.

3.1 Introduction

The concept of the floating-point FMA unit is first proposed in the IBM RS/6000 processor [19]. The FMA operation itself is a basic operation for many scientific and engineering applications. In addition, compared to the separate multiplier and adder, the FMA unit has the advantage of smaller area and higher accuracy. Moreover, the FMA unit can also be used to perform basic multiplication or addition operations. Due to these advantages, nowadays many processors use the FMA unit as the basic design unit. FMA operations were also added

[1]The content of this chapter is originally published in IEEE Transactions on Computers [27]. The manuscript has been reformatted for inclusion in this thesis.

Hao Zhang (HZ), Dongdong Chen (DC) and Seok-Bum Ko (SK) designed the study. HZ developed and optimized the architecture, developed the HDL code of the architecture, and performed logic synthesis and results analysis. DC gave suggestions on improving the architecture and analyzing the results. HZ prepared the manuscript with contributions from DC and SK to the manuscript structure, readability and analysis and discussion of the results.

in the IEEE 754-2008 standard [8].

In recent years, a lot of research effort has been put into the design of FMA architecture to reduce latency, area, and power consumption [28–32]. In addition to these works, in order to efficiently support multiple precisions operations for different applications in a single architecture and to support the applications that require multiple-precision operations at the same time [33], some dual-mode FMAs [34–36] are proposed in the literature. These FMA designs support both double-precision operations and single-precision operations. Moreover, multiple-precision FMAs that also support quadruple-precision operations, [37] and [38], are proposed in order to support the quadruple-precision operations for scientific applications [39].

In addition to these three precisions, the 16-bit half-precision is also included in the IEEE 754-2008 standard [8]. It was originally designed as a storage format to save memory resources. However, due to its smaller bitwidth, the implementation of arithmetic units based on half-precision is much more efficient than other floating-point formats [40]. Recently, half-precision is used more and more often to accelerate floating-point applications. Use of half-precision is suggested in deep learning applications to take the place of the conventional 32-bit floating-point format [10] [5]. Use of half-precision is also recommended in linear algebra [22] to speed up the iterative refinement process. Given these increasing applications, half-precision arithmetic units are suggested to be revisited in [41] where an FMA architecture for exact half-precision FMA operations is proposed. In many recent commercial central processing units (CPUs), such as Intel Knights Mill [42], and GPUs, such as NVIDIA Tesla P100 [43], half-precision operations are supported in hardware.

In addition to normal FMA operations, mixed-precision FMA operations are getting more popular recently. In order to increase the performance while maintaining high accuracy, many applications suggest performing multiplication in lower precision while accumulating products in higher precision. In the field of linear algebra, mixed-precision method is widely applied. In [44], single-precision is used for the majority of the computations while double-precision is used at the last few stages to refine the results. In [22], a similar mixed-precision method is extended by applying three different precisions in various computation stages. In a recent work [21], half-precision and double-precision mixed-precision operations are proposed

to utilize the half-precision Tensor Cores in GPUs to speed up iterative refinement solvers. The mixed-precision method is also widely used in deep learning applications. In deep learning applications, when computing an output feature map, a large amount of accumulation operations are required where the rounding errors are also accumulated. In order to reduce the accumulated rounding error, accumulation using higher precision is suggested in [10].

In [45], a mixed-precision FMA where the single-precision product is added to the double-precision addend was first proposed. Compared to a normal FMA, the mixed-precision FMA consumes more hardware due to the wider datapath for high precision accumulation. However, for a multiple-precision FMA architecture, the datapath for higher precision addend and addition are already available. Therefore, the possibility of implementing mixed-precision FMA with multiple-precision FMA can be explored. Furthermore, in a multiple-precision FMA architecture, multiple lower precision operations can be done in parallel. Therefore, the mixed-precision dot-product operations can be explored to increase the throughput of the accumulation. The area overhead will only be the hardware used for the alignment of multiple products. In [46], the dot-product operation is supported. However, in that design, the addend and the products are in the same precision, and thus it cannot benefit from the advantages of mixed-precision operations.

In this chapter, an efficient multiple-precision floating-point FMA architecture is proposed. Half-precision support is added in addition to the support of single-precision, double-precision, and quadruple-precision. Specifically, the proposed multiple-precision FMA architecture supports one quadruple-precision operation, or two parallel double-precision operations, or four single-precision operations, or eight parallel half-precision operations. The architecture of the proposed FMA is fully-pipelined so that each set of operations can be started every clock cycle. In addition to normal FMA operations, mixed-precision operations are supported. The mixed-precision operation in the proposed design is a 2-term dot-product accumulating to a higher precision addend. For double-precision, the proposed FMA supports one such mixed-precision dot-product operation. For single-precision and half-precision, two parallel and four parallel mixed-precision dot-products are supported, respectively. Quadruple-precision can already provide enough precision for most applications [47] and accumulating quadruple-precision to even higher precision is hardware expensive; there-

fore, mixed-precision operation is not applicable to quadruple-precision. The mixed-precision FMA operation can be realized by setting the operands of one multiplication of the 2-term dot-product to zeros.

The contributions of the proposed design in this chapter are summarized as follows:

- Proposing an efficient multiple-precision FMA architecture that supports half-precision, single-precision, double-precision, and quadruple-precision operations.
- Modifying the proposed FMA to also support mixed-precision dot-product and mixed-precision FMA operations.
- Supporting more functionalities compared to a conventional quadruple-precision FMA with only minor area overhead.
- Performing accumulation and dot-product operations with higher throughput compared to a standard mixed-precision FMA architecture.

3.2 The Proposed Design

The proposed FMA unit supports normal FMA operations $A \times B + C$. It can perform eight HP operations, four SP operations, two DP operations, or one QP operation with a latency of 3 pipeline stages. In addition to normal FMA operation, the proposed unit also supports mixed-precision FMA and mixed-precision dot-product operations. As discussed in Section 3.1, mixed-precision FMA operation [45] is recommended in many applications [10] [22]. It uses lower precision for multiplications to improve the performance while accumulating the results to higher precision to maintain high accuracy.

Different from the mixed-precision functionality in [45], where $A \times B + C$ (A and B in single-precision and C in double-precision) is supported, in the proposed design, $A_1 \times B_1 + A_2 \times B_2 + C$ (A_1, B_1, A_2, and B_2 are in lower precision and C is in higher precision) can be supported. This is due to the datapath for multiple-precision operations, since in the proposed FMA design, for example, one QP addend corresponds to two DP multiplications. Similarly, two DP addends each corresponds to two SP multiplier operands. Therefore, the proposed FMA will support one DP mixed-precision dot-product ($A_1 \times B_1 + A_2 \times B_2 + C$),

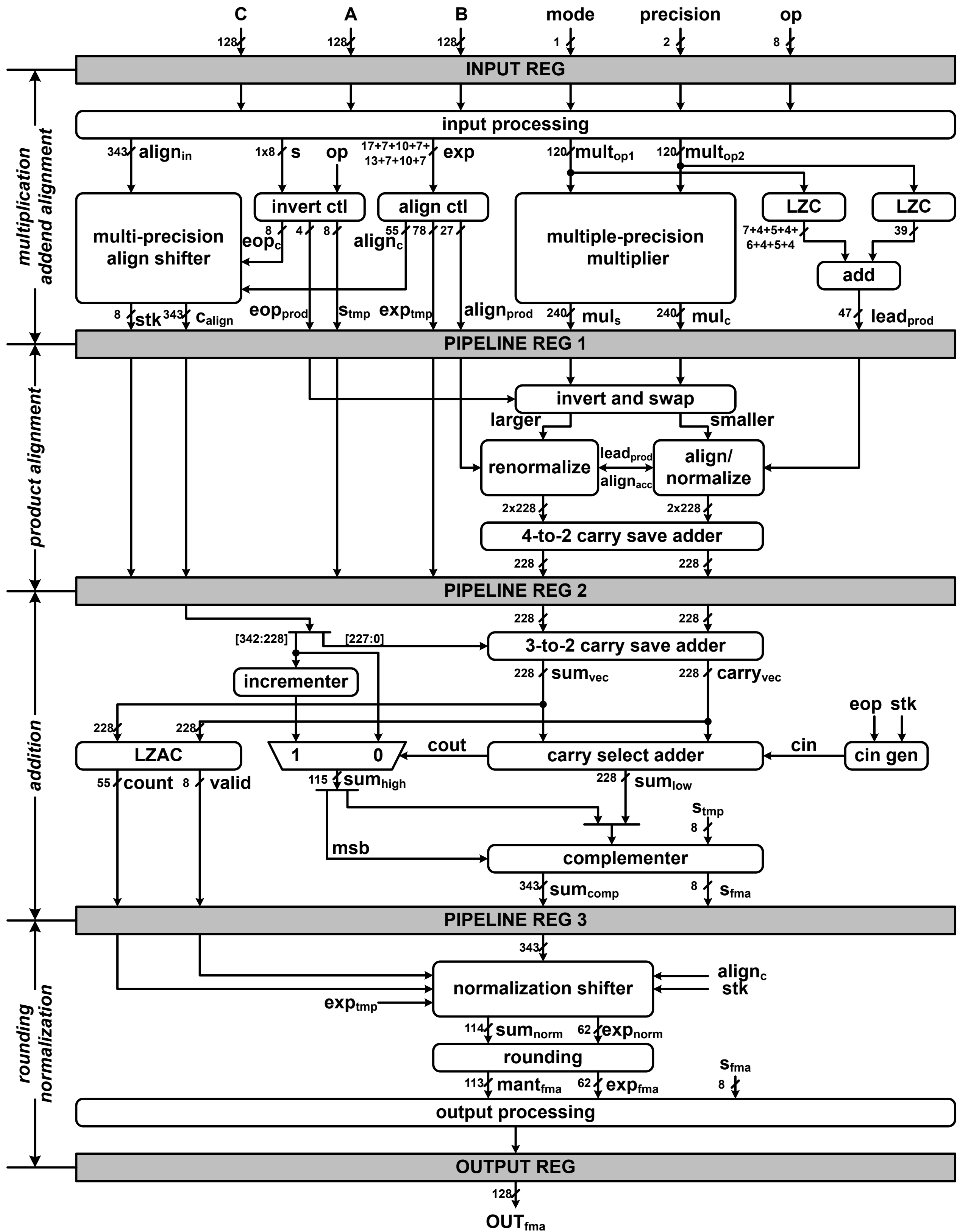


Figure 3.1: Datapath of the proposed multiple-precision fused multiply-add unit

or two parallel SP mixed-precision dot-products, or four parallel HP mixed-precision dot-products. For QP operation, since it can already provide enough precision for most of the applications [47], we do not provide higher precision accumulation support for QP operation. The dot-product operation was supported in [46], however, in that design, the addend was with the same precision as that of multiplication operands. In the proposed design, the addend is with higher precision that can help reduce the accumulated rounding error.

In addition to the supported 2-term dot-product operations, for SP (HP) numbers, 4-term dot-product with accumulating to DP (SP) addend, or for HP numbers, 8-term dot-product with accumulating to SP addend can also be realized. However, these modes are not supported in the proposed architecture. On one hand, multiple 2-term dot-product, compared to one 4-term or one 8-term dot-product, is more flexible in real applications where multiple independent dot-product operations can be implemented in parallel. For example, the FFT butterfly architecture [48] requires 2-term dot-product. If 4-term or 8-term dot-product is used to implement this FFT architecture, the hardware will not be fully utilized. In addition, the 8-term or 4-term dot-product can always be realized by using 2-term dot-product hardware. However, there may exist minor rounding errors since when using 2-term dot-product unit, a rounding will be performed after accumulating every two terms instead of only one final rounding as the 4-term or 8-term dot-product units do. On the other hand, the hardware cost to support 4-term or 8-term dot-product is much higher than 2-term dot-product because a more complex alignment shifter and control logic are required to align the products. Therefore, in order to make the proposed architecture more hardware efficient and more flexible for applications, the proposed FMA is chosen to support 2-term dot-product operation.

The datapath of the proposed multiple-precision FMA is shown in Figure 3.1. The whole datapath is split into four pipeline stages. The grey blocks represent the position of pipeline registers. The first pipeline contains the mantissa multiplier for A and B. It also contains the alignment shifter of C and the corresponding shifting control logic. The leading zero counting (LZC) logic in the first pipeline stage is used in mixed-precision operations to renormalize the subnormal products. The second pipeline stage contains the logic for newly added mixed-precision support. It includes the alignment shifter to align the two products and the carry

save adder to add two products together. The third pipeline stage contains the carry save adder to combine the aligned C and product of $A \times B$, the carry select adder to add the combined vectors, and the leading zero anticipation and counting (LZAC) logic to generate the leading zero numbers in the result. Finally, the normalization and rounding of the result is performed in the last pipeline stage.

When performing normal FMA operations, the logic for mixed-precision operations in the first pipeline stage can be gated and the second pipeline stage can be bypassed. Therefore, normal FMA operation has a latency of 3 clock cycles. When performing mixed-precision operations, four pipeline stages are required.

The 128-bit input can accommodate one QP number, two DP numbers, four SP numbers, and eight HP numbers. The proposed architecture is fully-pipelined, therefore each set of operations for different precisions can be started every clock cycle. A 1-bit signal *mixp* is used to control whether the proposed design works in normal FMA mode or mixed-precision mode. The 2-bit *mode* signal is used to distinguish different precisions operations. In mixed-precision mode, the *mode* is set to the precision of multiplier operands.

In the following subsections, the components for normal FMA, mixed-precision FMA, and mixed-precision dot-product operations are discussed.

3.2.1 Input Processing

The input processing module basically extracts each component from the IEEE 754-2008 format inputs and processes the sign, exponent, and mantissa into a unified format for different precision operations. The sign of each operand is organized into a 8-bit signal s. For HP operation, each bit represents the sign of one sub-operand. For SP operation, $s[1]$, $s[3]$, $s[5]$, and $s[7]$ are respectively used for each sub-operand while the other bits are set to zero. Similarly, for DP operation, $s[3]$ and $s[7]$ are used and, for QP operation, only $s[7]$ are used.

The exponents are also extracted and their values are evaluated to determine the implicit bit of the mantissa. When the exponent is zero, the implicit bit will be 0. Otherwise, the implicit bit will be 1. A total of eight exponent processing units are built. The first, third, fifth, and seventh units are of 5-bit each and are dedicated to HP operations. The second (sixth) unit is of 8-bit and is shared by SP1 and HP2 (SP3 and HP6). The fourth unit is

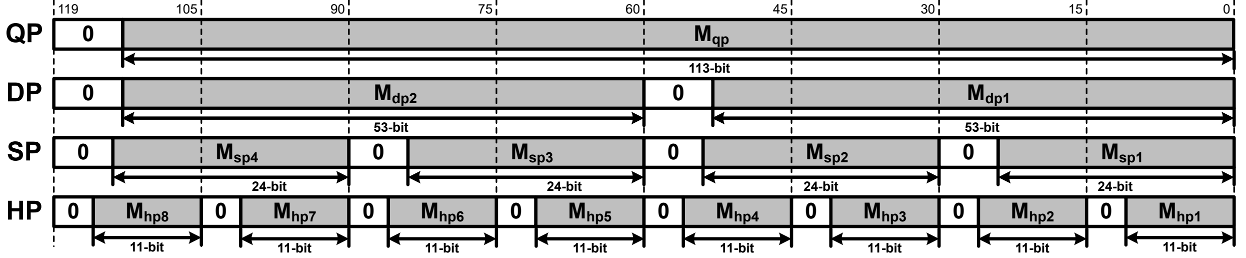


Figure 3.2: Unified mantissa format for different precisions

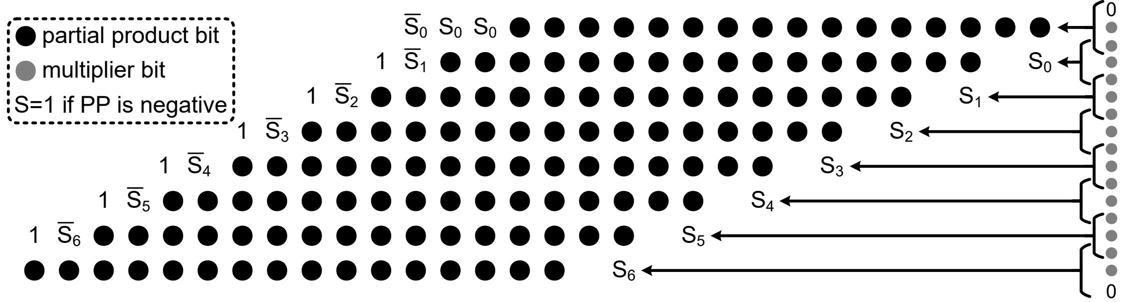


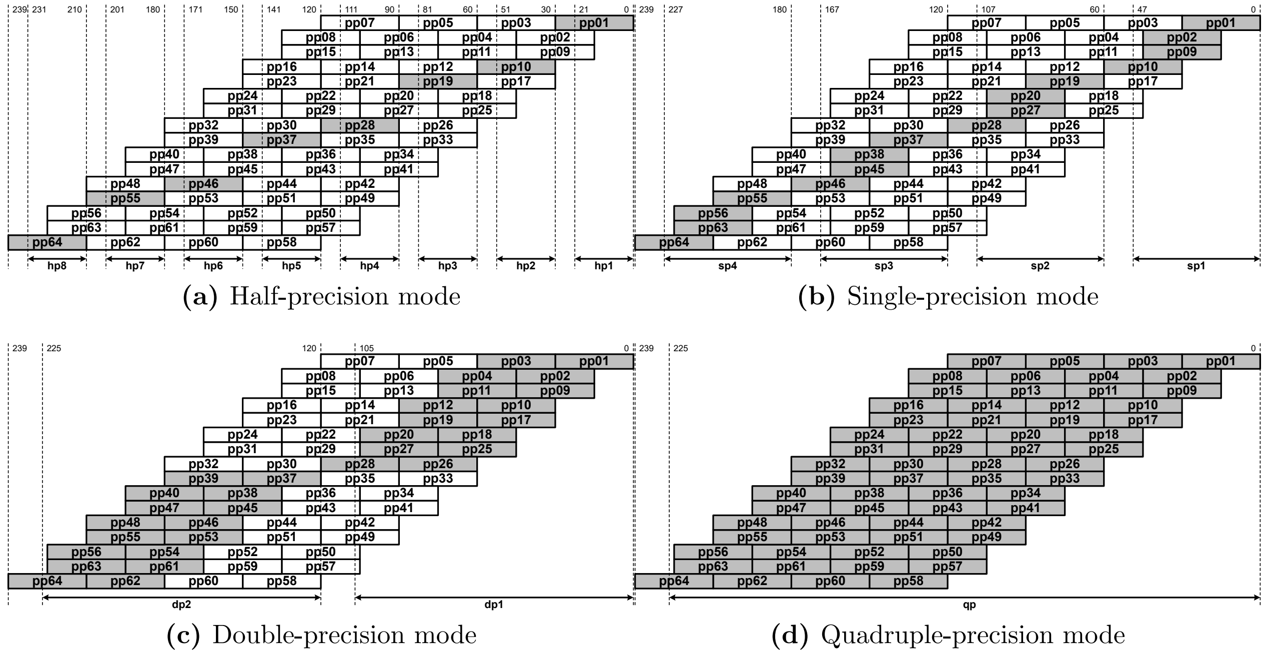
Figure 3.3: Partial products of the 15×15 radix-4 Booth multiplier

of 11-bit and is shared by DP1, SP2, and HP4. The eighth unit is of 15-bit and is shared by QP, DP2, SP4, and HP8. After determining implicit bit for mantissa, each of the eight unified exponent is extended by 2-bit for further processing. The 2-bit are used to prevent overflow during exponent addition and to handle subnormal cases.

The mantissa is also processed into a unified format for the multiple-precision multiplier. The unified format is shown in Figure 3.2. The total bidwidth of the unified mantissa is set to 120-bit. For lower precisions, each of the mantissa is prefixed with zeros to fill the 120-bit bitwidth. The value of each mantissa is evaluated to determine whether the corresponding operand is subnormal when the exponent is zero. This will affect alignment shifting amount calculation since for subnormal number, the actual exponent value is $e = E - bias + 1$ instead of $E - bias$.

3.2.2 Mantissa Multiplier

According to Figure 3.2, each unified mantissa format is divided into 8 parts. Each part contains the complete mantissa for a HP operand and contains partial mantissa for other precision operands. The higher precision multiplications are accomplished in a recursive



(a) Half-precision mode

(b) Single-precision mode

(c) Double-precision mode

(d) Quadruple-precision mode

Figure 3.4: Enabled multipliers and product regions in different precision modes

method. Therefore, to support the multiplication of QP numbers, a total of 64 small multipliers of size 15×15 are required. Each part of the mantissa in one operand a is multiplied by each part of the mantissa in operand b. Each 15×15 multiplier is implemented with radix-4 Booth multiplication algorithm [49]. Each 15×15 multiplier generates 8 partial products, as shown in Figure 3.3. These 8 partial products are accumulated with two levels of (4,2) carry-save adders to generate one carry-save format *pp*. There are a total of 64 carry-save format products *pp* generated, as shown in Figure 3.4(a). Assume x is the index of subword in operand a and y is the index of subword in operand b, then the index of *pp* is calculated by $(x-1) \times 8 + y$, for example, $pp16 = m2_a \times m8_b$.

For each precision mode, only the required 15×15 multipliers are enabled. Figure 3.4(a)-(d) shows the enabled 15×15 multipliers (gray rectangles) in each precision mode. When one 15×15 multiplier is not enabled, its outputs are all zeros so that they will not affect the results. In Figure 3.4, each rectangle represents one set of carry-save format product of one 15×15 multiplier. Therefore, the product array contains 32 rows of vectors. These 32 vectors will be accumulated by 4 levels of (4,2) carry-save adders to generate the final carry-save format product of the whole 120×120 multiplier.

In HP mode, eight 15×15 multipliers are enabled for each HP operation. The locations of these eight multipliers are shown in Figure 3.4(a). For modified Booth multiplier, there will be a carry-out of 1. For a single multiplier, this carry-out is out of the range of the possible product location and can be neglected. However, in the proposed design, this carry-out might affect the results of the adjacent multiplier. In order to ensure correct product computation, in HP mode, when performing partial product accumulation, any carry propagating through bit30, bit60, bit90, bit120, bit150, bit180, and bit210 will be discarded. As for HP operands, the effective mantissa is 11-bit. Therefore, the product cannot be larger than 22-bit. As a result, in HP mode, the products can be found in the least significant 22-bit of every 30-bit, as shown in Figure 3.4(a).

In SP mode, results of four of the 15×15 multipliers need to be combined to generate one SP result, as shown in Figure 3.4(b). As the result of each Booth multiplier has a carry-out which should not be considered as product bit, when combining results of small Booth multipliers for larger multipliers, these carry-out should be removed or be propagated outside the larger product range. In the proposed design, the sum vector of each carry-save format Booth multiplier result is extended with multiple 1s to reach the most significant bit (MSB) location of the larger product so that the carry-out bit will be propagated out of the larger product range. These extended vectors are then accumulated using (4,2)-compressors. Similar to the HP case, any carry propagation through bit60, bit120, and bit180 is discarded. As a SP product can be at most 48-bit, the products can be found in the least significant 48-bit of every 60-bit, as shown in Figure 3.4(b). For DP and QP mode, similar extension method and carry suppression method will be applied to ensure the correct product. The DP result can be found in the least significant 106-bit of every 120-bit, as shown in Figure 3.4(c). For QP, the result can be extracted from the least significant 226-bit of the result, as shown in Figure 3.4(d).

The designs in [37] and [38] use array multiplier as their mantissa multiplier. The reason is that, according to their unified mantissa format, the partial product of Booth multiplier [49] contains subword carry bits that require extra logic to handle when performing low precision operations. In order to avoid these extra delay, they choose to use array multiplier. In addition, the control logic for Booth encoding will be more complex when handling multiple

precisions.

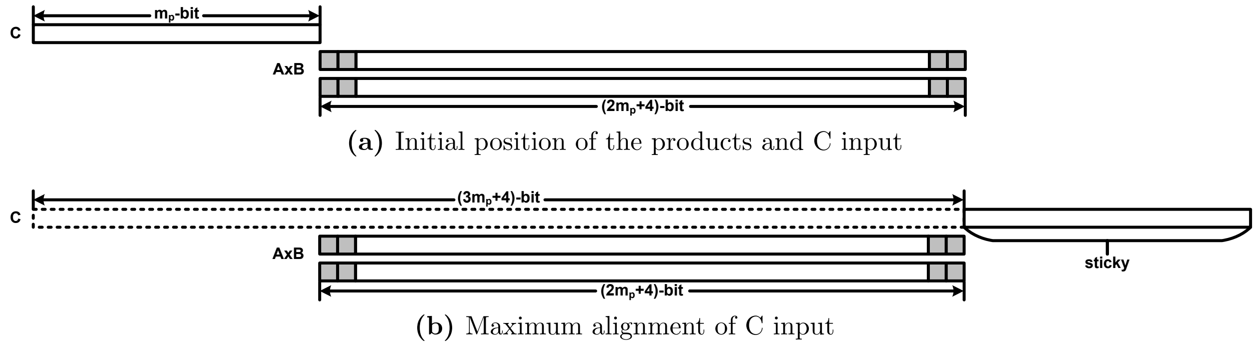
For the proposed design, in order to reduce the number of partial products and to improve the performance, the radix-4 Booth multiplier [49] is applied. The subword carry is properly handled. In addition, in the proposed design, the multiplier operands of each precision are divided into multiple 15-bit subwords as shown in Figure 3.2. The Booth encoding is applied to each of the 15×15 multiplier. As unified operand format is used, there is no need for the control logic for Booth encoding to handle multiple precisions. The only control logic used is to determine whether to enable the 15×15 multiplier according to precision mode.

Another way to handle the subword carry and signed partial product is to use full multiplier for each 15×15 multiplier. The result of each 15×15 multiplier will then be one vector instead of being in carry-save format. In this case, as the floating-point mantissa is unsigned number, the product will also be an unsigned number. Then the following stage accumulation will be simplified since there is no need to perform extension and the number of remaining partial products will be reduced from 32 to 16. In addition, there is no need to handle subword carry propagation. For example, for SP operation, the actual products of two adjacent operations are separated by 12-bit zeros. For the later 3 levels of (4,2) carry-save adders, the carry bit can propagate at most 3-bit more positions. Therefore, it cannot propagate into the position of the next SP product and thus there is no need to worry about the subword carry bit. This method will increase the critical path delay and is not applied in the proposed design. However, if more pipeline stages can be applied in the architecture and the multiplier can be divided into multiple pipeline stages, this method can be considered.

Radix-8 Booth multiplier can bring even fewer partial products compared to radix-4 Booth multiplier. However, for small size 15×15 multiplier, radix-8 Booth can generate only 2 fewer partial products than radix-4 Booth but the encoding logic is more complex and it needs additional logic to calculate $3 \times Multiplicand$. Therefore, for a more efficient multiplier implementation, radix-4 Booth multiplier is used instead of radix-8 Booth multiplier.

3.2.3 Alignment Shifter

The alignment shifter runs in parallel to the mantissa multiplier to reduce the latency of the FMA datapath. Similar to the alignment shifting method used in the basic FMA architecture,



(a) Initial position of the products and C input

(b) Maximum alignment of C input

Figure 3.5: Alignment of C operand in the proposed fused multiply-add unit

the product is anchored and the addend C is placed to the left of the product, as shown in Figure 3.5(a). Two additional bits between the addend C and the product are inserted to avoid overflow and to ensure correct rounding of C. The maximum alignment amount is reached when the most significant bit (MSB) of C is placed 2-bit to the right of the least significant bit (LSB) of the product, as shown in Figure 3.5(b), when all bits of C can be ORed to the sticky bit. The 2-bit gap is to ensure correct rounding for subnormal operands. Therefore, for the number format with m_p-bit mantissa, a $(3m_p + 4)$-bit alignment shifter is required. Specifically for QP, DP, SP, and HP operations, a 343-bit, 163-bit, 76-bit, and 37-bit alignment shifter is required, respectively. For mixed-precision mode, as the possible subnormal products will be renormalized, the maximum shift amount can be $2m_q + 4$ where m_q is the bitwidth of the higher precision addend.

To generate the alignment shifting amount, the exponent difference d between addend C and product $A \times B$ will be calculated, where $d = e_C - (e_A + e_B)$. As subnormal cases are handled, the exponent value is $e = E - bias + 1$ for subnormal numbers instead of $e = E - bias$. In addition, in mixed-precision mode, the bias for addend and product are different. Therefore, to simplify the design, the actual exponent value will be first calculated and later operations are all based on exponent instead of biased exponent. In mixed-precision mode, the exponents of two products, $A_1 \times B_1$ and $A_2 \times B_2$, are compared and the larger one will be used to compare with C exponent. With exponent difference d, the alignment shift amount can be calculated as $const - d$, where the $const$ is due to the initial position of C and $A \times B$. The value of $const$ is $m_p + 3$ or $m_q + 3$ in mixed-precision mode.

To enable resource sharing among all four precisions operations, the mantissa of addend C

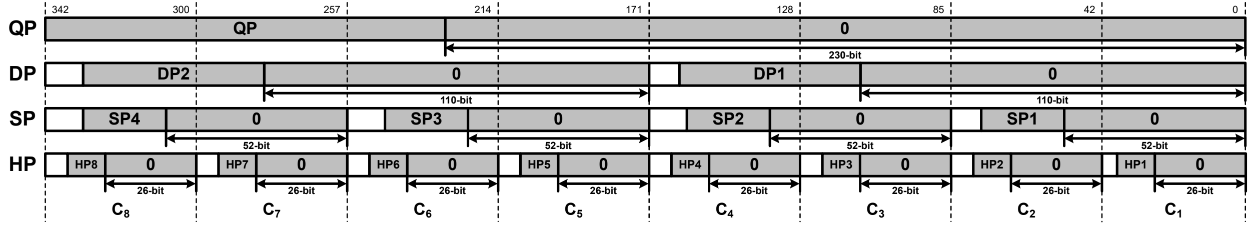


Figure 3.6: Unified input format for the alignment shifter of the proposed fused multiply-add unit

is rearranged. The unified format of the input for the alignment shifter is shown in Figure 3.6. The mantissa of each precision is extended with zeros to fill the required shifting bitwidth. For lower precision, the extended vectors are right aligned, so that the bit shifted out of the shifter can be used to generate the partial sticky bit.

The implementation of the alignment shifter is shown in Figure 3.7. It is composed of 8 smaller alignment shifters. In HP mode, all these 8 shifters run independently for shifting levels 0-5. In SP mode, every two of these shifters are combined where the bits shifted out from the former shifters need to be sent to the later shifters. Similarly, in DP mode, the first four and the last four shifters are combined as groups for the two DP operations. In QP mode, all eight shifters are combined.

The alignment shifter runs in multiple levels. The QP alignment requires a 9-bit shifting amount, the DP alignment needs a 8-bit shifting amount, the SP alignment requires a 7-bit shifting amount, and the HP alignment requires a 6-bit shifting amount. The eight shifters contain 6 shifting levels (lvl 0-5 in Figure 3.7) which consider the $[5:0]$ of the shifting amount. For the next level (lvl 6 in Figure 3.7), four one stage shifters will consider the seventh bit of the shifting amount for SP, DP, and QP operations. Then two one-stage shifters (lvl 7 in Figure 3.7) will consider the eighth bit of shifting amount for the DP and QP operations. Finally, the last one stage 343-bit shifter (lvl 8 in Figure. 3.7) will consider the 9th bit of the shifting amount for QP operation.

In mixed-precision mode, in addition to the alignment of the addend C, the two products are also required to be aligned. In the proposed architecture, the alignment of two products is put in an additional pipeline stage so that in normal FMA operation, these operations can be bypassed.

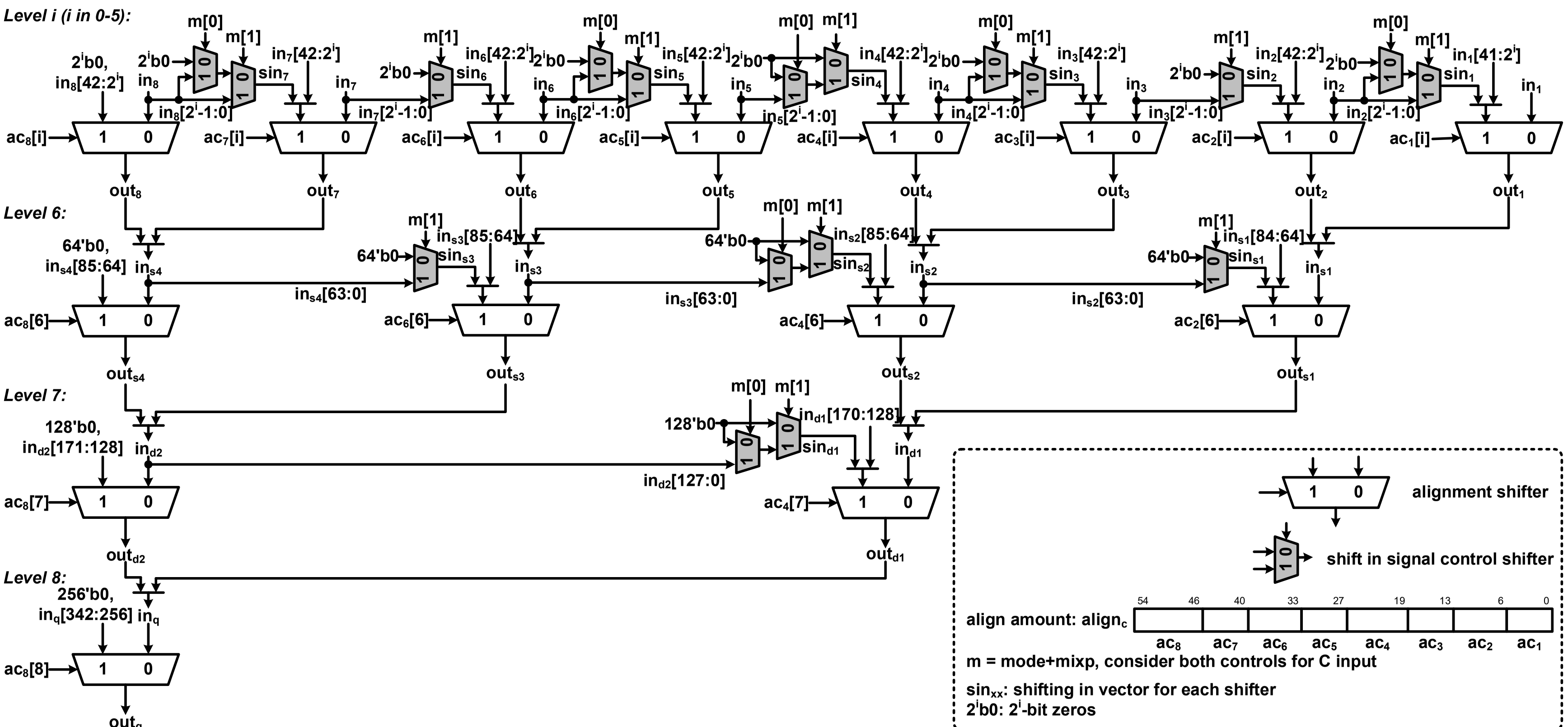


Figure 3.7: Design details of the alignment shifter of the proposed fused multiply-add unit

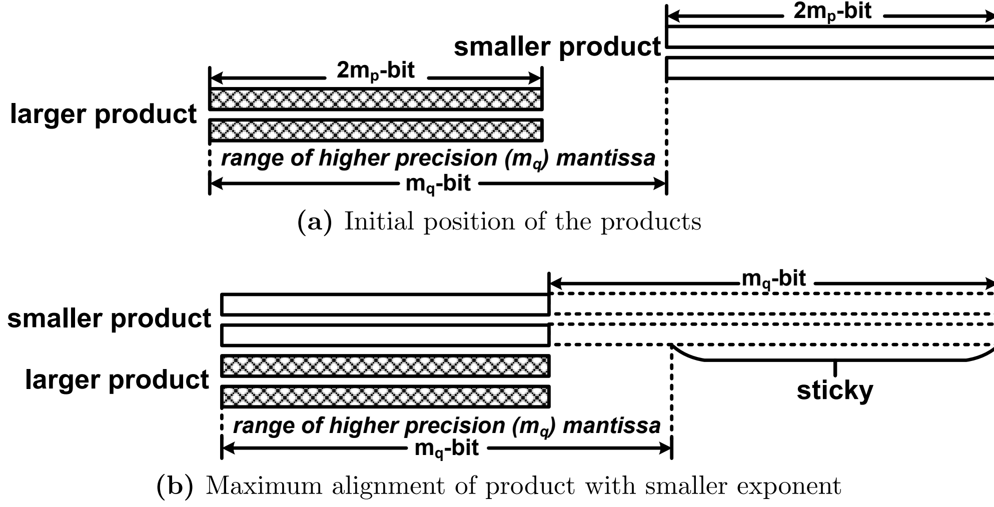
When performing multiplication, the product might become a subnormal number. However, this product is still a normal number in higher precision [45] [41]. For example, according to Table 2.1, for two subnormal numbers in HP, the minimum exponent is -14. The minimum exponent of the product is -28 which is still much larger than the minimum exponent of SP which is -126. Therefore, subnormal product in lower precision needs to be renormalized to perform normal operation in mixed-precision operations [45]. In the proposed design, we use two LZCs, each for one operand of the multiplication, in parallel to the multiplier, to generate the leading zero count in multiplication operand. The leading bit count in the product is equal to the sum of the leading bit counts of two operands.

Besides renormalizing the product, another way to handle subnormal product is to extend datapath. However, this method will lead to a significantly larger hardware cost. In this method, the adder datapath will be $3m_q + 5$ instead of $2m_q + 5$. The LZA bitwidth will become $2m_q$ instead of $m_q + 3$. And the alignment shifter for product will become $2m_q + 2$ instead of $m_q + 2$. Although these datapath are available in a multiple-precision FMA, using the wider datapath will lead to increased power consumption. Therefore, to reduce hardware cost, in the proposed design, we choose to renormalize the product.

The exponent difference between the two products are already obtained when preparing the larger product exponent to compare with that of C. The exponent difference will be further adjusted according to the leading bit count. That means if subnormal product happens, the exponent value after being normalized will be used to determine the alignment shifting amount. According to the status of the operands, there will be three different cases:

1. When both operands A and B are normal numbers, although the result might be subnormal in the same precision, the exponent value is still a normal exponent in higher precision. In addition, both A and B have no leading zeros, therefore, the leading zero count for their product is either one, in case $A \times B$ in $[1, 2)$, or zero, in case $A \times B$ in $[2, 4)$. The final exponent value of $A \times B$ is $e_A + e_B - lzc_P$ (e_A and e_B are exponent values without $bias$, lcz_P is the leading zero number in the product).

2. When both A and B are subnormal numbers, their exponents are both e_{min} in their precision. Assuming A has lzc_A leading zeros and B has lzc_B leading zeros, the product



(a) Initial position of the products

(b) Maximum alignment of product with smaller exponent

Figure 3.8: Alignment of products in mixed-precision dot-product mode

will have $lzc_P = lzc_A + lzc_B$ or $lzc_A + lzc_B + 1$ leading zeros. A leading digit detection will be performed to determine whether $lzc_A + lzc_B$ or $lzc_A + lzc_B + 1$ is the correct count. Then, the final exponent of $A \times B$ after being normalized is $2e_{min} - lzc_P$.

3. When one of A and B is subnormal, for example A, the exponent of their product is $e_{min} + e_B$. Assuming the leading zeros count for A is lzc_A, the product has $lzc_P = lzc_A$ or $lzc_A + 1$ leading zeros. Therefore, the final exponent of $A \times B$ after compensating the leading zeros count will be $e_{min} + e_B - lzc_P$.

The three cases can be combined that the final exponent used in exponent comparison module can be calculated as $e_P = e_A + e_B - lzc_P$. The larger e_P of two products will be used to compare with addend exponent e_{add} to generate the alignment amount for addend as the normal FMA does. The exponent difference of two products will also be calculated as $d_{prod} = |e_{PA_1B_1} - e_{PA_2B_2}|$. The product with smaller exponent will need to be right shifted d_{prod}-bit to be aligned with the product with larger exponent.

When implementing the alignment for the products, the renormalizing amount lzc_P and the alignment amount d_{prod} will be considered together in order to reduce the delay. For the two products, the one with larger e_P only needs to consider the renormalizing. Therefore, a left shifter is required for the larger product. For the one with smaller e_P, both lzc_P and d_{prod} are required to be considered. Both left shift and right shift are possible for the smaller

product. In order to simplify the shifter design, the smaller product is initially put to the right of the LSB of the higher precision product, as shown in Figure 3.8(a). Then, only a left shift of amount $shift_{prod} = shift_{max} + lzc_P - d_{prod}$ is required, where the $shift_{max}$ is the bitwidth of the mantissa in higher precision. For HP, SP, and DP mixed-precision operations, the $shift_{max}$ is 26-bit, 55-bit, and 115-bit, respectively.

The minimum value of $shift_{prod}$ is zero, because in the initial position, all bits of the smaller product will be compressed into the sticky bit and there is no need to further right shift the smaller product. The maximum value of $shift_{prod}$ is $shift_{max} + len_{prod} - 1$ where len_{prod} is the bitwidth of the product. It happens when the two exponent e_P are equal (and thus $d_{prod} = 0$) and the first non-zero bit of the smaller product is in LSB. In this case, the product needs to be left shift $len_{prod} - 1$ first for renormalizing and then left shift $shift_{max}$ to align with the other product. This process is shown in Figure 3.8(b). For the product with larger exponent, the maximum renormalizing amount is $len_{prod} - 1$.

In mixed-precision mode, the operations among products and addend are always addition. Therefore, the effective operations (eop_c for addend and eop_{prod} for product $A_2 \times B_2$) are only determined by the sign s of products and addend. The dot-product $A_1 \times B_1 + A_2 \times B_2$ is treated as a whole to simplify the analysis of eop_c. For $A_1 \times B_1 + A_2 \times B_2 + C$, the eop_c and eop_{prod} and the tentative sign of the result s_{tmp} can be determined by equation (3.1), where $s_{AB_{large}}$ is the sign of the larger product between $A_1 \times B_1$ and $A_2 \times B_2$. After multiplication, the second product of each mixed-precision operation needs to be inverted according to eop_{prod}.

$$\begin{aligned} eop_{prod} &= s_{A_1 B_1} \oplus s_{A_2 B_2} \\ eop_c &= s_{AB_{large}} \oplus s_c \\ s_{tmp} &= s_{AB_{large}} \end{aligned} \tag{3.1}$$

After inversion, these two sets of carry-save format products are swapped, where the one with larger exponent will be placed in the first place and the other one will be put in the second place. Then the alignment based on previous discussions will be performed. After alignment, these aligned products will be combined by one level of 4-2 carry save adder, and then passed to the normal FMA datapath to perform the remaining operations. For example, in HP mixed-precision dot-product mode, the rest of the FMA will run in SP

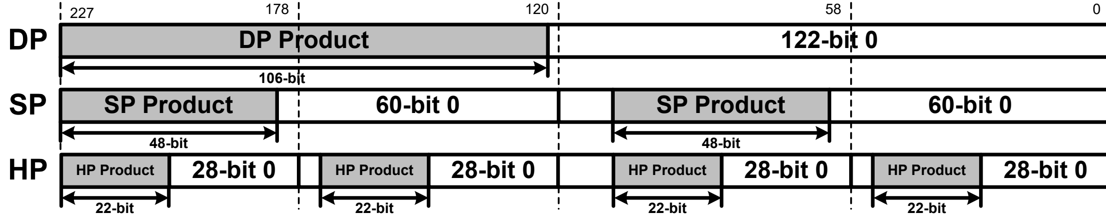


Figure 3.9: Unified input format of products alignment in mixed-precision operation mode

mode. To support product alignment in various precision modes, the corresponding products are rearranged as shown in Figure 3.9.

In [46], when performing dot-product operation, the alignment of products is done before the multiplication where the multiplier operands are shifted. However, in the proposed design, in order to handle the subnormal numbers in mixed-precision mode, renormalization after the multiplication is necessary. Therefore, in order to reduce the latency, the alignment of products is moved after the multiplication operation and combined with the renormalization shifting.

After combining the aligned products with one-stage carry save adder, the following processes will reuse the same hardware of the higher precision operations in normal FMA operations.

3.2.4 Adder

The diagram of the adder data-path used in the proposed FMA unit is shown in Figure 3.10. For QP operations, a total of 343-bit vectors need to be added. For the most significant 115-bit, an incrementer is used since only one of the two vectors contains useful data. For the less significant 228-bit, they need to be added using a carry propagate adder. In order to improve the performance, the 228-bit adder is divided into two smaller adders, one 108-bit adder for higher part and one 120-bit adder for lower part. For 108-bit higher adder, the results of both input carry 0 and 1 are calculated. And the correct result is selected by the carry out of the lower adder. Each of the two adders are further divided to fit lower precision operations. In lower precision, each sub-adder runs independently and in higher precision, the carry out from the lower order adder is sent to the higher order adder.

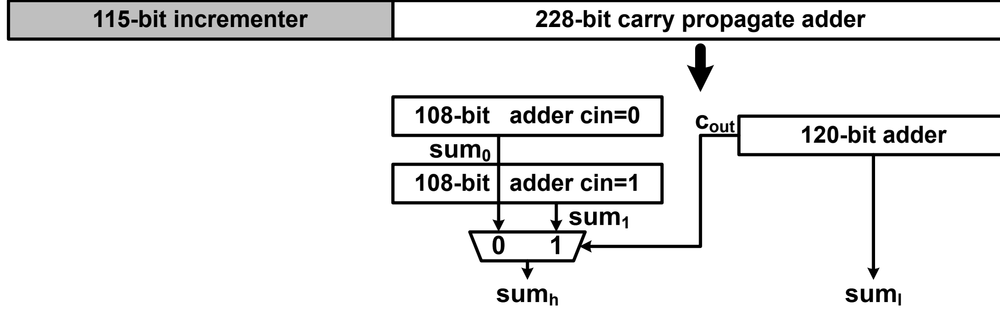


Figure 3.10: Diagram of the adder used in the proposed fused multiply-add

In normal FMA operation, the adder used for each precision is of $2m_q + 3$ in bitwidth (without considering the incrementer bitwidth for higher order part), where m_q is the mantissa precision of the addend (in normal FMA, this is also the precision of multiplier operand). In mixed-precision mode, the adder used for each precision becomes $m_q + 3$-bit. This $m_q + 3$-bit adder will occupy the higher order $m_q + 3$-bit of the available $2m_q + 3$-bit adder and the rest part will be disabled to save power.

3.2.5 Leading Zero Anticipation and Counting

The leading zero anticipator and counting unit runs in parallel with the mantissa adder. In order to support all four precision operations in a single unit, the inputs to the LZAC are rearranged as shown in the upper part of Figure 3.11. For lower precision operations, the input vectors are left aligned so that the LZAC can generate the correct amount of leading zeros. The trailing zeros will not affect the LZAC result.

The LZAC used in the proposed architecture is designed based on [50]. Since both positive and negative results can be generated by the adder, both leading zero (for positive result) and leading one (for negative result) need to be anticipated. Therefore, for the LZAC indicator, we use the unified indicator of both leading zero and leading one in [50]. According to [50], for k-bit operands A and B, the indicator generated by leading zero anticipator (LZA) unit should be:

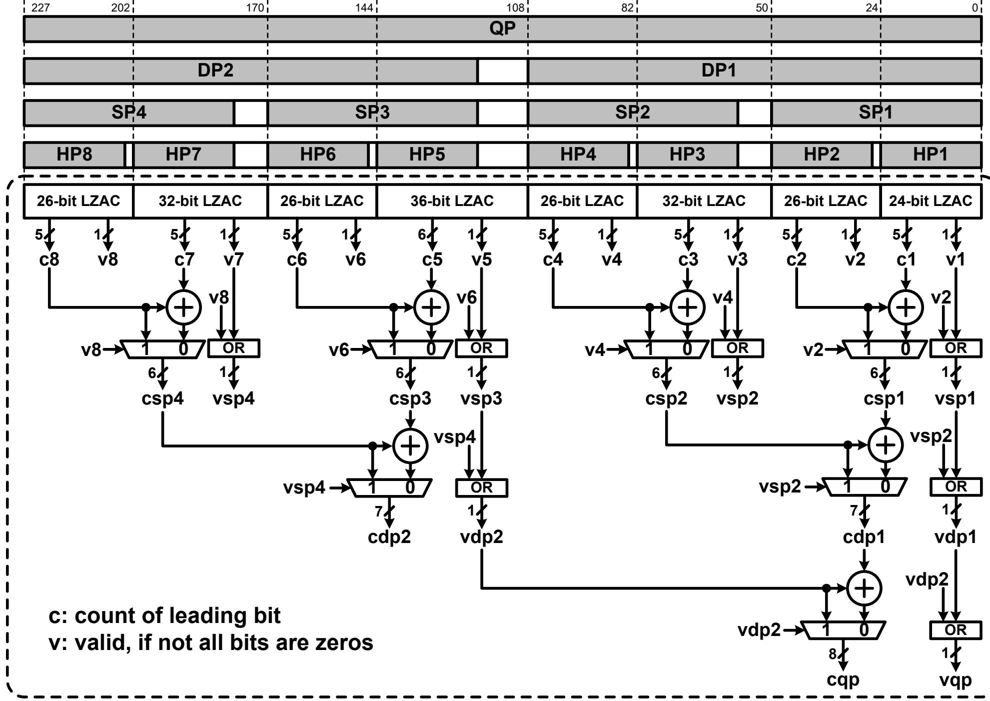


Figure 3.11: Unified input format for the leading zero anticipator and counting of the proposed fused multiply-add unit

$$
\begin{aligned}
f_{k-1} =& \overline{T}_{k-1} T_{k-2} \\
f_i =& T_{i+1}(G_i \overline{Z}_{i-1} + Z_i \overline{G}_{i-1}) + \\
& \overline{T}_{i+1}(Z_i \overline{Z}_{i-1} + G_i \overline{G}_{i-1}) \quad i < k-1
\end{aligned} \tag{3.2}
$$

where $T = A \oplus B$, $G = AB$, and $Z = \overline{A}\ \overline{B}$.

The detail of the LZAC is shown in Figure 3.11. The proposed design contains 8 smaller LZAC for each of the HP operations. LZAC results of every two LZACs are combined to determine the leading bit count for the four SP operations. For the next level, results of the first four LZACs are combined to determine the leading bit count for the one DP operation and the results of the last four LZACs are used for the other DP operation. In the last level, all 8 LZACs are combined to determine the leading bit count for QP operation.

A tree based structure is used for counting the leading bit count. A valid signal v is generated indicating not all bits of input indicator are zeros. The LZA used in the proposed design is not an exact LZA. An 1-bit error might be generated by LZA where the counting of the LZA will be 1-bit less than actual amount of leading bits. This 1-bit error can be

corrected in the final stage of the normalization shifter by detecting whether the leading bit of the result region is one (for subnromal result, this correction is disabled).

3.2.6 Normalization

The normalization shifter of the proposed FMA is similar to the alignment shifter, except that the normalization shifting is a left shifting. In the proposed design, a two-stage shifting method is applied where a constant shifter is applied first and then a dynamic shifter is used. When the exponent difference of C and the product (the larger product in mixed-precision mode) is larger than 2, the normalization amount is equal to the alignment amount. This shifting will be handled by the dynamic shifter and the constant shifter is not applied. Otherwise, a constant shifter is performed first with a shifting amount of $m_p + 2$ (or $m_q + 2$ in mixed-precision mode). Then a dynamic shifter will be performed with a shifting amount determined by the LZAC.

For the dynamic shifter, still four levels of shifters are applied. The first level contains 8 shifters, one for each of the HP operations. They consider shifting amount $[4 : 0]$. The next level contains 4 one-stage shifters that consider the sixth bit of shifting amount. They are used for SP, DP, and QP mode. The next level contains 2 one-stage shifters that consider the seventh bit of shifting amount. These two are used for DP and QP. Final level contains 1 one-stage shifter for QP operation to consider the eighth bit of shifting amount.

One more final stage shifting is added for correcting the error generated by the LZAC. For each result region, the leading bit is detected and when it is not one, 1-bit more left shifting is performed. For subnormal result (subnormal is detected when generating the normalization amount), this leading bit detection is disabled. For normal result, before performing the correction 1-bit shift, the exponent value will also be considered. When the exponent already reaches the minimum value before the correction shifting, this 1-bit shifting is not performed.

Before any shifting happens, the resulting exponent will be examined. If the resulting exponent is smaller than the minimum allowed exponent, then the result is subnormal and thus normalization shift will be performed with a shifting amount that makes the exponent to be the minimum exponent.

3.2.7 Rounding

In the proposed FMA architecture, 8 rounding units are designed. In HP mode, each operation uses one of the rounding units. The SP mode shares the first, third, fifth, and seventh rounding units with the HP mode. The DP mode shares the fourth and eighth rounding units with HP and SP mode. The QP mode will share the eighth rounding unit with HP, SP, and DP mode. The $roundTiestoEven$ rounding mode [8] is applied.

After rounding, the post processing module handles exceptional cases. And then the generated mantissa is combined with the corresponding sign and exponent to form the IEEE 754-2008 format output.

3.3 Results and Analysis

The model of the proposed FMA architecture is implemented using Verilog. Simulations with extensive testing vectors are performed to verify the functionality of the proposed design. The testing vectors are generated with the help of TestFloat [51]. In addition to the proposed architecture, the separate HP FMA, SP FMA, DP FMA, and QP FMA are also implemented to compare with the proposed multiple-precision design. Moreover, a dual-mode FMA architecture that uses the same techniques as the proposed FMA is also implemented. The purpose to implement this architecture is to compare with previous dual-mode FMA designs. The proposed dual-mode FMA supports DP FMA or two parallel SP FMA operations. It also supports SP mixed-precision operation (mixed-precision dot-product and FMA).

All these designs are synthesized with STM-90nm technology with normal case parameters (1.00V and 25℃) using Synopsys Design Compiler. The simulation of these designs are done in Modelsim. The power estimation is done with Synopsys PrimeTime PX using the synthesized netlist and value change dump (VCD) file generated from the post-synthesis simulation.

Each pipeline stage of the proposed FMA architecture is synthesized first. The synthesis results are shown in Table 3.1. Each pipeline stage is synthesized with a timing constraint of 2.0 ns. As shown in Table 3.1, the first pipeline stage consumes the most area and power

Table 3.1: Synthesis results of each pipeline stage of the proposed FMA

Stage	Delay (ns)	Area (μm^2)	Power (mW)
multiplication & addend alignment	2.0	466500	112.5
product alignment	2.0	45984	9.8
addition	2.0	175300	35.0
rounding & normalization	2.0	107006	19.7
Total	2.0	794790	177.0

due to the large mantissa multiplier. The area of the product alignment stage together with the area of LZC in the multiplication stage are the area overhead to support mixed-precision operations. As shown in Table 3.1, these extra logic only consumes 6.5% of the total area. With this small area overhead, the FMA can perform more functionality, such as mixed-precision dot-product and mixed-precision FMA, than conventional multiple-precision FMAs.

The whole proposed FMA and the separate HP, SP, DP, QP FMA are then synthesized. The synthesis results of each design are shown in Table 3.2. In Table 3.2, QP-MP-FMA is the design obtained by removing all logic for mixed-precision support from the proposed architecture. In addition, DP-MIX-FMA is the design obtained by removing all the QP datapath from the proposed design. DP-MP-FMA is the design obtained by removing all logic for mixed-precision support from DP-MIX-FMA. These DP-MP-FMA and DP-MIX-FMA designs are added because the overwhelming cost of the QP datapath might hide the cost or benefit of supporting mixed-precision operations. As shown in Table 3.2, the proposed FMA consumes only 38% more area compared to a conventional QP FMA. However, as discussed in previous sections, it can accomplish many more operations, such as parallel HP/SP/DP FMAs and mixed-precision operations, in addition to the QP FMA operation. The area overhead comes mainly from the multiplexers used to support multiple-precision data selection and the logic to support mixed-precision operations. On the other hand, in terms of functionality, we need 8 HP FMAs, 4 SP FMAs, 2 DP FMAs, 1 QP FMA, and several other mixed-precision FMAs to realize the same functionality as the proposed FMA provides. However, the combination of these FMAs consume much larger area compared to the proposed FMA. In addition, compared to QP-MP-FMA, the proposed design only con-

Table 3.2: Area and energy comparison of the proposed FMA with standalone FMA designs[1]

FMA Designs		Delay (ns)	Cycles	Area (μm^2)	Power (mW)	Energy/op[2] (J)			
						HP	SP	DP	QP
HP-FMA		0.90	3	16800	5.9	1.62×10^{-11}	N/A	N/A	N/A
SP-FMA		1.08	3	50400	13.6	N/A	4.41×10^{-11}	N/A	N/A
DP-FMA		1.40	3	166160	39.5	N/A	N/A	1.66×10^{-10}	N/A
DP-MP-FMA[3]		1.50	3	180610	43.8	4.92×10^{-11}	9.85×10^{-11}	1.97×10^{-10}	N/A
DP-MIX-FMA[3]	normal	1.50	3	191085	47.3	5.32×10^{-11}	1.06×10^{-10}	2.13×10^{-10}	N/A
	mixed		4			N/A	1.41×10^{-10}	2.83×10^{-10}	N/A
QP-FMA		1.90	3	574570	118.0	N/A	N/A	N/A	6.73×10^{-10}
QP-MP-FMA[4]		2.00	3	745842	160.0	1.20×10^{-10}	2.40×10^{-10}	4.80×10^{-10}	9.60×10^{-10}
Proposed (QP-MIX-FMA)	normal	2.00	3	794790	177.0	1.32×10^{-10}	2.65×10^{-10}	5.30×10^{-10}	1.06×10^{-9}
	mixed		4			N/A	3.55×10^{-10}	7.10×10^{-10}	1.42×10^{-9}

[1] Energy of mixed-precision mode is reported with addend (C) precision.
[2] energy/op = (delay×cycles×power)/num_parallel_operations.
[3] DP-MP-FMA: multiple-precision FMA supports 1 DP/2 SP/4 HP; DP-MIX-FMA: DP-MP-FMA with HP or SP mixed-precision supports.
[4] QP-MP-FMA: the proposed design removing the mixed-precision support.

sumes 6.5% more area. Therefore, with small area overhead, the proposed FMA can support many more functionalities, such as mixed-precision dot-product and mixed-precision FMA, than conventional multiple-precision FMAs support. After removing the QP datapath, the DP design with mixed-precision support, DP-MIX-FMA, only has 15% more area compared to the standard DP FMA. The area overhead of DP-MIX-FMA to support mixed-precision in a multiple-precision architecture compared to DP-MP-FMA is 5.8% after removing the QP datapath. The energy consumption per operation for standalone FMA units and those merged units are also shown in Table 3.2. Here the energy consumption of a single operation is reported. The total delay of the operation is calculated as the delay of each pipeline stage multiplied by the number of clock cycles. Due to the sharing of resources among different operation modes, the energy per operation is increased for each operation mode. However, with the proposed architecture, multiple functionalities, such as multiple-precision FMA, mixed-precision FMA, and mixed-precision dot-product, can be supported in a single architecture.

The comparison of the functionality of the proposed FMA with previous multiple-precision FMAs are shown in Table 3.3. Compared to other multiple-precision FMAs, the proposed FMA supports HP operations in addition to SP/DP/QP operations. In addition, the proposed FMA supports mixed-precision FMA (MIX-FMA) operations and mixed-precision dot-product operations. The design in [46] also supports dot-product operations, however, its addend and products are of the same precision. In addition, for dot-product operation, they choose to sum all the products together (the 4-term dot-product) instead of supporting multiple parallel 2-term dot-products. The 4-term dot-product can perform a single dot-product operation faster. However, for applications that require small size (2-term) of dot-product operations, the 4-term dot-product unit is not fully utilized. In Table 3.3, the throughput is compared in a unified method where $op/FO4$ is used. Within all the designs that support QP operation, the proposed design can achieve the highest throughput.

The comparison of the dual-mode mixed-precision FMA using the same techniques as the proposed FMA with some other dual-mode FMA works [34] [46] [35] [36] is shown in the upper part of Table 3.4. All these designs support one DP FMA operations or two parallel SP FMA operations. In addition, the proposed dual-mode FMA also supports mixed-

Table 3.3: Comparison of functionality with multiple-precision FMA designs

Design	Floating-Point Functions	Latency			Throughput		
		cycle	delay		op/cycle	op/sec	op/FO4
			ns	FO4		$(MOPS)$	$(\times 10^{-3})$
[35]	one DP FMA/MUL/ADD	3	3.40	34.3	1	294	29
	two SP FMA/MUL/ADD	3	3.40	34.3	1	294	29
[34]	one DP FMA/MUL/ADD	3	3.43	52.7	1	291	18
	two SP FMA/MUL/ADD	3	3.43	52.7	1	291	18
[46] DPFMA	one DP FMA	4	3.61	36.5	1	277	27
	two SP MUL	4	3.61	36.5	1	277	27
	one SP (2-term) Dot-Product	4	3.61	36.5	1	277	27
[46] QPFMA	one QP FMA	4	4.74	47.8	1	210	21
	two DP MUL	4	4.74	47.8	1	210	21
	four SP MUL	4	4.74	47.8	1	210	21
	one DP (2-term) Dot-Product	4	4.74	47.8	1	210	21
	one SP (4-term) Dot-Product	4	4.74	47.8	1	210	21
[36]	one DP FMA/MUL/ADD	8	3.24	49.8	1	308	20
	two SP FMA/MUL/ADD	8	3.24	49.8	1	308	20
[37]	one QP FMA/MUL	4	2.15	47.7	0.5	232	10
	one QP ADD	3	2.15	47.7	1	465	21
	two DP FMA/MUL/ADD	3	2.15	47.7	1	465	21
	four SP FMA/MUL/ADD	3	2.15	47.7	1	465	21
[38]	one QP FMA/MUL/ADD	3	3.41	110	1	293	9
	two DP FMA/MUL/ADD	3	3.41	110	1	293	9
	four SP FMA/MUL/ADD	3	3.41	110	1	293	9
Proposed	one QP FMA/MUL/ADD	3	2.00	44.5	1	500	22
	two DP FMA/MUL/ADD	3	2.00	44.5	1	500	22
	four SP FMA/MUL/ADD	3	2.00	44.5	1	500	22
	eight HP FMA/MUL/ADD	3	2.00	44.5	1	500	22
	one DP MIX-FMA	4	2.00	44.5	1	500	22
	two SP MIX-FMA	4	2.00	44.5	1	500	22
	four HP MIX-FMA	4	2.00	44.5	1	500	22
	one DP MIX (2-term) Dot-Product	4	2.00	44.5	1	500	22
	two SP MIX (2-term) Dot-Product	4	2.00	44.5	1	500	22
	four HP MIX (2-term) Dot-Product	4	2.00	44.5	1	500	22

Table 3.4: Comparison of the proposed FMA with previous works

Designs	Area		Delay		Latency				Throughput				Power
	μm^2	NAND2[1]	ns	FO4[1]	HP	SP	DP	QP	HP	SP	DP	QP	mW
[35] TSMC 180nm	708590	58081	3.40	34.3	N/A	3	3	N/A	N/A	1	1	N/A	-
[46] DPFMA TSMC 180nm	-	38509	3.61	36.5	N/A	4	4	N/A	N/A	1	1	N/A	-
[34] TSMC 130nm	286766	56228	3.43	52.7	N/A	3	3	N/A	N/A	1	1	N/A	35.2
[36] TSMC 130nm	149000	29215	3.24	49.8	N/A	8	8	N/A	N/A	1	1	N/A	17.8
Proposed Dual-Mode STM 90nm	**172014**	**39094**	**1.50**	**33.3**	**N/A**	**3/4**	**3/4**	**N/A**	**N/A**	**1**	**1**	**N/A**	**26.2**
[46] QPFMA TSMC 180nm	-	147838	4.74	47.8	N/A	4	4	4	N/A	1	1	1	-
[37] TSMC 90nm	718725	163346	2.15	47.7	N/A	3	3	4	N/A	1	1	0.5	161.2
[38] UMC 65nm	672046	420028	3.41	110	N/A	3	3	3	N/A	1	1	1	381.0
Proposed FMA STM 90nm	**794790**	**180634**	**2.00**	**44.5**	**3/4**	**3/4**	**3/4**	**3**	**1**	**1**	**1**	**1**	**177.0**

[1] 1 FO4 ≈ 31ps, 1 NAND2 ≈ 1.44 μm^2 @ 65nm; 1 FO4 ≈ 45ps, 1 NAND2 ≈ 4.4 μm^2 @ 90nm;
1 FO4 ≈ 65ps, 1 NAND2 ≈ 5.1 μm^2 @ 130nm; 1 FO4 ≈ 99ps, 1 NAND2 ≈ 12.2 μm^2 @ 180nm;

precision SP dot-product and mixed-precision SP FMA. The DPFMA in [46] supports SP dot-product but the addend is also SP number. These designs are synthesized using different semiconductor technologies. In order to make a fair comparison, the equivalent area (NAND2 gate count) and equivalent delay (FO4 delay) for each design are calculated. In terms of delay, the proposed dual-mode design has a 3%-36% shorter critical path delay compared to [34] [46] [35] [36]. The proposed design also requires 5 fewer pipeline stages than the design of [36] when performing normal FMA operations. The critical path of the FMA is usually located in the mantissa multiplier. In the proposed design, the radix-4 Booth multiplier that has smaller critical path delay than the array multiplier is utilized which effectively reduces the critical path delay. Compared to [34] and [35], the proposed dual-mode FMA has 33% and 30% smaller area, respectively. Both [34] and [35] use array multiplier that can simplify the multiple-precision design. However, the radix-4 Booth multiplier used in the proposed design consumes less area than array multiplier used in [34] and [35]. Although the mixed-precision support in the proposed design brings extra logic, the proposed design still has smaller area compared to [34] and [35]. Compared to [46], the proposed design has slightly (1.5%) larger area. The design in [46] only supports two SP multiplications in SP mode as shown in Table 3.3 whereas the proposed design supports two parallel SP FMA operations. Therefore, the logic for addend alignment in the proposed design is more complex. In addition, the mixed-precision dot-product in the proposed design requires a larger product alignment datapath than the normal dot-product in [46] which also leads to a larger area. The design in [36] uses Karatsuba algorithm [52] to reduce the number of required multiplier. In addition, the design in [36] uses 6 pipeline stages for the mantissa multiplier. The timing constraints for each pipeline stage will be loose which further helps to reduce the area during synthesis. Therefore, it has the smallest area among the four compared dual-mode designs. In terms of power consumption, [36] has the smallest power due to its small area. The proposed dual-mode design has 25% smaller power than the design of [34].

In the lower part of Table 3.4, the comparison of the proposed multiple-precision FMA with three previous multiple-precision FMAs that support up to QP operations, the QPFMA in [46], [37], and [38], is provided. The proposed design still has the smallest critical path

delay (6.7%-60% reduced) among these FMA designs due to the reduction of critical path delay by using radix-4 Booth multiplier. In terms of throughput, all designs except [37] have a throughput of 1 for all precisions. The QP support in [37] is designed with iterative method. The iterative method can help to reduce the area, however, the QP throughput is also reduced. In terms of area, compared to [38], the proposed design has 57% smaller area. The design in [38] is based on the FMA architecture that optimized for floating-point addition. When performing floating-point addition using that architecture, the mantissa multiplier can be bypassed and the latency of floating-point addition is reduced. In that architecture, the two-path floating-point addition algorithm is applied after the mantissa multiplier which leads to a large area consumption. The proposed FMA has 22% larger area compared to the QPFMA design in [46]. As shown in Table 3.3, the proposed design has full support of QP/DP/SP/HP FMA operations. However, the QPFMA in [46] only has full support for QP FMA and for DP and SP, only the multiplications are supported. Therefore, the proposed design has a more complex addend alignment logic. In addition, due to the fact that dot-product operation in the proposed design is supported in mixed-precision method, the alignment datapath is wider than that of [46] which leads to a larger area. Compared to [37], the proposed FMA has 10.6% larger area. The extra hardware comes from two aspects. On one hand, the proposed FMA requires more hardware to support HP FMA operations and the mixed-precision operations. On the other hand, because the QP operation of [37] is done in iterative method, the area of their mantissa multiplier is reduced. Therefore, although the proposed FMA uses resource efficient radix-4 Booth multiplier, it still consumes more area than [37]. However, with only 10.6% area overhead, the proposed FMA can support many more functionalities than [37], such as parallel HP FMAs, mixed-precision FMAs, and mixed-precision dot-products, as shown in Table 3.3. In terms of power consumption, the proposed design has 9.8% larger power consumption compared to [37] due to a larger area. Compared to [38], although the synthesis technology of [38] has smaller feature size than that used by the proposed design, the proposed design consumes 53% less power than [38] due to the smaller area of the proposed design. Overall, the proposed FMA can support more functionalities (HP FMA and mixed-precision operations) than the state-of-the-art multiple-precision FMA design [37] with no significant area overhead.

The design in [45] is a mixed-precision FMA that supports one SP multiplication result accumulating to a DP addend. For the proposed FMA, in SP mixed-precision dot-product mode, two dot-products work in parallel. Therefore, the proposed design can accumulate multiple sets of two vectors faster than [45]. In addition, the proposed design supports mixed-precision operations for other precisions, such as HP products accumulating to SP addend. The multi-function FMA design in [46] also supports dot-product operation. However, the addend has the same precision as the products. According to the discussion in [44] and [10], mixed-precision accumulation can effectively reduce the rounding error. Therefore, in practical applications, the proposed FMA design can provide more accurate results than [46], especially when the size of two vectors to be accumulated is large.

3.4 Summary

In this chapter, an efficient multiple-precision FMA architecture is designed and implemented. The proposed FMA architecture supports 1 quadruple-precision, 2 double-precision, 4 single-precision, or 8 half-precision operations. The proposed FMA also supports mixed-precision operations. Due to the availability of the datapath for various precisions in the proposed multiple-precision FMA architecture, the mixed-precision operations can be realized without significant area overhead. By adding extra logic, the proposed FMA supports mixed-precision FMA operation and mixed-precision 2-term dot-product operation for various precisions. Compared to a normal multiple-precision FMA, the mixed-precision FMA also supports mixed-precision FMA operations and mixed-precision dot-product operations with only 6.5% more area. The mixed-precision operations are favorable in many applications that use lower precision to improve performance and use higher precision to maintain accuracy. Compared to the state-of-the-art multiple-precision FMA designs, the proposed FMA newly adds support for half-precision FMA operations and mixed-precision operations with only 10.6% more area. The proposed FMA architecture can be used in efficient processor designs or specialized hardware accelerators. The support of parallel half-precision operations and the mixed-precision operations makes the proposed design suitable in accelerating deep learning applications.

Part III

Arithmetic Unit for Deep Learning Computation

Chapter 4

Multiple-Precision Multiply Accumulate Unit[1]

This chapter presents the design of a fixed-point and floating-point merged mixed-precision MAC unit for deep neural network training and inference. The proposed MAC unit supports both floating-point operations, for deep neural network training, and fixed-point operations, for deep neural network inference. Section 4.1 presents the motivations to design the proposed MAC unit. The design details of the proposed MAC architecture is presented in Section 4.2. The synthesis results of the proposed MAC unit is presented in Section 3.3. Section 4.4 concludes this chapter.

4.1 Introduction

Deep learning [54] has achieved great success in recent years. However, the hardware cost to implement a deep learning model is high since it contains large amount of parameters and is computationally intensive. Many research works [5] have been done recently in order to reduce the hardware cost. As the core of computation, the functionality of arithmetic unit can determine the functionality of the whole hardware processor. In this paper, we will focus on novel arithmetic unit design to enrich the functionality of deep learning processor.

For the inference operations of deep learning applications, 8-bit fixed-point or even lower precisions have been proved to be sufficient to maintain high accuracy [11] [12]. However,

[1]The content of this chapter is originally published in the proceedings of 2018 IEEE International Symposium on Circuits and Systems (ISCAS 2018) [53]. The manuscript has been reformatted for inclusion in this thesis.

Hao Zhang (HZ), Hyuk-Jae Lee (HL) and Seok-Bum Ko (SK) designed this study. HZ developed the architecture, developed the HDL model of the architecture, and performed logic synthesis and results analysis. HZ prepared the manuscript with contributions from HL and SK to the manuscript structure, readability and analysis and discussion of the results.

for training operations, larger data range is still required to handle the computation of gradients and back-propagation. Recent research work [9] has reported that 16-bit half-precision operations can successfully train the deep learning models. Therefore, in order to efficiently support both training and inference, a fixed/floating-point merged arithmetic unit is required.

In deep learning applications, a large amount of multiplication results need to be accumulated to obtain a single output. Therefore, a number format with larger range and higher precision is required to maintain the accuracy. This is suggested in [10] and [47]. The results of half-precision multiplication can be accumulated to single-precision format. Similarly, the results of 8-bit fixed-point multiplication can be accumulated to 32-bit fixed-point format. The accumulation to higher precision can avoid data loss due to rounding. A mixed single/double-precision fused multiply-add unit has been proposed in [45]. However, for deep learning applications, lower precision operations are preferred.

In this chapter, a novel fixed/floating-point merged mixed-precision multiply-accumulate unit is proposed to support the above mentioned deep learning training and inference operations. For deep learning training, the proposed architecture supports 16-bit half-precision multiplications and the product will be accumulated to a 32-bit single-precision accumulator. During inference operations, the two 16-bit inputs are filled with two sets of 8-bit fixed-point operands. The proposed architecture will perform two parallel 8-bit multiplications. These two products can be added and then accumulated to 32-bit fixed-point accumulator. The fixed/floating-point merged multiplication is realized by the Karatsuba algorithm [52]. With the proposed arithmetic unit, the deep learning processor can support training operations as well as high-throughput inference operations.

4.2 The Proposed Design

The data-path of the proposed fixed/floating-point merged multiply-accumulate unit is shown in Figure 4.1. It is similar to a standard FMA architecture [28], except the third operand (*accumulator*) is in higher precision. The whole design contains three pipeline stages. The first pipeline stage is for low-precision multiplication and alignment of the high-precision

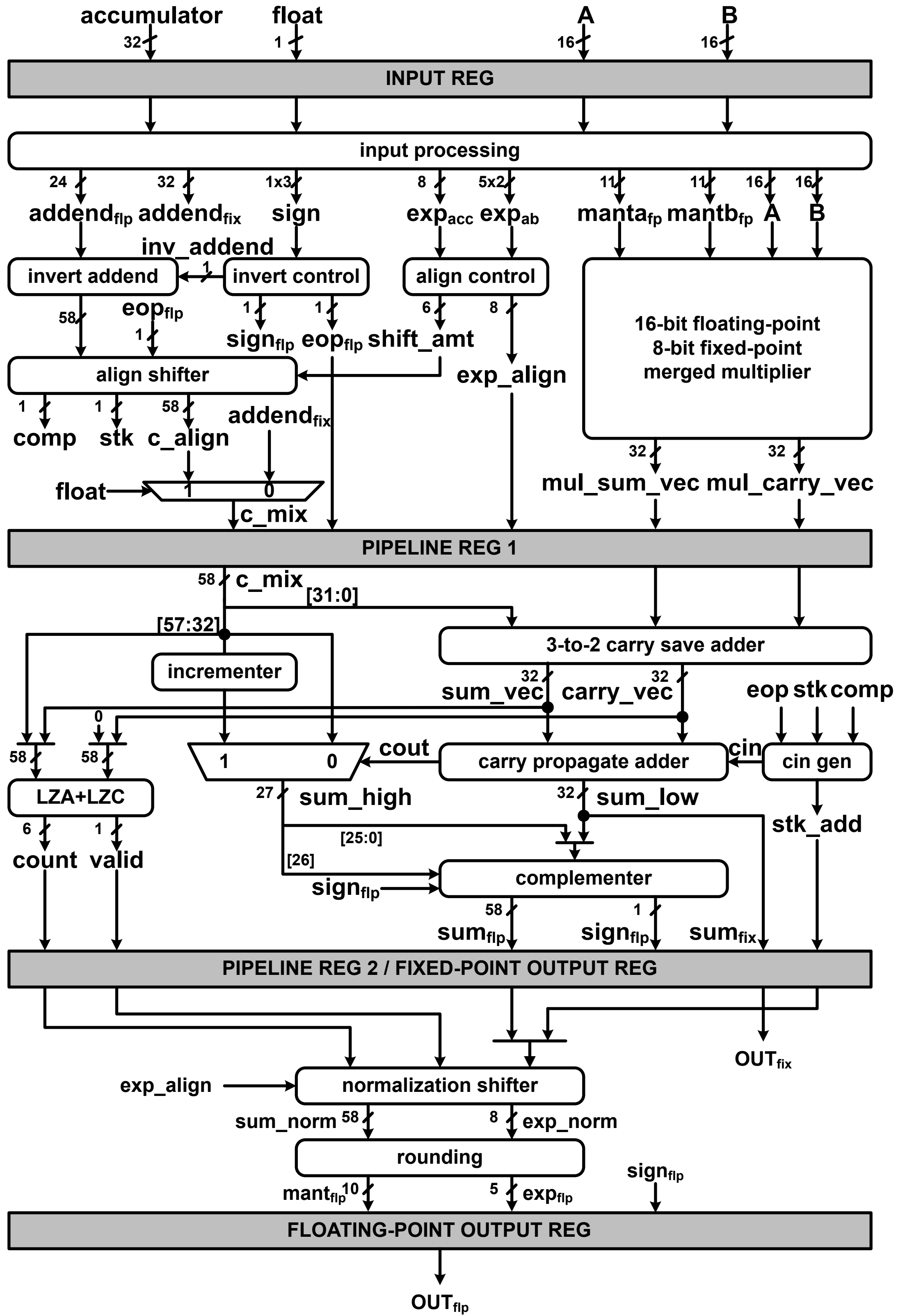


Figure 4.1: Data-path of the proposed fixed/floating-point merged mixed-precision multiply-accumulate unit

accumulator. The second pipeline stage is for the accumulation operation and leading zero detection for floating-point operations. The last pipeline stage is for floating-point normalization and rounding operation. The floating-point operation requires the whole three pipeline stages while the fixed-point operation requires the first two pipeline stages.

4.2.1 Floating-Point Mode

The proposed design accepts a control signal $float$ which is to control whether floating-point operation or fixed-point operation is performed. In floating-point (FLP) mode, A and B are two half-precision numbers and $accumulator$ is a single-precision number. The proposed architecture can accomplish $A \times B + accumulator$ operation.

The mantissa of A and B, $manta_{fp}$ and $mantb_{fp}$, are extracted and sent to the merged multiplier to generate the carry-save format result. In half-precision multiplication, 22-bit is enough for the product. However, in order to support 32-bit accumulation in fixed-point mode, the products are extended to 32-bit by padding zeros to the right of the actual products. At the same time, the accumulator is inverted in case of subtraction ($eop_{flp} = 1$) and then aligned to be ready for accumulation. Similar to the FMA design in [19], the $accumulator$ is placed two bits to the left of the carry-save format products. This is to ensure there will be no overflow when accumulating and the $accumulator$ only needs to be shifted to the right. In the proposed design, a 58-bit shifter is required. When generating the shifting amount, the exponents of A and B are first converted to single-precision exponent by adding the bias difference of half-precision and single-precision.

In the next stage, the lower 32-bit of the aligned $accumulator$ is combined with carry-save format products and then go through the carry-propagate adder to generate the lower part accumulation result. The higher part of the aligned $accumulator$ is incremented and the output carry of the lower carry-propagate adder determines whether incremented $accumulator$ is chosen. In parallel, an leading-zero anticipator and counter [50] is used to generate the shifting amount for normalization shifter.

In the last stage, the accumulation result is normalized and rounded to single-precision mantissa bit-length. The sign, exponent, and mantissa are combined to form the IEEE 754 single-precision format.

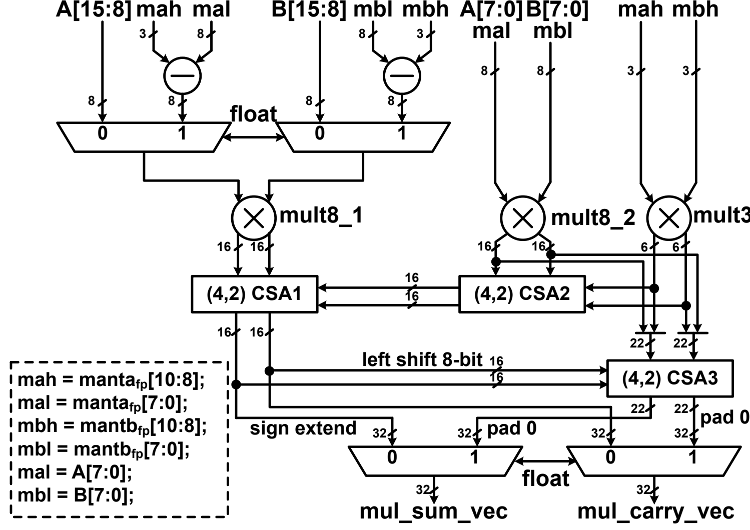


Figure 4.2: Architecture of the proposed fixed/floating-point merged multiplier

4.2.2 Fixed-Point Mode

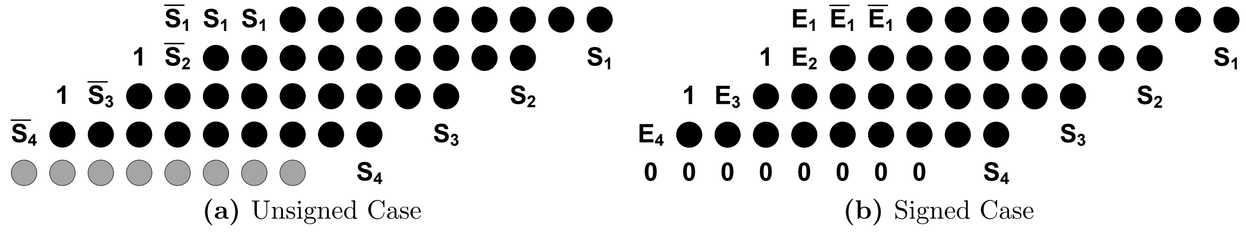
In fixed-point (FIX) mode, A (B) contains two sets of 8-bit fixed-point numbers A_h, A_l (B_h, B_l). The *accumulator* contains one 32-bit fixed-point number. The proposed design can perform $A_h \times B_h + A_l \times B_l + accumulator$ operation.

Fixed-point numbers are in two's complement format. Therefore, inversion is not required for *accumulator*. In addition, the alignment is also not required and the 32-bit *accumulator* is directly put to the lower 32-bit position. Moreover, complementer in the second stage is not required since the fixed-point is in two's complement format.

4.2.3 Merged Multiplier Design

The architecture of the merged FLP and FIX multiplier is shown in Figure 4.2. It contains two 8-bit multipliers, $mult8_1$ and $mult8_2$, and one 3-bit multiplier $mult3$. Two 8-bit multipliers, $mult8_1$ and $mult8_2$, are used for two parallel multiplications in FIX mode. These two products are accumulated by $(4,2)CSA1$ and then the results are sign extended to 32-bit to form the final multiplier output.

In FLP mode, the $manta_{fp} \times mantb_{fp}$ is calculated with the Karatsuba algorithm. Only one more 3×3 multiplier, in addition to these two 8-bit multipliers, is enough to generate



(a) Unsigned Case **(b)** Signed Case

Figure 4.3: Partial products arrangement of modified booth multiplier

half-precision product. $manta_{fp}$ and $mantb_{fp}$ are rewritten as:

$$\begin{aligned} manta_{fp} &= mah \cdot 2^8 + mal \\ mantb_{fp} &= mbh \cdot 2^8 + mbl \end{aligned} \tag{4.1}$$

Then according to the Karatsuba algorithm, $mah \times mbh$ is calculated by $mult3$, $mal \times mbl$ is calculated by $mult8_2$, and $(mah - mal) \times (mbh - mbl)$ is calculated by $mul8_1$. Here, an addition in $(4,2)CSA1$ is required to accumulate two products in FIX mode. Therefore, we modify the last term in equation (2) to $(A_1 - A_2) \times (B_2 - B_1)$, so that the subtraction of this term will become addition. Therefore, $mult3$ here calculates $(mah - mal) \times (mbl - mbh)$. $(4,2)CSA2$ will accumulate $mah \times mbh$ and $mal \times mbl$. The accumulated result will be combined with $(mah-mal)\times(mbl-mbh)$ by $(4,2)CSA1$ to generate $mah\times mbl+mal\times mbh$. Finally, $mah\times mbl+mal\times mbh$ is shifted by 8-bit and $mah\times mbh$ is shifted by 16-bit. These two together with $mal \times mbl$ are combined by $(4,2)CSA3$ to generate the final carry-save format product.

Since $mult3$ is used only for FLP mode, it is an unsigned multiplier. $mult8_1$ and $mult8_2$ are used in both FLP and FIX mode. In FLP mode, an unsigned multiplier is required while in FIX mode, a signed multiplier is required. The modified booth multiplier [49] is used for two 8-bit multiplier. The algorithm is modified to support both signed and unsigned multiplication.

The partial product arrangement of signed and unsigned booth multiplier is shown in Figure 4.3. The dots are either ±multiplicand or ±2×multiplicand. For unsigned case, the multiplicand is always extended with zero and in signed cases, the multiplicand is sign extended. This can be handled by the control signal $float$. The partial product generate logic is same for signed and unsigned cases. So the S bit which is the complement bit in the

Table 4.1: Synthesis results of the combinational logic of each pipeline stage

Pipeline	Delay (ns)	Area (μm^2)	Power (mW)
Stage 1	0.74	20003.75	7.5
Stage 2	0.70	7958.69	1.87
Stage 3	0.68	4227.95	0.63

case of negative partial product is used in common. In order to support signed cases, the E bit, which is used to determine whether the partial product and multiplicand have the same sign, is also generated. When extending $\pm 1\times$ or $\pm 2\times$multiplicand, as shown in Figure 4.3, whether to use S bit or E bit can be handled by $float$.

The unsigned case has one more partial product than the signed case. However, in 8-bit case, the most significant two bits of the last group of multiplier bits to generate partial product is always "00". Therefore, the last partial product is either zero or $1\times$multiplicand. We do not need partial product selector to generate the last partial product. In addition, in signed case, the $float$ control can be used to set the last partial product all zeros.

4.3 Results and Analysis

The proposed design is implemented with VHDL. Simulations with extensive testing vectors are performed to verify the functionality of the proposed design. The proposed design is then synthesized with STM-90nm technology with normal case parameters using Synopsys Design Compiler.

The combinational logic of each pipeline stage is first synthesized. The synthesis result of each pipeline stage is shown in Table 4.1. The critical path is in the first pipeline stage where the time consuming multiplier is implemented. The first stage also has the largest area. The multiplier is area consuming and alignment shifter also occupies large area.

The whole proposed architecture, including all pipeline registers, is synthesized as shown in Table 4.2. The proposed design has a worst case delay of 0.8 ns, occupies 42710.90 μm^2 area. The power consumption of the proposed design is 14.07 mW when running with a 0.8 ns clock period. Two parallel 8-bit multiplications accumulating to 32-bit fixed-point

Table 4.2: Comparison of the proposed multiply-accumulate unit with single-mode multiply accumulate unit

	Latency	Throughput (GOPS)	Worst Delay		Area		Power (mW)
			ns	FO4*	μm^2	NAND2*	
FIX MAC†	2	1.43	0.7	15.56	13257.91	3013	6.65
FLP MAC‡	3	1.25	0.8	17.78	40817.54	9276	13.59
Proposed MAC	2 (FIX) 3 (FLP)	2.50 (FIX) 1.25 (FLP)	0.8	17.78	42710.90	9707	14.07

† Two 8-bit fixed-point multiplication with 32-bit fixed-point accumulation
‡ 16-bit floating-point multiplication with 32-bit floating-point accumulation
* 1 FO4 ≈ 45ps, 1 NAND2 ≈ 4.4 μm^2 @ 90nm

accumulator can be accomplished in two clock cycles. One 16-bit floating-point multiplication accumulating to 32-bit floating-point accumulator can be accomplished in 3 clock cycles. The proposed architecture is fully pipelined. Each new operation can be started every clock cycle.

The area and power consumption of the proposed design under different delay requirements is shown in Figure 4.4. With the delay constraint becomes larger, the area and power of the proposed design become smaller. To achieve the best performance, in this paper, the point with the smallest delay is chosen as the design point. In other cases, one can choose the appropriate design point according to the requirements.

To the best of our knowledge, there is no such fixed/floating-point merged design appeared in the literature. In order to show the advantage of our proposed design, two single-mode multiply-accumulate units are designed. One is the fixed-point multiply-accumulate unit which supports two parallel 8-bit fixed-point multiplications and accumulates the products to 32-bit fixed-point number. This has the same functionality as the proposed design in FIX mode. The other one is a floating-point multiply-accumulate unit which supports 16-bit half-precision multiplication accumulating to 32-bit single-precision accumulator. This has the same functionality as the proposed design in FLP mode. The comparison of these two designs and our proposed design is shown in Table 4.2. The proposed design has similar worst case delay as the floating-point multiply-accumulate. This is because the critical path is in the multiplier. The support of fixed-point multiplication does not introduce much delay overhead. In terms of area, the proposed design has 4.6% larger area. This is due to the multiplexers

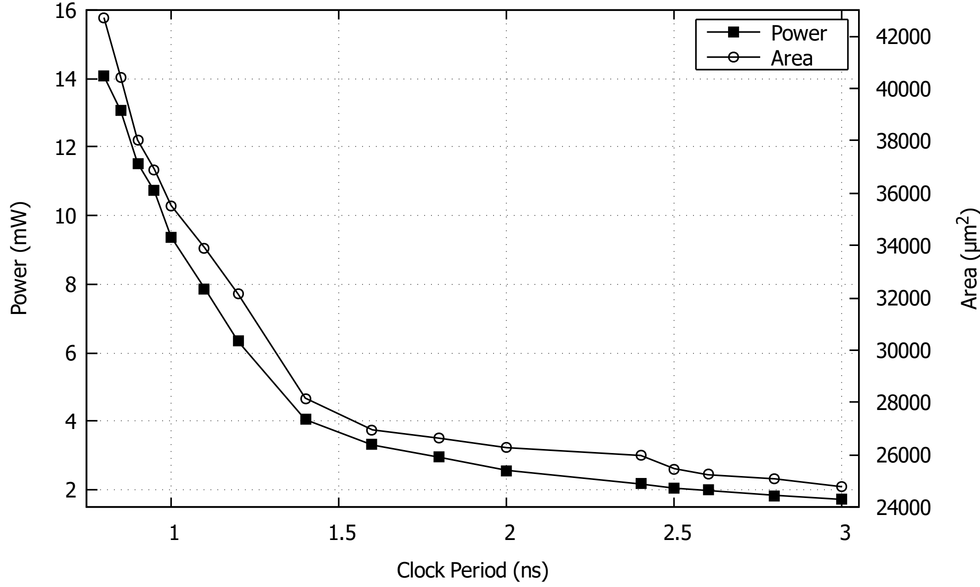


Figure 4.4: Area-Delay and Power-Delay curve of the proposed design

added to support fixed-point operations. The proposed design also consumes 3.5% more power. These overhead are negligible. However, compared to the FIX+FLP design, which is the pure combination of these two single-mode units, the proposed unit has 21% smaller area and 30.4% smaller power consumption while achieving the same functionality.

4.4 Summary

In this chapter, a fixed/floating-point merged mixed-precision multiply-accumulate unit is designed for deep learning processors. The proposed design supports 16-bit half-precision multiplication and accumulating the product to a 32-bit floating-point accumulator. This floating-point operation can be used for deep learning training. In addition, the proposed design also supports two parallel 8-bit fixed-point multiplications and accumulating the products to a 32-bit fixed-point accumulator. This mode can be used in deep learning inference. The Karatsuba algorithm is used to divide the mantissa multiplier of half-precision multiplication. The two 8-bit multipliers generated can be used to support two parallel 8-bit fixed-point multiplications. The higher precision used in accumulation is to avoid data loss during accumulation. The proposed unit can be used in deep learning processors to enable training and high-throughput inference.

Chapter 5

Flexible Multiple-Precision Multiply Accumulate Unit[1]

This chapter presents the design of a flexible multiple-precision MAC unit for deep learning training and inference operations. The proposed architecture is designed based on the characteristics of deep learning computing. By using the proposed unit, a higher throughput and improved energy efficiency can be achieved when processing deep learning applications. Section 5.1 presents the motivations to design such a MAC unit. The supported numerical formats of the proposed MAC unit are discussed in Section 5.2. Section 5.3 presents the details of the proposed design. The synthesis results of the proposed design and the comparison with other MAC units are presented in Section 5.4. A case study is presented in Section 5.5 to show the advantage of the proposed design. Section 5.6 presents the discussions of some questions related to the proposed design. Section 5.7 concludes this chapter.

5.1 Introduction

Deep learning [54] has achieved great success in recent years. In many applications, deep learning can achieve a performance that is near to or even better than human level. Although deep learning is powerful, the cost to implement a deep learning model is very expensive [5], especially in terms of computational intensity. In recent years, many research works have

[1]The content of this chapter is originally published in IEEE Transactions on Computers [55]. The manuscript has been reformatted for inclusion in this thesis.

Hao Zhang (HZ), Dongdong Chen (DC) and Seok-Bum Ko (SK) designed the study. HZ developed the methodology, optimized the architecture, developed the HDL code of the architecture, and performed logic synthesis and results analysis. DC gave suggestions on improving the architecture and analyzing the results. HZ prepared the manuscript with contributions from DC and SK to the manuscript structure, readability and analysis and discussion of the results.

been done on efficient implementation of deep learning algorithms on hardware. Among these works, the numerical format required by deep learning training and inference are extensively investigated.

Currently, most of the deep learning training jobs are done using GPUs with 32-bit single-precision floating-point [8] operations. However, the data-path of single-precision floating-point units is complex and the hardware cost of implementing single-precision units are expensive. These lead to a high energy consumption and a large latency when implementing deep neural networks in customized hardware. In order to reduce the hardware cost, in recent years, many research works [9, 11, 13, 23] are focused on reducing the numerical precision required by deep neural network training. In [9], the authors proposed to train deep neural network with 16-bit half-precision number format. The 12-bit floating-point format is utilized in training in [23]. In [11] and [13], 8-bit floating-point format is used. The standard single-precision, double-precision, and quadruple-precision are well supported in many arithmetic unit designs in the literature [27, 37, 56, 57] and in many commercial products [42, 43]. However, there are not many works discussing the support of 16-bit or even lower precision floating-point operations. Due to the small bit-width of 16-bit or even lower precision floating-point formats, the implementation of arithmetic units based on half-precision and even lower precision floating-point formats is much more efficient than other floating-point formats [40]. As the interest of using low precision floating-point formats in deep neural networks rises, the support of low precision in arithmetic units is required to be investigated.

Using low precision floating-point during the deep neural network training is expected to reduce the energy consumption compared to using the standard single-precision floating-point [58]. However, further energy reduction can be achieved when each operation step can use its minimum required precision instead of being forced to use a uniform precision for all steps. This idea is feasible for deep neural network implementation since the minimum required numerical precision to maintain accuracy is different for different deep neural networks [13] [15] [14]. Furthermore, even within the same deep neural network, different layers have different tolerance to the reduced numerical precision [13] [14]. Therefore, a flexible precision arithmetic unit is desired to further reduce energy consumption and to improve the performance of deep neural network operation.

For deep neural network computing, the dynamic range of a floating-point format (bit-width of exponent, BW_e) is more important than the precision (bit-width of mantissa, BW_m) [15]. With enough BW_e, the neural network can achieve satisfying accuracy and the accuracy of network does not have significant difference with various BW_m. The BFloat16 format introduced in Tensorflow [6] directly truncates the mantissa of single-precision numbers from 23-bit to 7-bit while reserves the 8-bit exponent. However, if the BW_e is not enough, the accuracy of the neural network will have a significant degradation. According to this feature, a flexible precision format can be achieved by a method where the total bit-width of a number is a constant but BW_e and BW_m can be mutually exchanged. In this method, the requirement of the representation range of the numerical format is first met and the remaining bits are allocated to mantissa. When the required data range is large, BW_e can be increased, so that the flexible format can still represent the data properly for deep neural network applications. When data range is small, BW_e can be reduced and thus the flexible format can represent the data more precisely with larger BW_m.

In addition to floating-point numbers, fixed-point numbers are often used for deep neural network inference. The works in [11] and [12] show that inference can be accomplished with 8-bit fixed-point numbers. The 16-bit fixed-point format is used by some deep neural network accelerators [24] [25]. Moreover, for more efficiency, binarized neural network [26] is proposed where neural network parameters are constrained to ± 1. For a versatile deep neural network processor, these fixed-point operations and binary operations are required to be supported.

In the literature, the Flexpoint [59], a software controlled flexible dynamic fixed-point format is proposed. In this format, the shared exponent can be dynamically changed to meet the dynamic range requirement of the deep learning computing. In addition, the flexible floating-point precision method has been applied in a recent work [60]. In [60], a tunable precision floating-point multiplier which supports 5 to 8-bit exponent and 4 to 24-bit mantissa is proposed. Their results show significant improvement in energy consumption. However, for deep learning applications, some improvements can be applied. First, as discussed in [13–15], deep neural networks might not need large BW_m when floating-point format is used. Second, between floating-point format and fixed-point format, the latter one is desired when performing inference. Although recent works [61] [62] show good inference results

with logarithmic number system, conventional arithmetic unit is still more popular in many different hardware designs. Third, parallel operations can be supported to compensate for throughput degradation in lower precision operations of the conventional arithmetic unit. Last but not the least, fused operation units, such as MAC [63] [64] and DOT, are preferred compared to separate multiplier and adder due to smaller area and better accuracy.

With all these requirements in consideration, in this paper, a new flexible multiple-precision multiply-accumulate unit is proposed. The proposed MAC unit supports both floating-point operations (for deep neural network training) and fixed-point operations (for deep neural network inference). For floating-point format, the proposed unit supports one 16-bit MAC operation (FLP16-MAC) or sum of two 8-bit multiplications plus a 16-bit addend (FLP8-DOT2). The bit-width of exponent and mantissa can be mutually exchanged to realize the flexible precision support. For 16-bit floating-point format, up to 8-bit exponents are supported, as 8-bit exponent, the exponent bit-width of standard single-precision format, can already represent nearly all neural network parameters. In 8-bit mode, the dot-product operation $A_1 \times B_1 + A_2 \times B_2 + C$ is supported where products of two parallel 8-bit multiplications can be added together and then accumulated to a 16-bit floating-point addend C. The BW_e and BW_m of the 16-bit addend can also be flexibly defined. For fixed-point format, the proposed unit supports one 16-bit MAC operation (FIX16-MAC), or sum of two 8-bit multiplications plus a 16-bit addend (FIX8-DOT2), or sum of four 4-bit multiplications plus a 16-bit addend (FIX4-DOT4). For fixed-point format, the location of the radix-point can be flexibly defined. In other word, the bit-width of integer part and fractional part can be exchanged. Furthermore, binary neural network operations are supported. The major contributions of this paper are summarized as follows:

- Propose the architecture of flexible multiple-precision multiply-accumulate unit.
  - Propose the method to correctly extract each component of all supported precisions.
  - Propose the method to correctly align the operands under all supported precisions for addition.
  - Propose the method to correctly round the results of all supported precisions,

Table 5.1: Supported formats of the proposed MAC unit

Operations	MAC Operands				
	A and B			C	
	total	exponent	parallelism	total	exponent
FLP8-DOT2 ($\sum_{i=1}^{2} A_iB_i + C$)	8-bit	1∼6-bit	2	16-bit	1∼8-bit
FLP16-MAC ($AB + C$)	16-bit	1∼8-bit	1	16-bit	1∼8-bit
	total	fraction	parallelism	total	fraction
FIX4-DOT4 ($\sum_{i=1}^{4} A_iB_i + C$)	4-bit	0∼4-bit	4	16-bit	0∼15-bit
FIX8-DOT2 ($\sum_{i=1}^{2} A_iB_i + C$)	8-bit	0∼7-bit	2	16-bit	0∼15-bit
FIX16-MAC ($AB + C$)	16-bit	0∼15-bit	1	16-bit	0∼15-bit

where roundTiesToEven [8] is supported.

  – Propose the method to support subnormal numbers of all supported precisions.

- A comprehensive analysis of the implementation results is performed. Compared to the standard floating-point unit, the proposed unit supports many more functions with only minor resource overhead.

- A case study of a simplified deep neural network application is performed with the proposed multiply-accumulate unit to show the power efficiency of the proposed unit.

- The proposed multiply-accumulate unit can be used in deep learning processors in datacenters or used as an neural network intellectual property (IP) core for FPGA devices.

5.2 Supported Numerical Formats

5.2.1 Numerical Format

Standard floating-point formats, including half-precision, single-precision, double-precision, and quadruple-precision, are defined in IEEE754-2008 standard [8]. These formats are composed of 1-bit sign (S), m-bit exponent (E), and n-bit mantissa (M). For half, single, double, and quadruple precisions, m (n) equals to 5 (10), 8 (23), 11 (52), 15 (112), respectively. There is always an implicit bit im for the mantissa part. For normal numbers, $im = 1$. For zero and subnormal numbers, $im = 0$. The numerical value the IEEE 754 format represents is:

$$f_p = (-1)^S \times (im + 2^{-n} \times M) \times 2^{E-bias} \tag{5.1}$$

where $bias = 2^{m-1} - 1$.

In the proposed MAC design, although the bit-width of exponent and mantissa are flexible, all the supported floating-point formats follow the IEEE 754 rule where there is always an implicit bit and the exponent part is biased.

When the exponent of a normalized number (with implicit bit equal to 1) is smaller than the minimum exponent of the corresponding format, the mantissa needs to be right shifted to bring the exponent back to the allowed range. If the difference between the exponent and the minimum exponent is smaller than the bit-width of the mantissa, this number is still representable and is called subnormal number. Otherwise, when using the IEEE 754-2008 default roundTiesToEven rounding mode, if the number is too small to be represented after rounding, it will be flushed to zero.

Fixed-point format is less complicated compared to the floating-point format. In the fixed-point format, numbers are encoded in two's complement format. They have p-bit integer and q-bit fraction. In the proposed design, p and q can be flexibly exchanged for different choices.

The numerical formats of each operand supported by the proposed multiply-accumulate unit in each operational mode are summarized in Table 5.1.

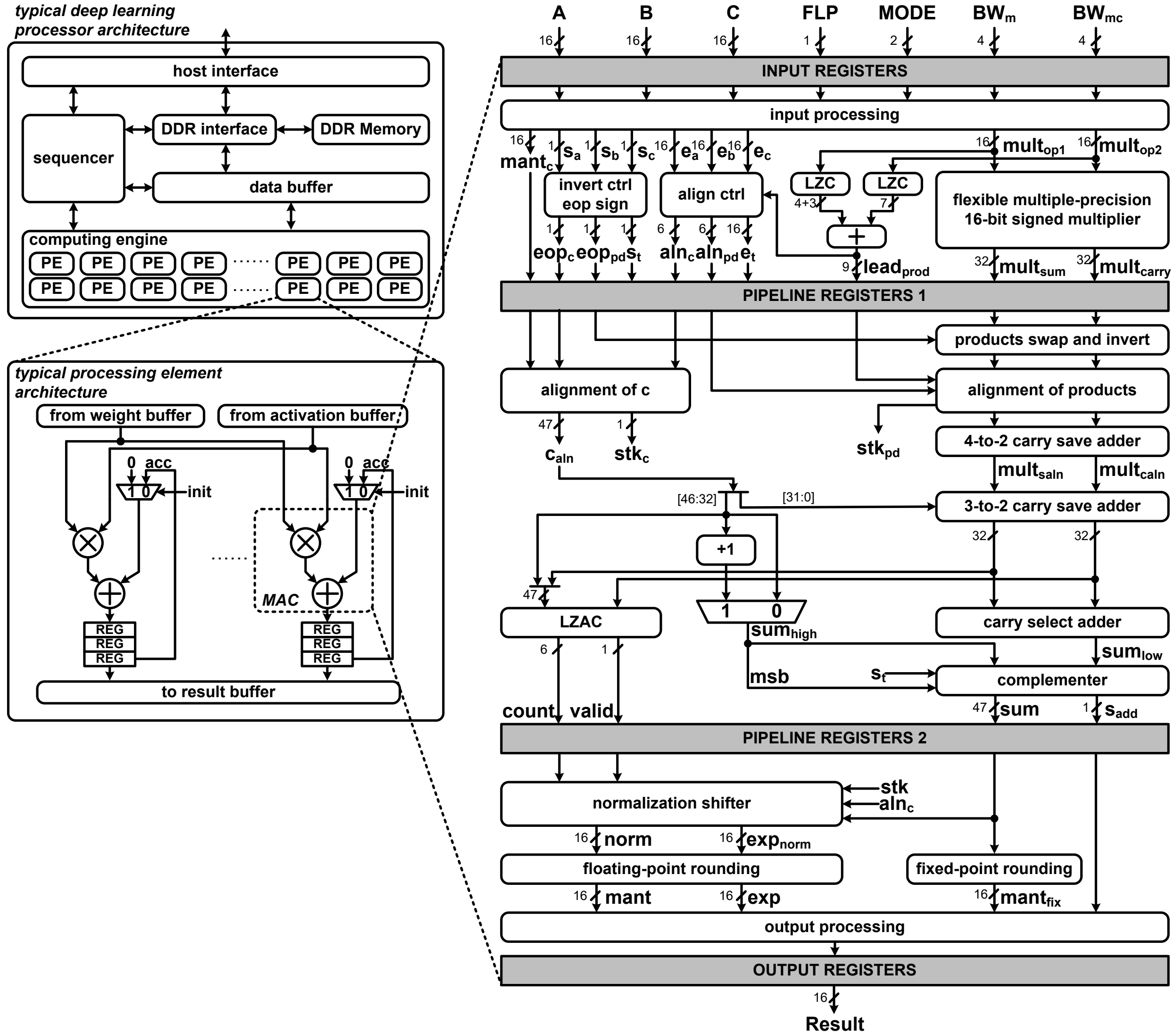


Figure 5.1: Datapath of the proposed flexible multiple-precision multiply-accumulate unit and its usage in deep learning processor

5.3 The Proposed Design

A typical deep learning processor architecture [65–67] contains host interface, DDR memory, on-chip buffer, sequencer, and processing elements (PE), as shown in Figure 5.1. The host interface is used to communicate with the host processors. It receives instructions and input data from host and sends results back to host. The sequencer receives the instructions and coordinates the operations of all other components. The DDR memory is used to save the data received from host and the processing results from PEs. The data to be processed in

the current operation is buffered in on-chip buffer. All the computations are performed in PEs.

Each PE unit contains many multiply-accumulate units and registers to perform dot-product operations or matrix multiplication operations for convolution layer and fully connected layer computations. Two multiplier inputs of the MAC unit are from activation buffer and weight buffer, respectively. The addend input is from the accumulation registers. At the initial cycle of a new operation, the addend is set to zero.

The core of the PE unit is the MAC unit. The functionality of the MAC unit can determine the functionality of the PE unit. On one hand, if the MAC unit only supports one precision format, then all the computations are forced to use the same precision format. Although the operations of other precisions can be achieved by software implementation, however, multiple iterations are usually required for that kind of operations and thus the performance will be degraded. On the other hand, if the MAC unit is versatile, then operations of various precisions can be supported in hardware which will benefit the performance. In addition, each application can choose to use their minimum required precisions so that the energy consumption will be reduced compared to the case of using one precision for all applications. In some cases, parallel operations can help improve the throughput and power efficiency. Based on these considerations, this paper aims to propose a MAC architecture with more functionality. It is designed to support various precisions at runtime. To make the proposed MAC architecture efficient, resource sharing among different precisions are extensively investigated to maintain the area overhead as small as possible compared to a single mode MAC unit.

The datapath of the proposed MAC unit is shown in Figure 5.1. The whole design is divided into three pipeline stages. The first pipeline stage contains the input processing module, where each component of the operands are extracted, the flexible mantissa multiplier, and the alignment and invert control modules. The second pipeline stage contains the alignment shifter for input C and products and then the carry select adder and in parallel the LZAC logic. The third pipeline stage contains normalization shifter and rounding modules.

The pipeline allocation of the proposed design is different from the typical MAC or FMA design, as shown in Figure 5.2, where the alignment of C is usually running in parallel

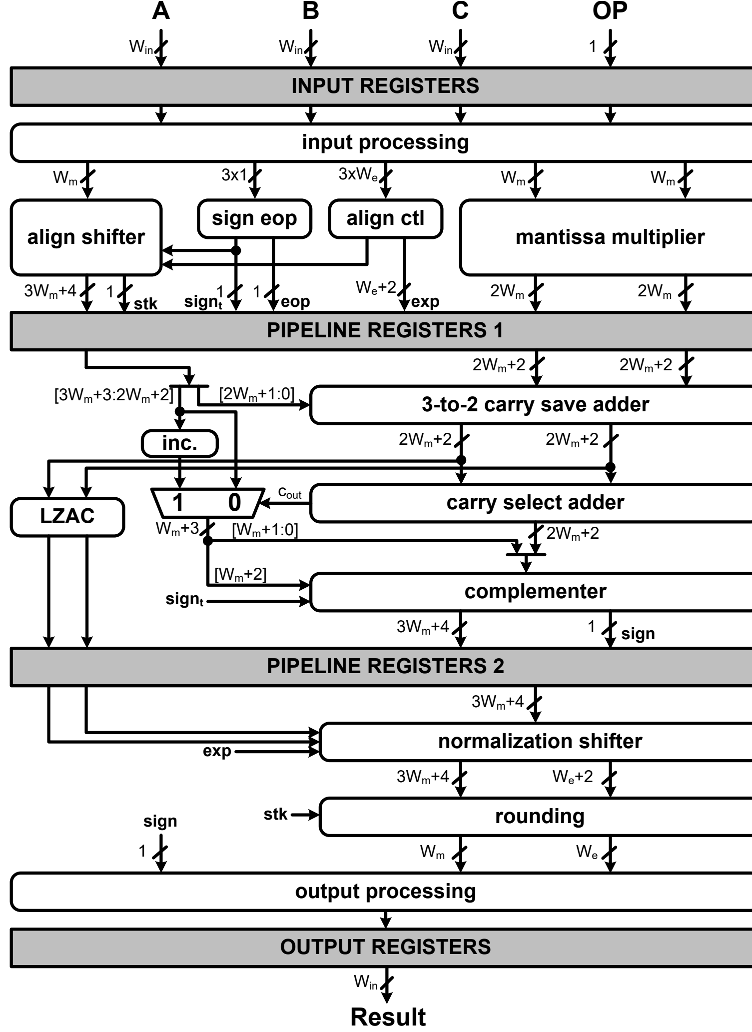


Figure 5.2: Datapath of conventional multiply-accumulate unit based on standard floating-point format

with the mantissa multiplier in the same pipeline stage. In the proposed design, due to the support of flexible precision, the alignment control module will take larger delay than that of a standard design. In this case, if alignment shifter of C still runs in the same pipeline stage, although the delay of an alignment shifter is small, the critical path delay of the first pipeline stage will be significantly larger than the following two pipeline stages. This can lead to an imbalanced pipeline allocation which may increase the total latency of a single operation. Through our preliminary experiment, we found that the second pipeline stage has smaller delay than the other two. In addition, the alignment shifter for the product has to be put in the second pipeline stage (products are not available until the multiplication finishes).

Therefore, we move the alignment shifter of C to the second pipeline stage. This makes the three pipeline stages balanced. This can be verified through the synthesis results shown in Table 5.2.

There are seven input signals to the proposed MAC unit. Three operands, A, B, and C, are of 16-bit for each. The 1-bit FLP is used to control weather the proposed MAC unit works in floating-point mode ($FLP = 1$) or fixed-point mode ($FLP = 0$). The 2-bit $MODE$ signal controls the precision mode of the proposed unit, where $MODE = 00$ represents binary mode, $MODE = 01$ represents 4-bit mode, $MODE = 10$ represents 8-bit mode, and $MODE = 11$ represents 16-bit mode. The 4-bit BW_m represents the bit-width of mantissa (in floating-point mode) or fraction (in fixed-point mode) in the operands A or B. The other 4-bit signal BW_{mc} has similar functionality but is used for operand C.

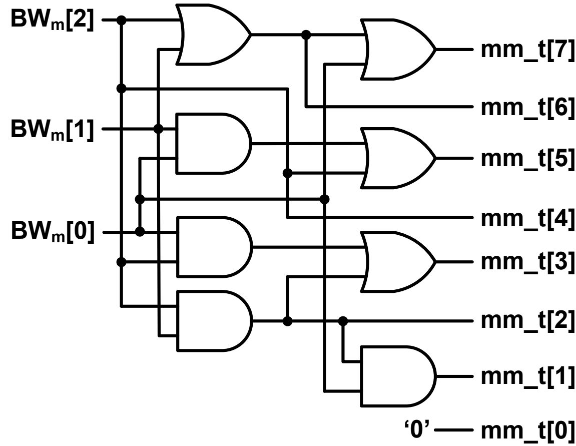
For the following subsections, the design details of each of the modules will be discussed. Emphasis will be put on design issues of flexible precision arithmetic unit, which include (1) operands extraction; (2) flexible multiplier; (3) flexible alignment control; (4) flexible rounding scheme; and (5) flexible subnormal handling.

5.3.1 Input Processing

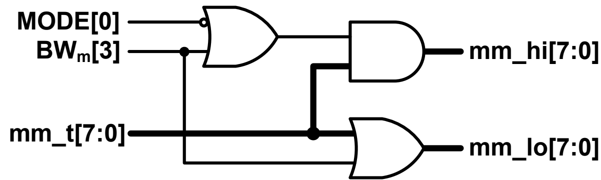
For fixed-point operands, although the bit-width of integer part and fraction part might vary, these two parts can be used as a whole and directly sent to the multiplier. For floating-point operands, however, the exponent part and mantissa part must be correctly divided. In addition, the implicit bit must be correctly prefixed to the mantissa.

To extract mantissa, a mask signal mm (mantissa mask) is generated from the input BW_m and applied to the input operands. For 8-bit floating-point, the bit-width of mantissa may vary from 1 to 6. Therefore, the least significant 3-bit of BW_m are used to represent the number of mantissa for two sets of 8-bit operands. Two sets of 8-bit masks, mm_hi and mm_lo will be generated for two sets of 8-bit operands.

In 16-bit floating-point mode, the bit-width of mantissa will be between 7 and 14, the whole 4-bit BW_m are required to represent the number of mantissa in operands. The two masks, mm_hi and mm_lo are reused in 16-bit mode and they are combined to represent the mantissa mask for 16-bit operands. Still, the lower 3-bit of BW_m are used to generate



(a) Generation of 8-bit mask

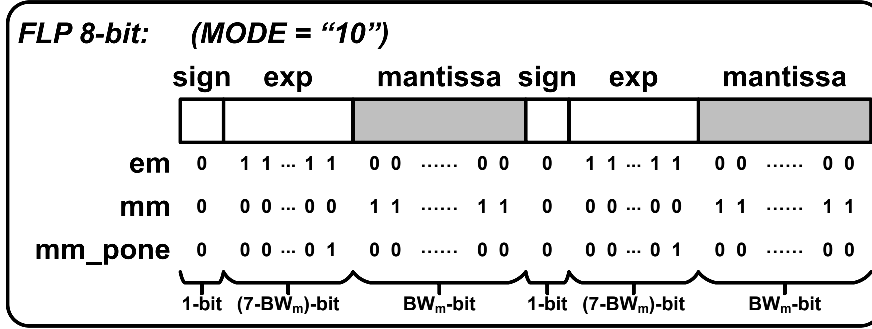


(b) Correction for 16-bit mode

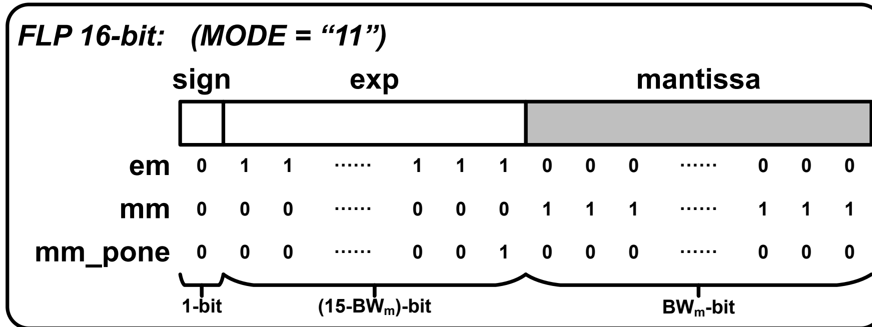
Figure 5.3: Circuit to generate the mantissa mask *mm_hi* and *mm_lo*

mm_hi and *mm_lo*, however, for 16-bit operations, one more step is required. On one hand, if $BW_m[3] = 0$, then mantissa bits are all in the least significant 8-bit of the operand, and thus *mm_hi* is set to all zeros while *mm_lo* is not changed. On the other hand, if $BW_m[3] = 1$, then the bit-width of the mantissa is no less than 8-bit, and thus *mm_lo* is set to all ones, representing the whole least significant 8-bit contains mantissa bits, while *mm_hi* is not changed.

The circuit diagram of this process is shown in Figure 5.3. *mm_t* is the 8-bit mask generated using the least significant 3-bit of BW_m. This *mm_t* is further processed by the circuit shown in Figure 5.3(b) to generate the mantissa masks *mm_hi* and *mm_lo* for 8-bit mode and 16-bit mode. The mantissa mask is only used in floating-point mode. To distinguish 8-bit and 16-bit floating-point mode, the least significant bit (LSB) of $MODE$ can be used. In 8-bit floating-point mode, $MODE[0] = 0$ and $BW_m[3] = 0$, therefore, *mm_hi* and *mm_lo* are the same as *mm_t*. In 16-bit floating-point mode, $MODE[0] = 1$. If $BW_m[3] = 0$, *mm_lo* will be the same as *mm_t* and *mm_hi* will be set to all zeros. If $BW_m[3] = 1$, *mm_lo* will be set to all ones and *mm_hi* will be the same as *mm_t*.



(a) Floating-point 8-bit mode



(b) Floating-point 16-bit mode

Figure 5.4: Format of mantissa mask *mm*, exponent mask *em*, and implicit bit mask *mm_pone*

The exponent mask *em* can be generated by inverting the mantissa mask and then set the MSB of the generated vector to zero because the MSB is always sign bit. For 8-bit mode, bit 7 which is the sign bit of lower 8-bit number is also set to zero.

These two masks *em* and *mm* will be applied to input operands by performing an AND operation to extract exponent and mantissa. To obtain the exponent value, the extracted exponent needs to be right shifted with an amount of BW_m. The extracted exponent will be used to determine the implicit bit. To add implicit bit to the mantissa, the *mm* will be used again. By adding 1 to *mm* (generating *mm_pone*), there will be a 1 generated in the position of implicit bit and leaving all other position as zeros. Then if the implicit bit is 1, *mm_pone* and generated mantissa vector can be ORed together to add implicit bit into mantissa. The format of these three masks are graphically shown in Figure 5.4. For operand C, the signal BW_{mc} is used to perform exponent and mantissa extraction and the process is the same as A and B. The overall diagram of input processing for operand A is shown in Figure 5.5. The

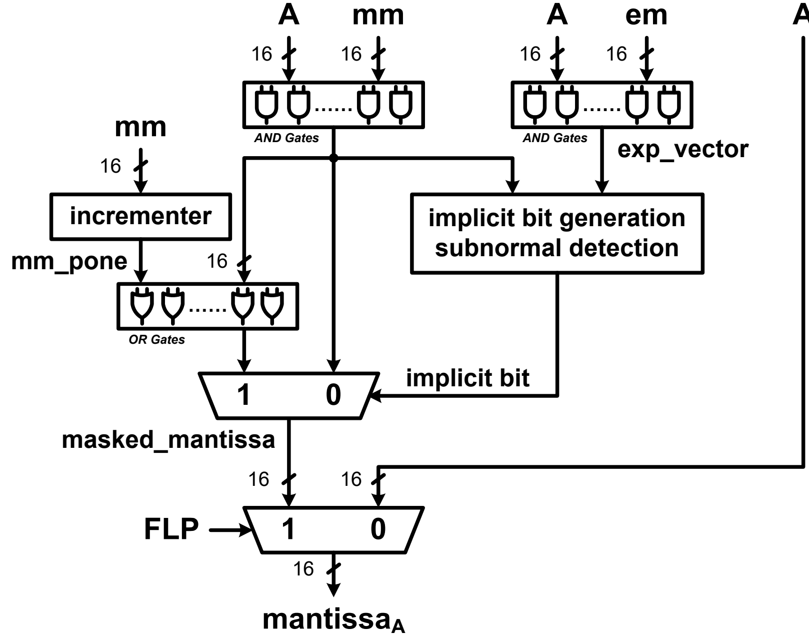


Figure 5.5: Diagram of input processing for A operand

same circuit can be applied to operands B and C.

5.3.2 Flexible Multiple-Precision Multiplier

For fixed-point operands, a signed multiplier is required. For floating-point operands, an unsigned multiplier is required. However, as the bit-width of floating-point mantissa is always smaller than the bit-width of fixed-point number (there is always at least 1-bit exponent for floating-point format), the mantissa can always be sign extended with zeros and converted to a signed positive number. Therefore, in a uniformed multiplier design, a signed multiplier is implemented.

In the proposed design, in order to reduce the cost of 16-bit multiplication, the radix-4 modified Booth multiplier [49] is applied. The multiplicand, multiplier, and the generated partial product (pp) array of the proposed flexible multiplier are shown in Figure 5.6. The partial products are generated with the method proposed in [68]. For each precision mode, the generated partial product array is shown in Figure 5.7. In Figure 5.7, S represents the sign of the corresponding partial product and E is the extended bit to the partial product. According to [68], $E = 1$ when the multiplicand and the partial product have the same sign

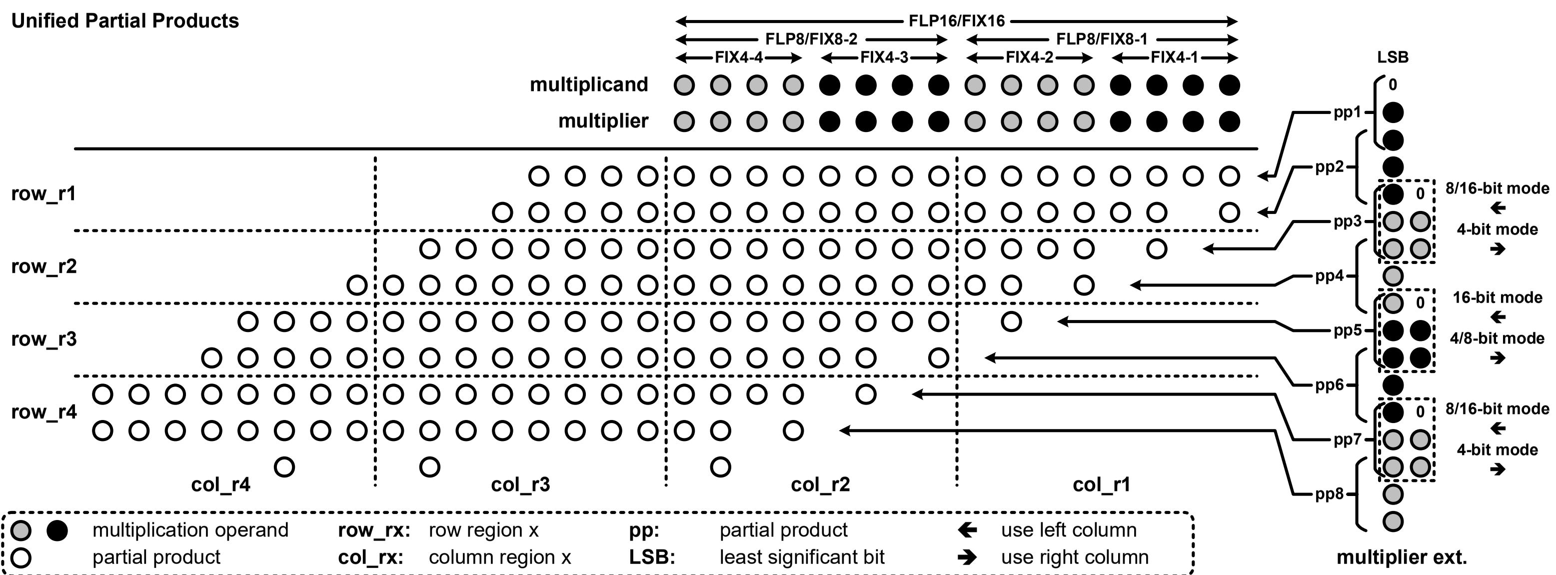
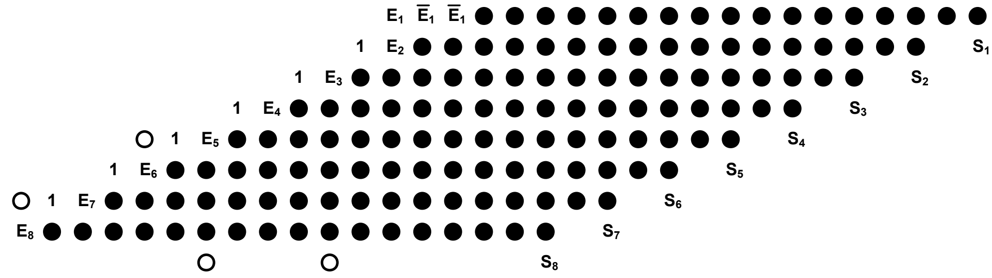
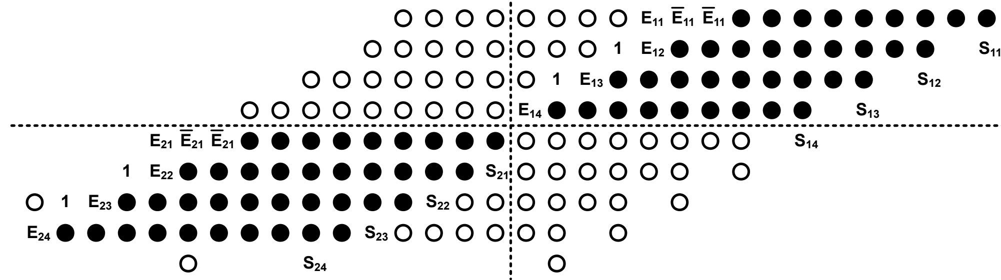


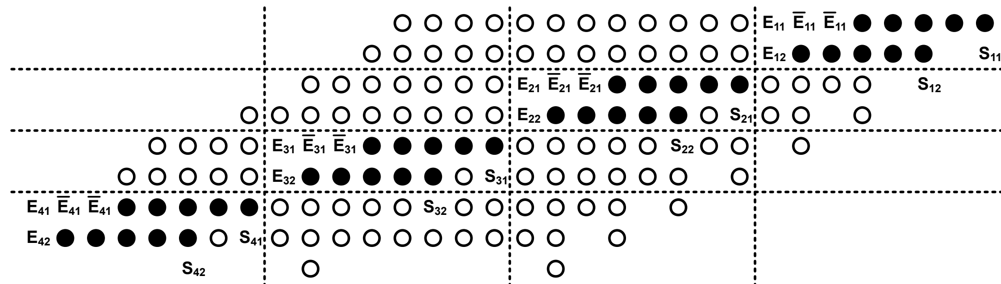
Figure 5.6: Partial product generation and partial product array of the proposed flexible multiple-precision multiplier



(a) 16-bit mode



(b) 8-bit mode



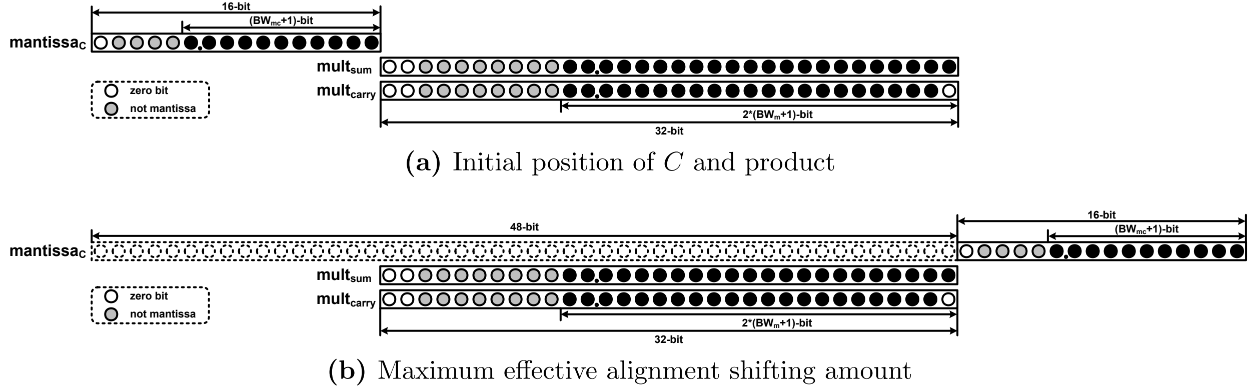
(c) 4-bit mode

Figure 5.7: Partial product array of each precision mode (white dots represent the bits not used; black dots represent partial products)

or partial product is $+0$. Otherwise, when the multiplicand and the partial product have opposite signs or partial product is -0, $E = 0$.

In 16-bit mode, the whole multiplicand and the multiplier are used to generate partial products. According to the radix-4 Booth multiplier algorithm, the multiplier is always padded with 1-bit zero after the LSB. Then the resulting 17-bit vector is divided into eight 3-bit groups as shown in Figure 5.6. Each bit group is used to generate one partial product. The generated partial product array in 16-bit mode is shown in Figure 5.7(a).

In order to support parallel multiplications in 8-bit and 4-bit mode, the partial product array is divided into multiple regions as shown in Figure 5.6. In 8-bit mode, two parallel



(a) Initial position of C and product

(b) Maximum effective alignment shifting amount

Figure 5.8: Alignment shifting of C with product in FLP16 mode

multiplications, MUL8-1 and MUL8-2, are supported. The partial products of MUL8-1 are in row_r1 and row_r2 in row direction, and col_r1 and col_r2 in column direction, as shown in Figure 5.7(b). Similarly, the partial products of MUL8-2 are in row_r3 and row_r4, and col_r3 and col_r4. To generate the four partial products in row_r1 and row_r2, the most significant 8-bit of the multiplicand are set to zeros and only the least significant 8-bit are used. When generating partial products in row_r3 and row_r4, the least significant 8-bit of the multiplicand are set to zeros and only the most significant 8-bit are used. In addition, as MUL8-2 is an independent multiplication, when generating $pp5$, the LSB of the corresponding multiplier group is set to 0 instead of using the 8th bit of the multiplier. Similarly, in 4-bit mode, as shown in Figure 5.7(c), only $FIX4-1$, $FIX4-2$, $FIX4-3$, or $FIX4-4$ is used to generate the partial products in row_r1, row_r2, row_r3, or row_r4, respectively. In addition, when generating $pp3$, $pp5$, and $pp7$, the LSBs of the corresponding multiplier groups are set to zero.

To ensure correct multiplication results for low precision modes, carry propagation during partial products accumulation is managed. In 8-bit mode, carry is not allowed to be propagated through the second vertical dash line in Figure 5.6 (or the vertical dash line in Figure 5.7(b)), and in 4-bit mode, carry is not allowed to be propagated through the three vertical dash lines in Figure 5.6 or Figure 5.7(c).

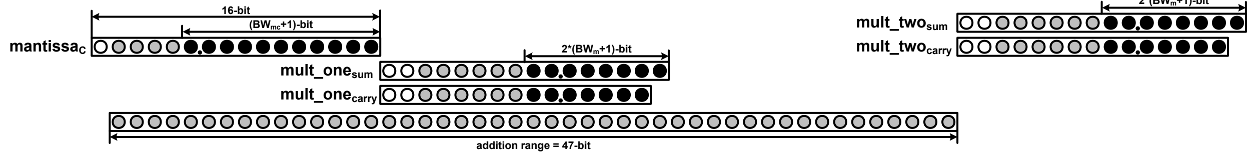


Figure 5.9: Alignment shift of C and two products in 8-bit floating-point mode

5.3.3 Alignment Control and Shifter

In 16-bit floating-point mode, the general alignment shifting method used in many previous FMA designs [28] is applied. The initial position of the processed mantissa of C operand and product $A \times B$ is shown in Figure 5.8(a). The mantissa of C is placed 2-bit to the left of the carry save format product. In 16-bit floating-point mode, BW_m can be at most 14-bit. Therefore, the 2-bit zeros gap are already included in $mult_{sum}$ and $mult_{carry}$. The mantissa of C is then right shifted according to the difference of exponents and the bit-width of mantissa. The shifting amount can be determined by equation (5.2):

$$shift_c = 32 + BW_{mc} - 2 \times BW_m - d \tag{5.2}$$

where $d = e_c - (e_a + e_b)$ is the exponent difference among the three input operands and e_a, e_b, and e_c are the exponent value of operand A, B, and C. The maximum effective shifting amount, $shift_{cmax}$ = 48-bit, occurs when all bits of the C mantissa are shifted to the right of the product, which is shown in Figure 5.8(b).

In the initial alignment position, the 2-bit zeros gap between the product and the addend is to ensure the alignment shifting of the addend is a single direction shifting so that the shifter design can be simplified. In the proposed design, when the mantissa bit-width BW_m becomes smaller, the gap between the product and the addend becomes larger. However, in those cases, the alignment shifter is still a single direction shifter. Therefore, in the proposed design, in order to simplify the shifter design, we do not force the gap to be 2-bit. Instead, a unified initial position of the addend and the product is used, as shown in Figure 5.8(a). The alignment shifting amount can always be calculated with equation (5.2).

For 8-bit floating-point mode, in addition to the alignment of C, the two products generated also need to be aligned. To simplify the shifting circuit, in the proposed design, the two products are compared first to determine the product with larger exponent. The product

with larger exponent is used as the anchored product. Both C and the other product will be aligned to the larger product. The initial position of the larger product $mult_one_{sum}$ and $mult_one_{carry}$, the smaller product $mult_two_{sum}$ and $mult_two_{carry}$, and mantissa of C is shown in Figure 5.9. The smaller product is placed to the right of the LSB of the addition range. This is because in subnormal cases, it needs to be left renormalized first and then right aligned. In the proposed design, the renormalization amount and the alignment amount will be considered at the same time, where the effective renormalization amount will be generated and the smaller product only needs a left shift. This can simplify the shifter design. The addition range shown in Figure 5.9 is 47-bit. The MSB position of the mantissa of C before alignment shifting is not included in the adder. This is because in floating-point mode, the mantissa of C could only occupy at most 15-bit. In fixed-point mode, 47-bit adder is large enough for 32-bit product accumulation.

In order to support subnormal numbers, the maximum alignment shifting happens when the MSB of the addend is shifted 2-bit to the right of the LSB of the product [69]. By extending the analysis in [69], when the multiplier operands and the addend have different mantissa bit-width, the gap to the right of the LSB of the product can be calculated with $BW_{mc} - BW_m + 2$. For 16-bit floating-point mode, when both the addend and the multiplier operands have 14-bit mantissa, as the exponent is only 1-bit, the addend C can never reach the maximum alignment shifting position as shown in Figure 5.8(b). For other mantissa bit-width, as the range of the addend is usually chosen to be equal to or larger than the range of the multiplier operands, the bit-width of the addend is smaller than the multiplier operand. Therefore, at most 2-bit gap to the right of the product LSB is enough to handle subnormal numbers. In this case, as the mantissa bit-width is smaller, there is more than 1-bit zero appear in the 16-bit addend. There is no need to add extra zero bit gap. For 8-bit floating-point mode, BW_{mc} and BW_m can be at most 14-bit and 6-bit, respectively. Therefore, at most 10-bit gap is required. As shown in Figure 5.9, at least 15-bit gap is available. Considering all these cases, there is no need to add extra zero gap to the right of the product LSB.

For 16-bit fixed-point mode, although the bit-width of fraction part can be flexibly changed, within a specific mode, the alignment shifting amount is a constant value.Therefore,

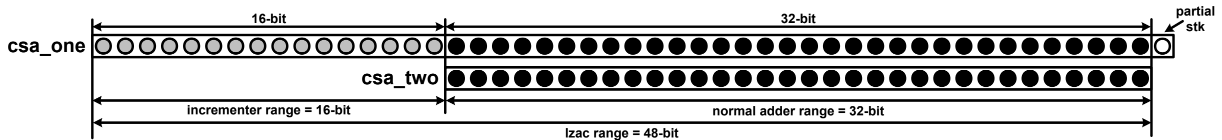


Figure 5.10: Addition arrangement and the leading zero anticipator and counting (LZAC) range of the proposed unit

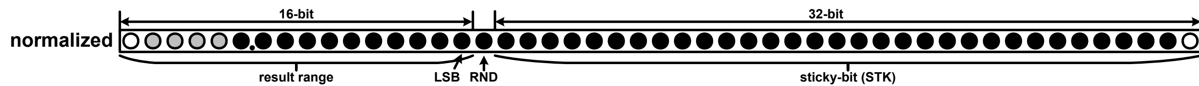


Figure 5.11: Rounding method for floating-point modes

for 16-bit mode, the alignment shifting can reuse the alignment shifter designed for floating-point mode by setting the shifting amount to a constant. The constant shifting amount for C, $shift_{const_c}$, can be determined by equation (5.3):

$$shift_{const_c} = 32 + BW_{mc} - 2 \times BW_m \tag{5.3}$$

where the BW_m and BW_{mc} represent the bit-width of fraction in fixed-point mode. For 8-bit and 4-bit fixed-point mode, there is no need to used alignment shifter. In these two modes, BW_{mc} can be set to be equal to $2 \times BW_m$ (If $BW_{mc} > 2 \times BW_m$, the extra fraction bits can never be used. If $BW_{mc} < 2 \times BW_m$, some data bits will be lost.). Before performing addition, the LSB of the addend can be directly put to the LSB position of the product.

5.3.4 Addition

The aligned C and products will be accumulated with carry save adders. The generated carry-save format vectors will be added using a carry propagate adder. For the proposed design, a total of 47-bit addition is provided. The lower 32-bit addition will be implemented with a carry select adder and the higher 15-bit will be implemented using an incrementer, as shown in Figure 5.10. For the higher 15-bit addition, both the results of $carry_in = 0$ and $carry_in = 1$ are generated. The output carry from the lower order adder will be used to select these two results. As in 4-bit mode or 8-bit mode, dot-product operation is performed. Therefore, for the adder there is no need to generate parallel separate results, instead only one single addition result is generated.

5.3.5 Leading Zero Anticipator and Counting

In parallel to the adder, the LZAC logic is implemented. As both positive result and negative result can be generated, both leading zeros and leading ones can occur. Therefore, the general case indicator presented in [50] is used.

The counting result of a normal LZA might have 1-bit error. This error can be easily corrected in the normalization shifter for a standard precision design. However, as the proposed unit supports flexible precision, it is complex to find the MSB position of the result and then detect the MSB value to determine whether a correction should be performed. Therefore, in order to reduce the cost of later normalization stage, the exact LZA unit in [70] is used in the proposed design.

As shown in Figure 5.10, the LZAC is applied to whole 48-bit range of the adder. When BW_{mc} is small, there might exist some leading zeros that are not occupied by the mantissa. Therefore this bit count should be subtracted from the LZAC count when determining the normalization shifting amount. This can be achieved by equation 5.4:

$$norm_shift = lzac_count - (15 - BW_{mc}) \tag{5.4}$$

5.3.6 Normalization Shifting

Normalization shifting is performed with a 6-level dynamic shifter to shift the 48-bit vector. As the LZA can generate the exact leading zero count, there is no need to add one more stage to compensate the LZA error. The normalization shifter will bring the MSB of the addition result back to the left of the radix point position, as shown in Figure 5.11.

After alignment shifting, the base exponent becomes $e_{base} = e_c + shift_c$. If $e_{base} - norm_shift > e_{min}$, then result is still a normal number and the normalization shifting amount is $norm_shift$ and the result exponent is set to $e_{base} - norm_shift$. Otherwise, the result will become a subnormal number. In this case, normalization shift amount is $e_{base} - e_{min}$ and the result exponent is set to e_{min}.

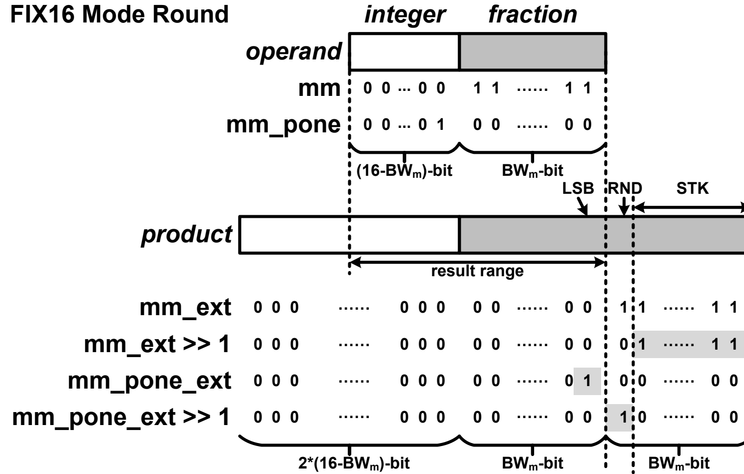


Figure 5.12: Rounding method for 16-bit fixed-point mode

5.3.7 Rounding

For floating-point mode, roundTiesToEven, which is the default rounding mode in IEEE 754-2008 [8], is implemented. After normalization, the result will be brought back to a position shown in Figure 5.11. Therefore, the rounding position for floating-point is fixed, which is the LSB position of the C operand. Therefore, the bits required to perform rounding, the LSB, the rounding bit, and the sticky bit, can be easily generated.

For 16-bit fixed-point operations, the product has larger fractional bit-width than the addend. Therefore, rounding is also required. Unlike the floating-point modes where the rounding position is fixed, in fixed-point case, the rounding position depends on the number of fraction bits in the input operands. To find the correct rounding position and generate rounding bits, the masks, mm and mm_pone, generated for input processing will be used again. The process of using mm and mm_pone to find three rounding bits, the LSB, the round bit (RND), and the sticky bit (STK), is shown in Figure 5.12. Both mm and mm_pone are extended with zeros to make 16-bit mm_ext and mm_pone_ext. Then these two extended vectors are right shifted by 1-bit. By ANDing mm_pone_ext with the result, the LSB can be extracted. Similarly, by using the shifted mm_ext and shifted mm_pone_ext, the STK and the RND can be generated as shown in Figure 5.12. By using these three bits, the fixed-point rounding can be performed. The floating-point mode and the fixed-point mode can share the incrementer to perform rounding plus one operation. For the rounding in integer part, a

simple truncation can be performed.

For other lower precision fixed-point operations, as the bit-width of C is always larger than operands in A and B, C can have the same or more fraction bits than the product. Therefore, there is no need to perform rounding for the fraction. Truncation can be performed for the integer.

5.3.8 Output Processing

After processing the sign, exponent, and mantissa separately, they need to be combined again to generate the final results. For fixed-point, a simple truncation can be used to generate the final 16-bit result.

For floating-point, basically two operations are required. As the mantissa is already right aligned to the LSB position, for mantissa part, we only need to remove the implicit bit. To do so, the mantissa mask mm will be used. The mm contains all ones at the position of mantissa (except implicit bit). By ANDing the mm with the result vector, the implicit bit can be removed.

The second operation for floating-point is to shift the exponent back to the correct position. In order to obtain the actual value of exponent to generate the shifting amount, exponent is right shifted with the amount of BW_m or BW_{mc} in alignment control module. At the output processing module, the exponent is shifted back with the amount of BW_{mc}.

Finally, the exponent vector and mantissa vector are ORed together and sign bit is added into the MSB position to form the final floating-point result.

5.4 Results and Analysis

The model of the proposed MAC architecture is implemented with Verilog HDL. Since the proposed design supports non-standard floating-point format, there is no simulation tool that can generate test vectors for all supported precisions. In order to simulate and verify the proposed design, a customized software model written in C language is built. With the designed software model, extensive testing vectors for each of the supported precisions can be generated. These testing vectors are used to simulate the proposed design in Modelsim.

Table 5.2: Synthesis results of each pipeline stage of the proposed design

Pipeline Stage	Delay (ns)	Area (μm^2)	Power (mW)
Multiply and Control	0.21	1177	0.69
Align and Add	0.23	1356	0.60
Normalize and Round	0.19	319	0.18

The proposed design is verified to work properly in all supported modes.

The proposed design is then synthesized in Synopsys Design Compiler using STM-28nm technology with typical case parameters (1.00V and 25℃). The timing and area metrics are generated. For power consumption measurement, the Verilog netlist generated by synthesis is simulated with the testing vectors again to obtain a signal activity file. Then the synthesized netlist and the signal activity file are imported to Synopsys PrimeTime PX for an accurate power estimation.

Each pipeline stage of the proposed design is synthesized first. The delay of each pipeline stage is analyzed and the result is shown in Table 5.2. In the preliminary experiment, the second pipeline stage (only contains carry save adder, carry propagate adder, and LZA) has a delay of 0.14 ns which is smaller than the other two stages. The worst case delay 0.3 ns appears in the first pipeline stage (contains multiplication and alignment shifter). This pipeline allocation is unbalanced. By moving the alignment shifter to the second pipeline stage, as the results shown in Table 5.2, the pipeline stage allocation is more balanced. The second pipeline stage consumes more area because it contains two alignment shifter in addition to the adder and LZA circuit. Overall, the multiplier still consumes the largest area and power consumption.

The whole design is then synthesized. The synthesis results show that after adding pipeline registers, the worst case delay of the pipelined design is 0.27 ns. In this timing constraint, the proposed design consumes an area of 2943 μm^2 and an average power consumption of 2.25 mW. The proposed design is also synthesized under different timing constraints to find the trade-off between timing and area, and timing and power. The area-delay curve and power-delay curve can be found in Figure 5.13. To obtain the best performance, the design point with the smallest worst case delay (0.27 ns) is chosen and the corresponding

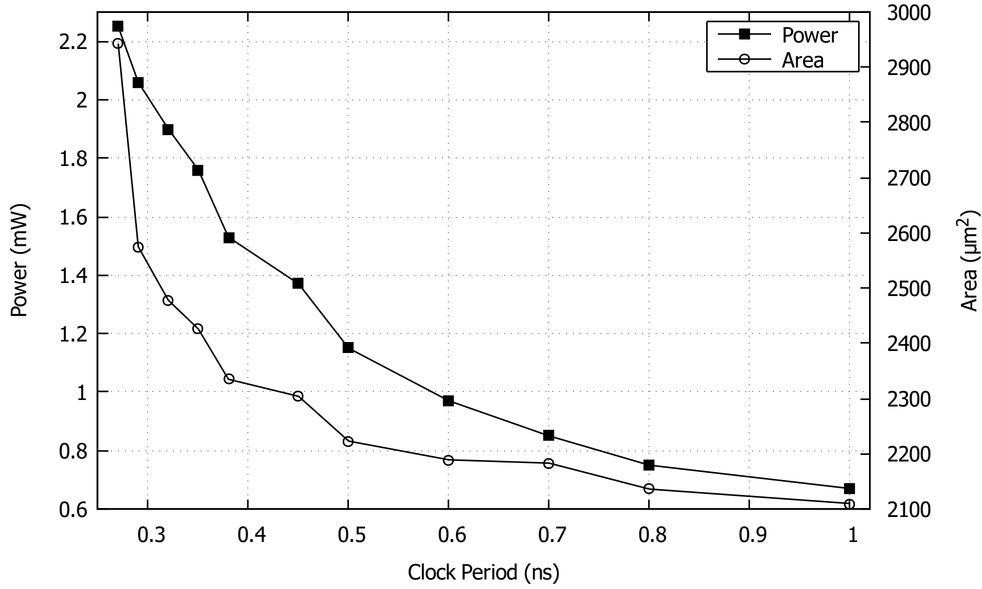
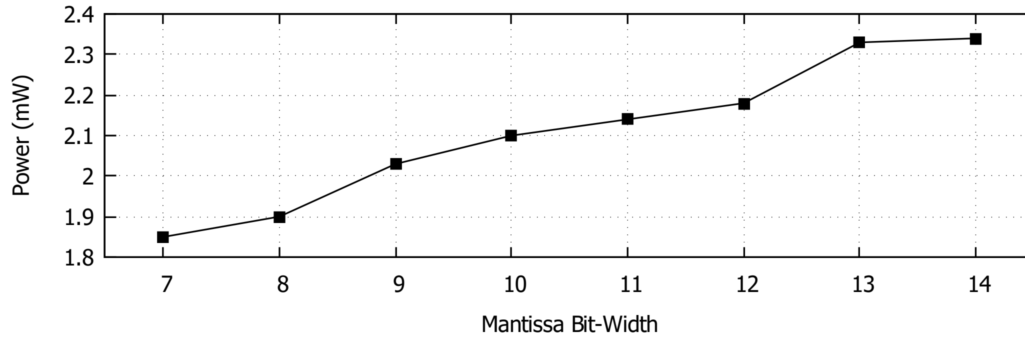


Figure 5.13: Power-Delay curve and Area-Delay curve of the proposed design under STM-28nm

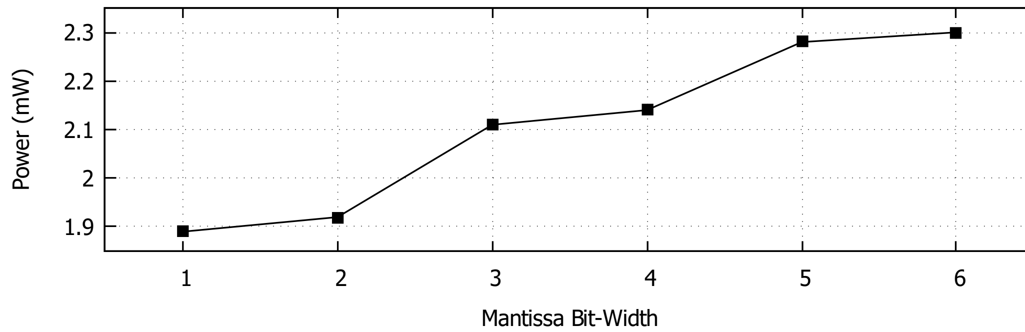
design metrics are compared with the standard arithmetic unit. In addition, this design point is used to analyze the power consumption of the proposed unit under different operational modes.

In different operational modes, the proposed design has different power value. In floating-point mode, by using different number of mantissa, the power number is different. This comes from the trade-off between multiplier and shifter. With larger bit-width for mantissa, more logic of the multiplier will be enabled to perform multiplication. However, as the exponent bit-width in this case is small, fewer levels of shifter can be enough. The power figure of the proposed design in 16-bit floating-point mode is shown in Figure 5.14(a). As the mantissa bit-width increases, the power consumption of the proposed design also increases.

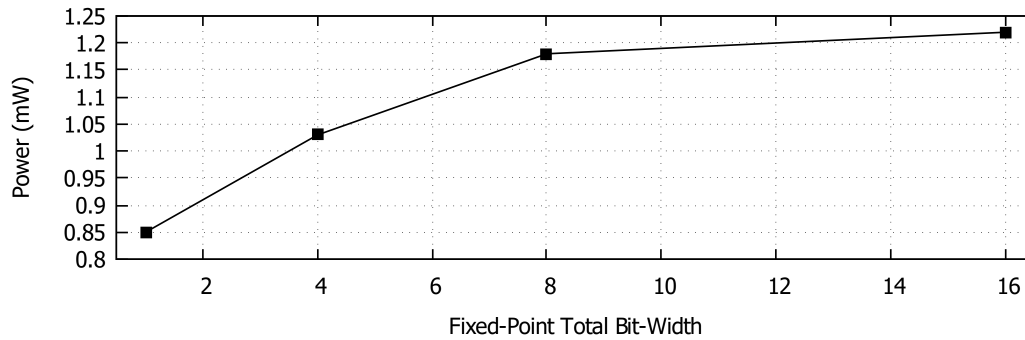
The power figure of the proposed design in 8-bit floating-point mode is shown in Figure 5.14(b). It has similar trend as the 16-bit floating-point mode where the power increases with the number of mantissa bit-width increases. The power consumption of a single 8-bit floating-point operation should be less than that of a 16-bit floating-point operation. However, as the proposed MAC unit supports two parallel 8-bit floating-point operations, the total power consumption is similar to a 16-bit floating-point operation as they actually share the same hardware resources.



(a) 16-bit floating-point mode



(b) 8-bit floating-point mode



(c) 8-bit floating-point mode

Figure 5.14: Power consumption of the proposed design in various operational modes (with timing constraint 0.27 ns)

Table 5.3: Comparison of the proposed MAC unit with standard arithmetic units

Design	Latency	Delay (ns)	Area (μm^2)	Power (mW)	Energy/op[#] ($\times 10^{-13} J$)					
					FLP32	FLP16	FLP8	FIX16	FIX8	FIX4
FIX16-MAC	3	0.16	1509	1.16	-	-	-	5.57	-	-
FLP16-MAC	3	0.25	2415	1.98	-	14.9	-	-	-	-
BFL16-MAC*	3	0.25	2140	1.75	-	13.1	-	-	-	-
FLP32-MAC	3	0.40	6689	5.65	67.8	-	-	-	-	-
MP-MAC†	3	0.26	2681	2.09‡	-	16.3	8.15	8.97	4.25	1.91
Proposed	**3**	**0.27**	**2943**	**2.25‡**	-	**18.2**	**9.11**	**9.88**	**4.78**	**2.08**

* MAC unit designed for BFloat16 format (1-bit sign, 8-bit exponent, and 7-bit mantissa).
† Multiple-Precision MAC unit supporting the same operation modes as the proposed design without flexible precision support.
‡ Average power of all supported operation modes.
# energy/op = (latency × delay × power)/number_parallel_operations.

In fixed-point mode, the power consumption of the proposed unit is much smaller than that of floating-point mode. The power figure of the proposed design under different fixed-point mode is shown in Figure 5.14(c). In fixed-point mode, all the shifting and alignment can be done with constant shifter. The alignment control and LZAC used in floating-point mode are disabled. In addition, there is no need to handle exponent for fixed-point modes. Moreover, the rounding unit for fixed-point mode is less complex compared to the floating-point rounding unit. The subnormal handling logics are also not required in fixed-point mode. Within fixed-point modes, when using lower precision, as shown in Figure 5.7, only part of the multiplier array are used. In addition, in lower precision mode, the higher order part of the adder is not used. All these lead to the power reduction.

To the best of our knowledge, there is no such flexible precision arithmetic unit design appearing in the literature. In order to show the advantage of the proposed design, the proposed design is compared with several standard arithmetic units, including a standard 16-bit fixed-point MAC (FIX16-MAC), a standard 16-bit floating-point MAC (FLP16-MAC), a standard MAC unit designed for BFloat16 format [6] (BFL16-MAC), and a standard 32-bit floating-point MAC (FLP32-MAC). Moreover, in order to measure the resource overhead introduced by the flexible precision support, a multiple-precision MAC unit (MP-MAC) is also designed. This MP-MAC unit does not have the flexible precision feature, but supports

the same operation mode as the proposed design. The comparison results of these designs are shown in Table 5.3. Compared to the standard 16-bit fixed-point MAC, the proposed design required larger area and power, and has larger delay. However, in addition to 16-bit fixed-point MAC operations, the proposed design also supports many other arithmetic operations including floating-point operations.

Compared to the standard 16-bit floating-point MAC unit, the proposed design has only 22% area overhead and 13% power consumption overhead. The proposed design is only 8% slower compared to the standard 16-bit floating-point unit. BFloat16 format has smaller BW_m than the standard half-precision format, and thus the area and power of the BFloat16 MAC become even smaller. Compared to the BFloat16 MAC unit, the proposed design has 37% area overhead and 28% power consumption overhead. However, in terms of functionality, the proposed design can support many other operations, such as flexible floating-point operations and fixed-point operations. With the proposed design, the applications will have more choices to select their most suitable numerical precisions.

The proposed MAC unit is also compared to a standard 32-bit floating-point unit. The standard 32-bit floating-point unit can provide 8-bit exponent for the application. In conventional processors, when the required bit-width of exponent is larger than 5, half-precision unit cannot perform the operations and one has to resort to the single-precision unit. However, in the proposed design, because the flexible precision support, 8-bit exponent can be provided in the same hardware. As discussed in the introduction, for neural network computing, the exponent part is more important than mantissa part and the bit-width of exponent should be satisfied first. The remaining bits are allocated to mantissa. Therefore, in terms of functionality, the proposed design can be compared with 32-bit single-precision MAC unit. As shown in Table 5.3, due to the large mantissa bit-width, the hardware costs of the single-precision unit is much higher than those of the proposed unit. Therefore, for deep learning applications, the proposed MAC unit can provide almost the same accuracy with much lower hardware cost.

Compared to the MP-MAC design, the proposed design has the flexible precision support. This flexible precision support introduces only 9.7% area overhead and 7.6% power overhead. The overhead of energy consumption under each operation mode is also small. The resource

overhead mainly comes from the mask operation during input and output processing. With small resource overhead, the support of flexible precision will bring more flexibility for the applications, especially the machine learning training and inference.

Note that the standard high precision unit can also be used to perform low precision operations, for example FIX16-MAC can be used to calculate FIX8 MAC. However, as there is no parallel low operations support, only one operation can be performed at each clock cycle. Although the power consumption is slightly reduced due to a reduced toggle rate at higher order bit positions, the energy per operation is almost the same as the one when performing operations for the highest supported precision. On the other hand, for the proposed design, parallel low precision operations are supported and thus the energy per operation at low precision can be significantly improved.

5.5 Case Study

In order to show the power merits of the proposed MAC unit in actual deep neural network applications, a case study of deep neural network inference operations is performed. In this case study, the precision required by neural network inference are extracted from the results of [71]. For inference operations, only the fixed-point formats are used in this study.

To simplify the testing process and to put emphasis on arithmetic unit of the deep learning processor, we do not use a whole deep learning processor architecture to perform this case study. Instead, we simulate the deep neural network computing process. For example, when simulating LeNet, input image data are used as the test vectors in the testbench of the proposed MAC unit. Then, the generated results, which are the first layer's outputs, are used again as the second layer's input. This process is repeated until the final neural network layer is processed. For each test case, 100 images are randomly selected from the validation set of MNIST [72] (for LeNet) and ImageNet database [73]. The signal activity file dumped during this simulation process is used to measure the power consumption of using the proposed MAC unit in Synopsys PrimeTime PX. In addition, the inference results from the simulation process are collected to evaluate the neural network accuracy when using the proposed MAC unit as computing elements.

This case study focuses on the different precision requirements of different layers and of different neural network models. In this case study, six neural network models are included: LeNet, AlexNet, NiN, GoogLeNet, VGG-M, and VGG-19. Only convolutional layers in these neural networks are considered because most of the neural network computations are in convolution layers [5]. The test set used in this case study contains both small scale and large scale neural networks. In addition, according to the minimum precision requirements of these networks in [71], this test set contains both small precision operations and large precision operations. Specifically, the precision configuration is: LeNet (2-3), AlexNet (9-7-4-5-7), NiN (8-8-7-9-7-8-8-9-9-8-7-8), GoogLeNet (10-8-9-8-8-9-10-8-9-10-8), VGG-M (6-8-7-7-7), and VGG-19 (9-9-9-8-12-10-10-12-13-11-12-13-13-13-13-13), where the number represents the bit-width of activation and weight in a layer. Under this configuration, these neural networks can achieve 99% of the accuracy when using 32-bit floating-point numbers.

In addition to the proposed MAC unit, the standard 16-bit fixed-point MAC is also used in this case study. The 16-bit fixed-point MAC is used in two ways: (1) FIX16: all activation and weight of the six neural networks are quantized to 16-bit fixed-point numbers and then fed to the 16-bit fixed-point MAC unit; (2) FIX16F: all activation and weight use the flexible precision configuration (the same as the one used for the proposed unit) and they are then sign extended to fit 16-bit bit-width. The results of the proposed MAC unit are compared with these two units.

The average power consumption when implementing using the proposed unit is 1.18 mW while the FIX16 implementation can achieve 1.61 mW and the FIX16F can achieve 1.16 mW. The FIX16 implementation consumes more power because it uses higher precision format than the proposed unit and the FIX16F, and thus the signal toggle rate is higher. The proposed unit consumes higher power than FIX16F because of the extra logic to support other operation modes.

The power efficiency of these implementations is also estimated, as it is a common merit to evaluate the performance of a deep learning hardware design [74]. In this case study, power efficiency is measured in terms of image processed per second per watt ($img/s/W$). To calculate power efficiency, the throughput in terms of img/s is first calculated. As only a simplified neural network computing model is used in this case study, we cannot actually

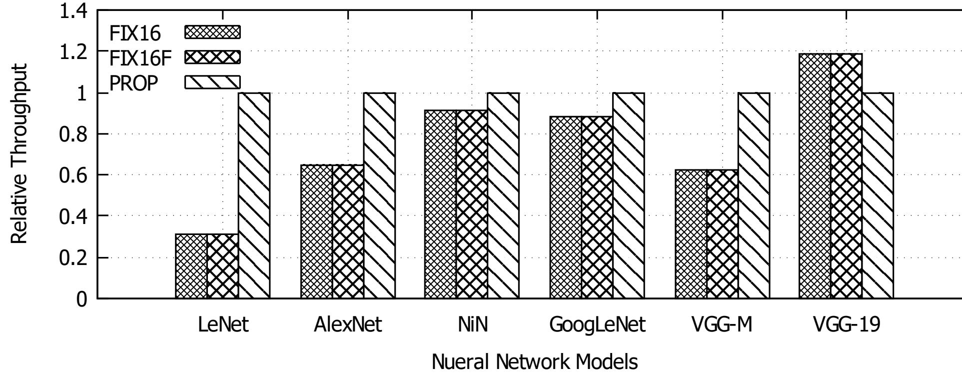


Figure 5.15: Relative throughput of three implementations for six neural networks

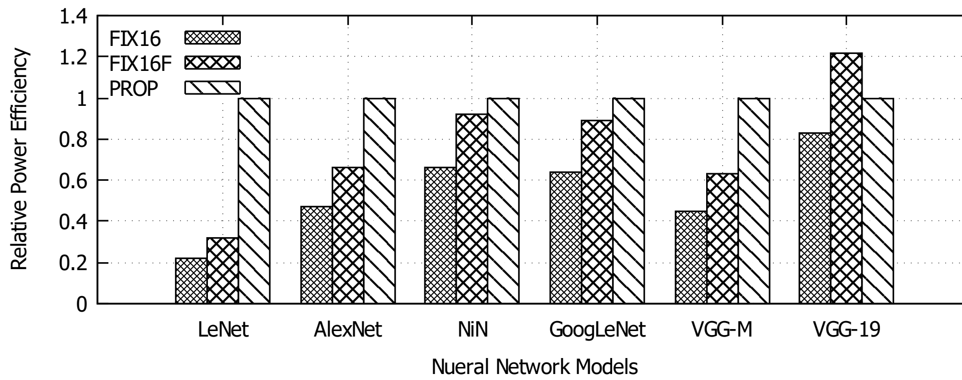


Figure 5.16: Relative power efficiency of three implementations for six neural networks

measure the exact processing time. Instead, the number of required MAC operations to process each image is used to represent the processing time. Therefore, the throughput can be computed with 1/(mac_operations × delay_of_each_cycle). The full results are shown in Figure 5.15. Here the throughput of FIX16 and FIX16F relative to the proposed unit is shown. Although FIX16 and FIX16F have smaller delay compared to the proposed design, for some neural network models, as the proposed unit has the ability to perform parallel low precision operations, the throughput of the proposed unit can be higher than FIX16 and FIX16F. For VGG-19, except the fourth layer, all other layers require more than 8-bit precision. In this case, operations cannot be done in parallel which leads to the throughput degradation for the proposed unit.

The power efficiency is then evaluated in terms of $img/s/W$. The full results are shown in Figure 5.16. Although the average power consumption of the proposed unit is higher than that of standard 16-bit fixed-point unit, as parallel low precision operations are supported

in the proposed unit, the proposed design can achieve an improved power efficiency under most of the neural network implementations. VGG-19 is still an exception since it cannot be quantized to 8-bit or less. Therefore, when low precision operations are feasible for a neural network mode, the proposed MAC unit can process with a good power efficiency.

Note that one may use parallel standard low precision units to improve the power efficiency, however, in this case, higher precision operations will not be supported and thus some neural networks, for example VGG-19, cannot be implemented. The proposed unit, on the other hand, also supports high precision operations and even floating-point operations which will be very flexible for neural network computing.

5.6 Discussion

The Primary Target of the Proposed Design

The primary target of the proposed unit is to provide the support for as many deep neural network models as possible (instead of optimizing the hardware for a specific neural network model). Up to now, different neural network models or different neural network operations show very diverse computing requirements. As discussed in this chapter and many other works in the literature, floating-point formats are preferred in neural network training. However, in inference, fixed-point operations are preferred. In addition, for different neural network models, their minimum required numerical precisions can also be different. In the future, with more new neural network related models or algorithms to be proposed, the diversity in computing requirements could be more obvious. With these considerations, the multi-precision unit becomes a good candidate to support as many precisions of computations as possible. Admittedly, the multi-precision unit has resource overhead over conventional standalone single mode unit, however, it provides the support for many more functions to enable a flexible yet efficient neural network computing capability. The general-purpose support for neural network computing is suitable to be used in server applications where different models and different operations are all required to be handled.

In practical applications, a 16-bit fixed-point unit can handle almost all inference operations. However, for some neural network models or layers, lower precisions are enough to

provide satisfying accuracy. In these cases, the 16-bit unit can still perform low precision computation. However, still only one computation can be performed each time while the rest of the resources are not utilized. In order to improve the resource utilization, the 8-bit and 4-bit fixed-point computation modes are included to make the computing unit a multi-precision unit. The parallel operations can also improve the throughput and thus the energy efficiency can be improved.

With multi-precision fixed-point unit, the inference operations can be handled efficiently. However, the training operations are not well supported with only fixed-point computations. Although the software emulation can use fixed-point hardware to accomplish floating-point operation, the performance is significantly degraded. In order to support training efficiently, the floating-point operational modes are included. In the proposed unit, both flexible 16-bit and 8-bit floating-point modes are supported mainly for training operations.

As the computing requirements of neural networks are diverse, for neural network hardware designs, it is important to find the balance between operational performance and flexibility. The proposed flexible multi-precision hardware can be seen as an 'extreme' design case for the flexibility (while the performance is not hurt significantly). It will be helpful in server usage where the computation requirements from users are very diverse. In the edge or end user sides, it is possible that not all supported modes are required. In these cases, the required modes can be extracted and the others can be removed. For example, if only inference engine is required for a specific application, then the floating-point support can be removed.

The Rule of the Proposed Design

The proposed unit can be used to realize flexible precision computing in hardware platform. However, it still depends on the deep learning algorithm itself to find the required precisions. The proposed design is the hardware part of a software and hardware co-design project. In the proposed unit, several arithmetic flags are used to indicate whether an overflow or underflow happens during the computations. If overflow or underflow happens, then a larger dynamic range format can be used.

In actual applications, the hardware platform always works together with the software

platform. In the software side, one can decide which numerical format to use in the computation base on accuracy requirements or algorithm requirements. Then the corresponding control or configuration information will be passed to the hardware platform. The hardware will be configured to a specific operational mode and perform the computation. After doing the computations, the hardware will return the results and flags to the software. If overflow or underflow happens, a change in numerical format is usually required. Otherwise, for example for deep learning applications, the software will evaluate the current accuracy and to determine whether changes in numerical format are required.

In the literature, these exists some works providing flexible floating-point libraries, such as the FlexFloat in [75]. In [75], a software floating-point library, FlexFloat, is proposed to support customized floating-point formats and arithmetic operations. It provides the flexibility to define arbitrary exponent and mantissa bit-width in order to support the transprecision computing. When performing arithmetic operation, the data is stored in a hardware supported format and is then constrained by the customized format (the 'sanitized' process in the reference paper). The computation is performed with the hardware supported format. The result is sanitized again and returned to the user. The actual precision tuning is done by another tool called 'fp-PrecisionTuning' [76].

When using such a software library in conventional hardware, a sanitizing process is required which is to convert the customized format into the hardware supported format. This process will slow down the computation and if the data value cannot be exactly represented by the customized format, rounding operation is required which will cause the loss of information. In addition, the precisions supported by the hardware may affect the exploration space when using 'fp-PrecisionTuning' to look for the optimal precisions.

When using the proposed flexible unit, with more supported precision modes, in most cases, the 'sanitizing' process can be avoided. In addition, the precision tuning tools are able to perform precision optimization in a larger exploration space. The proposed hardware architecture will facilitate the use of such software library.

In the literature, some software level reduced-precision deep learning tools are available, such as the Ristretto [11] (based on Caffe) and Intel Distiller [77] (based on Pytorch). These tools can be combined with our proposed hardware architecture to realize flexible deep neural

network computing.

The Support of Binary Neural Network

In the proposed architecture, the binary neural network can be computed. However, the performance cannot compete with pure binary neural network hardware (only XNOR logic with counter) due to the support of normal 4-bit, 8-bit and 16-bit computations.

The support of binary neural network can be divided into inference and training because the numerical formats required in these two process are different. For inference, the computation is more straightforward. Only binary logic XNOR operation and counter are used. To perform this operation, the XOR gate and inverter that are used in Booth decoding (the logic to determine whether partial product is ± 0 or $\pm Multiplicand$ or $\pm 2 \times Multiplicand$ using three bits of the multiplier) are reused as the XNOR gate. In addition, the deepest column in the partial product accumulation array is reused as the counter. As there are 8 Booth decoders used in the proposed design and the deepest column has 8-bit, therefore, 8 binary pixels can be computed each time.

The training of binary neural network requires more than XNOR logic. According to [26], during training, due to the computation of gradient, the parameters are still required to be saved as real numbers (instead of binary numbers) and their values are in [-1, 1]. The computation of these real valued parameters can be done with one supported mode of the proposed unit. The mode to be used (4-bit or 8-bit or 16-bit or floating-point) can be determined by the algorithm.

5.7 Summary

In this chapter, an efficient flexible multiple-precision multiply-accumulate (MAC) unit is designed for deep neural network training and inference. The proposed MAC unit is designed based on the requirements of deep neural network computing, for example, low-precision requirements, multiple-precision requirements, and flexible precision requirements for different operations and model. The proposed MAC unit supports both floating-point operations and fixed-point operations. For floating-point operations, the proposed MAC unit supports one

16-bit operations or two 8-bit operations. With flexible precision support, the bit-with of the exponent and mantissa can be mutually exchanged. The proposed unit also supports fixed-point operations for deep neural network inference. It supports one 16-bit operations, or two 8-bit operations, or four 4-bit operations. At the lowest precision, it also supports binary neural network operations. The proposed MAC unit can be used in deep learning processors to enable both training and inference in the same architecture and used together with reduced precision deep learning tools, such as Ristretto and Intel Distiller, to facilitate the deep neural network accelerator design towards more functionality and more energy efficiency. The proposed MAC architecture can also be used to design neural network IP cores for FPGA devices.

Chapter 6

Posit Multiply Accumulate Unit[1]

This chapter presents a posit based MAC architecture for deep learning computing. Posit number format has been proved to be efficient in deep learning applications. The design of a posit based MAC unit is expected to facilitate the use of posit number system in deep learning computing. Section 6.1 presents the motivations to design such a MAC architecture. The proposed posit MAC architecture is presented in Section 6.2. Section 6.3 presents the synthesis results of the proposed architecture. Section 6.4 concludes this chapter.

6.1 Introduction

The single-precision and half-precision floating-point formats defined in IEEE 754-2008 [8] have been widely used in deep learning [54] training and inference. However, since floating-point formats use a uniform representation for numbers, they are not efficient in deep learning applications where the network parameters are usually distributed non-uniformly [11].

Recently, the posit number format [16] is proposed. It is claimed to be more accurate than floating-point numbers. Also with the same bit-width, it can provide larger dynamic range than floating-point numbers. More importantly, the posit format represents numbers in a non-uniformly method. For smaller value, the posit format can provide more precision and for large value the precision is reduced. This non-uniform representation makes it suitable in

[1]The content of this chapter is originally published in the proceedings of 2019 IEEE International Symposium on Circuits and Systems (ISCAS 2019) [78]. The manuscript has been reformatted for inclusion in this thesis.

Hao Zhang (HZ), Jiongrui He (JH) and Seok-Bum Ko (SK) designed the study. HZ developed the parameterized architecture, developed the C-based generator and performed logic synthesis and results analysis. JH helped to develop HDL reference model and performed verification of the HDL code created by the C-based generator. HZ prepared the manuscript with contributions from SK and JH to the manuscript structure, readability and analysis and discussion of the results.

deep learning applications [16] [79], where it can achieve same level of accuracy with smaller bit-width.

In the literature, some works have been performed on the design of hardware posit multiplier [80] and posit adder [80] [81]. However, to the best of our knowledge, there is no work discussing the design of hardware posit fused arithmetic operations at the time of writing. The fused arithmetic unit, such as MAC unit, has several advantages over separate multiplier and adder. In the MAC unit, the multiplication part and addition part can share components with each other which leads to a reduced area and power consumption. In addition, rounding operation is performed only once after the final addition which can lead to a better accuracy. Due to these advantages, in deep learning applications, MAC units are widely used to perform convolution or dot-product operations [5] [53] [82]. Therefore, in order to facilitate the use of posit in deep learning, the design of MAC unit is required to be investigated. Moreover, due to the flexible posit format, the extraction of each component from posit format and the packing of resulting components to form posit format are relatively hardware expensive. In fused operations, more arithmetic operations can be done with performing only one extraction and packing. This is expected to make the operations faster and more area efficient compared to separate single arithmetic operation unit.

In this chapter, a posit MAC generator is proposed. The generator is developed with C language. By providing the total bit-width nb and exponent bit-width es to the generator, the Verilog HDL code of the corresponding posit MAC unit will be generated. The MAC architecture created by the proposed generator is combinational design, however a 5-stage pipeline strategy is also presented. The full rounding operation is performed on the result where the rounding to nearest even method, as defined in [8], is implemented. The functionality of the generated MAC designs under various design parameters are verified with extensive testing vectors. The worst case delay, area, and power consumption of the generated MAC unit under different design parameters are analyzed in this paper.

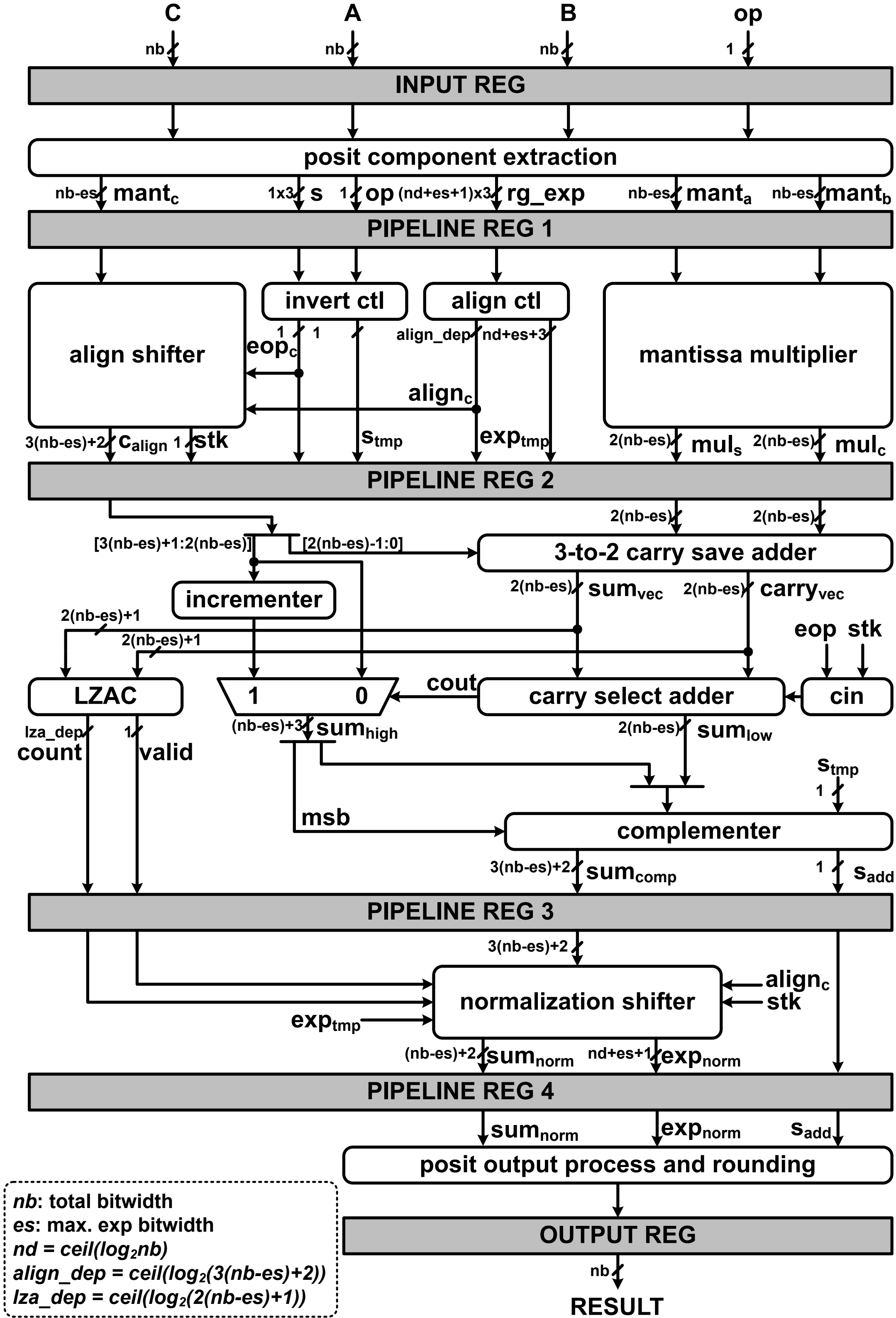


Figure 6.1: Datapath of the proposed posit multiply-accumulate unit architecture (The proposed generator will give combinational design. Pipeline in this figure is just an example.)

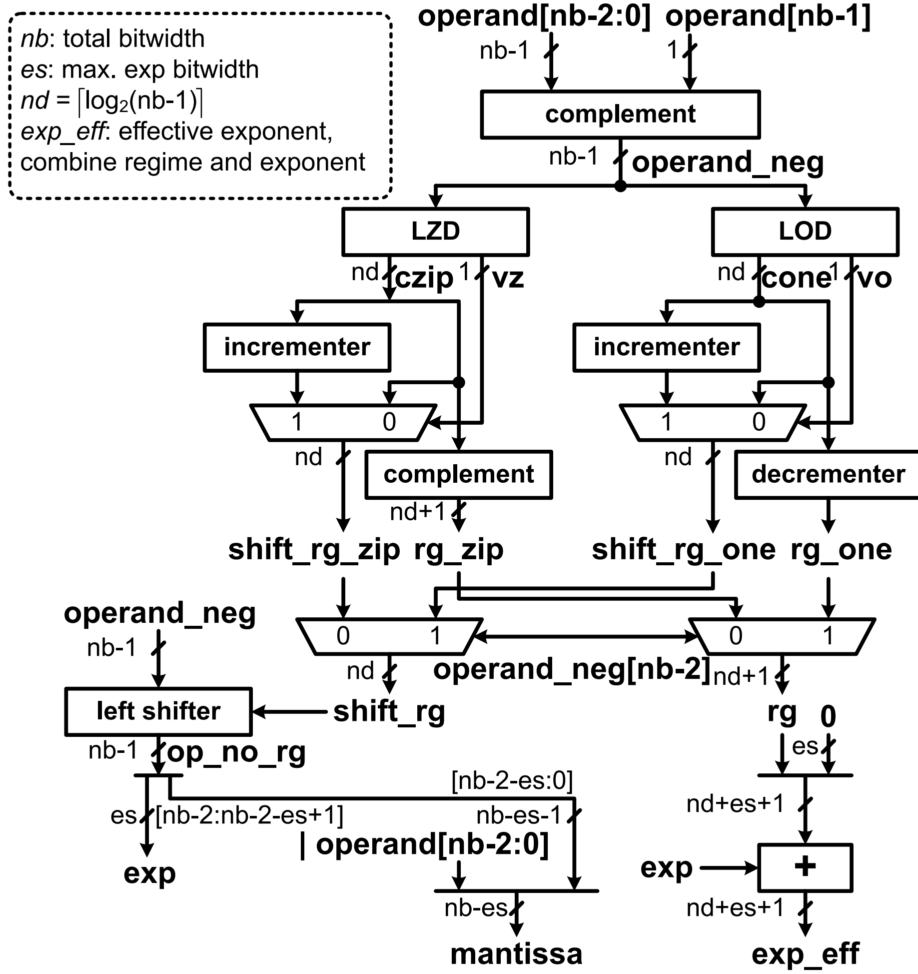


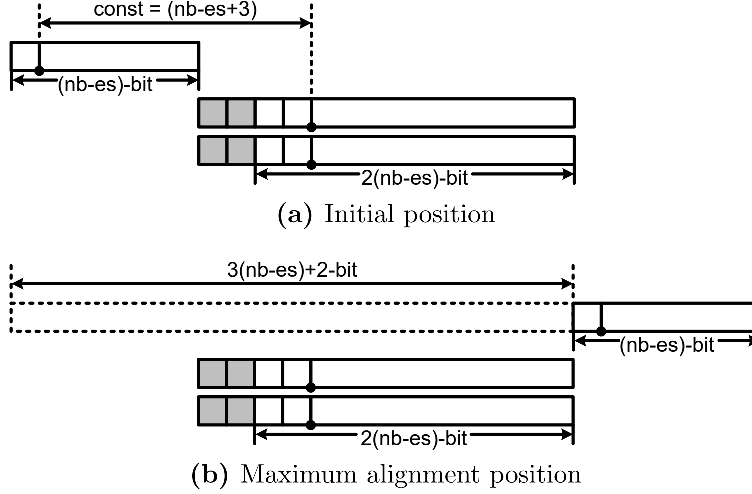
Figure 6.2: Diagram of posit format component extraction

6.2 The Proposed Design

The proposed posit MAC generator is designed based on the architecture shown in Figure 6.1. It basically follows a standard floating-point MAC or fused multiply-add unit architecture. The bit-widths of all datapath are parameterized so that it can be used to generate different designs with different nb and es.

6.2.1 Posit Component Extraction

The diagram of posit component extraction is shown in Figure 6.2. The sign bit is first evaluated to determine if the remaining parts need to be complemented. Then regime value is evaluated according to the definition in [16]. Then the operand will be left shifted to



(a) Initial position

(b) Maximum alignment position

Figure 6.3: Datapath of the alignment shifter

remove the regime bits. After shifting, the most significant es-bit will be exponent and the remaining bit prefixed with implicit bit will be mantissa. In this module, the regime bit and exponent bit are also combined together. The regime value rg is first left shift with es-bit and then added with exponent to generate the effective exponent. This effective exponent will be used in later stages for shifting control.

6.2.2 Mantissa Multiplier

Mantissa multiplier is a $(nb-es)$-bit unsigned multiplier. It is implemented with radix-4 modified Booth multiplication algorithm [49]. The partial products generated are accumulated by several levels of (4,2) carry save adders. The mantissa multiplier is also parameterized so that it can be generated for any required bit-width.

6.2.3 Alignment Shifter

The datapath of the alignment shifter is shown in Figure 6.3. The mantissa of C is first put 2-bit to the left of the product. The shifting amount is determined by the exponent difference and a constant value, $shift_{amount} = const - (expc_{diff} - (expa_{diff} + expb_{diff}))$ where $const = (nb - es) + 3$. C does not require any left shifting so the minimum shifting amount is zero. The maximum shifting happens when the whole C is shifted out to the right

of the product. The maximum shifting amount is $3(nb - es) + 2$.

6.2.4 Adder

After alignment, the least significant $2(nb - es)$-bit of aligned C will be combined with the product using one level of (3,2) carry save adder. Then the resulting sum vector and carry vector are added by a carry propagate adder. For the most significant $(nb - es) + 2$-bit position of aligned C, the addition can be done with an incrementer. The output carry of the lower order adder will determine whether the higher part need to be incremented or not. The two part results are combined to form the adder result. If the result is negative, it needs to be complemented.

6.2.5 Leading Zero Anticipator

The least significant $2(nb - es) + 1$-bit of both adder operands are sent to LZA to count the possible leading bit number in the adder result. As both positive and negative results are possible, we use the indicator that consider both leading zero and leading one in [50]. The counting of the indicator is done with a tree structure. The possible error of the leading zero anticipation will be corrected in the last stage of normalization shifter.

6.2.6 Normalization Shifter

The normalization shifter is implemented with a 2-stage shifter where a constant shifter is followed by a dynamic shifter. If exponent difference exp_{diff} of the three operands is smaller than 2, then a constant shifting with an amount $(nb - es) + 1$ is first performed and followed by a dynamic shifting with an amount counted by LZA. Otherwise, constant shifter is not used and the shift is performed by dynamic shifter with an amount equal to alignment amount. At the final stage, the possible 1-bit error introduced by LZA will be corrected. Also overflow will be checked and if overflow happens, a 1-bit right shift is performed. The exponent will be updated according to normalization shifting amount.

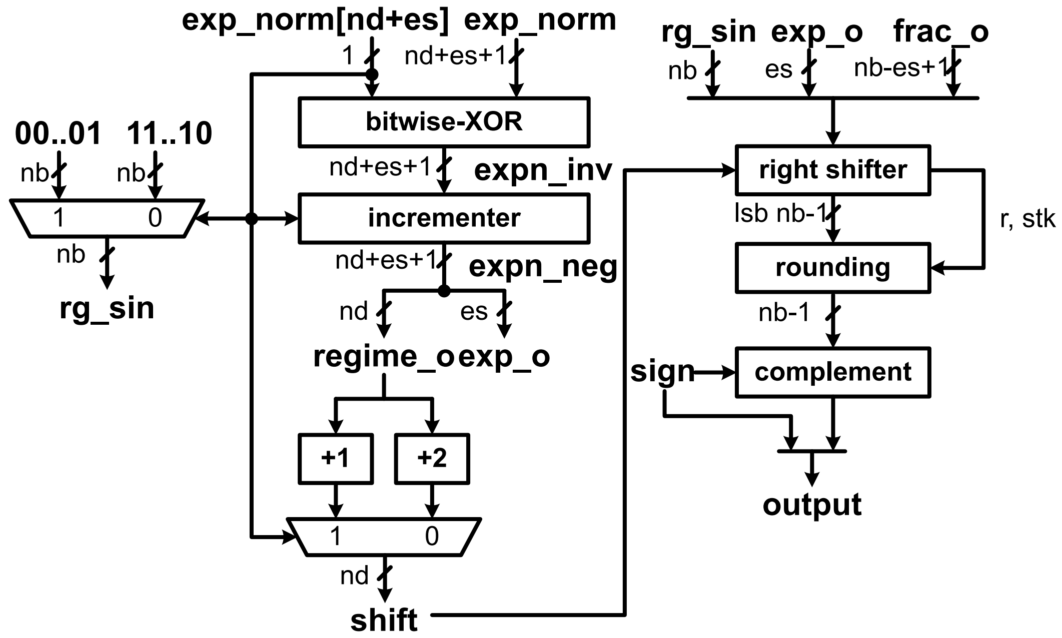


Figure 6.4: Diagram of posit output processing and rounding

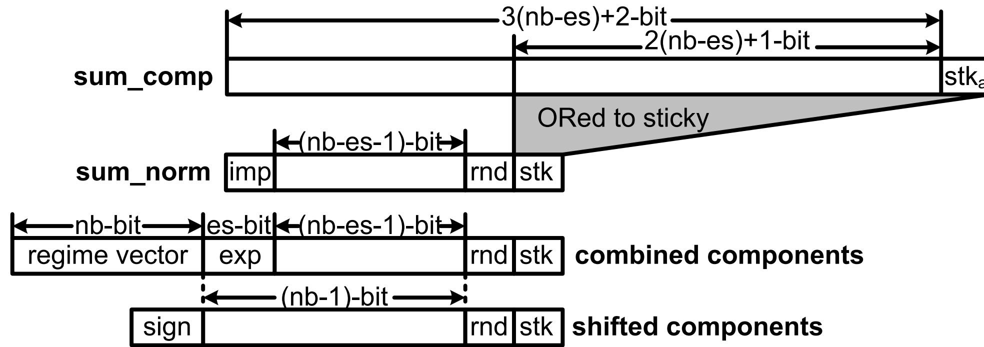


Figure 6.5: Rounding of the posit result

6.2.7 Posit Output Process and Rounding

The diagram of posit output process is shown in Figure 6.4. The combined exponent is first divided into regime and exponent. A nb-bit regime vector rg_sin is also generated. The resulting mantissa part will be prefixed with exponent and then regime vector. Then, depending on the regime value, a right shift is performed to the combined vector. Finally, the least significant $nb-1$-bit are reserved and combined with sign bit.

The rounding process is shown in Figure 6.5. The result of addition will be a $3(nb-es)+2$-bit vector. It will be normalized and then the least significant $2(nb-es)+1$-bit are condensed into the sticky bit. The vector is then prefixed with exponent and regime vector and a right

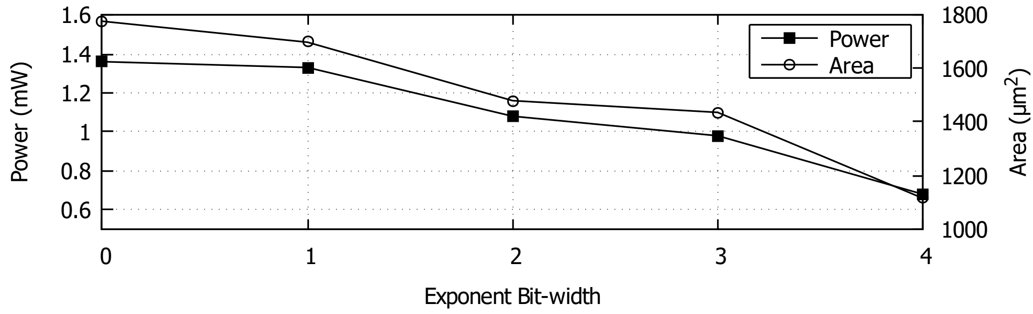
shifting is performed. During shifting, any bit shifted out are condensed into sticky bit. Finally, according to sticky bit, round bit and least significant bit, the round to nearest even can be performed.

6.3 Results and Analysis

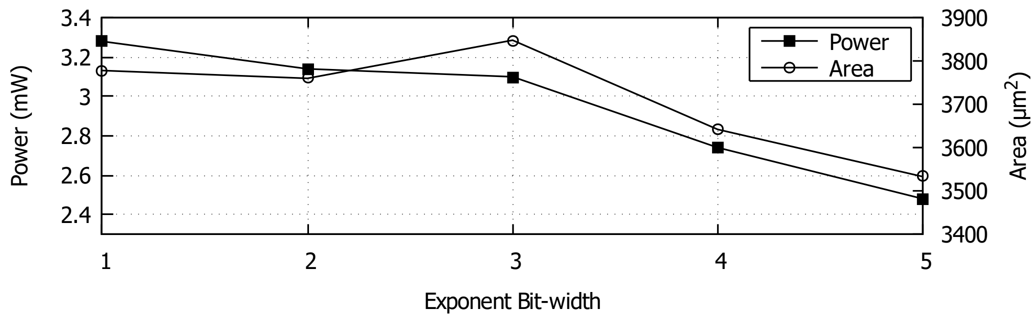
The Verilog codes of 8-bit, 16-bit, and 32-bit posit MAC with various exponent bit-widths are created by the proposed generator. For 8-bit, 16-bit, and 32-bit, the exponent bit-width range is 0-4, 1-5, and 1-8, respectively, where the largest exponent bit-width of each case is the same as that of the corresponding floating-point format. Each posit MAC design is simulated with extensive testing vectors to verify its functionality. The testing vectors are generated with the help of [83]. Then each MAC design is synthesized with Synopsys Design Compiler using STM-28nm technology library with typical case parameters (25℃, 1.00V).

The synthesis results are shown in Figure 6.6(a), Figure 6.6(b), and Figure 6.6(c). Within the same total bit-width, the delay achieved by different designs with various exponent bit-width is similar. Therefore, the 8-bit, 16-bit, and 32-bit posit MACs are synthesized with a timing constraint of 1.0 ns, 1.3 ns, and 1.6 ns, respectively. For each set of designs, the area and power consumption goes lower with the increase of the exponent bit-width. This is because with larger exponent bit-width, the mantissa bit-width will be reduced. Therefore, the bit-width of both mantissa multiplier and final adder will be reduced. Although the alignment and normalization control logic will be larger, however, those components only consume small portion of total resources.

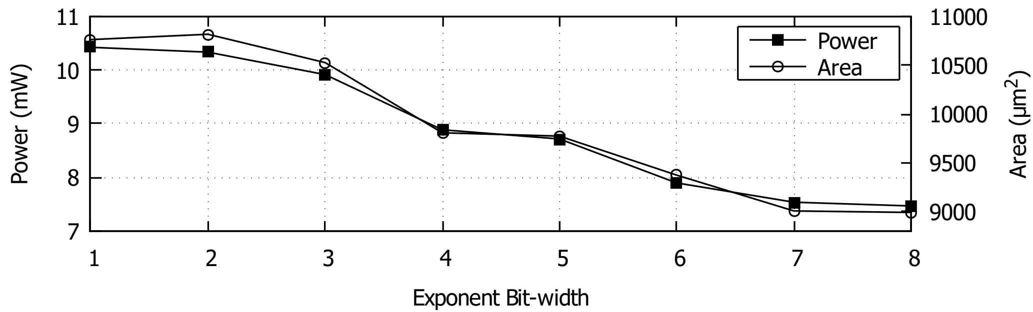
In order to compare the posit MAC with standard floating-point MAC, three standard floating-point MACs (8-bit, 16-bit, and 32-bit) are also designed and synthesized. The 8-bit format uses 4-bit for exponent and 3-bit for mantissa. The comparison results are shown in Table 6.1. Most of the internal operations are similar between posit MAC and floating-point MAC. However, posit has complicated input process to extract each component from posit format and complicated output process to pack each resulting component into the posit format. This leads to the larger delay and higher resource consumption compared to floating-point designs. Although the floating-point designs need to process subnormal numbers and



(a) 8-bit Posit MAC (timing constraint is 1.0 ns)



(b) 16-bit Posit MAC (timing constraint is 1.3 ns)



(c) 32-bit Posit MAC (timing constraint is 1.6 ns)

Figure 6.6: Area and power of 32-bit Posit MAC with various exponent bit-width (timing constraint is 1.6ns)

Table 6.1: Comparison of the Posit MAC with floating-point MAC

	Dynamic Range		Delay		Area		Power
	min	max	*ns*	FO4	μm^2	NAND2	*mW*
Std FP8	2×10^{-3}	3×10^{2}	0.60	25	872	1817	0.59
Posit(8,4)	1×10^{-29}	8×10^{28}	1.00	42	1116	2325	0.68
Std FP16	6×10^{-8}	7×10^{4}	0.75	31	2196	4575	1.80
Posit(16,5)	1×10^{-135}	7×10^{134}	1.30	54	3533	7360	2.48
Std FP32	1×10^{-45}	4×10^{38}	0.93	39	6081	12668	5.14
Posit(32,8)	8×10^{-2309}	8×10^{2311}	1.60	67	8992	18733	7.47

[1] 1 FO4 $\approx$ 24 *ps*, 1 NAND2 $\approx$ 0.48 μm^2 @ 28nm;

Table 6.2: Delay of each pipeline stage of Posit MAC

Pipeline Stage	Posit(8,0)	Posit(16,1)	Posit(32,3)
Input process	0.20	0.28	0.34
Multiply and Align	0.23	0.31	0.37
Adder and LZA	0.21	0.24	0.32
Normalize shift	0.19	0.20	0.28
Output process	0.20	0.29	0.35

perform exceptional cases handling, however, for the MAC architecture used in this paper, these components do not consume many resources [84]. Compared to floating-point unit with the same total bit-width, posit MAC consumes larger resources, however, the posit format can provide much better accuracy and data range for deep learning applications [16] [79]. In other word, to achieve the same level of accuracy, the posit format needs less total bit-width compared to floating-point. Therefore, with posit format, a MAC unit with smaller total bit-width, for example 8-bit instead of 16-bit, can be used in deep learning processor which is expected to result in a hardware efficient design.

In order to improve the throughput of the posit MAC unit in practical applications, pipeline method can be introduced to the posit MAC design. In this paper, a 5-stage pipeline example is shown in Figure 6.1. The worst case delay of each pipeline stage of three example posit MAC designs is shown in Table 6.2. As shown in Table 6.2, this pipeline strategy leads to a balanced pipeline stages. For modern processor designs a critical path delay of 15-25 FO4 is desired [85]. For the 32-bit pipelined posit MAC, as shown in Table 6.2, the worst

case delay of 0.37ns in 28nm library is roughly 15 FO4 which meets the speed requirements of processor design. Depending on the actual requirements, one can increase or reduce the number of pipeline stages in the posit MAC architecture.

6.4 Summary

In this chapter, a posit MAC unit generator is proposed to facilitate the use of posit number format in deep learning applications. A MAC unit architecture based on posit number format is proposed and the bit-widths of all datapath are parameterized to enable the design of MAC unit generator. The Verilog codes generated by the proposed generator are simulated and synthesized. The delay, area, and power merits of the generated MAC units with different total bit-width and exponent bit-width are analyzed. Finally, a 5-stage pipeline strategy is presented to make the design meet the speed requirements of modern processors.

Part IV

Deep Learning Computation using 3D Memory

Chapter 7

Improved Hybrid Memory Cube Architecture[1]

This chapter presents the design of an improved HMC architecture for accelerating weight-sharing deep neural networks. The proposed architecture is designed based on the original HMC architecture with minimal modifications to achieve deep neural network acceleration. Section 7.1 presents the motivations to design such a HMC. Section 7.2 introduces the weight-sharing methods applied in deep neural networks. The proposed HMC architecture, including the modified vault controller and modified instructions, are presented in Section 7.3. Section 7.4 presents the simulation results of the proposed HMC when performing convolution operations and their comparison with previous designs. Finally, Section 7.5 concludes the whole chapter.

7.1 Introduction

Deep learning [54] has achieved great success in recent years. It is widely used in many applications. However, behind its superior performance there is expensive computation cost and memory cost. In particular, the data transfer between processing unit and memory consumes most of the energy [5]. In order to reduce the energy consumption, many efficient hardware deep learning processors are proposed in the literature [5].

Currently, most of the deep learning processors are still designed using conventional

[1]The content of this chapter is originally published in the proceedings of 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS 2019) [86]. The manuscript has been reformatted for inclusion in this thesis.

Hao Zhang (HZ), Jiongrui He (JH) and Seok-Bum Ko (SK) designed the study. HZ developed the methodology and architecture and analyzed the results. JH modified the HMC simulator based on the architecture proposed by HZ and performed performance simulation. HZ prepared the manuscript with contributions from JH and SK to the manuscript structure, readability and analysis and discussion of the results.

DRAM [87] as their memory system. Theses are usually designed with Von Neumann architecture where an interface is used between the processing unit and main memory. This interface is becoming the limiting factor of the performance of deep learning processors. On one hand, the bandwidth within the memory interface is limited where the data transferring speed is much slower than the computing speed provided by modern processors. On the other hand, the data transfer through this interface consumes a significant part of total energy consumption of the whole system. In order to improve the memory performance from the above mentioned two aspects, the HMC [17], one of the 3D memory architecture, is proposed. The HMC stacks many DRAM layers vertically. The data exchange through different layers is realized by the TSV. HMC also contains a bottom logic layer where the vault controller and the interface to host processors are implemented. HMC also provides a lot of logic functions and arithmetic functions in the logic layer. Therefore, data can be processed within the logic layer instead of being transferred through interface to host processor. This ability can significantly reduce memory traffic and thus reduce the energy consumption.

In the field of deep learning model research, many recent research works are focused on compressing the deep neural network models to reduce their memory footprint [88–90]. In [89], the authors proposed to compress and quantize the weight parameters of the deep neural network model. So that there will be only limited number of weights (16 weights used in [89]). The advantage of this compression is that the input activations corresponding to the same weight value can be accumulated first. Then only one final multiplication with the shared weight is performed. Therefore, the total number of multiplications is reduced which will benefit both speed performance and energy consumption.

In this paper, the support of weight-sharing deep neural network in HMC architecture will be proposed by exploiting and applying the arithmetic functions provided by HMC to deep learning applications. Most parts of the conventional HMC architecture are kept with only minor modifications to the HMC architecture in order to avoid thermal problems [91]. The vault controllers and the HMC instructions are modified to perform the required weight-sharing deep neural network operations. The accumulations of data that correspond to the same weight are performed in memory using HMC atomic operation. The accumulated data are then used in multiplication with shared weights and then the resulting products are

accumulated together. This process uses a multiply-accumulate unit integrated in HMC. The major addition and multiplication are done inside memory and therefore, the memory traffic and thus the power consumption is significantly reduced.

7.2 Weight-Sharing Deep Neural Network

The convolution operation in deep neural network is calculated by performing a dot-product operation between the input activations and weight kernels. The dot-product result of each input feature map needs to be accumulated through each channel. In conventional deep learning operations, the deep neural network is trained with 32-bit single-precision floating-point number [8]. Due to the large bit-width and large dynamic range of 32-bit floating-point format, it is not uncommon that all weight parameters are of different values. Therefore, with a kernel of size $R \times S$ and input channel C, the total number of computations to generate an output point required are: $R \times S \times C$ multiplications and $R \times S \times C$ additions. For modern deep neural network [92], C is usually large and thus a large number of multiplications and additions are required.

To reduce the required number of multiplications, in [88] and [89], the authors proposed the network compression and quantization method. They found that, with fine-tunning after quantization, 16 different weight values can provide satisfying accuracy in most deep neural networks. Therefore, they quantize all weight parameters into 16 different numbers. They used 4-bit tag to identify each weight values. During computation, for the positions with same weight tag, they are accumulated first. The multiplication with weights is performed only once after the accumulation is finished. Therefore, the total number of multiplication required to generate an output point is reduced. Depending on the kernel size, no more than 16 multiplications are required for one output.

The process of calculating convolution using weight-sharing method is shown in Fig. 7.1. As shown in Fig. 7.1, 16 accumulation registers are used to accumulate the corresponding data. Then the multiplication and accumulation across different weights are performed. In Fig. 7.1, the multiplication and accumulation are performed in a parallel method. Besides, the multiplication and accumulation can also be performed using a MAC unit in multiple

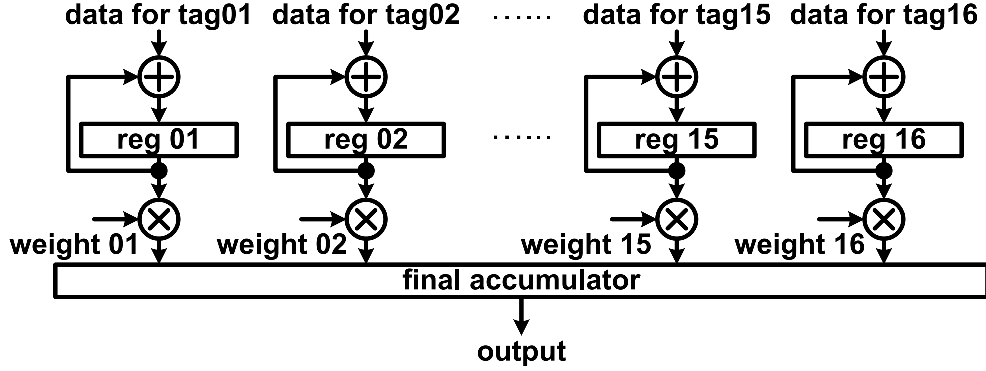


Figure 7.1: Process of convolution using weight sharing method

cycles.

7.3 The Proposed HMC Architecture

The proposed HMC architecture modifies the HMC architecture presented in HMC specification 2.1 [17]. The modifications are kept as small as possible. The newly introduced HMC operations are still light weight arithmetic operations so that there will be no significant thermal problems.

In order to support weight-sharing deep neural network operations, two HMC operations are introduced: one for the accumulation of data corresponding to the same weight and the other one is a MAC operation which is used for the multiplication of the accumulated data and weights and the accumulation of those generated products. The architecture of the modified HMC is shown in Fig. 7.2.

As discussed in Section 7.2, 16 different weight values are enough to maintain accuracy for most of the modern deep convolutional neural networks. We also modified the HMC architecture to accommodate the 16 weight cases. To enable parallel processing, 16 vaults will be used in the HMC architecture. During data transfer, data corresponding to the same weight are sent to the same vault in the HMC memory.

Both data and weight of the deep neural network are in the format of 16-bit fixed-point numbers because the 16-bit fixed-point operations have relatively low cost compared to floating-point operations and can still provide high accuracy for deep neural networks [93]. The data (the input to the deep neural network) will be transferred from storage or other

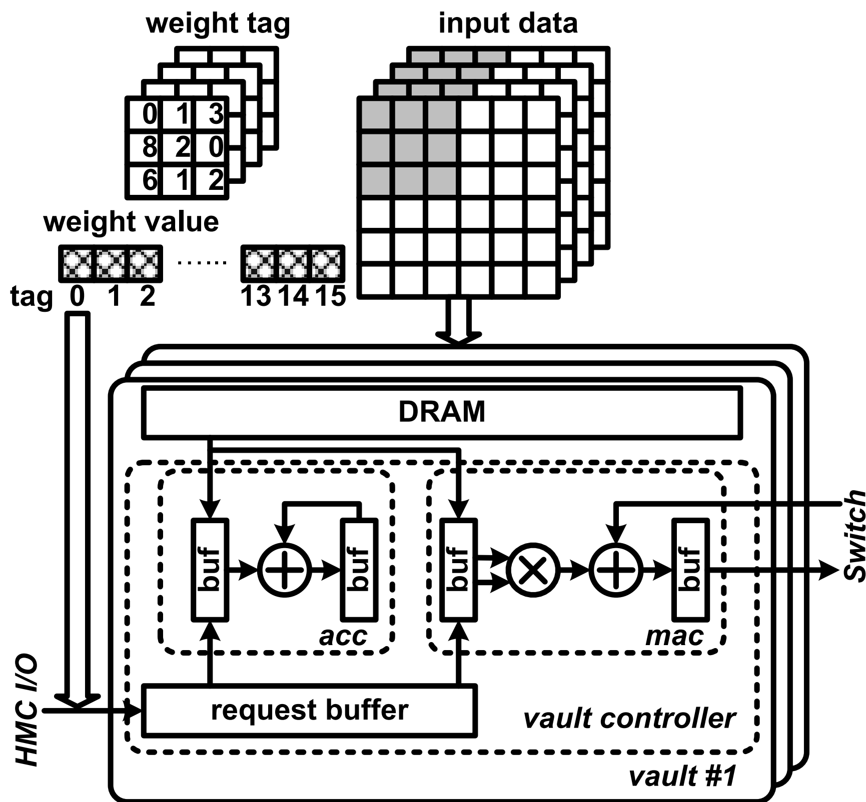


Figure 7.2: Architecture of the modified HMC

devices to the HMC vaults. The weights are loaded to the registers of host processor since there are only 16 different values. They will be used as the immediate value during the multiply-accumulate operation.

7.3.1 Instruction Set

In order for the HMC devices to start an operation, the host processor needs to initiate the operation by sending an instruction. To make the HMC operation more efficient, we use one

Table 7.1: Newly added instruction sets

Data Size	Data Type	Instruction Format
16 bytes	Fixed-Point	WS_ACC RD, RS, CNT
16 bytes	Fixed-Point	WS_MAC RD, RS

[1] RD: Destination register address.
[2] RS: Memory operand starting address.
[3] CNT: number of accumulation operations.

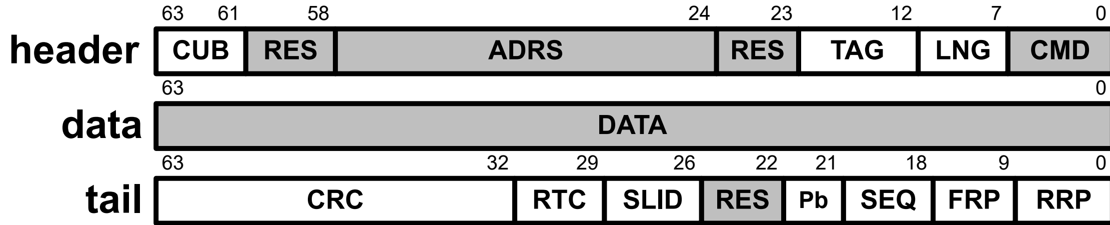


Figure 7.3: HMC memory request packet format

instruction to initiate the HMC operation and then HMC will perform consecutive operations based on the number of operations it obtains through the instruction. During this process, no new instruction is needed to trigger HMC operations from host processor.

To enable weight-sharing operations, two instructions are added to the HMC instruction set: WS_ACC for the accumulation of operation for data and WS_MAC for the weight multiplication and accumulation.

The format of the two added instructions is shown in Table 7.1. The WS_ACC instruction has 3 parameters: RD, RS, and CNT. RD represents the destination register address where the results of the accumulation will be stored. For weight-sharing operation, since there is a total of 16 weights, the number of accumulation registers is also 16. RS is the memory address which represents the starting address of a series of data. Finally, CNT is the number of accumulation operations required. CNT can be generated by host processor according to the distribution of weight kernels and the number of input channels.

The WS_MAC instruction only has 2 parameters: RD and RS since the CNT for WS_MAC is a constant number. We use 16 different weight values and thus use 16 registers to save the accumulated data for each weight. Therefore, the MAC operation is a dot-product of two 16-digit vectors. Similar as the parameters in WS_ACC instruction, RD represents the destination register address where the results of the MAC operation will be stored. RS is the memory address which represents the starting address of the accumulated data.

7.3.2 HMC Request Format

The HMC controller will translate the newly added HMC instructions into HMC memory request. The HMC memory request is in a packet format that is composed of a header, a data field, and a tail [17], as shown in Fig 7.3.

As shown in Fig. 7.3, one memory request packet has three reserved fields (marked as RES in Fig 7.3). The header has two reserved fields: one is 3-bit and the other one is 1-bit. The tail has one reserved field which is 4-bit. These three fields will be utilized to realize the function of the newly added two instructions.

For WS_ACC instruction, the CNT will be encoded in the RES regions. The three RES regions are all used which have in total 8-bit to encode the accumulation count. The 8-bit code can represent up to 256 times accumulation which is enough for most deep neural networks. When generating the memory request packet, the MSB of the CNT is put in 60th bit of header and the LSB is put in the 22nd bit of tail. The 23rd bit of header is put in between the other two RES fields. The command CMD field has a total of 7-bit so that a total of 128 different operations can be supported. The number of HMC operations defined in [17] is much less than 128. Therefore, the unused value in CMD field can be used to encode the newly added WS_ACC operation. In addition, RS is defined in $ADRS$ region and $DATA$ field is occupied by host operand which is 0 for WS_ACC because all data to be accumulated are from DRAM stacks. All these related fields in the memory request packet are highlighted in Fig. 7.3.

For WS_MAC instruction, since the number of operation is a constant of value 16. Therefore, only the 4-bit RES field in tail is used. Similar as the WS_ACC instruction, another unused value in CMD field is used to encode the WS_MAC operation. The $ADRS$ field is used to represent the starting memory address. The $DATA$ field now contains the weight value to be multiplied with the accumulated data.

The newly added two operation instructions can be handled with the conventional HMC memory request packet. Therefore, there is no need to modify the packet format of HMC and thus avoiding significant modification of the HMC architecture.

7.3.3 HMC Memory Control

When using the modified HMC for weight-sharing deep neural network operations, multiple vaults are used at the same time where each vault stores the data corresponding to the same weight value. Therefore, when doing data accumulation, multiple vaults need to be visited in parallel. However, as defined in [17], each HMC memory request can only visit one vault

in conventional HMC architecture. Therefore, in order to enable the multiple vaults parallel visit, the memory controller needs to be modified.

The modification to the HMC memory request controller is not complicated. In the proposed design, the original memory request is analyzed and it will be used to generate multiple vault access requests. So that each vault controller will receive the corresponding access request and then start the accumulation operation.

Multiple vaults access is required both in WS_ACC instruction and WS_MAC instruction. As in WS_ACC operation, each vault needs to perform the accumulation of data corresponding the same weight value. The accumulated results are still stored in multiple vaults. So that when doing the WS_MAC operation, these accumulated data needs to be collected from multiple vaults, each multiplied with the corresponding weight value and then accumulated.

7.3.4 HMC Weight-Sharing Operation

When implementing WS_ACC operation, the instruction contains the starting address and the number of accumulation required. Then, each vault uses the starting address to find the initial memory value and then perform consecutive accumulations. The accumulation operation is implemented as the atomic integer addition [17]. The arithmetic unit required is already available in conventional HMC architecture. There is no need to build extra logic for this accumulation operation.

For WS_MAC operation, a 16-bit fixed-point MAC unit is added to each vault controller. Each MAC unit is followed by a register to store the partial accumulated results. When processing, the 16 MAC units are used in a serial method where the MAC result of a previous operation is used as a input to the next MAC operation and it is accumulated to product generated in the next MAC operation. One operand of the MAC is from the vault memory which is the accumulated data. The other operand of the MAC is from a set of registers in host processor which is the weight value. As the operation of HMC should be initiated by host processor, an immediate value from host processor is required in the memory request packet. In the proposed design, we store weights in registers or buffers of host processor and use them as the immediate value when initiating WS_MAC operations. This method is

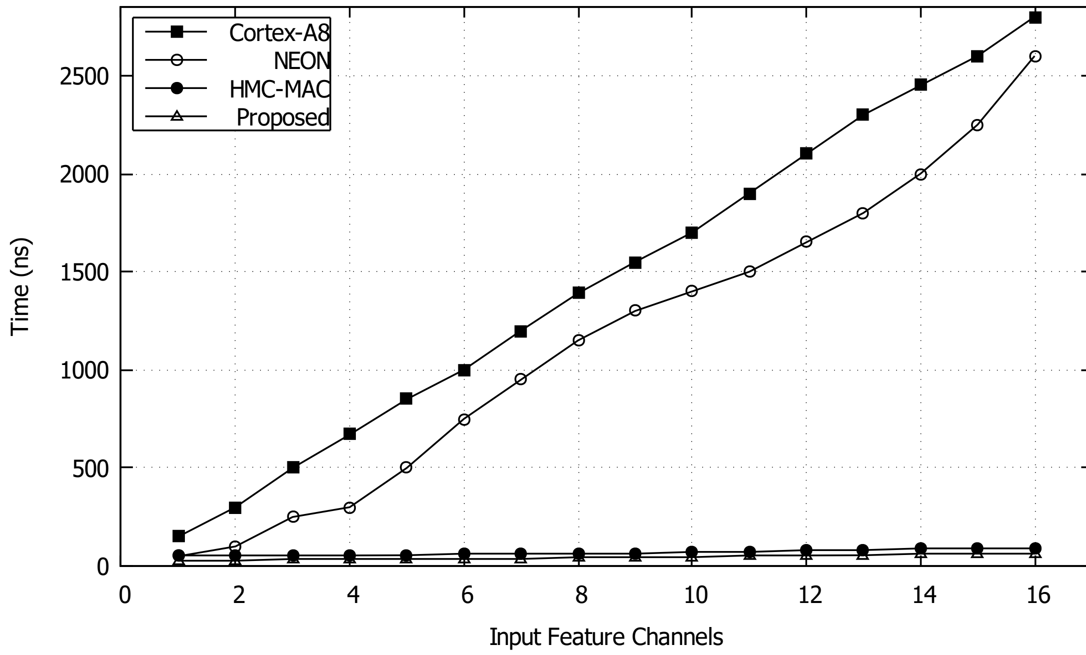
feasible since in weight-sharing deep neural network, there is only limited number of weight values.

7.4 Results and Analysis

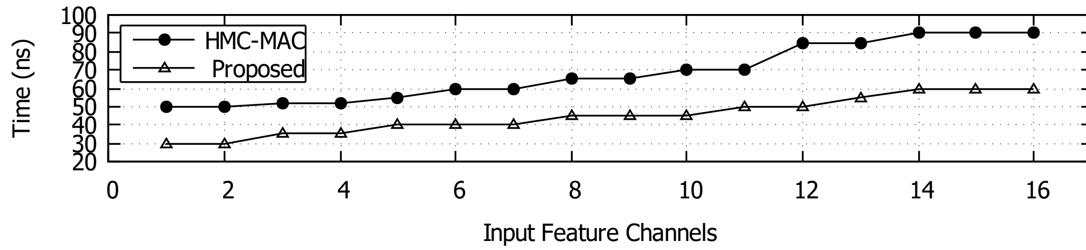
The proposed HMC architecture supporting weight-sharing deep neural network operations is implemented in CasHMC [94]. CasHMC is a cycle accurate simulator for HMC system. The authors of CasHMC also provide an example of the method to modify CasHMC in [91]. We follow the method in [91] to modify the CasHMC to realize the newly added two operations in the proposed HMC architecture. We simulate the process of using 3×3 kernels to perform weight-sharing convolution under different input channels.

The proposed HMC architecture is compared to the HMC-MAC design in [91] and two CPU-based MAC execution models (ARM Cortex-A8 and ARM Cortex-A8 with NEON). These architectures support MAC operations and are used to perform the 3×3 convolution operation under different input channels in a conventional computation method (non weight-sharing, and 1 more input channel will lead to 9 more MAC operations). The simulation of the ARM architecture is done with gem5 simulator [95].

The simulation results of performing convolutions in different platforms are shown in Fig. 7.4. We simulate the timing performance when the number of input channels ranging from 1 to 16. In conventional convolution operation method, by using 3×3 kernel, each input channel will need 9 MAC operations. Therefore, the total number of MAC operations is equal to 9 times the number of input channels. For the CPU-based MAC operation (Cortex-A8 and NEON), the timing has a significant increase with the increase of the number of MAC operations. In weight-sharing method, with more input channels, the number of accumulation is increased. However, the number of multiplication is always a constant. In addition, the accumulation of data is performed in parallel in the proposed HMC architecture, therefore, with different number of input channels, the timing of the proposed HMC only has minor changes. The trend of timing of the proposed HMC is similar to that of [91] since both designs are based on parallel HMC operations. Compared to [91], due to the weight sharing operations, the number of multiplication required is significantly reduced and thus the timing



(a) Comparison with HMC and ARM based operations



(b) Enlarged details of two HMC operations

Figure 7.4: Timing of performing convolution with 3×3 kernel under different input channels

of the proposed HMC is faster than [91].

As most of the HMC simulators do not support power consumption estimation yet, the power comparison is not presented in this paper. However, we could expect that the proposed HMC architecture has smaller power consumption than [91], Cortex-A8, and NEON since both in-memory operations and weight-sharing method significantly reduce the memory interface traffic. In addition, the modification to the HMC architecture is not significant. Therefore, there will be no thermal problems.

7.5 Summary

In this chapter, a HMC architecture is proposed for weight-sharing deep convolutional neural networks. The proposed HMC architecture is modified based on the conventional HMC architecture. Two new instructions are introduced to support the accumulation of data corresponding to the same weight and to support the MAC operations between the weight and accumulated data. The HMC controller is modified to support parallel vaults operations. In addition, a simple MAC unit is integrated into vault controller to support in-memory MAC operations. The proposed HMC architecture can perform convolution by on average 30% faster than other HMC. The execution time is relatively stable under different input channels. The proposed HMC can be integrated in efficient deep learning implementation for smart systems.

Part V

Conclusion

Chapter 8
Summary and Future Work

8.1 Summary

Deep learning is able to achieve superior performance in a variety of applications. However, due to the intensive computation and high energy consumption, the cost of applying deep learning methods in applications is still expensive. To facilitate the deployment of deep learning methods in various applications, improving the speed performance and the energy efficiency of deep learning implementations becomes necessary. In this thesis, several novel fused arithmetic unit architectures are proposed in order to improve the speed and energy efficiency of the deep learning computation. All the proposed works can be divided into three categories: (1) optimizing the deep learning computation in general-purpose arithmetic units. (2) optimizing deep learning specific arithmetic units. Within the second category, the designs based on both the conventional fixed-point and floating-point number formats and the new posit format are investigated. (3) optimizing deep learning computation in 3D memory architecture.

The organization of the thesis and the deep learning computation features supported by each proposed design are graphically shown in Figure 8.1. The optimization for deep learning computation starts with the general-purpose arithmetic unit presented in Part II Chapter 3 of this thesis (denoted as MP-FMA in Figure 8.1). The MP-FMA is designed based on a general-purpose multiple-precision floating-point FMA architecture. In order to be compatible with the IEEE 754-2008 standard [8] for supporting general-purpose computation, only the standard precisions defined in IEEE 754-2008 standard [8] are supported. The MP-FMA design has four deep learning computation features: supporting floating-point computation for deep learning training and inference; supporting low precision computation to improve

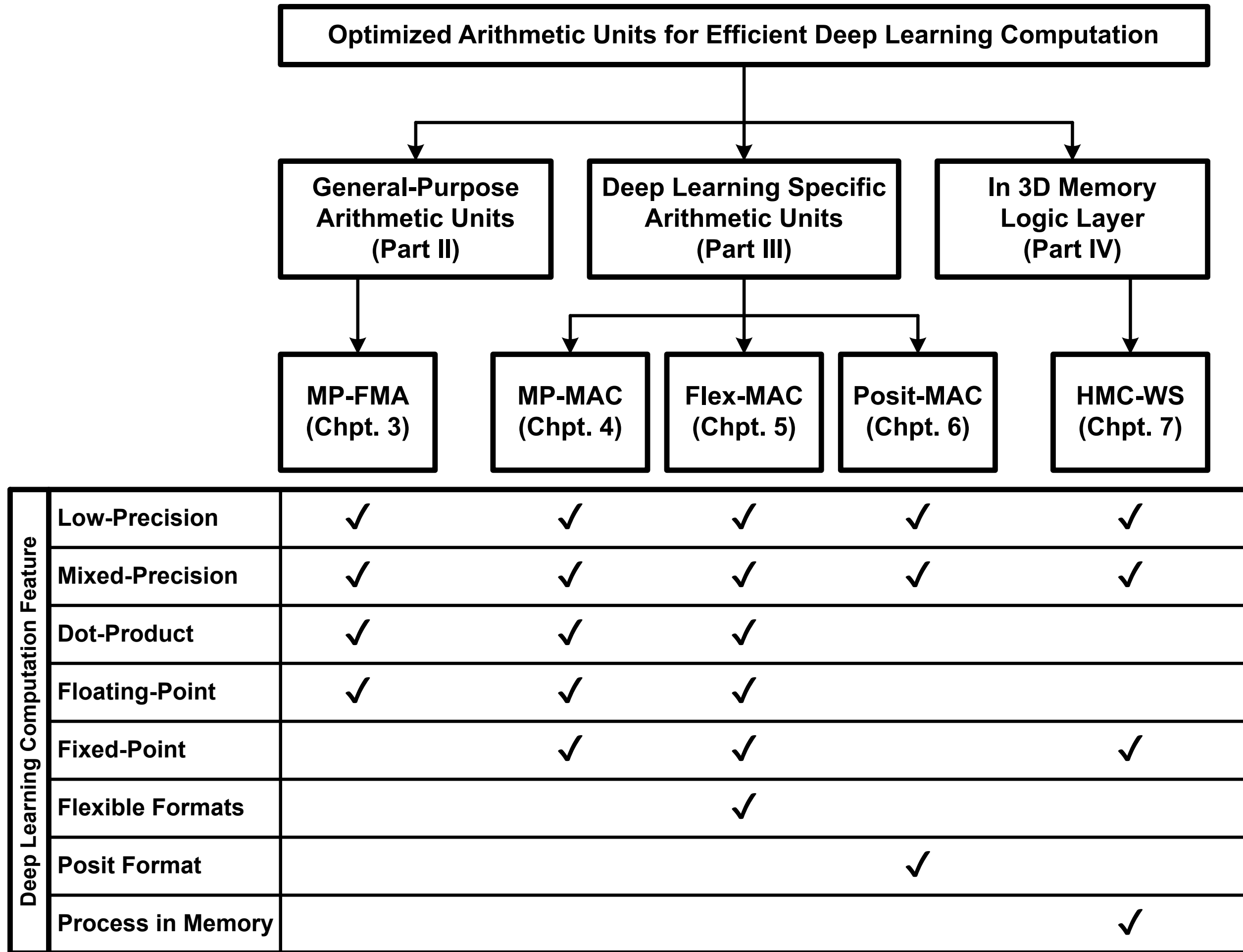


Figure 8.1: Organization of the thesis and the deep learning computation features supported by each proposed design

speed performance and to reduce energy consumption; supporting mixed-precision computation to maintain results accuracy; and supporting dot-product computation to increase deep learning computation throughput. In addition to these deep learning computation features, the proposed MP-FMA supports many other functions for various numerical precisions. The functions supported by the proposed MP-FMA and their possible applications are summarized in Table 8.1.

Compared to a normal multiple-precision FMA, the proposed MP-FMA also supports mixed-precision FMA operations and mixed-precision dot-product operations with only 6.5% more area. Compared to the state-of-the-art multiple-precision FMA designs, the proposed FMA newly adds support for half-precision FMA operations and mixed-precision operations with only 10.6% more area. The proposed FMA architecture can be used in efficient processor

Table 8.1: Functions supported by the proposed MP-FMA and their applications

Precision	Operation	Throughput (op/cycle)	Application
HP	FMA/MUL/ADD	8	Deep Learning Computation
	MIX-FMA†	4	
	MIX-DOT-PRODUCT†	4	
SP	FMA/MUL/ADD	4	Graphics Processing Digital Signal Processing
	MIX-FMA†	2	
	MIX-DOT-PRODUCT†	2	
DP	FMA/MUL/ADD	2	Scientific Computation
	MIX-FMA†	1	Virtual Reality
	MIX-DOT-PRODUCT†	1	Linear Algebra Simulation
QP	FMA/MUL/ADD	1	Financial Computation

† Reported with the precision of multiplier operands. The addend uses the next higher precision format.

designs or specialized hardware accelerators. The support of parallel half-precision operations and the mixed-precision operations makes the proposed design suitable in accelerating deep learning applications.

Then for deep learning processors, three deep learning specific arithmetic units are proposed in Part III of this thesis. In this series of designs, more deep learning computation features are considered, as shown in Figure 8.1. For example, the support of both fixed-point operation and floating-point operation in a single architecture meets different precision requirements of deep learning training and inference. In Chapter 4, a fixed-point and floating-point merged mixed-precision multiply-accumulate unit (denoted as MP-MAC in Figure 8.1) is designed for deep learning processors. In this design, the floating-point operations and the fixed-point operations are merged and realized in a single architecture. Two operational modes are supported in the proposed MP-MAC design. In floating-point mode, 16-bit multiplication is supported and the product is accumulated to a 32-bit addend. In fixed-point mode, two parallel 8-bit multiplications are supported. These two products are accumulated to a 32-bit fixed-point addend in the adder stage. The functions supported by the proposed MP-MAC and their usage in deep learning applications are summarized in Table 8.2. Compared to a half-precision multiply-accumulate unit (accumulating to single-precision), the proposed architecture has only 4.6% area overhead. With the proposed MP-MAC architec-

Table 8.2: Functions supported by the proposed MP-MAC and their usage

Mode	Equation	Operand Bit-Width			Usage
		A_i	B_i	C	
Floating-Point	$AB + C$	16-bit	16-bit	32-bit	Deep Learning Training
Fixed-Point	$A_1B_1 + A_2B_2 + C$	8-bit	8-bit	32-bit	Deep Learning Inference

ture, floating-point operations and fixed-point operations are merged in a single architecture whose area is smaller than a combination of a separate floating-point MAC and a fixed-point MAC. For deep learning computation, within the same chip area, the arithmetic unit density can be improved by using the proposed MP-MAC architecture and thus the computation throughput is expected to be improved.

In Chapter 5, an efficient flexible multiple-precision multiply-accumulate unit (denoted as Flex-MAC in Figure 8.1) is designed for deep neural network training and inference. As shown in Figure 8.1, the Flex-MAC unit includes almost all deep learning computation features using conventional numerical formats and in conventional computer architecture. Similar to the MP-MAC design presented in Chapter 4, the Flex-MAC design also supports both fixed-point operations and floating-point operations in a single architecture. However, within each number category, more numerical precisions are supported (for example 4/8/16-bit fixed-point versus 8-bit only in MP-MAC). In addition, a flexible precision support is introduced for both floating-point modes and fixed-point modes. For floating-point modes, with a constant total bit-width, the bit-width of exponent and mantissa can be mutually exchanged. For fixed-point modes, the bit-width of integer and fraction can be flexibly defined. The functions supported by the proposed Flex-MAC are shown in Table 8.3 (the same content as Table 5.1 in Chapter 5). The flexible precision support enables computations with different precisions for different deep learning models or layers. Moreover, the multiple-precision architecture increases the throughput and the energy efficiency of low precision computations. The proposed Flex-MAC provides the flexible precision computation support in hardware level. It can be used together with software tools, such as Ristretto [11] and Intel Distiller [77], to realize flexible precision deep learning training or inference.

Compared to the standard 16-bit half-precision MAC unit, the proposed Flex-MAC unit

Table 8.3: Functions supported by the proposed Flex-MAC unit

Mode	Equation	Operand Bit-Width				
		A and B			C	
		total	exponent	parallelism	total	exponent
FLP8-DOT2	$\sum_{i=1}^{2} A_iB_i + C$	8-bit	1∼6-bit	2	16-bit	1∼8-bit
FLP16-MAC	$AB + C$	16-bit	1∼8-bit	1	16-bit	1∼8-bit
		total	fraction	parallelism	total	fraction
FIX4-DOT4	$\sum_{i=1}^{4} A_iB_i + C$	4-bit	0∼4-bit	4	16-bit	0∼15-bit
FIX8-DOT2	$\sum_{i=1}^{2} A_iB_i + C$	8-bit	0∼7-bit	2	16-bit	0∼15-bit
FIX16-MAC	$AB + C$	16-bit	0∼15-bit	1	16-bit	0∼15-bit

provides more flexibility with only 21.8% area overhead. Compared to a standard 32-bit single-precision MAC unit, the proposed Flex-MAC unit requires much less hardware cost but still provides 8-bit exponent in the numerical format to maintain large dynamic range for deep learning computation. The proposed Flex-MAC architecture can be used in sever processors where various deep learning models are required to be processed. It can also be used to design neural network IP cores for FPGA devices.

The conventional numerical formats, such as the fixed-point and floating-point formats, are widely used in modern computation systems. However, for specific application, some newly proposed number format may bring better hardware efficiency. Posit format [16] is one of the recent proposed number format. Its non-uniform number encoding fits well with the deep learning data distribution. In Chapter 6, a posit multiply-accumulate unit (denoted as Posit-MAC in Figure 8.1) architecture is proposed to facilitate the use of posit number format in deep learning applications. Unlike the floating-point format, posit does not have standard number formats. Therefore, instead of designing a single architecture with specific data-path bit-width, an architecture with parameterized data-path bit-width is proposed and a Verilog code generator is implemented to generate MAC design for any posit format. The process of generating Verilog code using the proposed generator is shown in Figure 8.2.

As shown in Figure 8.2, the proposed generator is written in C language. The total bit-width and the exponent bit-width are used to configure the proposed generator. In addition, a pipeline option can be used to determine whether a pipelined design is generated.

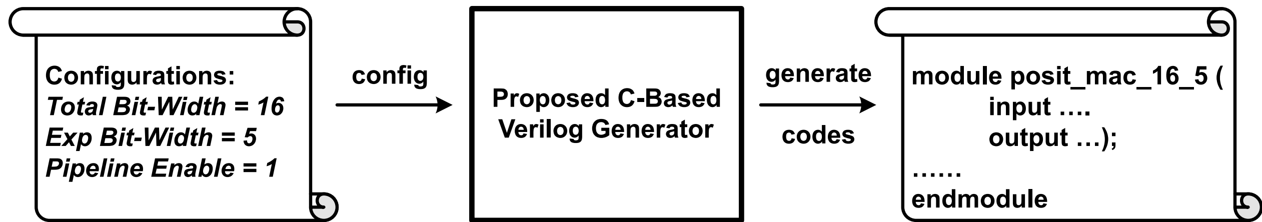


Figure 8.2: Using the proposed posit MAC generator to generate Verilog code

The Verilog HDL codes generated by the proposed generator are verified with extensive testing vectors. The delay, area, and power merits of the generated MAC units with different total bit-width and exponent bit-width are analyzed. With the same total bit-width, the hardware cost of posit MAC is larger than floating-point based MAC. However, deep learning models are expected to achieve the same level of accuracy with smaller bit-width posit format (than floating-point format). With smaller bit-width format and computation, the speed performance and energy efficiency of deep learning computation are expected to be improved.

The arithmetic units proposed in this thesis are modeled using either Verilog HDL or VHDL. Each of the proposed design is verified with extensive testing vectors using Modelsim. The synthesis is done with corresponding CMOS libraries using Synopsys Design Compiler. The power consumption merit is estimated using Synopsys PrimeTime PX. All the proposed arithmetic units are designed for ASIC platform. However, it is not difficult to map them on FPGA devices if required by some applications.

In various deep neural network accelerator or processor designs, the arithmetic unit can occupy almost 30% of the total area [96] and consume around 40% of the total energy at runtime [97] [98]. Therefore, the efficiency of the arithmetic unit is vital to the whole processor. For the proposed designs in this thesis, multiple-precision units are realized with minor area overhead compared to the standard arithmetic unit. In addition, by applying multiple-precision architecture, when low precision operations are being executed, both the throughput and the energy efficiency can be improved due to the parallel operation support. Moreover, by introducing parallel low precision operations, mixed-precision dot-product operations, and flexible precision operations, the proposed unit can realize many more functionalities than the conventional standard arithmetic unit. As the computation requirements of different deep learning models or even different layers in a model are diverse, these newly added features

enable a very flexible yet efficient deep learning computation.

Finally, the improvement of deep learning computation is explored with 3D memory architecture. In Part IV Chapter 7, a HMC architecture (denoted as HMC-WS in Figure 8.1) is proposed for weight-sharing deep convolutional neural networks. The HMC architecture has a logic base layer where logic operations can be implemented. The proposed HMC-WS is modified from the conventional HMC architecture. Two new instructions are introduced to support weight sharing accumulation and weight sharing multiply-accumulate operations. The adder included in the normal HMC architecture is used in each vault to perform accumulation. In addition, a simple MAC unit is added to each vault to perform the multiply and accumulate operation between accumulated data and weight. Moreover, in order to support parallel vaults operations, the HMC controller is modified to regenerate parallel vaults access instructions. The proposed HMC-WS architecture is simulated in CasHMC and the results show that the proposed HMC architecture can perform convolution by on average 30% faster than other HMC based designs. In conventional DRAM based system, because the memory interface bandwidth is limited, the execution time is nearly proportional to the amount of data to be processed. However, in HMC architecture, the computation is done inside the memory, and thus the execution time is relatively constant under different numbers of input channels. The proposed HMC can be integrated in artificial intelligence cores to achieve faster yet energy efficient implementation.

8.2 Future Work

In the future, more research works can be performed based on the designs presented in this thesis. Three short-term works are discussed below:

First, a flexible deep neural network accelerator can be designed based on the flexible multiply-accumulate unit presented in Chapter 5 of this thesis. The proposed flexible MAC unit will be used to build the processing elements. Correspondingly, the datapath, the controller, and the memory system are required to be modified to accommodate the flexible precision computation. When using such a flexible accelerator, the neural network model to be implemented will be compressed with the help of software tools, such as Ristretto

[11] and Intel Distiller [77]. Then the processed model can be ported to the deep neural network accelerator to perform inference operations. As the proposed flexible MAC unit also supports floating-point operations, deep neural network training can be accomplished with the proposed flexible MAC unit. In order to perform training operations, the interface between the flexible deep neural network accelerator and the deep learning frameworks needs to be developed.

Second, more operational modes can be introduced to the flexible MAC unit presented in Chapter 5 of this thesis. In the current design, although flexible precision operation is supported, the two multiplication operands, corresponding to the activation and the weight of a deep learning model, are still required to be in the same numerical format. In some recent deep learning research works, different numerical formats are applied to the activation and the weight [15]. In order to support this kind of deep learning computation, the design of 'asymmetric' arithmetic unit can be investigated. Moreover, currently all the supported operational modes in the flexible MAC unit are still performing exact computations. Approximate computation modes can be introduced along with the current computation modes to achieve better energy efficiency.

Third, a posit based deep neural network accelerator can be designed based on the posit multiply-accumulate unit proposed in Chapter 6 of this thesis. Based on the current implementation results, two major design questions are required to be solved during the design of posit based deep neural network accelerator. First is the bit-width selection of the posit numbers. According to the current analysis, posit has much larger dynamic range than the conventional floating-point numbers. Therefore, a small bit-width posit format is expected to meet the computation requirements of deep neural network. This can be explored with the help of deep learning frameworks. Second, the posit implementation has resource overhead compared to the floating-point implementation, such as the extra input and output processing. Therefore, when designing deep neural network accelerator, the systolic array architecture can be considered to achieve one input/output processing with multiple internal computations. In this way, the posit hardware overhead can be reduced.

Related research works will continue focusing on the requirements of deep learning computation. In addition to the three aspects mentioned above, some other trends of deep learning

computation will be considered, such as the efficient support for dynamic network, the support for very sparse activations, and the support for stochastic rounding [99]. Moreover, the design of novel numerical format specific for deep learning computation can be considered, such as the work in [100].

In a long-term, the design of efficient neuromorphic computation system [101] will be explored. A neuromorphic computation system is composed of many processing nodes. Each node contains a simple processor and memory blocks. All nodes communicate with each other through a router network. Neuromorphic computation system uses such a distributed structure that is able to avoid the bandwidth bottleneck between the computation cores and the memory that occurs in Von Neumann architecture. In a neuromorphic computer, each node is a light-weight neural network processor. Therefore, the proposed techniques in this thesis for efficient deep neural network computation can be applied and extended to neuromorphic computer to obtain better energy efficiency and performance.

References

[1] 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*, ser. NIPS'12, 2012, pp. 1097–1105.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv e-prints, arXiv:1512.03385*, Dec 2015.

[3] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-Level Classification of Skin Cancer with Deep Neural Networks," *Nature*, vol. 542, pp. 115–118, Feb 2017.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, pp. 484–489, Jan 2016.

[5] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[6] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org.

[7] 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*, ser. MM '14. ACM, 2014, pp. 675–678.

[8] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[9] P. Micikevicius, S. Narang, G. D. Jonah Alben, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed Precision Training," *arXiv preprint arXiv:1710.03740*, Oct 2017.

[10] B. Catanzaro, "Computer Arithmetic in Deep Learning [Keynote Talk]," in *Computer Arithmetic (ARITH), 2016 23rd IEEE Symposium on*, July 2016.

[11] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5784–5789, Nov 2018.

[12] *Deep Learning with INT8 Optimization on Xilinx Devices*, WP486 ed., Xilinx, Apr. 2017.

[13] Z. Deng, C. Xu, Q. Cai, and P. Faraboschi, "Reduced-Precision Memory Value Approximation for Deep Learning," *Hewlett Packard Labs, HPL-2015-100*, Dec 2015.

[14] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets," *arXiv e-prints arXiv:1511.05236*, Nov 2015.

[15] L. Lai, N. Suda, and V. Chandra, "Deep Convolutional Neural Network Inference with Floating-Point Weights and Fixed-Point Activations," *arXiv e-prints arXiv:1703.03073*, Mar 2017.

[16] Gustafson and Yonemoto, "Beating Floating Point at Its Own Game: Posit Arithmetic," *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, pp. 71–86, Jun 2017.

[17] "Hybrid Memory Cube Specification 2.1," *Hybrid Memory Cube Consortium*, 2015.

[18] "High Bandwidth Memory (HBM) DRAM," *JEDEC-JESD235B*, 2018.

[19] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59–70, Jan 1990.

[20] P. Micikevicius, "Mixed-Precision Training of Deep Neural Networks," Available online: https://devblogs.nvidia.com/mixed-precision-training-deep-neural-networks/.

[21] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers," in *International Conference on Supercomputing*, 2018, pp. 47:1–47:11.

[22] E. Carson and N. Higham, "Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions," *MIMS Preprint*, July 2017.

[23] M. Ortiz, A. Cristal, E. Ayguade, and M. Casas, "Low-Precision Floating-Point Schemes for Neural Network Training," *arXiv e-prints arXiv:1804.05267*, Apr 2018.

[24] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 101–108.

[25] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, “Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’17, 2017, pp. 45–54.

[26] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” *arXiv e-prints arXiv:1602.02830*, Feb 2016.

[27] H. Zhang, D. Chen, and S. Ko, “Efficient Multiple-Precision Floating-Point Fused Multiply-Add with Mixed-Precision Support,” *IEEE Transactions on Computers*, vol. 68, no. 7, pp. 1035–1048, Jul 2019.

[28] T. Lang and J. D. Bruguera, “Floating-Point Multiply-Add-Fused with Reduced Latency,” *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 988–1003, Aug 2004.

[29] J. D. Bruguera and T. Lang, “Floating-Point Fused Multiply-Add: Reduced Latency for Floating-Point Addition,” in *17th IEEE Symposium on Computer Arithmetic (ARITH’05)*, June 2005, pp. 42–51.

[30] E. Quinnell, E. E. Swartzlander, and C. Lemonds, “Floating-Point Fused Multiply-Add Architectures,” in *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, Nov 2007, pp. 331–337.

[31] P. M. Seidel, “Multiple Path IEEE Floating-Point Fused Multiply-Add,” in *2003 46th Midwest Symposium on Circuits and Systems*, vol. 3, Dec 2003, pp. 1359–1362 Vol. 3.

[32] E. Quinnell, E. E. Swartzlander, and C. Lemonds, “Bridge Floating-Point Fused Multiply-Add Design,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 12, pp. 1727–1731, Dec 2008.

[33] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, “Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems,” *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 4, pp. 457–466, Nov. 2007.

[34] K. Manolopoulos, D. Reisis, and V. A. Chouliaras, “An Efficient Dual-Mode Floating-Point Multiply-Add Fused Unit,” in *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, Dec 2010, pp. 5–8.

[35] L. Huang, L. Shen, K. Dai, and Z. Wang, “A New Architecture For Multiple-Precision Floating-Point Multiply-Add Fused Unit Design,” in *18th IEEE Symposium on Computer Arithmetic (ARITH ’07)*, June 2007, pp. 69–76.

[36] V. Arunachalam, A. N. J. Raj, N. Hampannavar, and C. Bidul, “Efficient Dual-Precision Floating-Point Fused-Multiply-Add Architecture,” *Microprocessors and Microsystems*, vol. 57, pp. 23 – 31, 2018.

[37] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, “Low-Cost Binary128 Floating-Point FMA Unit Design with SIMD Support,” *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 745–751, May 2012.

[38] K. Manolopoulos, D. Reisis, and V. Chouliaras, “An Efficient Multiple Precision Floating-Point Multiply-Add Fused Unit,” *Microelectronics Journal*, vol. 49, pp. 10 – 18, 2016.

[39] D. Ma and M. A.Saunders, *Solving Multiscale Linear Programs Using the Simplex Method in Quadruple Precision.* Springer International Publishing, 2015, pp. 223–235.

[40] P. Konsor, “Performance Benefits of Half Precision Floats,” Available online: https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats.

[41] N. Brunie, “Modified Fused Multiply and Add for Exact Low Precision Product Accumulation,” in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, July 2017, pp. 106–113.

[42] T. Trader, “How ‘Knights Mill’ Gets Its Deep Learning Flops,” Available online: https://www.hpcwire.com/2017/06/22/knights-mill-gets-deep-learning-flops/.

[43] *NVIDIA Tesla P100 Whitepaper*, WP-08019-001 ed., NVIDIA, 2016.

[44] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, “Accelerating Scientific Computations with Mixed Precision Algorithms,” *Computer Physics Communications*, vol. 180, no. 12, pp. 2526 – 2533, 2009.

[45] N. Brunie, F. de Dinechin, and B. de Dinechin, “A Mixed-Precision Fused Multiply and Add,” in *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, Nov 2011, pp. 165–169.

[46] M. Gök and M. M. Özbilen, “Multi-Functional Floating-Point MAF Designs with Dot Product Support,” *Microelectronics Journal*, vol. 39, no. 1, pp. 30 – 43, 2008.

[47] N. Higham, “The Rise of Multiprecision Arithmetic [Keynote Talk],” in *Computer Arithmetic (ARITH), 2017 24th IEEE Symposium on*, July 2017.

[48] A. Kaivani and S. Ko, “Floating-Point Butterfly Architecture Based on Binary Signed-Digit Representation,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 1208–1211, March 2016.

[49] A. D. Booth, “A Signed Binary Multiplication Technique,” *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[50] M. S. Schmookler and K. J. Nowka, “Leading Zero Anticipation and Detection-A Comparison of Methods,” in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, 2001, pp. 7–12.

[51] J. Hauser, “Berkeley TestFloat,” Available online: http://www.jhauser.us/arithmetic/TestFloat.html.

[52] A. Karatsuba and Y. Ofman, “Multiplication of Many-Digital Numbers by Automatic Computers,” in *Proceedings of the USSR Academy of Sciences*, vol. 145, 1962, pp. 293–294.

[53] H. Zhang, H. J. Lee, and S. Ko, “Efficient Fixed/Floating-Point Merged Mixed-Precision Multiply-Accumulate Unit for Deep Learning Processors,” in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[54] Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning,” *Nature*, vol. 521, pp. 436–444, May 2015.

[55] H. Zhang, D. Chen, and S. Ko, “New Flexible Multiple-Precision Multiply-Accumulate Unit for Deep Neural Network Training and Inference,” *IEEE Transactions on Computers*, pp. 1–1, 2019.

[56] D. Tan, C. E. Lemonds, and M. J. Schulte, “Low-Power Multiple-Precision Iterative Floating-Point Multiplier with SIMD Support,” *Computers, IEEE Transactions on*, vol. 58, no. 2, pp. 175–187, Feb. 2009.

[57] H. Zhang, D. Chen, and S.-B. Ko, “Area- and Power-Efficient Iterative Single/Double-Precision Merged Floating-Point Multiplier on FPGA,” *IET Computers Digital Techniques*, vol. 11, no. 4, pp. 149–158, 2017.

[58] Y. Lee, Y. Choi, S.-B. Ko, and M. Ho Lee, “Performance Analysis of Bit-Width Reduced Floating-Point Arithmetic Units in FPGAs: A Case Study of Neural Network-Based Face Detector,” *EURASIP Journal on Embedded Systems*, vol. 2009, no. 1, pp. 4:1–4:11, Jul 2009.

[59] U. Koster *et al.*, “Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks,” *arXiv e-prints arXiv:1711.02213*, Nov 2017.

[60] A. Nannarelli, “Tunable Floating-Point for Energy Efficient Accelerators,” in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, June 2018, pp. 29–36.

[61] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, “Efficient Mitchell’s Approximate Log Multipliers for Convolutional Neural Networks,” *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 660–675, May 2019.

[62] D. Miyashita, E. H. Lee, and B. Murmann, “Convolutional Neural Networks using Logarithmic Data Representation,” *arXiv e-prints, arXiv:1603.01025*, Mar 2016.

[63] A. A. Del Barrio, N. Bagherzadeh, and R. Hermida, “Ultra-Low-Power Adder Stage Design for Exascale Floating Point Units,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3s, pp. 105:1–105:24, Mar. 2014.

[64] M. Imani, D. Peroni, and T. Rosing, “CFPU: Configurable Floating Point Multiplier for Energy-Efficient Computing,” in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2017, pp. 1–6.

[65] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan 2017.

[66] L. Du, Y. Du, Y. Li, J. Su, Y. Kuan, C. Liu, and M. F. Chang, "A Reconfigurable Streaming Deep Convolutional Neural Network Accelerator for Internet of Things," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, pp. 198–208, Jan 2018.

[67] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, "Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, July 2018.

[68] G. W. Bewick, "Fast Multiplication: Algorithms and Implementation," Ph.D. dissertation, Stanford University, Feb 1994.

[69] E. M. Schwarz, M. Schmookler, and S. D. Trong, "FPU Implementations with Denormalized Numbers," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 825–836, July 2005.

[70] J. D. Bruguera and T. Lang, "Leading-One Prediction with Concurrent Position Correction," *IEEE Transactions on Computers*, vol. 48, no. 10, pp. 1083–1097, Oct 1999.

[71] 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)*, Oct 2016, pp. 1–12.

[72] Y. LeCun and C. Cortes, "MNIST Handwritten Digit Database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[73] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[74] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL™Deep Learning Accelerator on Arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, 2017, pp. 55–64.

[75] G. Tagliavini, A. Marongiu, and L. Benini, "FlexFloat: A Software Library for Transprecision Computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018.

[76] N. Ho, E. Manogaran, W. Wong, and A. Anoosheh, "Efficient Floating Point Precision Tuning for Approximate Computing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2017, pp. 63–68.

[77] Intel, "Neural Network Distiller by Intel AI Lab: A Python Package for Neural Network Compression Research." Available online: https://github.com/NervanaSystems/distiller.

[78] H. Zhang, J. He, and S. Ko, "Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2019, pp. 1–5.

[79] S. H. F. Langroudi, T. Pandit, and D. Kudithipudi, "Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit," *arXiv e-prints, arXiv:1805.08624*, May 2018.

[80] A. Podobas and S. Matsuoka, "Hardware Implementation of POSITs and Their Application in FPGAs," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 138–145.

[81] M. K. Jaiswal and H. K. So, "Architecture Generator for Type-3 Unum Posit Adder/Subtractor," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[82] J. Garland and D. Gregg, "Low Complexity Multiply Accumulate Unit for Weight-Sharing Convolutional Neural Networks," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 132–135, July 2017.

[83] "SoftPosit-Python," https://posithub.org/docs/PositTutorial_Part1.html, accessed: Oct 2018.

[84] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefevre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Springer International Publishing AG, 2018.

[85] N. Burgess and C. N. Hinds, "Design of the ARM VFP11 Divide and Square Root Synthesisable Macrocell," in *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, June 2007, pp. 87–96.

[86] H. Zhang, J. He, and S. Ko, "Improved Hybrid Memory Cube for Weight-Sharing Deep Convolutional Neural Networks," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Mar 2019, pp. 122–126.

[87] "DDR4 SDRAM STANDARD," *JEDEC79-4B*, Jun 2017.

[88] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv e-prints arXiv:1510.00149*, Oct 2015.

[89] 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," *arXiv e-prints arXiv:1602.01528*, Feb 2016.

[90] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," *arXiv e-prints arXiv:1612.00694*, Dec 2016.

[91] D. Jeon, K. Park, and K. Chung, "HMC-MAC: Processing-in Memory Architecture for Multiply-Accumulate Operations with Hybrid Memory Cube," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 5–8, Jan 2018.

[92] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv e-prints arXiv:1409.1556*, Apr 2015.

[93] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," *arXiv e-prints, arXiv:1502.02551*, Feb 2015.

[94] D. Jeon and K. Chung, "Cashmc: A cycle-accurate simulator for hybrid memory cube," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 10–13, Jan 2017.

[95] N. Binkert *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[96] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[97] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 751–764, Apr 2017.

[98] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen, "DaDianNao: A Neural Network Supercomputer," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 73–88, Jan 2017.

[99] Y. LeCun, "1.1 Deep Learning Hardware: Past, Present, and Future," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, Feb 2019, pp. 12–19.

[100] J. Johnson, "Rethinking Floating Point for Deep Learning," *arXiv e-prints arXiv:1811.01721*, Nov 2018.

[101] P. A. Merolla *et al.*, "A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.