ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

APPROXIMATE ARTIFICIAL NEURAL NETWORK HARDWARE AWARE SYNTHESIS TOOL

Ph.D. THESIS

Mohammadreza ESMALI NOJEHDEH

Department of Electronic & Communication Engineering

Electronic Engineering Programme

JULY 2021

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

APPROXIMATE ARTIFICIAL NEURAL NETWORK HARDWARE AWARE SYNTHESIS TOOL

Ph.D. THESIS

Mohammadreza ESMALI NOJEHDEH
(504162212)

Department of Electronic & Communication Engineering

Electronic Engineering Programme

Thesis Advisor: Assoc. Prof. Dr. Mustafa ALTUN

JULY 2021

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

YAKLAŞIK YAPAY SİNİR AĞI İÇIN DONANIMA DUYARLI SENTEZ ARACI

DOKTORA TEZİ

Mohammadreza ESMALI NOJEHDEH
(504162212)

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı: Assoc. Prof. Dr. Mustafa ALTUN

TEMMUZ 2021

Mohammadreza ESMALI NOJEHDEH, a Ph.D. student of ITU Graduate School student ID 504162212, successfully defended the dissertation entitled "APPROXIMATE ARTIFICIAL NEURAL NETWORK HARDWARE AWARE SYNTHESIS TOOL", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Dr. Mustafa ALTUN** Istanbul Technical University	.....
Jury Members :	**Prof. Dr. Sıddıka Berna ÖRS YALÇIN** Istanbul Technical University	.....
	Asst. Prof. Dr. Tuba AYHAN MEF University	.....
	Assoc. Prof. Dr. Burcu ERKMEN Yıldız Technical University	.....
	Asst. Prof. Dr. İsmail ÇEVİK Istanbul Technical University	.....

Date of Submission : 26 May 2021
Date of Defense : 2 July 2021

To my family,

FOREWORD

My most sincere and special gratitude goes to my supervisor Assoc. Prof. Dr. Mustafa ALTUN. Thank you for the continuous support and extraordinary supervisory to my PhD study. It was my fortune and honor to have the great supervision from you. I would also like to thank Dr. Levent AKSOY, a friend and colleague who changed my way of thinking, my attitudes to this world and my vision to the future. I also wish to thank the members of the thesis committe; Prof. Dr. Sıddıka Berna ÖRS YALÇIN and Asst. Prof. Dr. Tuba AYHAN for their comments and valuable advice. Last but by no means least, I would give my special gratitude to my families, especially my parents. They have sacrificed so much to support my life and study. I cannot express my thankful heart to them with bare language.

July 2021 Mohammadreza ESMALI NOJEHDEH

TABLE OF CONTENTS

Page

FOREWORD ..... **ix**
TABLE OF CONTENTS ..... **xi**
ABBREVIATIONS ..... **xiii**
LIST OF TABLES ..... **xv**
LIST OF FIGURES ..... **xvii**
SUMMARY ..... **xix**
ÖZET ..... **xxi**
1. INTRODUCTION **1**
2. APPROXIMATE COMPUTING **5**
2.1 Background and Preliminary Works 5
2.2 Ripple-Carry Adder Design 9
2.2.1 1-bit full adder design 9
2.2.2 n-bit ripple-carry adder design 14
2.3 Approximate Multiplier Design 21
2.3.1 Design of 1-bit approximate full adder (APFA) and half adder (APHA) 21
2.3.2 n-bit wallace-tree multiplier design 25
2.4 Experimental Results 26
2.4.1 Area, power, delay, and energy versus average error 27
2.4.2 Image processing: peak signal to noise ratio (PSNR) versus area saving 32
2.4.3 Neural network: misclassification rate versus area saving 32
3. ANN HARDWARE REALIZATION **35**
3.1 Introduction 35
3.2 Background 37
3.2.1 ANN basics 37
3.2.2 Multiplierless constant multiplications 37
3.2.3 Related work 39
3.3 Design Architectures 40
3.3.1 Parallel design 41
3.3.2 Time-Multiplexed design 41
3.3.2.1 SMAC_NEURON ARCHITECTURE 42
3.3.2.2 SMAC_ANN ARCHITECTURE 44
3.4 Finding the Minimum Quantization Value 44
3.5 ANNs Under the Shift-Adds Architecture 46
3.5.1 Multiplierless ANN design under the parallel architecture 46
3.5.2 Multiplierless ANN design under the time-multiplexed architectures 46
3.6 SIMURG: The CAD Tool 47
3.7 Experimental Results 49

4. EFFICIENT HARDWARE REALIZATION OF ANNS BY APPROXIMATE BLOCKS 55
4.1 Introduction 55
4.2 Approximate Blocks for ANN 56
4.2.1 Approximate adders 56
4.2.2 Approximate multipliers 57
4.2.3 Approximate level 59
4.2.3.1 *SMAC_NEURON* 59
4.2.3.2 *SMAC_ANN* 61
4.3 Experimental Results 62
4.3.1 Pen-digit problem 63
4.3.2 MNIST problem 69
5. CONVOLUTION LAYER 75
5.1 Introduction 75
5.1.1 Convolution layer 76
5.2 Experimental Results 81
6. CONCLUSIONS 85
REFERENCES 87
CURRICULUM VITAE 95

ABBREVIATIONS

AAL : Adders Approximation Level
ANN : Artificial Neural Network
APAD : Approximate Adder
ASIC : Application-Specific Integrated Circuit
BNN : Binary Neural Networks
BHA : Behavioral Accuracy
CAD : Computer Aided Design
CAVM : Constant Array-Vector Multiplication
Cin : Carry In
CMVM : Constant Matrix-Vector Multiplication
CMOS : Complementary Metal-Oxide Semiconductor
CNN : Convolutional Neural Networks
Cout : Carry Out
CPU : Central Processing Unit
CSD : Canonical Signed Digit
DBR : Digit-Based Recoding
EAAED : Estimated Average Absolute Error Distance
EXAD : Exact Adder
FA : Full Adder
FPGA : Field Programmable Gate Array
GPU : Graphics processing Unit
HA : Half Adder
HAC : Hardware Accuracy
HMR : Hardware Misclassification Rate
IOT : Internet of Things
LEBZAM : Least Significant Bit Zero Approximate Multiplier
LLS : Largest Left Shift value
MAC : Multiply Accumulate unit
MCM : Multiple Constant Multiplication
MLP : Multilayer Perceptrons
MR : Misclassification Rate
PBAM : Probabilistic Based Approximate Multiplier
PDP : Power-Delay Product
PSNR : Peak Signal-to Noise Ratio
RNN : Recurrent Neural Networks
SCM : Single Constant Multiplication
SLS : Smallest Left Shift value
SMAC : Single Multiply Accumulate unit
SRAM : Static Random-Access Memory
TAED : Total Absolute Error Distance

LIST OF TABLES

Page

Table 2.1 : Truth table of sample approximate adders. **7**
Table 2.2 : Truth tables of exact and approximate 1-bit adders. **13**
Table 2.3 : Calculation of E_i for example 1. ..... **17**
Table 2.4 : Values of E_i's for different APAD combinations. **17**
Table 2.5 : Synthesis of 8-bit adders. **20**
Table 2.6 : Truth table of the proposed approximate full Adder APFA. **23**
Table 2.7 : Truth table of the proposed approximate half adder APHA. **24**
Table 2.8 : 1-Bit adder results. **29**
Table 2.9 : 8-Bit adder results. **30**
Table 2.10 : 8-Bit$\times$8-Bit multiplier results. **31**
Table 2.11 : Neural network misclassification rates for different area savings. **34**
Table 3.1 : Details of ANNs on training and hardware design. **49**
Table 4.1 : SMAC_NEURON using approximate multipliers. **65**
Table 4.2 : SMAC_NEURON using approximate multipliers and adders. **66**
Table 4.3 : SMAC_ANN using approximate multipliers. **67**
Table 4.4 : SMAC_ANN using approximate multipliers and adders. **68**
Table 4.5 : SMAC_ANN for 668-128-10 structure. **72**
Table 4.6 : SMAC_NEURON for 668-128-10 structure. **73**
Table 4.7 : SMAC_NEURON for 668-256-256-256-10 structure. **74**
Table 5.1 : The *axonal-based* model data flow for convolutional computation. **79**
Table 5.2 : The proposed method data flow for convolutional computation. **80**

LIST OF FIGURES

Page

Figure 2.1 : Ripple carry adder structure. ... **7**
Figure 2.2a : APAD1. ... **12**
Figure 2.2b : APAD2. ... **12**
Figure 2.2c : APAD3. ... **12**
Figure 2.2d : APAD4. ... **12**
Figure 2.2 : Karnough Maps of APADs. ... **12**
Figure 2.3 : Demonstration of steps for example 3. ... **18**
Figure 2.4 : 4×4 bit exact wallace-tree multiplier. ... **23**
Figure 2.5a : AM3 [1]. ... **33**
Figure 2.5b : AX1 [2]. ... **33**
Figure 2.5c : INAX [3]. ... **33**
Figure 2.5d : Trunct. ... **33**
Figure 2.5e : Exact. ... **33**
Figure 2.5f : Proposed. ... **33**
Figure 2.5 : Mean filter results. ... **33**
Figure 2.6a : Mmn1 [4]. ... **33**
Figure 2.6b : Mmn2 [4]. ... **33**
Figure 2.6c : Minho [5]. ... **33**
Figure 2.6d : Trunct. ... **33**
Figure 2.6e : Exact. ... **33**
Figure 2.6f : Proposed. ... **33**
Figure 2.6 : Results for blending of two images. ... **33**
Figure 3.1 : Artificial neuron. ... **36**
Figure 3.2 : ANN with two hidden layers. ... **37**
Figure 3.3 : Single constant multiplication (SCM). ... **38**
Figure 3.4 : Multiple constant multiplication (MCM). ... **38**
Figure 3.5 : Constant array vector multiplication (CAVM). ... **39**
Figure 3.6 : Consant matrix vector multiplication (CMVM). ... **39**
Figure 3.7 : Implementation of a CMVM operation. ... **40**
Figure 3.8 : DBR method [6]. ... **41**
Figure 3.9 : The algorithm of [7] optimizing the number of operations. ... **42**
Figure 3.10 : Neuron computations at the k^{th} layer of ANN. ... **42**
Figure 3.11 : Multiply-accumulate (MAC) block in the neuron computation. ... **43**
Figure 3.12 : Neuron computations at the k^{th} layer of ANN using MAC blocks. ... **43**
Figure 3.13 : ANN design using a single MAC block. ... **45**
Figure 3.14 : Neuron computations at the k^{th} layer using a CMVM block. ... **46**
Figure 3.15 : Multiplierless realization of neuron computations. ... **47**
Figure 3.16 : ANN designs under the parallel architecture. ... **52**

Figure 3.17 : ANN designs under the SMAC_NEURON architecture. **52**
Figure 3.18 : ANN designs under the SMAC_ANN architecture..... **53**
Figure 4.1 : Ripple carry adder. **56**
Figure 4.2 : Exact 4-bit unsigned multiplier. **58**
Figure 4.3 : Approximate 4-bit unsigned multiplier. **58**
Figure 4.4 : Misclassification rate for SMAC_NEURONarchitecture..... **61**
Figure 4.5 : Misclassification rate for SMAC_ANNarchitecture. **62**
Figure 4.6 : Energy save percentages. **70**
Figure 4.7 : Area save percentages of ANN. **71**
Figure 5.1 : Multiply-accumulate (MAC) block in the neuron computation. **77**
Figure 5.2 : Axonal-based model. **78**
Figure 5.3 : Dendritic-based model. **78**
Figure 5.4 : The computation of convolutional layer. **78**
Figure 5.5 : The computation of convolutional layer. **79**
Figure 5.6 : Experimental results..... **83**

APPROXIMATE ARTIFICIAL NEURAL NETWORK HARDWARE AWARE SYNTHESIS TOOL

SUMMARY

In the previous decade, artificial neural networks (ANNS) have attracted considerable attention from researchers in many areas and have become a favorite method; from business to aerospace applications.

We live in the information age where this information feeds artificial intelligence (AI). According to Forbes' estimate, over the last two years alone 90 percent of the data in the world was generated. At first glance, processing more information may seem like a dissipation of more power in central processing units(CPUs) and graphic processing units (GPUs) or spending more time to obtain the results, but for the portable systems due to limitations in battery capacity, power, and hardware area limitations, different concerns emerge. For example, less consumption of energy is vital to extend the battery supporting time for mobile devices.

The problem starts to be bold when software engineers regardless of the hardware sources (especially for portable devices) develop different ANNs architecture, where they intend to achieve a network with the best performances. Similarly, hardware engineers' AI knowledge is limited and any change within hardware design in lack of this knowledge may yield a catastrophic defect in the expected performance. As a result, this uninformed state yields a gap between the hardware and software sides of ANNs. The emerged gap provides a pitch to hardware and software researchers to play their best performance, where more information about the rival side makes their performance more eye-catching.

By obtaining this gap, the co-design method or hardware-aware training methods become prevalent recently. The object of this dissertation is also to develop a methodology to realize the ANNs with minimum hardware cost by regarding the software performance.

Limitation in hardware cost, consumed energy, and dissipated power for devices leads designers to find new architectures and approaches. Approximate computing is one of them, where this method is an useful technique for error essence systems. By leveraging the approximate level, a trade-off between the output accuracy and hardware cost is attainable. For example, assume a 1-bit exact adder costs 18 transistors, and by removing 3 transistors, a new approximate adder by 15 transistors is achievable, but the new approximate adder generates inexact results when the input is $(0,0)$, and suppose that the results for the rest set of the inputs$((0,1),(1,0),(1,1))$ are correct. Therefore, the approximate adder saves 3 transistors at the cost of 1 inexact result.

Generally, approximate computing is apple of designers' eye in applications with error tolerance capability, consequently, error tolerance inherence of ANNs nominates approximate computing as a potential method to reduce the hardware complexity of ANNs. Since multipliers and adders are fundamental building blocks of ANNs, in this thesis, by introducing novel approximate multipliers and adders we replace them with exact adders and multipliers. As mentioned earlier, approximate computing is a trade-off between accuracy and hardware cost, to adjust this trade-off, we synthesized the proposed approximate blocks based on the desired error metric. Also, we proposed an equation to calculate the mean absolute error of the introduced approximate multiplier and adders. Based on our best knowledge, the proposed approximate blocks are the only ones which are synthesized based on the mean error value.

In next step, we introduced a new error metric called the approximate level to evaluate the performance of the proposed approximate blocks in ANNs. On the other hand, ANNs are made up of a lot of multipliers and adders, where the search space for the best combination of these blocks grows with the increase of bit-width or neuron numbers. To tackle this problem and by exploiting the proposed error metric, we introduce a new search algorithm to find the appropriate combination of the approximate and exact versions of the arithmetic blocks by taking into account the expected accuracy of ANNs.

Also, in this thesis we realized ANNs under different synthesis techniques to obtain the pros and cons of each approach. Since the parallel architecture requires a large area we considered the time-multiplexed architecture as the main architecture method, where computing resources are re-used in the multiply-accumulate (MAC) blocks.

As an application, the MNIST and Pen-digit database are considered. To examine the efficiency of the proposed method, various architectures and structures of ANNs are realized. Our experimental results show that exploiting the proposed approximate multipliers yields smaller area and power consumption compared to those designed using previously proposed prominent approximate multipliers. Also, according to these results, concurrent use of approximate multipliers and adders provides remarkable results in terms of hardware cost, where we obtain 60% and 40% reduction in energy consumption and occupied area of the ANN design with the same or better hardware accuracy compared to the exact adders and multipliers.

To demonstrate the proposed method's scalability, we propose an efficient method to realize a convolution layer of convolution neural networks (CNNs). Inspired by the fully-connected neural network architecture, we introduce an efficient computation approach to implement convolution operations.

YAKLAŞIK YAPAY SİNİR AĞI İÇİN DONANIMA DUYARLI SENTEZ ARACI

ÖZET

Yapay Sinir Ağları (YSA) geçtiğimiz on yılda pek çok alanda araştırmacılar ve yatırımcılar tarafından büyük ilgi görüyor. Forbes'in tahminine göre "Yalnızca son iki yılda dünyadaki verilerin yüzde 90'ının üretildiği" bilgi çağında yaşıyoruz. İlk bakışta, daha fazla bilginin işlenmesi CPU'larda ve GPU'larda daha fazla gücün harcanması veya sonuç elde etmek için daha fazla zaman harcanması gibi görünüyor, ancak taşınabilir sistemler için kısıtlı pil kapasitesi, güç ve donanım alanı sınırlamalar nedeniyle farklı endişeler ortaya çıkıyor. Bu sınırlamaları göz önünde bulundurarak tasarımcılar yeni mimarilere yönleniyorlar, mesela pil destek süresini uzatmak için daha az enerji tüketimi hayati önem taşır ve kullanılan mimari en az güç tüketecek şekilde tasarlanıyor.

Geleneksel olarak, bir devre veya sistem bloğunun tasarımında olası uygulamalar gözetilerek en kötü durum analizi yapılır ve sadece en yüksek doğruluk istenen uygulamaya göre tasarım yapılır. Uygulamaların ayrı ayrı gerektirdiği minimum işlem doğrulukları gözetilmez. Örnek vermek gerekirse, telefonlarımızda sıklıkla kullandığımız hesap makinesi ve fotoğraf iyileştirme/küçültme uygulamalarının ikisi için de aynı aritmetik mantık birimi (AMB) kullanılır. Oysaki hesap makinesi için AMB'nin hatasız işlem yapması elzemken, fotoğraf uygulaması için yüksek doğruluk şart değildir ve bu işlem yaklaşık hesaplama yapan bir AMB ile de yapılabilir. Böylece güç tüketimi önemli ölçüde azaltılabilir.

Önerilen tezde yaklaşık hesaplama yapabilen ve düşük güç tüketimli aritmetik devre blokları tasarlanacaktır ve bu bloklar yapay sınır ağlarında farklı doğrulukta kullanılmak üzere değişik mimarilerde kullanım özelliklerine sahip olacaktır.

Bir üst seviyede, yani sistem seviyesinde, tasarlanacak devre bloklarının seçilen mimaride nasıl en verimli şekilde kullanılacağının/yapılandırılacağının belirlenmesi gerekmektedir ve bu oldukça zor bir problemdir. İçerisinde n adet aritmetik işlem bloku bulunan ve her bir işlem blokunun m adet yapılandırılabilir seviyesi olan bir sistem için, optimum çözüm en kötü halde m^n değişik durumu gözetilerek bulunur. Çözüm bulmak için brute-force (kaba kuvvet) yaklaşımı kullanmak pratik limitlerin oldukça uzağındadır. Bu çalışmada, optimum çözümlere yakın çözümler bulabilen hiyerarşik bir yaklaşım geliştirilmektedir. Önerdiğimiz yaklaşım, sistemden istenen doğruluk veya kalite seviyesine göre, her bir devre blokunun sağlaması gereken doğruluk performansını belirlemektedir ve sonuç olarak sistemin güç tüketimi minimize edilmektedir.

Bu tezde hesaplama devreleri toplama ve çarpma devreleri olarak iki ayrı kolda incelenip, ilk adımda farklı güç, alan ve hata profillerine sahip temel toplama ve çarpma devreleri lojik olarak sentezlenmektedir. Bu aşamada temel toplama devresinden kasıt, 1 bitlik tam toplayıcı devresidir; temel çarpıcı devresi ise çarpıcı mimarisine göre değişiklik gösterir. Aritmetik işlem devresinden beklenen hata profilini sağlayacak N bitlik toplayıcıya/çarpıcıya ise farklı temel aritmetik işlem blokları birleştirilerek ulaşılmaktadır. Birleştirme aşamasında, yeni bir algoritma önererek en düşük güç tüketimi ile istenen hatanın altında kalmayı başaran aritmetik işlem devreleri sentezlenmektedir.

Bu tezde, bir öğrenme ağı, yaklaşık aritmetik işlem devrelerinin hatalarını tolere edebilecek şekilde kurulması hedeflenmektedir. Farklı ağ mimarilerinin yaklaşık hesaplamaya duyarlılığı analiz edilerek, yaklaşık hesaplamaya uygun, değişen paradigmaları takip edebilen, kısmen veri gürültüsüne karşı dayanıklı bir öğrenme tekniği seçilecektir.

Sistemin başarımı öğrenme ağı ve algoritmasının parametrelerine (katman sayısı, öğrenme adımı vs.) hassasiyetle bağlıdır. Bu öğrenme parametrelerinin, hata profilleri belli düşük güç tüketimli aritmetik işlem devreleri ile birlikte optimize edilmesi hedeflenmektedir.

Alan ve güç verimliliğini dikkate alarak farklı toplayıcı ve çarpıcı mimarileri arasında, Ripple Carry toplayıcı ve Wallace-Tree çarpıcı, devre sentezlerinde kullanılmaktadır.

Yaklaşık Rıpple Carry toplayıcı ile ilgili çalışmaların incelenmesinde, geleneksel hatasız tasarım metodolojisine bir eğilim görüyoruz. Bu tasarımlarda n-bitlik toplayıcının toplam performansı belirlemek için 1- bitlik toplayıcının ölçümleri toplanıyor. Her ne kadar bu yöntem, güç, alan ve gecikme metriklerine geçirilse, toplam n-bitlik Ripple Carry toplayıcının ortalama veya en kötü hata durumu, toplam hata toplamdan oldukça farklı olabilir.

Bununla motive olmuş, ilk önce 1-bit tam toplayıcı "Sum" ve "Carry", bitleri birbirinin hatasın azaltarak tasarlanmış, üstelik öyle toplayıcılar tasarlanıyor ki, iki ardışık yaklaşık toplayıcı, biriktirme hataları üretemiyorlar. Örneğin, eğer bir 1-bit toplayıcı beklenen lojik 0 çıktısına yanlışlıkla 1 ürettiği durumunda, bir sonraki toplayıcının çıkışının hatasız veya hatayı düşürerek sonuç verdiğini garanti ediyoruz, yani ikinci toplayıcı beklendiği 1 sayısı yerine ya 0 yâda hatasız 1 çıkışı üretiyor.

Sonuç olarak, önerilen toplayıcılar diğer literatürdeki çalışmalara göre ayni hata kısıttı sağlayarak daha küçük devre alanı ve daha az güç tüketimi sunar. Bir Rıpple Carry toplayıcıyı uygulamak için başka bir yaklaşım, yaklaşık lojik sentezi araçlarını kullanmaktır. Bu araçlar, alanı belirli bir hata kısıtlamasıyla optimize etmek için kullanılan genel amaçlı araçlardır. Neredeyse en uygun çözümleri bulmak, geleneksel yaklaşık olmayan sentez araçlarına kıyasla çok daha fazla zamana ihtiyaç duyduğundan ve / veya geniş alan elde ettikleri için toplayıcı ve çarpıcı için uygun bir yöntem değildir. Örneğin, bu yöntemler ile alan tasarruflu bir 32-bit toplayıcısını uygularken, basit kesme yöntemi aynı hata değeri için daha çok alan tasarrufu sağlarlar.

Genel olarak toplayıcı ve çarpıcının hatasını hesaplamak için bütün giriş ihtimalleri denemek zorundayız. Bit uzunluğunu artırarak hesaplama süresinin üstel olarak artırmasını göz önünde bulundurmalıyız. Bu tez çalışmasında hata hesabı yapmak için, bir matematik denklemi geliştirilmektedir. Önerilen yöntem bit uzunluğu ile doğrusal olarak artıyor ve hata hesap yapmak suresin büyük bir ölçüde azaltılıyor. Öte yandan,

bu denklemi kullanarak değişik hatalar için yaklaşık toplayıcı üreten bir sentez metodu önerilmektedir. Sistematik sentez tekniğimiz hata hesaplamaları açısından oldukça hızlı ve aynı zamanda doğrudur. Ayrıca alan değerleri, lojik sentezi araçlarında elde edilenlerden çok daha küçüktür.

Önerilen tezde, yaklaşık toplayıcı ile birlikte, üç aşamadan oluşan Wallace-Tree çarpanlarını da inceliyoruz. Çarpıcılar için hem kısmi birikiminde hem de nihai sonuç toplanmasında yaklaşık tam ve yarı-toplayıcılar öneriyoruz.

Tasarım stratejimiz, 1-bit toplayıcıların girdi atamalarının ortaya çıkma ihtimallerine dayanmaktadır yanı daha düşük olasılıkları olan giriş atamaları için daha yüksek hata oranları belirleriz. Örnek olarak, iki girişi olan bir devreyi düşünün ve iki farklı senaryoyu düşünün. İlk önce, her iki giriş de 1 ve 0 değerini, 1/2 eşit olasılıkla alır. İkinci senaryoda, girişler 1/4 ve 3/4 olasılıklarla sırasıyla 1 ve 0 değerini alır. Bu iki senaryoda, belirli bir hata sınırlaması için alan optimizasyon tekniklerinin farklı olması gerektiğini yorumluyoruz. Her girdi atamasına karşılık gelen bir hata, ilk senaryo için toplam hataya eşit şekilde katkıda bulunurken, ikinci senaryo için farklıdır. Bu nedenle, örneğimizde 1/4 olasılıklı girdi atamasına karşılık 3/4 olasılık için olanlardan daha hatalı çıktılara sahip oluyor. Bu da, girdilerin olasılığına dayanarak, Wallace-Tree çarpanının yapı taşları olarak tam ve yarım toplayıcı sentezlememize neden olur.

Bu çalışmada önerdiğimiz sentez tekniğini, aynı hata kısıtlamasını sağlayarak, literatürdekilere kıyasla en küçük alanı ve buna bağlı olarak güç tüketimini sunmaktadır. Ayrıca, sistematik sentez tekniğimiz hata hesaplamaları açısından oldukça hızlı ve hata hesaplaması kesin doğrudur.

Bu tezde, her tekniğin artılarını ve eksilerini elde etmek için farklı sentez teknikleri altında yapay sınır ağları gerçekleştirilmektedir. Paralel bir mimaride geniş alan gereksinimi nedeniyle, YSA'lar bu çalışmada, tekrarlanan çoğaltma biriktirme (ÇB) blokları ile gerçekleştirilmektedir. Uygulama olarak MNIST ve pendigit veri tabanları deneyimlenmektedir. Verimliliği incelemek için Önerilen yöntemle YSA'nın çeşitli mimarileri ve yapıları gerçekleştirilmiştir. Deneysel sonuçlar, önerilen yaklaşık çarpıcılar kullanılarak tasarlanan YSA'ların diğer yaklaşık çarpıcılara göre daha küçük bir alana sahip ve daha az enerjı tükettiğini göstermektedir.

Hem yaklaşık toplayıcılar ve hem de yaklaşık çarpıcılar kullanarak doğru çalışan devreye göre, alanda ve enerji tüketiminde sırasıyla50% ve 60% 'a varan azalma sağladığı gösterilmektedir.

Bu çalışmanın sonunda YSA'dan esinlenerek, Konvolüsyon işlemi gerçekleştirilmektedir. Önerilen yöntemde, clocklama yöntemini değiştirerek sonuçlar daha kısa zamanda elde edinmektedir. Önerilen yöntemin etkinliğini değerlendirmek için, filtreler 3×3, 5×5 ve 7×7 boyutlarla denenmektedirler. Filtrelerin girişleri 28×28 piksel olarak MNIST datasetinden alınmaktadır. Deneysel sonuçlara göre, önerilen metodu kullanarak bekleme suresinde 50% azalma ve güç tüketiminde 97% tasarruf görünmektedir.

1. INTRODUCTION

In recent years, artificial neural networks (ANNs) have achieved significant fame in different research areas, including medical image processing [8], face detection [9], and semantic segmentation [10]. Complementary, progression in graphics processing units (GPUs) and central processing units (CPUs), cause calculate on the immense hardware resources like clouds or supercomputers to become prevalent. However, for portable devices, due to their limited memory, the number of processing units, and the battery capacity, the realization of ANNs in these devices is impractical. Here, the main concern for this work arises.

An investigation of ANNs complexity reduction within the literature shows that studies are categorized in software level and hardware level commonly. Some valuable studies provide a survey of topic progressing [11, 12]. At the software level, apart from hardware consideration, the determination of ANNs structure during the training process is intended to obtain a network with minimum parameters. On the other hand, at the hardware level, distinct from the software side, different techniques are employed for reducing the hardware cost of bulky ANNs. Consequently, training based on devoted hardware, and applicable hardware modeling through the software, provide helpful results to diminish ANNs complexity.

At the software level, [13] provides a theoretical analysis of quantization error. In this study, by focusing on the derivation of finite precession error analysis techniques, the minimum bit number for forward retrieving and back-propagation is calculated. Binary weight network and XNOR networks are proposed in [14]. These two approximations are exploited to realize the standard convolution neural networks. Logarithmic computation concept is presented in [15, 16]. This encoding method enables ANNs to eliminate bulky digital multipliers. Determining logarithmic values for weights during the training process, aides to replacing of digital multipliers by shift operations with acknowledging that the multiplicands are constant in power-two numbers. By considering that ANNs consist of multiplication of different matrices, optimizing

the loops is another approach to accelerating network [17], where optimum sharing of these partial terms in the multiple constant multiplications, reduces hardware complexity [18, 19]. Beyond synthesis methods, other approaches like stochastic and approximate neural networks are common in literature. Applying stochastic computational units in neural networks results in error maintains within 10 percent of floating-point implementation [20]. The accuracy of stochastic computation may not be comparable with the conventional method but, low circuit area and power consumption make this method favorable for hardware implementation.

At the hardware level, different field-programmable gate array (FPGA) and application-specific integrated circuit (ASIC) circuits are investigated for accelerating network. To overcome the problem of memory access in large ANNs, a custom multi-chip machine-learning architecture is introduced in [21]. A specialized chip consists of a microcontroller, accelerator, and on-chip SRAM is introduced for always-on subsystems of mobile/Internet of Things (IoT) devices in [22]. Also, apart from customized chips, different hardware architectures by focusing on arithmetic operations are exploited to hinder the bulky area problem of ANNs. ANNs realization under multiplier accumulated units (MAC) is an approach to reduce hardware occupied area and power consumption by considering an increase in delay.

According to MAC-based implementation, ANN hardware structure can classify into two models: axonal-based [23] and dendritic-based [24] models. For axonalbased model, every single input of layers is multiplied by related weights of all neurons of the layer, and all outputs calculate simultaneously, as a result, for axonal-based model obtaining all inputs at the same time is unnecessary. However, for accumulating different multiplication results, extra memory is essential. On the other hand, in dendritic-based model, the value of the next neuron is calculated by multiplying all inputs with related neuron weights and accumulating them. This method results in the sequential generation of outputs, and every step of calculation needs to obtain all inputs to start. In [25], by combining these two architecture, parallel computing is enabled in two successive layers to achieve smaller latency in the computing time of the whole network.

Since the multiplier is a core block of MAC, they dominate calculation time, so designing the multipliers has become an important consideration. Conventional

multipliers consist of an array of Full Adders (FA) to adding partial products and final adders. Exploiting Wallace tree structure with different compressor leads to delay reduction in multipliers.

Based on the error-tolerant inherency of neural networks, approximate neural networks or ANNs with approximate blocks are a favorable approach to realize ANNs, where the trade-off between hardware complexity and accuracy is explored through the approximate level. Approximation for both computation and memory access is investigated in [26], also, the impact of neurons on the output quality is determined to approximate the computation and memory accesses of certain less critical neurons to obtain the maximum efficiency under a given quality constraint. The exact adders and multipliers in the MAC blocks are replaced by the approximate ones in [27]. The exploitation of approximate units yields respectively up to 64% and 43% reduction in energy and area of the ANN design for PENDIGIT data set with a slight decrease in the hardware accuracy. An evaluation of a large pool of approximate multipliers consist of 100 deliberately design, and 500 cartesian genetic programmings (CGP) based multipliers in ANNs, is accomplished in [28]. Also, to determine the critical features of multipliers in ANNs, different error parameters efficacy is investigated. According to this study, the CGP based multipliers introduced in [29] are better suited for use in the investigated ANN.

Beyond the hardware architecture, there are different methods which are related to ANNs based on the application, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs) and multilayer perceptrons (MLPs) based networks. Since our utmost concern is energy efficiency, we focus on MLPs which provide better energy scaling for applications implemented by energy-stringent stand alone devices such as always-on sensors and ASIC chips that detect anomaly [22, 30].

Each layer that comprises MLP has computation workload, which is composed of relatively basic operations, i.e., multiplication, addition, and activation. As the network is layered, arithmetic operations run in a pipeline (feed-forward) by axonal based model, where inputs of one layer wait for the outputs of the previous layer. In this thesis, ANNs are implemented using MAC blocks in two separate architectures to investigate the area and latency trade-off. A single MAC is used to realize each neuron

computation in each layer in the first one, called (*SMAC_NEURON*), and a single MAC is used to implement the entire ANN in the second one, called (*SMAC_ANN*).

Furthermore, we present an effective hardware implementation of ANNs using approximate adders and multipliers in time-multiplexed architectures, taking into account the ANN hardware accuracy. The exact adders and multipliers in the MAC blocks and parallel units are replaced with approximate adders and multipliers to form approximate ANNs. Furthermore, we present an algorithm for determining the approximate level of multipliers and adders, using the approximation level of blocks to investigate the trade-off between hardware complexity and accuracy. In contrast to the methods of [31] [29], the generation of an approximate multiplier and adder with different bit-widths of inputs under the specified approximation level in this thesis can be done in linear time. As shown in [26], by using approximate multipliers of different approximation levels for the neuron computations at different layers, the ANN hardware complexity can be greatly reduced. Our experiments show that ANNs with the proposed approximate multiplier occupy less area and consume less energy than the exact versions with only a slight loss in accuracy. It is also demonstrated that, by using the proposed approximate adders, the ANN hardware complexity can be further reduced.

The rest of this thesis is organized as follows. The approximate computing and the proposed blocks are discussed in Chapter 2. ANNs architecture is investigated in Chapter3. The exploiting of the proposed approximate blocks in the ANNs, and the algorithm to replace the approximate blocks with their exact versions are given in Chapter4. Also, a new implementation method of convolutional layers are described in Chapter5. finally, Chapter 6 concludes the thesis.

2. APPROXIMATE COMPUTING

2.1 Background and Preliminary Works

As Moore's Law starts to lose its validity, not only the number of transistors on a chip but more severely the chip's power dissipation have reached a critical point at which new circuit design techniques enabling low-power and low-area designs are highly desired [32]. Approximate computing is a class of methods that relaxes the necessity of exact equivalence between a computing system's specification and implementation. This relaxation provides an opportunity to save in design area, delay, or power dissipation at cost of inaccuracy for the calculations. allows you to trade numerical performance precision for design area, delay, or power dissipation savings. Approximate computing is used to increase area, power, and energy efficiency in applications that do not need high precision, such as image processing and learning.

Since these applications are dominated by arithmetic blocks, designing approximate adders and multipliers are extensively investigated in the literature, especially in the last decade [1, 2, 4, 5, 29, 33–37]. The implementation of approximate arithmetic blocks in logic and circuit level is also the subject of this research. Ripple-carry adders and Wallace-tree multipliers are chosen for synthesis based on their area and power efficiency among different adder and multiplier architectures, with the object of minimizing circuit area while satisfying a given error restriction.

We see a common tendency in related studies on approximate ripple-carry adders to assume that the more erroneous outputs (*Sum* and *Carry*) mean less accurate designs [1–3]. It must be considered, Offsetting errors, or errors in separate outputs that entirely or partly cancel each other, are not taken into account in this assumption. Motivated by this, we start by designing 1-bit full adders with offsetting errors in *Sum* and *Carry*. We illustrate how applying an error to output will improve accuracy and reduce the area of inexact adders. Similarly, we construct the ripple-carry adder at a higher level such that two consecutive estimated 1-bit full adders cannot generate

build-up errors, i.e., errors in separate outputs are canceling each other. For example, if a 1-bit adder generates an erroneous logic 1 output, which is supposed to be logic 0, we can guarantee that the output of the neighbor adder will be error-free or will generate an error with a logic 0 output, which is expected to be logic 1.

Consider the example of adding two 2-bit $(01)_2$ and $(10)_2$ binary numbers in Figure 2.1 to illustrate superposition, offsetting, and build-up terminology. In this example, two different approximate adders (*approxA*, *approxB*) are exploited to execute addition operation, the truth-table of these adders for evaluated binary numbers also is given in TBL.2.1. Consider that, however *Sum* and *Cout* results are inexact for the first approximate adder, still error distance is 1 for given inputs, where superposition between *Sum* and *Carry* cause to error distance be 1 for given inputs $(011)_2$ yet *Sum* and *Carry* both are inexact. The reason is incremation in *Sum* reliefs by decreasing in *Cout*, this superposition assists to minimizing *Sum* and *Cout* simultaneously. Furthermore according to *approxA* truth table, two consecutive *approxA* will not accumulate each other errors, as shown in Figure 2.1 *approxA* first bit result for $(010)_2$ inputs is $(10)_2$ (decimal 2), while exact result is $(01)_2$ (decimal 1), i.e. lesser than approximate result, correspondingly second adder inexact result for $(101)_2$ inputs is $(01)_2$ (decimal 1) lesser than exact result $(10)_2$ (decimal 2). As a result increment in first adder compensate with decrements by second adder. In essence it is not possible two successive *approxA* accumulate each other error (build up error). Contrarily for 2-bit *approxB* adders, inexact results is lesser than exact result, therefore two consecutive *approxB* adder make build-up error. *approxA* case shows, superposition can violate for adders with different error profile.

Approximate ripple-carry adders are built in two ways in the literature: 1)implement 1-bit full adders in transistor level and then creating a ripple-carry adder; 2) using synthesis tools to specifically implement a ripple-carry adder in logic level. For the first approach, approximate 1-bit adders are usually derived from standard mirror adders and XOR/XNOR based adders by eliminating transistors and/or replacing certain portions of the adders with smaller circuitries [1–3, 33]. Error dependencies in terms of offsetting and build-up errors are not taken into account in these experiments. It must be mention that, first implementation method is non-systematic, relying heavily on the designer's intuition and experiences. In the second approach, the introduced

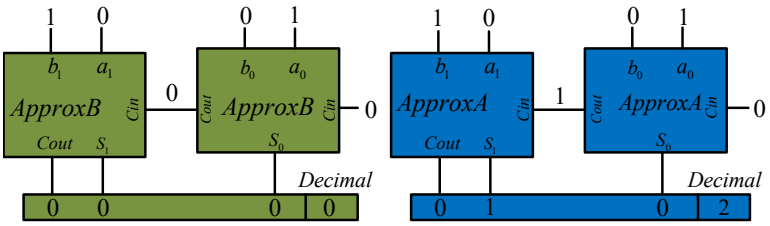

Figure 2.1 : Ripple carry adder structure.

Table 2.1 : Truth table of sample approximate adders.

Inputs			ApproxA		ApproxB	
A	B	Cin	Cout	Sum	Cout	Sum
0	1	0	0	1	0	0
1	0	1	0	1	0	1
1	0	0	1	0	0	0

tools are general-purpose tools, not strictly for ripple-carry adders [31, 38–43]. Furthermore, since determining near-optimal solutions takes far longer than traditional non-approximate synthesis methods, these tools generally suffer from long runtimes. For example, the method in [41] realizes a 32-bit ripple-carry adder with a 10% area savings, while truncation and the proposed methods in this thesis yield 25% and 32% area savings for the same worst-case error value, respectively. The run-time effects would be much worse if an average error value was used.

Motivated by the restrictions of transistor and logic level approximation methods, we suggest a systematic synthesis methodology based on a novel error measurement process. Our synthesis method is ideal for adders because it is fast, precise, and scalable. Furthermore, taking into account n-bit synthesis and the relationship between approximate adders, we can achieve a series of adders with no build-up errors. This function also helps in the formation of an n-bit adder with a different error profile in order to achieve the minimum cost for the desired error metric.

Our proposed systematic synthesis technique is both fast and reliable in terms of error calculation. For example, the synthesis of a 64-bit ripple carry adder takes less than a second. Also, the design area produced by the proposed logic synthesis tools is less when compared to the other methods. For example, given a 0.5% average error for a 32-bit adder, we achieve a 50% smaller area than [40], which can be considered the best area-efficient method in the literature.

Along with ripple-carry adders, we investigate Wallace-tree multipliers, which have three stages: partial product generation, partial product aggregation, and final result addition.

To implement the approximate multiplier, we initially investigated the different approximate multipliers in the literature to obtain the efficiency of different approaches. The approximate 2×2 multiplier is implemented in [44] to produce partial products and exact adders for the accumulation tree. A new approximate adder is recommended for product aggregation in [37], where exact adders are used to restore errors in the final results. Compressors are employed to speed up product accumulation and lessen the tree size in [37, 45]. Two approximate 4-2 compressors with four different approximate multiplier are proposed in [4]. Also, a new multiplier by error recovery is obtained by updating this compressor in [5]. Additionally, truncation and rounding techniques are used in the least significant columns of partial products in certain studies [46, 47]. Also, bit-wise multipliers are proposed in [48, 49], where detecting the leading one block saves area and power for desired error values. In [50], a reconfiguration-oriented adder is suggested. Despite the fact that reconfigurable circuits can be adjusted for various error values, they require additional control blocks, which increases their size and power consumption. These experiments are not included in this work, since we concentrate on reaching minimal area and power rather than reconfigurability.

Unlike the previous studies, we consider using an approximate full-adder and half-adder for both partial product accumulation and final result summation. The probability of input assignments is the basis for our architecture strategy, where for input assignments with lower probabilities we allocate higher error rates. As an example, consider a circuit with two inputs in two separate schemes. Inputs take the value of logic 1 and 0 with equivalent probabilities of 1/2 for the first one. In the second case, all inputs have odds of 1/4 and 3/4 for logic 1 and 0, respectively. We suggest that hardware cost optimization strategies for a given error constraint should be dissimilar for these two scenarios. Although each error associated with each input assignment contributes to the cumulative error in the first example, this is not the case for the second one. As a result, we have more erroneous outputs in our example corresponding to input assignments with 1/4 probabilities compared to the input assignments with 3/4

probabilities. This leads us to implement full and half adders based on the likelihood of inputs as the building blocks of the Wallace-tree multiplier. According to our experiments, introduced synthesis technique, as compared to those in the literature, provides the smallest area while meeting the same error restriction. Furthermore, our systematic synthesis method is both fast and reliable in terms of error calculation.

2.2 Ripple-Carry Adder Design

There are two stages to our synthesis strategies, which are described in the following two subsections. We begin by building a library of approximate 1-bit full adders with various error rates. In the second stage, we use the obtained library to systematically synthesize an n-bit ripple-carry adder from the least to the most significant bits; the adder satisfies the specified error restriction while taking up the least amount of area.

2.2.1 1-bit full adder design

The binary inputs to a 1-bit full adder are A, B, and $C(Carry\ or\ Cin)$, as well as *Cout* and *Sum* comprises the output.

The estimated and actual decimal values of the output are denoted by $\hat{y}$ and y, respectively. It's worth noting that they're in the decimal range $[0-3]$. We use total absolute error distance (TAED) as an error metric:

$$\text{TAED} = \sum_{i=0}^{7} \mid y_i - \hat{y}_i \mid \tag{2.1}$$

where i denotes the truth table's ith input assignment. For example, if both *Cout* and *Sum* are zero for all input assignments, this is truncation with zero circuit area cost, and TAED $= 12$. However, we will show that TAED $= 4$ can achieve zero cost. TAED $= 1$, TAED $= 2$, and TAED $= 3$ are also used to build adders in this thesis.

The offsetting errors in *Sum* and *Cout* play key role in our designs, and are exploited in our synthesis technique. Let's take a look at two examples to help clarify the offset concept, consider three separate approximate adders, each of which has an incorrect output for a specific input assignment. The first adder has a $0\rightarrow1$ error in *Sum*, so TAED $= 1$; the second adder has a $1\rightarrow0$ error in *Cout*, so TAED $= 2$; and the third adder, which is the proposed one, has both of the errors, so TAED

$= 1$, because simultaneous 0→1 and 1→0 errors in *Sum* and *Cout* for the same input assignment results in a change of 1 in TAED. Simultaneous error in *Sum* and *Cout* means permission has been issued to alleviate the hardware cost of both *Sum* and *Cout*. Consequently, the third approximate adder has a much smaller area than the other two.

Note that, error in *Cout* and *Sum* yields 2 and 1 change in TAED, respectively. On the other hand, simultaneous 0→1 and 1→ 0 errors occurring in *Sum* and *Cout* for the same input assignments results in a change of 1 in TAED; we name this error as the offsetting error. As a second example, consider the summing operation of $3_{(10)}$ and $10_{(10)}$ by an exact adder (EXAD) is as follows.

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ + \; 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 1 \end{array}$$

Assume the *Sum* and *Cout* results are complemented for the adder allocated to the least significant bit. The first bit is computed using the approximate adder *ApproxA*, while the remaining bits are computed using exact adders (EXADs). As a result TAED changes by one, and the error distance is one. The following diagram depicts this.

$$0 \rightarrow 1$$

EXAD ↶ APAD

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ + \; 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 1 & 0 \leftarrow 1 \end{array}$$

The suggested design approach's key concept is to apply offsetting errors 0→1 and 1→0 to the same input assignments which their results in output bits are non-identical.

According to the truth-table of one-bit exact adder in Table 2.2, 6 of 8 inputs combination have different *Sum* and *Cout* values that we will use in our approximation technique. By considering these assignments and choosing the solutions providing the lowest literal costs in sum-of-products (SOP) expressions, we present four separate approximate adders APAD1, APAD2, APAD3, and APPAD4 with TAED values of 1, 2, 3, and 4, respectively.

Table 2.2 contains the proposed approximate adders in the form of truth tables. We'll go into each adder form in detail in the sections below.

Logic synthesis of APAD1, TAED = 1: Offsetting errors in both *Sum* and *Cout* for one input assignment. There are $\binom{6}{1}$ candidates for synthesis, and 2 of them have the minimum literal cost.

Figure 2.2a shows one of them and the outputs' literal expression is as follow.

$Cout = B + ACin$
$Sum = ABCin + A\overline{B}\,\overline{Cin} + \overline{A}B\overline{Cin}$

With this solution, one of the prime implicants of *Sum* is eliminated. Moreover a prime implicant group of *Cout* is expanded. It must be mention changing in 000 and 111 results, yields minimum saving of the literals.

Logic synthesis of APAD2, TAED = 2: Offsetting errors in both *Sum* and *Cout* for two input assignments. There are $\binom{6}{2}$ candidates for synthesis, and 3 of them have the minimum literal cost.

Figure 2.2b shows one of them with following expressions.

$Cout = A$
$Sum = \overline{A}B + \overline{A}Cin + BCin$

Logic synthesis of APAD3, TAED = 3: Since we reached a literal cost of 1 for *Cout* in APAD2, no further 0-1 transitions are preferred for *Cout*. However, for *Sum* we use one more error. As a result, offsetting errors in both *Sum* and *Cout* for two input assignments, and an error in *Sum* for one input assignment return TAED = 3. There are $\binom{6}{2}\binom{4}{1}$ candidates for synthesis, and 9 of them have the minimum literal cost.

Figure 2.2c shows one of them with following expressions.

$Cout = A$
$Sum = \overline{A}Cin + B$

Logic synthesis of APAD4, TAED = 4: Similar to APAD3 synthesis, we use offsetting errors in both *Sum* and *Cout* for two input assignment. Additionally, we apply errors in *Sum* for two input assignments, so TAED = 4. There are $\binom{6}{2}\binom{4}{2}$ candidates for synthesis, and 9 of them have the minimum literal cost.

Figure 2.2d shows one of them with following expressions for outputs.

$Cout = A$

$Sum = B$

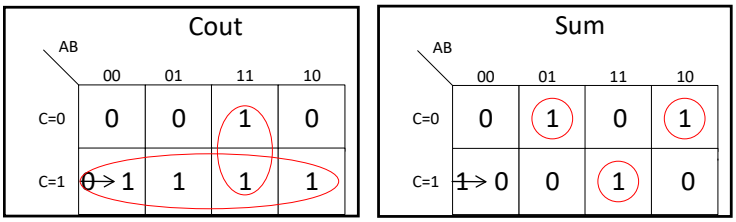

Figure 2.2a : APAD1.

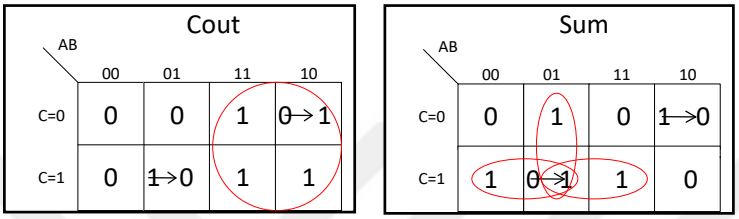

Figure 2.2b : APAD2.

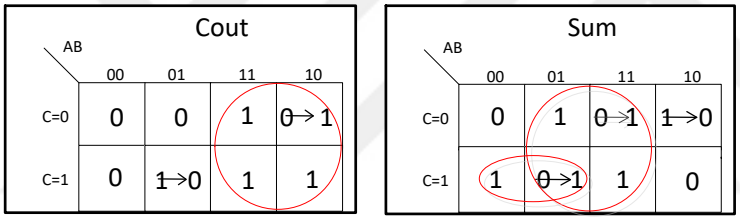

Figure 2.2c : APAD3.

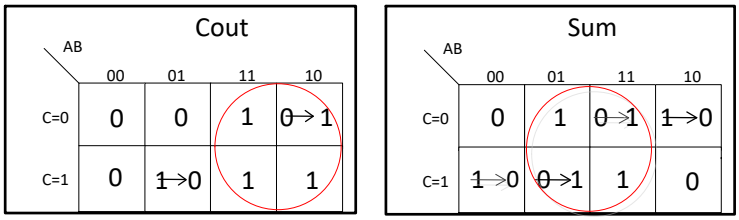

Figure 2.2d : APAD4.

Figure 2.2 : Karnough Maps of APADs.

Table 2.2 : Truth tables of exact and approximate 1-bit adders.

Inputs			Adder Type																			
			FA			APAD1				APAD2				APAD3				APAD4				
A	B	Cin	Cout	Sum	Decimal	Cout	Sum	Error	Decimal	Cout	Sum	Error	Decimal	Cout	Sum	Error	Decimal	Cout	Sum	Error	Decimal
0	0	0	0	0	0	0✓	0✓	0	0	0✓	0✓	0	0	0✓	0✓	0	0	0✓	0✓	0	0
0	0	1	0	1	1	0✓	1✓	0	1	0✓	1✓	0	1	0✓	1✓	0	1	0✓	0✗	-1	0
0	1	0	0	1	1	1✗	0✗	+1	2	0✓	1✓	0	1	0✓	1✓	0	1	0✓	1✓	0	1
0	1	1	1	0	2	1✓	0✓	0	2	0✗	1✗	-1	1	0✗	1✗	-1	1	0✗	1✗	-1	1
1	0	0	0	1	1	0✓	1✓	0	1	1✗	0✗	+1	2	1 ✗	0✗	+1	2	1✗	0✗	+1	2
1	0	1	1	0	2	1✓	0✓	0	2	1✓	0✓	0	2	1✓	0✓	0	2	1✓	0✓	0	2
1	1	0	1	0	2	1✓	0✓	0	2	1✓	0✓	0	2	1✓	1✗	+1	3	1✓	1✗	+1	3
1	1	1	1	1	3	1✓	1✓	0	3	1✓	1✓	0	3	1✓	1✓	0	3	1✓	1✓	0	3

2.2.2 n-bit ripple-carry adder design

We composed the ripple-carry adder such that no build-up errors can occur when two approximate 1-bit full adders are used in a sequence. For example, if a 1-bit adder generates an erroneous logic 1 output, which should be logic 0, we should guarantee that the output of the neighbor adder will be error-free or will generate an offsetting error with a logic 0 output, which should be logic 1. The following lemma specifies the conditions that must be met in order to avoid build-up errors.

Lemma 1. *Consider a ripple-carry adder with various 1-bit approximate adders. If and only if all of the following conditions are fulfilled, build-up errors in successive adders will be eliminated:*

1. *For all input assignments causing error, Cout* $= \overline{Cin}$*;*
2. *For all input assignments causing a positive error that increases the expected output, all corresponding Couts should be the same ; and*
3. *For all input assignments causing a negative error, decreasing the expected output, all corresponding Couts should be the same.*

Proof. The proof comes in the form of a contradiction. If a 1-bit adder has an assignment that causes an error with *Cout* = *Cin*, so repeatedly using this adder will result in build-up errors. A build-up error occurs when the second or third condition is violated even the first is met. Finally, meeting these three criteria is enough to avoid build-up errors. □

To illustrate this lemma, consider a 2-bit adder in which the least significant bit adder is APAD4 and the second adder may be any of the proposed APADs . Let's look at all of the potential inputs and their inexact outcomes. Start with 001 as an APAD4 input, which yields 0s for both *Cout* and *Sum*. According to Table 2.2, this outcome is less than the actual result, so we call it a negative error. We examine inputs with *Cin* = 0 for the second adder since *Cout* of APAD4 is *Cin* of the second adder.

According to Table 2.2 when *Cin* is 0, the generated results are exact results or positive errors. i.e if APAD4 generates negative error, it is impossible to have a positive error

for the second adder. Next, consider the other input 011 causing negative error; same considerations are applicable for this case too. For positive error cases, inputs are 100 and 110; for both cases *Cout* value is 1. According to Table 2.2, if there is a positive error, *Cin* is always 0. Therefore two consecutive positive error is impossible too.

To satisfy Lemma 1, we shrink our 1-bit approximate adder library. Initially, it consists of 2 APAD1, 3 APAD2, 9 APAD3, and 9 APAD4, and it becomes 2 APAD1, 2 APAD2, 2 APAD3, and 2 APAD4, all satisfying Lemma 1 with $Cin = 0$ for positive errors, $Cin = 1$ for negative errors, and $Cout = \overline{\text{Cin}}$. Note that for each APAD type, we have two options with identical area and error performances. For simplicity we use the adders given in Table 2.2, where all of proposed adders, satisfying Lemma 1 for all experiments.

In our synthesis technique, starting from the least to the most significant bit, we benefit the ordering of APAD4-APAD3-APAD2-APAD1-EXAD, where this array is justified with the following lemma.

Lemma 2. *Consider two successive 1-bit adders in a ripple-carry adder. To achieve minimum area with a given error constraint, the one closer to the least significant bit should have a larger or an equal TAED value compared to the one closer to the most significant bit.*

Proof. By contradiction, assume that the statement is wrong. By interchanging the two adders, we achieve a smaller error with the same area. However, the minimum area should have a negative correlation with the given error constraint. There is a contradiction. □

To understanding this lemma consider two scenarios for 2-bit adder. In the first scenario the least significant bit adder is APAD4 and the second bit adder is APAD1. In the second scenario the least significant bit adder is APAD1 and the second bit adder is APAD4. Both cases hold same area, but the first scenario results in smaller error.

In our synthesis technique, we use average absolute error distance (AAED) that is obtained with dividing TAED by the number of input assignments. For example AAED values of APAD4, APAD3, APAD2, and APAD1 are 4/8, 3/8, 2/8, and 1/8, respectively. We model AAED value and named it estimated average absolute error

distance (EAAED). For an n-bit ripple-carry adder:

$$\text{EAAED} = \sum_{i=0}^{n-1} E_i 2^{i-1} \tag{2.2}$$

where E_i represents the error contribution of the ith 1-bit adder from the least significant bit.

$$E_i = \frac{2}{3} \sum_{a \in \{-1,0,1\}} \sum_{b \in \{-1,1\}} P(i-1:a, i:b) \mid 0.5a + b \mid \tag{2.3}$$

where a and b represent the error values of the $(i-1)$th and ith 1-bit adders, respectively. Since, Equation 2.2.2 gives the error contribution of the ith adder, $b = 0$ case, no error in the ith adder, is excluded. In the equation, P represents a probability that the $(i-1)$th and ith adders have errors of a and b, respectively. The constant factor $2/3$ is the ratio of the error contribution of the ith adder (X) to the total error contribution of the ith and the $(i-1)$th adders ($0.5X + X$). In a similar fashion, $\mid 0.5a + b \mid$ represents the total error distance caused by the the $(i-1)$th and the ith adders. In calculating P's we use conditional probability such that $P(i-1:a, i:b) = P(i-1:a)P(i:b \mid i-1:a)$. The following example elucidates our calculation steps of E_i given in Equation 2.2.2.

Example 1. *Calculate E_i if the $(i-1)$th and the ith adders are both APAD4.*

Table 2.3 gives the calculations by using the truth table of APAD4, previously given in Table 2.2. There are six cases in Table 2.3 corresponding to six rows in the table for different assignments of a and b. For the first and the sixth cases P's are zero, since two successive positive error or negative error is impossible. For the second case, $P(i-1:-1) = 2/8$ since APAD4 has 2 input assignments causing -1 error among 8 total assignments. Additionally, $P(i:+1 \mid i-1:-1) = 2/4$ since -1 error causes *Cout* $= 0$ for the $(i-1)$th adder, so the ith adder's C is logic 0 and it has 2 input assignments causing $+1$ error among 4 total assignments with C $= 0$. A similar justification can be done for the fifth case. For the third and the fourth cases, $P(i-1:0) = 4/8$ and since $a = 0$, $P(i:b \mid i-1:0) = P(i:b) = 2/8$.

Table 2.3 : Calculation of E_i for example 1.

a	b	$P = P(i-1:a) \times P(i:b \mid i-1:a)$	$\mid 0.5a+b \mid$	$\sum_{b\in\{-1,1\}}$
-1	-1	$2/8 \times 0$	$3/2$	0
-1	$+1$	$2/8 \times 2/4$	$1/2$	$1/16$
0	-1	$4/8 \times 2/8$	1	$2/16$
0	$+1$	$4/8 \times 2/8$	1	$2/16$
$+1$	-1	$2/8 \times 2/4$	$1/2$	$1/16$
$+1$	$+1$	$2/8 \times 0$	$3/2$	0
$\frac{2}{3}\sum_{a\in\{-1,0,1\}}\sum_{b\in\{-1,1\}} P(i-1:a,i:b) \mid 0.5a+b \mid = 4/16$				

Table 2.4 : Values of E_i's for different APAD combinations.

i \ $i-1$	APAD1	APAD2	APAD3	APAD4
APAD1	$2.66/32$	✘	✘	✘
APAD2	$2.33/32$	$4.66/32$	✘	✘
APAD3	$2.33/32$	$4.33/32$	$6.66/32$	✘
APAD4	$2/32$	$4/32$	$6/32$	$8/32$

Table 2.4 gives E_i values for all different combinations of APAD's as the $(i-1)$th and the ith adders with satisfying Lemma 2. Since C $= 0$ for the ripple-carry first adder, we can obtain E_1 as 2/4 for APAD4 and APAD3, and 1/4 for APAD2 and APAD1 by using the truth tables in Table 2.2.

Example 2. *Calculate EAAED of an 8-bit ripple-carry adder having APAD4-APAD4-APAD4-APAD4-APAD2-APAD1-EXAD-EXAD 1-bit adders ordered from the least to the most significant bit.*

With $E_1 = 2/4$, and using Table 2.4: $EAAED = \frac{2}{4}2^0 + \frac{8}{32}(2^1+2^2+2^3) + \frac{4}{32}2^4 + \frac{2.33}{32}2^5 = 8.33$.

Constructed on Lemma 2 and the proposed error calculation method summarized in Table 2.4, our synthesis technique consists of the following 5 steps.

1. Start with an exact ripple-carry adder consisting of EXAD's.
2. From the least to the most significant bit, replace EXAD's with APAD4's until the calculated error value is larger than the given target error value.
3. Repeat the second step for APAD3, APAD2, and APAD1 instead of APAD4, respectively, replacing the unchanged EXAD's. Save the solution.

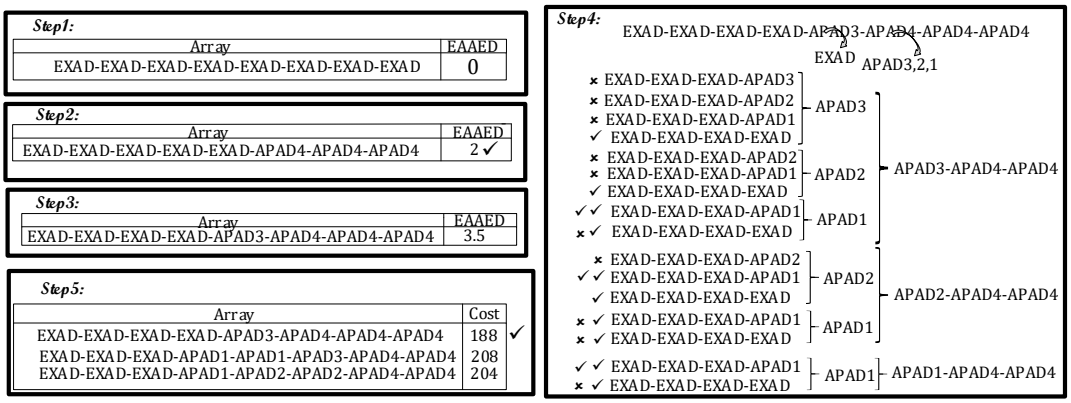

Figure 2.3 : Demonstration of steps for example 3.

4. Replace all APAD3's, APAD2's, and APAD1's with EXAD's in Step 3; replace the last APAD4 (most significant one) respectively with APAD3, APAD2, and APAD1; apply the second step with APAD3, APAD2 and APAD1 instead of APAD4. Save solutions.

5. Using area costs of APAD4, APAD3, APAD2, APAD1, and EXAD, select the best solution with minimum area cost.

Note that due to the essence of the proposed method, without using any error detection block that causes area overhead, build-up errors are fully eliminated. To elucidate our synthesis technique, we present an example.

Example 3. *With a given target AAED* $= 3.9$*, synthesize an approximate 8-bit ripple-carry adder. Suppose that the 1-bit adders are implemented with a generic library consisting of NAND2 gates (*4 *transistors) and inverters (*2 *transistors); APAD4, APAD3, APAD2, APAD1, and EXAD has transistor costs of* $0, 12, 20, 32$, *and* 44, *respectively.*

Steps are shown in Figure 2.3. In Step 4 check-marks are for satisfying given error and Lemma 2.

By examining our methodology, the importance of the second and third steps was shown. The fourth stage involves going backward to check more candidates for the minimum area. We have three solutions in the fifth stage, and the one with the smallest area cost wins. Our tests show that the first answer is usually the best. Different area costs of 1-bit adders, on the other hand, will alter this. For example, suppose the area costs for EXAD, APAD1, APAD2, APAD3, and APAD4 are $70, 45, 20, 10$, and 0,

respectively. Table 2.5 compares our synthesis technique with the exhaustive search technique for different AAED's.

Table 2.5 gives the order of approximate adders in the 8-bit ripple carry adder for different AAED values. The desired error is an upper limit for the error value. The exhaustive search is carried out by examining all approximate and exact adder combinations that match Lemma 2. Also, for exhaustive search the exact AAED values by evaluating all possible input combinations are calculated. On the other hand, the proposed method calculates the AAED value by the introduced equation . The results indicate the efficiency of the proposed method in terms of the run-time. Furthermore, estimated AAED values are almost identical to exact AAED values. There are some deviations, but this is to be expected because our calculation technique only considers the target adder and the previous adder when applying conditional probability to adder pairs. All of the previous adders should be considered in a fully precise calculation. This type of calculation, however, would result in impractical run times.

It's worth noting that different technologies may result in different APAD area sizes. Our synthesis algorithm, on the other hand, is unaffected by technology and always finds near-optimal area solutions.

In Table 2.5, run-times for the proposed and exhaustive search methods are listed. It should be noted that, thanks to the proposed simple error calculation technique, the proposed synthesis method is both fast and accurate enough when compared to other logic synthesis tools, which are typically based on try and check and suffer from run-time problems as bit length increases. In contrast, any bit number can be used in the proposed synthesis method without restriction in run-time.

Table 2.5 : Synthesis of 8-bit adders with the proposed synthesis technique and exhaustive search.

Desired Error	Proposed Method										Exhaustive Search									
	Ripple-carry								Est. AAED	Time (s)	Ripple-carry								Exact AAED	Time (s)
1.5	E	E	E	E	E	2	4	4	1.5	.02	E	E	E	E	E	2	4	4	1.5	$>10^4$
2.9	E	E	E	E	2	2	4	4	2.665	.022	E	E	E	E	2	2	4	4	2.875	$>10^4$
4.5	E	E	E	1	2	4	4	4	4.165	.016	E	E	E	1	2	4	4	4	4.46	$>10^4$
7.5	E	E	E	3	4	4	4	4	7	.02	E	E	E	3	4	4	4	4	7.125	$> 10^4$
18	E	1	2	4	4	4	4	4	16.66	.017	E	1	2	4	4	4	4	4	17.59	$> 10^4$
48	2	4	4	4	4	4	4	4	48	.016	2	4	4	4	4	4	4	4	48	$> 10^4$
64	4	4	4	4	4	4	4	4	64	.023	4	4	4	4	4	4	4	4	64	$> 10^4$

2.3 Approximate Multiplier Design

Wallace-tree multipliers are implemented in three stages using 1-bit full adders, 1-bit half adders, and AND gates. For 4-bit inputs, this is shown in Figure 2.4 . The multiplier inputs a0, a1, a2, a3 and b0, b1, b2, b3 are ANDed in the first stage. The AND gates' outputs are then fed into 1-bit adders.

The multiplier inputs take logic 1 and 0 values with equal probabilities of 1/2, the adder inputs take logic 1 and 0 values with 1/4 and 3/4 probabilities, respectively. As a result of this, we prefer erroneous outputs with 1/4 probability corresponding to the input assignments, and we develop full and half adders based on the inputs' occurrence probabilities. As a result, we never use the APADs, which were previously used for the synthesis of ripple-carry adders.

Another motivation for designing 1-bit adders is to achieve $Cout = 0$, which either converts the preceding full adder to a half adder or rules out the preceding half adder without sacrificing accuracy, i.e., obtaining 0 at the *Cout* eliminates a input of next stages. As a result, we are not using approximate full adders in the second and third stages; instead, we use approximate half adders.

We have two steps for the synthesis of an approximate multiplier. We start by building a library of approximate 1-bit full and half adders. In the second step, we use the library to systematically synthesize an n-bit Wallace-tree multiplier from the least to the most significant bits; the multiplier satisfies the given error constraint while taking up the smallest amount of space. These two steps are explained in detail in the following two subsections.

2.3.1 Design of 1-bit approximate full adder (APFA) and half adder (APHA)

Logic synthesis of APFA, TAED = 1.375:

We design a novel approximate full adder to implement in the first stage of the Wallace tree multiplier and called it APFA. We have two considerations for our design approach; obtain error by minimum probability and set the *Cout* value to 0.

Table 2.6 shows the truth table of the proposed approximate adder APFA. The offsetting error is applied to inputs by minimum occurrence probability. Also to set the *Cout* value to 0, an error in the last assignment is applied. The outputs' expressions are as follows.

$Cout = 0$

$Sum = A + B + Cin$

Also the adder's TAED value is given by:

$$\text{TAED} = \sum_{i=0}^{7} 8 \mid y_i - \hat{y}_i \mid P_i \tag{2.4}$$

where P_i denotes the probability of the ith input assignment occurring. The TAED value for APFA is calculated as follow.

$\text{TAED} = 8(3/64 + 3/64 + 3/64 + 2/64) = 1.375$

Logic synthesis of APHA, TAED = 0.25, 1.34, 0.58:

The half adders are commonly exploited in the wallace-tree architecture. Table 2.7 shows our proposed approximate half adder's truth table. The design strategies are same as the full adder's method. For the last input assignment, with a probability of $1/16$, we use offsetting errors in both *Sum* and *Cout*. The following expressions demonstrate the output of proposed approximate half adder.

$Cout = 0$

$Sum = \text{A} + \text{B}$

Also, the TAED value of an half adder is determined as follows:

$$\text{TAED} = \sum_{i=1}^{4} 4 \mid y_i - \hat{y}_i \mid P_i \tag{2.5}$$

where P_i is the possibility that the ith input assignment will occur. APHAs are used in all stages, unlike APFAs, which are only used in the first stage. As a result, as shown in Table 2.7, different P_i values exist for three distinct cases. Case 1 is exploited in first stage. When APHA's A and B inputs are connected to APFA's *Sum*, Case 2 will employ. Also, when APHA's input A is connected to an AND gate's output and APHA's input B is connected to an APFA *Sum*, the Case 3 will employ.

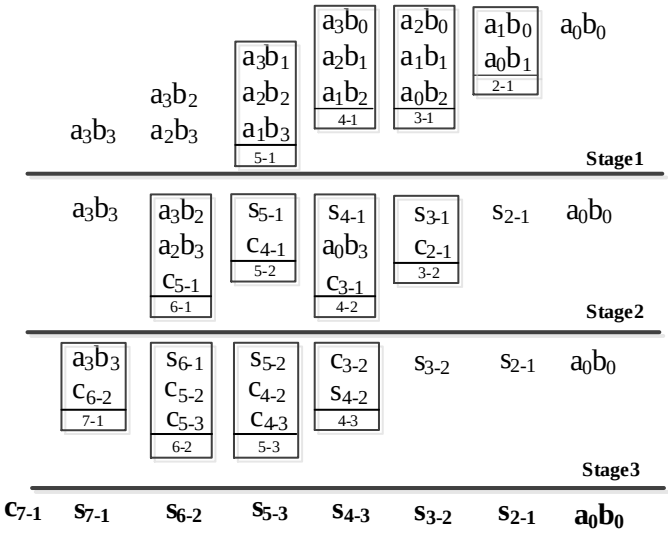

Figure 2.4 : 4×4 bit exact wallace-tree multiplier.

Table 2.6 : Truth table of the proposed approximate full Adder APFA.

Inputs			Adder Type				Error	
			EXAD		APFA			
A	B	Cin	Sum	Cout	Sum	Cout	Error	Probabiility
0	0	0	0	0	0✓	0✓	0	$3/4 \times 3/4 \times 3/4 = 27/64$
0	0	1	1	0	1✓	0✓	0	$3/4 \times 3/4 \times 1/4 = 9/64$
0	1	0	1	0	1✓	0✓	0	$3/4 \times 1/4 \times 3/4 = 9/64$
0	1	1	0	1	1✘	0✘	-1	$3/4 \times 1/4 \times 1/4 = 3/64$
1	0	0	1	0	1✓	0✓	0	$1/4 \times 3/4 \times 3/4 = 9/64$
1	0	1	0	1	1✘	0✘	-1	$1/4 \times 3/4 \times 1/4 = 3/64$
1	1	0	0	1	1✘	0✘	-1	$1/4 \times 1/4 \times 3/4 = 3/64$
1	1	1	1	1	1✓	0✘	-2	$1/4 \times 1/4 \times 1/4 = 1/64$

By considering Table 2.7, TAED values of 0.25, 1.34, and 0.58 have been determined for Cases 1, 2, and 3.

The obtained full adders and half adders will be employed as the building blocks of the approximate multipliers. The choices between different adders provide ease of mind for designers to pick up among different adders based on the application and expected errors.

Table 2.7 : Truth table of the proposed approximate half adder APHA.

Inputs		Adder Type				Error	Error		
		EXAD		APHA			Probability		
A	B	Sum	Cout	Sum	Cout	Error	Case1 (Stage1)	Case2 (Stage2-3)	Case3 (Stage2-3)
0	0	0	0	0✓	0✓	0	$3/4 \times 3/4 = 9/16$	$27/64 \times 27/64 = 729/4096$	$3/4 \times 27/64 = 81/256$
0	1	1	0	1✓	0✓	0	$3/4 \times 1/4 = 3/16$	$27/64 \times 37/64 = 999/4096$	$3/4 \times 37/64 = 111/256$
1	0	1	0	1✓	0✓	0	$1/4 \times 3/4 = 3/16$	$27/64 \times 37/64 = 999/4096$	$1/4 \times 27/64 = 27/256$
1	1	0	1	1✘	0✘	-1	$1/4 \times 1/4 = 1/16$	$37/64 \times 37/64 = 1369/4096$	$1/4 \times 37/64 = 37/256$

2.3.2 n-bit wallace-tree multiplier design

The multiplier synthesis technique is convenient to use than the ripple-carry adder, also, the error calculation method is effectively accurate. Restricted exploitation of APFA and APHA yields only negative errors and allowing us to calculate the exact error value by summing the errors. Furthermore, using APFA or APHA reduces the number of inputs of the proceeding adder by one, so only two consecutive approximate adders can be employed.

We exploit the AAED as an error metric for the synthesis of approximate multipliers. This value is obtained by dividing the TAED value by the number of input assignments. For example, the AAED of APFA is 1.375/8.

Regarding input probability and negative error inherent of APFA and APHA, the n-bit multiplier AAED value is formulated as below.

$$\mathrm{AAED} = \sum_{i}\sum_{j} \mathrm{AAED}_{i-j} 2^{i-1} \tag{2.6}$$

where AAED_{i-j} represents the error contribution of the adder in the ith column and the jth row of the Wallace-tree structure. Check that the multiplier in Figure 2.4 has 7 columns and 3 rows.

Our synthesis technique consists of the following 5 steps:

1. Start with an exact Wallace-tree multiplier.
2. Replace an exact adder having the smallest column and row numbers (column numbers are more significant) with an approximate adder. Calculate AAED.
3. Update the multiplier structure without loss of accuracy by converting full adders to half adders and/or ruling out half adders.
4. Repeat the second and the third steps until the calculated error value AAED is larger than the given target error value and store result.
5. Obtain the area cost of the multiplier by using the area costs AND gates, exact adders, and approximate adders.

To elucidate our synthesis technique, we present an example.

Example 4. *With a given target AAED* $= 1$*, synthesize an approximate* 4*-bit*$\times$4*-bit Wallace-tree multiplier. Suppose that the circuits are implemented with a generic library consisting of NAND2 gates (*4 *transistors) and inverters (*2 *transistors); AND2 gate, APHA, APFA, exact half adder, and exact full adder has transistor costs of* $6, 8, 16, 14$*, and* 44*, respectively.*

In the first step, we have an exact multiplier having 6 full adders, 6 half adders, and 16 AND2 gates as shown in Figure 2.4.

In the second step, we start with the half adder in the place 2-1, to be replaced by APHA ($\text{AAED}_{\text{2-1}} = 0.25/4$).

In the third step, we first rule out the half adder in 3-2 since $c_{\text{2-1}} = 0$, the half adder in 3-2 becomes $s_{\text{3-1}}$ that also makes $c_{\text{3-2}} = 0$. Similarly, the half adder in 4-3 is ruled out.

In the fourth step since AAED is smaller than the target error rate, we repeat the second and third steps. We replace the full adder in 3-1 with APFA ($\text{AAED}_{\text{3-1}} = 1.375/8$) that converts the full adder in 4-2 to an half adder. The total error is given by AAED $= (0.25/4)2^1 + (1.375/8)2^2 = 0.8125$. Since the next approximation in 4-1 would make AAED exceed the target error of 1, we stop here.

In the last step, since the final multiplier structure has 4 exact full adders, 3 exact half adders, 1 APFA, 1 APHA, and 16 AND2 gates, we achieve the total area cost of 338 (24% area saving).

2.4 Experimental Results

In the three subsections that follow, we evaluate the proposed adders and multipliers. Initially, we compare the proposed adders and multipliers' area, delay, power, and energy performances with those of prominent studies in the literature for the same AAED values. The image processing applications of mean filter and bit-wise multiplication operations are performed with PSNR and area saving values are reported in the second subsection. In the third subsection, an artificial neural network is used to perform a learning application that demonstrates the trade-off between area saving

and misclassification rate. All of the circuits are implemented in the same environment using the Cadence Genus tool with TSMC 0 18μm CMOS technology.

2.4.1 Area, power, delay, and energy versus average error

The hardware costs of the proposed 1-bit adder and other well-known adders in literature are given in Table 2.8. While XOR/XNOR based adders from [3] and mirror-based adders from [1] are synthesized at the transistor level, the rest of the adders including the introduced adders, are implemented under logic synthesis algorithms using standard gate libraries. We consider the best adders in terms of hardware cost among many different 1-bit adders in the literature for comparison.

Also, we considered the capability of running a consecutive blocks for investigated adders. For example, due to stability problems, the INAXA1 is not considered in this study. Only AMA3 is chosen for comparison among the adders in [1], because it performs much better than the other AMAs. The proposed APAD4 is also the same as AMA5, but the design methodology of AMAs and our method are completely different. It's worth noting that the occupied area and dissipated power for APAD4 and AMA5 are zero.

All of the proposed APADs in Table 2.8 are derived from an exact adder using the synthesis method described above. Because mirror and XOR/XNOR based exact adders are built at the transistor level with low power consideration, their area and power are inevitably smaller than those of a standard logic-level exact adder. According to this table, the proposed APADs perform better in most cases for the same TAED values.

To evaluate the efficiency of the proposed synthesis technique the proposed 8-bit ripple-carry adders hardware specifications are compared with different methods in the literature, and the results are given in Table 2.9. According to this table, for different AAED values, results for design area, dissipated power, delay time, and power-delay product (PDPg) are given.

Note that, to obtain the AAED value, we considered all combinations for the inputs (256×256). AMA3 and INAXA3 represent the performance of the transistor-level method; these adders were chosen based on the results for one-bit adders in Table 2.8.

Among the logic-synthesis approximation methods, the Evoapprox adders in [29] were chosen because the library generated in this study covers all competitive adders. Among the various adders introduced in [29], we chose adders that save the most area for a given AAED value. The results in Table 2.9 show that the proposed and Evoapprox adders come out on top, but the introduced adders in [29] are limited to 8-bit due to a long run-time problem. Our proposed adders are generally the best in terms of area; the adders are comparable in terms of the other specifications. It should be noted these findings imply that transistor-level synthesis methods are inefficient for multi-bit adder.

A similar analysis was done in Table 2.10 for the proposed multipliers compared to other Wallace-tree multipliers. Note that for the exact multipliers, compressor-based multipliers [4, 5] generally occupies less area when compared to the adder-based multipliers [29].

Based on our employed technology for synthesis of the exact version of these multipliers, the design area for [4,5] and [29] are $7348\mu m^2$ and $8097\mu m^2$, respectively. Due to the essence of the compressor-based multiplier, for the small values of error, the approximation procedure is not applicable, and hardware costs of their exact version are given instead in Table 2.10. According to Table 2.10, the proposed multipliers almost always hold the smallest design area and delay time among investigated studies.

Table 2.8 : 1-Bit adder results.

Adder Type	Results					
	Area μm^2	Delay ρs	Average Power μw	Average Energy ρws	Worst case Power μw	TAED
Exact Adder	148	1080	348	14.98	2932	0
APAD1	148	955	582	14.24	1470	1
APAD2	74	220	208	2.63	1060	2
APAD3	55	183	226	2.41	930	3
AXExact [3]	53	9600	154	602	1280	0
INXA3 [3]	44	9500	133	358	649	2
INXA2 [3]	48	2800	353	456	772	2
AMAccurate [1]	169	718	231	14.27	1680	0
AMA3 [1]	58	1200	290	13	616	3
Logic1 [39,51]	235	1050	417	18.68	2430	1
Logic2 [39,51]	84	957	291	14.1	1260	2
Logic4 [39,51]	105	970	229	13.3	1800	4
Carving [52]	105	1029	233	13.06	1800	4
BLASYS [53]	64	947	717	10.98	1086	4

Table 2.9 : 8-Bit adder results.

Adder Type	AAED																			
	0.75				2.2				3.3				6.6				11			
	Area μm^2	Power μw	Delay ηs	*PDP* aJ	Area μm^2	Power μw	Delay ηs	*PDP* aJ	Area μm^2	Power μw	Delay ηs	*PDP* aJ	Area μm^2	Power μw	Delay ηs	*PDP* aJ	Area μm^2	Power μw	Delay ηs	*PDP* aJ
AMA3 [1]	925	622	2167	13.4	746	500	2263	11.3	746	500	2263	11.3	652	412	2368	9.7	558	303	2472	7.4
INAXA3 [3]	869	594	2539	15	784	523	2853	14.9	784	523	2853	14.9	699	448	3166	14.2	615	366	3480	12.7
Evoapprox [29]	850	561	1657	9.3	630	363	1694	6.1	608	357	1148	4	458	194	1736	3.4	392	161	1566	2.5
Truncation	982	637	2459	15.6	850	560	2204	12.3	718	453	1949	8.8	718	453	1949	8.8	586	349	1694	5.9
Proposed	834	542	1912	10.3	649	400	1657	6.6	571	341	1403	4.8	439	231	1148	2.6	370	187	1275	2.4

Table 2.10 : 8-Bit×8-Bit multiplier results.

Multiplier Type	AAED																			
	1				5				10				20				50			
	Area μm^2	Power μw	Delay ηs	PDP aJ	Area μm^2	Power μw	Delay ηs	PDP aJ	Area μm^2	Power μw	Delay ηs	PDP aJ	Area μm^2	Power μw	Delay ηs	PDP aJ	Area μm^2	Power μw	Delay ηs	PDP aJ
Momeni-1 [4]	7348	4797	6765	324	7225	4871	7139	347	7148	4668	7148	334	7065	4504	7374	332	6849	4232	7052	298
Momeni-2 [4]	7348	4797	6765	324	7244	4860	7090	344	6884	4460	7051	314	6789	4431	7120	315	6313	3770	7099	268
Minho [5]	7348	4797	6765	324	7147	4976	6890	343	7147	4976	6890	343	7094	4887	7150	349	6924	4680	6980	327
Evoapprox [29]	7922	5422	9162	496	7787	5522	9422	520	7297	5153	8899	458	6498	4342	10534	457	5701	4040	8150	329
Truncation	8006	5329	6469	346	7649	5363	5529	297	7338	5029	6760	340	7022	4398	6417	282	6222	4301	5370	230
Proposed	7721	5374	5628	302	7150	4992	5729	286	6959	4858	5573	270	6485	4410	5726	252	5761	3960	5450	216

2.4.2 Image processing: peak signal to noise ratio (PSNR) versus area saving

In order to obtain adders and multipliers performance within an application, mean filter and bitwise multiplication are employed in this section. For the identical area saving value, images' PSNR values are considered as a performance metric in Figure 2.5 and Figure 2.6.

For the mean filter application, a gaussian noise with a mean zero of 0.008 is injected into a reference image. Then, the mean filter via different approximate adders is exploited to smooth the noisy image, and results are shown in Figure2.5.

All the employed approximate ripple carry adder are 8-bit, and save saves 75 % of design area compared with the exact version. For some of the approximate adders, 75% of the area is not achievable, so the maximum possible area save value is considered for them. According to the results, the proposed method posses maximum PSNR value among investigated adders.

Bit-wise multiplication is employed to evaluate the efficiency of investigated approximate multipliers. The size of multipliers is 8-bit$\times$8-bit. Also, the area save value is 40%, if this value is not achievable, the maximum possible value is considered. The results are shown in Figure 2.6, according to this result, the proposed multiplier obtains maximum performance between investigated cases.

2.4.3 Neural network: misclassification rate versus area saving

As a second application, we realized ANNs by exploiting approximate blocks. The pen-digit handwritten digit recognition problem [54] is considered to speculate the input pattern by trained networks. Pen-digit has 16 values as inputs, and the output is a number between 0 to 9, where generally models by 10 output neuron.

Our designed ANN structure is 16-100-10, the inputs are unnormalized, where the adder and multiplier input's bit-widths are 12 and 8, respectively. The area is calculated as gates number in this section, and for different area save values, the misclassification rate is given in Table2.11. The employed multipliers and adders are selected by considering their performance in Table 2.9 and Table 2.10.

Figure 2.5a : AM3 [1].

Figure 2.5b : AX1 [2].

Figure 2.5c : INAX [3].

Figure 2.5d : Trunct.

Figure 2.5e : Exact.

Figure 2.5f : Proposed.

Figure 2.5 : Mean filter results with approximate 8-Bit adders. Area saves are a)73% b)50% c)53% d)75% e)0% f)75%. PSNR values are a)14.22dB b)6.05dB c)14.22dB d)19.12dB e)NA f)37.61dB.

Figure 2.6a : Mmn1 [4].

Figure 2.6b : Mmn2 [4].

Figure 2.6c : Minho [5].

Figure 2.6d : Trunct.

Figure 2.6e : Exact.

Figure 2.6f : Proposed.

Figure 2.6 : Results for blending of two images by approximate 8-Bit×8-Bit multipliers. Area saves are a)32% b)40% c)32% d)40% e)0% f)40%. PSNR values are a)13.86dB b)13.86dB c)16.51dB d)15.10dB e)NA f)37.12dB.

Table 2.11 : Neural network misclassification rates for different area savings.

Multiplier Type	Adder Type	Area Saving				
		5%	13%	25%	34%	41%
Truncation	Truncation	3.03	3.0303	4.54	18.45	58.119
Evoapprox [29]	AMA3 [1]	3.0017	3.259	3.259	3.5163	13.52
Proposed	Proposed	2.9445	3.0017	3.0303	3.6	3.6021

3. ANN HARDWARE REALIZATION

3.1 Introduction

Recent years have seen a tremendous interest in artifical neural networks (ANNs), their successful applications in a wide range of problems, including image recognition [8] and face detection [9], their promising development on graphical processing units (GPUs) [55], and their efficient hardware implementations on different design platforms, such as analog, digital, hybrid very large scale integrated circuits (VLSI), and field programmable gate-arrays (FPGAs) [56].

An ANN is a computing system built up by a number of simple and highly interconnected processing elements [57]. As shown in Figure 3.1, its fundamental unit, called neuron, sums the multiplication of weights by input variables, adds the bias value to this summation, and propagates this result to the activation function. While the bias value has the effect of increasing or decreasing the input of the activation function, the activation function limits the amplitude of the neuron output [58]. Mathematically, the neuron behavior can be defined as following.

$$y = \sum_{i=1}^{n} w_i x_i \tag{3.1}$$

$$z = \phi(y + b) \tag{3.2}$$

Where (n) denotes the number of input variables and weights. On the other hand, Figure 3.2 presents an ANN design including hidden and output layers where each circle denotes a neuron.

Observe from Figure 3.2 that the hardware complexity of an ANN depends heavily on weight and bias values and is dominated by a large number of multiplications of constant weights by input variables. Over the years, many algorithms and design architectures have been introduced to reduce the hardware complexity of ANNs [15, 19, 25, 59–63].

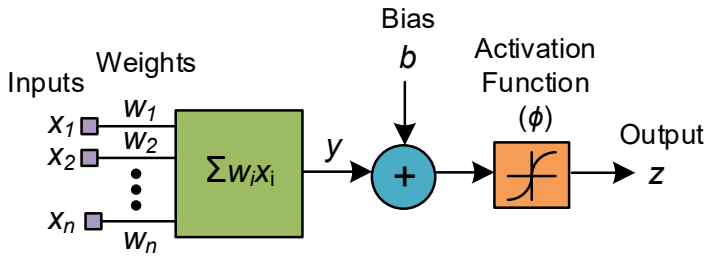

Figure 3.1 : Artificial neuron.

In this thesis, we explore the hardware complexity of ANNs under the parallel and time-multiplexed architectures. Note that a time-multiplexed design, where computations are realized at a time, re-using the computing resources, is preferred to a parallel design in applications with a strict area requirement. However, since the time-multiplexed design needs multiple clock cycles to obtain the final result, it has a higher latency and energy consumption with respect to the parallel design [64]. To further explore the area versus latency and energy consumption trade-off, in this study, we consider two time-multiplexed architectures. Furthermore, since floating-point multiplication and addition operations take up more area and energy than their integer equivalent [65], the floating-point weight and bias values observed through the training phase are transferred to integers.

Since the sizes of integer weight and bias values have a direct impact on the hardware complexity, we introduce a technique that can find the minimum quantization value, sacrificing a little loss in the hardware accuracy.

Also, for each design architecture, we propose an algorithm that can tune the weight and bias values such that the hardware complexity is reduced avoiding a loss in the hardware accuracy. Furthermore, since the ANN design includes a large number of multiplications of constant weights by input variables and these weights are determined beforehand, these constant multiplications are realized under the shift-adds architecture using the fewest number of addition/subtraction operations found by previously proposed optimization algorithms [7, 66, 67].

This chapter implies that, different design architectures present alternative ANN realizations with different hardware complexity so that a designer can choose the one that fits best in an application. Consequently, obtained trade-off provides feasibility to realize the ANNs based on the desired available hardware.

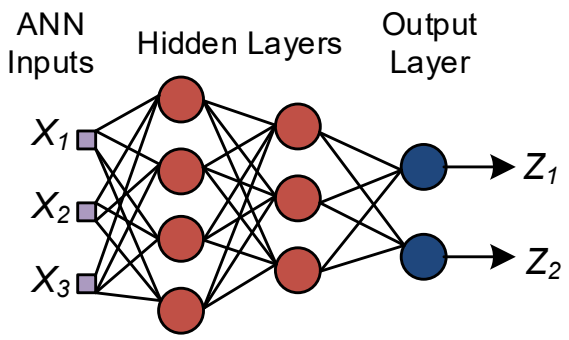

Figure 3.2 : ANN with two hidden layers.

3.2 Background

3.2.1 ANN basics

Although the design techniques presented in this article can be applied to different ANN architectures, such as convolutional and recurrent, we consider the feedforward ANNs which do not include any feedback loop. Given the ANN structure including the number of inputs, outputs, layers, and neurons in each layer and the activation functions in each layer, the weight and bias values of ANN are determined in a training phase where the error between the desired and actual values is reduced using an iterative optimization algorithm. State-of-art training algorithms [19, 68, 69] consist of efficient techniques on initialization, optimization, and stopping criteria and include a number of activation functions.

The training process is generally carried out offline on processors and/or GPUs. In the testing process, the ANN response on the applied inputs is computed using the weight and bias values determined in the training phase. The ANN computation is generally carried out online on a hardware design platform, such as application specific integrated circuits (ASIC) and FPGAs.

3.2.2 Multiplierless constant multiplications

In many applications, such as digital signal processing, cryptography, and compilers, multiplying constants by variable(s) is a common and essential operation [70]. Constant multiplications can be divided into four categories, as shown in Figure. 3.3-3.6.

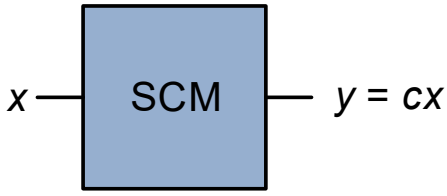

Figure 3.3 : Single constant multiplication (SCM).

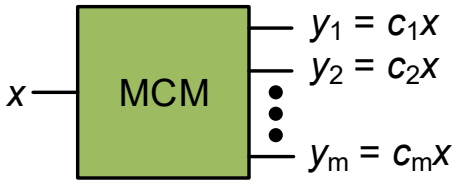

Figure 3.4 : Multiple constant multiplication (MCM).

1. The *single constant multiplication* (SCM) operation realizes the multiplication of a single constant c by a single variable x, *i.e.*, $y = cx$.

2. The *multiple constant multiplication* (MCM) operation computes the multiplication of a set of m constants C by a single variable x, *i.e.*, $y_j = c_j x$ with $1 \leq j \leq m$.

3. The *constant array-vector multiplication* (CAVM) operation implements the multiplication of a $1 \times n$ constant array C by an $n \times 1$ input vector X, *i.e.*, $y = \sum_k c_k x_k$ with $1 \leq k \leq n$.

4. The *constant matrix-vector multiplication* (CMVM) operation realizes the multiplication of an $m \times n$ constant matrix C by an $n \times 1$ input vector X, *i.e.*, $y_j = \sum_k c_{jk} x_k$ with $1 \leq j \leq m$ and $1 \leq k \leq n$.

Observe that the CMVM operation is the most general case and corresponds to an SCM operation when both m and n are 1, to an MCM operation when $m > 1$ and n is 1, and to a CAVM operation when m is 1 and $n > 1$.

Since the constants are determined beforehand, these constant multiplications can be realized using addition, subtraction, and shift operations under the shift-adds architecture. Parallel shifts can be implemented in hardware using only wires without paying any area cost. The digit-based recoding (DBR) [6] is a straightforward shift-adds design technique that can achieve constant multiplications in two steps: i) define the constants under a particular number representation, such as binary or

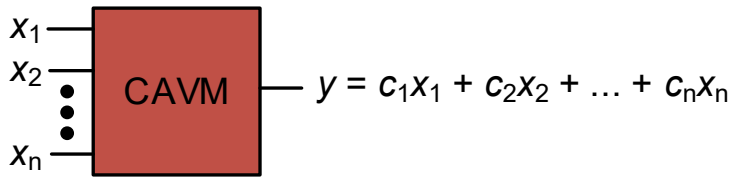

Figure 3.5 : Constant array vector multiplication (CAVM).

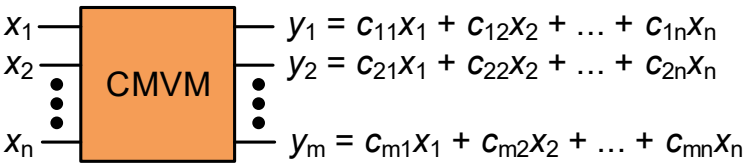

Figure 3.6 : Consant matrix vector multiplication (CMVM).

canonical signed digit (CSD)[1]; ii) for the nonzero digits in the representation of constants, shift the input variables according to digit positions and add/subtract the shifted variables with respect to digit values. As a simple example, consider the CMVM operation in Figure 3.7. Its direct realization needs 4 multiplication and 2 addition operations. The DBR method finds a solution with a total number of 8 adders and subtractors when constants are defined under the CSD representation as shown in Figure 3.8.

The number of adders/subtractors can be further reduced by maximizing the sharing of common partial products among constant multiplications [7, 66, 67, 71–73]. Returning to our example, the algorithm of [7] finds a solution with 4 operations sharing the subexpression $(x_1 + x_2)$ as shown in Figure 3.9. Moreover, prominent algorithms, that can find multiplierless designs of constant multiplications taking into account the gate-level area, delay, power dissipation, and throughput of the design, are introduced in [74–77]. Furthermore, efficient algorithms are proposed for the multiplierless realization of time-multiplexed constant multiplications in [78–80].

3.2.3 Related work

For the multiplierless realization of neural networks, binary neural networks (BNNs), where weights values and activation functions are constrained to be either 1 or -1, were introduced in [60]. It is shown that BNNs drastically reduce the memory size and the

[1] An integer can be written in CSD using n digits as $\sum_{i=0}^{n-1} d_i 2^i$, where $d_i \in \{-1, 0, 1\}$. The nonzero digits are not adjacent and a constant is represented with a minimum number of nonzero digits under CSD.

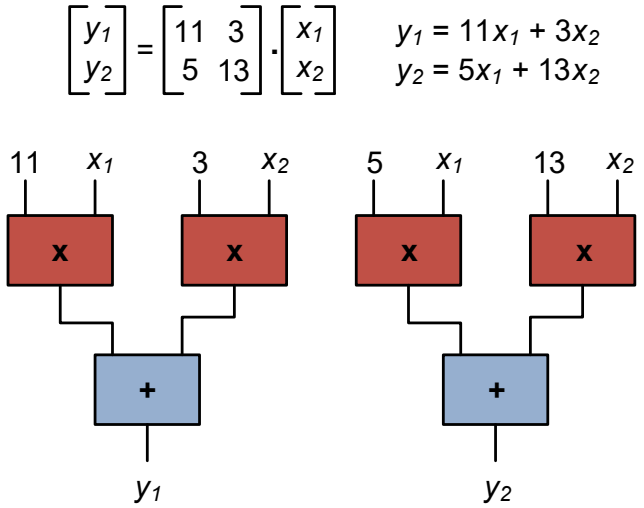

Figure 3.7 : Implementation of a CMVM operation realizing $y_1 = 11x_1 + 3x_2$ and $y_2 = 5x_1 + 13x_2$.

number of accesses to the memory during training, and replace multipliers with XOR operators in hardware. However, they lead to a worse accuracy when compared to conventional neural networks [61]. Lesser nonzero numbers mean lesser adder and subtractor; hence in [61, 62], the weights are obtained with minimum nonzero values during the training phase. In [15], floating-point weights in each layer are dynamically quantized, fixed-point weights are defined in binary representation, and the ANN is implemented in a hardware accelerator.

The multiplierless hardware realization of ANNs is considered in [63] where the multiplication of weights by input variables is realized in a bit-serial fashion, defining weights under the CSD representation.

In [19], for the time-multiplexed realization of ANN design, a post-training algorithm, that tunes weights to reduce the hardware complexity, is introduced and the multiplication of constant weights by input variables in each neuron at each layer is realized under the shift-adds architecture.

The multiply-accumulate (MAC) block is a fundamental operation in the time-multiplexed design architecture. In [81], delay-efficient MAC structure uses accumulators and carry-save adders to reduce its high latency. MAC-based implementation efficient implementation of ANN designs on FPGAs using MAC blocks is investigated in [82]. MAC blocks have recently been used to realize neuromorphic cores using two versions, axonal-based and dendritic-based [29].

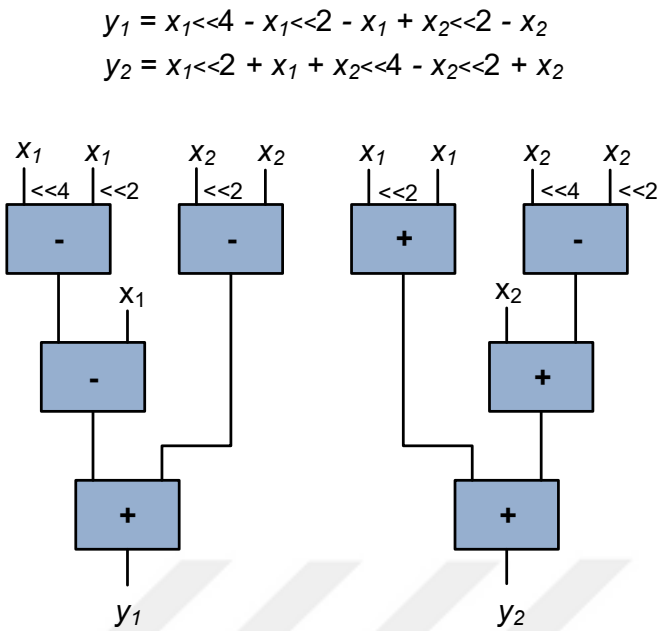

Figure 3.8 : DBR method [6].

3.3 Design Architectures

In this section, we present parallel and time-multiplexed design architectures used to realize ANNs in hardware.

3.3.1 Parallel design

Figure 3.10 presents the realization of neuron computations at the k^{th} layer where m and n are the number of outputs (or neurons) and inputs at this layer, respectively.

Under the parallel architecture, after the ANN inputs are applied, neuron computations at each layer are obtained concurrently, and the output values are obtained simultaneously, i.e, unlike to the time-multiplexed design all the outputs are calculated by one clock.

3.3.2 Time-Multiplexed design

The MAC block is a fundamental operation in an ANN design under the time-multiplexed architecture. As shown in Figure 3.11, it can be used to realize the neuron computation given in Figure 3.1, re-using the multiplication and addition operations. Note that, the control block which is actually a counter, synchronizes the multiplication of a weight by an input variable, and the result is added to the accumulated value stored in the register R. For clarity, the clock and reset signals

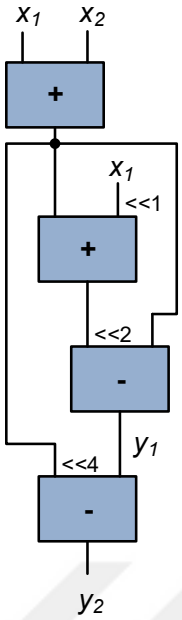

Figure 3.9 : The algorithm of [7] optimizing the number of operations.

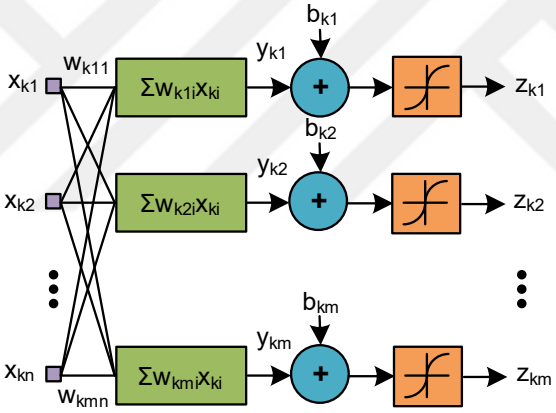

Figure 3.10 : Neuron computations at the k^{th} layer of ANN.

are not shown in this diagram. Under this architecture, the neuron computation is obtained after $n+1$ clock cycles. The size of the counter and multiplexers, which is determined by the number of weights and input variables, the size of the multiplier, which is determined by the maximum bitwidths of the input variables and weights, and the size of the adder and register, which is determined by the bitwidth of the inner product of inputs and weights, all contribute to the MAC block's design complexity, *i.e.*, $y = \sum_{i=1}^{n} w_i x_i$.

In this subsection, we present two time-multiplexed architectures to design the whole ANN using MAC blocks. Under the first architecture, called smac_neuron, each neuron at each layer is realized using a single MAC block and under the second

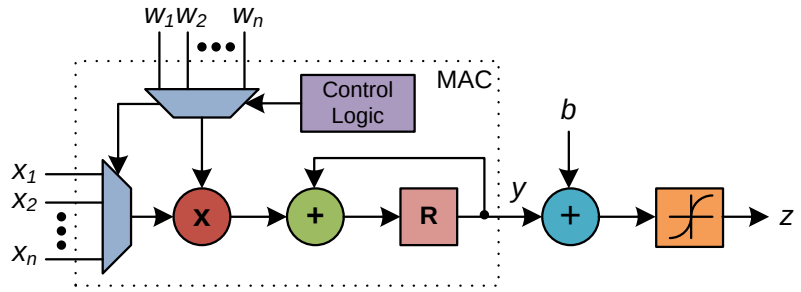

Figure 3.11 : Multiply-accumulate (MAC) block in the neuron computation.

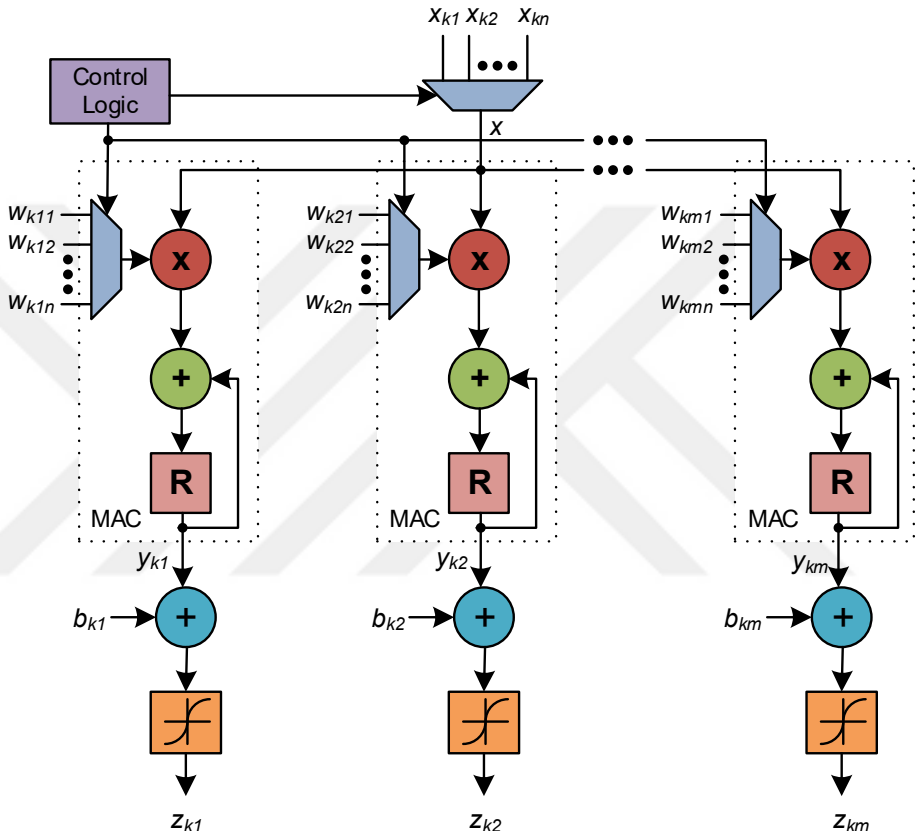

Figure 3.12 : Neuron computations at the k^{th} layer of ANN using MAC blocks.

architecture, called smac_ann, the whole ANN is realized using a single MAC block. In following, these architectures are described in detail.

3.3.2.1 SMAC_NEURON ARCHITECTURE

The neuron computations at the k^{th} layer of an ANN using m MAC blocks and a control block are shown in Figure. 3.12. The multiplication of associated weights by input variables is synchronized by the control block. If each layer of an ANN has η_i neurons, where $1 \leq i \leq \lambda$ and λ is the number of layers, the necessary number of MAC blocks is $\sum_i \eta_i$, i.e., the total number of neurons.

The number of inputs and weights determines the complexity of the operation and registers of MAC blocks.The number of inputs at each layer determines the control block's complexity. Since the neuron computations are obtained layer by layer, the neuron computations in the subsequent layer begin after the neuron computations in the previous layer are completed. This is accomplished simply by producing an output signal at each layer, which flags that, all neuron computations have been completed, thus preventing the hardware from performing excessive computations and lowering power consumption. The computation of the whole ANN with λ layers and ι_i inputs at each layer, where $1 \leq i \leq \lambda$, is obtained after $\sum_i(\iota_i + 1)$ clock cycles.

3.3.2.2 SMAC_ANN ARCHITECTURE

The ANN design using a single MAC block is demonstrated in Figure 3.13, where the clock and reset signals are omitted for clarity. The control block in this diagram contains three counters that synchronize the multiplication of a weight by an input variable, the addition of a bias value to each inner product, and the activation function. The number of layers, the number of inputs at each layer, and the number of outputs (or neurons) at each layer are all represented by these counters.

The variables $X_1, X_2, \ldots X_n$ denote the ANN's primary inputs, and these variables are multiplied by the associated weights during the first layer computations. While the maximum number of inputs at all layers determines the size of multiplexers for input variables, the total number of weight and bias values determines the size of multiplexers for weight and bias values.

The maximum bitwidth of all input variables and weights determines the size of the multiplier in the MAC block, while the maximum bitwidth of the multiplication of weights by input variables in the whole ANN determines the size of the adder and register. Furthermore, the maximum number of outputs at each layer determines the number of registers used to store the outputs. We note that the computation of the whole ANN with λ layers, ι_i inputs at each layer, and η_i neurons at each layer, where $1 \leq i \leq \lambda$, is obtained after $\sum_i(\iota_i + 2)\eta_i$ clock cycles. By considering the number of the clock cycles and neurons, we proposed an algorithm to find the minimum quantization value.

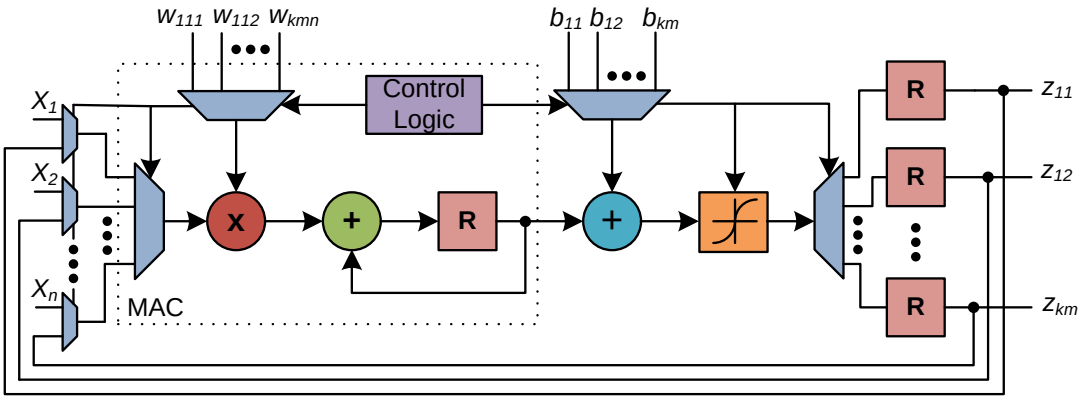

Figure 3.13 : ANN design using a single MAC block.

3.4 Finding the Minimum Quantization Value

In this section, we present a technique proposed for finding the minimum quantization value to convert the floating-point weight and bias values to integers and methods introduced for tuning weight and bias values to reduce the ANN design complexity under the parallel and time-multiplexed architectures.

As mentioned earlier MATLAB neural network tool is employed to train the desired network. By default, MATLAB stores all numeric variables as double-precision floating-point values. To decrease the design complexity, we convert the floating-points to integers. By regarding the ANN accuracy, we find the minimum bit-width of weights and biases. To do so, we first create a validation data set by randomly shifting 30% of the training data set to this set, which is then used to compute the hardware accuracy. Following is a summary of the suggested technique:

1. Set the quantization value, q, and the ANN accuracy in hardware, $ha(q)$, to 0 in both software and hardware.

2. Increase the value of q by 1.

3. Multiply each floating-point weight and bias value by 2^q and find the smallest integer greater than or equal to the result of this multiplication.

4. Using the integer weight and bias values, compute the $ha(q)$ value for the validation data set.

5. If $ha(q) > 0$ and $ha(q) - ha(q-1)$ is greater than 0.1%, go to Step 2.

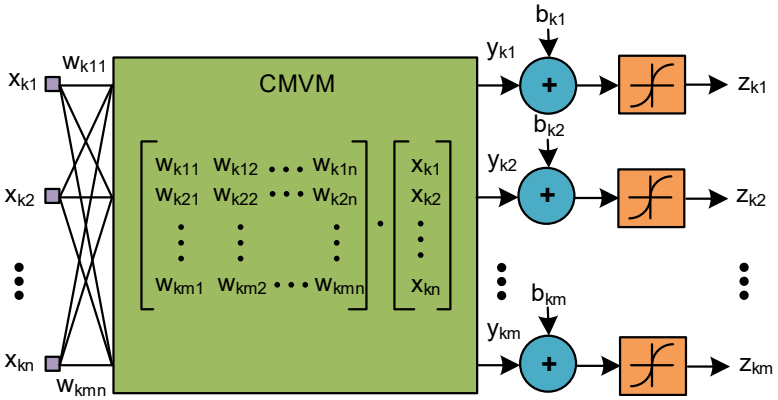

Figure 3.14 : Neuron computations at the k^{th} layer using a CMVM block.

6. Otherwise, return q as the minimum quantization value.

It's worth noting that, in order to use limited size weight and bias values, we lose only 0.1% in ANN precision in hardware computed on the validation data collection. Consider that all the preprocessing tasks must execute on the validation data set and the test date use only in the final step.

3.5 ANNs Under the Shift-Adds Architecture

This section presents the multiplierless realizations of ANN designs under the parallel and time-multiplexed architectures.

3.5.1 Multiplierless ANN design under the parallel architecture

A straight-forward way for the multiplierless realization of ANN under the parallel architecture is to describe each inner product at each layer, *i.e.*, $y_{k1}, y_{k2}, \ldots y_{kn}$ shown in Figure 3.10, as a CAVM operation and to implement each CAVM block independently under the shift-adds architecture. We use the algorithm of [67] to optimize the number of adders/subtractors in the multiplierless designs of these CAVM blocks.

As shown in Figure 3.14, all inner products at the k^{th} layer can be described as a CMVM operation and the number of adders/subtractors in the multiplierless realization of the CMVM block can be reduced using the algorithm of [7]. Thus, the possible sharing of subexpressions can be increased, reducing the number of adders and subtractors in the multiplierless ANN design.

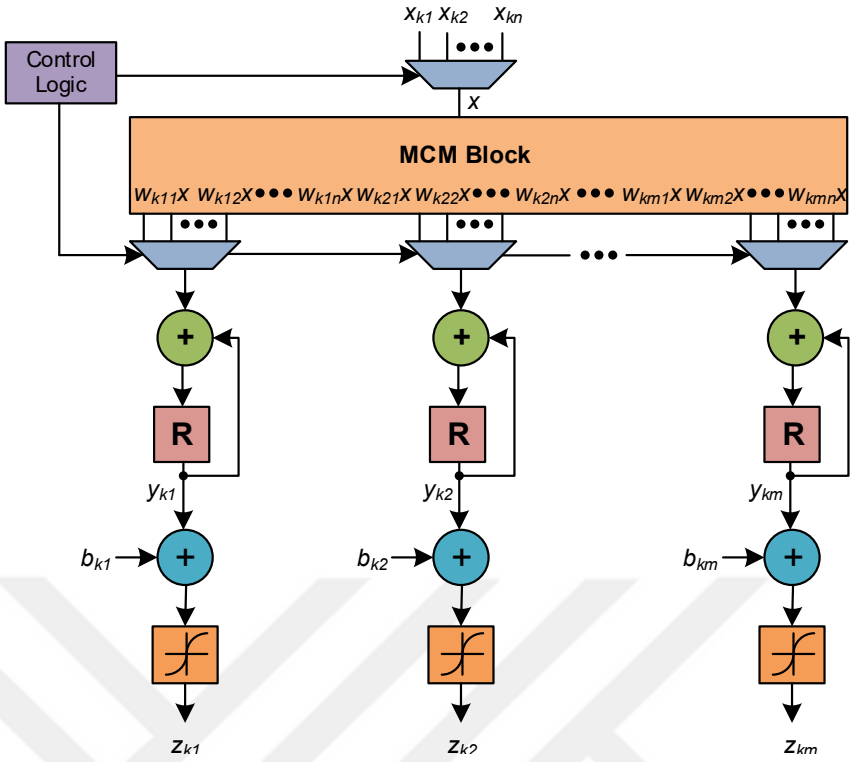

Figure 3.15 : Multiplierless realization of neuron computations at the k^{th} layer under the SMAC_NEURON architecture.

3.5.2 Multiplierless ANN design under the time-multiplexed architectures

Under the SMAC_NEURON architecture, multiplications of related weights by input variables at the k^{th} layer, which is shown in Figure. 3.12, can be computed in an MCM block and redirected to the corresponding adders using multiplexers as shown in Figure. 3.15.

To increase the sharing of partial products and thus minimize the necessary number of adders/subtractors, instead of using an MCM block for each neuron, a single MCM block is used, which realizes the multiplication of all weights in a layer by an input variable. To find the shift-adds realization of the MCM block with the fewest number of adders/subtractors, the exact algorithm of [66] is used.

Similarly, the multiplierless realization of ANN under the SMAC_ANN architecture presented in Figure 3.13 can be obtained when the multiplication of all weight values by the selected input variable is implemented using an MCM block. However, since one multiplier is replaced by a large number of adders/subtractors, such a multiplierless realization increases the hardware complexity significantly.

3.6 SIMURG: The CAD Tool

In this section, we present our CAD tool called SIMURG developed to generate automatically the hardware description of an ANN under the design architectures given in Section 3.3 and the multiplierless design techniques described in Section 3.5.

The weight and bias values of the ANN are specified using a state-of-the-art method, given the ANN structure, which includes the number of inputs, outputs, hidden layers, and neurons in the hidden layers, as well as the type of activation function of neurons for each layer. In this study, we used MATLAB neural network toolbox [69] to train the ANN, but for this chapter we investigated, two different training methods, ZAAL [83], and pytorch [68] in addition to the MATLAB to demonstrate the effect of the training method.

Generally, Training tools include the conventional and stochastic gradient descent methods, and the Adam optimizer [84] as an iterative optimization algorithm. They have different weight initialization techniques, such as Xavier [85], He [86], and a fully random method. They also have a variety of stopping requirements, such as the number of iterations, early stopping using validation data, and loss function saturation.

It can describe sigmoid, hard sigmoid (hsig), hyperbolic tangent, hard hyperbolic tangent (htanh), linear (lin), rectified linear unit (ReLU), saturating linear (satlin), and softmax [87] as a activation functions for neurons in each layer.

To realize ANNs in hardware level, initially floating-point weight and bias values determined during the training phase and converted to integers with given quantization technique. The ANN design is described in hardware automatically with SIMURG tool. This realization is based on the ANN structure given to a training algorithm, the integer weight and bias values, and the design architecture, *i.e.*, parallel, SMAC_NEURON, or SMAC_ANN. The activation functions used in SIMURG are *hsig*, *htanh*, *lin*, *ReLU*, and *satlin* due to their simplicity in hardware. The tool can define the multiplication of constant weights by input variables in a behavioral fashion. Also, it can find the multiplierless realizations of these constant multiplications as described in Section 3.5. The tool also generates a test-bench and necessary files to verify the

ANN design and the synthesis scripts automatically. The SIMURG tool with its limited number of functions is available at https://github.com/leventaksoy/ANNs.

3.7 Experimental Results

In this chapter, we used the pen-based handwritten digit recognition problem [54] as an application to evaluate different architecture, structure, and training method of ANNs.

In the convolutional neural network design of this application, we implemented 5 feedforward ANN structures with different number of layers and number of neurons in layers, denoted as p_{in}–η_1–η_2–…–η_λ, where p_{in} stands for the number of ANN primary inputs, which is equal to 16, and η_k, where $1 \leq k \leq \lambda$, indicates the number of neurons in the k^{th} layer. Note that the activation function of each neuron in the hidden and output layers in training (hardware) was respectively *tanh* (*htanh*) and *satlin* (*satlin*). The activation functions were determined based on the software test accuracy found in training.

The ANNs were trained using 7494 data and tested using 3498 data. Table 3.1 presents the training and hardware design details on different ANN design structures. In this table, *sta* and *hta* denote the software test accuracy, and hardware test accuracy, respectively. Floating-point weight and bias values were converted to integers using the minimum quantization value determined as described in Section 3.4. In the ANN hardware design, bitwidths of ANN inputs and outputs at each layer were determined as 8.

Observe from Table 3.1 that different architectures lead to ANN designs with different hardware accuracy. However, they yield software test accuracy values close to hardware test accuracy values. Note that the difference between the software and hardware test accuracy is due to the quantization value, bitwidths of ANN inputs and outputs at each layer, and different activation functions used in training and hardware design.

In this work, we present gate-level results of ANN designs implemented in three different architectures, namely parallel, SMAC_NEURON, and SMAC_ANN, as described in Section 3.3. To allow a reasonable analogy with time-multiplexed designs, flip-flops were applied to the ANN design outputs in parallel designs. The Cadence

Table 3.1 : Details of ANNs on training and hardware design.

Structure	ZAAL [83]			PYTORCH [68]			MATLAB [69]		
	sta	hta	tnzd	sta	hta	tnzd	sta	hta	tnzd
16-10	84.6	86.0	431	85.5	85.1	374	89.1	89.3	374
16-10-10	94.1	93.6	855	95.9	95.2	950	95.9	95.9	857
16-16-10	96.0	95.9	1245	95.6	95.6	1338	96.9	95.0	1291
16-10-10-10	94.7	94.0	1121	95.8	95.6	1190	96.4	94.7	1121
16-16-10-10	96.6	96.6	1432	96.7	96.7	1608	96.6	95.2	1560
Average	93.2	93.2	1017	93.9	93.6	1092	95.0	94.0	1041

RTL Compiler with the TSMC 40nm design library was used to synthesize ANN designs that were defined in Verilog hardware description language.

Alongside the MATLAB ANN tool, we have trained the ANN using Pytorch and the ZAAL tool, which was developed in our lab. Since the topic of this thesis is hardware optimization of ANNs, we will only use the MATLAB tool to train networks in the following chapter, which is more general.

It's worth noting that different software strategies can help reduce ANN complexity too. In order to explore the impact of a design architecture on the ANN hardware complexity.

Figs. 3.16-3.18 present respectively area (in μm^2), latency (in ns), and energy consumption (in pJ) results of ANN designs under the parallel, SMAC_NEURON, and SMAC_ANN architectures where constant multiplications are described in a behavioral fashion. It's worth noting that the ANN output is obtained by multiplying the clock time by the number of clock cycles. Iteratively, the clock time was shortened by using the retiming technique in the synthesis tool. The test data in simulation was used to produce the switching activity data required for the computation of power dissipation. The ANN design was also checked using this test data set. Latency and power dissipation are multiplied to calculate energy consumption.

Observe that weight and bias values found by different training algorithms lead to ANN designs with different hardware complexity where their impact is clearly observed on ANN designs under the parallel architecture since there exist a large number of constant multiplications. On the other hand, while ANN designs under the SMAC_ANN architecture have the smallest area, the ones under the parallel architecture occupy the largest area. However, the latency of ANN designs under

the parallel architecture is significantly smaller than those of ANN designs under the time-multiplexed architectures. Moreover, ANN designs under the SMAC_ANN architecture consume the most energy. Note that area, latency, and energy consumption values of ANN designs under the SMAC_NEURON architecture are in between those of ANN designs under the parallel and SMAC_ANN architectures.

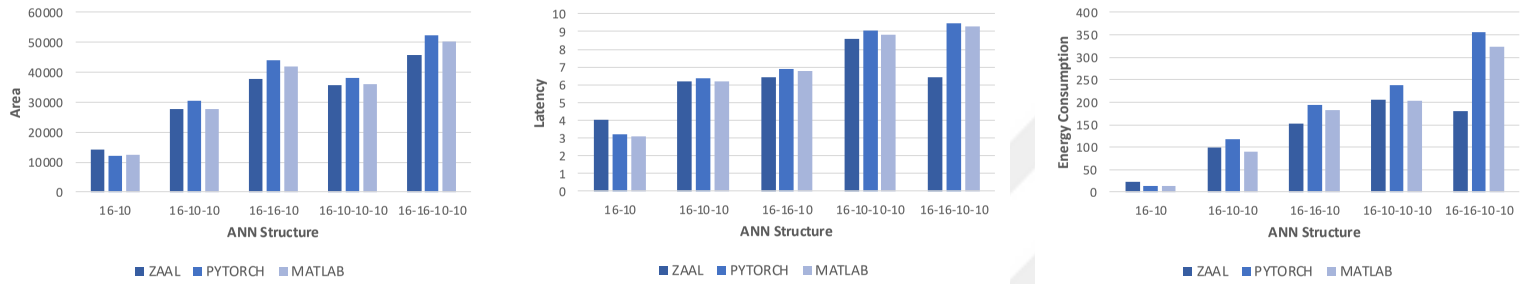

Figure 3.16 : ANN designs under the parallel architecture when constant multiplications are described in a behavioral fashion.

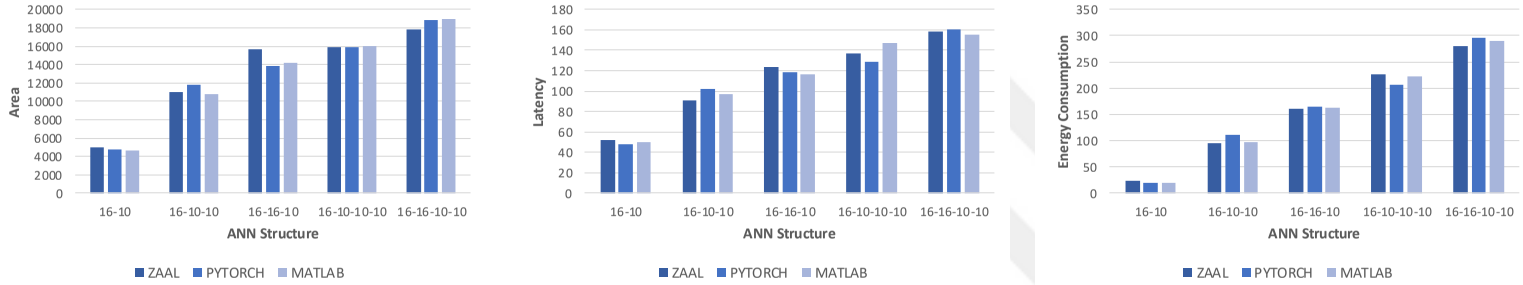

Figure 3.17 : ANN designs under the SMAC_NEURON architecture when constant multiplications are described in a behavioral fashion.

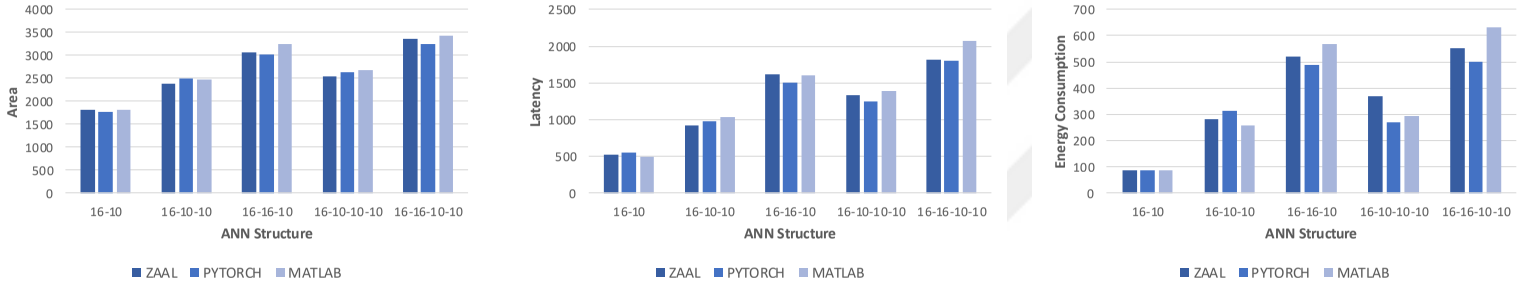

Figure 3.18 : ANN designs under the SMAC_ANN architecture when constant multiplications are described in a behavioral fashion.

4. EFFICIENT HARDWARE REALIZATION OF ANNS BY APPROXIMATE BLOCKS

4.1 Introduction

The previous chapters served us essential acknowledge about the approximate computing and ANNs hardware realization. We will exploit this knowledge to implement the ANN by employing approximate blocks in this chapter. As discussed in Chapter 3, ANNs building blocks are neuron that sums the multiplication of input variables by weights as expressed in equation 3.1. Since Multipliers and adders dominate ANNs hardware, the approximate versions of these arithmetic blocks are replaced by their exact version in this chapter. Also, Chapter 3 reveals the efficiency of the time-multiplexed architectures in terms of hardware complexity, consequently the main focused architecture is MAC-based method in this chapter.

By investigate the literature we obtain many efficient algorithms for reducing ANN hardware complexity, including [15, 26, 28, 60–62]. This chapter will deal with the implementation of ANNs in hardware using approximate adders and multipliers in a time-multiplexed architecture when accounting for the ANN hardware accuracy. In order to do this, the exact adders and multipliers in the MAC blocks are replaced with approximate adders and multipliers. We also present a novel approximate multiplier in this chapter, which allows us to investigate the trade-off between hardware complexity and error at the multiplier output by varying the approximation level. It is worth noting that, contrary to the methods of [29, 31], the generation of an approximate multiplier with different bit-widths of inputs under the given approximation level can be performed in linear time. Using approximate multipliers by various approximation levels for the neuron computations at different layers, will greatly reduce the ANN hardware complexity [26].

Experiments show that, ANNs with the proposed approximate multiplier take up fewer area and use less energy than those with previously suggested approximate multipliers

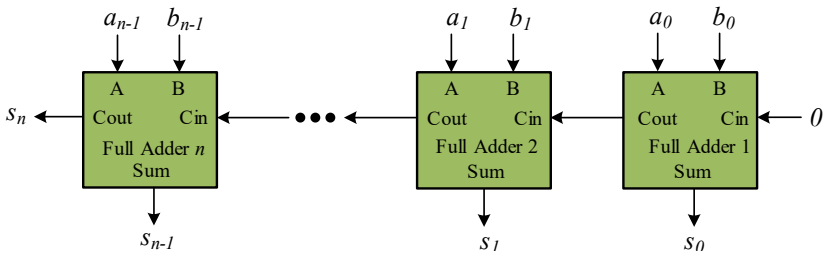

Figure 4.1 : Ripple carry adder.

in [29, 88]. It is also shown that, by using approximate adders, the ANN hardware complexity can be further reduced.

4.2 Approximate Blocks for ANN

4.2.1 Approximate adders

Generally a n-bit ripple carry adder made up of n 1-bit full adders ,which is demonstrated in Figure. 4.1. FA's input bits are represented by A, B, and carry-in (Cin), while its output bits are represented by Sum and carry-out ($Cout$). Conventionally it is presumed that simultaneous errors on both the Sum and $Cout$ outputs of FA can produce a greater erroneous result in literature [3, 31]. We had showed this assertion, though, ignores the fact that, while an error on one of an FA block's outputs increases the error at the adder output, an error on the other output can reduce the error at the adder output. As we had discussed in Chapter 2, the simultaneous error in full adder outputs, results in a considerable area save compared to the other full adders in the literature. We had given the truth table of the proposed adder in Table 2.2. Also, we had presented a synthesis method for obtaining a n-bit approximate ripple carry adder, which replaces the exact FAs with APADs under the given error value.

To simplify the implementation of ANNs by approximate blocks, we introduce a new error metric for the n-bit ripple carry adders in this chapter. According to Chapter 2 results, for different AAED values, APAD4 dominates the array of n-bit ripple carry adder. As a result, we suggest the number of APAD4 as a new error metric which we called it approximate level. Therefore, the approximate level for n-bit ripple-carry adders is the number of APAD4. For example, if the ripple-carry adder is 8 bit and its approximate level is 2, it means the 2 least significant one-bit adders are APAD4 and the rest 6 one-bit adders are exact adders.

4.2.2 Approximate multipliers

We also introduce a new design methodology for implementation of approximate multipliers, to suit better in ANNs. The realization of an exact multiplier is divided into two steps: partial product generation by AND gates, and partial product accumulation by half adders (HAs) [1] and FAs. An exact 4-bit unsigned multiplier structure is shown in Figure 4.2. HAs and FAs are represented by rectangular blocks of two and three entries, respectively.

Our proposed synthesis tool in Chapter 2, replaces exact HA and FA blocks in the exact multiplier with their approximate versions, taking into account the error at the multiplier output. The proposed method generates approximate multipliers called PBAM, where these multipliers are probability-based approximate multipliers in the design of an approximate multiplier, by considering the probability of occurring of logic 0 and 1 at the outputs of all HA and FA. Also, we considered the CGP method of [29], where approximate multipliers generated by this method are derived from the exact multipliers.

In this chapter, we suggest a new approximate multiplier called LEBZAM, which is applied by setting the least significant outputs of an exact multiplier to zero, where r is the approximation level. The following is a description of the synthesis method: i) Set the exact multiplier's r least significant outputs to 0; ii) Remove all FA and HA blocks needed to realize the exact multiplier's r least significant outputs. The realization of a 4-bit approximate multiplier when r is 3 is seen in Figure 4.3. Consider that, the proposed approximate multiplier is completely different than truncation multipliers , where the proposed multipliers set the least significant bits to zero, but the truncation multipliers eliminates these bits.

In contrast to the approximate multipliers of [29, 31], an approximate multiplier LEBZAM can be conveniently obtained by providing the approximation level and bit-widths of the inputs. As a result, in this section, under different ANN architectures, which were discussed in chapter 3, and by exploiting approximate blocks by different approximate levels, we are attempting to reduce the cost of ANN hardware.

[1] Half adder is obtained when one of the inputs of FA is set to 0.

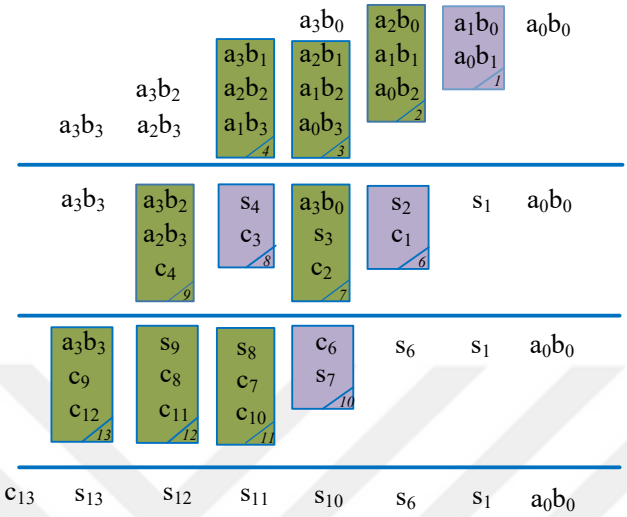

Figure 4.2 : Exact 4-bit unsigned multiplier.

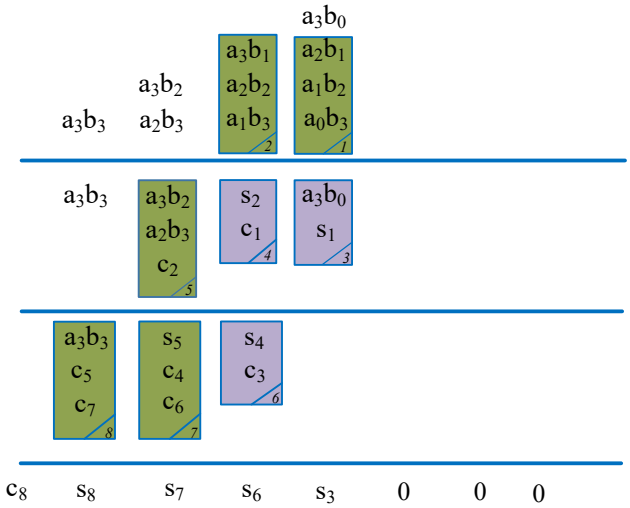

Figure 4.3 : Approximate 4-bit unsigned multiplier with the least significant 3 bits are set to logic value 0.

4.2.3 Approximate level

4.2.3.1 *SMAC_NEURON*

To determine the approximate level of the multipliers and adders based on the misclassification rate (MR) for *SMAC_NEURON* architecture following steps are obtained.

1. Set the hidden layer number n to 1.
2. Set the approximate level AL_n to 0.
3. Increase AL_n value by 1.
4. Calculate Approximation Misclassification rate AMR.
5. If $AMR - MR < tolerable\_error$ go to Step 3, otherwise increase n value by 1.
6. If $n < n_{max} + 1$ save $AL_n - 1$ as the approximate level of n^{th} layer and and return to step2.
7. save $AL_n - 1$ as the approximate level of output.

These steps are taken separately for adders and multipliers. We must acknowledge that starting with multipliers or adders will result in the same approximate level values for the blocks, in this study, we apply the proposed method for approximate multipliers at first. To shrink the search space of adders' approximation level (AAL), the minimum level value of AAL is set to the determined multipliers' approximate level (MAL) value increment by one. Additionally, the error distance values of LEBZAM are negative or zero for all cases, based on this error pattern, the approximate level of multipliers and adders for all neurons in each layer are chosen identical. Contrarily selecting a higher approximate level for any neuron comparing to other neurons at the same layer, leads to a negative bias of that neuron i.e. the neuron with the higher approximate level, posses a scanty output regarding to other neurons, where this biasing results a disturb in accuracy.

An arithmetic unit with m-bitwidth output, posses a number between 0 and m as a approximate level($(m+1)$options). By considering n-bit $\times$ n-bit multiplier for

MAC unit, the adder output bitwidth value will be $(2n+1)$, and the total possible combination number of approximate level for adder and multiplier for each neuron will be $(2n+1)\times(2n+2)$. Also by considering that there are η neurons in λ layers, the total possible combination for a ANN is formulated as follow.

$$(4n^2+6n+2)\sum_{i}^{\lambda}\eta_i \tag{4.1}$$

By exploiting the proposed method, the approximate level values of the multipliers are identical for all of the neurons, according to the method, we increment the *MAL* value by 1 until the error deviation becomes greater than the given error limit value. *MAL* value of n-bit multiplier is a number between 0 and $2n$, hence by exploiting the proposed method, the total investigated case is *MAL* for all neurons in each layer. In line with the proposed method, the minimum number of *AAL*value is equivalent to the determined *MAL* value increment by one. Similarly, the same steps are obtained to find the *AAL* value; consequently, the total investigated cases is $2n-MAL-AAL$ for all neurons of each layer. The total number of examined cases for the whole network is given by following formula.

$$\sum_{i}^{\lambda}(MAL+1)+(AAL-MAL) \tag{4.2}$$

As an example, consider a Pen-digit handwritten digit recognition problem [54], the trained network architecture consists of 16 inputs, 50 neurons in the hidden layer, and 10 outputs. Assume the input and weights bit-width is 8, consequently the all possible combination of *MAL* and *AAL* is:

$(4(8^2)+(6\times 8)+2)\times(50+10)=18360$

whereas, by exploiting the proposed method, the significant reduction in the explored cases occurs.

The *MR* deviation percentage for different *MALs* and *AALs* values are shown in Figure.4.4. Based on the proposed method, after seven iterations, the MAL_1 value is set to 6. To find this value all the other arithmetic units are set to their exact versions, and their approximate level is 0. Note that setting MAL_1 value to 7 causes to MR value becomes greater than the given value where the tolerable error value is considered 1% of MR for this example. The same steps are employed to obtain the MAL_2 value.

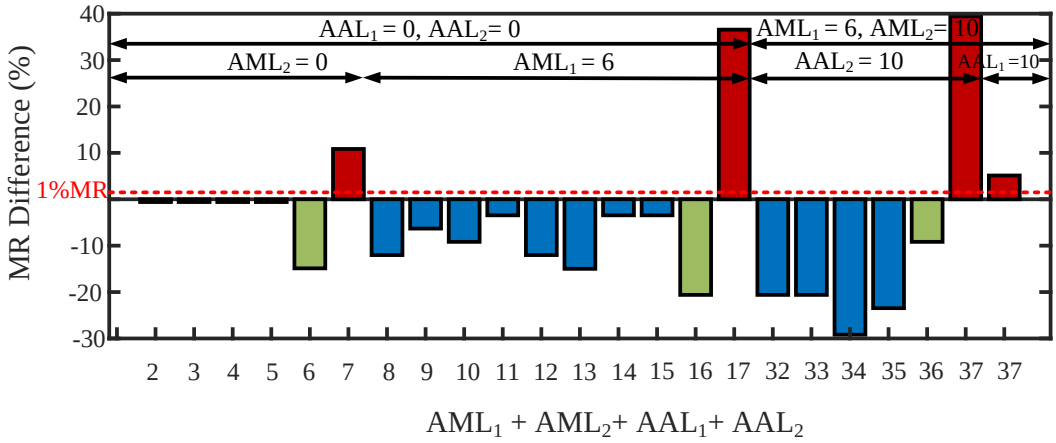

Figure 4.4 : Misclassification rate for SMAC_NEURON architecture by the different approximate levels of multipliers and adders.

According to the proposed algorithm, to investigate AAL_1 and AAL_2 values, the starting points are set to their corresponding *MAL* value in each layer. As shown in Figure.4.4, the *AAL* value for layer1 and layer2 is set to 10. Note that the total examined case number for this example is:

$(7+(10-7))+(10+(10-10))=20$

which is negligible in comparison to the all possible 9180 cases.

4.2.3.2 *SMAC_ANN*

To determine the approximate level of multipliers and adders in *SMAC_ANN* architecture following 7 steps are obtained.

1. Set the *MAL* and *AAL* to 0.
2. Increase *MAL* value by 1.
3. Calculate Approximation Misclassification rate *AMR*.
4. If $AMR - MR < tolerable\_error$ go to Step 2, otherwise save $MAL-1$ as the approximate level of multiplier.
5. Increase *AAL* value by 1.
6. Calculate Approximation Misclassification rate *AMR*.
7. If $AMR - MR < tolerable\_error$ go to Step 4, otherwise save $AAL-1$ as the approximate level of adder.

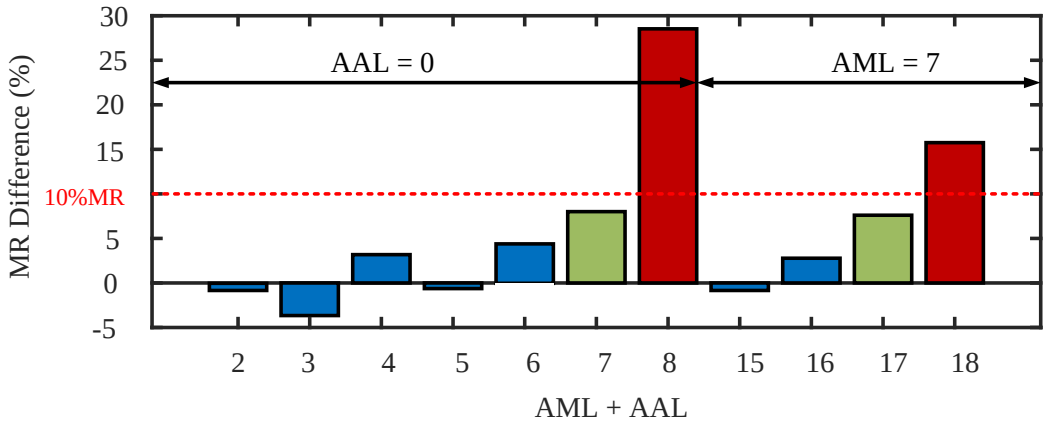

Figure 4.5 : Misclassification rate for SMAC_ANNarchitecture by the different approximate levels of multipliers and adders.

The *SMAC_ANN* architecture comprises a singular MAC unit as the arithmetic unit; consequently, determining the approximate level of multiplier and adder is more straightforward compared to the *SMAC_ANN* architecture.

Distinct from the ANN structure, the total possible combination of *MAL* and *AAL* values is correlated with the output bit-width of arithmetic units. By assuming that the multipliers and adders output bit-widths are j and k, respectively, the number of all possible combinations of *MAL* and *AAL* will be $j \times k$. On the other hand, by exploiting the proposed method, the number of investigated cases shrinks to $MAL + (AAL - MAL)$ cases.

As an example, Pen-digit handwritten digit recognition problem is employed by the same parameters but under *SMAC_ANN* architecture. The applied method results are shown in Fig4.5. Starting by the multiplier, after 8 iterations the MAL value is set to 7, and the tolerable error value is considered as 1.1 of *MR*. According to the proposed method, AAL initial value is set to $MAL + 1$, and after 4 iterations, the AAL value is set to 10 correspondingly. Take consider the total examined cases is 12, whereas all possible combination is (16×20) by considering that, the output bit-width is 16 and 20 for multipliers and adders, respectively.

4.3 Experimental Results

In this section, MNIST and Pen-digit data-set are exploited to train ANNs. The MNIST handwritten digit classification problem is a well-known dataset in computer vision and deep learning applications. The inputs for the MNIST are the real image pixels of the

handwritten numbers. On the other hand, the inputs for the Pen-digit are 16 attributes of the images. Therefore, Pen-digit requires a less complex ANN structure to predict the expected numbers. The ANN was trained using the MATLAB [69] deep learning toolbox. The training and test inputs were normalized within -1 and 1, the weights were randomly initialized and they were modified using a backpropagation-based learning approach to minimize the error between the obtained and desired response. The ANN designs hardware description language is Verilog and were synthesized using the Cadence Genus tool with the TSMC 40nm design library.

The results for two different applications are provided in two different following sections.

4.3.1 Pen-digit problem

We used the pen-digit handwritten digit recognition problem [54] as a first research application. A Feedforward ANN was implemented, which includes 16 inputs, a hidden layer by 16 neurons, and 10 neurons at the output layer. The hidden and output layer activation functions were symmetric saturating linear, and softmax, respectively. The ANN was trained by 7494 data and was tested by 3498 data. The misclassification rate after training was calculated as 4.85%. Following the conversion into integers of the floating-point weight and biasing values when the q quantization value was set to 8, the behavioral ANN design using exact adders and multipliers was described and the hardware misclassification rate was identified as 5%.

The designs of the ANN in this study will be implemented using approximate adders and multipliers without exceeding the HMR limit of 5.5%. Also, ANNs will be implemented under two different SMAC_NEURON and SMAC_ANN architectures and using 3 different approximate multipliers, which are PBAM, LEBZAM, and the logic level approximate multipliers [29]. We note that all the exploited approximate adders are our proposed adders. The approximate [29], *mul12s 2NM* and *mul12s 2KM* multipliers have 12-bit inputs and are selected among other multipliers for their minimum area and error. Notice that, the approximation levels of adders and multipliers in the hidden and output layers have been systemically calculated taking HMR values into account.

ANN designs' gate-level performance presented in Tables 4.1-4.2, in which *area*, *delay* and *power* respectively stand for total area in μm^2, the delay in the critical path which is determined to be the clock period in *ns*, and total power dissipation in *mW*. The time at *ns* that is needed to achieve ANN output after input, calculated as the clock time multiplication with the number of clock cycles required to achieve the ANN output, is *latency*. For our proposed ANNs the number of clock cycles required in order to get an ANN output is calculated as 34 and 468 for *SMAC_NEURON* and *SMAC_ANN*. Energy consumption (*energy*), in *pJ* calculated as a latency multiplied by power dissipation.

We observe that the retiming procedure in the synthesis tool has increased the clocking period. Use the test data in simulation to produce the switching activity data for calculating of the power dissipation. Also, the test data was used to validate the ANN design and calculate the hardware misclassification rate.

The results of gate-level ANN designs presented by the Table 4.1. The architecture for ANN is SMAC NEURON, where just approximate multipliers in MAC blocks have been replaced by the exact versions. Note that since approximate multipliers in [29] are optimized for a fixed bit length, they may be worse in the area, latency, and energy consumption values than ANN's with exact multipliers. Also, consider that logic synthesis tools optimize the exact multipliers separately, especially for widely-used bit widths like 8×8 or 16×16.

According to these results, the use of approximate multipliers LEBZAM, by finding the appropriate approximate multipliers in the hidden layers and in the output layers, can reduce the ANN hardware complexity. In addition, the largest reduction in the area, latency, and consumed energy is obtained by LEBZAM multipliers. Note that, by simply changing the approximation level of multipliers the trade-off between hardware complexity and accuracy is obtainable.

Table 4.2 presents the gate-level results of ANN designs under the SMAC_NEURON architecture where both exact adders and multipliers in the MAC blocks are replaced by the approximate ones. The experimental results indicate that concurrent use of approximate multiplier and adders significantly reduces the complexity of the ANN.

Table 4.1 : Results of SMAC_NEURON architecture using approximate multipliers.

Multiplier Type	Approximation Level Hidden	Output	area	delay	latency	power	energy	HMR	area gain	energy gain
Behavioral	0	0	15327	3.58	121.68	1.44	174.77	5.00	0%	0%
*mul*12*s*_2*NM* [29]	NA	NA	13929	3.72	126.31	1.23	155.04	5.12	9%	11%
*mul*12*s*_2*KM* [29]	NA	NA	17227	3.70	125.80	1.44	181.33	5.00	-12%	-3%
PBAM [88]	7	11	13276	3.57	121.35	1.31	159.14	4.84	9%	13%
PBAM [88]	7	12	12992	3.66	124.37	1.30	161.52	5.03	15%	8%
PBAM [88]	8	11	12761	3.41	115.91	1.26	145.51	5.37	17%	17%
LEBZAM	6	9	11999	3.68	125.02	1.00	125.21	5.03	22%	28%
LEBZAM	7	11	10224	3.45	117.40	1.04	122.05	4.80	33%	30%
LEBZAM	7	12	9723	3.41	116.01	0.94	109.41	5.09	37%	37%

Table 4.2 : Results of SMAC_NEURON architecture using approximate multipliers and adders.

Multiplier Type	Approximation Level				area	delay	latency	power	energy	HMR	area gain	energy gain
	Hidden		Output									
	Mul	Add	Mul	Add								
Behavioral	0	0	0	0	15327	3.58	121.62	1.44	174.77	5.00	0%	0%
*mul*12*s*_2*NM* [29]	NA	10	NA	14	11854	3.92	133.14	0.59	78.76	5.17	22%	55%
*mul*12*s*_2*KM* [29]	NA	9	NA	15	13133	3.95	134.30	0.69	92.48	5.34	14%	47%
PBAM [88]	7	7	12	11	10226	3.66	124.37	0.61	76.25	5.03	33%	57%
PBAM [88]	7	7	12	12	9798	3.64	123.86	0.61	75.70	5.20	33%	57%
PBAM [88]	7	7	12	13	9534	3.66	124.37	0.62	77.25	5.17	39%	56%
LEBZAM	6	10	9	13	10392	3.58	121.72	0.58	70.11	5.31	32%	60%
LEBZAM	7	12	10	13	8801	3.61	122.88	0.55	67.32	4.88	43%	61%
LEBZAM	7	11	10	14	8989	3.61	122.81	0.52	63.68	4.97	41%	64%

Table 4.3 : Results of SMAC_ANN architecture using approximate multipliers.

Multiplier Type	Approximation Level	area	delay	latency	power	energy	HMR	area gain	energy gain
Behavioral	0	3180	3.52	1646.42	0.35	569.33	5.00	0%	0%
*mul*12*s*_2*NM* [29]	NA	3278	3.72	1738.62	0.29	499.80	5.00	-3%	12%
*mul*12*s*_2*KM* [29]	NA	3279	3.77	1764.83	0.29	504.74	5.00	-3%	11%
PBAM [88]	0	3287	3.79	1774.19	0.29	518.38	5.00	-3%	9%
PBAM [88]	7	3194	3.76	1760.15	0.28	499.60	4.83	-1%	12%
PBAM [88]	8	3148	3.24	1518.19	0.28	431.60	5.34	2%	24%
LEBZAM	5	3189	3.69	1725.98	0.27	472.95	4.94	-2%	8%
LEBZAM	6	3152	3.68	1724.58	0.28	489.60	4.90	1%	14%
LEBZAM	7	3091	3.56	1664.68	0.27	449.89	4.80	3%	21%

Table 4.4 : Results of SMAC_ANN architecture using approximate multipliers and adders.

Multiplier Type	Approximation Level		area	delay	latency	power	energy	HMR	area gain	energy gain
	Mul	Add								
Behavioral	0	0	3180	3.52	1646.42	0.35	569.33	5.00	0%	0%
*mul*12*s*_2*NM* [29]	NA	13	2908	3.40	1590.26	0.25	391.63	5.06	9%	31%
*mul*12*s*_2*KM* [29]	NA	13	3140	3.68	1721.30	0.26	451.51	5.46	1%	21%
PBAM [88]	7	10	2972	3.55	1659.53	0.26	426.62	5.03	7%	25%
PBAM [88]	8	9	2978	3.59	1679.18	0.25	421.98	5.03	6%	26%
PBAM [88]	7	11	3029	3.84	1798.52	0.25	448.54	4.66	5%	21%
LEBZAM	6	14	3046	3.53	1652.51	0.28	469.89	4.95	4%	17%
LEBZAM	7	12	3041	3.62	1692.29	0.26	440.25	4.66	4%	23%
LEBZAM	7	13	3021	3.53	1650.17	0.26	426.73	5.40	5%	25%

According to Table 4.2, our approximate multipliers provide the maximum increase in areas and energy consumption up to 43% and 64%, respectively.

As discussed in chapter 3, a single MAC unit processes the operations under the SMAC_ANN architecture. By replacing the multiplier with approximate versions, new ANNs are obtained, and results are given in Table 4.3. According to the results, the proposed LEBZAM multipliers save more energy and area when compared with the other methods. In addition to the LEBZAM multipliers, changing the exact adders to the approximate counterparts leads to further reduction according to Table 4.4.

Interestingly, the use of approximate adders and multipliers can also improve the hardware accuracy as can be observed on Tables 4.2 and 4.4.

4.3.2 MNIST problem

As the second application, we considered the MNIST handwritten digit recognition problem [89]. The dataset consists of gray-scale images from NIST, which is normalized to fit into (28×28) pixel boxes. The ANN is employed to predict the digits among 10 integers (0-9) based on the input pixels.

To examine the performance of the proposed method on a different structure, we implemented the feedforward ANN with two different structures; 3 hidden layers by 256 neurons for the first case, and a hidden layer by 128 neurons for the second case. The activation functions of the hidden and output layer were symmetric saturating linear, and softmax, respectively.

The ANN was trained using 60000 data and was tested using 10000 data. Also, the quantization factor q was set to 12 for employed ANN by exploiting the proposed steps in Chapter 3.

By converting input data to 12 bits integer, 116 of 784 input pixels remained unchanged for the train and test data. In a different expression, for 12-bit resolution, 116 of the pixel values are identical for the whole of the train and test data. MATLAB automatically pre-read the data and remove unchanged data because eliminating these values will not affect the performance of ANN. By training ANN through MNIST database at the software level, the computed MRs for test data were calculated as 2.60 and 2.24 for 668-256-256-256-10 and 668-128-10 structure, respectively.

To evaluate the efficiency of the proposed multipliers and the algorithm among the other literature, proposed multipliers are compared with CGP based multipliers which are introduced in [29]. According to [28] study, CGP based multipliers holds better result among all the deliberately approximate multipliers.

The ANNs were implemented under the SMAC_NEURON and SMAC_ANN architectures using the proposed approximate adders in this thesis, the approximate multipliers of [29], and our proposed multipliers LEBZAM and PBAM. The signed approximate multipliers of [29], have constant 12-bit$\times$12-bit and 16-bit$\times$16-bit inputs. Also, according to *SMAC* synthesis method, inputs bit-widths of multipliers are non-identical for this architecture. To adopt the [29] multipliers with the employed structure, the multipliers from the library of [29] by maximum bit-width were selected, then we removed the gates and in-outs of extra bits. Note, we have systematized the approximation levels on the hidden layers and output layers of the PBAM and LEBZAM multipliers in accordance with the HMR value. The ANN designs were described in Verilog and synthesized using the Cadence Genus tool with the TSMC 40nm design library.

The energy and area save for trained ANN by ANN by 668-128-10 structure and under *SMAC_NEURON* architecture are depicted in Figure. 4.6 and Figure. 4.7 respectively. To evaluate the efficiency of the proposed method compared to the introduced method in [29], 63 cases by different multipliers and adders are exploited. As shown in these figures, exploiting simultaneously approximate multipliers and adders according to the proposed method always hold more save in energy and area with the same HMR values compared to the other methods.

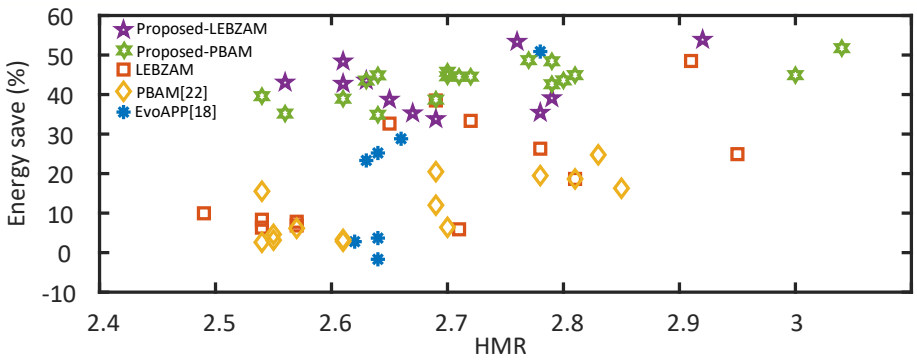

Figure 4.6 : Energy save percentages of ANN for different approximate methods in terms of hardware misclassification rate.

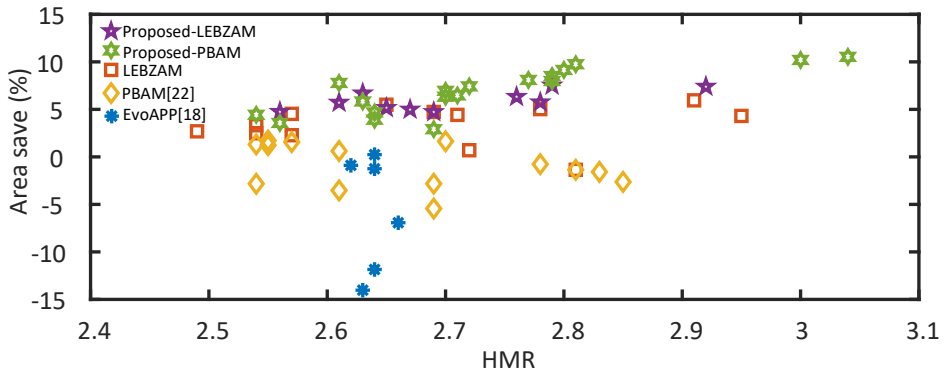

Figure 4.7 : Area save percentages of ANN for different approximate methods in terms of hardware misclassification rate.

Tables 4.6-4.5 present the gate-level results, where *area*, *delay*, and *power* stand respectively for total area in μm^2, the delay in the critical path which is determined to be the clock period in *ns*, and total power dissipation in *mW*. Again, *latency* refers to the time in *ms* needed to produce an ANN output when an input is added. This value was calculated as the number of clock cycles needed multiply by the clock period. The number of clock cycles required to obtain the ANN output under the SMAC_NEURON and SMAC_ANN is respectively computed as 798 and 87060 for 668-128-10 structure and, 1440 and 306196 for 668-256-256-256-10 structure. Again, energy consumption in μJ is calculated as the latency and dissipated power multiplication. The clock period improving technique again has been applied. Also, the test data employed to design validation, and save switching activity by generating the ".saif" file.

The gate level results of the SMAC_NEURON architecture, in which the exact multipliers and adders of the blocks of MAC are replaced by the approximate ones, are shown in Table 4.6 and Table 4.7. Approximate multipliers of [29] may have worse area values than those of ANN using exact multipliers for the reasons which were discussed in section 4.3.1. On the other hand, the use of LEBZAM multipliers can save in the ANN hardware complexity by obtaining the proper approximation levels of multipliers at the different layers. In addition, the proposed approximate LEBZAM multiplier leads to the most significant decrease in area, latency and power consumption. Notice that a change in the approximation level of the multipliers and adders will explore the compromise between hardware complexity and accuracy.

By exploiting the proposed algorithm, the deviation limit is set to 2.5% of HMR and by regarding that the HMR value is 2.63, the maximum tolerable HMR is 2.69 for

668-128-10 structure. The MAL value of LEBZAM multipliers are obtained as 7 and 16 for the hidden and output layers. Also, AAL values are calculated as 8 and 17, respectively, for the hidden layer and output layer. Furthermore, to obtain the performance of the proposed approximate level algorithm, different cases beyond this algorithm are given in Table 4.6. It must be noted that, due to the large search space of MAL and AAL values, the proposed algorithm only finds near-optimal values by acceptable variance.

By investigation of Table 4.6 results, observe that the simultaneous use of approximate multipliers with the introduced approximate adders in [88], reduces the ANN hardware complexity significantly. The maximum gain on area and energy consumption reaches up to 6% and 48% using the approximate multipliers and adders with improving in the accuracy. Table 4.7 presents the gate-level results of ANN designs under the SMAC_NEURON architecture for 668-256-256-256-10 architecture. According to the table result, exploiting approximate multipliers and adders yields up to 39% and 5% save in energy and area respectively with a small degradation in the accuracy.

The gate-level results of ANN designs under the SMAC_ANNarchitecture are given in Table 4.5. The results indicate the efficiency of the proposed method and blocks. According to the SMAC_ANNarchitecture, only one multiplier and adder are replaced by the approximate versions. As a result, the cost saving of hardware for this architecture is lesser than the SMAC_NEURON architecture. Note that the proposed approximate units lead to the largest gains on energy consumption.

Table 4.5 : Results of SMAC_ANN architecture for 668-128-10 structure using approximate multipliers and adders.

Multiplier Type	Approximation Level		latency	power	energy	HMR	Energy gain
	Mul	Add					
Behavioral	0	0	688.04	9.33	6418.81	2.63	0%
LEBZAM	6	7	666.18	8.01	5333.17	2.63	17%
PBAM [88]	7	7	720.94	7.73	5570.29	2.7	13%
EvoAPP [29]	HDG	0	695.78	8.30	5773.67	2.04	10%
EvoAPP [29]	GAT	0	659.57	8.59	5667.82	2.78	12%

Table 4.6 : Results of SMAC_NEURON architecture for 668-128-10 structure using approximate multipliers and adders.

Multiplier Type	Approximation Level				area	delay	latency	power	energy	HMR	Area gain	Energy gain
	Hidden		Output									
	Mul	Add	Mul	Add								
Behavioral	0	0	0	0	200831	9.56	7.63	21.52	164.10	2.63	0%	0%
PBAM [88]	7	0	16	0	206506	9.29	7.41	18.70	138.62	2.54	-3%	16%
LEBZAM	7	0	16	0	189796	7.75	6.18	17.88	110.52	2.65	5%	33%
EvoAPP [29]	HDG/KQ	0	HDG/KQ	0	224621	8.71	6.95	17.67	122.73	2.64	-12%	25%
EvoAPP [29]	HFZ/K5	0	HFZ/K5	0	214757	8.30	6.62	17.64	116.82	2.66	-7%	29%
EvoAPP [29]	GAT/NM	0	GAT/NM	0	157275	5.85	4.66	17.28	80.61	2.78	22%	51%
EvoAPP [29]	2KM	0	0	0	229000	8.39	6.70	18.78	125.83	2.63	-14%	23%
EvoAPP [29]	0	0	HDG/KQ	0	202615	8.94	7.13	22.38	159.54	2.62	-1%	3%
EvoAPP [29]	0	0	12N/GAT	0	203342	9.14	7.29	21.68	158.07	2.64	-1%	4%
EvoAPP [29]	0	0	K5/HFZ	0	200349	9.56	7.63	21.87	166.89	2.64	0%	-2%
PBAM [88]	7	8	14	15	191431	8.98	7.17	12.63	90.48	2.64	5%	45%
PBAM [88]	8	8	15	17	185200	9.18	7.33	13.67	100.18	2.61	8%	39%
PBAM [88]	7	7	16	16	191925	10.47	8.35	11.87	99.13	2.54	4%	40%
LEBZAM	7	7	15	17	191286	9.09	7.25	12.88	93.42	2.56	5%	43%
LEBZAM	7	8	15	15	189343	8.76	6.99	12.11	84.66	2.61	6%	48%
LEBZAM	7	8	16	17	187396	9.03	7.21	12.85	92.62	2.63	7%	44%
LEBZAM	7	7	16	16	190466	9.55	7.62	13.19	100.53	2.65	5%	39%
LEBZAM	8	8	14	14	188112	6.70	5.35	14.31	76.49	2.76	6%	53%

Table 4.7 : Results of SMAC_NEURON architecture for 668-256-256-256-10 structure using approximate multipliers and adders.

Multiplier Type	Approximation Level								area	delay	latency	power	energy	HMR	Area gain	Energy gain
	Hidden1		Hidden2		Hidden3		Output									
	Mul	Add	Mul	Add	Mul	Add	Mul	Add								
Behavioral	0	0	0	0	0	0	0	0	939076	8.05	11.6	141.4	1639.6	3.96	0%	0%
PBAM [88]	7	0	7	0	7	0	14	0	971948	7.31	10.5	131.5	1384.6	4.22	-4%	16%
PBAM [88]	7	7	7	7	7	7	14	14	892979	9.03	13.0	77.1	1002.3	4.22	5%	39%
LEBZAM	6	0	6	0	6	0	14	0	929330	9.00	12.9	107.9	1398.7	4.25	1%	6%
LEBZAM	6	6	6	6	6	6	14	14	919664	9.17	13.2	84.8	1120.2	4.25	2%	32%
EvoAPP [29]	DG/KQ	0	DG/KQ	0	DG/KQ	0	DG/KQ	0	1051004	9.22	13.3	111.9	1486.3	3.99	-12%	9%
EvoAPP [29]	FZ/K5	0	FZ/K5	0	FZ/K5	0	FZ/K5	0	1000478	9.57	13.8	105.1	1449.5	4.05	-7%	12%

5. CONVOLUTION LAYER

5.1 Introduction

The main focus of this dissertation is the efficient hardware realization of the approximate ANNs. In the previous chapters, we proposed a methodology to implement the desired ANNs with minimum hardware complexity. All the investigated techniques were based on the feed-forward ANNs. We discussed that there is a common tendency to train and test the networks on CPUs and GPUs due to their processing strength. However, their large power consumption makes this method impractical for portable devices where the number of processing units, battery capacity, and memory is limited. These considerations make application-specific integrated circuits (ASICs) a favorable method for hardware implementation. To reduce the hardware complexity by considering an increase in latency, ANNs based on Multiply-accumulated (MAC) units and multiplierless designs are proposed in Chapter3. Also by introducing novel approximate units for MAC blocks, a remarkable reduction in power consumption and occupied area are obtained for fully-connected ANNs in Chapter4.

Beyond the discussed fully-connected ANNs, convolutional neural networks (CNN) provide remarkable results in achieving better performances by extracting features from the training data. In this chapter, we will address how to realize CNN convolution layers by exploiting our techniques.

The CNNs consist of convolutional layers followed by the fully-connected layers. Practically, the fully-connected layers are used to classify the inputs' features which are provided by filters or convolution layers. Due to numerous memory access and a large power consumption, ASIC implementation of a CNN with millions of parameters is impractical in the parallel fashion.

CNNs' hardware complexity is dominated by convolution layers where each convolution is a sum of weighted neighboring pixels. On the other hand, fully-connected ANN is a vector multiplication of inputs with related weights. Inspired by the fully-connected ANNs, a new computational method for convolution layers are realized based on the MAC units to reduce the hardware complexity of convolution layers [90]. Exploiting MAC units enables designers to reduce power consumption and silicon area considerably by a hybrid operation (parallel-serial).

By considering the similarity of the fully-connected ANNs and the convolution operation, we propose an efficient computational method to reduce both the number of employed MAC units and the number of clock cycles. Experimental results shows that our proposed computational method results in reduction in area, power dissipation and, number of clock cycles in comparison to the generic computation method introduced in [90] work.

5.1.1 Convolution layer

A convolution layer contains a set of filters whose parameters are specified during the training phase. The convolution operation on an input image using kernel filters extract fundamental features from the image. The inputs and filters are formed in 3 dimensions:height H, width$W$ and channel C.

The convolution operation is represented in Figure 5.4. The height and weight of the filters are smaller than those of the input values. The filter plane slides over the entire input image step by step and, the output is the result of multiple convolutions. As an example, convolution operation in Figure 5.5, is considered as $C_{in} = 1$, $H_{in} = 4$, $W_{in} = 4$, $C_{out} = 1$, $H_f = 2$, $W_f = 2$ and $S = 1$, where S stands for a stride value. In the convolution operation process, stride is the number of pixel-shifting in each operation. The number and the size of the filters varies for different applications.

Serial processing is an alternative approach for parallel fashion computing where re-using a unit, results in a reduction in hardware complexity by an increase in latency. As mentioned in Chapter5, ANNs' hardware complexity are dominated by multipliers and adders. ANNs can be designed under the time-multiplexed architecture using the MAC blocks. The structure of the unit is represented in Figure 5.1, where each neuron in a layer is replaced by a single MAC unit. Hardware realization of the

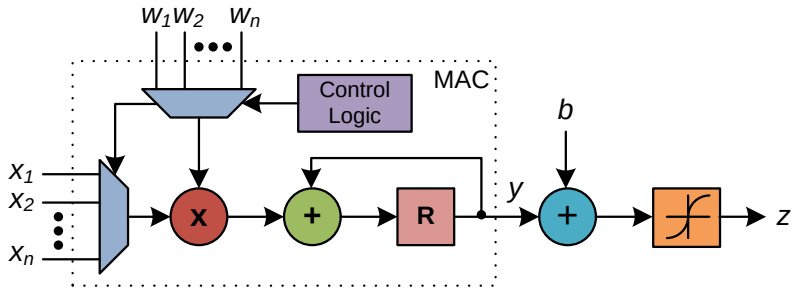

Figure 5.1 : Multiply-accumulate (MAC) block in the neuron computation.

ANNs under MAC units can be classified into two models: *axonal-based* [23] and *dendritic-based* [24] models.

For *axonal-based* model which is shown in Figure 5.2, every single input of a layer is multiplied by the related weights of all neurons in a layer, where all the outputs are calculated simultaneously. As a result, *axonal-based* model does not require obtaining all the inputs at the same time. The process is realized step by step for all inputs and, the control logic unit is a simple counter which counts from 1 to i, where i is the number of inputs. To obtain all of the neuron's output, $i+1$ clock cycles is required.

For *dendritic-based* model which is shown in Figure 5.3, the value of a next neuron is calculated by multiplying all inputs with the related neuron's weights and accumulating them. This method results in the sequential generation of outputs and every step of the calculation needs to obtain all the input values to start the next layer calculation. This model needs $n+1$ clock cycles to determine all neurons' output values where n is the number of outputs. Also by combining these two models, parallel computing is enabled in two successive layers to achieve smaller latency in the whole network computing time [25].

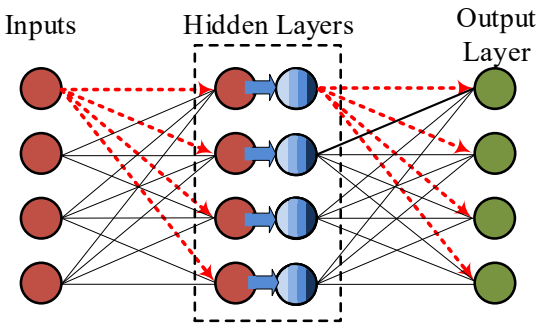

Figure 5.2 : Axonal-based model.

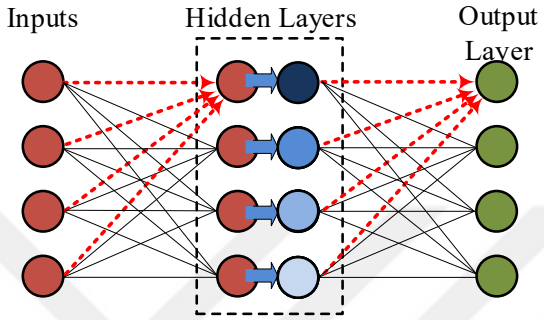

Figure 5.3 : Dendritic-based model.

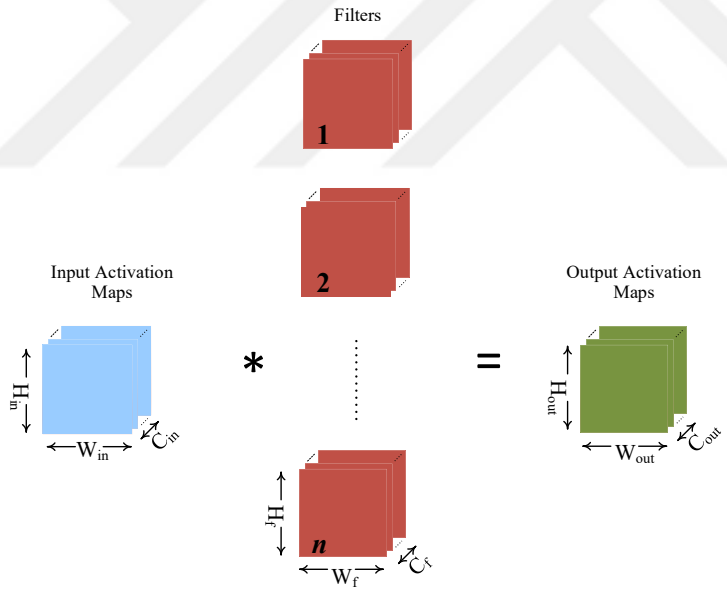

Figure 5.4 : The computation of convolutional layer.

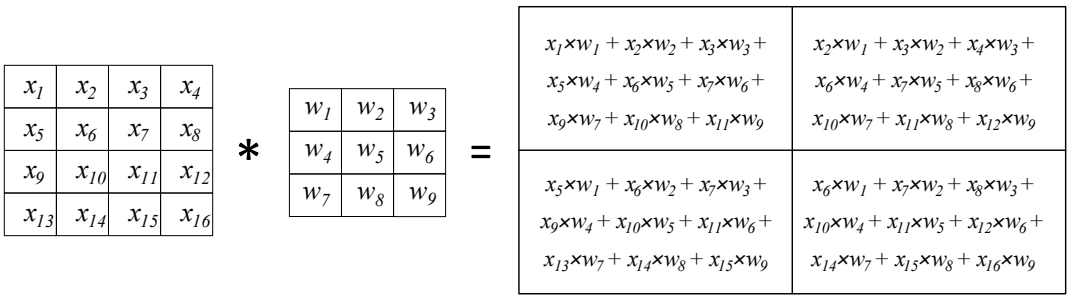

Figure 5.5 : The computation of convolutional layer.

Table 5.1 : The *axonal-based* model data flow for convolutional computation.

Clock Cycles	Neuron #1	Neuron #2	Neuron #3	Neuron #4
clk #1	$X_1 \times \mathbf{W_1}$	$X_1 \times 0$	$X_1 \times 0$	$X_1 \times 0$
clk #2	$X_2 \times \mathbf{W_2}$	$X_2 \times \mathbf{W_1}$	$X_2 \times 0$	$X_2 \times 0$
clk #3	$X_3 \times \mathbf{W_3}$	$X_3 \times \mathbf{W_2}$	$X_3 \times 0$	$X_3 \times 0$
clk #4	$X_4 \times 0$	$X_4 \times \mathbf{W_3}$	$X_4 \times 0$	$X_4 \times 0$
clk #5	$X_5 \times \mathbf{W_4}$	$X_5 \times 0$	$X_5 \times \mathbf{W_1}$	$X_5 \times 0$
clk #6	$X_6 \times \mathbf{W_5}$	$X_6 \times \mathbf{W_4}$	$X_6 \times \mathbf{W_2}$	$X_6 \times \mathbf{W_1}$
clk #7	$X_7 \times \mathbf{W_6}$	$X_7 \times \mathbf{W_5}$	$X_7 \times \mathbf{W_3}$	$X_7 \times \mathbf{W_2}$
clk #8	$X_8 \times 0$	$X_8 \times \mathbf{W_6}$	$X_8 \times 0$	$X_8 \times \mathbf{W_3}$
clk #9	$X_9 \times \mathbf{W_7}$	$X_9 \times 0$	$X_9 \times \mathbf{W_4}$	$X_9 \times 0$
clk #10	$X_{10} \times \mathbf{W_8}$	$X_{10} \times \mathbf{W_7}$	$X_{10} \times \mathbf{W_5}$	$X_{10} \times \mathbf{W_4}$
clk #11	$X_{11} \times \mathbf{W_9}$	$X_{11} \times \mathbf{W_8}$	$X_{11} \times \mathbf{W_6}$	$X_{11} \times \mathbf{W_5}$
clk #12	$X_{12} \times 0$	$X_{12} \times \mathbf{W_9}$	$X_{12} \times 0$	$X_{12} \times \mathbf{W_6}$
clk #13	$X_{13} \times 0$	$X_{13} \times 0$	$X_{13} \times \mathbf{W_7}$	$X_{13} \times 0$
clk #14	$X_{14} \times 0$	$X_{14} \times 0$	$X_{14} \times \mathbf{W_8}$	$X_{14} \times \mathbf{W_7}$
clk #15	$X_{15} \times 0$	$X_{15} \times 0$	$X_{15} \times \mathbf{W_9}$	$X_{15} \times \mathbf{W_8}$
clk #16	$X_{16} \times 0$	$X_{16} \times 0$	$X_{16} \times 0$	$X_{16} \times \mathbf{W_9}$

Table 5.2 : The proposed method data flow for convolutional computation.

Clock Cycles	Neuron #1	Neuron #2	Neuron #3	Neuron #4	Neuron #5	Neuron #6	Neuron #7	Neuron #8	Neuron #9
clk #1	$\mathbf{X_1} \times W_1$	$\mathbf{X_2} \times W_2$	$\mathbf{X_3} \times W_3$	$\mathbf{X_5} \times W_4$	$\mathbf{X_6} \times W_5$	$\mathbf{X_7} \times W_6$	$\mathbf{X_9} \times W_7$	$\mathbf{X_{10}} \times W_8$	$\mathbf{X_{11}} \times W_9$
clk #2	$\mathbf{X_2} \times W_1$	$\mathbf{X_3} \times W_2$	$\mathbf{X_4} \times W_3$	$\mathbf{X_6} \times W_4$	$\mathbf{X_7} \times W_5$	$\mathbf{X_8} \times W_6$	$\mathbf{X_{10}} \times W_7$	$\mathbf{X_{11}} \times W_8$	$\mathbf{X_{12}} \times W_9$
clk #3	$\mathbf{X_5} \times W_1$	$\mathbf{X_6} \times W_2$	$\mathbf{X_7} \times W_3$	$\mathbf{X_9} \times W_4$	$\mathbf{X_{10}} \times W_5$	$\mathbf{X_{11}} \times W_6$	$\mathbf{X_{13}} \times W_7$	$\mathbf{X_{14}} \times W_8$	$\mathbf{X_{15}} \times W_9$
clk #4	$\mathbf{X_6} \times W_1$	$\mathbf{X_7} \times W_2$	$\mathbf{X_8} \times W_3$	$\mathbf{X_{10}} \times W_4$	$\mathbf{X_{11}} \times W_5$	$\mathbf{X_{12}} \times W_6$	$\mathbf{X_{14}} \times W_7$	$\mathbf{X_{15}} \times W_8$	$\mathbf{X_{16}} \times W_9$

5.2 Experimental Results

To evaluate the performance of our proposed method, we considered convolutional operation with 3 different filter sizes. The employed filter sizes are 3×3, 5×5 and, 7×7. Additionally, to compare the efficiency of our proposed method, we provided the hardware cost of *axonal-based* computation method. As an input, we considered the MNIST handwritten digit recognition data set for the convolution process, where the size of the images are 28×28 pixels.

The operation designs were described in Verilog and synthesized using Cadence Genus tool with the TSMC 40*nm* design library. Hardware implementation results are represented in Figure 5.6. As discussed in Section 5.1.1, the convolution operation is processed based on the fully-connected ANN model. According to the convolution process essence, the output pixels size decreases by increasing the size of filters.

Experimental results show our proposed method requires %14, %26 and, %38 less clock numbers for 3×3, 5×5 and, 7×7 filters, respectively, when compared to the exploited *axonal-based* model in [90].

Latency (μs) in this work denotes as a required time for the output to be obtained after the input is applied. Latency is determined as the multiplication of clock period by the number of clock cycles to obtain the ANN output. The clock period was reduced by using the re-timing technique in the synthesis tool iteratively. Due to the simplicity of our proposed method structure, the resulted clock period of our proposed method by re-timing technique is lesser than the *axonal-based* model. As a result, the latency reduction value is even greater for our proposed method, when compared to the *axonal-based* model.

According to the experimental results in Figure 5.6, our proposed method obtains the output 31%, 46% and, 49% faster for 3×3, 5×5 and, 7×7 filters, respectively, when compared to the other method. As discussed in Section 5.1.1, the filter size determines the numbers of required MAC units, and this value is negligible for our proposed method when compared to the *axonal-based* model. Contrarily to the *axonal-based* model, all the exploited MAC units are active in our proposed method. The realization

of the convolution operation by small numbers of MAC units in our proposed method yields a remarkable reduction in term of silicon area and total power dissipation. According to the experimental results in Figure 5.6, our proposed approach saves around 85% more area when compared to the *axonal-based* model.

In this study, the switching activity data required for the computation of power dissipation were generated using the test data in the simulation where the test data consists of 10000 image samples. The experimental results indicate the efficiency of the power consumption for our proposed method. According to Figure 5.6, the dissipated power of our proposed method is only 4%, 3% and, 9% of the conventional *axonal-based* model for 3, 5 and, 7 filters, respectively.

Due to remarkable reduction in latency and power consumption for our proposed method, the energy reduction reached to 98% of the *axonal-based* model as represented in Figure 5.6. We note that, the energy consumption computed as the multiplication of latency by power dissipation.

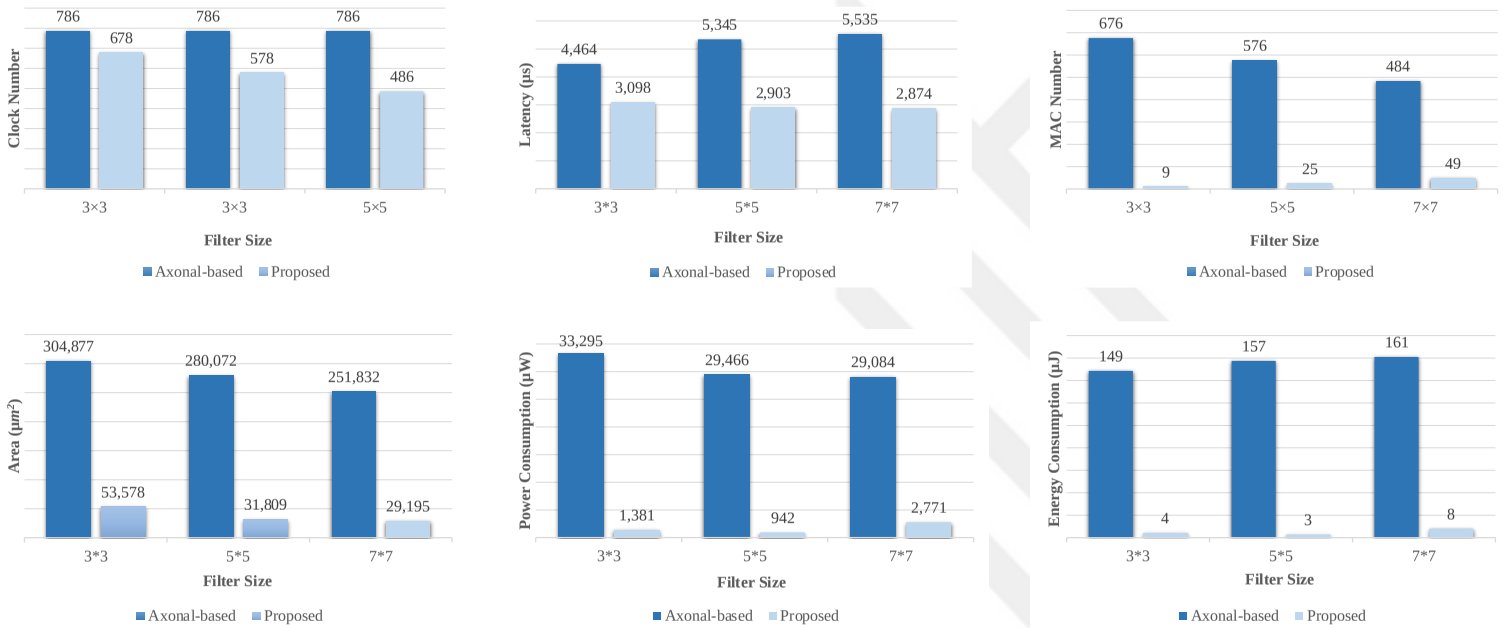

Figure 5.6 : Experimental results of the proposed method vs the *axonal-based* model.

6. CONCLUSIONS

In this thesis, we initially perform area optimization techniques for approximate ripple-carry adders and Wallace-tree multipliers to satisfy a given error constraint. Our techniques are accurate and fast, in courtesy of the proposed error calculation techniques that consider error dependencies of building blocks of adders and multipliers as well as occurrence probabilities of input assignments.

In the next step, we investigated different synthesis techniques to realize feed-forward ANNs. To reduce the bulky area of the ANNs, we discussed time-multiplexed method and developed two different time-multiplexed realization method which we called them *SMAC_NEURON* and *SMAC_ANN*. The experimental results showed that however for small ANN, *SMAC_ANN* dissipates minor power and occupies a lesser area when compared to the *SMAC_NEURON* but, this assumption is not valid for the ANN, which posses many layers or neurons. The reason is, by increasing the size of the ANN structure, controlling the MAC unit by clock cycle dominates the whole design, and the area or power for processing is negligible compared to the control unit.

By using the proposed multipliers on ANNs, we discovered that the area and energy usage in the design of ANN have decreased significantly relative to the approximate multipliers already proposed in the literature. Also, we showed that exploiting proper approximate adders based on the employed multipliers can reduce the complexity of structure without changing in the accuracy. To exploit the proposed multipliers and adders in ANNs structure based on the desired accuracy, we offered the approximate level as a novel error metric. The generation of the approximate arithmetic units based on this error metric can be done in linear times for different bit-width inputs as opposed to the other methods.

According to the experimental results, the introduced metric has a linear relationship with ANN accuracy. Furthermore, we proposed an algorithm to determine the approximate level of multipliers and adders by considering the desired accuracy. Experimental results clearly show that the use of approximate adders and multipliers

in the ANN designs reduces the design complexity significantly with the same hardware accuracy, compared to the ANN designs using exact adders and multipliers. Finally, to prove the efficiency of the proposed method in the CNN applications, we presented hardware efficient implementation of the convolution layers under the time-multiplexed architecture where computing resources are re-used using MAC blocks. The conventional MAC-based realization, which is known as the axonal-based model, suffers from high latency. Also, a high number of idle MAC units in the mentioned method yields in a leakage power dissipation. To overcome these drawbacks, we introduced a novel computing approach to speed up the convolutional computation by $2\times$ while only use roughly 2% of the area, power and, energy of the conventional MAC-based method. As future work, we plan to realize a CNN completely under this proposed structure, and obtain the efficiency of our proposed approach for the CNN in real-world applications.

REFERENCES

[1] **Gupta, V.**, **Mohapatra, D.**, **Raghunathan, A. and Roy, K.** (2013). Low-Power Digital Signal Processing Using Approximate Adders, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *32*(1), 124–137.

[2] **Yang, Z.**, **Jain, A.**, **Liang, J.**, **Han, J. and Lombardi, F.** (2013). Approximate XOR/XNOR-based adders for inexact computing, *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, pp.690–693.

[3] **Almurib, H.A.F.**, **Kumar, T.N. and Lombardi, F.** (2016). Inexact designs for approximate low power addition by cell replacement, *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp.660–665.

[4] **Momeni, A.**, **Han, J.**, **Montuschi, P. and Lombardi, F.** (2015). Design and Analysis of Approximate Compressors for Multiplication, *IEEE Transactions on Computers*, *64*(4), 984–994.

[5] **Ha, M. and Lee, S.** (2017). Multipliers with Approximate 4-2 Compressors and Error Recovery Modules, volume PP, pp.1–1.

[6] **Ercegovac, M. and Lang, T.** (2003). *Digital Arithmetic*, Morgan Kaufmann.

[7] **Aksoy, L.**, **Costa, E.**, **Flores, P. and Monteiro, J.** (2012). Multiplierless Design of Linear DSP Transforms, *VLSI-SoC: Advanced Research for Systems on Chip*, Springer Berlin Heidelberg, pp.73–93.

[8] **Li, Q.**, **Cai, W.**, **Wang, X.**, **Zhou, Y.**, **Feng, D.D. and Chen, M.** (2014). Medical image classification with convolutional neural network, *International Conference on Control Automation Robotics Vision*, pp.844–848.

[9] **Li, H.**, **Lin, Z.**, **Shen, X.**, **Brandt, J. and Hua, G.** (2015). A Convolutional Neural Network Cascade for Face Detection, *IEEE Conference on Computer Vision and Pattern Recognition*, pp.5325–5334.

[10] **Noh, H.**, **Hong, S. and Han, B.** (2015). Learning Deconvolution Network for Semantic Segmentation, *IEEE International Conference on Computer Vision*, pp.1520–1528.

[11] **Cheng, J.**, **Wang, P.s.**, **Li, G.**, **Hu, Q.h. and Lu, H.q.** (2018). Recent advances in efficient computation of deep convolutional neural networks, *Frontiers of Information Technology & Electronic Engineering*, *19*(1), 64–77.

[12] **Misra, J. and Saha, I.** (2010). Artificial neural networks in hardware: A survey of two decades of progress, *Neurocomputing*, *74*(1), 239 – 255.

[13] **Holi, J.L. and Hwang, J..** (1993). Finite precision error analysis of neural network hardware implementations, *IEEE Transactions on Computers*, *42*(3), 281–290.

[14] **Rastegari, M.**, **Ordonez, V.**, **Redmon, J. and Farhadi, A.** (2016). XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks, *Computer Vision – ECCV 2016*, Springer International Publishing, pp.525–542.

[15] **Tann, H.**, **Hashemi, S.**, **Bahar, R.I. and Reda, S.** (2017). Hardware-Software Codesign of Accurate, Multiplier-free Deep Neural Networks, *Design Automation Conference (DAC)*, pp.28:1–28:6.

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

[17] **Li, G.**, **Li, F.**, **Zhao, T. and Cheng, J.** (2018). Block convolution: Towards memory-efficient inference of large-scale CNNs on FPGA, *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp.1163–1166.

[18] **Aksoy, L.**, **da Costa, E.**, **Flores, P. and Monteiro, J.** (2008). Exact and Approximate Algorithms for the Optimization of Area and Delay in Multiple Constant Multiplications, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *27*(6), 1013–1026.

[19] **Aksoy, L.**, **Parvin, S.**, **Nojehdeh, M.E. and Altun, M.** (2020). Efficient Time-Multiplexed Realization of Feedforward Artificial Neural Networks, *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp.1–5.

[20] **Brown, B.D. and Card, H.C.** (2001). Stochastic neural computation. I. Computational elements, *IEEE Transactions on Computers*, *50*(9), 891–905.

[21] **Chen, Y.**, **Luo, T.**, **Liu, S.**, **Zhang, S.**, **He, L.**, **Wang, J.**, **Li, L.**, **Chen, T.**, **Xu, Z.**, **Sun, N. and Temam, O.** (2014). DaDianNao: A Machine-Learning Supercomputer, *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.609–622.

[22] **Lee, S.K.**, **Whatmough, P.N.**, **Brooks, D. and Wei, G.** (2019). A 16-nm Always-On DNN Processor With Adaptive Clocking and Multi-Cycle Banked SRAMs, *IEEE Journal of Solid-State Circuits*, *54*(7), 1982–1992.

[23] **Arthur, J.V.**, **Merolla, P.A.**, **Akopyan, F.**, **Alvarez, R.**, **Cassidy, A.**, **Chandra, S.**, **Esser, S.K. and Modha, D.S.** (2012). Building block of a programmable neuromorphic substrate: A digital neurosynaptic core, *International Joint Conference on Neural Networks (IJCNN)*, pp.1–8.

[24] **Akopyan, F.**, **Sawada, J.**, **Cassidy, A.**, **Alvarez-Icaza, R.**, **Arthur, J.**, **Merolla, P. and Modha, D.S.** (2015). TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *34*(10), 1537–1557.

[25] **Park, H. and Kim, T.** (2018). Structure optimizations of neuromorphic computing architectures for deep neural network, *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp.183–188.

[26] **Zhang, Q.**, **Wang, T.**, **Tian, Y.**, **Yuan, F. and Xu, Q.** (2015). ApproxANN: An Approximate Computing Framework for Artificial Neural Network, *DATE*, pp.701–706.

[27] **Esmali Nojehdeh, M.**, **Aksoy, L. and Altun, M.** (2020). Efficient Hardware Implementation of Artificial Neural Networks Using Approximate Multiply-Accumulate Blocks, *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp.96–101.

[28] **Ansari, M.S.**, **Mrazek, V.**, **Cockburn, B.F.**, **Sekanina, L.**, **Vasicek, Z. and Han, J.** (2020). Improving the Accuracy and Hardware Efficiency of Neural Networks Using Approximate Multipliers, *IEEE Transactions on Very Large Scale Integration Systems*, *28*(2), 317–328.

[29] **Mrazek, V.**, **Hrbacek, R.**, **Vasicek, Z. and Sekanina, L.** (2017). EvoApproxSb: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods, *(DATE)*, pp.258–261.

[30] **Yamada, Y.**, **Sano, T.**, **Tanabe, Y.**, **Ishigaki, Y.**, **Hosoda, S.**, **Hyuga, F. and Yoshikawa, T.** (2020). A 20.5 TOPS Multicore SoC With DNN Accelerator and Image Signal Processor for Automotive Applications, *IEEE Journal of Solid-State Circuits*, *55*(1), 120–132.

[31] **Bernasconi, A. and Ciriani, V.** (2014). 2-SPP Approximate Synthesis for Error Tolerant Applications, *Euromicro Conference on Digital System Design*, pp.411–418.

[32] **Kish, L.B.** (2002). End of Moore's law: thermal (noise) death of integration in micro and nano electronics, *Physics Letters A*, *305*(3), 144 – 149.

[33] **Gupta, V.**, **Mohapatra, D.**, **Park, S.P.**, **Raghunathan, A. and Roy, K.** (2011). IMPACT: IMPrecise adders for low-power approximate computing, *IEEE/ACM International Symposium on Low Power Electronics and Design*, pp.409–414.

[34] **Han, J. and Orshansky, M.** (2013). Approximate computing: An emerging paradigm for energy-efficient design, *2013 18th IEEE European Test Symposium (ETS)*, pp.1–6.

[35] **Hanif, M.A.**, **Hafiz, R.**, **Hasan, O. and Shafique, M.** (2017). QuAd: Design and Analysis of Quality-Area Optimal Low-Latency Approximate Adders, *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, ACM, New York, NY, USA, pp.42:1–42:6.

[36] **Jiang, H.**, **Han, J.**, **Qiao, F. and Lombardi, F.** (2016). Approximate Radix-8 Booth Multipliers for Low-Power and High-Performance Operation, *IEEE Transactions on Computers*, *65*(8), 2638–2644.

[37] **Liu, C.**, **Han, J. and Lombardi, F.** (2014). A Low-power, High-performance Approximate Multiplier with Configurable Partial Error Recovery, *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp.95:1–95:4.

[38] **Miao, J.**, **Gerstlauer, A. and Orshansky, M.** (2013). Approximate logic synthesis under general error magnitude and frequency constraints, *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp.779–786.

[39] **Shin, D. and Gupta, S.K.** (2010). Approximate Logic Synthesis for Error Tolerant Applications, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp.957–960.

[40] **Venkataramani, S.**, **Roy, K. and Raghunathan, A.** (2013). Substitute-and-simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, EDA Consortium, San Jose, CA, USA, pp.1367–1372.

[41] **Venkataramani, S.**, **Sabne, A.**, **Kozhikkottu, V.**, **Roy, K. and Raghunathan, A.** (2012). SALSA: Systematic Logic Synthesis of Approximate Circuits, *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, ACM, New York, NY, USA, pp.796–801.

[42] **Wu, Y. and Qian, W.** (2016). An Efficient Method for Multi-level Approximate Logic Synthesis Under Error Rate Constraint, *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, ACM, New York, NY, USA, pp.128:1–128:6.

[43] **Zou, C.**, **Qian, W. and Han, J.** (2015). DPALS: A dynamic programming-based algorithm for two-level approximate logic synthesis, *2015 IEEE 11th International Conference on ASIC (ASICON)*, pp.1–4.

[44] **Kulkarni, P.**, **Gupta, P. and Ercegovac, M.** (2011). Trading Accuracy for Power with an Underdesigned Multiplier Architecture, *2011 24th Internatioal Conference on VLSI Design*, pp.346–351.

[45] **Wang, Z.**, **Jullien, G.A. and Miller, W.C.** (1995). A new design technique for column compression multipliers, volume 44, pp.962–970.

[46] **King, E.J. and Swartzlander, E.E.** (1997). Data-dependent truncation scheme for parallel multipliers, *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems and Computers (Cat. No.97CB36136)*, volume 2, pp.1178–1182 vol.2.

[47] **Schulte, M.J. and Swartzlander, E.E.** (1993). Truncated multiplication with correction constant [for DSP], *Proceedings of IEEE Workshop on VLSI Signal Processing*, pp.388–396.

[48] **Hashemi, S.**, **Bahar, R.I. and Reda, S.** (2015). DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications, *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, IEEE Press, Piscataway, NJ, USA, pp.418–425.

[49] **Saadat, H.**, **Bokhari, H. and Parameswaran, S.** (2018). Minimally Biased Multipliers for Approximate Integer and Floating-Point Multiplication, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *37*(11), 2623–2635.

[50] **Ye, R.**, **Wang, T.**, **Yuan, F.**, **Kumar, R. and Xu, Q.** (2013). On reconfiguration-oriented approximate adder design and its application, *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp.48–54.

[51] **Ichihara, H.**, **Inaoka, T.**, **Iwagaki, T. and Inoue, T.** (2015). Logic simplification by minterm complement for error tolerant application, *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pp.94–100.

[52] **Scarabottolo, I.**, **Ansaloni, G. and Pozzi, L.** (2018). Circuit carving: A methodology for the design of approximate hardware, *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp.545–550.

[53] **Hashemi, S.**, **Tann, H. and Reda, S.** (2018). BLASYS: Approximate Logic Synthesis Using Boolean Matrix Factorization, *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, ACM, New York, NY, USA, pp.55:1–55:6.

[54] **Alimoglu, F. and Alpaydin, E.** (1997). Combining Multiple Representations and Classifiers for Pen-based Handwritten Digit Recognition, *International Conference on Document Analysis and Recognition*, pp.637–640.

[55] **Krizhevsky, A.**, **Sutskever, I. and Hinton, G.E.** (2012). ImageNet Classification with Deep Convolutional Neural Networks, *International Conference on Neural Information Processing Systems*, pp.1106–1114.

[56] **Misra, J. and Saha, I.** (2010). Artificial Neural Networks in Hardware: A Survey of Two Decades of Progress, *Neurocomputing*, *74*(1-3), 239–255.

[57] **Hecht-Nielsen, R.** (1990). *Neurocomputing*, Addison-Wesley.

[58] **Haykin, S.** (1999). *Neural Networks: A Comprehensive Foundation*, Prentice-Hall.

[59] **Courbariaux, M.**, **Bengio, Y. and David, J.P.** (2015). Binaryconnect: Training Deep Neural Networks with Binary Weights During Propagations, *ICNIPS*, pp.3123–3131.

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

[61] **Ding, R.**, **Liu, Z.**, **Blanton, R.D. and Marculescu, D.** (2018). Quantized Deep Neural Networks for Energy Efficient Hardware-based Inference, *Asia and South Pacific Design Automation Conference*, pp.1–8.

[62] **Sarwar, S.S.**, **Venkataramani, S.**, **Raghunathan, A. and Roy, K.** (2016). Multiplier-less Artificial Neurons Exploiting Error Resiliency for Energy-Efficient Neural Computing, *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp.145–150.

[63] **Szabo, T.**, **Antoni, L.**, **Horvath, G. and Feher, B.** (2000). A full-parallel digital implementation for pre-trained NNs, *IJCNN*, pp.49–54.

[64] **Parhi, K.** (1999). *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons.

[65] **Horowitz, M.** (2014). Computing's Energy Problem (and what we can do about it), *IEEE International Solid-State Circuits Conference*.

[66] **Aksoy, L.**, **Gunes, E.O. and Flores, P.** (2010). Search algorithms for the multiple constant multiplications problem: Exact and approximate, *Microprocessors and Microsystems, Embedded Hardware Design*, *34*(5), 151–162.

[67] **Aksoy, L.**, **Flores, P. and Monteiro, J.** (2014). ECHO: A novel method for the multiplierless design of constant array vector multiplication, *International Symposium on Circuits and Systems*, pp.1456–1459.

[68] **Paszke, A.**, **Gross, S.**, **Chintala, S.**, **Chanan, G.**, **Yang, E.**, **DeVito, Z.**, **Lin, Z.**, **Desmaison, A.**, **Antiga, L. and Lerer, A.** (2017). Automatic differentiation in PyTorch, *Conference on Neural Information Processing Systems, Autodiff Workshop*.

[69] **The MathWorks Inc.**, (2020). Deep Learning Toolbox, Natick, Massachusetts, United States, `https://www.mathworks.com/help/deeplearning/`.

[70] **Gustafsson, O.** (2007). Lower Bounds for Constant Multiplication Problems, *IEEE Transactions on Circuits and Systems II: Express Briefs*, *54*(11), 974–978.

[71] **Boullis, N. and Tisserand, A.** (2005). Some Optimizations of Hardware Multiplication by Constant Matrices, *IEEE Transactions on Computers*, *54*(10), 1271–1282.

[72] **Gustafsson, O.** (2007). A Difference Based Adder Graph Heuristic for Multiple Constant Multiplication Problems, *International Symposium on Circuits and Systems*, pp.1097–1100.

[73] **Y. Voronenko, M.P.** (2007). Multiplierless Multiple Constant Multiplication, *ACM Transactions on Algorithms*, *3*(2).

[74] **Kang, H.J. and Park, I.C.** (2001). FIR Filter Synthesis Algorithms for Minimizing the Delay and the Number of Adders, *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, *48*(8), 770–777.

[75] **Demirsoy, S.S.**, **Dempster, A.G. and Kale, I.** (2002). Power analysis of multiplier blocks, *International Symposium on Circuits and Systems*, pp.297–300.

[76] **Aksoy, L.**, **Costa, E.**, **Flores, P. and Monteiro, J.** (2010). Optimization of Area and Delay at Gate-Level in Multiple Constant Multiplications, *Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pp.3–10.

[77] **Kumm, M.**, **Hardieck, M. and Zipf, P.** (2017). Optimization of Constant Matrix Multiplication with Low Power and High Throughput, *IEEE Transactions on Computers*, *66*(12), 2072–2080.

[78] **Demirsoy, S.**, **Kale, I. and Dempster, A.** (2007). Reconfigurable Multiplier Constant Blocks: Structures, Algorithm and Applications, *Springer Circuits, Systems and Signal Processing*, *26*(6), 793–827.

[79] **Aksoy, L.**, **Flores, P. and Monteiro, J.** (2014). Multiplierless design of folded DSP blocks, *ACM Transactions on Design Automation of Electronic Systems*, *20*(1), 14:1–14:24.

[80] **Möller, K.**, **Kumm, M.**, **Kleinlein, M. and Zipf, P.** (2016). Reconfigurable constant multiplication for FPGAs, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *36*(6), 927–937.

[81] **Seo, Y. and Kim, D.** (2010). A New VLSI Architecture of Parallel Multiplier–Accumulator Based on Radix-2 Modified Booth Algorithm, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *18*(2), 201–208.

[82] **Nedjah, N.**, **da Silva, R.M.**, **Mourelle, L.M. and da Silva, M.V.C.** (2009). Dynamic MAC-based architecture of artificial neural networks suitable for hardware implementation on FPGAs, *Neurocomputing*, *72*(10), 2171 – 2179.

[83] **M.E. Nojehdeh, S.parvin, M.** (2021). Efficient Hardware Realizations of Feed-forward Artificial Neural Networks.

[84] **Kingma, D.P. and Ba, J.** (2014). Adam: A Method for Stochastic Optimization, *arXiv e-prints*, arXiv:1412.6980.

[85] **Glorot, X. and Bengio, Y.** (2010). Understanding the difficulty of training deep feedforward neural networks, *International Conference on Artificial Intelligence and Statistics*, pp.249–256.

[86] **He, K.**, **Zhang, X.**, **Ren, S. and Sun, J.** (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, *arXiv e-prints*, arXiv:1502.01852.

[87] **Nwankpa, C.**, **Ijomah, W.**, **Gachagan, A. and Marshall, S.** (2018). Activation Functions: Comparison of Trends in Practice and Research for Deep Learning, *arXiv e-prints*, arXiv:1811.03378.

[88] **Nojehdeh, M.E. and Altun, M.** (2020). Systematic synthesis of approximate adders and multipliers with accurate error calculations, *Integration*, *70*, 99 – 107.

[89] **LeCun, Y.**, **Cortes, C. and Burges, C.**, (2010), Mnist handwritten digit database. AT&T Labs.

[90] **Ardakani, A.**, **Condo, C.**, **Ahmadi, M. and Gross, W.J.** (2017). An architecture to accelerate convolution in deep neural networks, *IEEE Transactions on Circuits and Systems I: Regular Papers*, *65*(4), 1349–1362.

CURRICULUM VITAE

Name Surname : Mohammadreza Esmali Nojehdeh

Place and Date of Birth :

E-Mail :

EDUCATION :

- **B.Sc.** : 2010, Ardabil University, Electrical Engineering
- **M.Sc.** : 2015, Istanbul Technical University, Faculty of Electrical and Electronics, Department of Electrical Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2016-Now Istanbul Technical University at Emerging Circuits and Computation (ECC) Group.

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **M. E. Nojehdeh**, M. Altun (2019). Systematic synthesis of approximate adders and multipliers with accurate error calculations, Integration, ISSN 0167-9260, https://doi.org/10.1016/j.vlsi.2019.10.001.

- **M. E. Nojehdeh**, L. Aksoy and, M.Altun, (2020). Efficient Hardware Implementation of Artificial Neural Networks Using Approximate Multiply-Accumulate Blocks, 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Limassol, Cyprus, 2020, pp. 96-101, doi: 10.1109/ISVLSI49217.2020.00027.

- L. Aksoy, S. Parvin, **M. E. Nojehdeh** and M. Altun, (2020). Efficient Time-Multiplexed Realization of Feedforward Artificial Neural Networks, 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Sevilla, 2020, pp. 1-5, doi: 10.1109/ISCAS45731.2020.9181002.

- **M. E. Nojehdeh**, M. Altun, (2021). Energy Efficient Hardware Implementation of Feed-Forward artificial Neural Networks Using Approximate Arithmetic Blocks, ELSEVIER, Integration.(Under Review)
- **M. E. Nojehdeh**, S. Parvin and, M. Altun, (2021). Efficient Hardware Realizations of Feed-forward Artificial Neural Networks, ELSEVIER, Integration.(Under Review)
- **M. E. Nojehdeh**, S. Parvin and, M. Altun, (2021). Efficient Hardware Implementation of Convolution Layers Using Multiply-Accumulate Blocks, IEEE Computer Society Annual Symposium on VLSI (ISVLSI).

F. M. SURNAME	APPROXIMATE ARTIFICIAL NEURAL NETWORK HARDWARE AWARE SYNTHESIS TOOL	2020