

HARDWARE NEURAL ACCELERATOR IMPLEMENTATION FOR IoT BASED APPLICATIONS

M.Tech Thesis

By
C SANDEEP
2002102017



**DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE**

JUNE 2022

HARDWARE NEURAL ACCELERATOR IMPLEMENTATION FOR IoT BASED APPLICATIONS

A THESIS

*Submitted in partial fulfillment of the
requirements for the award of the degree
of
Master of Technology*

By
C SANDEEP



**DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE**

JUNE 2022

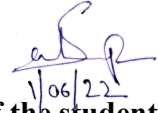


INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

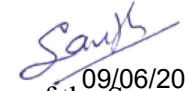
I hereby certify that the work which is being presented in the thesis entitled **HARDWARE NEURAL ACCELERATOR FOR IoT BASED APPLICATIONS** in the partial fulfillment of the requirements for the award of the degree of **MASTER OF TECHNOLOGY** and submitted in the **DEPARTMENT OF ELECTRICAL ENGINEERING, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from August 2020 to June 2022 under the supervision of **Prof. Santosh Kumar Vishvakarma, Professor, Department of Electrical Engineering, Indian Institute of Technology Indore**.

The matter presented in this thesis has not been submitted by me for the award of any other degree at this or any other institute.



Signature of the student with date
C Sandeep

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.



09/06/2022
Signature of the Supervisor of
M.Tech. thesis (with date)

Prof. Santosh Kumar Vishvakarma

Mr. C Sandeep has successfully given his M.Tech. Oral Examination held on **07/06/2022**.



Signature(s) of Supervisor(s) of M.Tech. thesis
Date: 09/06/2022



Signature of PSPC Member #1
Date: 09.06.2022



Convener, DPGC
Date: 09/06/2022



Signature of PSPC Member #2
Date: 09.06.2022

ACKNOWLEDGEMENT

I wish to thank my supervisor, Prof. Santosh Kumar Vishvakarma, Professor, IIT Indore, for his patience, guidance, and support. I have benefited greatly from their wealth of knowledge. I am extremely grateful that he took me on as a student and continued to have faith in me throughout the year. He has been very supportive since day one and I am grateful to him for devoting his time in guiding and motivating me to make the right decision. I am thankful to him for providing me with the opportunities that shaped my M.Tech to be as it is today. I would also like to thank Ph.D. scholar Gopal Raut for his technical guidance and supportive during M.Tech thesis project.

I sincerely thank my PSPC members, Prof. Ram Bilas Pachori, and Dr. Jayaprakash Murugesan, for their useful suggestions and valuable remarks during M.Tech project work. Their encouraging words and thoughtful, detailed feedback have been very important to me. I also acknowledge IIT Indore for providing me the necessary infrastructure and research facilities for the project work. I am thankful to all the faculty members of the Discipline of Electrical Engineering for their kind support during my M.Tech work.

Finally, I owe a lot to my family, friends, and classmates for always encouraging and supporting me to stay motivated and focused on completing the project.

C Sandeep

Dedicated to my
family

Abstract

HARDWARE NEURAL ACCELERATOR FOR IoT BASED APPLICATIONS

In recent years, on-chip Neural Network Accelerators have grown in popularity as one of the best algorithms for numerous key detection and classification problems in image, speech, and a variety of other never-ending applications in intelligent system design. While their on-chip presence is desirable, their high computational demands continue to be a roadblock in the development of the next generation of System on Chips. Although hardware accelerators for Convolutional Neural Networks are frequently simple designs, a traditional design approach has not been very successful, as they often require a lot of silicon area and power. Fixed point processing, according to researchers, can result in a large reduction in resource use while having a negligible influence on accuracy. Many issues that are challenging for other computational models, such as image processing, pattern recognition, prediction, and classification, are well-suited to a Convolutional Neural Network. Hardware designs can feature a more parallel structure of CNNs to improve performance or lower implementation costs, especially for applications that require high parallel computation. However, hardware platforms have a lot of unique disadvantages, including limits with high data precision, which is related to the hardware cost of the necessary computation, and the hardware implementation's lack of reconfigurability compared to software. Due to resource-intensive parts such as multipliers, current Convolutional Neural Network hardware implementations have an excess area need. The present work addresses this challenge by proposing a Co-ordinate Rotation Digital Computer (CORDIC)- based neuron architecture (RECON) implemented using In-SRAM In-Memory Computing and modified Static Manchester Carry Adder, which can be configured to compute multiply-accumulate (MAC). The CORDIC-based architecture uses linear relationship to realize MAC, whereas CORDIC algorithm uses minimum resources to realize

different mathematical operations. The CORDIC architecture is area and power-efficient with the overhead of lower throughput. In this project, we propose a Pipelined CORDIC based MAC using In-Memory Computing and modified Static Manchester Carry adder. This proposed design significantly increases the throughput by 40% when compared with conventional methods. Choosing 8-bit precision improves the power consumption and area utilization. When operated at 1V, the power consumption decreases by 38%. Hence, the proposed design offers the best throughput among the state of art and consumes lesser power and area when designed at 8-bit precision and operated at 1 V.

TABLE OF CONTENTS

TITLE PAGE.....	I
DECLARATION PAGE.....	II
ACKNOWLEDGEMENT.....	III
DEDICATION PAGE.....	IV
ABSTRACT.....	V
TABLE OF CONTENTS.....	VII
LIST OF FIGURES.....	XI
LIST OF TABLES.....	XV

Chapter 1: Introduction	1
1.1 Artificial Intelligence	1
1.1.1 Applications of Artificial Intelligence	2
1.1.2 Categorization of Artificial Intelligence	3
1.2 Machine Learning	3
1.2.1 Supervised Machine Learning algorithms	4
1.2.2 Unsupervised Learning	4
1.2.3 Semi-supervised Learning	4
1.2.4 Reinforcement Learning	5
1.3 Deep Neural Network	5
1.3.1 Neural Networks and DNNs	7
1.4 Overview of DNNs	9
1.4.1 Convolutional Neural Networks	10
1.5 DNN Development Resources	14
1.5.1 Frameworks	15
1.5.2 Popular Data Sets for Classification	15
1.6 Hardware of DNN processing	18
1.7 Machine Learning Accelerators	21
1.7.1 Heterogenous Computing Platforms	22
1.7.2 ASICs and FPGA	22
 Chapter 2: In-Memory Computing	 23
2.1 The Emergence of In-Memory Computing	23
2.2 Objective of In-Memory Computing	23

2.3 IMCs Fundamental Principles and Prominence	24
2.4 In-SRAM based IMC	26
2.4.1 6T SRAM	27
2.5 SRAM based IMC using local and global bitlines	29
2.5.1 Advantages of Fast and Reliable IMC design	31
2.5.2 $Abar.B$ generation using IMC	31
2.5.3 Read Enable and Write driver	32
 Chapter 3: CORDIC Algorithm	 35
3.1 Background	35
3.2 Algorithm	35
3.3 Accumulator Registers	39
3.4 Computation modes	40
3.4.1 Rotation mode	42
3.4.2 Vectoring mode	42
3.4.3 Arctangent	43
3.4.4 Vector Magnitude and Cartesian-Polar Transformation	43
3.5 Expanding the Computation Domain	43
 Chapter 4: Modified Static Manchester Carry Adder/Subtractor	 47
4.1 Adders and Carry-Skip Adders	49
4.2 Dynamic Manchester Carry Chain Adder	49
4.3 Modified Static Manchester Carry Chain Adder	51
 Chapter 5: MAC Design Using CORDIC Algorithm	 55
5.1 CORDIC Algorithm	56
5.1.1 CORDIC configuration to MAC	58
5.2 Pipelining to increase performance of MAC	58
5.3 Number of pipelining stages	61

Chapter 6: Results and Discussions	63
6.1 Delay Estimations	64
6.2 Power Estimations	65
6.3 Functional Verification	66
6.4 Conclusion	67
References	68

LIST OF FIGURES

Figure No.	Figure Title	Page No.
Fig. 1.1	Deep learning in the context of artificial intelligence	6
Fig. 1.2	Connections to a neuron in the brain. x_i , w_i , $f()$, and b are the activations, weights, nonlinear function, and bias, respectively	7
Fig. 1.3	Simple neural network	8
Fig. 1.4	Example of an image classification task. The machine learning platform takes in an image and outputs the confidence scores for a predefined set of classes	10
Fig. 1.5	Dimensionality of convolutions. (a) 2-D convolution in traditional image processing. (b) High dimensional convolutions in CNN	12
Fig. 1.6	Convolutional neural networks	12
Fig. 1.7	Various forms of nonlinear activation functions	14
Fig. 1.8	Various forms of pooling	15
Fig. 1.9	MNIST (10 classes, 60000 training, 10000 testing) versus ImageNet (1000 classes, 1.3 million training, 100000 testing) data set	17
Fig. 1.10	Mapping to matrix multiplication for fully connected layers(a) Matrix Vector multiplication is used when computing a single output feature map from a single input feature map. (b) Matrix Multiplications is used when computing N output feature maps from N input feature maps	20

Fig. 1.11	Mapping to matrix multiplication for convolutional layers. (a) Mapping convolution to Toeplitz matrix. (b) Extend Toeplitz matrix to multiple channels and filters	21
Fig. 2.1	Basic Principles and Significance of IMC	25
Fig. 2.2	Schematic of 6T SRAM	27
Fig. 2.3	6T SRAM based IMC schematic showing bitline computing	30
Fig. 2.4	Schematic of SRAM based IMC using local and global bit lines showing bitline computing.	30
Fig. 2.5	Variation of (a) drain current w.r.t external gate voltage of baseline JLFET, (b) internal gate voltage with external gate voltage, (c) drain current w.r.t gate voltage of NC JLFET for distinct gate lengths. Symbol (Δ) denote simulation results whereas lines (—) represent the developed model.	32
Fig. 3.1	i^{th} step iteration in cordic algorithm	37
Fig. 3.2	The CORDIC Rotation mode	40
Fig. 3.3	Cordic Vectoring Mode	42
Fig. 3.4	Functions that can be computed using the CORDIC algorithm	46
Fig. 4.1	Schematic for obtaining carry propagator(P) and sum output from global bitlines and carry input	48
Fig. 4.2	Schematic for obtaining sum output by performing XOR operation of P and C_i	48
Fig. 4.3	Schematic of Dynamic Manchester carry adder	50
Fig. 4.4	Glitches due to floating intermediate node	51

Fig. 4.5	Schematic of Dynamic Manchester carry adder	52
Fig. 4.6	G_{sub} or G_b or $(Xbar.Y)$ generation using local bitlines of proposed In-SRAM IMC architecture	54
Fig. 5.1	Sign N-bit precision recursive CORDIC architecture that can realize multiply- accumulate computation in linear mode ($m = 0$ & $E_i = 2^{-i}$)	56
Fig. 5.2	n-stage pipelined MAC design using CORDIC algorithm	57
Fig. 5.3	MUX for selecting P_0 (adder) or P_{0b} (subtractor) with select pin	60
Fig. 5.4	Generation of MAC symbol	62

LIST OF TABLES

Table No.	Table Title	Page No.
Table 4.1	Operation of Manchester Carry Adder	53
Table 4.2	Adder and subtractor outputs for X and Y operands	53
Table 5.1	For high-performance MAC operation, iteration-level calculation is given for MAC computation using CORDIC in linear mode for fixed (8, 7) representation Processing at each nth stage in pipeline architecture.	61
Table 6.1	Network Inference Accuracy	64
Table 6.3	Delay comparison for the proposed design and the state-of-the-art for MAC computation @45nm TT process corner for 8-bit precision.	65
Table 6.4	Variation of Dynamic power with voltage TT process corner for 8-bit precision	65
Table 6.5	Computation of MAC using CORDIC and IMC for fixed point representation	66
Table 6.6	Computation of MAC using CORDIC and IMC for fixed point representation for different input, bias and weight	66

Chapter 1

INTRODUCTION

1.1 Artificial Intelligence

In recent years, artificial intelligence (AI) has gained traction. The modern world has been transformed by artificial intelligence (AI) and machine learning (ML). There has been great improvement in this field during the last twenty to thirty years. AI and machine learning are now applied in a variety of industries, including cancer detection and speech recognition. The immense quantity of processing power and the vast amount of data available have enabled this phenomenal expansion. Computers can learn from data and improve themselves using AI and machine learning techniques. These algorithms are capable of foreseeing the future.

Artificial intelligence (AI) is the simulation of human intelligence in machines that are programmed to think and act like humans. AI is a term used to describe any machine that mimics human mental functions such as learning and problem-solving. Beyond artificial intelligence, rationalizing and acting with the best possibility of achieving a given goal is a desirable characteristic. Machine learning is a subset of artificial intelligence that refers to the concept of computer programs learning and adapting to new data without the need for human intervention. Deep learning technology automates learning by ingesting massive volumes of unstructured data including text, photos, and videos.

When most people hear the term "artificial intelligence," they immediately think of a robot. This is due to the fact that high-budget films and novels depict

human machines destroying the planet. Nothing, however, can conceal the reality.

Artificial intelligence is founded on the idea that human intelligence may be described in such a way that machines can readily copy and accomplish activities ranging from simple to sophisticated. The purpose of artificial intelligence is to mimic human cognitive capacities. Researchers and developers in this discipline are making rapid progress in precisely defining behaviors like learning, reasoning, and cognition. Some experts think that in the near future, developers will create systems that can learn or reason about any subject beyond human capabilities. Others, on the other hand, are skeptical because all cognitive activity have values derived from human experience.

As technology progresses, earlier artificial intelligence criteria become obsolete. Machines that calculate basic calculations or recognize text using optical character recognition, for example, are no longer called artificial intelligence because these functions are now regarded standard computer functions.

AI is constantly improving to benefit a wide range of sectors. A multidisciplinary approach based on mathematics, computer science, linguistics, psychology, and other disciplines is used to wire machines.

1.1.1 Applications of Artificial Intelligence

The possibilities for AI are infinite. This technology is utilised in numerous domains and industries. In the healthcare industry, AI is being tested and used to dispense drugs and treatments to patients as well as execute surgical procedures in the operating room.

Computer chess and self-driving automobiles are two further instances of artificial intelligence machines. Because each action has an impact on the final result, each of these machines must evaluate the results of one of the tasks.

The outcome in chess is victory. To operate in anti-collision mode, autonomous vehicles' computer systems must consider and calculate all external data.

Artificial intelligence can also be used in the finance sector. It helps banks fight fraud by detecting and marking debit cards and big account deposits that engage in unusual banking and financial activity. Trading was made easier and simpler with AI software. This is accomplished through expediting the appraisal of securities' supply, demand, and price.

1.1.2 Categorization of Artificial Intelligence

Artificial intelligence can be classified into two types: weak and powerful. Weak artificial intelligence is a system that is meant to perform a single task. Video games, such as the chess example above, and personal assistants, such as Amazon's Alexa and Apple's Siri, are examples of weak AI systems. When you ask the assistant a question, it responds. Artificial intelligence systems that are robust perform tasks that are considered human-like. These are typically more complex and difficult systems. They are programmed to handle circumstances in which they may be required to solve problems without the assistance of a human. Self-driving cars and hospital operating rooms are examples of uses for these technologies.

1.2 Machine Learning

Machine learning is an artificial intelligence (AI) technology that allows systems to learn and improve on their own, without the need for programming. Machine learning tries to create computer algorithms that can utilise data to learn for themselves. The learning process starts with observations or data, such as examples, direct experience, or instruction, in order to find patterns in data and make better decisions in the future based on the examples we provide. The basic goal is for computers to learn on their own, without the need for human involvement, and to change their behavior accordingly.

1.2.1 Supervised Machine Learning algorithms

This method includes a label identifying the desired solution for each case in the training data entered into the system. The data is primarily divided into two categories: training data and test data (score data). A model or theory is refined via training packages. This assumption is nothing more than a pedagogical mathematical statement. We examine the mistake while evaluating the model (the difference between the expected value and the value collected by the model). The goal of the training is to refine the model to achieve the lowest possible error. Training tests are used to evaluate education. The actual data can then be used to perform the appropriate function (prediction, distribution, etc.).

1.2.2 Unsupervised Learning

The polar opposite of supervised learning is unsupervised learning. The training data does not include labels in this case. The system must learn on its own. These are essentially data point grouping strategies. Let's imagine you have a lot of client information on your website or in your store. You can use the clustering method to locate a group of clients who are similar to each other. You may see which customers you've worked with the most using this method. This categorization can be used to create marketing campaigns.

Detecting discrepancies is another crucial challenge. Credit card fraud detection, industrial defect detection, and data analytics data detection and discovery are just a few examples.

1.2.3 Semi-supervised Learning

This is when the majority of the training data is unlabeled, with only a few labelled. As a result, most semi-supervised learning algorithms combine

supervised and unsupervised learning techniques. One of the better instances of supervised learning is Google Photos. If you tag a person's name on a photo, Google Photos can detect that person's face and display all of their photos.

1.2.4 Reinforcement Learning

Reinforcement learning differs significantly. This is the area where the system can work. You are awarded or punished depending on your actions. The system then changes the policy with recommended and prohibited behaviours. This process continues until you determine the best course of action for your scenario. Booster learning is commonly used by robots to learn how to walk.

1.3 Deep Neural Network

Many recent AI applications are built on the foundation of deep neural networks (DNNs). The number of applications using DNNs has exploded since the creative use of DNNs for image classification and speech recognition. DNNs are employed in a variety of applications, including self-driving automobiles and cancer diagnosis. DNNs can now outperform humans in many of these domains. DNN's outstanding performance stems from its capacity to extract high-level features from raw sensory data after statistical training on vast volumes of data in order to efficiently represent the input space. This is in contrast to the prior approach, which relied on pre-made features or regulations. Higher DNN accuracy, on the other hand, increases computational complexity. Although general-purpose computers, particularly graphics processing units (GPUs), have long formed the backbone of DNN processing, there is growing interest in giving more specialized acceleration to DNN computers. D.N.N. (also known as deep learning) is a large branch of AI, science, and technology that aims to create intelligent machines that can attain human-made goals, according to computer scientist John McCarthy. Figure 1.1 depicts deep learning relationships with all artificial intelligence.

The brain is continually being studied by scientists. Neurons, on the other hand, are commonly recognized as the brain's primary computing component. The human brain has roughly 86 billion neurons on average. As illustrated in Figure 1.2, neurons are connected to multiple pieces called dendrites and contain output elements called axons. Neurons receive signals from the dendrites and process them in order to generate signals in the axons. Activation is the term for these inputs and outputs. Neuron axons spread out and connect to the dendrites of numerous other neurons. A synapse is the connection between the axon's branches and the dendrites. The typical number of synapses in the human brain is 10^{14} to 10^{15} . The advantages of efficient machine learning algorithms are apparent. Instead of the laborious and random

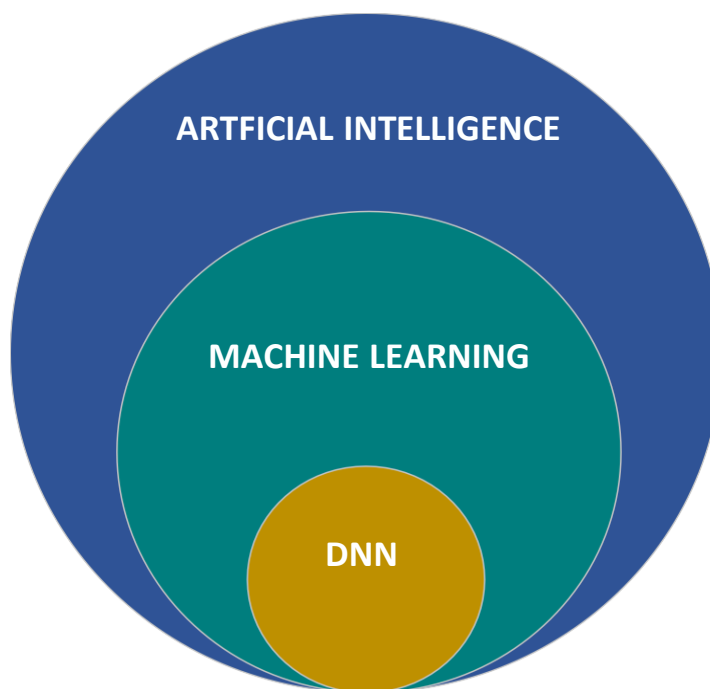


Figure 1.1: Deep learning in the context of artificial intelligence [91]

way of creating one individual program to solve each problem in the domain, a single machine learning algorithm must learn how to handle each new problem through a process called learning. A branch of machine learning known as brain-inspired computing exists. Because the brain is the most well-known "machine" for learning and problem solving, it's only reasonable to explore for machine learning techniques. Thus, brain-inspired computing is a

program or algorithm that works in some way similar to how the brain works in its most basic form. This is not an attempt to build a brain; rather, the program seeks to simulate some parts of learning how the brain works.

The synapse's most important function is to modify the scale it goes through (x_i), as seen in Figure 1.2. This evaluation actor may relate to weight (w_i). It is thought that the brain learns through changes in synaptic pressure. As a result, different input responses correspond to different weights. The tissues in the brain (what we call programs) remain the same as learning regulates pressure in response to learning cues. This property makes the brain an excellent source of ideas for machine learning systems. Peak computing is a sub-domain of the brain-inspired computing paradigm. The interaction between dendrites and axons is analogous to nail stimulation, which provides inspiration for this sub-area. The information delivered is not exclusively dependent on the spine's amplitude. Instead, it relies on the pulse's time and the fact that the neuron's calculations are based on a single value and the time link between the pulse width and additional pulses. IBM True North [65] is an example of a brain game-inspired initiative.

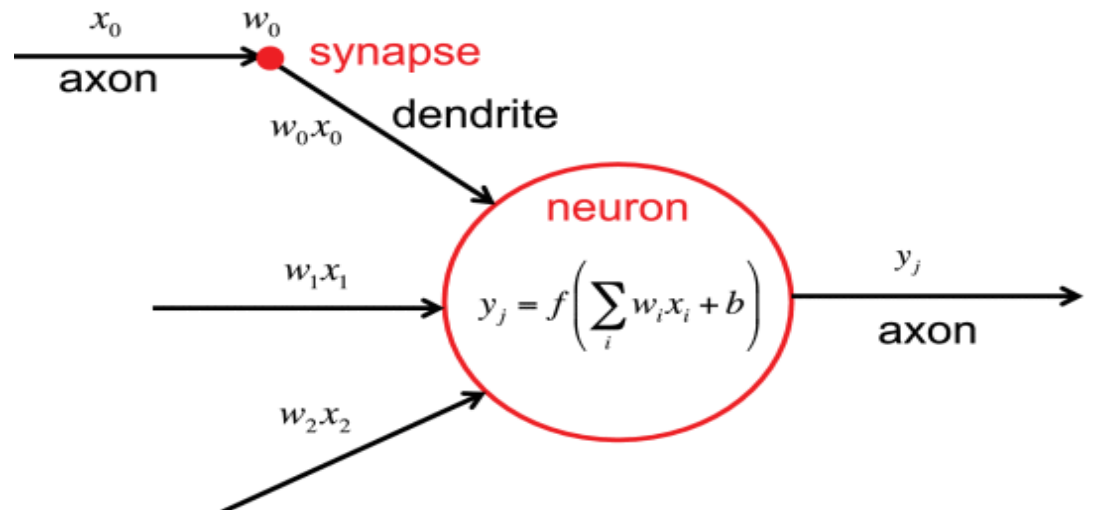


Figure 1.2: Connections to a neuron in the brain. x_i , w_i , $f()$, and b are the activations, weights, nonlinear function, and bias, respectively [91]

1.3.1 Neural Networks and DNNs

The idea of weighted inputs in neuron computations influenced neural networks. These weighted sums correspond to synaptic scaling values, which are then aggregated by neurons. Neurons only create weighted sums since the calculations involving the cascade of neurons are simple linear algebraic operations. Neurons, on the other hand, have functional operations that act on their combined inputs. This looks to be a nonlinear function, with neurons producing output only when the input exceeds a portion of the threshold. As a result, the neural network uses a nonlinear function that equals the weighted sum of the input values. A schematic example of a computational neural network is shown in Figure 1.3.

Finally, the user receives the final network output after the weighted amount is delivered from one or more hidden layers to the outgoing layer. The output of nerve cells is called activation to blend the brain-inspired phrase with a neural network. Activation/pressure designation is used in this article.

In Figure 1.3 shows an example calculation at each level: $y_j = f(\sum_{i=1}^3 W_{ij} * x_i + b)$, where W_{ij} , x_i and y_j are the weights, activate the input, activate the output. For simplicity, the phrase bias b has been removed from Figure 1.3.

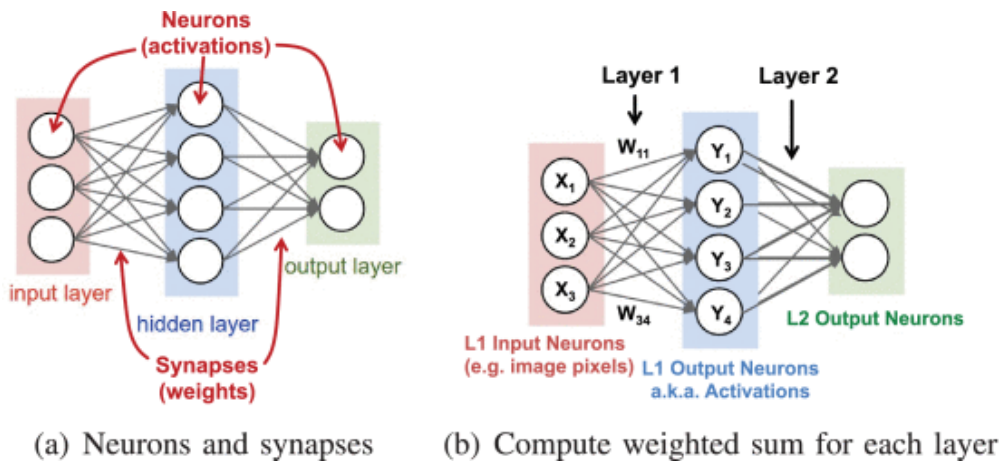


Figure 1.3: simple neural network

Deep learning is a branch of neural networks in which the neural networks comprise three or more layers, i.e. two or more hidden layers. The number

of network layers employed in deep learning today varies between 5 and over 1,000. The term Deep Neural Network (DNN) is used in this article to refer to neural networks typically employed in deep learning. DNNs can learn higher-level functions that are more complicated and abstracted than tiny neural networks. The usage of DNNs to process visual data is an example of this. Image pixels are fed into the DNN's first layer in such applications. The output of the layer can show different low-level image elements including lines and edges. These elements were blended at the next level based on the likelihood of the higher-level elements. Lines, for example, are blended into shapes, which are then combined into a set of shapes. Finally, given all of this data, the network is likely to contain specific objects or scenes. DNNs can achieve good performance in a variety of applications thanks to this deep layer of capability.

1.4 Overview of DNNs

Depending on the application, DNNs come in a range of shapes and sizes. Popular shapes and sizes are also continually evolving to improve precision and efficiency. The input to the DNN in both circumstances is a set of values that reflect the data to be evaluated by the network. Image pixels, sampling amplitudes for sound waves, or numerical representations of any system or game state are examples of these values.

Input processing networks are divided into two categories: feed and flatness. All calculations in the feedforward network are done as a sequence of actions on the previous output level. The final set of operations generates network output such as the likelihood that an image will contain a specific object, the likelihood that an audio sequence will contain a specific word, a bounding box around the object in the picture, or the steps that should be taken. The network in such a DNN has no memory. Regardless of the sequence of inputs previously delivered to the network, the input-output is always the same.

The circulatory neural network (RNN), also known as the short-term memory network (LSTM), on the other hand, has an internal memory where long-term dependency might affect outcomes. Some intermediate jobs are stored within these networks and generate values that are utilised as inputs to other input processing activities. The feed network will be the subject of this article since 1) the RNN's core calculation is still the weighted sum (i.e. matrix-vector multiplication) of the feed network. 2) Hardware acceleration of RNNs has received relatively little attention to far.

Only fully connected layers (FCs) [also known as multi-layer sensors (MLPs)] can be found in DNNs. Each output activation at the CC level comprises the sum of the weights of all input activations (i.e. all outputs are connected to all inputs). This necessitates a great deal of memory and bookkeeping. Fortunately, in many cases, the link between activations may be erased without impacting accuracy by setting the pressure to zero. As a result, a thin sticky coating is created. Bearings with thin contact.

Limit the number of weights that affect the outcome to make the calculation more efficient. Only if each output is a fixed size input window function can there be structural sparsity. Calculating each output with the same set of weights results in much improved efficiency. Weight sharing refers to the reuse of the same weight value and can greatly reduce balance sheet storage requirements.

Structured calculations in convolutions produce in a fashionable DNN layer with weight shared and windowed splitting, as shown in Figure 1.7. Only a small area of input activation is used to generate the weighted total for each output activation (i.e. all pressures out of range are set to zero). This area is also known as the "permitted area." Furthermore, all outputs use the same set of weights (e.g. the filter is an unchanged space). CONV stands for persuasion-based layers (Conv layers).

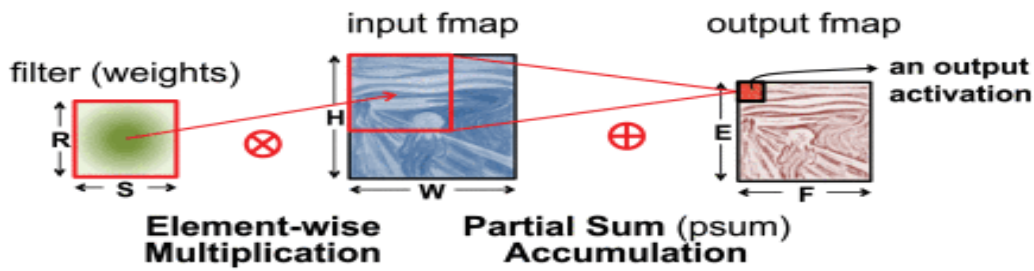
1.4.1 Convolutional Neural Networks

As shown in Figure 1.6, a Converged Neural Network (CNN) is a

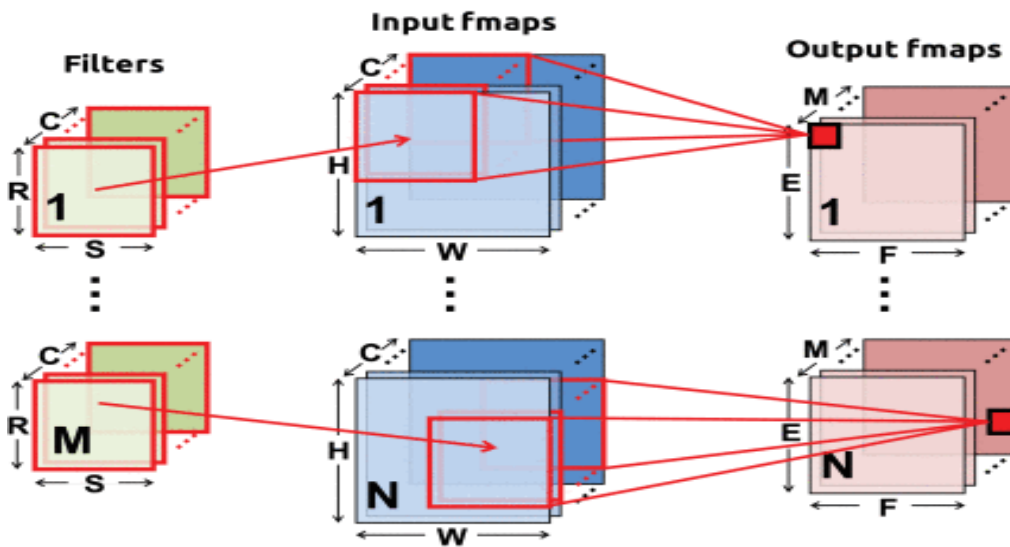
common type of DNN made up of multiple CONV layers. Each layer in such a network produces a function map, which is a higher abstraction of input data (fmap). It saves crucial but unique data. Using very deep hierarchies, modern CNNs can achieve good performance. Image recognition [48], speech recognition [82], gaming [88], and robotics [55] are just a few of the areas where CNNs are employed.

As demonstrated in Figure 1.7, each CONV layer in a CNN is mostly made up of multidimensional beliefs. The layer input operation in this calculation is made up of two-dimensional input function maps (ifmaps), each of which is referred to as a channel. Each channel is combined into a filter bank and a separate 2D filter. A single 3D filter is typically referred to as a collection of 2D filters. The conviction findings are summarized in each channel at each point. You can also give the filter results a one-dimensional offset. Some modern networks [34] do not allow them to be used in storage compartments. This calculation is the activation of the output data in the output function map., which forms a channel (ofmap).

Additional output channels for the same input can be formed using additional 3D filters. Finally, multi-input function maps can be combined into a package that reuses filter weights.



(a) 2-D convolution in traditional image processing



(b) High dimensional convolutions in CNNs

Figure 1.5: Dimensionality of convolutions. (a) 2-D convolution in traditional image processing. (b) High dimensional convolutions in CNN [91]

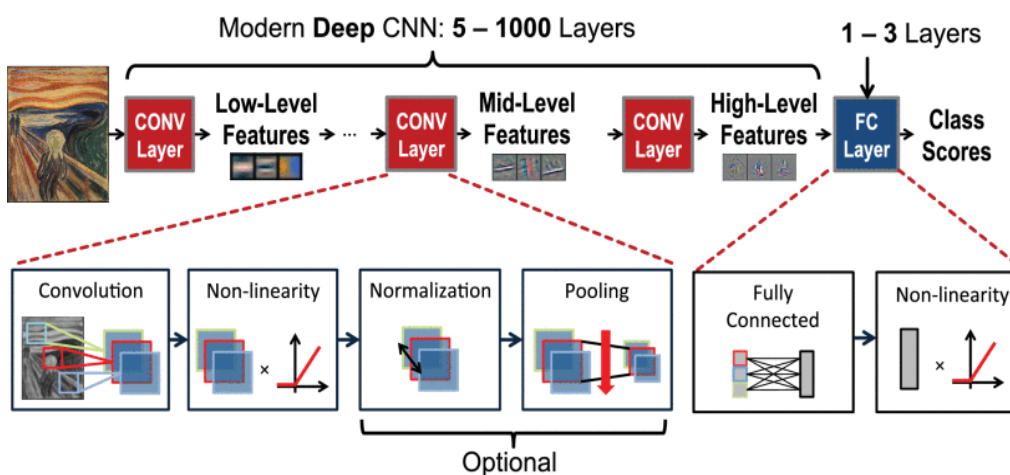


Figure 1.6: Convolutional neural networks

DNNs feature extra levels such as nonlinearity, aggregation, and normalization in addition to the CONV and CC levels. Each of these layers' functions and calculations are outlined below.

1)Nonlinearity: After each CONV or CC level, the nonlinear activation function is frequently used. Figure 1.7 shows how to introduce nonlinearities for DNNs using various nonlinear functions. Traditional nonlinear functions like sigmoid or hyperbolic tangent, as well as modified linear units (ReLU) [68], have recently gained prominence. Because of its simplicity and capacity to deliver quick learning, it has been popular throughout time. To increase accuracy, many ReLU variations such as ReLU leakage [62], parametric ReLU [35], and exponential LU [16] have been examined. Finally, in speech recognition difficulties, a nonlinearity known as max out, which takes the maximum value of two intersecting linear functions, is effective [107], [108]

2)Pooling: A unit is a calculation of variables that reduces the dimension of a function map. The network can endure slight changes and distortions because to individual channel bonding. Unions reduce the number of values in the receiving field by combining or matching them. It can be changed to fit the size of the receiving fields (e.g. 2x2) and join procedures indicated in Figure 1.10. (e.g. max or average). When blocks, not blocks, overlap, interconnection occurs (i.e. the steps are the same). Pool Dimensions) One or more summits are usually used, resulting in limited views (e.g. functional maps).

3)Normalization: Managing the distribution of input data across multiple layers can significantly speeding up training and increase accuracy. As a result, the layer's input activation distribution (σ , μ)is normalized to have a mean value of 0 and standard deviation. In batch normalization (BN), the normalized values are shifted and scaled, as shown in the figure. Here the parameters (γ , θ)

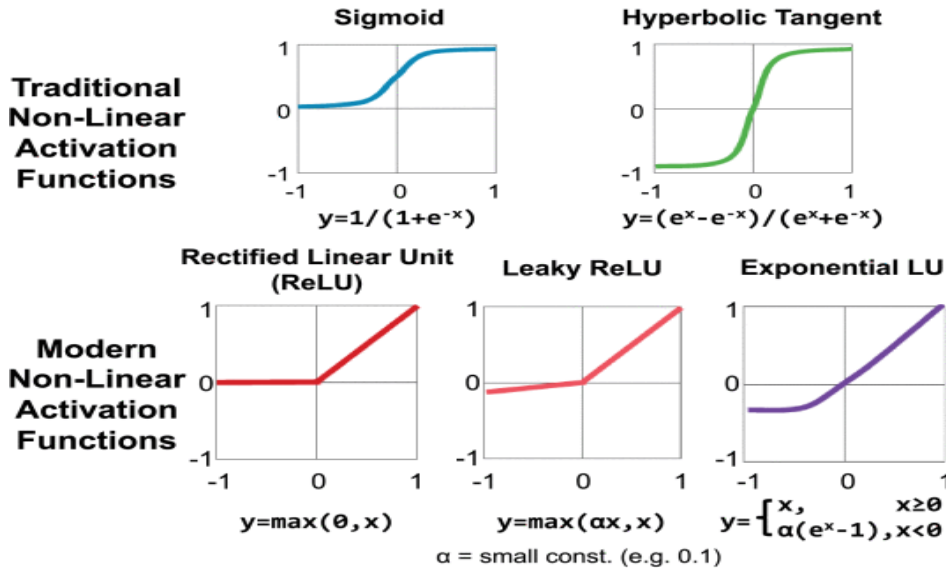


Figure 1.7: Various forms of nonlinear activation functions

is learned during training. is a small constant to avoid numerical problems. In the past, local response normalization (LRN) was used, inspired by lateral neuroscience suppression. The nerve cells fired here (i.e., high-value activation) must weaken neighboring cells (i.e., stimulate low-value activation). However, BN is now considered standard practice in CNN design, while LRN is mainly out of date. LRN is usually done after the nonlinear function. However, BN is mainly done between the CONV or CC level and the nonlinear function. Suppose BN is implemented immediately after the CONV or CC level. In that case, its calculation can be combined with the CONV or CC level pressure without further calculation.

1.5 DNN Development Resources

Several deep learning frameworks have been developed to make DNN creation easier and to share trained networks from multiple sources. The DNN programming library is one of these open-source libraries. The University of California at Berkeley (UC Berkeley) [42] opened Caffe in 2014. C, C++, Python, and MATLAB are all supported. Google introduced Tensorflow in 2015, which supported C++ and Python. It is more flexible than Caffe and supports multiprocessors and GPUs, with computations described as dataflow

charts to handle tensors (multidimensional matrices). Torch, created by Mozilla, is another prominent framework.

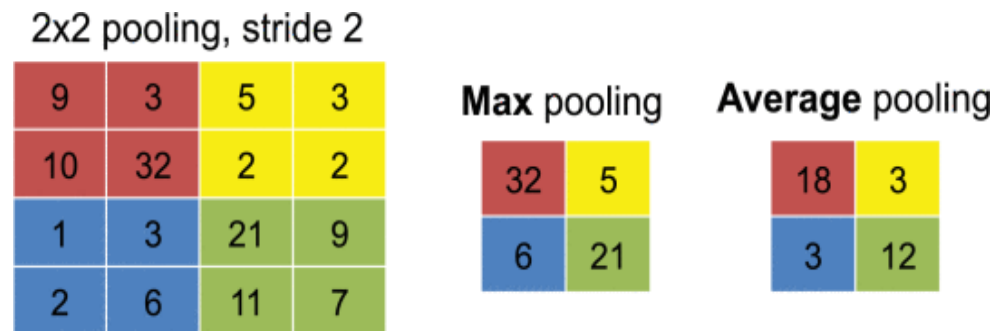


Figure 1.8: . Various forms of pooling [91]

1.5.1 Frameworks

The existence of these Frameworks is beneficial not only to DNN researchers and developers. It is, nonetheless, critical for the development of high-performance or more efficient DNN computing processes. Because they make substantial use of a set of basic operations such as CONV level manipulation, frames can include optimised software or hardware accelerators. The frame user is unaware of the acceleration. Most frames, for example, can take advantage of Nvidia's cuVNN library to operate Nvidia GPUs quickly. Unique hardware accelerators can also be transparently integrated, as with the Eyeriss chip [11].

Finally, these Frameworks provide hardware researchers with a rich source of workload. It can create prototype projects for various workloads, profile various workloads, and investigate software and hardware trade-offs.

1.5.2 Popular Data Sets for Classification

When comparing different DNN models, it's vital to keep the data complexity in mind. For example, in the MNIST dataset [51], writing numbers is significantly easier than assigning an object to one of the 1000 classes

required in the ImageNet dataset [81]. (Figure 1.9). For complex tasks, the DNN size (number of weights) and number of MACs are predicted to be bigger than for simple computations, requiring more power and bandwidth. For instance, LeNet-5 [53] is intended for image classification, while AlexNet [48], VGG-16 [89], GoogLeNet [94], and ResNet [34] are intended for image classification.

There is a large amount of AI data accessible that may be used to assess the accuracy of a certain DNN. Comparing the accuracy of various methods requires public data sets. Classifying your images is the simplest and most common process. To put it another way, select the 1N class that the image belongs to. There is no location or way to locate it.

MNIST is a numerical classification system that was introduced in 1998 [51]. It has a handwritten number image of 28x28 pixels. There are ten lessons (10 digits), 60,000 instructional images, and 10,000 test images in all. When MNIST was originally introduced, LeNet-5 had a 99.05 percent accuracy rate. Drop Connect has improved neural network management accuracy to 99.79 percent [99] since then. As a result, MNIST is currently considered a fundamental data collection.

CIFAR is a 2009 dataset of 32 by 32-pixel colour photographs of a variety of things [47]. CIFAR is a subset of 80 million Tiny Images data [95]. CIFAR-10 has ten classifications that are mutually exclusive. 50,000 class photographs (5,000 per class) and 10,000 test photos are available (1,000 per class). When it was first introduced, the two-part deep belief network had a CIFAR-10 accuracy of 64.84 percent [46]. Since then, using fractional max join [67], the accuracy has been increased to 96.53 percent.

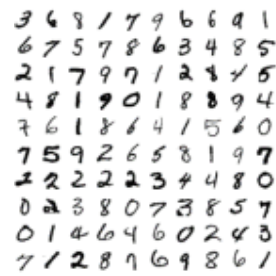
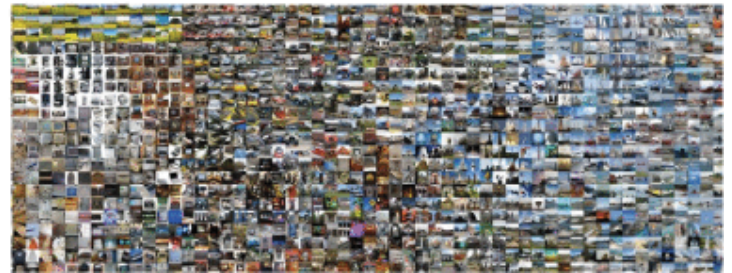
MNIST**ImageNet**

Figure 1.9: MNIST (10 classes, 60000 training, 10000 testing) versus ImageNet (1000 classes, 1.3 million training, 100000 testing) data set

ImageNet is a massive data set that was originally released in 2010. Dataset from 2012. It has 1000 classes and a 256 256-pixel colour picture. The basis for grouping synonyms and manipulating ambiguous words in the same object category is WordNet classification. That is, ImageNet categories are organized in a hierarchy. The ImageNet hierarchy has been divided into 1,000 classes to avoid any overlap. There are many subcategories in the ImageNet dataset, including 120 different dog breeds. I have 1.3 million command images (732-1300 per class), 100,000 test photos (100 per class), and 50,000 test images in total (50 per class). The ImageNet Challenge reports the accuracy of the picture categorization task using two measures: Top-5 and Top-1 error.

The top 5 error indicates that if one of the top 5-point categories is the correct one, the classification is correct. Top 1 must be the category with the highest score. In 2012, the ImageNet Challenge winner had an accuracy of 83.6 percent in the top five categories (much better than 73.8 percent, second place this year without using DNN). Among 2017, the best accuracy in the top five was 97.7%. MNIST is a rather basic data collection, as evidenced by the summary of the various picture categorization data sets. ImageNet is a large data set with a wide range of classes. As a result, while assessing the accuracy of a given DNN, it's critical to consider the amount of data being evaluated.

1.6 Hardware of DNN Processing

Many current hardware systems offer particular features for controlling DNNs as a result of their popularity. The Intel Knights Mill processor, for example, has unique deep learning vector instructions [20], and the Nvidia PASCAL GP100 GPU offers 16-bit Floating Point Count (FP16), which speeds up deep learning computations by doing two FP16 operations on a single high-precision core. There are also systems created expressly to manage DNNs, such as the Nvidia DGX-1 server and a specialised Facebook Big Basin DNN server [18]. DNN pins have also been seen in programmable gate arrays and other chip embedded (SoC) systems as as Nvidia Tegra and Samsung Exynos (FPGAs). As a result, it's critical to understand how these platforms handle data and how application-specific DNN accelerators can be created to boost throughput and reduce energy consumption.

MAC (multiplication and accumulation operation), which may be easily parallelized, is the essential component of CONV and FC bearings. To achieve high performance, an extremely parallel data paradigm is frequently utilised, including temporal and spatial structures, as shown in Figure 1.12. The timing architecture is mostly implemented on the CPU or GPU, and it employs vector (SIMD) or parallel approaches to increase concurrency. Yarn (SIMT). The central control of many ALUs is used in this ad hoc architecture.

These ALUs can only receive data from other ALUs in the memory hierarchy and cannot communicate directly. The spatial architecture, on the other hand, employs data flow processing. In other words, the ALU can build a processing chain, allowing data to flow from one location to another. A memo file or registry file can be used to store control logic and local memory for each LA. The Processing Machine refers to an ALU with local memory (PE). In ASIC and FPGA designs, spatial architecture is frequently employed for DNNs. This section discusses various design concepts for efficient machining on various platforms without compromising precision (i.e. all methods in this section give

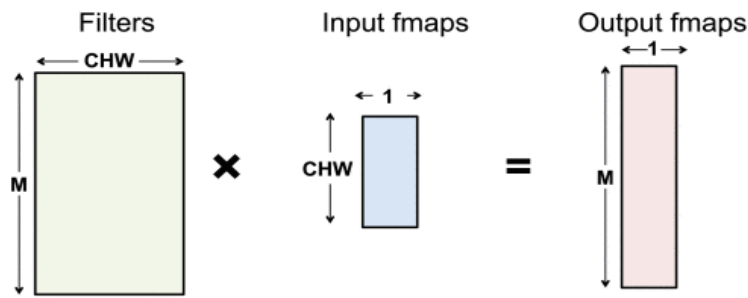
the same result). We explore how the computational transformation of the core minimizes the amount of multiplications to boost throughput for ad hoc architectures like CPU's and GPU's. We examine how data streams improve energy consumption by enhancing the memory hierarchy's reuse of inexpensive memory data for spatial topologies utilized in accelerators.

1.6.1 Accelerate Kernel Computation on CPU and GPU Platforms

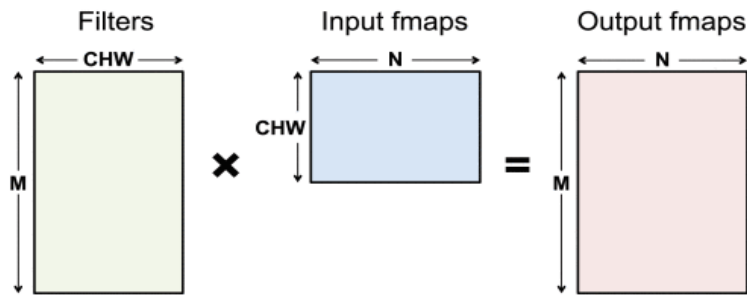
Parallel technologies such as SIMD or SIMT are used to run MACs in parallel on the CPU and GPU. The same controls and memory are used by all LAs (registration file). The CC and CONV levels are frequently translated to matrix multiplication on various platforms (i.e. core computing). Matrix multiplication is shown in Figure 1.13 for NC layers. The number of 3D filters determines the filter matrix's height. The width is the number of weights per 3D filter [Input Channel (C) \times Width (W) \times Height (H), R = W and S = H are CC layers]; The height of the input function map matrix is the number of activations per 3D input function map (C \times W \times H). The width is the number of 3D input function maps [Figure 1.13]; Finally, the matrix height of the output function map is the number of channels in the output function map (M). The width is the number of three-dimensional (N) output function maps. Each function here has a map function. CC layer dimension 1 \times 1 \times number of output channels (M).

The CONV layer of the DNN can also be mapped to multiply the matrix with the weaker Toeplitz matrix depicted in the figure. 19. The input function map contains redundant data when matrix multiplication is used on the CONV layer. Figure 1.14 shows the matrix. This can result in inefficient storage or complicated memory access patterns.

The software library is optimised for matrix multiplication processors (such as Open-BLAS, Intel MKL, and GPUs) (e.g. cuBLAS, cuDNN, etc.). Matrix multiplication refers to the storage tier of these platforms, which is measured in megabytes at the highest level.

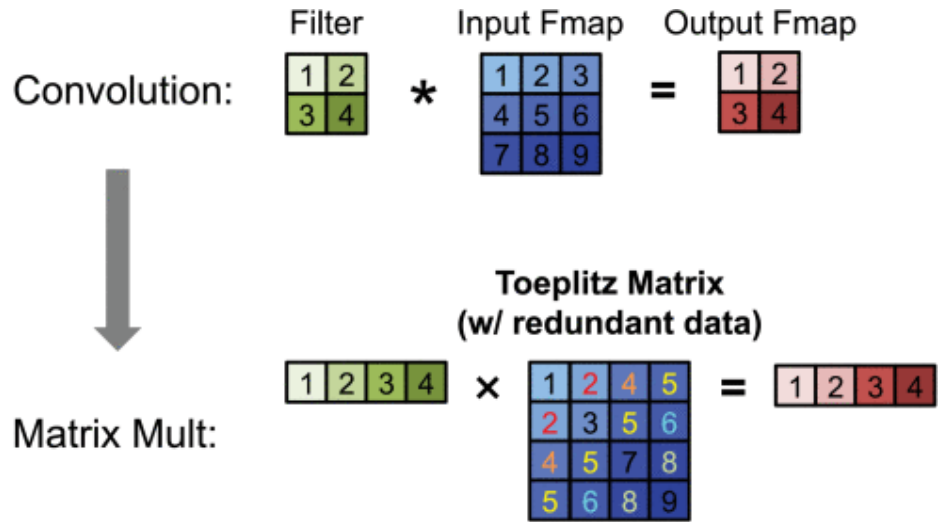


(a) Matrix Vector multiplication is used when computing a single output feature map from a single input feature map.

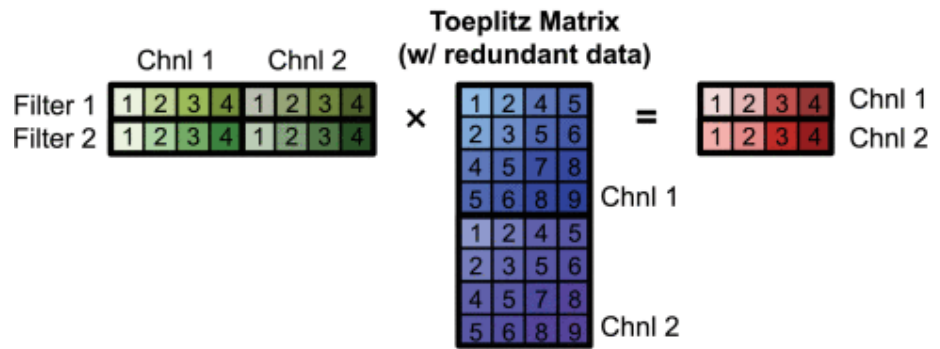


(b) Matrix Multiplications is used when computing N output feature maps from N input feature maps.

Figure 1.10: Mapping to matrix multiplication for fully connected layers
 (a) Matrix Vector multiplication is used when computing a single output feature map from a single input feature map.
 (b) Matrix Multiplications is used when computing N output feature maps from N input feature maps



(a) Mapping convolution to Toeplitz matrix



(b) Extend Toeplitz matrix to multiple channels and filters

Figure 1.11: Mapping to matrix multiplication for convolutional layers. (a) Mapping convolution to Toeplitz matrix. (b) Extend Toeplitz matrix to multiple channels and filters

1.7 Machine Learning Accelerators

Machine learning algorithms, particularly Neural Networks, are processed or co-processed by ML accelerators. Earlier attempts at machine learning accelerators used digital signal processing to do pixel-to-pixel multiplication and accumulation. On computers, graphic processing units, or GPUs, provide acceleration. More specialized chips are being developed since the development of CNN. A quick explanation of various ML accelerator methodologies follows.

1.7.1 Heterogeneous Computing Platforms

Heterogeneous computing refers to a platform that combines many types of dedicated hardware, such as a CPU with GPU and DSPs. Over a single chip, an FPGA with a processor is embedded. Because the technique of ANN or CNN is similar to that of image processing, GPUs, which are built to handle image and video processing, are particularly efficient in implementing these algorithms. However, there are other trade-offs, including expense, area, and space. Because GPUs cannot be used as coprocessors in smartphone SoCs, the industry is collaborating with ASICs to integrate them into their platforms.

1.7.2 ASICs and FPGA

In the research field of ML accelerators, ASIC and FPGA are obvious alternatives. FPGAs are commonly utilised in servers and other applications. Furthermore, newer FPGA chips contain a processor, enabling for SW-HW co-design. Intel created Nervana and Movidius, two ASICs. NVIDIA has a division dedicated to smart vehicle hardware development. Novel Hardware for Artificial Neural Networks for FPGA or ASICs is discussed in the following chapters.

Chapter 2

IN MEMORY COMPUTING

The approach of performing computer calculations entirely in computer memory is known as **in-memory computation (or in-memory computing)** (e.g., in RAM). This word usually refers to large-scale, sophisticated calculations that necessitate specialized systems software to execute on multiple computers in a cluster. As part of a cluster, the computers pool their RAM so that the calculation is essentially done over multiple computers and takes advantage of the combined RAM space of all the machines.

In-Memory Computing permits for remarkable results (multiple times faster) and indeed the extent of never-ending amounts of data, but also greater accessibility to an increase in information sources. It provides real-time insights by backing up through RAM but also encoding it in parallel, allowing providers to connect immediate responses to events. This creates opportunities that could be used in descriptive and predictive programs that share the same technology platform, as well as transaction - oriented data management guided by legit analytics.

2.1 The Emergence of In-Memory Computing

In-memory computing, or IMC, is becoming increasingly prevalent. This is attributable to higher demand for accelerated big information processing and predictive analysis, the need to refine layout as the number of data references rises, and advancements in technology that are lowering TCO.

2.2 Objective of In-Memory Computing

To achieve a sustainable competitive advantage and fulfil present and future

needs for highest quality of service, manufacturers must cope with the steady increase in the efficiency statistics and the never-ending perceptions for faster and accurate performance.

As a result, In-Memory cloud services technologies are gaining traction. Because the primary objective of In-Memory computing is to acquire and scrutinize a vast amount of information in a brief period.

Traditional business intelligence (BI) programs, which are typically focused on disc storage and database systems which use the SQL database language, are impractical for today's BI requirements, that include super-fast query processing and reliable data scalability.

In-memory processing relies solely on data stored in RAM and eliminates all slow data accesses. By eliminating the latency that is common when accessing hard disc drives or SSDs, overall computing performance is considerably enhanced. The calculation as well as the data in memory are managed by software running on one or more computers, and in the event of multiple computers, the programme breaks the computation into smaller jobs that are sent to each computer to run in parallel. In-memory computing is frequently performed using in-memory data grids (IMDG). Hazelcast IMDG is one such example, which allows users to conduct sophisticated computations on massive data sets over a cluster of hardware servers at breakneck speed.

2.3 IMC's Fundamental Principles and Prominence

Data storage and expandability — a software's, network's, or application's capacity to maintain consistently advancing amounts of information, or its potential to be elastically enhanced to accommodate that expansion — are the foundations of in-memory computing. This is accomplished using two major technologies: Parallelization and random-access memory (RAM).

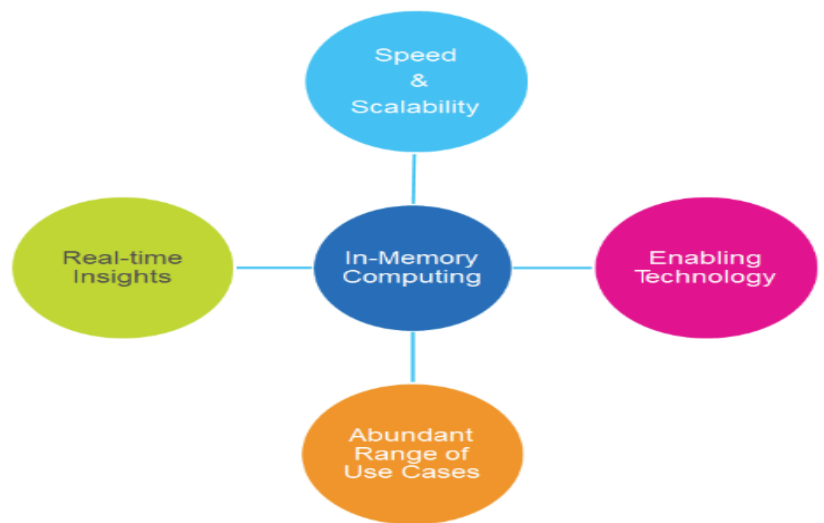


Figure 2.1: Basic Principles and Significance of IMC

High Speed and Scalability: In-Memory Computer technology exhibits excellent functionality and reliability by gathering data in RAM and archiving it. This is more than 100 times faster than any other solution for data processing and querying, guaranteeing effective and unparalleled functionality for any task.

For scalability, In-Memory Computing hinges on parallelized distributed processing, which is vital for massive data processing. Distributed data transformation, as opposed to a single, centralized database managing and providing computing technology to all linked systems, is a computer-networking platform where several computers located in different locations share computing resources.

Real-time Insights: IMC incorporates business logic, predictive analysis, and collected from a range of source information (multi-model store). In this context, In-Memory Computing is so much more than just providing a faster analysis than before; but also, about becoming significant predictors during the analysis process!

By addressing massive amounts of broadcasting, hot, and historical data in real - time, IMC enables for real-time experienced data analysis and machine learning for quick perspectives that are instantaneously used by networked

business logic with the memory fabric. Whenever something ends up happening that has the potential to impact company's operations, customer behaviour, regulatory compliance, and other factors, an instant acknowledgment of the effects and consequences is provided, allowing for quick reaction and decision-making.

Furthermore, consistent predictive modelling, which takes advantage of the opportunity to process and analyse millions of events per second, helps to minimize undesirable consequences such as equipment breakdown, customer churn, and computer hackers, among other things.

A Wide Range of Applications: Of course, in-memory computing is beneficial to companies that deal with large amounts of data, particularly those in consumer-facing sectors such as retail, financial sectors, insurance, transportation, telecommunications, and utilities. Risk and transaction management in financial institutions, payment and insurance fraud detection, consumer product trade promotion simulations, and real-time/personalized advertising are all examples. In-Memory Computing, on the other hand, can profit greatly any industry or market where proper analysis, perspectives, and predictions based on broadcasting and historical data deliver economic potential, such as geographic information analysis, preventive analytics, and network optimization in transportation.

Enabling Technology: Without the implementation of IMC, many of today's methods and services would not be possible. Applications that use blockchain based (which enables digital information to be distributed but not duplicated) or apps that use geospatial/GIS processing for transportation contain this enabling technology function (such as real-time direction on recommended route, traffic congestion and hazard).

2.4 SRAM Based IMC

Data transport consumes over a hundred times as much energy as math. As a result, data-movement overheads have significantly limited the performance of traditional "von-Neumann type" compute-centric processors.

When using SRAM-IMC in pattern-matching applications, the automata state transitions take place entirely within the memory. This eliminates the expense of division mispredictions and unusual memory accesses in CPU-based processing. We can also use a connector with high fan-in and fan-out to connect an input vector with several candidate sequences. SRAM-IMC can thus provide significant parallelism. Operations can help with cryptographic techniques, graph indexing, and database applications. Executing search and comparison activities in memory can help compression, encrypting, and search engines. IMC-based logical procedures can help with cryptography, graph indexing, and database applications.

The IMC based on SRAM can use 6T, 8T, or 10T SRAM. However, there are serious concerns with read disturbance and read noise margin erosion. The next sections go over several SRAM architectures.

2.4.1 6T SRAM

SRAM known as static random-access memory and provides bistable latching circuitry to store each bit. The term "static RAM" distinguishes it from "dynamic RAM," which includes constant replenishment. SRAM can store data, but it is still volatile in the sense that data will be lost when the memory is turned off.

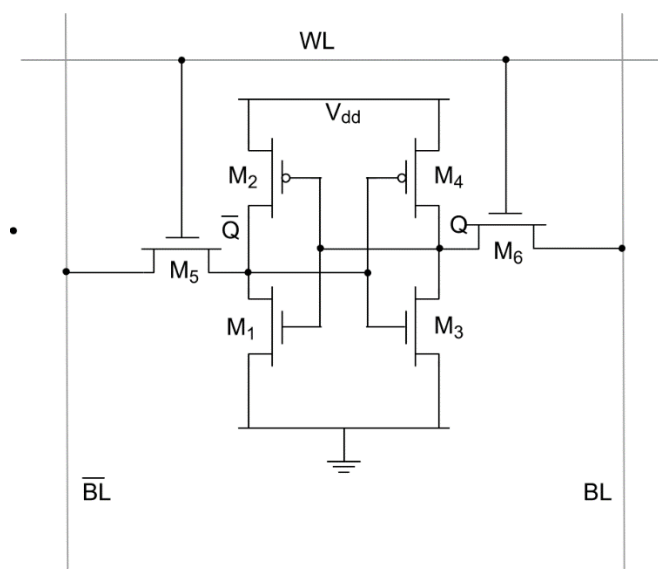


Fig 2.2 Schematic of 6T SRAM

Three modalities are available for the 6T SRAM cell: (1) Standby mode (2) Read Operation

(3) Write Operation

(1) Standby mode

Word line is not asserted in standby mode (word line=0), so the pass transistors M5 and M6, which connect the 6t cell to the bit lines, are disabled. This indicates that the cell can't be accessed. As long as the two cross-coupled inverters are connected to the supply, they will continue to feedback each other, and data will remain in the latch.

(2) Read Operation

In read mode, the word line (word line=1) is asserted, enabling both the access transistors that connect the cell to the bit lines. Values in nodes (Q and Q bar) are now moved to bit lines. Assume 1 is stored at node 'Q,' therefore the bit line bar will discharge through the driver transistor (M1) and is pulled up through the Load transistors (M2) toward VDD, resulting in a logical 1. SRAM cell design necessitates read stability (do not disturb data when reading).

(3) Write Operation

Assume that the cell was originally set to store a 1 and that we want to change it to a 0. To do so, lower the bit line to 0V and elevate the bit bar to VDD, then pick the cell by elevating the word line to VDD.

Inverter threshold is often fixed at $V_{DD}/2$ since each inverter is designed to match PMOS and NMOS. M5 functions in saturation if we want to write 0 at node a. Its source voltage is set at 1 at first. M3's drain terminal is originally set to 1, but M5 pulls it down because M5 is a stronger access transistor than

M1. Now that M3 is on and M2 is off, a new value has been written, causing the bit line to be dropped to 0V and the bit bar to be set to VDD. To operate in write mode, SRAM must have write-ability, which is defined as the minimal bit line voltage required to flip a cell's state.

Disadvantages of 6T SRAM

Two separate back-to-back inverters (two pull-up PMOS and two pull-down NMOS transistors) and two NMOS connect transistors associated to the bitlines with the gates linked to the wordline can be seen in Figure 2.2. During read, the wordline has been stated, and the change in voltage between bitlines is detected by a sensing amplifier. In the read cycle, access transistors and pull-down transistors are being used. Greater pull-down transistors (PDL and PDR) and poorer access transistors enhance RSNM. On the other hand, WM is improved by better access transistors and poorer pull-up transistors. Because of the added capacity, the SRAM cell can implement at quite low supply voltages (i.e. low VDDmin) with negligible threshold voltage volatility, albeit at the cost of much more space. SRAM cell stability had already deteriorated markedly, particularly at lower voltages, as process variations through sub-100 nm technologies continue to grow. 6T SRAM memory-based IMC solutions struggle with read disturb information leakage, restricting profitability for low-power or high-performance applications.

2.5 SRAM based IMC using local and global bit lines

Following a precharge phase to Vdd, IMC operations can be performed in standard SRAM memories by additively accessing two WLs. Based on the states of the available bitcells, either of the BLs are discharged, as shown in Figure 2.3. Finally, the outcome of an AND implementation is contained in the BL (or BL bar) (respectively NOR). These processes are the foundation of

bitline computing. As implemented to typical 6T bitcells in contrary states, the PMOS transistor of one bitcell seems to be poorer than that of the connectivity and pulldown transistors of all other bitcell, the cell may swap, associated with information corruption. This impact can indeed be illustrated using variation models, notably as in slowP-FastN CMOS corner.

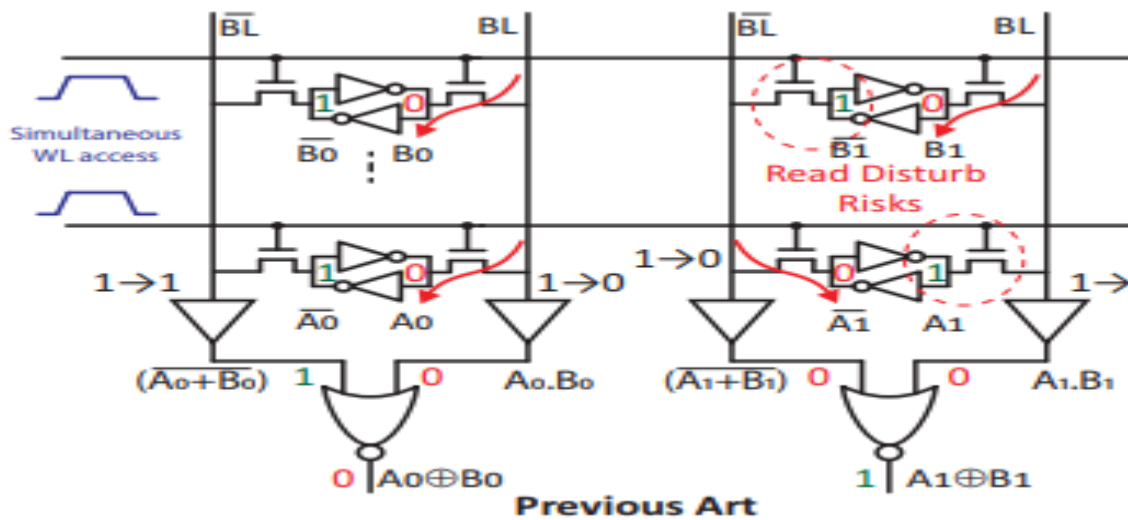


Fig 2.3: 6T SRAM based IMC schematic showing bitline computing

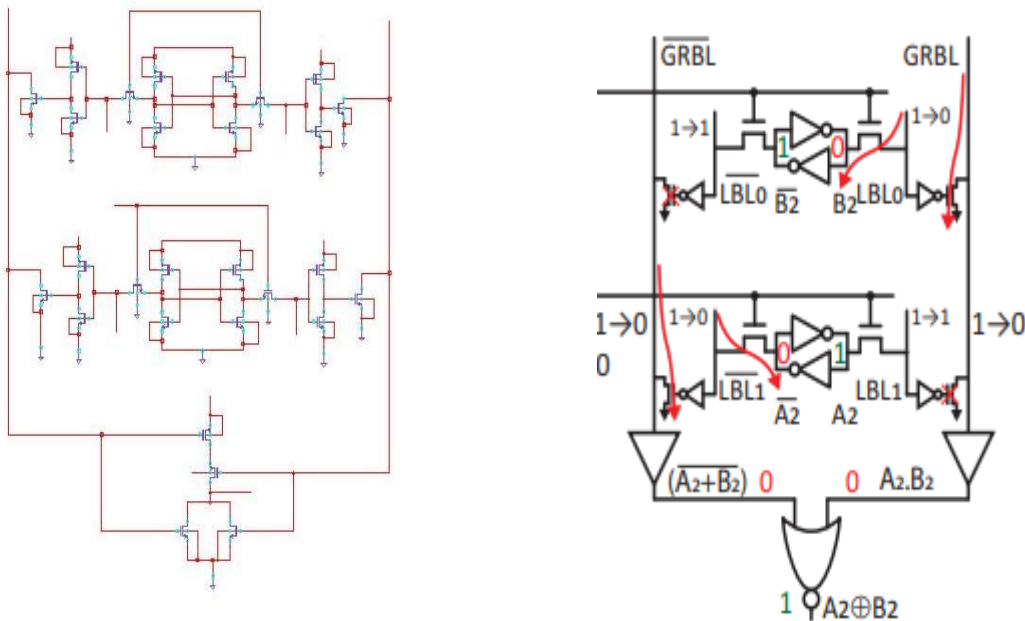


Fig2.4: Schematic of SRAM based IMC using local and global bit lines showing bitline computing.

SRAM centered IMC utilizes local and global bit lines to perform bitline computing-based IMC activities with 6T SRAM bitcells while ignoring read disturb issues. The two major bitcells in this architecture have always been connected to multiple LBLs, as shown in fig 2.4, reducing the risk of bitcell shorting. Within an LBL, an IMC procedure is much like a conventional read. Both of the GRBLs then are discharged through the read ports, based on the states of the convenient bitcells. As earlier stated, the GRBL (resp. GRBL) presents the conclusion of an AND (resp. NOR) in between two operands. This method allows the WL to cover its entire dynamic voltage range while avoiding the use of 10T bitcells.

2.5.1 Advantages of fast and reliable IMC design

With this innovative in-SRAM IMC design, you can perform bitwise, addition, shift, and copy activities in memory. It could be used to make a 28nm mass elevated CMOS technology PDK, and its functionality can be demonstrated utilising CMOS variance and layout aware simulations.

- Local read bitlines are used to
 - (i) clear corruption of data issues during in-SRAM processing operation and
 - (ii) facilitate to operate at a high frequency (2.2Ghz at 1V).
- This design also aids in masking the latency of in-memory addition carry transmission and instituting a fast carry adder to improve accuracy (60-70 percent improvement).

2.5.2 $Abar \cdot B$ generation using IMC

The global bitlines of proposed IMC architecture, will not be generating ($Abar \cdot B$) value. For this, we need to take help of the local bitlines, as the local bitlines contains operand value and its complemented value. So, the idea is to

perform ‘AND’ operation of local bitlines of the two operands. Suppose local bitline (Lbly0) contains the value of A and Lblby0 contains $Abar$. Similarly, local bitline (Lblx0) contains the value of B and Lblby0 contains $Bbar$.

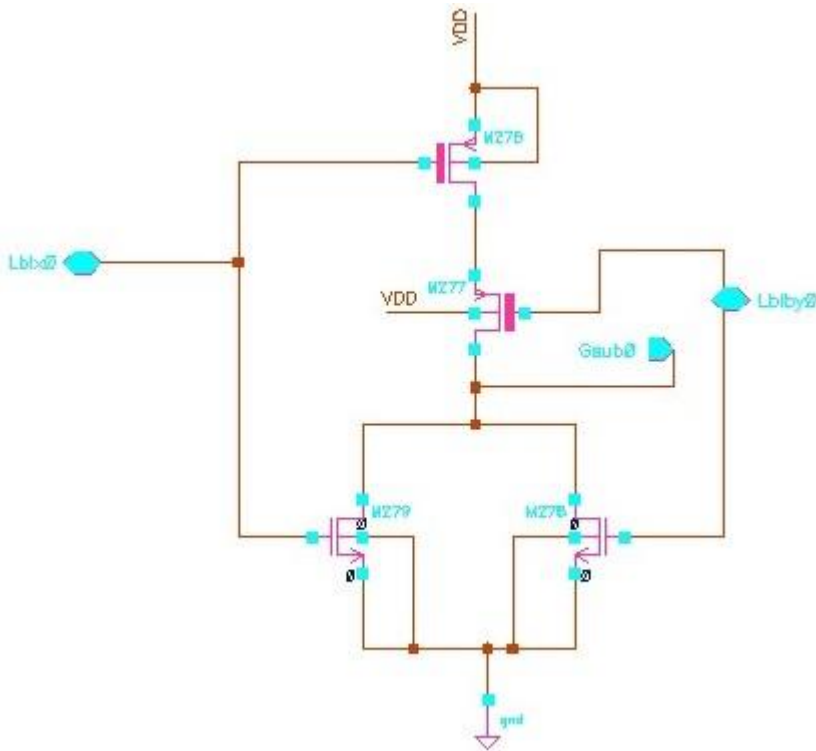


Fig 2.5 Schematic for generating $(Abar . B)$ using local bitlines

As shown in fig 2.5, ‘AND’ operation is performed for local bitlines Lblx0 (contains B) and Lblby0 (contains $Abar$) which results in generation of $(Abar . B)$. This result is very much useful when designing Add/Sub- block which will be discussed in next chapters.

2.5.3 Read Enable and Write Drivers

The precharge circuitry charges the global bitlines and local bitlines to VDD before every read operation as shown in fig 2.6. The write drivers are used to write data into the memory cell using the local bitlines. The schematic of write driver is shown in fig 2.7.

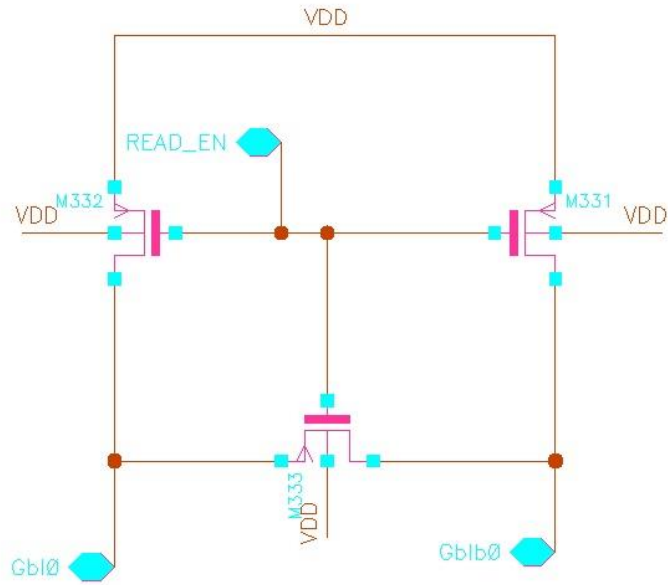


Fig 2.6 Schematic of Pre-charge circuitry for read operation

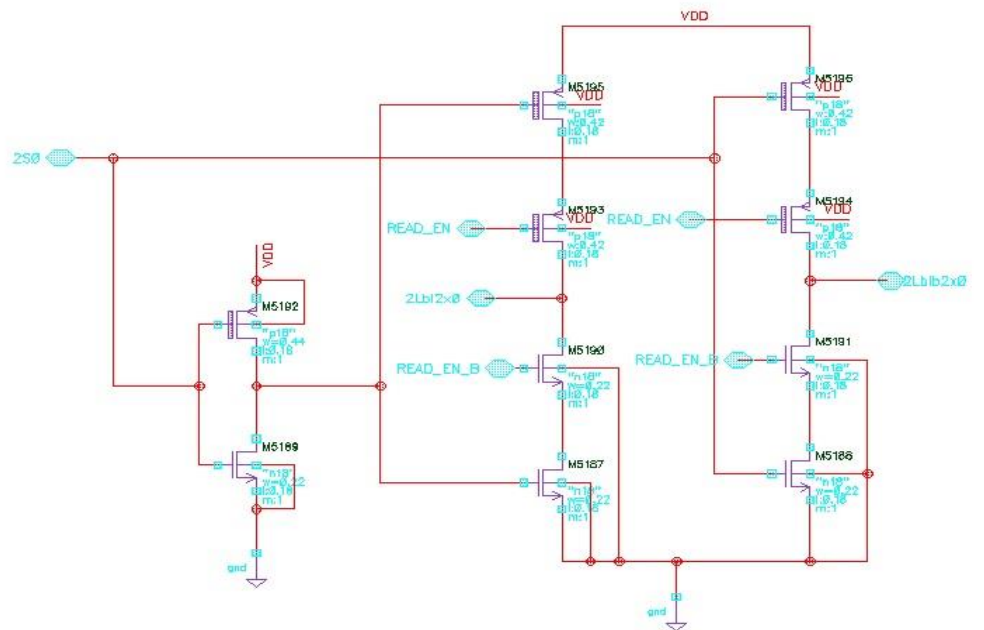


Fig 2.7 Schematic of write driver

This way the In-SRAM IMC architecture can be used for performing various logical operations.

Chapter 3

CORDIC ALGORITHM

CORDIC (Coordinate Rotation Digital Computer) is a technique for calculating the values of trigonometric functions. CORDIC is unique among computation algorithms in that it can be constructed quickly using only addition, subtraction, and bit shift operations. It isn't as fast as the riding approach, but it can save a lot of space, making it ideal for applications where space is more critical than speed.

3.1 Background

Walder first suggested the CORDIC method in 1959 in his article CORDIC Trigonometric Computing Technique. Walter demonstrated in 1971 that the algorithm's scope could be expanded to incorporate hyperbolic and linear functions in addition to derived trigonometric equations.

With three n-bit registers, three n-bit adders/subtractors, three shifters, and a small lookup table, Walter demonstrated a hardware operation. Three n-bit registers, three 1-bit adders/subtractors, and a sequence of shift gates determining which part of the register is supplied to the adder/subtractor are all part of Volder's sequential circuit. Both strategies are discussed in this study, as well as the prospect of applying this algorithm in neural networks.

3.2 Algorithm

Rotation or vectoring are the two ways in which the algorithm operates. The set of functions that the algorithm may compute is determined by both modes. The initial vector, as well as the x and y components of the rotation angle, are given in rotation mode. After the vector is rotated by the specified rotation angle, the hardware continuously computes the x and y components of the vector.

The two components are input in Vectoring mode, and the magnitude and angle of the original vector are determined. This is accomplished by rotating the input vector until the x axis is aligned. By noting the angle of rotation to obtain this alignment, you may determine the angle of the initial vector. The x component of the vector is identical to the value of the starting vector when the method is completed.

The CORDIC algorithm uses only addition, subtraction, and shift to execute calculations. This is accomplished by alternately rotating the input vector and gradually converge to the final rotation vector. This is done in a set of steps that are well-defined. The first rotation step sees the vector rotated radian $\pi/4$.

$$\begin{aligned} y_1 &= \pm X_0 = R_0 \cdot \sin(\theta_0 \pm \pi/4) \\ x_1 &= \pm Y_0 = R_0 \cdot \cos(\theta_0 \pm \pi/4) \end{aligned} \quad (3.1)$$

Where x_0 and y_0 represent the input vector aligned at the origin with magnitude R_0 and angle θ_0 :

$$\begin{aligned} x_i &= R_i \cdot \cos \theta_i \\ y_i &= R_i \cdot \sin \theta_i \end{aligned} \quad (3.2)$$

In each following step, new angle of rotation α_i is calculated such that:

$$\alpha_i = \tan^{-1}(2^{-i}), \text{ where } i > 0 \quad (3.3)$$

This constraint is necessary because it allows the rotation computations in each step to be completed with simply an add/subtract and a shift.

Figure 3.1 depicts the appearance of each step iteration. The cordic algorithm will select whether to rotate the vector by $+\alpha_i$ or $-\alpha_i$. at each iteration. The outcome of both decisions is depicted in the diagram. In the $(i + 1)^{th}$ step, the expression for the rotated vector is

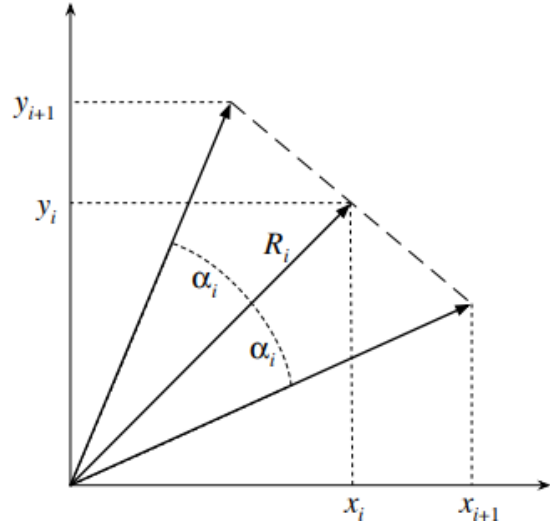


Figure 3.1: i^{th} step iteration for cordic algorithm

$$x_{i+1} = \sqrt{1 + 2^{-2i}} \cdot \cos(\theta_i \pm \alpha_i)$$

$$y_{i+1} = \sqrt{1 + 2^{-2i}} \cdot \sin(\theta_i \pm \alpha_i) \quad (3.4)$$

when applying the constraint in eq3.3 the shifting and adding becomes evident.

$$x_{i+1} = \frac{1}{K} (x_i d_i \cdot 2^{-i} \cdot y_i)$$

$$y_{i+1} = \frac{1}{K} (y_i d_i \cdot 2^{-i} \cdot x_i)$$

$$\text{where } K_i = \sqrt{1 + 2^{-2i}}, d_i = \pm 1 \quad (3.5)$$

K_i is for magnitude error term, and d_i stands for direction. The shifting action of a constant term corresponds to the 2^{-i} terms. This shifted value is added or subtracted from the component's current value. Cross addition was the term used by Volder to describe this procedure. The algorithm's ability to be used in digital hardware is due to the Cross addition.

As shown in Figure 3.1, each rotation step increases the magnitude of the input vector by a factor of $\sqrt{1 + 2^{-2i}}$. The algorithm's derivation from the supplied transform, which rotates a vector by a specific angle, introduces this inaccuracy.

$$\begin{aligned}x' &= x \cdot \cos \varphi - y \cdot \sin \varphi \\y' &= y \cdot \cos \varphi + x \cdot \sin \varphi\end{aligned}\tag{3.6}$$

These terms can be arranged using the basic trigonometric identity $\tan \alpha = \frac{\sin \alpha}{\cos \alpha}$.

$$\begin{aligned}x' &= \cos \alpha (x - y \cdot \tan \alpha) \\y' &= \cos \alpha (y + x \cdot \tan \alpha)\end{aligned}\tag{3.7}$$

We get the same relationship in eq3.3 as we did in eq3.5 by applying the same limitation. The error term arises from the presence of a cosine term in eq3.7, which is independent of rotation direction because cosine is symmetric around the y-axis. Furthermore, because this error increases with each step, the overall error in the algorithm is independent of the input angle if the number of iterations is set.

if the first rotation is given by,

$$\begin{aligned}x_1 &= \sqrt{1 + 2^{-2(1)}} R_0 \cos (\theta_0 + d_1 \alpha_1) \\y_1 &= \sqrt{1 + 2^{-2(1)}} R_0 \sin (\theta_0 + d_1 \alpha_1)\end{aligned}\tag{3.8}$$

then the second rotation is given by

$$\begin{aligned}x_1 &= \sqrt{1 + 2^{-2(1)}} \cdot \sqrt{1 + 2^{-2(2)}} \cdot R_0 \cdot \cos (\theta + d_0 \alpha_0 + d_1 \alpha_1) \\y_1 &= \sqrt{1 + 2^{-2(1)}} \cdot \sqrt{1 + 2^{-2(2)}} \cdot R_0 \cdot \sin (\theta + d_0 \alpha_0 + d_1 \alpha_1)\end{aligned}\tag{3.9}$$

n-th rotation extended and definition arrived

$$\begin{aligned}
 x_{n+1} &= \left[\sqrt{1 + 2^{-2(1)}} \cdot \sqrt{1 + 2^{-2(2)}} \dots \sqrt{1 + 2^{-2(1)}} \right] \cdot R_0 \cdot \cos (\theta + d_0 \alpha_0 \\
 &\quad + d_1 \alpha_1 + \dots + d_n \alpha_n) \\
 y_{n+1} &= \left[\sqrt{1 + 2^{-2(1)}} \cdot \sqrt{1 + 2^{-2(2)}} \dots \sqrt{1 + 2^{-2(1)}} \right] \cdot R_0 \cdot \sin (\theta + d_0 \alpha_0 \\
 &\quad + d_1 \alpha_1 + \dots + d_n \alpha_n)
 \end{aligned}$$

3.3 Accumulator Registers

Three accumulator registers are required by the CORDIC Design: the X register, the Y register, and the angle accumulator Z. As it rotates, the X and Y registers contain the current x and y component vectors. The overall rotation amount performed at the current iteration is stored in the angle accumulator.

In eq3.10. $Z_n = d_0 \alpha_0 + d_1 \alpha_1 + \dots + d_n \alpha_n$, the angle accumulator contains the arguments to the sine and cosine terms. The Z register must always include the expression because this term is comparable to the desired rotation angle-constrained to $\alpha_i = \tan^{-1}(2^{-i})$.

A small lookup table can be used to store the arctangent terms. Since $d = \pm 1$, all that is required to compute the next value in the Z register is an addition or a subtraction. This table is relatively short, as each iteration of the algorithm only requires one row. Because the number of iterations is fixed in hardware, the table's size remains constant.

The direction of rotation is also determined by the accumulator register. In rotation mode, the direction is determined by the sign of the Z register. The Y register regulates the direction in Vectoring mode.

3.4 Computation Modes

There are two modes of operation for the Cordic algorithm: rotation and vectoring. Which set of functions can be computed and how the values in the X, Y, and Z registers change each iteration are determined by the mode of operation.

3.4.1 Rotation Mode

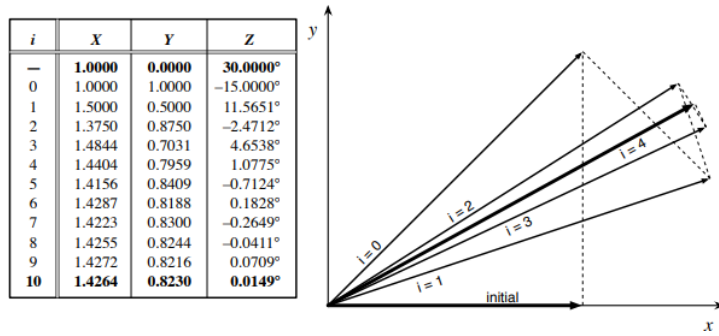


Figure 3.2: The CORDIC Rotation mode

The input vector is rotated over a defined angle in Rotation mode. The initial value of the X and Y registers is used as the input vector. The amount of rotation is entered into the Z register. The purpose of each iteration should be to reduce the value in the angle accumulator to 0 in order to rotate the input vector over the input angle. The only way to control the value in the Z register is through the d values, because the arctangent values for each iteration are fixed. The decision values in Rotation Mode are set to

$$d_i = +1 \text{ if } Z_i \geq 0 \text{ else } d_i = -1 \quad (3.12)$$

with this definition of the decision function, after n iterations of the algorithms, it is known what the values in each of the registers will be:

$$X_n = K_n [X_0 \cos(Z_0) - Y_0 \sin(Z_0)]$$

$$Y_n = K_n [Y_0 \cos(Z_0) + X_0 \sin(Z_0)]$$

$$Z_n = 0$$

$$K_n = \prod_n \sqrt{1 + 2^{-2n}} \quad (3.13)$$

In Figure 3.2 shows what happens to input vector each iteration in the algorithm. In this example, the vector is initially aligned with the x-axis. The magnitude of the vector is initially aligned with the x-axis. The magnitude of the vector increases with each step, as the angle of the vector converges on 30°. The values of the X,Y, and Z registers are also shown. In the following sections, descriptions of how various functions can be computed using Rotation mode will be discussed.

Sine, Cosine and Polar-Cartesian Transformation

The sine and cosine functions are integral to the Rotation mode, and can be easily deduced from eq3.13. All that is required is to set the Y register to 0 and the X register to the scaling factor needed. If the magnitude of the input vector had no gain, the X register could be set to 1, and sine and cosine values could be read directly from the Y and X registers, respectively. The gain, on the other hand, necessitates some scaling prior to computation. The contents of register will be determined after n iterations of the method.

$$\begin{aligned} X_n &= K_n X_0 \cos (Z_0) \\ Y_n &= K_n Y_0 \sin (Z_0) \\ Z_n &= 0 \end{aligned} \quad (3.14)$$

To account for the gain, the Z register will be initialized with θ , the Y register with 0, and the X register with K_n in order to determine the sine or cosine of an angle. The CORDIC procedure for computing sine and cosine is shown in Figure 3.1. As necessary, the initial vector is aligned. The scale cosine value is represented by the X register, whereas the scale sine value is represented by

the Y register. By dividing the final numbers by $K_{10} = 1.6468$, we can get the unscaled value. The method, for example, produces after 11 iterations.

For Polar-to-Cartesian coordinate transformation, the approach to compute sine and cosine in the same way. since the transformation is described as

$$x = r \cdot \cos \theta$$

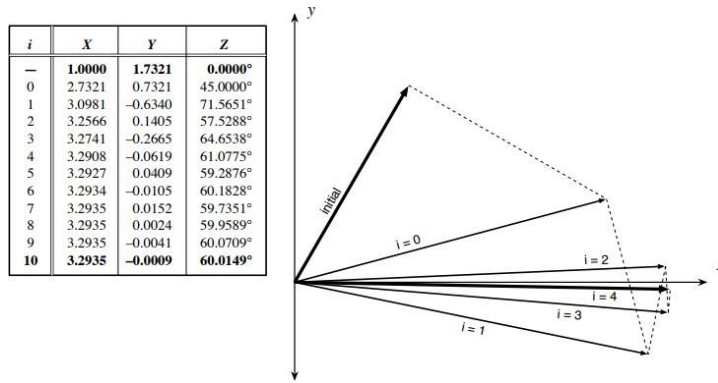


Figure 3.3: Cordic Vectoring Mode

$$y = r \cdot \sin \vartheta$$

All that is necessary to perform the transformation is to once again load ϑ in to Z and load X with rK_n , and Y with 0.

3.4.2 Vectoring Mode

The equations used to update the registers stay the same in Vectoring mode, but the function used to determine the rotation direction changes. The algorithm in Vectoring mode aims to align the input vector with the x axis, which means the Y register values should converge on zero. In rotating mode, the decision function is

$$d_i = +1 \text{ if } Z_i \geq 0 \text{ else } d_i = -1 \quad (3.16)$$

The input vector converges on the x-axis with each iteration of the algorithm, as shown in Figure 3.3. The magnitude of the vector rises with each iteration of the algorithm, just as it does in rotation mode. The direction of rotation is determined by the sign of Y, with the purpose of bringing the value to 0. The angle of the input vector with respect to the x-axis is computed and determined in this example.

3.4.3 Arctangent

When in Vectoring mode, the arctangent function is computed intrinsically in the Z register, as shown in eq3.17. Angle must be stated as a ratio of the two values in the X and Y registers in order to compute the arc-tangent of an angle. As seen in Figure 3.3, it is feasible to initialise X with 1.0 and Y with α . The result of $\arctan(\sqrt{3})$ is calculated accurately as 60.

$$\begin{aligned} Z_n &= Z_0 + \tan^{-1}\left(\frac{Y_0}{X_0}\right) \\ Z_n &= 0 + \tan^{-1}(\sqrt{3}) \\ Z_n &= 60^\circ \end{aligned} \quad (3.18)$$

3.4.4 Vector Magnitude and Cartesian-Polar Transformation

As previously stated, the scaled magnitude of the input vector is contained in the final value in the X register. This fact, together with the intrinsic computation of arctangent at the same time, means that the CORDIC vectoring mode performs a Cartesian to Polar coordinate transformation automatically, exactly as the Rotation mode does.

3.5 Expanding The Computation Domain

The CORDIC algorithm as presented thus far can only compute functions

based on the sine and cosine functions. This is a Consequence of the circular rotations performed in each step. The algorithm is capable of performing linear and hyperbolic rotations as well, which expands the set of functions that the algorithm can compute. To allow for these new domains, a new factor is introduced to the set of cordic equations. Its value determines in which coordinate system the algorithm will operate. This factor is defined as

$$m \in \{-1, 0, 1\} \quad (3.20)$$

The hyperbolic domain has a m value of -1, the linear domain has a m value of 0, and the circular domain has a m value of +1. When this factor is used to eq3.5, the cordic algorithm has the following general-purpose definition (assuming n iterations):

$$X_n = K_n [X_0 \cos(\alpha_n \sqrt{m}) + Y_0 \sqrt{m} \sin(\alpha_n \sqrt{m})]$$

$$Y_n = K_n [X_0 \sin(\alpha_n \sqrt{m}) - Y_0 \sqrt{m} \cos(\alpha_n \sqrt{m})]$$

$$Z_n = Z_0 + (\alpha_n)$$

(3.21)

As indicated in section 3.2, is the basic rotation executed at each stage, and n is the total number of rotations performed. This rotation is defined in such a way that the operations for each step in the algorithm are reduced to shifts and additions:

$$\alpha_i = \begin{cases} \tan^{-1}(2^{-i}), & m = +1 \\ 2^{-i}, & m = 0 \\ \tanh^{-1}(2^{-i}), & m = -1 \end{cases}$$

When these requirements are met, the algorithm becomes:

$$x_{i+1} = \frac{1}{K_i} (x_i - m \cdot d_i \cdot 2^{-i} \cdot y_i)$$

$$y_{i+1} = \frac{1}{K_i} (y_i - m \cdot d_i \cdot 2^{-i} \cdot x_i)$$

$$z_{i+1} = z_i - d_i \cdot \alpha_i$$

$$\text{Where } K_i = \sqrt{1 + m \cdot 2^{-2i}}, d_i = \pm 1, m \in \{-1, 0, 1\}$$

(3.23)

The CORDIC growth factors' values are shown in 3.1. The numbers have been rounded to the nearest nine decimal places. The number of iterations till the displayed value remains constant is shown in the constant after column. The displayed number can be used for any implementation with 15 or more iterations, with the exception of the linear domain, which always has a factor of 1.

The x and y components are kept in X and Y registers, as before. When working in the hyperbolic domain, the hyperbolic tangent is employed instead of the standard tan function.

Domain	Computation Mode	
	Rotation	Vectoring
Circular ($m = 1$)	$X = K_n X_0 \cos(Z_0)$ $Y = K_n X_0 \sin(Z_0)$ $\tan Z_0 = \frac{Y}{X}$	$X = K_n \sqrt{X_0^2 + Y_0^2}$ $Z = Z_0 + \tan^{-1}\left(\frac{Y_0}{X_0}\right)$
Linear ($m = 0$)	$Y = Y_0 + X_0 Z_0$	$Z = Z_0 + \frac{Y_0}{X_0}$
Hyperbolic ($m = -1$)	$X = K_n X_0 \cosh(Z_0)$ $Y = K_n X_0 \sinh(Z_0)$ $\tanh Z_0 = \frac{Y}{X}$ $e^{Z_0} = X + Y$	$X = K_n \sqrt{X_0^2 - Y_0^2}$ $Z = Z_0 + \tanh^{-1}\left(\frac{Y_0}{X_0}\right)$ $\ln(\lambda) = 2Z$, with $X_0 = \lambda + 1$, $Y_0 = \lambda - 1$ $\sqrt{\lambda} = \frac{1}{K_n} X$, with $X_0 = \lambda + \frac{1}{4}$, $Y_0 = \lambda - \frac{1}{4}$

Figure 3.4: Functions that can be computed using the CORDIC algorithm

What functions can be computed, and in which mode and domain can they be computed, are summarized in the figure above.

In addition to the addition of the m domain selector, the algorithm must be modified in a few other ways when used in various modes.

The method begins in the circular domain with the registers in their initialized states and a sequence of I values of the form 0,1,2,3,4,... and so on until the desired number of iterations has been accomplished. Because I is equal to 0, no shift is performed in the first step. The sequences in the linear domain go 1,2,3,4,5,... and so on, with a shift in the first step.

Chapter 4

MODIFIED STATIC MANCHESTER CARRY ADDER/SUBTRACTOR

As mentioned in chapter 2, the In-SRAM IMC using local bitlines and global bitlines is useful in obtaining ‘AND’ and ‘NOR’ operation results at the global bitlines (*GRBL* and *GRBL_bar*). This is very much helpful as ‘AND’ operation of the two operands can be used as carry generation when designing multi-bit adder. Further, if we perform ‘NOR’ operation of the two bitlines we get ‘XOR’ operation as shown fig 2.4.

$$GRBL = A . B = G$$

$$GRBL_bar = Abar . Bbar \quad (4.1)$$

Now performing NOR operation on *GRBL* and *GRBL_bar*,

$$\begin{aligned} (GRBL + GRBL_bar)bar &= ((A . B) + (Abar . Bbar))bar \\ &= P(Carry Propagation) \end{aligned}$$

$$P = (A . B)bar . (Abar . Bbar)bar$$

$$P = (Abar . Bbar) . (A + B)$$

$$P = Abar . B + A . Bbar$$

$$P = A \wedge B \quad (4.2)$$

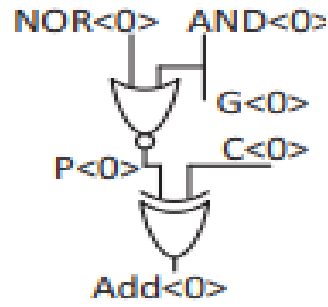


Fig4.1: Schematic for obtaining carry propagator(P) and sum output from global bitlines and carry input

Hence, we obtain ‘XOR’ operation on performing ‘NOR’ operation on the global bitlines. This can be used as carry propagator (P). With carry generator (G) and carry propagator (P), we proceed to design “Modified Static Manchester Carry Adder/Subtractor” block. This circuit increases the speed of adder/subtractor operation by obtaining the following stages carry input with the help of carry generator (G) and carry propagator (P).

Once the propagator (P) and carry input (C_i) are available, sum output can be obtained by performing XOR operation of P and C_i .

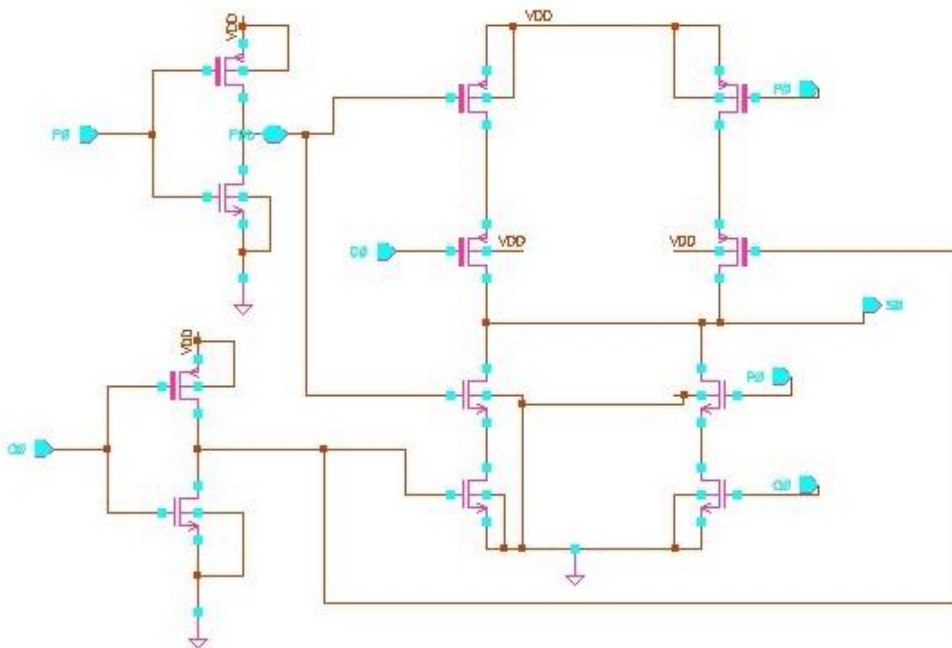


Fig 4.2: Schematic for obtaining sum output by performing XOR operation of P and C_i

4.1 Adders and Carry-Skip Adders

The adder is the most critical component in an ALU and for creating a MAC using the cordic algorithm. For conventional number systems, there are many different types of adders; we'll go over some of them according to Blaauw's classification. The synonym and asymptotic time complexity of the adders are also given:

- 1) simple carry-ripple adder : $O(n)$,
- 2) carry-predict (look-ahead) adder : $O(\log n)$,
- 3) carry-skip (bypass) adder : $O(n^{\frac{1}{l}})$, where " l " represents the number of skip layers and
- 4) carry-select (conditional sum) adder : $O(\log n)$

Carry-predict adders increase the performance of the simple carry-ripple adder by making the slow signals come earlier. The carry skip adders increase the performance of the standard carry-ripple adder by increasing the availability of early signals while trading time for resources. To reduce the number of levels in the carry-select adder, the early signals are replicated at the expense of additional resources. In a Manchester adder block, the carry-skip delay is proportional to block size.

4.2 Dynamic Manchester Carry Chain Adder

A chain of pass-transistors is utilised to implement the carry chain in the Manchester Carry-Chain Adder. The Manchester carry chain, as shown in fig. 4.3, is developed using dynamic logic and implement the essential logical function:

$$C_{i+1} = G_i + P_i \cdot C_i$$

where C_i is the carry out.

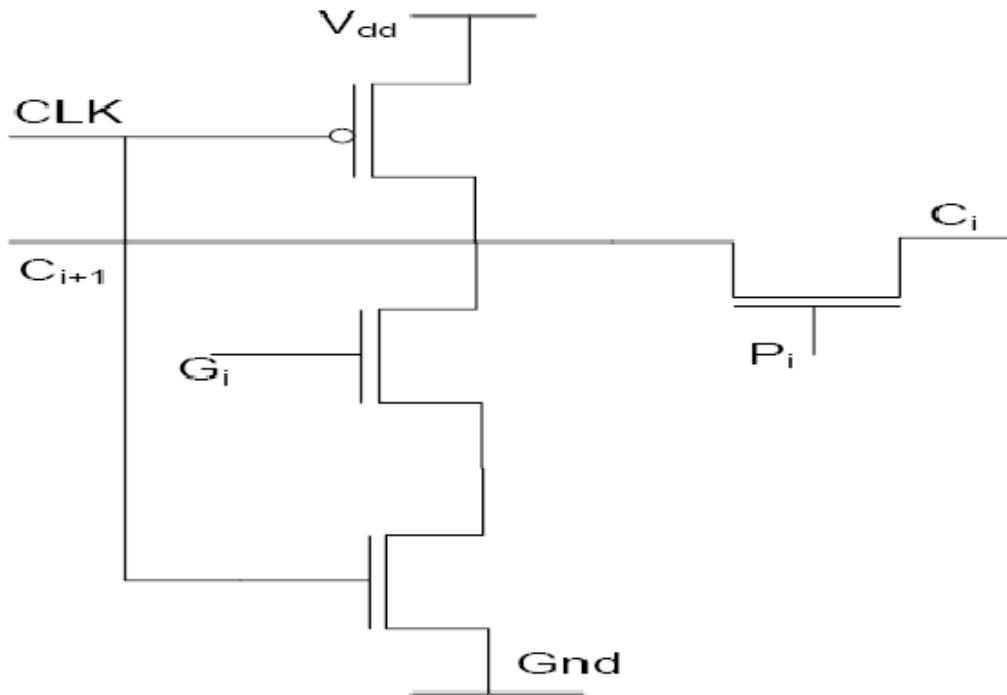


Fig 4.3: Schematic of Dynamic Manchester carry adder

All intermediate nodes (such as C_{i+1}) are charged to Vdd during precharge. Cout k is discharged during the evaluation phase if Cin0 is an incoming carry and the prior propagate signal (P_i) is high.

Because the dynamic Manchester Carry Adder is built using domino logic, if P_i and G_i are both '0' during the evaluation phase, the intermediate node will be floating. As a result, the value at the node discharges continually, causing value at the node to be distorted. This distortion causes glitches by affecting the value at other intermediary nodes. As a result of these glitches, we proposed a modified Static Manchester Carry Adder.

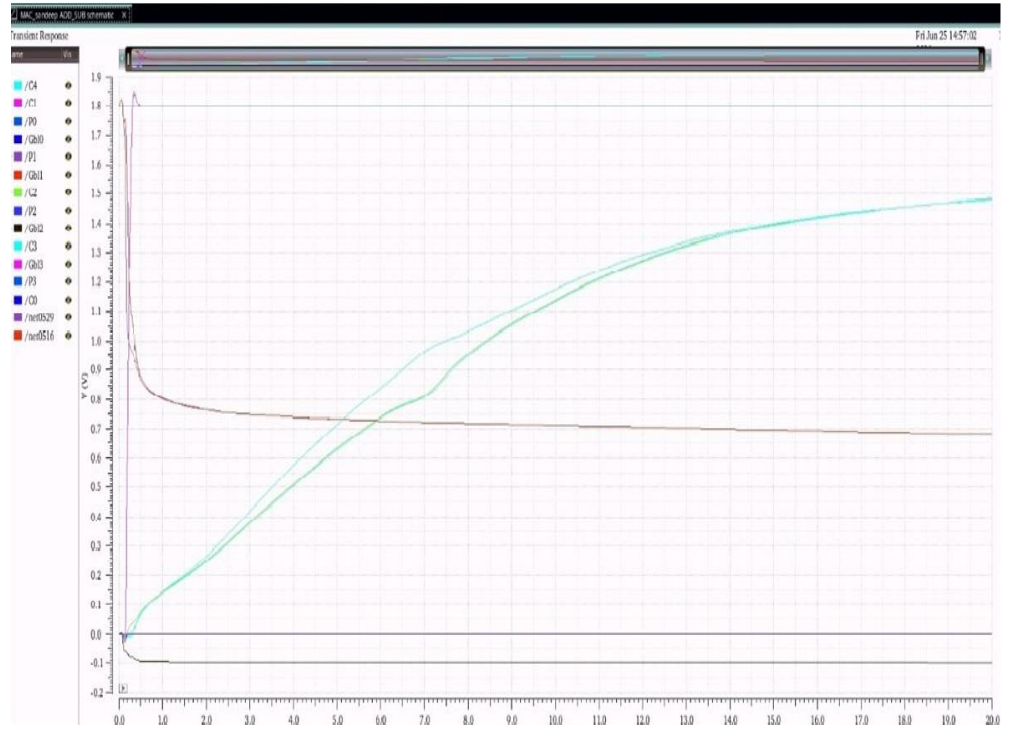


Fig 4.4: Glitches due to floating intermediate node

4.3 Modified Static Manchester Carry Chain Adder

Modified Static Manchester Carry Adder can overcome the problems of floating node caused due to Dynamic Manchester Carry Adder. In modified Static Manchester Carry Adder, the pass transistors will pass the inverted value of carry when carry propagation (P_S) is high. When carry generation (G) is high, the intermediate node pulled down to '0'. On inverting the value, we can obtain the value of C_{i+1} . Table 4.1 shows the operation of Manchester Carry adder.

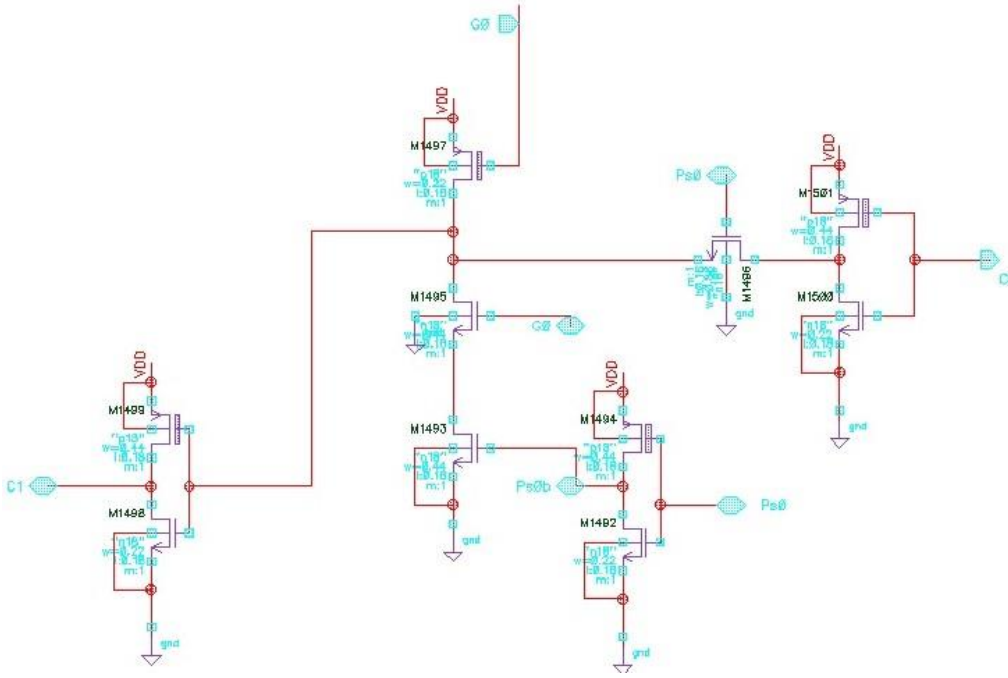


Fig 4.5: Schematic of Modified Static Manchester carry adder

4.3.1 Sizing of the transistors

The operation of static Manchester carry adder is explained in table 4.1. From table 4.1, it's clear that when $P=1$ and $G=0$, the value of C_{i+1} must be equal to C_i . When $P=1$ and $G=0$, the pass transistor(M1496) and pmos transistor(M1497) both are on. So, switched on pmos transistor tries to pull up the intermediate node and turned-on pass transistor tries to pass the inverted value of C_i as shown in fig 4.5. If $C_i=0$, both the pass transistor and pmos transistor pulls the intermediate node value to high.

If $C_i=1$, then the pmos transistor pulls the intermediate node to high and pass transistor pulls down to '0'. Whereas the intermediate node needs to obtain the value of '0'.

To achieve this, the strength of pass-transistor is increased i.e the width of pass transistor is increased and the pmos transistor is made weaker by decreasing its width.

A	B	G	P	K	C_{i+1}
0	0	0	0	0	0
0	1	0	1	0	C_i
1	0	0	1	0	C_i
1	1	1	0	0	1

Table 4.1: Operation of Manchester Carry Adder

4.4 Modified Static Manchester Carry Adder as Subtractor

Table 4.2 shows that sum and difference output is same for adder and subtractor.

The equation for borrow generation can be changed as follows,

$$B_{out} = \sim(X \wedge Y) \cdot B_{in} + Xbar \cdot Y$$

$$B_{out} = (Xbar \wedge Y) \cdot B_{in} + Xbar \cdot Y \quad (4.3)$$

Equation 4.3 is similar to carry output equation with one of the input (X) inverted.

$$C_{out} = (X \wedge Y) \cdot C_{in} + X \cdot Y \quad (4.4)$$

Components	Sum / Difference	C_{out}/B_{out}
Adder	$S = X \oplus Y \oplus C_{in}$	$C_{out} = X \oplus Y \cdot C_{in} + XY$
Subtractor	$D = X \oplus Y \oplus B_{in}$	$B_{out} = \overline{X \oplus Y} \cdot B_{in} + \overline{X} Y$

Table 4.2: Adder and subtractor outputs for X and Y operands

This logic can used to get subtraction output from modified static Manchester carry adder. This can be used to obtain Static Manchester carry subtractor.

For Manchester carry subtractor, we need borrow propagation (P_b) and borrow generation (G_b). From equation 4.3, we obtain,

$$\begin{aligned} B_{out} &= (Xbar \wedge Y) \cdot B_{in} + Xbar \cdot Y \\ B_{out} &= P_b \cdot B_{in} + G_b \end{aligned} \quad (4.5)$$

Hence, we need to obtain P_b and G_b . For obtaining G_b , the local bitlines of the proposed In-SRAM IMC architecture used as shown in fig. P_b is already obtained in the design generating sum output as shown in fig 4.2.

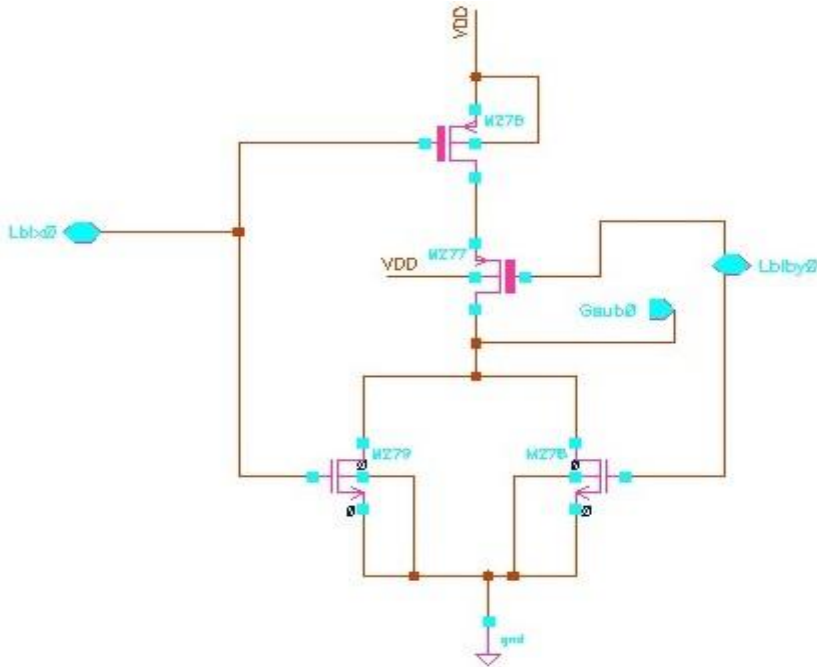


Fig 4.6: G_{sub} or G_b or $(Xbar \cdot Y)$ generation using local bitlines of proposed In-SRAM IMC architecture

This way modified Static Manchester Carry Chain Adder can be used as both adder and subtractor.

Chapter 5

MAC Design using CORDIC Algorithm

5.1 CORDIC Algorithm

CORDIC (Coordinate Rotation Digital Computer) is an iterative algorithm that calculates 2-dimensional vector rotation in various coordinate systems to calculate a variety of mathematical relationships; the beauty of CORDIC algorithm is that it only uses shift, addition/subtraction operations, and memory elements. The focus of this thesis is to compute multiplication in a linear coordinate system using the CORDIC method. Figure 5.1 depicts the generalised CORDIC hardware design. The CORDIC employs a scaled form of real rotation calculation called pseudo rotation. The pseudo rotation co-ordinate calculation equations are:

$$\begin{aligned}X_{n+1} &= X_n - Y_n \cdot \tan(\alpha_n) \\Y_{n+1} &= Y_n + X_n \cdot \tan(\alpha_n) \\Z_{n+1} &= Z_n - \alpha_n\end{aligned}\tag{5.1}$$

The pseudo rotation is computed using trigonometric calculations in the CORDIC equations provided in Eq. 5.1. As illustrated in Eq.5.2, the linear converges CORDIC equations form of the pseudo rotation coordinate equations. For CORDIC computing, three components are required: a multiplexer, shift register and an adder/subtractor.

$$\begin{aligned}X_{n+1} &= X_n - m \cdot Y_n \cdot d_n \cdot 2^{-n} \\Y_{n+1} &= Y_n + X_n \cdot d_n \cdot 2^{-n} \\Z_{n+1} &= Z_n - d_n \cdot E_n\end{aligned}\tag{5.2}$$

$$d_n \in \{-1, +1\}; i = 0, 1, 2, \dots, n-1$$

A circular, linear, and hyperbolic coordinate system are shown by the modes $m \in \{1, 0, -1\}$ respectively.

E_n is the memory elements which are different with respect to computation mode at each n^{th} iteration which is equal to 2^{-n} , $\tan^{-1}(2^{-n})$ and $\tanh^{-1}(2^{-n})$ for linear, circular and hyperbolic rotation mode respectively.

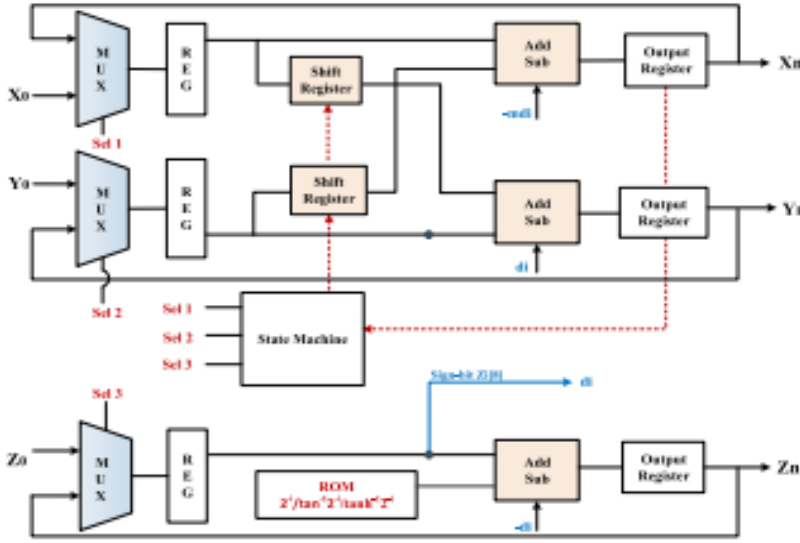


Fig 5.1 Sign N-bit precision recursive CORDIC architecture that can realize multiply- accumulate computation in linear mode ($m = 0$ & $E_i = 2^{-i}$)

5.1.1 CORDIC configuration to MAC

At each iteration, the CORDIC will be placed in linear rotation mode with variable mode m set to zero and E_n set as 2^{-n} pre-calculated memory element. Eq. 5.3 shows the revised CORDIC equations for pseudo rotation calculations in linear rotation mode. For the i^{th} iteration, the output at Y_{i+1} is an accumulation of Y_i and the X_i shifted version. It should also be highlighted that the computation can be used to multiply and accumulate data.

$$X_{i+1} = X_i$$

$$Y_{i+1} = Y_i + X_i \cdot d_i \cdot 2^{-i}$$

$$Z_{i+1} = Z_i - d_i \cdot 2^{-i} \quad (5.3)$$

$$d_i \in \{-1, +1\}; i = 0, 1, 2, \dots, n-1$$

The convolutional neural network's main processing block is multiplication and accumulation. In addition, comparable operations can be computed using the CORDIC method in linear rotation mode, which uses shift registers and adders instead of multipliers.

The computation insights multiply-accumulate operation at Y_{i+1} is valid when the $Z_0 \rightarrow 0$. The input, bias value, and matching weight are represented by X_{in} , Y_{in} , Z_{in} , respectively. Both X_{in} (input) and Z_{in} (weight) are assumed to be fractions with an 8-bit fixed-point representation in this calculation. In order to compute with 8-bit accuracy, we must iterate a set of equations until $Z=0$ or a maximum of 8 iterations. The barrel shifter can be used to achieve the i -bit right shift input X_0 during the i^{th} iteration. With the exception of bit-shift and direction, this implementation technique is equivalent to normal shift and add multiplication. The CORDIC algorithm for multiplication of X_0 and Z_0 is shown in Eq. 5.4, which is derived from Eq. 5.3. Input (X_0), bias value (Y_0), and matching weight (Z_0) are represented by X_{in} , Y_{in} , and Z_{in} as illustrated in fig 5.1.

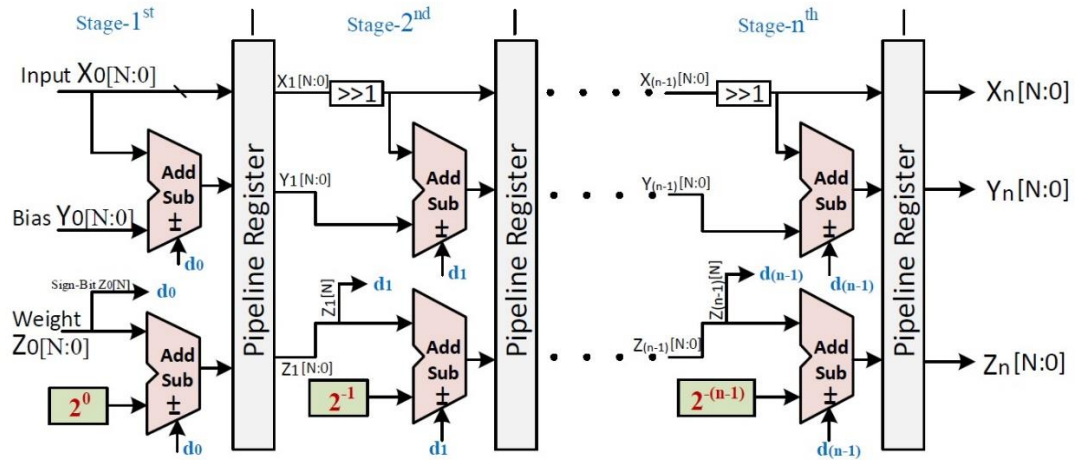


Fig 5.2: n-stage pipelined MAC design using CORDIC algorithm

Here, $X_{out} = X_{in}$ & when, $Z_{out} \rightarrow 0$

$$Y_{out} = Y_{in} + X_{in} * Z_{in} \quad (5.4)$$

The multiplication at Y_{out} i.e ($X_{in} * Z_{in}$) can be achieve through addition of shifted version of X_0

with input X_0 as explained below using Eq.5.5.

$$x_j = x_{jN} * w_N = x_{jN} * \sum_{i=1}^j a_i * 2^{-i}$$

$$x_j = \sum_{i=1}^j a_i * x_{jN} * 2^{-i} \quad (5.5)$$

x_j is made up of a shifted form of input x_0 with regard to weight z_0 , according to Eq. 5.5. The typical right shift and add based multiplication is used in this approach. By pushing a_i to zero one bit at a time, the unknown coefficient a_i can be discovered. In CORDIC architecture, if the i_{th} bit of input N is non-zero, x_i is first right-shifted by ‘i’ bits and then added to the current value of y_i . When x_j (Y_n in Eq. 5.3) is set to zero, all bits are checked, and x_j includes the signed product of input feature x_0 and weight z_0 . Due to its iterative structure, a single MAC operation requires ‘i’ iterations to compute the final desired output, which is the only reason for its low speed.

5.2 Pipelining to increase performance of MAC

Different hardware architectures for 2D rotational CORDIC in linear rotation have been devised to make hardware more efficient and boost throughput. Furthermore, the CORDIC iteration is mutually exclusive, implying that they are independent of one another. As a result, pipelining for high throughput applications at the expense of area overhead is desirable. Figure 5.2 depicts a modified n-stage pipeline CORDIC architecture that improves performance. After the initial (n-1) clock cycles, it generates the desired output at each clock.

For the intended output, CORDIC-based work input is n-bit right shifting at the nth stage. It necessitated the use of a barrel shifter, which increased the amount of space available overhead. We employed single bit shift at each level in our project, which eliminated the requirement for a barrel shifter. Furthermore, each level has only two adder/subtractor logic blocks, as shown in Figure 5.2.

As demonstrated in Figure 5.3, we improved the performance of the MAC by using pipeline CORDIC stages for multiplication. The suggested multiplier uses dynamic fixed-point encoding and has the same input and output precision. We utilise the same input and output format, therefore $i_{b\ in}=i_{b\ out}=w$, where w is the number of integer bits omitting the sign bit. However, in the case of fixed-point formats, the final decision must be made depending on the target application that requires the most accuracy and precision.

The output of the multiplier is added together using an adder with additional overhead bits, i.e. $\log_2 N$.

5.2.1 Designing of Adder/Subtractor block using IMC

As illustrated in Fig. 5.2, the MAC operation is carried out using the CORDIC method. The mode variable, m , is set to 0 in linear mode, while E_i is set to 2^{-i} . The overall output is then translated using equation 5.2, as seen below.

$$\begin{aligned} Y_{i+1} &= Y_i + X_i \cdot d_i \cdot 2^{-i} \\ Z_{i+1} &= Z_i - d_i \cdot 2^{-i} \end{aligned} \quad (5.2)$$

The direction signal d_i is determined by Z_i 's sign bit. The d_i direction determines the capability of the add/sub-block utilised in CORDIC architecture. The direction signal is significant since it aids in iterative convergence of the algorithm. The rotation direction for the i^{th} iteration is d_i , and the output at Z converges to 0.

Add/sub-block can be designed using IMC as mentioned in chapter 2 and 4. Based on the value of d_i , the IN-SRAM IMC using local bitlines and global bitlines (proposed design) and Modified Static Manchester Carry Adder operates as an add/sub block.

As seen in chapter 4, the static Manchester carry adder can operate as subtractor when one of the input is inverted and then carry generator(G) and carry propagator(P) values are obtained.

Hence, we need to obtain P_b (borrow propagator) and G_b (borrow generator). For obtaining G_b , the local bitlines of the proposed In-SRAM IMC architecture used. P_b is already obtained in the design while generating sum output as shown in fig 4.2.

For this operation, a mux is designed to operate in addition/subtraction mode with d_i (sign bit of Z_i) as select bit as shown in fig 5.3.

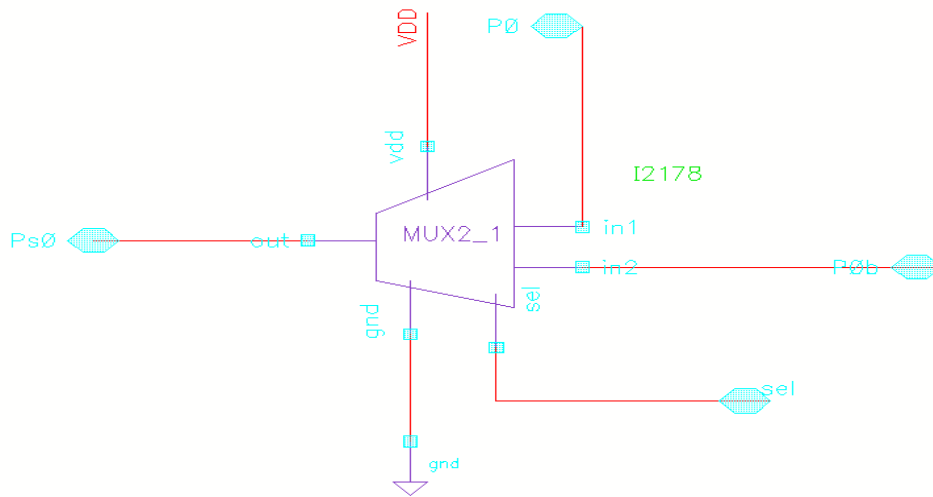


Fig 5.3: MUX for selecting P_0 (adder) or P_{0b} (subtractor) with select pin

5.3 Number of pipelining stages

To compute the N bit data for the iterative CORDIC, we must complete N clock cycles to obtain the output data. For example, if we use cordic to do an 8-bit multiplication operation, we must wait for 8 clock cycles to get the right result. This is the problem with iterative CORDIC MAC, but we can get around it by using pipelining. However, we should not use 8 step pipelining to boost throughput because this is not the best method. As a result, we ran an experiment to determine the number of clock cycles required to provide accurate output values. The experiment revealed that roughly 4 or 5 clock cycles are required to obtain the output value, with precision saturating after 4 or 5 values. We get the same accuracy with 5 clock cycles instead of 8 clock cycles. Table 5.1 shows that after 5 clock cycles, the desired result is obtained.

Table 5.1: For high-performance MAC operation, iteration-level calculation is given for MAC computation using CORDIC in linear mode for fixed (8, 7) representation Processing at each nth stage in pipeline architecture.

n	d_n	X_{n+1} → X₀	Y_{n+1} → Y_i + d_i X_i * 2⁻ⁱ	Z_{n+1} → 0
Pipelined stage	$(n + 1)^{th}$	Initial Conditions/Inputs		
	iteration	Input = 0.65625 ₁₀	bias = 0.00 ₁₀	weight = 1.296875 ₁₀
initial	–	0.1010100	0.0000000	1.0100110
0	+1	0.1010100	0.1010100	0.0100110
1	+1	0.0101010	0.1111110	-0.0011010
2	-1	0.0010101	0.1101001	0.0000110
3	+1	0.0001010	0.1110011	-0.0001010
4	-1	0.0000101	0.1101110 ₂ (0.85937 ₁₀)	-0.0000010 (≈ 0)

5.4 Generation of MAC symbol

A symbol for MAC is created containing input, bias, weights, and read enable as input to the MAC and bias output at the end of 3rd stage gives the MAC output which is represented in the MAC symbol and weight output of 3rd stage of pipeline is also obtained which will be tending to zero.

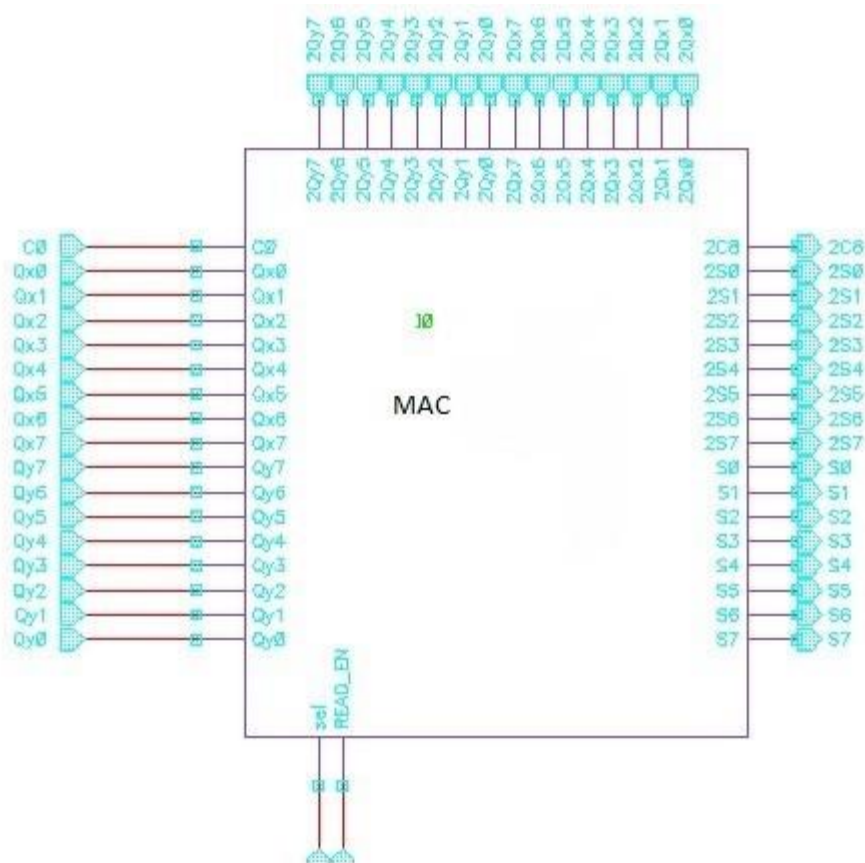


Fig 5.4 : MAC symbol generated for 3 stage pipelined CORDIC based MAC design using In-Memory Computing and Static Manchester Carry Adder

In the next chapter, we will be discussing the results of proposed MAC design.

Chapter 6

Results and Discussions

We covered the difficulties of a hardware implementation of CNN in FPGA in the previous chapters. The use of resources and the precision of data bits are the two most pressing concerns. The two difficulties are addressed by our proposed architecture. For example, more bit precision is required for better accuracy, but with the suggested architecture, we can achieve acceptable accuracy with lower bit precision.

In this chapter, we discuss the validation of the proposed architecture and how it is better compared to other architecture.

We investigated the inference accuracy on LeNet using MNIST and CIFAR-10 datasets to validate the compatibility of suggested architecture in a CNN model. Both fixed point and multi-bit precision (i.e. 2,4,8,16,32-bit) precision are observed. We can see that the accuracy loss for 8-bit and 32-bit precision is quite small (less than 2%). In comparison to 32-bit precision, 8-bit calculation reduced 4 computational costs. Furthermore, the proposed high-performance MAC is built with a pipelined CORDIC architecture and a modified LeNet architecture. Table 6.1 compares the inference accuracy of the proposed and tensor library-based computations.

We've fixed the precision at 'fixed 8, 7' for further discussion. For physical parameters, the VLSI design should be efficient, and computation approximation is one option. Furthermore, deep neural network algorithm learning is error-resistant. According to the results of a Pareto analysis of pipeline stages, a five-stage pipeline architecture is efficient for MAC calculation and provides the requisite final accuracy. Because it is responsible for chip-area and on-chip power consumption, the best pipeline stages have been used. In addition, as previously mentioned, the computation of five pipeline stages involves approximation. Different error metric formulae are used to check the output calculation. Table 6.2 shows the error metric equations

for mean square error, mean absolute error, average error, and standard deviation.

Table 6.1: Network Inference Accuracy

Bit-Precision		LeNet Inference Accuracy (%)		
Dynamic Fixed-Point	MNIST		CIFAR-10	
	Tensor	CoMAC	Tensor	CoMAC
32-bit	99.1	97.8	81.7	77.1
16-bit	98.7	97.1	81.2	76.8
8-bit	98.2	96.1	80.7	76.1
4-bit	97.6	94.2	79.6	75.1
2-bit	85.9	80.1	48.0	44.2

6.1 Delay Estimations

The proposed 3-stage pipelined CORDIC based MAC architecture increases the speed of operation of MAC because of pipelining and adder/subtractor design using IMC and modified Static Manchester Carry Adder. In case of IMC, the necessary computations take place while reading data from the memory itself. Further, when these computations are given to the modified Static Manchester Carry Adder, the delay is further reduced. Pipelining, results in parallel computing of data. Result of these, significant improved in the delay is observed compared to state of the art. Table 6.3 justifies significant improvement in delay.

Table 6.3: Delay comparison for the proposed design and the state-of-the-art for MAC computation @45nm TT process corner for 8-bit precision.

	Delay(ns)
Wallace Tree	6.54
Shift-add	4.27
Vedic multiplier	7.66
Proposed CORDIC MAC using IMC	1.59

6.2 Power Estimation

Table 6.4: Variation of Dynamic power with voltage TT process corner for 8-bit precision

Voltage	Dynamic Power(μ W)
1.8	385
1.7	320.8
1	144.4

Table 7.4 shows the variation of dynamic power for Typical-Typical (TT) process corner for 8-bit precision with different voltages. As the voltage comes down, we can observe significant decrease in the dynamic power of MAC. In order to reduce the dynamic power, it's best suited to operate MAC at 1V. When operated at 1V, both the dynamic power and delay are significantly reduced.

6.3 Functional Verification

The MAC is also functionally verified for various inputs, bias and weights. It is verified on 3-stage pipelined architecture. The following tables shows the results at each stage for various input, bias and weight. The tables also shows that Z_{n+1} converging to zero at the end of 3rd stage of pipelined architecture. With MAC output obtaining at Y_{n+1} , at the end of 3rd stage of pipelined architecture.

Table 6.5: Computation of MAC using CORDIC and IMC for fixed point representation

n	d_n	$X_{n+1} \rightarrow X_0$	$Y_{n+1} \rightarrow Y_i + d_i X_i * 2^{-i}$	$Z_{n+1} \rightarrow 0$
Pipelined stage	$(n+1)^{th}$ iteration	Initial Conditions/Inputs		
		Input = 0.65625 ₁₀	bias = 0.00 ₁₀	weight = 1.296875 ₁₀
initial	–	0.1010100	0.0000000	1.0100110
0	+1	0.1010100	0.1010100	0.0100110
1	+1	0.0101010	0.1111110	-0.0011010
2	-1	0.0010101	0.1101001	0.0000110

Table 6.6: Computation of MAC using CORDIC and IMC for fixed point representation for different input, bias and weight

n (Pipelined stage)	d_n $(n+1)^{th}$ iteration	$X_{n+1} \rightarrow X_0$ Input	$Y_{n+1} \rightarrow Y_n + X_n \cdot d_i \cdot 2^{-i}$ Bias	$Z_{n+1} \rightarrow 0$ Weight
Initial	-	0.1011000	0.0011000	0.1110100
0	+1	0.0101100	0.1110000	1.1110100
1	-1	0.0010110	0.1000100	0.0110100
2	+1	0.0001011	0.1011000	0.0010100

6.4 Conclusion

The CNN accelerator design is implemented using GPUs, FPGAs, and ASICs. GPUs have the advantage of design freedom, but they are inefficient in terms of energy consumption. The number of hardware resources available to FPGAs, such as MAC units and on-chip memory, is restricted. Convolution was difficult due to a lack of MAC units on the FPGA. ASICs use less power and take up less area. As a result, it's excellent for MAC design. Higher bit precision computation (32 bit or 64 bit) is likewise more expensive in terms of area and power. As a result, a decrease in hardware complexity (i.e., bit precision) and a faster response time without sacrificing accuracy are highly desirable. To overcome the above limitations, we propose a Pipelined CORDIC based MAC using In-Memory Computing and modified Static Manchester Carry adder. This proposed design significantly increases the throughput by 40% when compared with “Shift and Add”. Choosing 8-bit precision improves the power consumption and area utilization. When operated at 1 V, the power consumption decreases by 38%. Hence, the proposed design offers the best throughput among the state of art as shown in table 7.3 and consumes lesser power and area when designed at 8-bit precision and operated at 1 V.

Bibliography

- [1] Naveen Suda et al. "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks". In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2016, pp. 16–25.
- [2] W. Simon, J. Galicia, A. Levisse, M. Zapater and D. Atienza, "A Fast, Reliable and Wide-Voltage-Range In-Memory Computing Architecture," 2019 56th ACM/IEEE Design Automation Conference (DAC), 2019, pp. 1-6.
- [3] G. Raut, S. Rai, S. K. Vishvakarma and A. Kumar, "RECON: Resource-Efficient CORDIC-Based Neuron Architecture," in IEEE Open Journal of Circuits and Systems, vol. 2, pp. 170-181, 2021, doi: 10.1109/OJCAS.2020.3042743.
- [4] S. Yin, Z. Jiang, J. Seo and M. Seok, "XNOR-SRAM: In-Memory Computing SRAM Macro for Binary/Ternary Deep Neural Networks," in IEEE Journal of Solid-State Circuits, vol. 55, no. 6, pp. 1733-1743, June 2020, doi: 10.1109/JSSC.2019.2963616.
- [5] H. Chen, J. Li, C. Hsu and C. Sun, "Configurable 8T SRAM for Enabling in-Memory Computing," 2019 2nd International Conference on Communication Engineering and Technology (ICCET), 2019, pp. 139-142, doi: 10.1109/ICCET.2019.8726871.
- [6] P. K. Chan and M. D. F. Schlag, "Analysis and design of CMOS Manchester adders with variable carry-skip," in IEEE Transactions on Computers, vol. 39, no. 8, pp. 983-992, Aug. 1990, doi: 10.1109/12.57038.
- [7] A. Agrawal, A. Jaiswal, C. Lee and K. Roy, "X-SRAM: Enabling In-Memory Boolean Computations in CMOS Static Random-Access Memories," in IEEE Transactions on Circuits and Systems I: Regular

- Papers, vol. 65, no. 12, pp. 4219-4232, Dec. 2018, doi: 10.1109/TCSI.2018.2848999.
- [8] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. “FPGA-based accelerators of deeplearning networks for learning and classification: A review”. In: IEEE Access 7 (2018), pp. 7823–7859.
 - [9] Nathan Silberman and Sergio Guadarrama. “TensorFlow-slim image classification library”. In: Retrieved December 14 (2013), p. 2019.
 - [10] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: nature 529.7587 (2016), pp. 484–489.
 - [11] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: arXiv preprint arXiv:1409.1556 (2014).
 - [12] Naveen Suda et al. “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks”. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2016, pp. 16–25.
 - [13] Vivienne Sze et al. “Efficient processing of deep neural networks: A tutorial and survey”. In: Proceedings of the IEEE 105.12 (2017), pp. 2295–2329.
 - [14] Vivienne Sze et al. “Efficient processing of deep neural networks: A tutorial and survey”. In: Proceedings of the IEEE 105.12 (2017), pp. 2295–2329.
 - [15] Vivienne Sze et al. “Hardware for machine learning: Challenges and opportunities”. In: 2017 IEEE Custom Integrated Circuits Conference (CICC). IEEE. 2017, pp. 1–8.
 - [16] Christian Szegedy et al. “Going deeper with convolutions”. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2015, pp. 1–9.

- [17] Li Du et al. “A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.1 (2017), pp. 198–208.
- [18] Zidong Du et al. “ShiDianNao: Shifting vision processing closer to the sensor”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015, pp. 92–104.
- [19] Hadi Esmaeilzadeh et al. “Neural acceleration for general-purpose approximate programs”. In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2012, pp. 449–460.
- [20] M EVERINGHAM, L VAN GOOL, CKI WILLIAMS, et al. *Pascal VOC data sets*.
- [21] Clément Farabet et al. “Cnp: An fpga-based processor for convolutional networks”. In: *2009 International Conference on Field Programmable Logic and Applications*. IEEE. 2009, pp. 32–37.
- [22] Rafael Gadea Gironés et al. “FPGA implementation of a pipelined on-line backpropagation”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 40.2 (2005), pp. 189–213.
- [23] Hans Graf et al. “A massively parallel digital learning processor”. In: *Advances in Neural Information Processing Systems* 21 (2008), pp. 529–536.
- [24] Denis A Gudovskiy and Luca Rigazio. “Shiftcnn: Generalized low-precision architecture for inference of convolutional neural networks”. In: *arXiv preprint arXiv:1706.02393* (2017).