Heterogeneous Chip Multiprocessor: Data Representation, Mixed-Signal Processing Tiles, and System Design

by

Kayode A. Sanni

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

January, 2019

Abstract

With the emergence of big data, the need for more computationally intensive processors that can handle the increased processing demand has risen. Conventional computing paradigms based on the Von Neumann model that separates computational and memory structures have become outdated and less efficient for this increased demand. As the speed and memory density of processors have increased significantly over the years, these models of computing, which rely on a constant stream of data between the processor and memory, see less gains due to finite bandwidth and latency. Moreover, in the presence of extreme scaling, these conventional systems, implemented in submicron integrated circuits, have become even more susceptible to process variability, static leakage current, and more. In this work, alternative paradigms, predicated on distributive processing with robust data representation and mixed-signal processing tiles, are explored for constructing more efficient and scalable computing systems in application specific integrated circuits (ASICs).

The focus of this dissertation work has been on heterogeneous chip multi-processor (CMP) design and optimization across different levels of abstraction. On the level

ABSTRACT

of data representation, a different modality of representation based on random pulse density modulation (RPDM) coding is explored for more efficient processing using stochastic computation. On the level of circuit description, mixed-signal integrated circuits that exploit charge-based computing for energy efficient fixed point arithmetic are designed. Consequently, 8 different chips that test and showcase these circuits were fabricated in submicron CMOS processes. Finally, on the architectural level of description, a compact instruction-set processor and controller that facilitates distributive computing on System-On-Chips (SoCs) is designed. In addition to this, a robust bufferless network architecture is designed with a network simulator, and I/O cells are designed for SoCs.

The culmination of this thesis work has led to the design and fabrication of a heterogeneous chip multi-processor prototype comprised of over 12,000 VVM cores, warp/dewarp processors, cache, and additional processors, which can be applied towards energy efficient large-scale data processing.

Primary Reader: Andreas G. Andreou

Secondary Reader: Ralph Etienne-Cummings

Committee Member: Philippe O. Pouliquen

Acknowledgments

This dissertation work would not have been possible without the support of many individuals. Firstly, I would like to express my deepest gratitude to my advisor, Andreas Andreou, who has been an insightful and enduring mentor to me throughout my graduate studies. He has not only challenged me to diligently approach research with an open-mind, but has also been patiently supportive of me throughout the years. I'm grateful for all the candid and unembellished talks we've had that has constructively led to many improvements in my work.

I would like to specially thank my secondary reader, Ralph Etienne-Cummings, for not only his service on my committee, but also for his indispensable support that has always been available to me throughout my studies here at Hopkins. Additionally, I am very grateful to my other committee member, Philippe Pouliquen, who has been an invaluable source of help to me throughout my dissertation work.

I am grateful to other ECE faculty members at Hopkins, who have been instrumental in my research. I thank Pedro Julian for his unwavering support, advice, and insight throughout the years. I thank Charbel Rizk, who I have had the pleasure

of collaborating with on projects that has been useful for my research. I thank Jim West, who has been supportive of me, and offered me great advice over the years. I also thank my first PI, Jeffrey Gray, a professor in the Chemical and Biomolecular Engineering Department here at Hopkins. He introduced me to interdisciplinary research during my first research internship, and was a very supportive mentor to me, even before starting graduate school.

A large portion of my dissertation work was supported by the DARPA UPSIDE project HR0011-13-C-0051 through BAE Systems. I am grateful to Mike Graziano, Isidoros Doxas, and Louise Sengupta, who managed the project the first two years while at BAE systems, for their support and encouragement. I would like to acknowledge Tomás Figliolia for his significant contribution to this project. I am appreciative to Christian Mayr from the Technische Universität Dresden, Germany for his support and inspiration in this project that was useful for my dissertation. I am also grateful for the other funding I have received over the year from the NSF grant SCH-INT 1344772, the ONR MURI N000141010278, and the NSF grant INSPIRE SMA 1248056.

I am very thankful to all my fellow colleagues that I have been fortunate to learn from and collaborate with during my tenure in the Andreou Lab including: Joseph Lin, Recep Özgün, Thomas Murray, Tomás Figliolia, Daniel Mendat, Gaspar Tognetti, Guillaume Garreau, Kate Fischl, Martìn Villemur, Christos Sapsanis, Jeff Craley, Alejandro Pasciaroni, Valerie Rennoll, and Jonah Sengupta. I thank Francisco

Tejada too for all the help he has provided me. I specially thank Tomás and Martìn for teaching me how to use the synthesis and place and route tool, which I was able to use extensively in my thesis. I also specially thank Gaspar for the numerous of insightful discussions we've had that has been instrumental in my dissertation work.

I am grateful to my other colleagues, who have been helpful in one way or another. I thank the members of the Computational Sensory-Motor Systems Laboratory headed by Ralph Etienne-Cumming, especially Jamal Molin, Jack Zhang, and Adebayo Eisape, who I've had the pleasure to collaborate with over the years. I am grateful to Ebuka Arinze, who has been a brother to me and given me invaluable advice and support throughout the years. I would like to acknowledge my friends from JHU BGSA and the ECE GSA for all their support in and outside of school.

I would like to also thank the ECE department and staff including: Janel Johnson, Nicole Aaron, Debbie Race, Cora Mayenschein, Ruth Scally, Sathappan Ramesh, Eileen Miller, Laura Granite, Dana Walter-Shock, Melissa Gibbins, and Barbara Sullivan for all the work they've done in the background, and the timely help they given me over the years. I would also like to thank the Assistant Dean for Graduate and Postdoctoral Academic Affairs, Christine Kavanagh, who has assisted me through my time here at Hopkins.

I am grateful to the Meyerhoff Scholarship Program at UMBC for their financial and academic support throughout my undergraduate career. The Meyerhoff program was instrumental in preparing me for graduate school. Specifically, I would like to

thank late LaMont Toliver, Keith Harmon, Mitsue Wiggs, Sharon Johnson, Alicia Halls, and Michael Goodwin for all their support.

Last but not least, I am truly grateful for my family and friends. Right from an early age, my parents, Muibat and Mustapha Sanni, have instilled in me the value and importance of education. They taught me to work hard and to never settle. I am undoubtedly grateful for their untiring sacrifice and support throughout my life. I would like to thank my siblings, Olatunde Sanni, Folashade Sanni, and Adekunle Sanni, who have been very supportive of my academic career. I also extend my gratitude to my church community, including my pastors late Paul Akapo, Taiwo Fagbuyi, and Uwem Ekpo for all their support.

To quote Isaac Newton, "If I have seen further it is by standing on the shoulders of Giants."

Dedication

This thesis is dedicated to my parents, Muibat and Mustapha, who have inspired and supported me throughout my academic career.

Contents

List of Tables

LIST OF TABLES

List of Figures

Chapter 1

Introduction

Envisaged through Moore's law decades ago,[1] transistors have scaled to smaller sizes, which has paved the way for exponential improvements in the functionality and performance of integrated circuits over the years. Conventional computing paradigms, such as Von Neumann architecture, have improved through this scaling with faster processor speed and more functionality through higher circuit density. Nonetheless, the very nature of the Von Neumann architecture with the separation of computational and memory structures requires a constant stream of data between memory and processor, even in the ideal case of infinite bandwidth and zero latency. For realistic implementations with finite bandwidth and latency, designers make recourse to prefetching and caching schemes, which although resolves latency issues, do not however alleviate energy cost issues of constantly streaming data.

Furthermore as emerging technologies are scaling with conventional integrated

circuits, physical limiting factors affecting performance and power consumption has escalated and misaligned what is achievable in principle and in practice. As transistors have scaled to the order of a few nanometers, thinner dielectric has become more susceptible to leakage current; manufacturing variations create a higher disparity and mismatch in the design, and noise has become a bigger factor with supply voltage scaling. Unconventional architectures, such as an analog-array processor,[2] and charge-injection device (CID) processor,[3–6] have been proposed as energy efficient alternatives to existing paradigms to rectify this scaling issue.

In this work, systems and architectures for chip multiprocessors (CMPs) that exploit bio-inspired data encoding and mixed-signal processing are designed and optimized across different levels of abstraction and description for constructing heterogeneous chip multiprocessor. Conceptually, this hierarchical system design approach is visualized through the diagram shown in Figure 1.1. Generally, system design begins at the algorithmic level of description, where the system's functionality is established. Then, these algorithms or functions are mapped into a coherent architecture, where the the macro system blocks, communication protocols, data representation are defined in the architectural and representation level of description. This coherent architecture is then translated into an integrated circuit, which is implemented with physical devices from a technology process in the circuit and device level of description.

Specifically in this dissertation, work is done on the representation level of descrip-

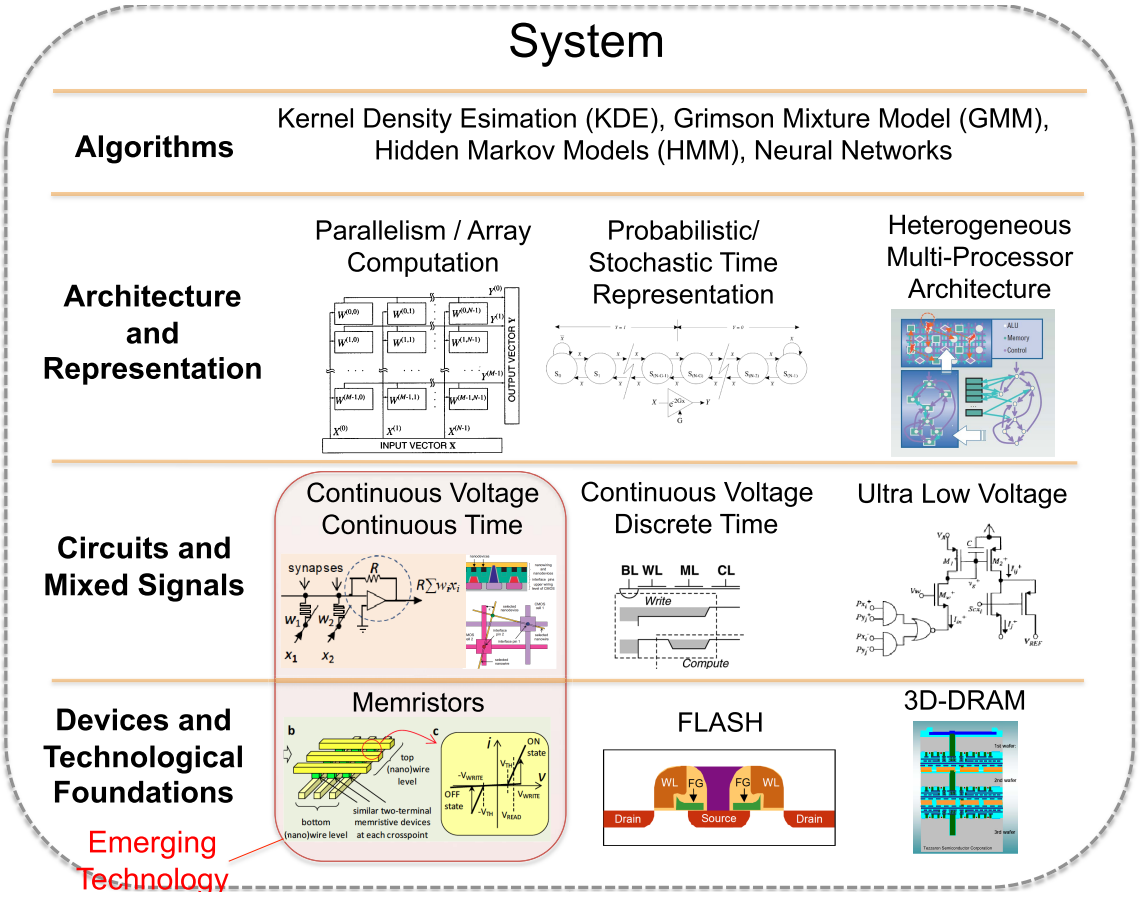


Figure 1.1: Levels of Abstraction for System Design of a Chip Multiprocessor.

tion that explores a random pulse-density modulation (RPDM) encoding in building computationally-efficient system for implementing machine learning and image processing algorithms. This alternative modality of encoding has been useful in stochastic computation for simplifying complex functions into bit-wise logic operations that can be implemented with much less area and energy costs compared to conventional computing systems.[7,8] Similar to neurobiological systems, such as the human brain that operates on low-dimensional spike trains, data in this modality of computation is implicitly encoded in the statistics of unary samples ergo allowing for a robust en-

coding that trades off precision (variance) for either time or resource. With continued technology trends pushing the brinks of transistor scaling, the need for error-tolerant systems and architecture that are resilient to scaling issues has made alternative computing paradigms, such as stochastic computing, attractive.

Moreover, on the level of circuit description, mixed-signal integrated circuits hinged on charge-based computing have been designed and fabricated in submicron CMOS processes for energy-efficient fixed point arithmetic. In the analog domain, products can be done implicitly and efficiently at the thermal noise limits on a capacitor when one input is encoded as the voltage, and the other as capacitance. Exploiting this principle, vector-vector multipliers were designed in this work that use programmable capacitor arrays for computing inner products and a sigma-delta ($\Sigma\Delta$) modulator for decoding these results back to the digital domain.[9] Additionally, a mixed-signal multiply and add processor based on a successive approximation architecture is designed and fabricated in a submicron CMOS process as a competitive alternative to conventional digital signal processor (DSP).

Finally, on the level of architectural description, different system blocks were designed for facilitating distributive computing in CMPs. A compact instruction-set processor that interfaces with both a switch circuit token-ring and packet-switching mesh network-on-chip is designed that works as a direct memory access (DMA) and processing unit (PU) controller on a SoC. This instruction set processor was integrated with a mixed-signal vector-vector multiplication processing unit (PU VVM),

an auxiliary memory unit (PU CACHE), and other PUs such as DSPs and morphological processors. Moreover, a bufferless scalable network architecture is designed and emulated in a network simulator. This network presented in this thesis not only provides a flexible communication solution that efficiently routes based on data traffic, but also prevents network deadlocks and livelocks, and can handle faulty routing channels due to increased fabrication defect with scaling network sizes. Other relating work includes the design of custom I/O cells implemented in a large SoC chip.

The culmination of this dissertation is the multifaceted design of heterogeneous mixed-signal processing units and system modules for a high performance computing and energy efficient 2.5D multiprocessor system-on-chip (MPSoC). Comprised of 12,544 VVM cores, a high bandwidth memory interface with 2 network on chips, ARM Cortex-M0 cores, and additional auxiliary units in 246.8mm^2 silicon area, this chip was designed to be useful for wide area motion imagery with gigabytes of framed images, large scale inference task with deep convolutional neural networks, and more.

This thesis is divided into 5 chapters. Chapter 1, this one, provides an introduction and overview of this work. Chapter 2 discusses a bio-inspired modality of data representation, and how it was used to construct computational efficient architecture predicated on stochastic computation primitives. Chapter 3 presents mixed-signal architectures iteratively designed and optimized for energy-efficient vector-vector multiplication. Moreover, Chapter 4 presents a mixed-signal architecture, inspired from the successive approximation analog-to-digital converter (SA ADC), as a competitive

alternative to conventional DSPs for multiply-add operations. Finally, Chapter 5 details the design of noteworthy units and modules of a heterogeneous CMP prototype implemented in a submicron CMOS process. Also in this chapter, some large-scale applications of this CMP are presented with results from test chips.

Chapter 2

Stochastic Representation for Computational Efficiency

The human brain is capable of computing over a quadrillion synaptic operations per second; all while only consuming roughly 12 watts.[10] Even with the exponential improvements in technology from advances in integrated circuits as envisioned through Moore's law,[1] the efficiency that is observed in human brains is still many orders of magnitude superior to current state of the art processors. The computational efficiency of these neurobiological systems can be attributed to clever exploitation of the underlining physics and mediums to efficiently represent and process information.

Generally, computing systems use time, space, and energy as physical resources for processing information. Conventional computing paradigms over the years have leveraged energy, while exploiting space and time for improving performance and

throughput. Nonetheless, now in the age of big data, the increased computation cost for intelligent data exploitation has made these computing systems inadequate and less desirable for efficient computing at a large-scale.[11] In this work, an alternative data representation, inspired from neurobiological systems, is explored for designing and implementing more computational efficient systems and architectures.

2.1 Data Representation

Biological and electronic signals can be categorized based on their representation in time and in value. Four main signal representations are distinguished in prior work.[12,13] Signals that are continuous in value, commonly known as analog signals, can either be represented as continuous-value continuous-time (CVCT) signals, such as sound signals, or as continuous-value discrete-time (CVDT) signals, such as clocked analog signals. On the other hand, signals discretized in value, conventionally known as digital signals, can be represented as discrete-value continuous-time (DVCT) signals, such as anisochronous pulse-time modulated (PTM) signals, or discrete-value discrete-time (DVDT), such as pulse code modulated (PCM) signals. An illustration of these signal representations with example signals is shown in Figure 2.1.

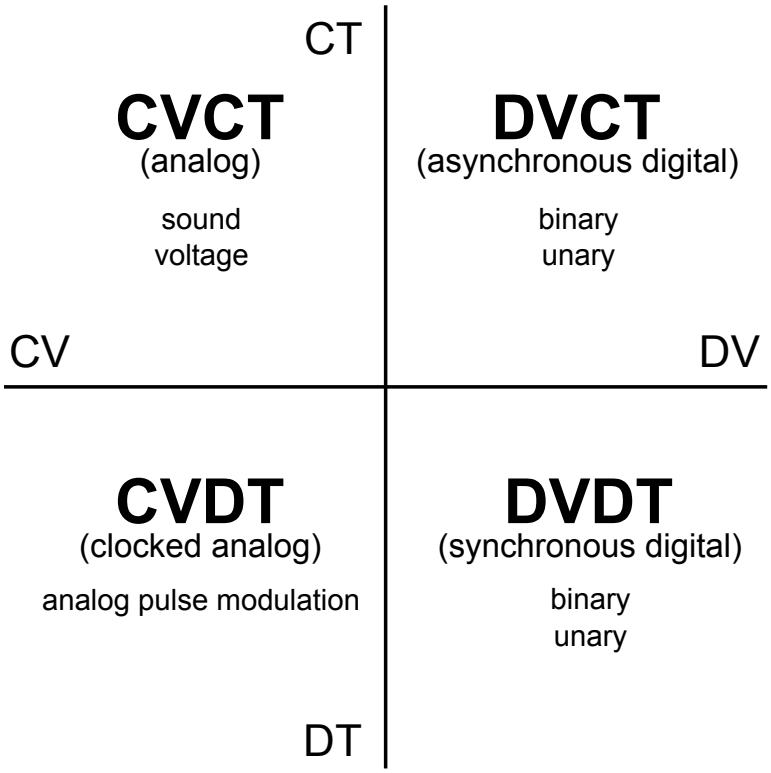


Figure 2.1: Four Types of Signal Representation. The graph is broken up into four quadrant separated by continuity in time and value. In the upper left quadrant are continuous-value continuous-time (CVCT) signals. In the upper right quadrant are the discrete-value continuous-time (DVCT) signals. In the lower left quadrant are the continuous-value discrete-time (CVDT) signals. In the lower right quadrant are the discrete-value discrete-time (DVDT) signals.

2.1.1 Analog vs. Digital

Even with the overwhelming popularity and integration of DVDT signal representation as the standard for conventional computing paradigms, there are distinct advantages and disadvantages within each of these analog and digital domains that can be exploited depending on the application. Generally, analog signals need less resources for information encoding as many bits can be encoded in one wire, while digital signals encode 1-bit of information per wire. Nonetheless, analog signals are

more susceptible to inaccuracies in encoding because of mismatches in physical devices and noise from thermal fluctuations or coupling devices, and digital signals are more robust due to less information encoding per resource. Additionally, computation in the analog domain can be done using underlining physic governed by laws such as Kirchoff's current and voltage laws (KCL and KVL), which greatly simplifies system implementation. However, analog signals are not restored after computation stages compared to digital signals, and can thus accumulate noise from cascaded stages. Consequently, complex digital systems with many computing stages are generally easier to design than complex analog systems with many computing stages. These advantages and disadvantages across domains highlight trade-offs that can be exploited for computational efficiency.

2.1.2 Unary vs. Binary

In digital representation, information can be encoded in different schemes based on time and space. Standard digital computers, which are based on a binary numeral system, use positional notation to represent order of magnitudes in a base-2 format, where each bit represents either a '0' or '1' value. This system has been widely adapted for decades because of its straightforward implementation in digital logic gates. Alternatively, a unary coding scheme such as pulse width modulation (PWM) or random pulse density modulation (RPDM) can also be adapted which equally weights all bits in a vector encoding information. Consequently, bit ordering

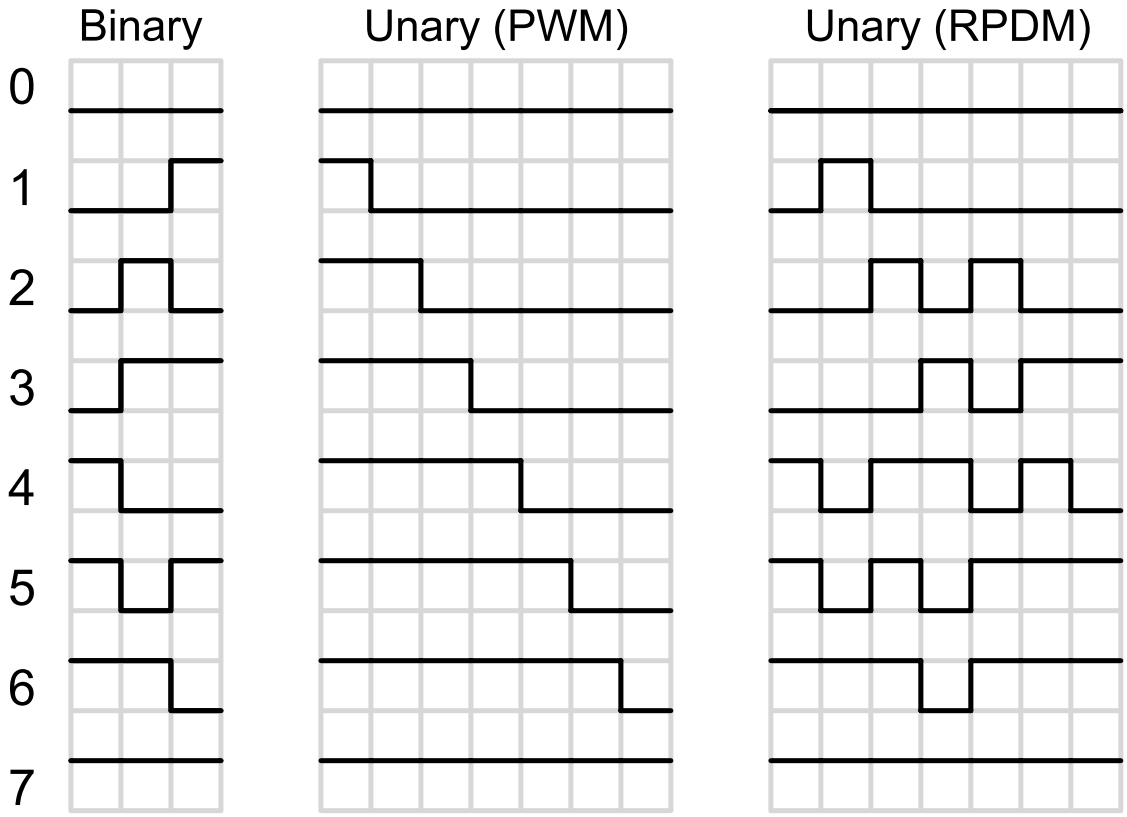


Figure 2.2: Examples of Binary and Unary Encoding. The numbers 0-7 encoded in 3-bit binary, 7-bit unary PWM, and 7-bit unary RPDM.

is unimportant in these representations as information is encoded as the density of '1's in the vectors. Figure 2.2 shows an example of data encoding in both binary and unary representations. Evidently, digital binary representation is more compact than unary representation as the number of bits in binary scales logarithmically with the number of quantifiable levels as opposed to linearly for unary coding. Nonetheless, unary encoding schemes are more error-tolerant as the error injected per bit flip is constant and fixed at $1/2^N$ for N-bit resolution in a range of $[0, 1]$, while a bit flip in a binary representation can potentially inject an error equivalent to half

the range (0.5 for a similar range of $[0, 1]$). With transistors scaling to the orders of a few nanometers, parasitic factors from manufacturing variations and noise have become a more significant concern, which has made digital computing systems based on standard binary-radix arithmetic less efficient. In Section 2.2, a bio-inspired unary representation based on stochastic encoding is detailed as an alternative for computing architectures.

2.2 Stochastic Encoding and Computing

Formally introduced in the 1960s, stochastic computing has been an alternative to traditional digital logic that exploits probabilistic encoding of information to simplify complex computations to simple bitwise logic operations.[7,8] In stochastic computing, a multi-bit value X is encoded temporally or spatially as a random unary stream. A sample of this unary stream is denoted as a Bernoulli-distributed *stochastic number* x_k, whose mean value $E[x_k] = p$ encodes X by mapping its range to $[0, 1]$. Figure 2.3 shows an example of encoding the value $4/8 = 0.5$, which can be represented as 4 in a 3-bits unsigned binary representation, as a vector of stochastic numbers.

Moreover, in this domain, operations such as multiplication, weighted-sums, absolute differences, and more can be mapped into simple logic functions as seen in Figure 2.4. Multiplications can be done with an AND logic gate, absolute differences can be done using an XOR logic gate, and weighted sums can be done using a

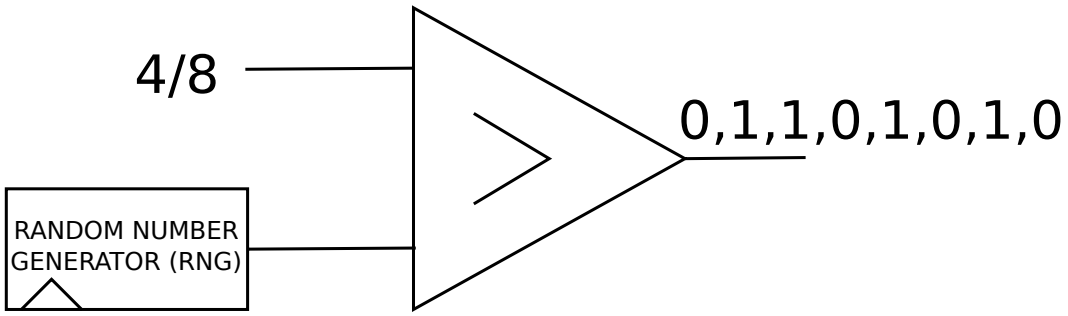


Figure 2.3: Stochastic Number Encoding. Encoding 4/8 into a vector of stochastic numbers using a random number generator (RNG) and a digital comparator.

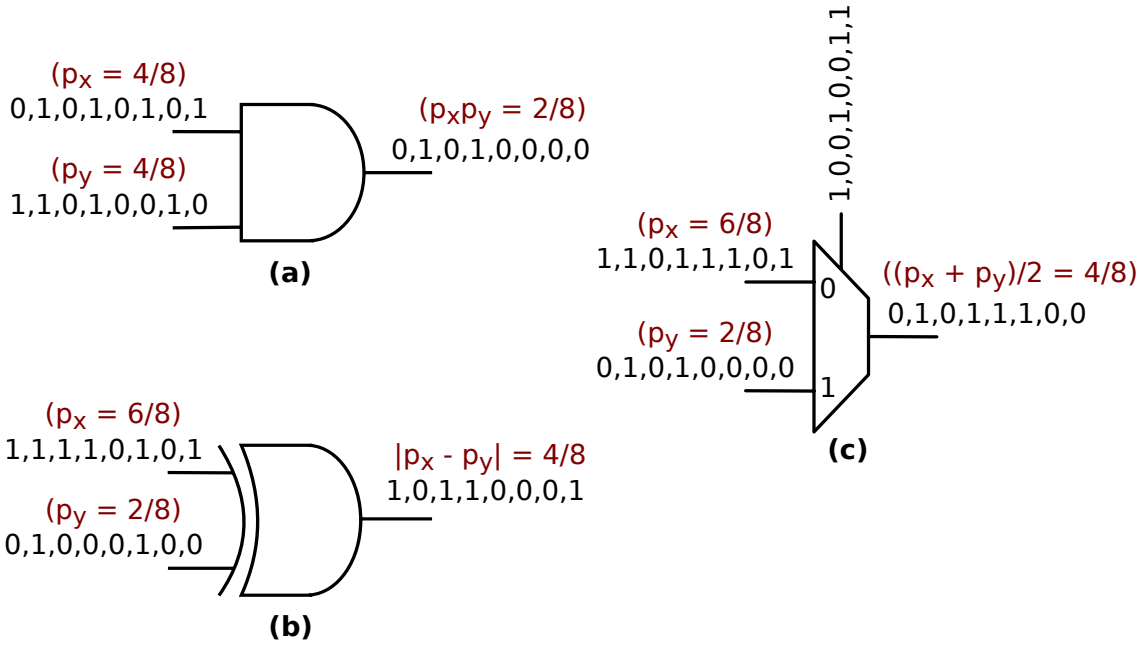


Figure 2.4: Stochastic Operations. (**a**) An AND logic gate for multiplication. (**b**) A XOR logic gate for absolute difference (**c**) A multiplexer for weighted sum of two inputs.

multiplexer.

After computing in this domain, the resulting stochastic numbers can be decoded back into a binary number using a counter, and then scaling the count to the appropriate output range. This decoding can be seen in the example illustrated in Fig-

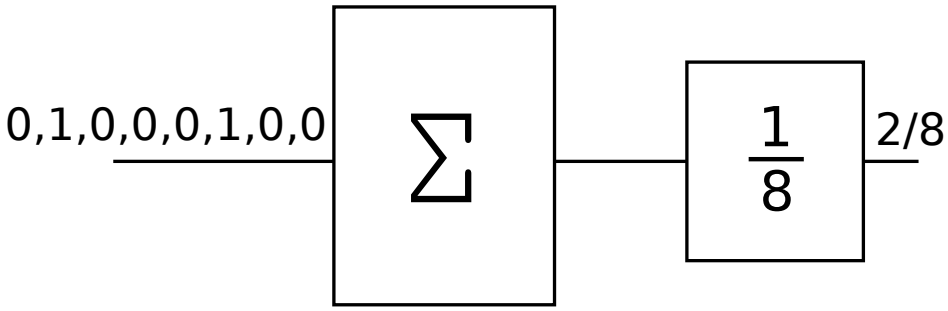


Figure 2.5: Stochastic Number Decoding.

ure 2.5. Similar to neurobiological systems, such as the human brain that operates on low-dimensional spike trains, information in the stochastic domain is implicitly represented in the statistics of the stochastic numbers ergo allowing for a robust encoding with a trade-off of variance for either time or space.

Stochastic computation has been useful for a variety of applications. Specifically, stochastic computation has been exploited for numerous of practical applications in image processing. Peng Li et al. explore stochastic computational elements for implementing digital image processing algorithms.[14,15] In both works, the authors show how computational intensive functions such as the tanh and exponentiation functions can be implemented with sequential logic in the stochastic domain for doing fault-tolerant image segmentation. A comparison of hardware implementations of kernel density estimation (KDE) image segmentation algorithm with conventional binary radix logic and stochastic logic shows the conventional implementation rapidly degrading with the increased injection of soft errors, while the stochastic implementation is resilient to these errors. Additionally, Weikang Qian shows how any function can be evaluated and approximated using synthesizable Bernstein polynomials, which

can be implemented in a simple reconfigurable stochastic architecture;[16,17] this work has been applied to cutting hardware cost for image processing applications such as gamma correction. Also, an intelligent imaging device that integrates stochastic arithmetic with a digital pixel array for doing local image processing is presented in.[18] Nonetheless, much more work has been done in integrating stochastic computation based architectures for image processing.[19–21]

In addition to image processing, stochastic computation has also been applied to simplifying computational intense arithmetic operations such as division and square-root,[22] matrix operations such as inverse-matrix transforms,[23] error-correction for digital communication standards,[24] neural networks,[25–27] and more. With continued technology trends pushing the brinks of transistor scaling, the need for robust systems and architecture that are noise-tolerant has made alternative computing paradigms, such as stochastic computing, attractive.

2.2.1 Precision Analysis

By analyzing the encoding and decoding operation in stochastic computing, an expression of the output precision as a function of the number of stochastic numbers can be derived. As briefly described in Section 2.2 and also shown in Figure 2.6, a stochastic number x_k encoded from a binary input X, follows a Bernoulli distribution with a probability of 1 given as $p = X/2^M$, where M is the bit precision of X. That

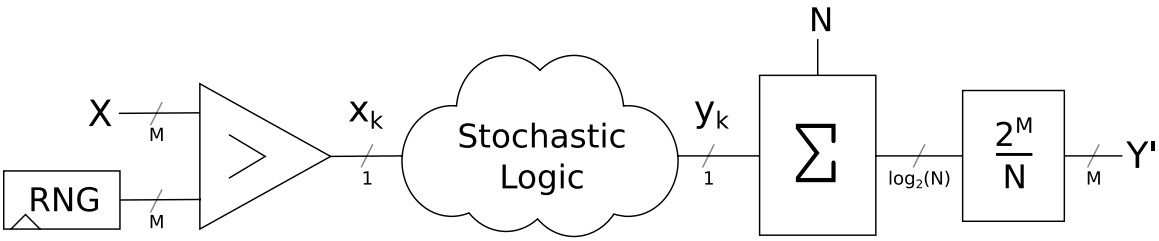


Figure 2.6: Stochastic Encoder, Logic, and Decoder. A binary input X is encoded into stochastic numbers, transformed through logic, and then the result is decoded into a binary output Y'.

is,

$$x_k \sim Bern\left(p = \frac{X}{2^M}\right) \tag{2.1}$$

Moreover, the decoded output Y' from summing and scaling the resulting stochastic number outputs y_ks, as shown in Figure 2.6, is a Binomially-distributed random variable that approximates the output Y. Assuming that the random numbers y_k are independent and identically distributed (i.i.d), then by the central limit theorem, Y' can be approximated as

$$Y' \sim B\left(n = N, p = Y\right) \xrightarrow[N\to\infty]{} \mathcal{N}\left(\mu = Y, \sigma^2 = 2^{2M}\left(\frac{\frac{Y}{2^M}(1-\frac{Y}{2^M})}{N}\right)\right), \tag{2.2}$$

where N is the integration count, and M is the bit precision of the output Y. Noticeably shown in the Equation 2.2, the variance σ^2 of the normal distribution approximating the output Y is inversely proportional to the integration count N. Thus, the variance in the stochastic domain can be tuned based on the integration count, as increasing the count reduces variance and decreasing the count increases the variance.

Moreover, since the decoded output is a random variable, an absolute finite bound can not be derived for the precision of this output. Nonetheless, a probability measuring confidence on an error bound in the decoded output can be deduced, and it can be used as an indication of certainty to attain a particular output precision. The output precision is derived from the number of quantifiable levels across the output range. For an output to attain a certain precision of M-bits, all adjacent output levels must be discernible by 1 least significant bit (LSB) with a value of $1/2^M$. Using the LSB value as a bound on the standard deviation σ of the decoded output's distribution, a probability can be derived from the cumulative distribution function (CDF) that expresses a certainty in a confidence interval centered around the mean. In the context of the decoded output, this probability signifies the certainty that a specific output level will have at most a specific standard deviation or error. In the case of an arbitrary bound of 2σ, the probability of this confidence interval can be deduced as

$$P(\mu - 2\sigma \leq Y' \leq \mu + 2\sigma) \approx 0.95. \tag{2.3}$$

For a value of 2σ bounded by half of an LSB (half because adjacent codes with this same error will sum to 1 LSB), Equation 2.3 can be modified to

$$2\sigma = 2\sqrt{\frac{p(1-p)}{N}} \leq \frac{1}{2}\frac{1}{2^M}, \tag{2.4}$$

where p is the expected output. Equation 2.4 can be simplified to

$$p(1-p) \cdot 2^{2M+4} \leq N. \tag{2.5}$$

For the worst case probability $p = 0.5$, the total N needed to guarantee 95% of samples with M-bit precision is bounded by

$$2^{2M+2} \leq N \tag{2.6}$$

Generalizing for 1σ, 2σ, and 3σ, the plot in Figure 2.7 shows the necessary integration count N to achieve different bit precision with different certainty on a logarithmic scale, where $P(s) = P(X - k\sigma \leq Y' \leq Y + k\sigma)$ for $k\sigma$s. The integration count scales exponentially with the output bit precision. Although manageable for lower bit precision, the large integration count for high bit precision makes stochastic computation less ideal for applications where accuracy and precision are critical.

2.2.2 Pseudo-Random Numbers

Inaccuracies in stochastic computation may be attributed to three main factors: random fluctuations in the stochastic number representation, correlation across the numbers that are being computed, and physical errors.[19] As described in Section 2.2.1, random fluctuations are accredited to the statistics of the random numbers, which

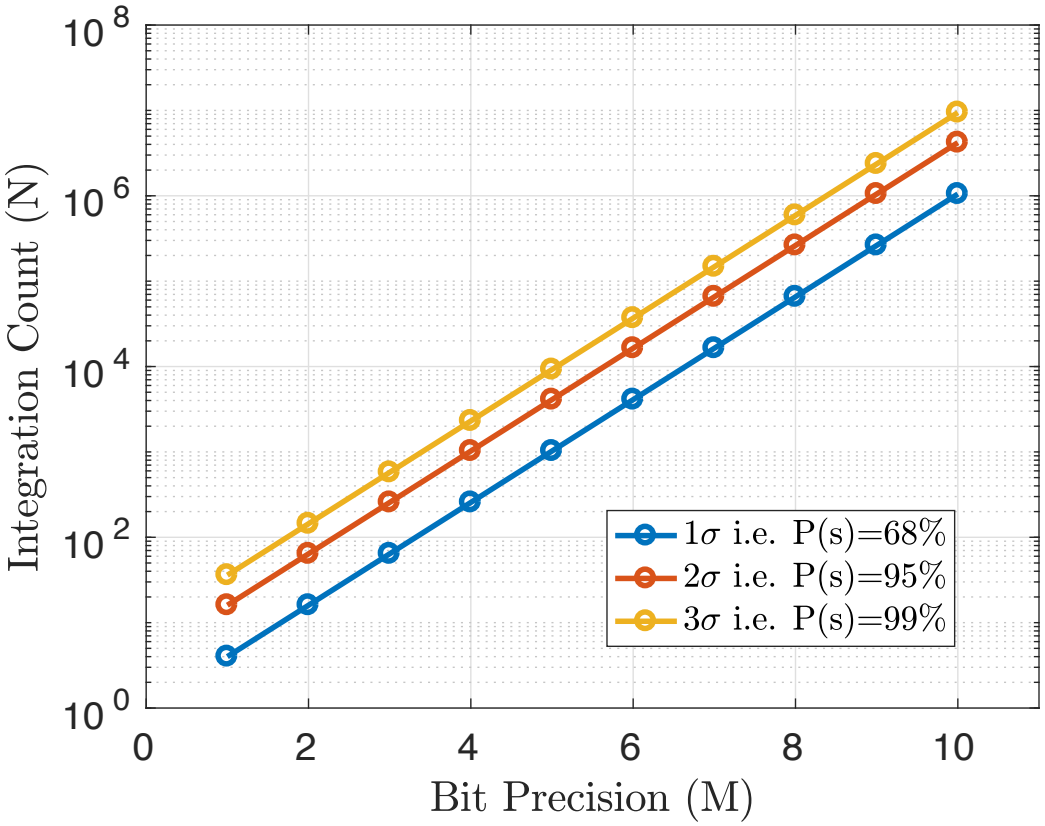


Figure 2.7: Stochastic Number Precision. The plot shows the necessary integration count to achieve a certain bit precision for a given confidence interval. $P(s)$ represents the probability of samples lying within $s\sigma$ of the mean.

produce erroneous stochastic numbers. Pseudo-random numbers, such as maximum-length sequences from linear feedback shift registers (LFSR), can be used to eliminate random fluctuations induced into the stochastic numbers during encoding. With the appropriate feedback taps, LFSR can generate periodic maximal-length sequences, which although deterministic have good statistical properties, and have proven useful for stochastic computation architectures.[14,19,20,27] Thus, using a M-bit maximal-length LFSR for the random number generator in Figure 2.3 and 2.6, guarantees an

exact representation of the input X in the stochastic domain in $2^M - 1$ samples, and hence removing random fluctuations in the stochastic number during encoding. Nonetheless, inaccuracies can still be introduced from correlated stochastic numbers and physical errors.

2.3 A Deep Belief Network Using Stochastic Computation

Recently, deep belief networks, a subset of deep neural networks, have shown remarkable performance in a variety of classification and recognition tasks involving vision,[28,29] speech,[30,31] and more. Exemplary at dimensionality reduction,[32] DBNs have excelled at high-dimensional complex classification task, such as image classification, because of their ability to encode high-dimensional data as low-dimensional features over multiple layers. DBNs have also shown a significant advantage of performance scalability with network size. However, due to the computational intensive nature of these networks, DBNs have not been well suited for modern computing implementations for real world tasks. Nonetheless, work has been done to implement more efficient DBNs that optimize for time using Graphical Processing Units (GPUs)[33,34] and that optimize for power implementing spike-based and event-driven system.[35–39] Under this dissertation work, unconventional methodologies based on stochastic computing are explored for designing a computational efficient DBN in an

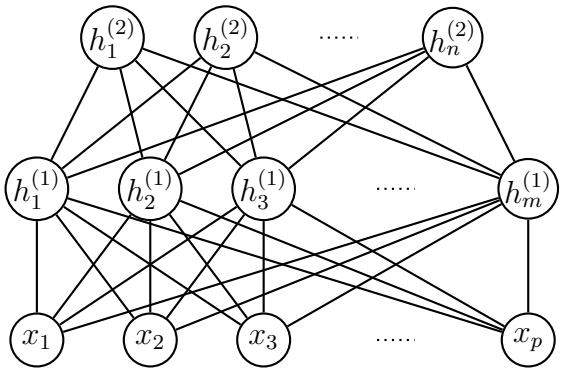


Figure 2.8: A 2 Hidden Layer Deep Belief Network. Hidden layer $h^{(1)}$ encodes features from the input layer x, and hidden layer $h^{(2)}$, which is the output layer of this network, encodes more abstract features from the features in hidden layer $h^{(1)}$.

attempt to optimize for power, while still maintaining good performance.[27]

2.3.1 Background and Theory

Deep belief networks, which were introduced by Hinton and his collaborators in 2006,[40] are multi-layered generative graphical models that represents layers as Restricted Boltzmann Machines (RBMs),[41] and greedily train them with unsupervised learning algorithms.[42] Figure 2.8 shows the graphical model of a 2-hidden layer DBN. RBMs, which are the building blocks of DBNs, are energy-based models that encode useful properties of the variables in the shape of its energy function. Generalized from exponential family models, a probability distribution can be defined through an energy function as

$$P(x) = \frac{e^{-\mathcal{E}(x)}}{Z}, \tag{2.7}$$

where $\mathcal{E}(x)$ is the energy function, and Z is the normalization factor $\sum_x e^{-\mathcal{E}(x)}$ known as the partition function. Applying gradient descent on the log-likelihood of the observed data, an energy-based model like RBMs can learn features of the data.

Boltzmann machines have the modeling capacity to represent complex distributions through a set of observed visible units $\mathbf{x}$, and unobserved hidden units $\mathbf{h}$. This modifies the probability distribution defined in Equation 2.7 to

$$P(\mathbf{x}) = \sum_{\mathbf{h}} P(\mathbf{x}, \mathbf{h}) = \sum_{\mathbf{h}} \frac{e^{-\mathcal{E}(\mathbf{x},\mathbf{h})}}{Z} \tag{2.8}$$

Moreover, RBMs form bipartite graphs between these layers of visible units $\mathbf{x}$, and hidden units $\mathbf{h}$, with no intra-layer connections. This creates a conditional independence between the visible and hidden units across the layers, which mathematically can be seen as

$$\begin{aligned} p(\mathbf{h}|\mathbf{x}) &= \prod_i p(\mathbf{h_i}|\mathbf{x}) \\ p(\mathbf{x}|\mathbf{h}) &= \prod_j p(\mathbf{x_j}|\mathbf{h}) \end{aligned} \tag{2.9}$$

The energy function of an RBM is defined as

$$\mathcal{E}(\mathbf{x}, \mathbf{h}) = -\mathbf{b}^T\mathbf{x} - \mathbf{c}^T\mathbf{h} - \mathbf{h}^T\mathbf{W}\mathbf{x} \tag{2.10}$$

where $\mathbf{b}$ and $\mathbf{c}$ are vectors of offset values associated with the visible units in vector $\mathbf{x}$

and hidden units in vector $\mathbf{h}$ respectively, and $\mathbf{W}$ is a weight matrix associated with visible to hidden unit connections.

In the case that $\mathbf{h_i}$ and $\mathbf{x_j}$ are binary units, $P(\mathbf{h_i}|\mathbf{x})$ and $P(\mathbf{x_j}|\mathbf{h})$ from Equation 2.9 can be derived as

$$\begin{aligned} P(\mathbf{h_i} = 1|\mathbf{x}) &= \frac{e^{-(-\mathbf{c_i} - \mathbf{W_i x})}}{1 + e^{-(-\mathbf{c_i} - \mathbf{W_i x})}} = sigm(\mathbf{c_i} + \mathbf{W_i x}) \\ P(\mathbf{x_j} = 1|\mathbf{h}) &= \frac{e^{-(-\mathbf{b_j} - \mathbf{W_j h})}}{1 + e^{-(-\mathbf{b_j} - \mathbf{W_j h})}} = sigm(\mathbf{b_j} + \mathbf{W_j h}) \end{aligned} \tag{2.11}$$

where $sigm$ is the sigmoid function, commonly used as the activation function in artificial neurons.

In training a deep belief network, the log-likelihood gradient is computed on training data for learning parameters associated with the energy function. For the distribution defined in Equation 2.8, this gradient is deduced as

$$\frac{\partial \log p(x)}{\partial \theta} = \frac{\partial \mathcal{F}(x)}{\partial \theta} - \frac{1}{Z} \sum_{\tilde{x}} e^{-\mathcal{F}(x)} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta} \tag{2.12}$$

where x is a single visible unit, h is a single hidden unit, the partition function $Z = \sum_x e^{-\mathcal{F}(x)}$, and $\mathcal{F}(x)$ is the free energy function inspired from physic, which is given as

$$\mathcal{F}(x) = -\log \sum_h e^{-\mathcal{E}(x,h)} \tag{2.13}$$

Computing this gradient is intractable because of the second term in Equation 2.12, which requires taking the expectation over all configurations of $\tilde{x}$ under the distri-

bution of the model. As opposed to deriving an analytical expression for the log-likelihood gradient, a Markov Chain Monte Carlo (MCMC) sampling method, such as Gibbs sampling, is used for estimating this gradient. Because of the conditional independence of intra-layer units in RBMs, block Gibbs sampling can be applied for iteratively sampling hidden and visible units in a Markov chain for training the network. This sampling chain can be visualized as

$$\mathbf{x}^{(0)} \Rightarrow \mathbf{h}^{(0)} \Rightarrow \mathbf{x}^{(1)} \Rightarrow \mathbf{h}^{(1)} \Rightarrow \ \ldots \ \mathbf{x}^{(t)} \Rightarrow \mathbf{h}^{(t)}$$

As $t \to \infty$, $\mathbf{x}^{(t)}$ and $\mathbf{h}^{(t)}$ are guaranteed to be accurate samples from $p(\mathbf{x}, \mathbf{h})$. However, since the luxury of sampling till convergence can not be afforded, alternative algorithms have been devised for efficiently sampling for learning.

One in particular, that has gained a lot of popularity, is the Contrastive Divergence (CD_k) learning algorithm.[43] CD_k initializes the Markov chain to samples from the training set, and then generates samples from just k-steps of Gibbs sampling. This learning algorithm has been applied for unsupervised training of DBNs with great success, even with only one step of Gibbs sampling ($k = 1$).

The parameter update equations for training a network with binary units using

the CD_k algorithm is given as

$$\begin{aligned}
\mathbf{W}^{(t+1)} &= \mathbf{W}^{(t)} + \eta(\mathbf{h}^{(t)}\mathbf{x}^{(t)} - sigm(\mathbf{c_i}^{(t)} + \mathbf{W_i}^{(t)}\mathbf{x}^{(t)})\mathbf{x}^{(t+1)}) \\
\mathbf{b}^{(t+1)} &= \mathbf{b}^{(t)} + \eta(\mathbf{x}^{(t)} - \mathbf{x}^{(t+1)}) \\
\mathbf{c}^{(t+1)} &= \mathbf{c}^{(t)} + \eta(\mathbf{h}^{(t)} - sigm(\mathbf{c_i}^{(t)} + \mathbf{W_i}^{(t)}\mathbf{x}^{(t)}))
\end{aligned} \tag{2.14}$$

where η is the learning rate, and $\mathbf{h}^{(t+1)}$ and $\mathbf{x}^{(t+1)}$ are samples generated from their conditional probabilities.

Furthermore, with top layer supervised learning, DBNs can also be implemented as deep feedforward neural networks for applications in discriminative tasks, such as classification.

2.3.2 Architecture and Implementation

For this work, a deep belief network, implemented with stochastic computation primitives, was constructed for doing classification. The network, consisting of 1 visible layer of 784 units (28x28 image input), and 2 hidden layers of 200 and 10 units respectively, was trained offline in MATLAB on the MINST training dataset using conventional techniques and algorithms detailed in Section 2.3.1, and then implemented as a feedforward neural network for classification.

In this feedforward network, a set of input data is fed into the first layer of the network, and then features in the hidden units are hierarchical computed till the top

layer. After a full hidden layer is computed, it becomes the visible layer for the next hidden layer. For each hidden unit, the conditional probability of that unit given the visible units, defined in Equation 2.9, is computed. These probability values are then used to generates samples used in the next layer. Computing Equation 2.9 for each hidden unit is a non-trivial task, and so stochastic computation is exploited for simplifying the computation.

First, the inputs x, weights W, and offsets c are encoded into stochastic numbers. Because the weight and offset parameters are unconstrained during training, and can be negative or positive, a bipolar coding format is used, where '0's represents -1 and '1's represents $+1$ in the stochastic domain. This is achievable with the same architecture described in Figure 2.3, given that the random number generator is uniformly distributed across the same domain as the signed input.

In the stochastic domain, the multiplication of the weight and the input can be done efficiently with a simple XNOR gate, which was derived from the truth table. Then, using an up-down counter that counts up for every '1' in the stochastic numbers and down for every '0' in the stochastic numbers, all the weighted inputs can be summed up together. The result from this step is then normalized to the appropriate range.

Finally, the sigmoid function is computed on the weighted sum to derive the output for the hidden unit. To simplify this computation, a piecewise-linear function

approximating the sigmoid activation function is computed instead as:

$$p(\mathbf{h_i} = 1|\mathbf{x}) = \begin{cases} 0, & \text{if} \quad (\mathbf{c_i} + \mathbf{W_i x}) \leq -4 \\ \frac{1}{8}\Big(\mathbf{c_i} + \mathbf{W_i x}\Big) + \frac{1}{2}, & \text{if} \ -4 < (\mathbf{c_i} + \mathbf{W_i x}) < 4 \\ 1 & \text{if} \quad (\mathbf{c_i} + \mathbf{W_i x}) \geq 4 \end{cases} \tag{2.15}$$

Stochastically, this piecewise-linear function can be computed by offsetting the counter for integrating the stream, and bit-shifting the result from the integration. Figure 2.9 shows a plot of the sigmoid activation function compared to the approximate piecewise-linear function. Moreover, the same approximate piecewise-linear activation function used for classification, was also used for training.

The architecture for computing each hidden unit, given the visible units was composed, and can be seen in Figure 2.10. Because of the independence between intra-layer hidden units imposed from the restriction on the Boltzmann machines, these computation blocks were parallelized for the hidden units in a layer, which allowed for a trade-off between computation time and resources. The full architecture is synthesized in a Kintex 7350 field programmable gate array (FPGA) board, as seen in Figure 2.11. Optimized for the classification task, 20 parallel computation nodes, RAM for locally storing the parameter and the activation outputs, and some periphery logic to control the nodes, were implemented in the FPGA. Moreover, the architecture was designed with a generic framework to allow for parameterizable synthesis tailored for

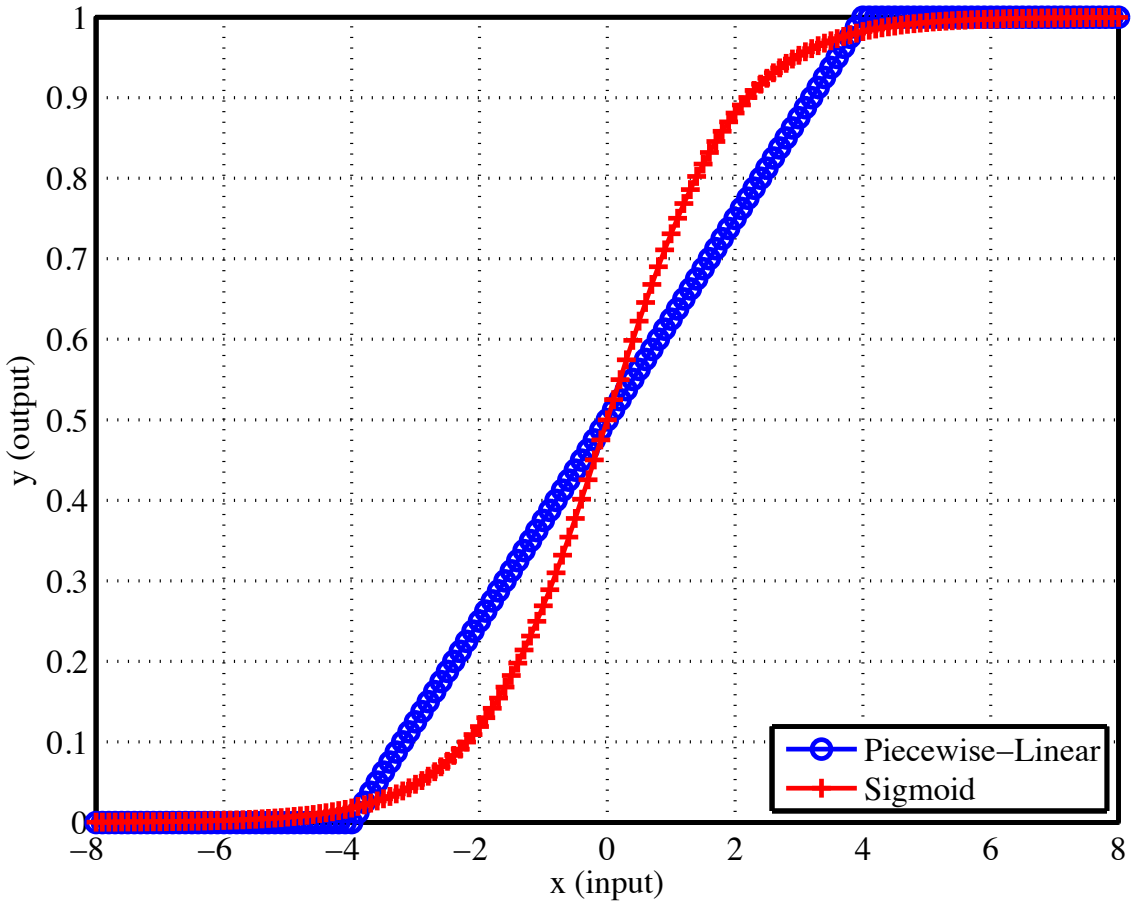


Figure 2.9: Sigmoid Activation Function and an Approximate Piecewise-Linear Function. The sigmoid activation function is in blue, and the approximate piecewise-linear function is in red.

the application.

2.3.3 Results

This deep belief network architecture is evaluated using the MNIST database of handwritten digits.[44] After training in MATLAB, and implementation on a FPGA

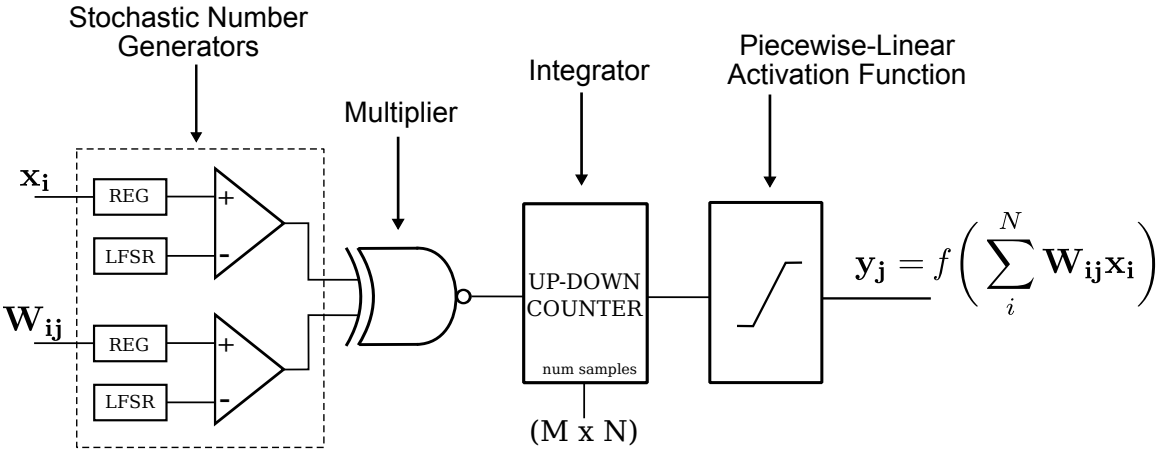


Figure 2.10: Architecture for the Computation Node for the DBN Using Stochastic Logic.

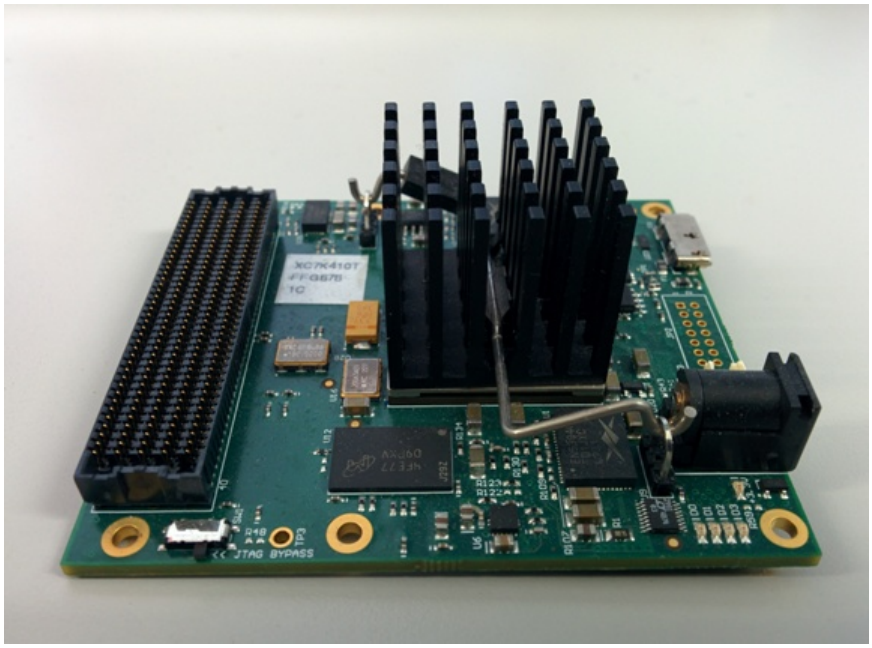


Figure 2.11: Kintex FPGA board used for implementing the DBN architecture.

board, this deep belief network was evaluated at variable computation precision (varied with the number of stochastic numbers) on the test dataset of the MNIST database. A stochastic computation precision of 8 bits to 16 bits is used in this test-

Table 2.1: DBN Experimental Results.

Computation Precision	Mean Abs. Error	Classifications per Second	Test Acc (%)
8 bits	0.1580	24.59	10.3
9 bits	0.0609	23.39	69.5
10 bits	0.0430	19.93	81.8
11 bits	0.0151	16.60	92.4
12 bits	0.0107	12.19	94.1
13 bits	0.0090	8.16	94.2
14 bits	0.0038	4.88	94.0
15 bits	0.0019	2.72	94.2
16 bits	0.0009	1.45	94.2

ing, which corresponds to 256 and 65536 samples respectively (i.e 2^N relationship). These results are compared to results produced from this network implemented using conventional arithmetic logic with double floating-point precision in MATLAB. For analyzing the accuracy of this computation, the mean absolute error of the top layer activation using stochastic computation, with respect to the activation from the network implemented with conventional arithmetic, is computed. Table 2.1 displays for each stochastic computation precision, the mean absolute error, the classification rate on the FPGA, and the classification accuracy.

Also, a plot of the classification accuracy over different stochastic computation precision is included in Figure 2.12, and the mean absolute error is plotted in Figure 2.13.

Computing in the stochastic domain with 12-bit stochastic computation precision or more generated results similar to the results produced with double floating-point precision.

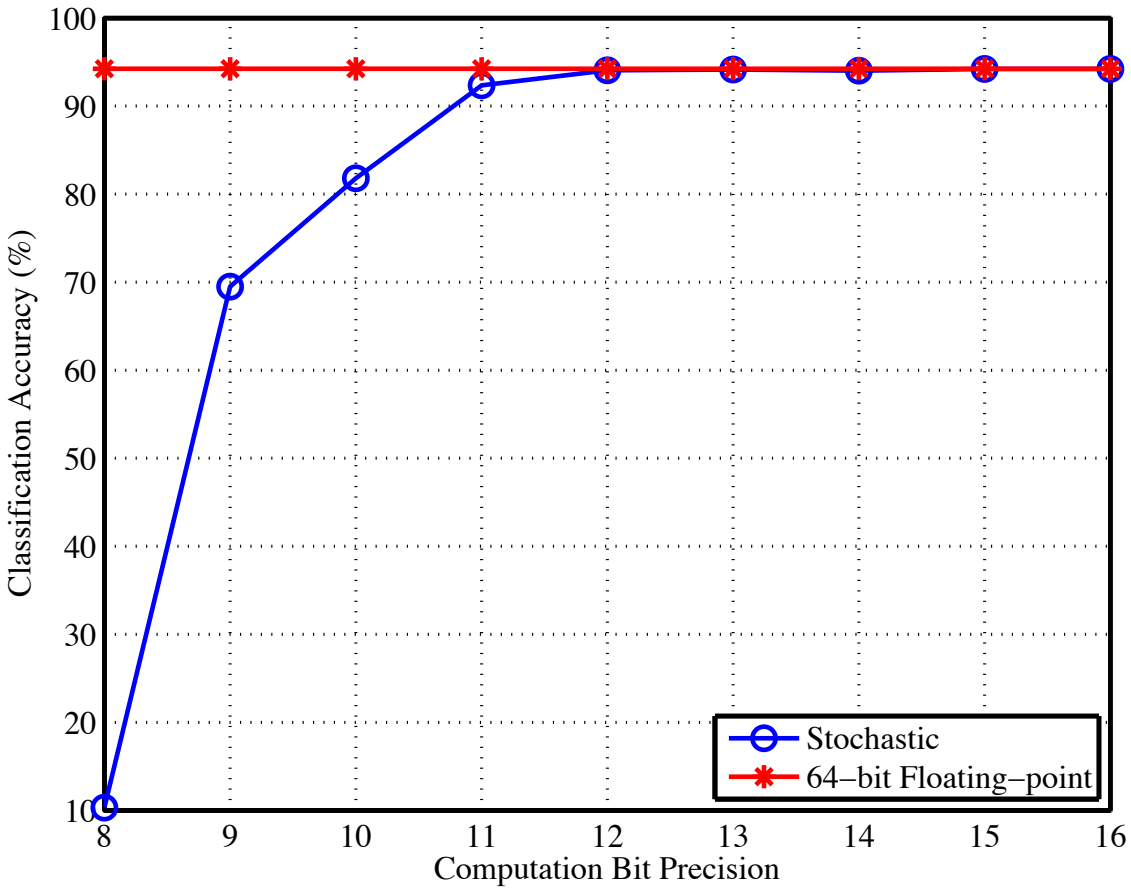


Figure 2.12: Plot of the Classification Accuracy on the MNIST Test Dataset with Different Stochastic Computation Precision for the Network. The plot in red is the classification accuracy attained with a network implementing conventional binary arithmetic at double floating-point precision. Plot in the blue is the accuracy using variable stochastic computation precision.

2.4 Adaptive Background Modeling Using Stochastic Computation

Background modeling and subtraction is widely used in a variety of signal processing applications to model the background and distinguish foreground data for tasks

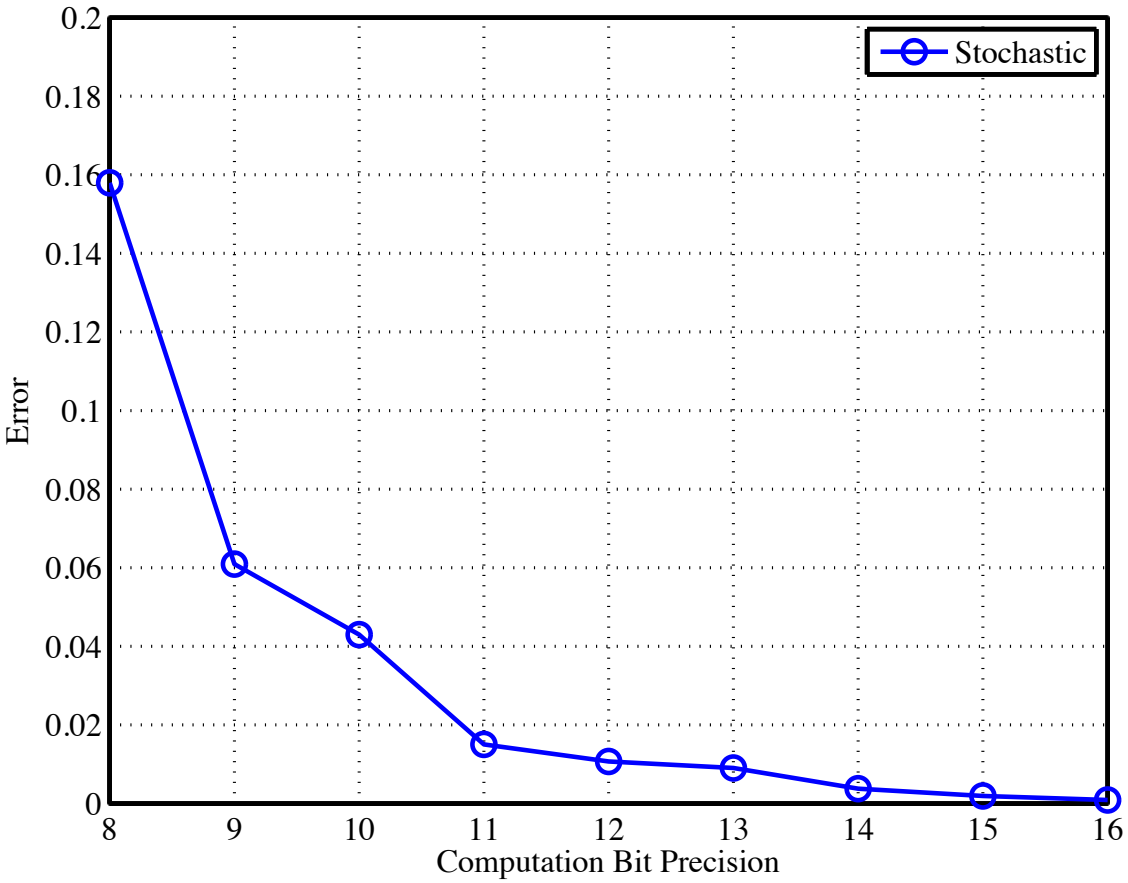


Figure 2.13: Plot of the Mean Absolute Error of the Top Layer Activation on the MNIST Test Dataset. The mean absolute error for each precision in the stochastic computation is computed with respect to the top layer activation results with double floating-point precision.

involving classification, recognition, surveillance, and more. Some state of the art implementations for image processing have used simple methodologies such as frame difference across previous images for foreground detection, and arithmetic mean across successive images to construct the background model[45].[46] However such architectures don't capture variance in the background pixels due to fluctuations in the lighting

and scenery, variation in the image sensor, and more. On the otherhand, statistical models for background modeling and subtraction have seen a lot of success for this application; this includes a background subtraction method based on an online mixture model of Gaussian distributions,[47] an online Bayesian method,[48] and more. Additionally, many other methods have been developed over the years, notably, a method based on a feed-foward neural network.[49] Andrew Sobral presents a comprehensive review of these background subtraction algorithms evaluated with both synthetic and real datasets.[50]

2.4.1 Parametric Probabilistic Background Model

One of the most ubiquitous image background modeling and segmentation methods is based on a parametric probabilistic background modeling algorithm proposed by Stauffer and Grimson.[47] In this image segmentation method, each pixel is independently modeled by a mixture of weighted Gaussian distributions, which can be expressed as

$$P(X_t) = \sum_{i=1}^{K} w_{i,t} \cdot \eta(X_t, \mu_{i,t}, \Sigma_{i,t}), \tag{2.16}$$

where $P(X_t)$ is the probability of observing the current pixel sample X_t in a composite of k Gaussian distributions with weights $w_{i,t}$ and mean and covariance values $\mu_{i,t}$ and $\Sigma_{i,t}$ respectively. Through a background modeling process, each distribution is labeled either foreground or background. Then, using a set of heuristic update rules,

Table 2.2: Parameters for Stauffer et al. Image Segmentation Algorithm.

Parameters	Description	Nominal Values
α	learning factor	0.05
k	number of distributions	3
w_0	initial weight	0.3
σ_0^2	initial variance	0.0138
β	distribution threshold	2.5
b_{th}	background threshold	0.75

the parameters of each pixel's distributions are updated for each new image frame, and the current pixels are classified based on their matching distribution's label. By modeling each pixel with a multitude of distributions, this method can adaptively learn multimodal background scenes, which is useful for dynamic scenes with various of lighting changes and flickering events. This image segmentation method is governed by four main processes: pixel initialization, pixel match evaluation, distribution parameter update, and distribution labeling for classification.

Each pixel's distribution parameters are initialized in the beginning of this algorithm. These parameters, which are listed in Table 2.2, include a learning factor α, number of distributions k, initial distribution weight w_0, initial variance σ_0^2, distribution match threshold β, and background threshold b_{th}. The nominal values shown in the table are sample values used in running this image segmentation architecture; these values are normalized to a [0,1] range. Also, the initial mean for the pixel's distribution is initialized to the starting pixel values from the first image frame. Furthermore, apart from initialization at the beginning of algorithm, the pixel distribution parameters are also re-initialized when no distribution matches the current

pixel; the lowest weighted distribution is replaced with a new distribution with the aforementioned initial mean, variance, and distribution weight.

The pixel match evaluation is vital for discerning which underlining distribution best matches the current pixel sample or if none of the distributions match at all. As opposed to using a costly expectation-maximum algorithm based approach for clustering and updating the distributions, a simple on-line K-means approximation is used instead. Using this simpler method, pixels are checked against each of the K distributions using the Mahalanobis distance metric, which is given as

$$|X_t - \mu_{k,t}| \leq \beta\sigma_{k,t}, \tag{2.17}$$

where β is the distribution match threshold defined in Table 2.2, $\mu_{k,t}$ and $\sigma_{k,t}$ are the parameters of the k-th Gaussian distribution at time t. The distribution that best represents the current pixel will have the lowest deviation from the mean of the distribution, and also be within the distribution match threshold.

When the best matching distribution for a pixel is within the distribution threshold, then the distribution parameters are updated as

$$\begin{aligned} \mu_t &= (1-\rho)\mu_{t-1} + \rho X_t \\ \sigma_t^2 &= (1-\rho)^2\sigma_{t-1}^2 + \rho(X_t - \mu_t)^2 \\ \rho &= \alpha \cdot \eta(X_t|\mu_t, \sigma_t^2), \end{aligned} \tag{2.18}$$

where ρ, scaled by the learning factor α expresses how much of a factor the new current pixel has on the updated mean and variance. Moreover, the weights for each pixel's distributions are updated depending if the current pixel matched or not. The update for the weight is given as

$$w_t = (1 - \alpha)w_{t-1} + \alpha M \tag{2.19}$$

where M is a 1-bit value that represent whether or not the distribution is the best match for the current pixel.

Finally, after the best matching distribution is computed and the parameters are updated, the background and foreground labels are assigned for all distributions for pixel classification. This is achieved by first sorting all k Gaussian distributions for each pixel by the ratio of their weight to their standard deviation (that is w_k/σ_k). Distributions that gain more evidence from pixels over the frames will gravitate towards the top of the ordering, and more transient background distribution tend towards the bottom of the ordering. From the ordering of the distributions, the set of background distribution is computed as the first B distribution from the expression

$$B = arg\, min_b(\sum_{k=1}^{b} w_k > b_{th}) \tag{2.20}$$

where b_{th} is the background distribution threshold, as listed in Table 2.2, that bounds how many distributions are identified as background based on the sum of the weights.

Furthermore, the label of the best matching distribution is then assigned as the classification label for the current pixel.

2.4.2 Architecture Using Stochastic Computation

The popular Stauffer and Grimson online image segmentation method was revised and implemented with stochastic logic for computational efficient unimodal background modeling and subtraction. As opposed to modeling each pixel with a mixture of Gaussian distributions, each pixel is instead modeled with just one distribution, which captures the variance and mean of the hypothesized background for that pixel. Thus, an implementation of the image segmentation method described earlier can be simplified into three main subblocks, as shown in the block diagram in Figure 2.14.

First, the current pixel is evaluated against the pre-existing distribution using the Mahalanobis distance metric. If the current pixel is within the threshold bound of standard deviations with respect to the mean, then it is considered a match, otherwise it is not. Depending on if the current pixel matches the distribution, the distribution weight is updated. If that weight falls below a background threshold, which is a bound on the minimum evidence in a distribution to be considered a background distribution, then the distribution is reinitialized by resetting the parameters. The final subblock then updates these parameters accordingly, where if the distribution is not being reinitialized, then the parameters are updated through update rules

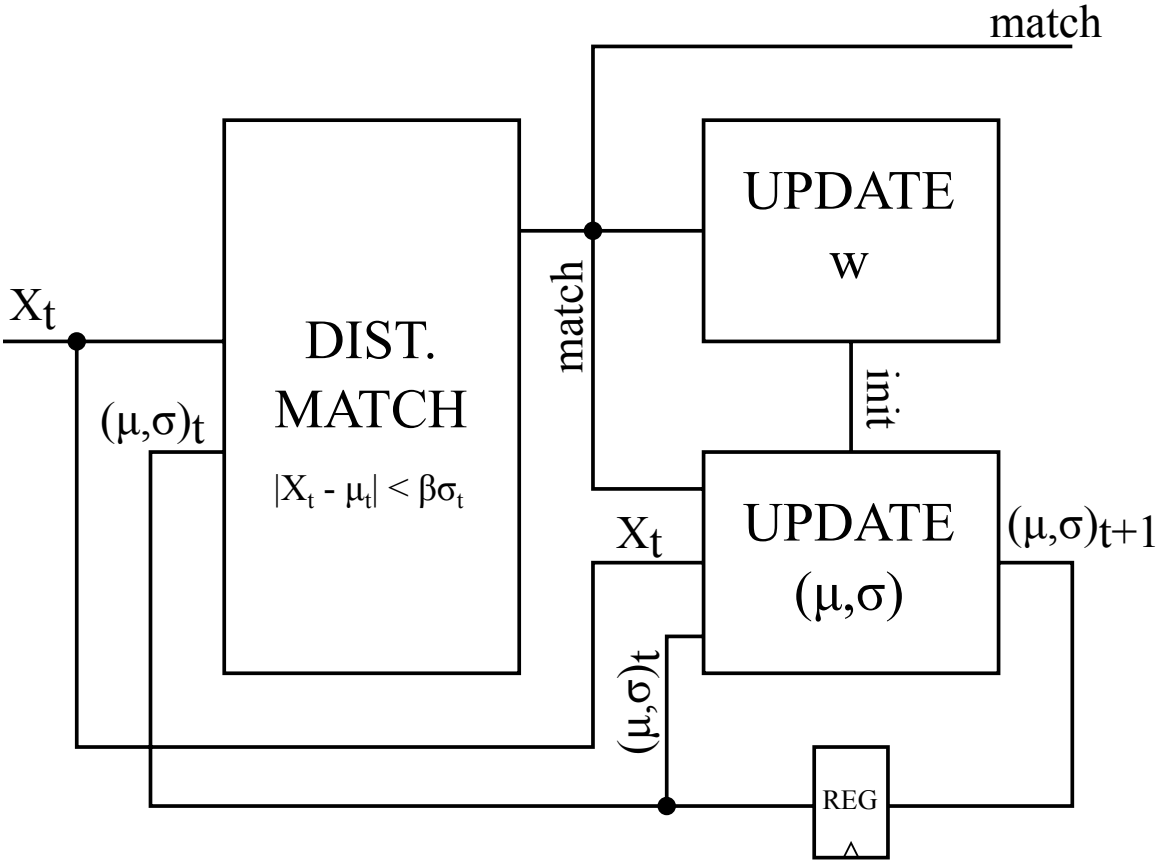


Figure 2.14: Block Diagram of the Unimodal Background Modeling Method.

based on a learning factor. These updated parameters — the mean, variance, and distribution weight – are used for the next processing cycle. Moreover, the pixel is classified as background if the distribution was considered a match from the distance metric, otherwise it is labeled as foreground.

The architecture for the unimodal background modeling and subtraction method

presented in Figure 2.14 computes the following expressions:

$$\begin{aligned} |X_t - \mu_t| &< \beta\sigma_t \\ \mu_t &= (1-\rho)\mu_{t-1} + \rho X_t \\ \sigma_t &= (1-\rho)\sigma_{t-1} + \rho|X_t - \mu_t| \\ w_t &= (1-/rho)w_{t-1} + \rho M_{t-1}. \end{aligned} \tag{2.21}$$

This includes an additional conditional operation for checking when to reinitialize the distribution parameters and the corresponding selector logic for updating the parameters. Furthermore, the total operations necessary for computing these expressions and executing the parameter reinitialization for the background modeling method include: multiplication, absolute difference, and addition operations; these operations can be mapped to basic logic gates in the stochastic domain.

Using stochastic computation primitives, this image segmentation architecture is implemented as the circuit shown in Figure 2.15. In this architecture, first the current pixel and the distribution parameters are encoded stochastically using stochastic number generators. Then, the distribution match, evaluated from the Mahalanobis distance metric, is computed using 4 basic logic gates; an XOR gate computes the absolute difference $|X_t - \mu_t|$, an AND gate computes the product $\beta\sigma$, and two NOR gates are used to generate the up and down signals for an up-down counter that measures whether or not the distribution matches the current pixel. The parameters μ_t, σ_t, and w_t are updated using a 2-by-1 multiplexer. The updated parameters are

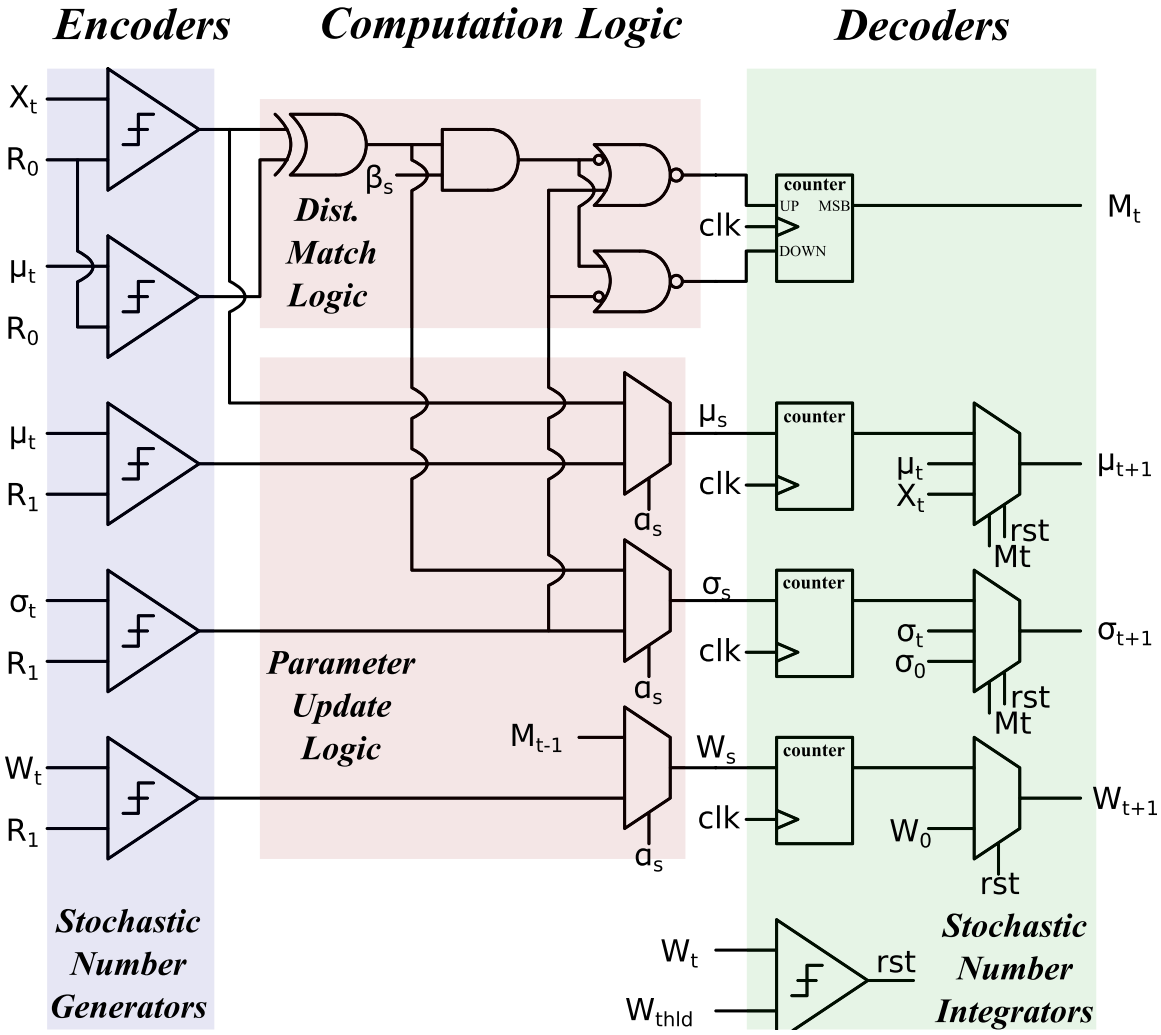


Figure 2.15: Image Segmentation Architecture with Stochastic Logic.

decoded back into binary values using counters.

The full macro synthesis of the complete architecture for N-bit stochastic computation precision is detailed in Table 2.3. Compared to a conventional architecture, which has to implement multi-bit binary multipliers, this stochastic computation based architecture uses minimal hardware resources with low energy cost, and also provides flexible precision with a trade-off of precision and processing time. The

Table 2.3: Image Segmentation Architecture Macro Synthesis.

Logic Primitive	Count
N-bit Comparator	6
N-bit Counter	4
N-bit 2-1 Multiplexer	5
2-input NOR	2
2-input AND	1
2-input XOR	1

(a) Input (b) Background (c) Foreground

Figure 2.16: Background Modeling and Subtraction with 8-bit Stochastic Computation Precision. (a) The input image from a video. (b) The background learned after multiple frames. (c) The foreground mask extracted from the current frame.

revised architecture based on probabilistic computing was implemented on a Kintex 7350-K410T FPGA board, as shown in Figure 2.11. It was then evaluated with a traffic video dataset for different stochastic computation precision. Figure 2.16 and 2.17 show the architecture learning the background and extracting a foreground mask at 8-bit (256 unary samples) and 12-bit (4096 samples) computation precision respectively. The results for the background and foreground mask in both cases are taken from the same image frame. Noticeable from the results, the architecture has a more accurate representation of the background of the traffic scene when computing with 4096 samples (12-bit stochastic computation precision) as opposed to 256

(a) Input (b) Background (c) Foreground

Figure 2.17: Background Modeling and Subtraction with 12-bit Stochastic Computation Precision. (a) The input image from a video. (b) The background learned after multiple frames. (c) The foreground mask extracted from the current frame.

samples. Nonetheless, the results of the foreground masks differ in the number of false-positive foreground pixels, which can be attributed to camera movement jitters during recording for the test dataset. By tuning the distribution match threshold β, the background threshold b_{th}, and the learning factor α, a more accurate foreground mask can be achieved in both cases. Thus, increasing the stochastic computation precision past 8 bits doesn't improve the quality of the foreground mask. By incorporating stochastic logic in this implementation of a unimodal background modeling and subtraction method, the trade-off of precision and energy for processing time can be exploited for more computational efficient segmentation.

2.5 Conclusion

Despite the traction in recent years of stochastic computing based architectures for applications in signal processing, machine learning, and more, still a major con-

cern of these alternative solutions, is how they compare to the conventional deterministic computing systems. Although advantages such as error-tolerant encoding, simpler computation implementation, and robust encoding/decoding for flexible precision have been exploited in stochastic computing, the additional energy cost for encoding and decoding in this stochastic domain is a significant constraint that limits its computation efficiency. In the case of solely implementing multipliers or edge detectors, an analysis has been presented by Rajit Manohar[51] that shows conventional deterministic approaches more favorable than stochastic approaches even in low-precision scenarios. Nonetheless, these advantages for stochastic computing still exist, and if properly exploited, can be implemented for more computational efficient systems.

2.5.1 Comparative Analysis

In order to gain more insight on the comparison of stochastic and deterministic computing, the image segmentation architecture presented in Section 2.4.2 was implemented in a conventional fixed-point representation with deterministic logic and then in a stochastic representation with stochastic logic. The two designs were synthesized with 16nm FinFET ARM standard cells. Both design were synthesized for different computation bit precision, which scales the full design of the fixed-point implementation, and only scales the encoders and decoders for the stochastic implementation. Table 2.4 shows the synthesis results comparing the encoders, logic,

Table 2.4: Synthesis Results for Stochastic and Deterministic Implementation of a Image Segmentation Methods.

	SC (Encoder)	SC (Logic)	SC (Decoder)	SC	Conv.
Area	$8.4\mu m^2$	$2.4\mu m^2$	$33.3\mu m^2$	$44.1\mu m^2$	$140.6\mu m^2$
VDD	0.8V	0.8V	0.8V	0.8V	0.8V
Frequency	2GHz	2GHz	2GHz	2GHz	0.5GHz
Power	3.2μW	1.1μW	48.8μW	53.1μW	41.8μW
Energy	1.6fJ	0.6fJ	24.4fJ	26.6fJ	83.3fJ
Precision	1-bit unary	1-bit unary	1-bit unary	1-bit unary	4-bit binary

and decoders of the stochastic implementation, and also the fixed-point implementation at a maximum of 4-bit computation precision. In the table, SC refers to the stochastic implementation of the image segmentation method, and Conv. is the conventional fixed-point implementation. Both implementation were synthesized at the same supply voltage, but since the complexity differs between the implementations, they were synthesized at different speeds. The stochastic implementation, which operates on 1-bit unary samples, can run up to 2GHz and has a total estimated area of $44.1\mu m^2$, while the conventional implementation can run at 0.5GHz and has a total estimated area of $140.6\mu m^2$. Although the stochastic implementation has roughly 3 times less area and energy cost than the conventional implementation, it has to be run for more clock cycles to achieve the same precision. However, just considering the energy cost from the stochastic logic, it would take over 138 clock cycles to use as much energy using the conventional implementation. Assuming minimal errors from encoding (see Section 2.2.2), than the energy cost from just the computation in the stochastic domain is more favorable than the cost of computing with the conventional implementation. Nonetheless, as seen earlier, the energy cost of encoding and decod-

ing for the stochastic domain is too costly, and the total energy cost to achieve the same precision would not be be worth it with the stochastic implementation.

This assessment is done for this architecture for 4, 6, 8, 10, and 12 bits of computation precision, and the effective energy per operation is measured for each case. In the case of stochastic, a rough estimate of 2^N clock cycles is used for N-bit precision (assuming pseudo-random numbers are used for exact encoding and there is no correlation across streams). A plot of this result can be seen in Figure 2.18. Even though stochastic computing scales exponentially with the bit precision, computing in the stochastic domain for this image segmentation architecture provides an energy saving over the conventional implementation for up to 10bits precision without considering encoding and decoding cost. Nonetheless, the encoding and decoding energy cost makes the stochastic implementation even less attractive at higher bit precision. In order to fully capitalize on the strengths of stochastic computation, the encoding and decoding cost must be minimized.

2.5.2 Exploiting Stochastic Computing for ASIC

A typical architecture exploiting stochastic computation, will have stochastic number generators for encoders, some logic that does computation, and then counters for decoding the results. Most gains realized in this domain of computing is realized from the simplification of the complex operations as simple logic gates. Nonetheless, these gains may be overshadowed by the significant energy cost from encoding and

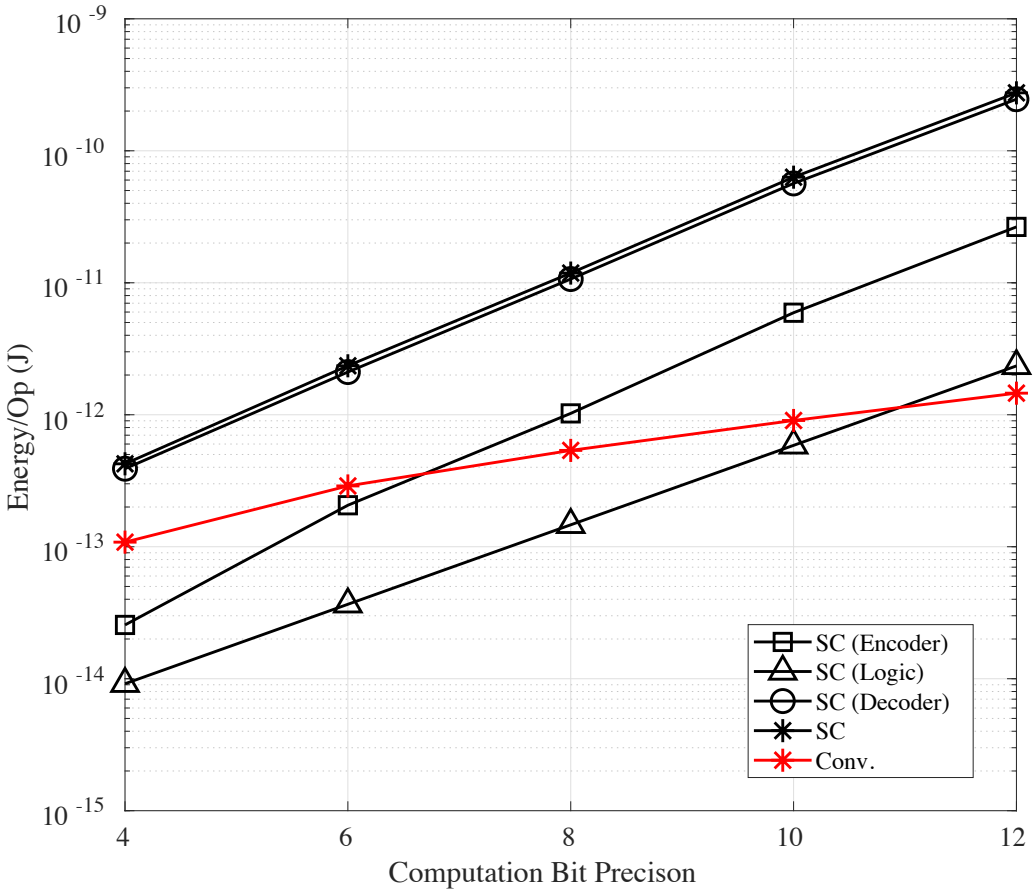


Figure 2.18: Plot of Energy Cost per Operation Comparison between Stochastic and Deterministic Implementation. An image segmentation method is implemented and synthesized using stochastic logic and conventional deterministic logic for different computation bit precision. The energy cost per operation is measured and plotted for both implementation across different precision.

decoding. In custom application specific integrated circuit (ASIC) designs, stochastic computation can be leveraged for simplifying the underlining circuit implementation of the computation, while exploiting mixed-signal representation and circuit techniques for mitigating encoding and decoding costs.

CHAPTER 2. STOCHASTIC REPRESENTATION FOR COMPUTATIONAL EFFICIENCY

As opposed to full digital representation for stochastic computing systems, as seen in the comparative analysis in Section 2.5.1, alternative modalities of representation that capitalize on the inherent physical structures can be exploited for computational efficiency. Work by Figliolia et al.,[52] show how the random telegraph noise sensed on a single transistor can be used a source for a random number generator, which could be used in a stochastic number generator. Additionally, computing implicitly as charge using switch-capacitor, or current using memristors or non-volatile memory, may allow for more computational efficient systems with less costly decoders.

Furthermore, since stochastic computation is inherently error-tolerant due to it's probabilistic encoding, voltage scaling can be leveraged for computing efficiently. The encoders and logic circuits can operate on a low voltage domain, which may be more susceptible to noise, while the decoder is run on a standard voltage domain to ensure the decoded result isn't perturbed by the scaling factors. This multiple voltage domain can take advantage of low voltage for better energy efficiency, while still being resistant to scaling parasitic factors.

Chapter 3

Mixed-Signal Architectures for VVM

As one of the most ubiquitous computation in machine learning and signal processing, vector-vector multiplications (VVM) form the basis of several vital algorithms, such as neural networks, k-means clustering, support vector machines (SVM), and linear vector quantization (LVQ). With the advent of big-data, the computational intensity of these algorithms have grown enormously, which in turn has drastically increased the energy cost of conventional processing paradigms. Even with the technological advances in microelectronics envisioned through Moore's law,[1] the scaling power constraints have posed challenges for data processing centers and computers at the scale.[11] Additionally, with the emergence of mobile computing platforms constrained by power and bandwidth for distributed computing, the necessity for

more energy efficient scalable local processing has become more important. Unconventional compute-in-memory architectures such as the analog winner takes all associative memory,[53] charge-injection device (CID) processor,[3,5,6] and analog-array processing[2] have been proposed as alternatives.

In this dissertation work, mixed-signal architectures, that exploit charge-based computing, are explored as energy efficient alternatives to conventional digital signal processors (DSPs) for vector-vector multiplications. These architectures capitalize on a unary representation for computing weighted sums in the analog domain as charge on a programmable capacitor array at the thermal noise limits. In order to convert this charge back into the digital domain, a first-order sigma-delta ($\Sigma\Delta$) modulator is used as an analog-to-digital converter (ADC). These architectures were designed and validated through 5 different test chips in submicron CMOS processes. Through these design iterations, different programmable capacitor array design are explored, and the $\Sigma\Delta$ ADC was optimized. Additionally, a stochastic unary representation is explored for minimizing area and energy cost of the computation. Furthermore, the measured results from these chips are presented, and some applications of these chips for image and signal processing are detailed.

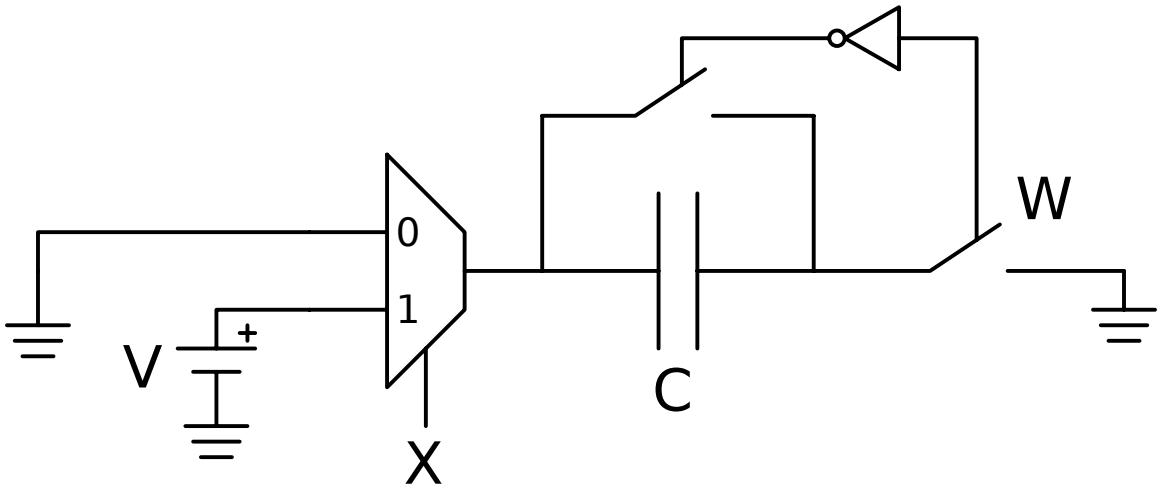


Figure 3.1: Mixed-Signal 1-bit Unsigned Multiplier for Unsigned Products.

3.1 Charge-Based Computing

In the analog domain, the product of two inputs can be computed as the total charge in a capacitor by programming the capacitance and voltage accordingly. By scaling the capacitance and the voltage to the thermal noise limit, this computation can be done efficiently with minimal energy cost. Exploiting this principle, a 1-bit mixed-signal multiplier can be constructed as the architecture shown in Figure 3.1. The output Y, which is stored as charge on the capacitor, is given as

$$Y = (WX)Q_u, \tag{3.1}$$

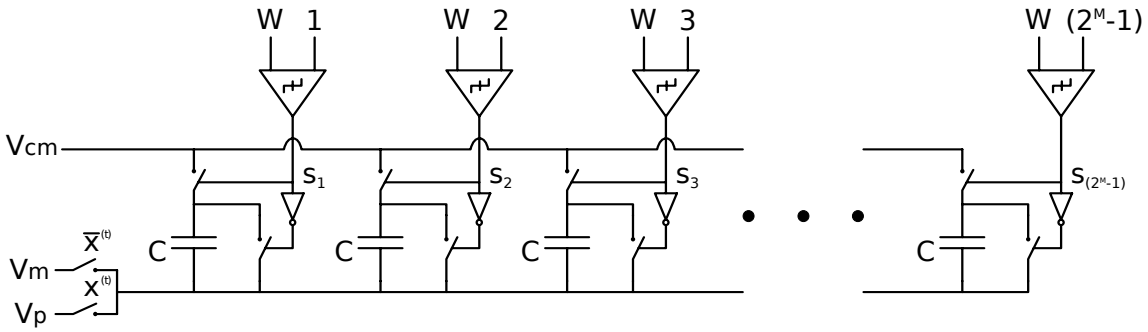


Figure 3.2: Mixed-Signal Scalar Multiplier

where a 1-bit weight W modulates the capacitance, and a 1-bit input X selects the voltage across the capacitor. The unit charge Q_u on this capacitor is given as

$$Q_u = CV, \tag{3.2}$$

for a capacitance C and voltage V.

This convention can be expanded to scalar products of multi-bit weights and inputs, W and X respectively, if the weight W encodes the total capacitance in a programmable capacitor array, and the input X temporally encodes the voltage potential across this capacitor array. The product of the scalar values is then represented as the aggregate of partial charge stored on the capacitor array over a set of sampling periods. The architecture for this multiplier can be seen in Figure 3.2. Using magnitude comparators, an M-bit weight W, which is given as

$$W = \sum_{i=0}^{2^M-1} s_i, \tag{3.3}$$

is decoded into a set of unary samples s_i that connect or disconnect unit capacitors in an array. As opposed to a binary-weighted encoding, the unary encoding of the weights with unit capacitors mitigates computation inaccuracies due to capacitance mismatch from fabrication variations.

An N-bit input X is encoded as a pulse-density modulated (PDM) value in time that select the voltage across the capacitor array. That is,

$$X = \sum_{t=0}^{2^N-1} x^{(t)}, \tag{3.4}$$

where the sum of the unary samples $x^{(t)}$s represent the input X over 2^N time samples. Thus, the product of the M-bit weight W and the N-bit input X can be constructed as the total charge accumulated on the programmable capacitor over 2^N time samples. Mathematically, this can be seen as

$$\begin{aligned} Q &= \sum_{t=0}^{2^N-1} Q_t \\ &= \sum_{t=0}^{2^N-1} \Big(W x^{(t)} \Big) Q_u \\ &= \Big(WX \Big) Q_u, \end{aligned} \tag{3.5}$$

where $Q_t = (Wx^{(t)})Q_u$ is the partial charge at time sample t.

Moreover, the architecture can be generalized to computing vector-vector multiplications (VVMs) by constructing an array of programmable capacitors as shown in

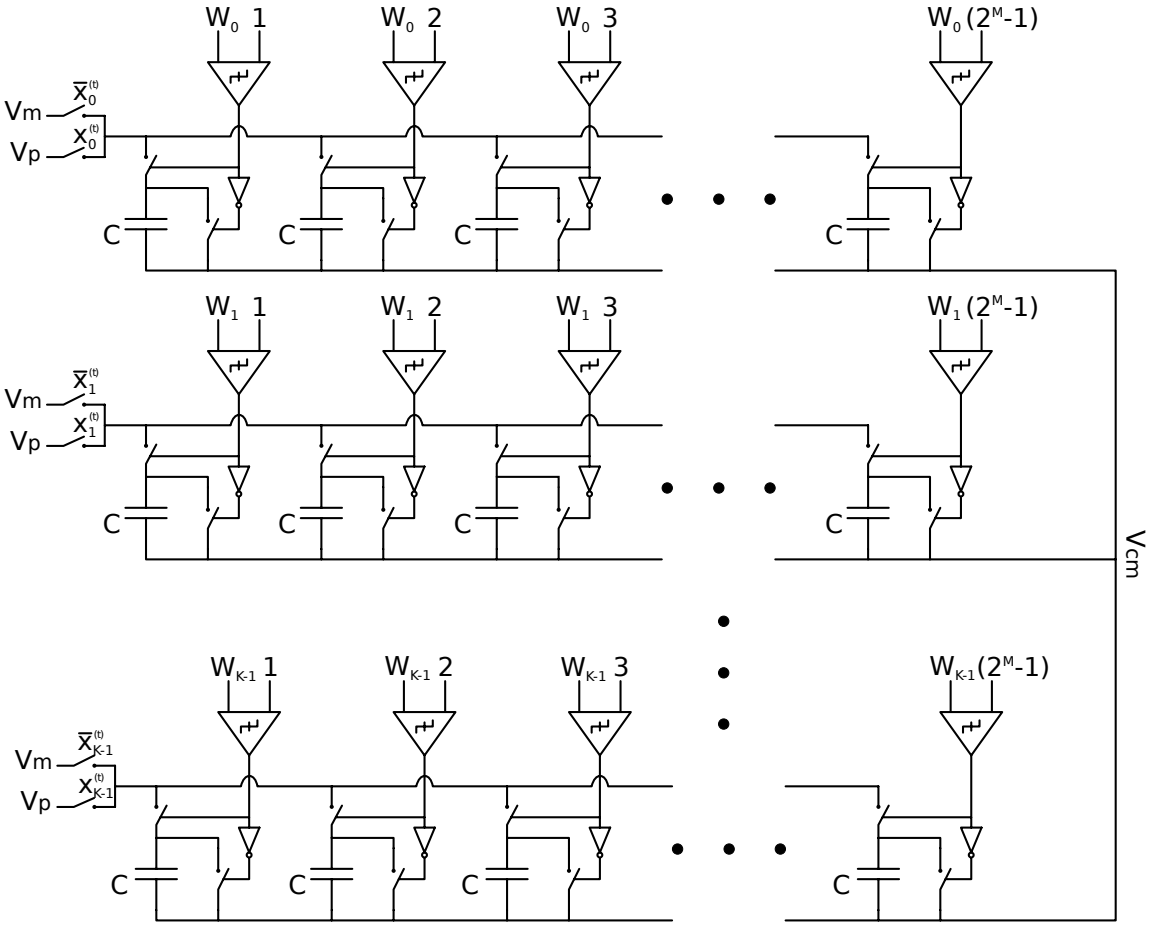


Figure 3.3: Mixed-Signal Vector-Vector Multiplier.

Figure 3.3. Similar to Equation 3.5, the total charge accumulated on the array can

be expressed as

$$\begin{aligned} Q &= \sum_{t=0}^{2^N-1} Q_t \\ &= \sum_{t=0}^{2^N-1} \left(\sum_{k=0}^{K-1} W_k x_k^{(t)} \right) Q_u \\ &= \left(\sum_{k=0}^{K-1} W_k \sum_{t=0}^{2^N-1} x_k^{(t)} \right) Q_u \\ &= \left(\sum_{k=0}^{K-1} W_k X_k \right) Q_u, \end{aligned} \tag{3.6}$$

where W_ks are M-bit weights, X_ks are N-bit inputs, and $x_k^{(t)}$ is the unary time sample of the input X_k.

3.1.1 Energy and Thermal Noise Limit

The total energy used by a supply to charge a capacitor a delta voltage V is given as

$$E = CV^2 \tag{3.7}$$

where half the energy is stored on the capacitor, and the other half is dissipated while charging. As this energy scales quadratically with voltage and linearly with capacitance, scaling the total charge by modulating the voltage and/or capacitance can be exploited for minimizing this energy cost. In relation to charge-based computing, these mixed-signal multipliers, as shown in Figure 3.1, 3.2, and 3.3, can be scaled to

the thermal noise (kTC) limit for energy efficiency.

Described as thermal agitations of charge carriers in electrical conductors, the kTC noise power spectral density (PSD) in a RC circuit is derived as

$$V_R^2(f) = 4kTR, \tag{3.8}$$

where $V_R^2(f)$ is the PSD per hertz of bandwidth of the kTC noise in a non-ideal resistor, k is the Boltzmann constant, T is temperature, and R is resistance. The thermal noise in the resistor is white and independent of the frequency. Furthermore, for a given bandwidth, the root mean square (RMS) of this voltage is

$$V_{RMS} = \sqrt{4kTR\Delta f}. \tag{3.9}$$

In a RC circuit composed of a resistor and capacitor, the noise is shaped by the low-pass filtering of the circuit. The effective noise bandwidth given as

$$\Delta f = \frac{1}{2\pi}\int_0^\infty |H(\omega)|^2 d\omega = \frac{1}{2\pi}\int_0^\infty \frac{1}{1+(\omega RC)^2} d\omega = \frac{1}{2\pi}\frac{\pi}{2RC} = \frac{1}{4RC} \tag{3.10}$$

Substituting in Equation 3.9, we have

$$V_{RMS} = \sqrt{4kTR\left(\frac{1}{4RC}\right)} = \sqrt{\frac{kT}{C}}. \tag{3.11}$$

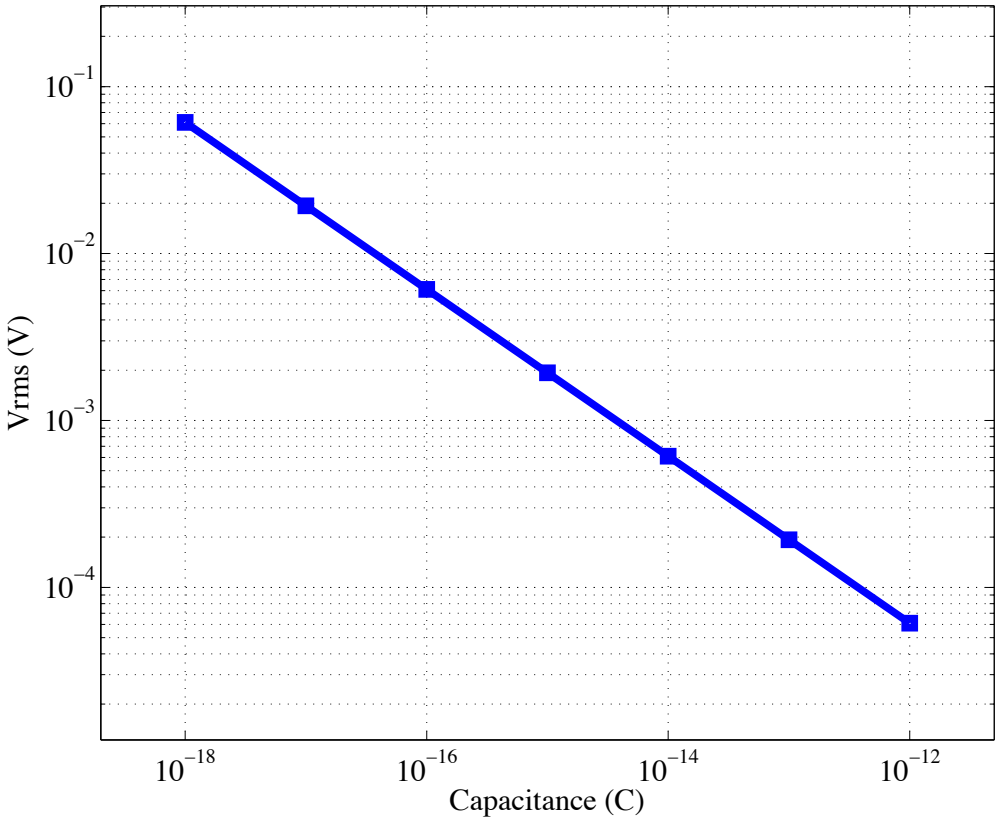


Figure 3.4: Plot of the Root Mean Square Thermal Noise in a Capacitor.

The RMS thermal noise voltage in the RC circuit is independent of the resistance, and only a function of the temperature T and capacitance C. Graphically, this voltage noise as a function of capacitance can be seen in the logarithmic-scale plot in Figure 3.4. Exploiting this curve while considering capacitance mismatch, charge leakage, and parasitic capacitance can be advantageously used to build energy efficient circuits.

3.2 Analog-to-Digital Conversion

For the mixed-signal vector-vector multiplier, an ADC is important for converting the inner product as charge into a digital value. Nyquist-based converters use a one-to-one correspondence between the input and output samples to convert across domains. However, these converters scale exponentially in size with bit precision, and are highly suceptible to linearity and accuracy issues due to mismatches in the analog components. Oversampling converters, such as $\Sigma\Delta$ modulators, exploit oversampling to obtain higher output precision with a trade-off of conversion speed. Fundamentally, this increased resolution is attained through spectrum noise-shaping that improves the signal-to-noise ratio (SNR) within the signal bandwidth.[54] $\Sigma\Delta$ modulators have been widely implemented in low power systems[55,56] for audio processing,[57,58] biomedical application,[59–61] and more. Specifically in this work, a simple first-order $\Sigma\Delta$ modulator is used for converting the analog output into the digital domain.

3.2.1 First-Order $\Sigma\Delta$ Modulator

A first-order $\Sigma\Delta$ modulator, which is comprised of low-resolution circuit blocks, can be used to approximate analog values over time as pulse-density modulated sequences. As shown in Figure 3.5, a $\Sigma\Delta$ modulator for converting continuous-time analog value sampled in timed $u(t)$ into a unary PDM signal $v(t)$, can be constructed with an integrator, a 1-bit ADC, and a 1-bit DAC.

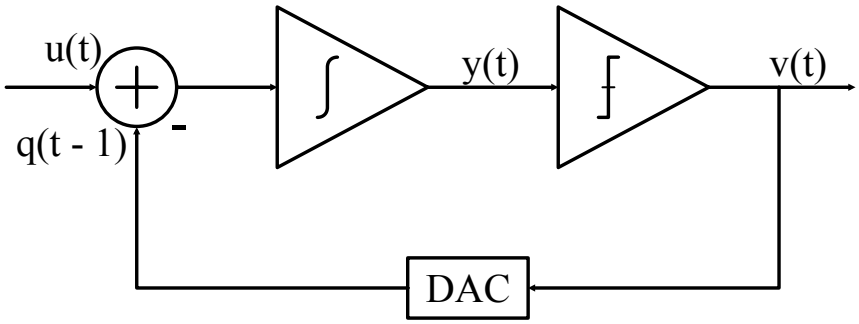


Figure 3.5: First-Order ΣΔ Modulator Block Diagram.

In the time domain, the output from the integrator $y(t)$ can be expressed as

$$y(t) = y(t-1) + u(t-1) - q(t-1), \tag{3.12}$$

where $u(t-1) - q(t-1)$ is the previous input to the integrator, and the $y(t-1)$ is the previous integrator output. The quantization error $Q_e(t)$, from the 1-bit ADC in Figure 3.5 is given as

$$Q_e(t) = v(t) - y(t). \tag{3.13}$$

Combing Equation 3.12 and 3.13 results in

$$v(t) = Q_e(t) + y(t-1) + u(t-1) - q(t-1) \tag{3.14}$$

Furthermore, for an ideal 1-bit DAC, the output $v(t)$ can be approximated as $q(t)$,

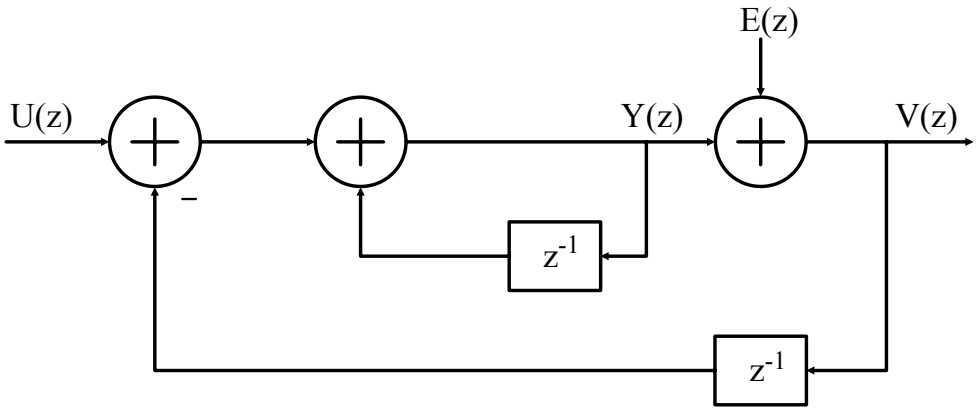


Figure 3.6: First-Order ΣΔ Modulator Z-Model Block Diagram.

which implies that

$$\begin{aligned} v(t) &= Q_e(t) + y(t-1) + u(t-1) - v(t-1) \\ &= u(t-1) + Q_e(t) - (v(t-1) - y(t-1)) \\ &= u(t-1) + Q_e(t) - Q_e(t-1). \end{aligned} \tag{3.15}$$

Thus, the output of the ΣΔ modulator $v(t)$ is equivalent to a quantized value of the input delayed by one time period $u(t-1)$ summed with the difference in quantization error $Q_e(t) - Q_e(t-1)$.

In the z-domain the first-order ΣΔ modulator shown in Figure 3.5 can be transformed into the block diagram shown in Figure 3.6. The output of the integrator can be expressed as

$$Y(z) = z^{-}1Y(z) + U(z) - z^{-1}V(z), \tag{3.16}$$

and the output of the $\Sigma\Delta$ modulator $V(z)$, can be derived as

$$\begin{aligned} V(z) &= Y(z) + E(z) \\ &= z^{-1}Y(z) + U(z) - z^{-1}V(z) + E(z) \\ &= U(z) + E(z) + z^{-1}(Y(z) - V(z)) \\ &= U(z) + (1 - z^{-1})E(z). \end{aligned} \tag{3.17}$$

Assuming that the DC value of $E(z)$ is finite, then for $z = 1$, the DC values of u and v follow from Equation 3.17 as $V(1) = U(1)$. Thus, high precision can be obtained for DC inputs.

Furthermore, the signal transfer function $STF(z)$ is 1 (unity), and the noise transfer function $NTF(z)$ is $1 - z^{-1}$. The in-band power of the quantization noise can be estimated from the squared magnitude of the NTF in the frequency domain for $z = e^{j2\pi f}$ as

$$|NTF(e^{j2\pi f})|^2 = (2\sin(\pi f))^2. \tag{3.18}$$

Thus, shown from the analysis above, the NTF is a high-pass filter that suppresses the quantization noise at and around DC, and amplifies the noise out of band in higher frequency. This noise-shaping improves the SNR of the ADC, which improves the output precision.

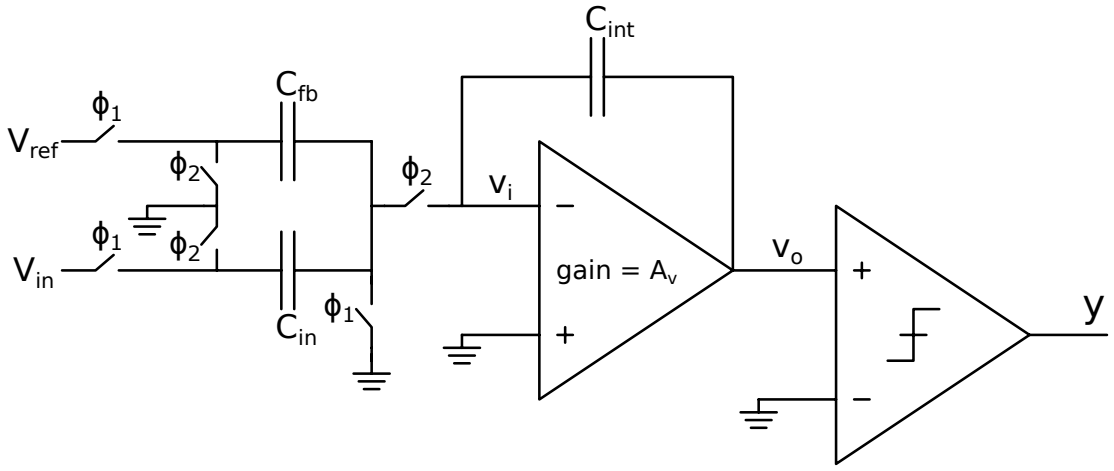


Figure 3.7: Single-Ended Switched-Capacitor ΣΔ Modulator Circuit.

3.2.2 Switched-Capacitor Implementation

To work in conjunction with the programmable capacitor array, a charge-based ΣΔ modulator that incorporates switched-capacitor circuits is implemented as the ADC. The partial charge, as described as Q_t in Equation 3.5, is decoded each time sample t into a 1-bit PDM signal. In a single-ended topology, this can be done using the circuit shown in Figure 3.7. Operating under two phases, ϕ_1 and ϕ_2, this circuit converts a charge Q_{in}, which is equivalent to $C_{in}V_{in}$, into a 1-bit signal y in time. During ϕ_1, the input charge Q_{in} is stored on the input capacitor. Then during ϕ_2, that charge is integrated onto the integrating capacitor, which causes a delta voltage on v_o. When that voltage v_o reaches or exceeds the common-mode threshold, then the signal y goes high with a logic level of '1'. This triggers a feedback charge, which is removed on the ϕ_1 phase through the feedback capacitor C_{fb}. This feedback charge is sized to the maximum possible input charge per cycle Q_{max}. Moreover,

this circuit implements all the subblocks for the ΣΔ modulator, which includes the integrator with the operational amplifier (op-amp) and the integrating capacitor, the quantizer with the voltage comparator, and the 1-bit DAC, which is implied as the voltage selector for the V_{ref} through the output y. The average value of y in time will represent the input charge $Q_{in} = C_{in}V_{in}$ relative to the maximum input charge Q_{max}.

As shown in Equation 3.6, the vector-vector multiplication operation done on the programmable capacitor array can be represented as

$$\left(\sum_{k=0}^{K-1} W_k X_k\right) Q_u = \sum_{t=1}^{N} Q_t \tag{3.19}$$

where N is the number of clock cycles for summing the partial charge Q_t. Using this switched-capacitor ΣΔ circuit, the partial charge Q_t in time is normalized to the maximum charge Q_{max} and converted as

$$\begin{aligned} \frac{1}{NQ_{max}} \sum_{t=1}^{N} Q_t &= \frac{1}{NQ_{max}} \left(\left(\sum_{t=1}^{N} y(t) \right) Q_{max} + Q_{res} \right) \\ &= \frac{1}{N} \sum_{t=1}^{N} y(t) + \frac{Q_{res}}{NQ_{max}}, \end{aligned} \tag{3.20}$$

where Q_{res} is the residual charge still stored on the integrating capacitor after N cycles of converting, and $y(t)$ is the sampled output of the comparator that controls when a feedback charge equivalent to Q_{max} is removed from the integrating capacitor.

Thus, the normalized weighted sum is approximated as

$$\frac{1}{K}\sum_{k=0}^{K-1} W_k X_k \approx \frac{1}{N}\sum_{t=1}^{N} y(t), \tag{3.21}$$

where the weights W_ks and inputs X_ks are normalized to a range of $[0, 1]$, and the second term from Equation 3.20 tends to 0 for large number of clock cycles N. Thus, the vector-vector multiplication result can be converted as the PDM signal y from the $\Sigma\Delta$ circuit shown in Figure 3.7.

Moreover, a differential topology is adapted for this switched-capacitor ADC in order to minimize parasitic factors due to charge injection from the switches, capacitive feedthrough, and coupled noise on the input. For this implementation, the circuit shown in Figure 3.7 is revised into circuit shown in Figure 3.8. Similarly, this circuit operates just as the single-ended circuit, except it uses an additional set of capacitors for integrating the negated input and feedback charge symmetric to the common-mode voltage. This negated charge is translated into a voltage output vm_o, which is symmetric to the other voltage output from the integrator vp_o with respect to the common-mode voltage. Based on which voltage is greater, negative feedback is applied through feedback charge injected or removed from the corresponding integrating capacitor. The magnitude of this charge is sized to match the maximum input charge Q_{max}. Moreover, the comparator output, which tracks the sign of the charge integrated on the integrating capacitors, is an expression of the ratio of the maxi-

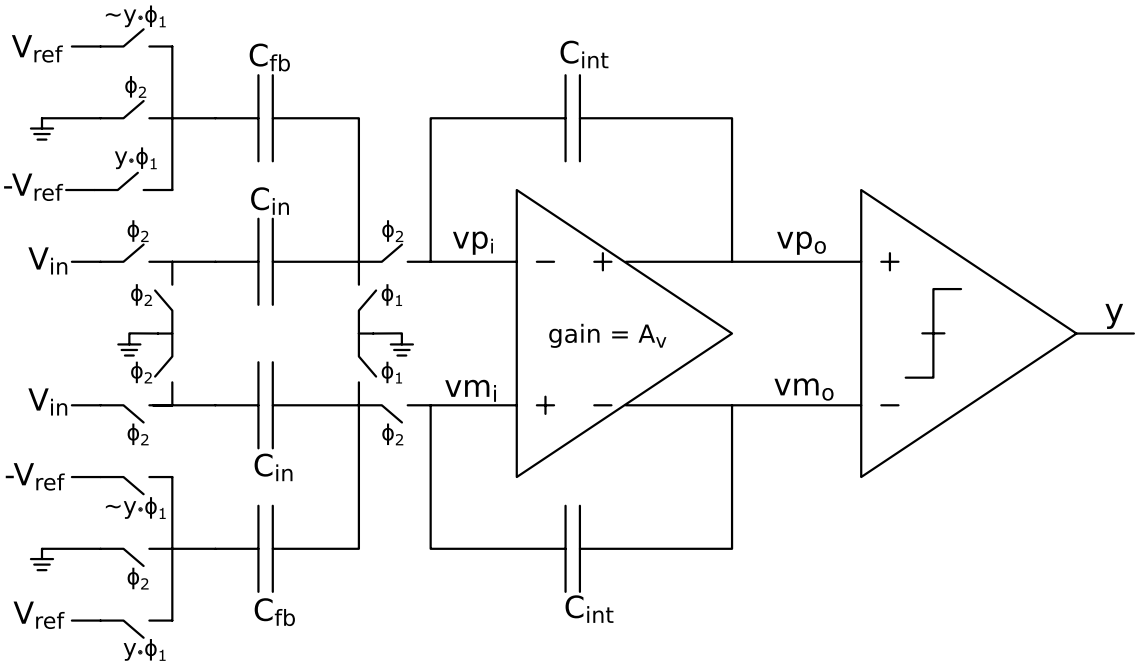


Figure 3.8: Fully-Differential Switched-Capacitor ΣΔ Modulator Circuit.

mum input charge being injected or removed to maintain no charge in the integrating capacitor.

Similar to Equation 3.20, the partial charge Q_t accumulated in time can be normalized and converted using the circuit in Figure 3.8 as

$$\begin{aligned}\frac{1}{NQ_{max}}\sum_{t=1}^{N} Q_t &= \frac{1}{NQ_{max}}\left(\left(\sum_{t=1}^{N} y(t)Q_{max} + (1-y(t))(-Q_{max})\right) + Q_{res}\right) \\ &= 2\left(\frac{1}{N}\sum_{t=1}^{N} y(t)\right) - 1 + \frac{Q_{res}}{NQ_{max}},\end{aligned} \tag{3.22}$$

where Q_{res} is the residual charge still stored on the integrating capacitor proportional to the voltage output vp_o after N cycles of converting, and $y(t)$ is the sampled output of the comparator that controls when a feedback charge equivalent to Q_{max} is injected

or removed from that integrating capacitor. Put simply, the normalized weighted sum is approximated as

$$\frac{1}{K}\sum_{k=0}^{K-1} W_k X_k \approx 2\left(\frac{1}{N}\sum_{t=1}^{N} y(t)\right) - 1, \tag{3.23}$$

where the weights W_ks and inputs X_ks are normalized to a range of $[0, 1]$, and the third term from Equation 3.22 tends to 0 for large number of clock cycles N. Thus, by summing and scaling the sampled outputs of the comparator $y(t)$ in time, the vector-vector multiplication can be converted from charge into a digital value. This summing and scaling post-processing can be done simply with a counter, and bit shifting the count output. In the following sections, the subblocks of the switched-capacitor $\Sigma\Delta$ modulator circuit are detailed.

3.2.2.1 Integrator

The integrator for the $\Sigma\Delta$ circuit, detailed earlier, uses a fully-differential op-amp for transferring the input and feedback charge to the integrating capacitor. In the single-ended case, as shown in Figure 3.7, the voltage output of the integrator v_o is given as

$$v_o = A_v(-v_i), \tag{3.24}$$

where A_v is the voltage gain. With a very large input resistance in the op-amp, current mainly flows through and from the input and feedback capacitors to the integrating

capacitors. Thus by Kirchoff's current law, we have

$$j\omega C_{fb}(V_{ref} - v_i) + j\omega C_{in}(V_{in} - v_i) = j\omega C_{int}(v_i - v_o). \tag{3.25}$$

The negative input in the op-amp v_i can be expressed as

$$v_i = \frac{C_{fb}V_{ref} + C_{in}V_{in} + C_{int}v_o}{C_{fb} + C_{in} + C_{int}}. \tag{3.26}$$

Combining Equation 3.24 and 3.26 gives

$$v_o = -A_v\left(\frac{C_{fb}V_{ref} + C_{in}V_{in} + C_{int}v_o}{C_{fb} + C_{in} + C_{int}}\right). \tag{3.27}$$

Simplifying the expression in Equation 3.27 results in

$$v_o = \left(\frac{C_{in}}{C_{fb} + C_{in} + C_{int}(1 + A_v)}V_{in}\right) + \left(\frac{C_{fb}}{C_{fb} + C_{in} + C_{int}(1 + A_v)}V_{ref}\right) \tag{3.28}$$

For large gain $A_v >> 1$, the output voltage approximates to

$$v_o = -\left(\frac{C_{in}}{C_{int}}V_{in} + \frac{C_{fb}}{C_{int}}V_{ref}\right). \tag{3.29}$$

Thus, with the two input terminals of the op-amp at virtual ground (the common-mode voltage), the total charge accumulated on the integrating capacitor $Q_{int}(t)$ at

time t can be expressed as

$$\begin{aligned} Q_{int}(t) &= C_{int}\left(\frac{C_{in}}{C_{int}}V_{in} + \frac{C_{fb}}{C_{int}}V_{ref}(t)\right) + Q_{int}(t-1) \\ &= C_{in}V_{in} + C_{fb}V_{ref}(t) + Q_{int}(t-1) \\ &= Q_{in} + Q_{fb}(t) + Q_{int}(t-1), \end{aligned} \tag{3.30}$$

which is the sum of the input charge Q_{in}, the current feedback charge $Q_{fb}(t)$, and the previous charge in the integrating capacitor $Q_{int}(t)$.

Low voltage circuits, inspired from the work of Dessouky et al. [57,62] and Mayr et al[61,63] are adapted for a low power implementation of the op-amp and analog periphery blocks in the integrator. The fully-differential op-amp circuit can be seen in Figure 3.9. This multi-stage op-amp is designed with minimal transistor stacking to facilitate lower voltage supply to minimize power cost. The first stage of the op-amp is a differential source-coupled pair (transistors $M1$ and $M2$) biased with current $Ibias$ from transistors $M7$ and $M8$. The common-mode feedback for this stage is attained through the cross-coupled transistors $M3$ - $M6$, where both outputs are fed back for positive and negative feedback to both outputs. The second stage is composed of a common-source amplifier, which provides another stage of amplification with the addition of a miller compensation capacitor and a nulling resistor to satisfy stability in the closed-loop design. Furthermore, the output of this amplifier is connected to a source follower to decouple the op-amp output from the compensation capacitor which limits the slew rate.

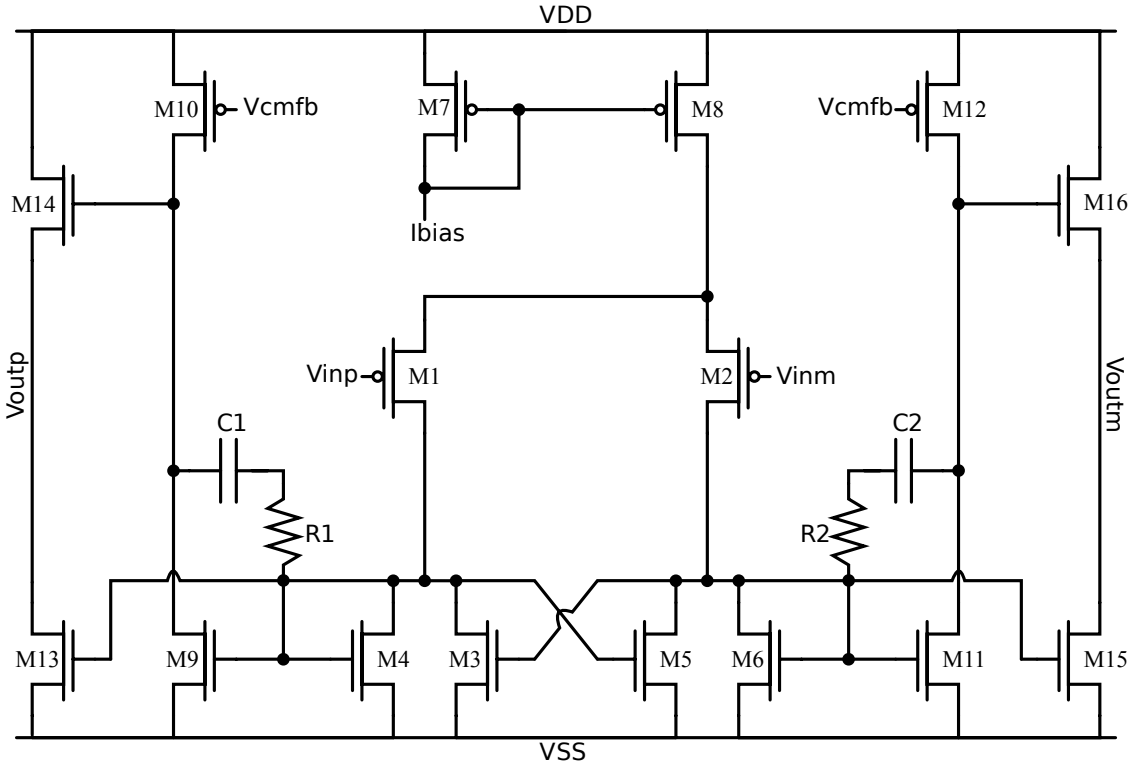


Figure 3.9: Fully-Differential Operational Amplifier Circuit.

The common-mode voltage of the second-stage of the op-amp is controlled by a dynamic switched-capacitor common-mode feedback circuit shown in Figure 3.10. This circuit operates under two phases, ϕ_1 and ϕ_2, to balance the op-amp outputs, $Voutp$ and $Voutm$, around the output common-mode voltage Vcm. During the ϕ_1 phase, a zero net charge is injected onto the $C1$ capacitors, and if the op-amp outputs, $Voutp$ and $Voutm$, are not symmetric around the common-mode voltage $Vcmfb$, then a nonzero charge is injected onto the $C2$ capacitors. Afterwards, in the ϕ_2 phase, charge is redistributed between the $C1$ and $C2$ capacitors, and the common-mode voltage changes accordingly.

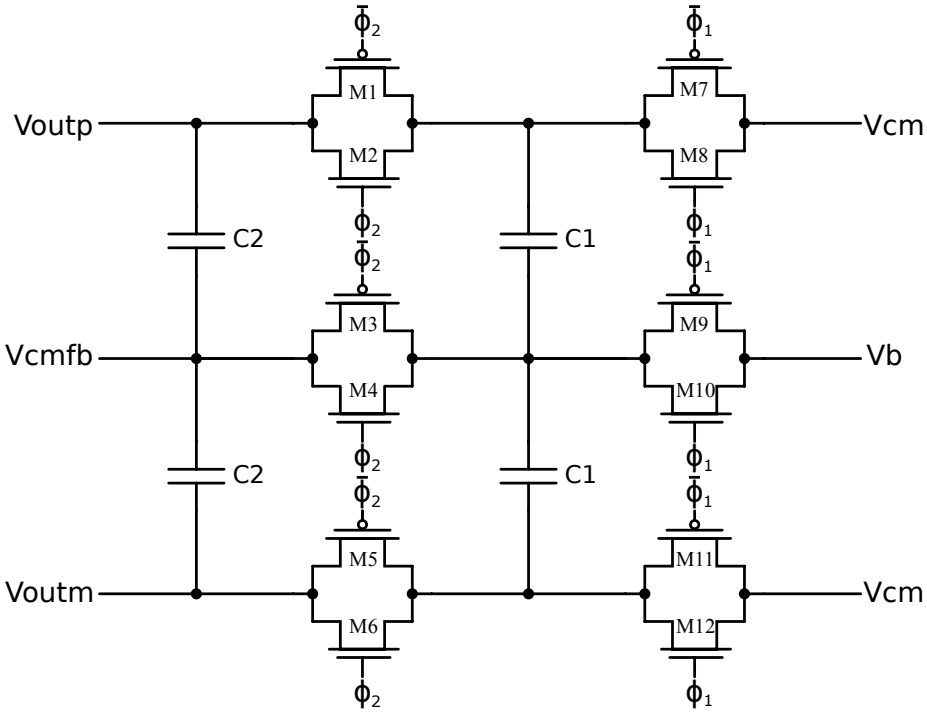


Figure 3.10: Switched-Capacitor Common-Mode Feedback Circuit.

The charge on the C_1 capacitors during the ϕ_1 phase is set from the external supplies Vcm and Vb as

$$Q_{\phi_1} = 2(Vb - Vcm)C_1, \tag{3.31}$$

and the charge on the C_1 capacitors during the ϕ_2 phase, after charge is redistributed, is

$$Q_{\phi_2} = (Voutp - Vcmfb)C_1 + (Voutm - Vcmfb)C_1. \tag{3.32}$$

The change in this charge from the ϕ_1 to the ϕ_2 phase, creates a proportional delta

voltage in the common-mode voltage $Vcmfb$ with a relationship of

$$\Delta Vcmfb \propto \frac{Q_{\phi_1} - Q_{\phi_2}}{2(C1 + C2)}. \tag{3.33}$$

By combining Equations 3.31, 3.32, and 3.33, the delta common-mode feedback voltage $\Delta Vcmfb$ can be expressed as

$$\Delta Vcmfb \propto \frac{C_1}{C1 + C2}\left(Vb - Vcmfb + \frac{(Voutp + Voutm)}{2} - Vcm\right). \tag{3.34}$$

Thus, the common-mode feedback of the op-amp will tend to the external voltage bias Vb, while the average of the op-amp outputs will tend towards the external output common-mode voltage Vcm. The convergence rate of these voltages are controlled by the $C1$ to $C2$ capacitance ratio.

3.2.2.2 Comparator and Latch

The differential analog outputs of the integrator is quantized into a 1-bit digital value for the feedback loop of the $\Sigma\Delta$ circuit through a voltage comparator. Since there are no critical design constraints for the quantizer because of the robustness of the ADC, a simple low-voltage comparator design is adapted; this circuit can be seen in Figure 3.11. In the comparator, the output is reset to a metastable state during the ϕ_1 phase, and then the output swings to the corresponding rail from the p-type differential stage with positive feedback from transistors $M3$ and $M4$.

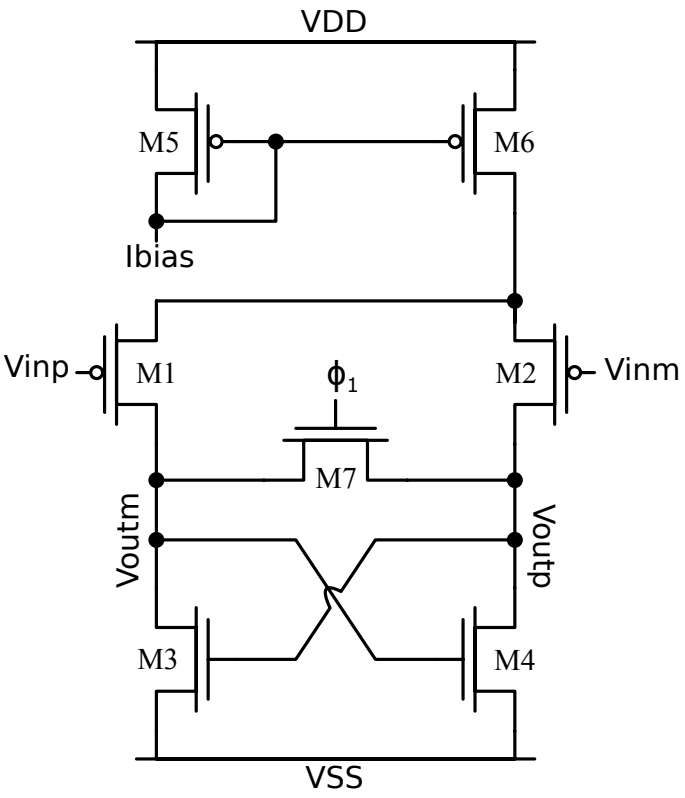


Figure 3.11: Low-Voltage Comparator Circuit.

Moreover, a low-voltage latch circuit, as shown in Figure 3.12, stores the output of the comparator during phase ϕ_2. The outputs Q and nQ are used as the selectors for the voltage of the feedback charge for the next clock cycle in the $\Sigma\Delta$ modulator.

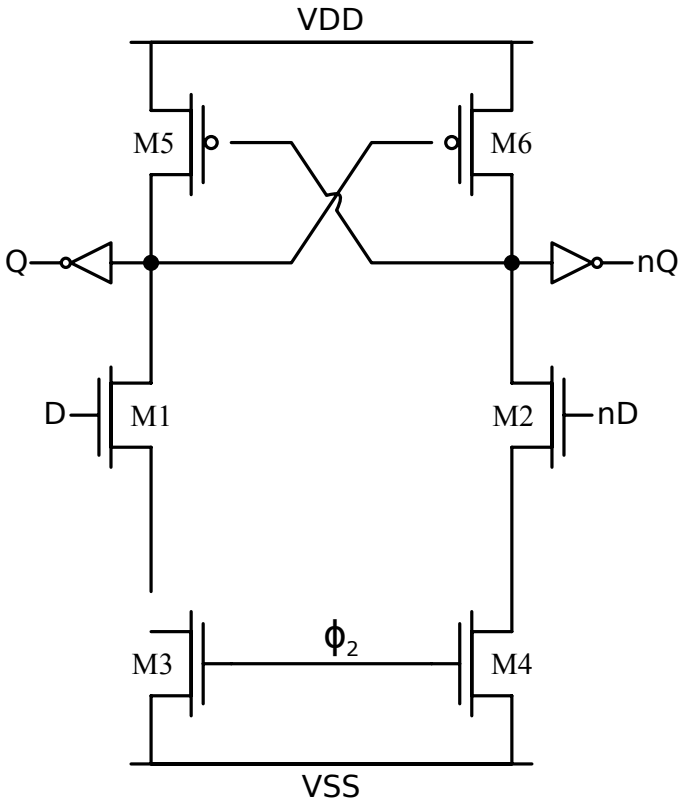


Figure 3.12: Low-Voltage Latch Circuit.

3.3 GF1 VVM: A 6-bit MAC Processor in 65nm CMOS

Based on the circuits presented in Section 3.1 and 3.2, a mixed-signal architecture for computing vector-vector multiplications is designed in the IBM 65nm CMOS process.[9] This design utilizes a unary weighted capacitor array for computing the products and sums as charge, and then converts the charge into a digital value using a switched-capacitor $\Sigma\Delta$ ADC.

3.3.1 Unary Weighted Capacitor Array

The programmable capacitor array used for computing the inner product is composed of a two-dimensional (2D) grid of unit capacitor. Each weight is encoded as an aggregate of unary bits in space that modulates the capacitance on the array, while each input is encoded, unary in time, as the voltage potential applied across a row of unit capacitors. The implicit product of the capacitance and voltage, results in a discrete amount of charge, which is summed in the array as the inner product over time. In the following subsections, the design methodology, physical layers (PHY), and simulation results for the capacitor array are presented.

3.3.1.1 Unit Capacitor

Proper design of the unit capacitors for the capacitor array is critical for achieving good accuracy and minimizing energy and area cost in implementation the VVM architecture. In order to align towards the design objectives, the capacitors are designed to maximize capacitance density, minimize inaccuracies due to process voltage and temperature (PVT) variation, and lessen the effects of parasitic nodes that cause non-linearity in the array.

The MOS capacitor, which is formed from a MOSFET transistor with its source, drain, and bulk tied together as one terminal and the gate as the other terminal, provides the best capacitance density per unit area because of its thin gate oxide separating the terminals. Nonetheless, because of its non-linear capacitance relationship

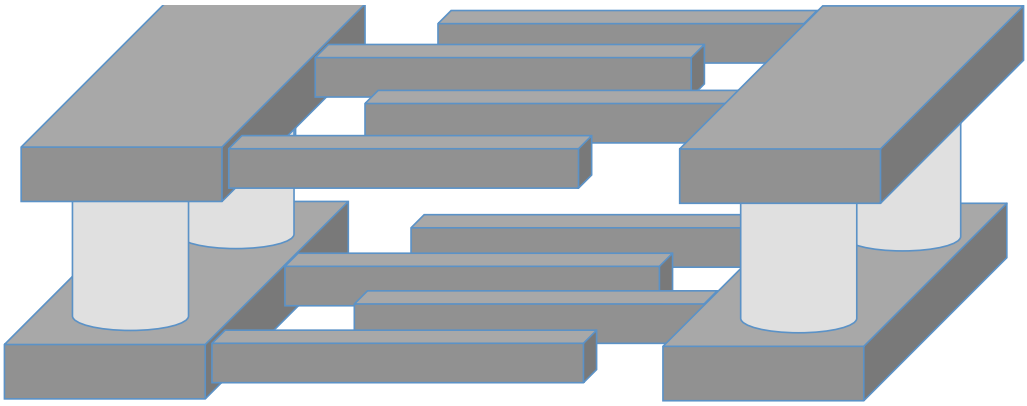


Figure 3.13: A Vertical Natural Capacitor (VNCAP).

to voltage variation, the MOS capacitor is ill-suited for the array design. Alternatively, metal-oxide-metal (MOM) capacitors, which form plates between sheets of metals, are viable options for this design because of its capacitance invariance to voltage changes. Specifically, a back end of line (BEOL) vertical natural capacitors (VNCAP), which is a MOM capacitor that achieve higher capacitance density through multiple metal layer stacking, is used. The VNCAP is composed of inter-digitated metal fingers over set of metal layers, where the total capacitance is derived from the side and the fringe capacitance between the fingers of the two terminals. Figure 3.13 shows a cross-sectional view of a VNCAP designed across 2 metal layers.

The unit capacitor for the array was sized to 5fF capacitance. The root mean square voltage noise from thermal agitations, which is detailed in Equation 3.11, for this capacitor is deduced as 910mV. Thus, for a 50mV range, the signal-to-noise ratio can guarantee 6-bits of precision for encoding in the array.

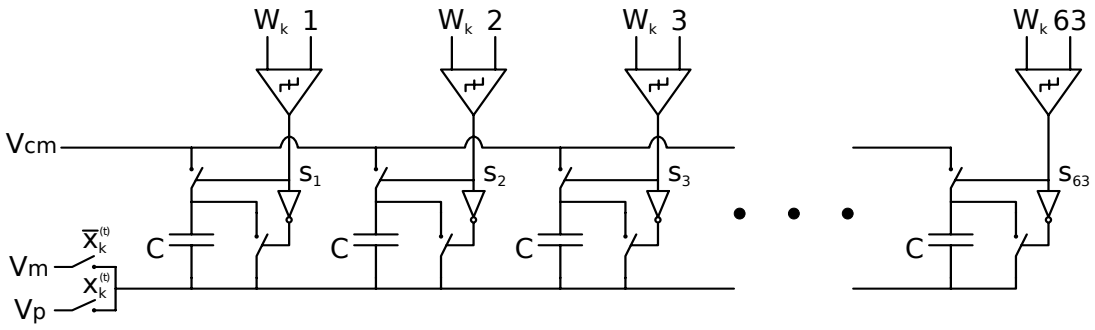


Figure 3.14: Row Slice of the Programmable Capacitor Array for GF1 VVM. Magnitude comparators are used to decode the binary weight into unary bits that disconnect/connect the capacitors for encoding the weight.

3.3.1.2 2D Capacitor Array

The 2D capacitor array is composed of 9 rows of 63 unit capacitors, where each row of the array encodes a 6-bit binary weight W_k as capacitance using magnitude comparators in the circuit shown in Figure 3.14. A unary encoding scheme is adapted for representing each weight to facilitate a monotonic relationship between the capacitance and weight value. Although a binary-weighted design would be more compact, the proposed unary capacitor array implements the additional decoder to minimize inaccuracies from capacitor mismatch by only using unit capacitors.

Each magnitude comparator is a subtractor circuit that computes the difference of the inputs and outputs the sign bit. For comparing two inputs, A and B, the

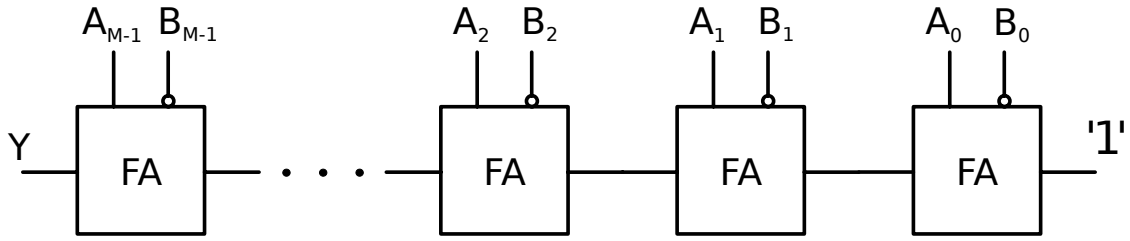


Figure 3.15: Subtractor Circuit for a Magnitude Comparator.

comparison output Y can be deduced as

$$\begin{aligned} A - B \geq 0 &\implies Y = 0 \\ A - B < 0 &\implies Y = 1. \end{aligned} \tag{3.35}$$

The architecture for this subtractor circuit can be seen in Figure 3.15. The inputs, A and B, are M-bit values that are compared using the subtractor, and the output Y is the most significant bit (MSB) of the difference. Since the output is only dependent on the carry bits, the sum bits are disregarded. The underlining building block of this magnitude comparator is based on a partial full adder (FA) circuit with just the carry out logic. This circuit is shown in Figure 3.16. Because one of the inputs, B, in the carry out circuit is always constant, the circuit shown in Figure 3.16 can be simplified to two cases — B = 0 and B = 1. The circuit for these cases can be seen in Figure 3.17. Thus, each magnitude comparator in the programmable capacitor design can be simplified to a cascode of NAND and NOR logic gates. This gives a 66% reduction in gate count, as the carry out circuit is decreased from 12 to 4 transistors.

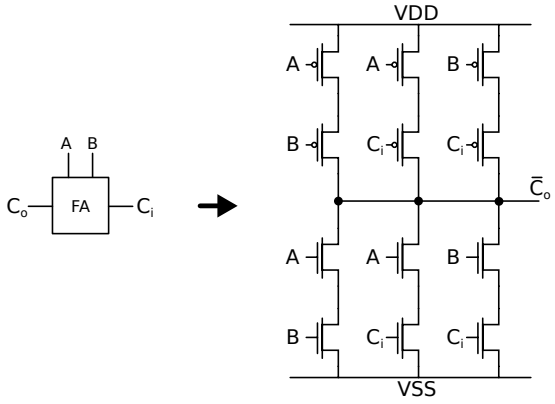
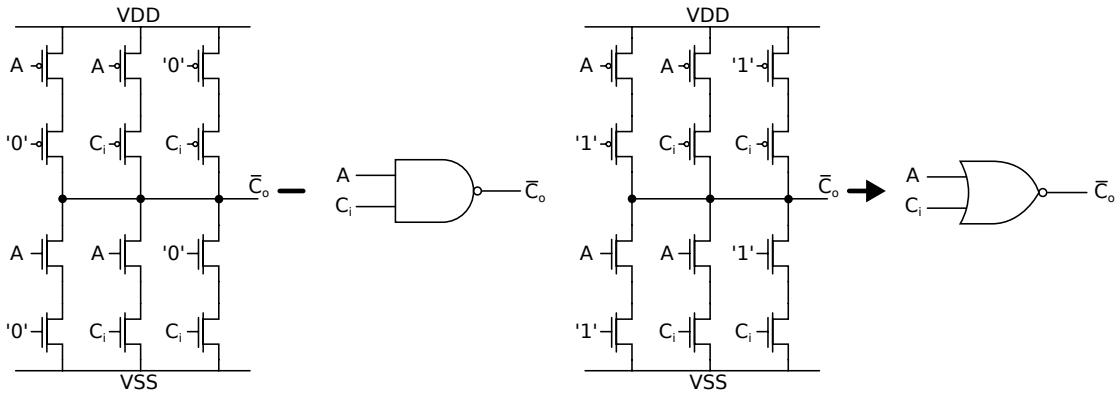


Figure 3.16: A Partial Full Adder Circuit with just the Carry Out Logic for the Magnitude Comparator.



(a) B = 0 (b) B = 1

Figure 3.17: Simplified Carry Out Circuit for the Magnitude Comparator.

A MATLAB script was written in order to generate the RTL description for any bit precision unary encoder. The code for this script, as shown below, infers the logic blocks for each magnitude comparator depending on which constant is connected.

```
%ENCODING SCHEME
%0 -> '0', 1 -> '1', 2 -> inverter, 3 -> NAND, and 4 -> NOR

nbits = 6;
num = fliplr(logical(dec2bin(0:1:(2^nbits)-1,nbits) == '1'));
blocks = zeros(2^nbits,nbits);
```

```
for i = 1 : (2^(nbits) - 1)
    for j = 1 : nbits
        %initial block assignment
        if  (mod(j,2) == 0 && num(i,j) == 0) ||...
            (mod(j,2) == 1 && num(i,j) == 1)
            blocks(i,j) = 4;
        else
            blocks(i,j) = 3;
        end

        %further simplification
        if j == 1
            if blocks(i,j) == 3
                blocks(i,j) = 2;
            elseif blocks(i,j) == 4
                blocks(i,j) = 0;
            end
        elseif j > 1
            if blocks(i,j-1) == 0 && blocks(i,j) == 3
                blocks(i,j) = 1;
            elseif blocks(i,j-1) == 1 && blocks(i,j) == 4
                blocks(i,j) = 0;
            end
        end
    end
end
```

Parameterized by the bit precision, *nbits*, the script was used to generate the structural composition for the 6-bit decoder for the weight.

The schematic and layout for the 2D capacitor array slice was then designed and tested; Figure 3.18 shows the layout construction for a generic row slice for this architecture. The full 2D capacitor array, as shown in Figure 3.3, was constructed with these row slices. The layout for this design is shown in Figure 3.19. As mentioned earlier, the full array contains 9 rows of 63 unit capacitors and the magnitude

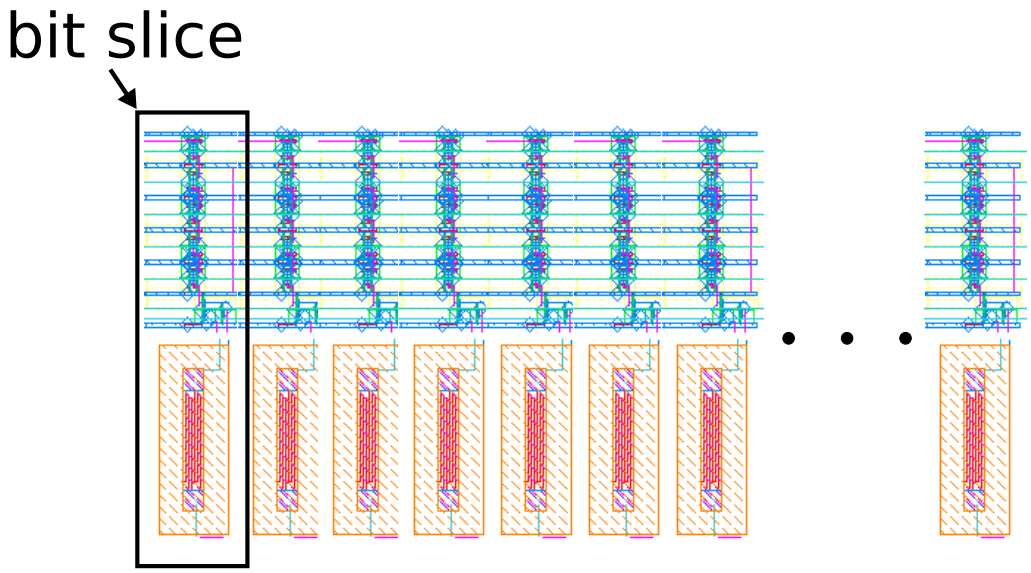


Figure 3.18: Layout of a Generic Row Slice for the Capacitor Array in a 65nm CMOS Process.

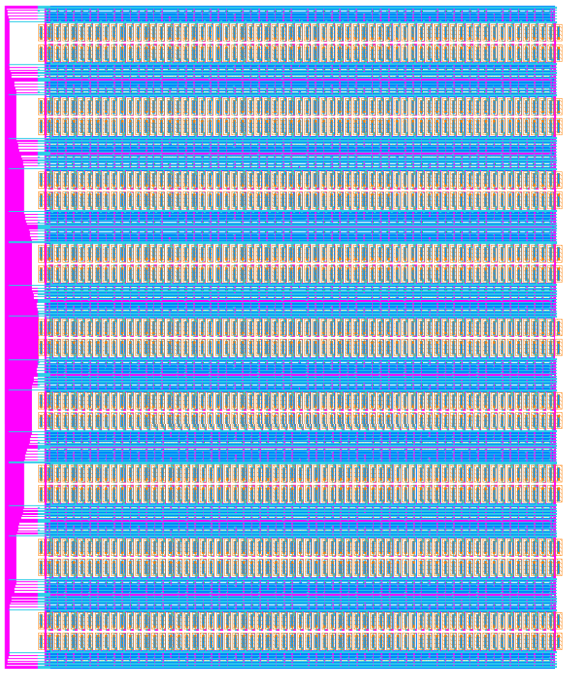


Figure 3.19: Layout for the 2D Capacitor Array in GF1 VVM.

comparators for decoding the binary weights.

Furthermore, the 2D capacitor array was simulated in order to analysis the transfer curve of this VVM architecture with respect to the weight values. In order to measure the transfer curve, the charge injected on the array is integrated onto another capacitor using an ideal integrator, and the delta voltage for the output is measured. Transfer curves plots for different input voltage bias values are shown in Figure 3.20. Specifically, this plot shows two transfer curves for input voltage bias values of 60mV and 240mV. The integrating capacitor was sized to 1.53pF, and the minimum LSB value measured on the output was a delta voltage of 0.74mV and 0.17mV for a input voltage bias of 240mv and 60mV respectively. The differential non-linearity (DNL) and integral non-linearity (INL) for both transfer curves don't exceed an LSB equivalence in the output voltage. Nonetheless, these simulation plots don't capture the capacitance mismatch that is expected from fabrication variations.

Moreover, the 2D capacitor array was also simulated in order to measure the energy cost for different weight (capacitance) values. As described in Equation 3.7, the energy cost of the VVM computation on the array scales linearly with the capacitance and quadratically with voltage. The validation of this is illustrated in the simulation results shown in Figure 3.21. This plot shows the energy measurements for different weights encoded on the capacitor array for computing VVMs with respect two different input voltage bias values of 60mV and 240mV. The total energy includes not just the energy used to charge the capacitors, but also the energy dissipated through

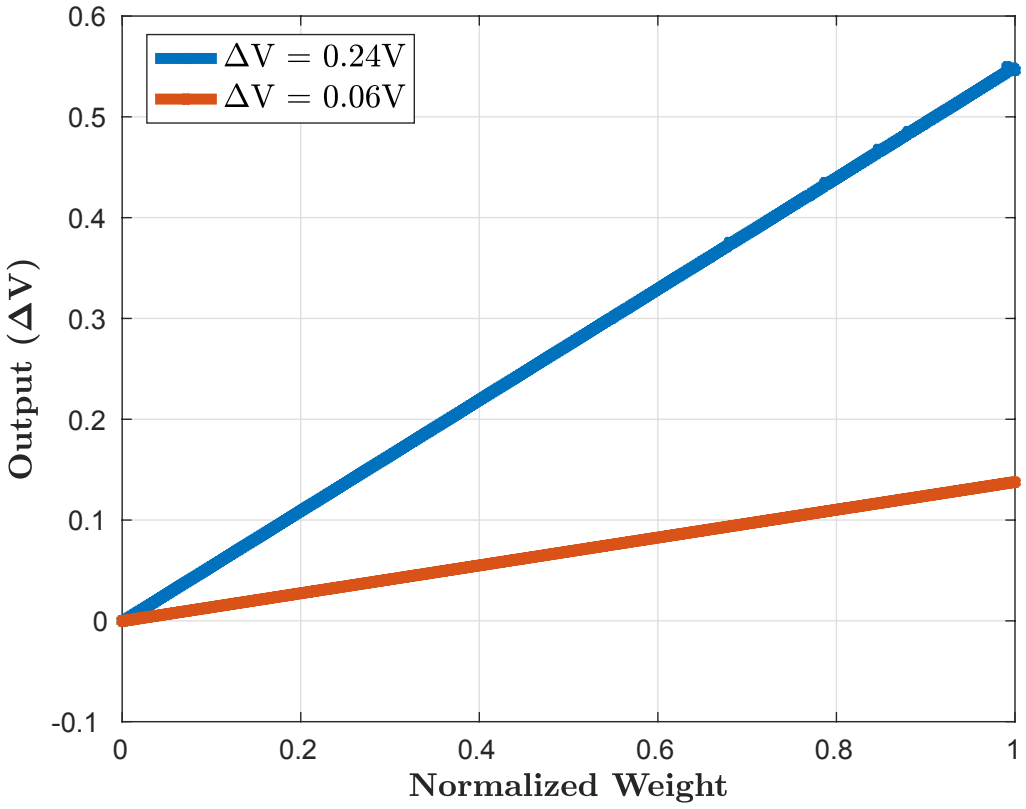


Figure 3.20: Transfer Curve Plots of the Weight (Capacitance) to Output (Voltage) for the VVM. The transfer curve is generated for two input voltage bias values of 60mV and 240mV. The weight value corresponds to the capacitance of the array normalized to the maximum capacitance.

the magnitude comparators for decoding the binary weightss.

To measure this energy, the current drawn from all the power supplies is integrated on another capacitor with a fixed known capacitance. Since the energy is given as

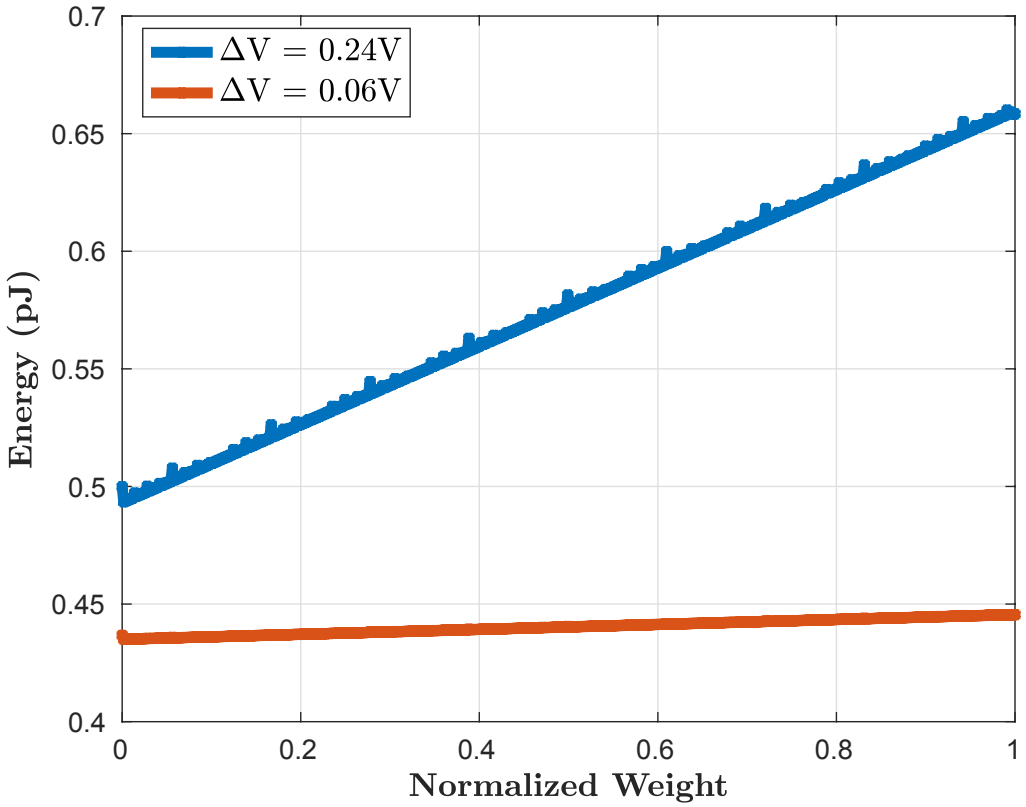


Figure 3.21: Plots of Energy Cost for the Capacitor Array with Different Weights and Input Voltage. The energy is measured with an input voltage bias of 60mV and 240mV for all discrete values of the weight encoded as capacitance on the array.

the integral of power respect to some time, the energy can be deduced as

$$\begin{aligned} E &= \int_{t0}^{t1} P(t)dt \\ &= \int_{t0}^{t1} I(t)V_{sup}dt \\ &= \int_{t0}^{t1} \frac{CV(t)}{dt}V_{sup}dt \\ &= CV_{sup}(V_{t1} - V_{t0}), \end{aligned} \tag{3.36}$$

where the V_{sup} is the fixed supply voltage, C is the capacitance of the capacitor used to integrate the current from the power supply, and $V_{t1} - V_{t0}$ is the delta voltage corresponding to the charge injected onto the measuring capacitor.

The simulation results from Figure 3.21 show that even with an input voltage bias of 240mV, the energy cost of the array to compute each cycle doesn't exceed a picojoule(pJ). It also shows the energy scale quadratically with the input voltage bias as predicted from the theory. Furthermore, the energy cost is relative to the weight values, and about 300fJ is dissipated on average from just the decoding logic.

3.3.2 VVM Core Architecture

The architecture for the vector-vector multiplier, which includes the programmable capacitor array and the analog-to-digital converter, can be seen in Figure 3.22. A differential topology is adapted for this design. The weight is encoded on two separate capacitor arrays for generating the corresponding VVM outputs as charge. These charges are then used in the first-order switched-capacitor $\Sigma\Delta$ modulator for generating the 1-bit PDM output. In the ADC, the feedback capacitors were sized to match the maximum input charge respective to a feedback voltage. Additionally, the integrating capacitors were sized such that the thermal noise in that capacitor did not exceed a LSB in the output voltage. Overall, a total of 9 analog biases, 7 voltages and 2 currents, are used in this circuit for the VVM, and they can be seen in Table 3.1.

Furthermore, the ADC for this VVM core was simulated prior to tape-out, and

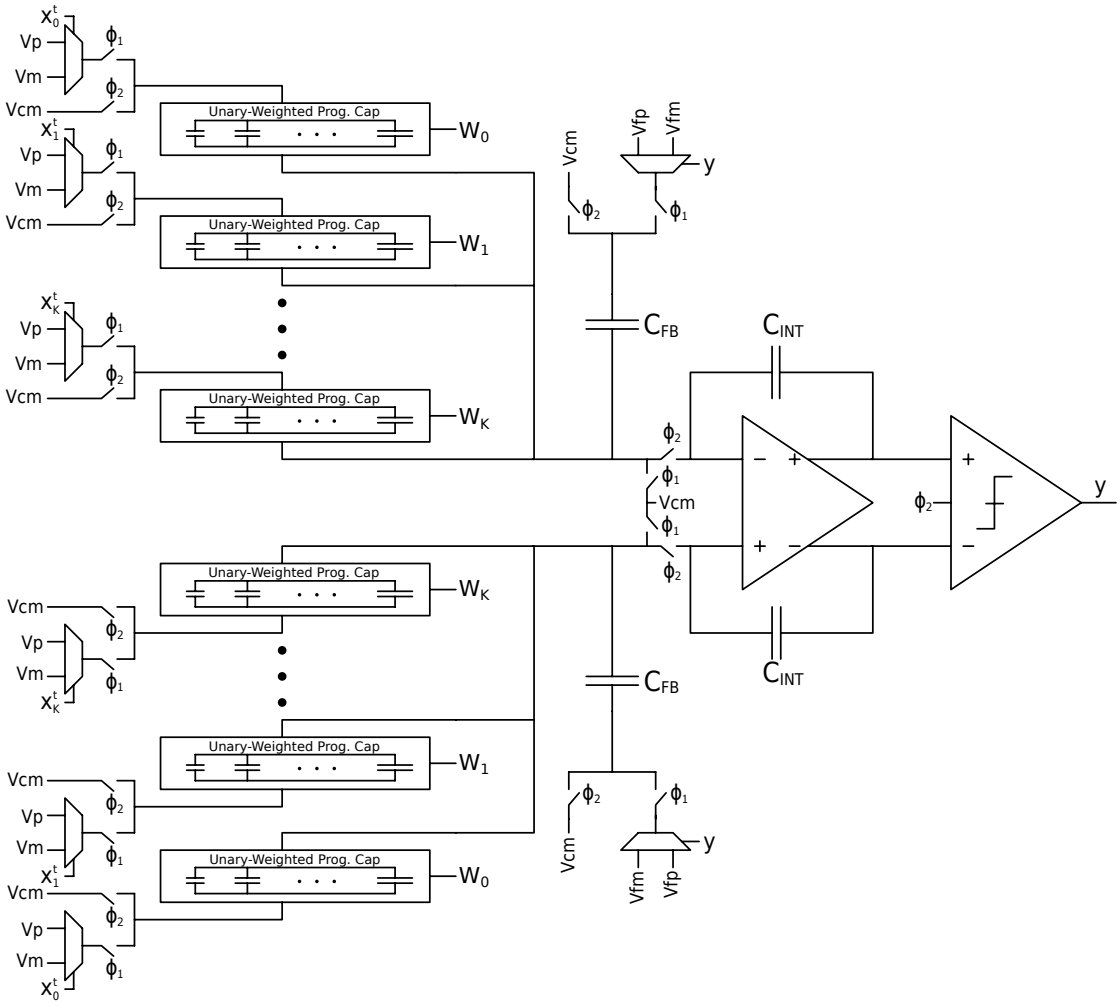


Figure 3.22: GF1 VVM Core Architecture.

the characteristics of the ADC can be seen in Table 3.2. Subsequently, the full mixed-signal VVM core was simulated for functional verification. The final layout for this core can be seen in Figure 3.23. The full mixed-signal core measures $348\mu m$ by $271\mu m$ ($0.09mm^2$) in area. Although the ADC accounts for less than 10% of the area core, it accounts for a significant portion of the power and energy cost because of the static power used in the ADC.

Table 3.1: Analog Biases for the VVM Core.

Name	Description	Nominal Value
Vcmi	Input common-mode voltage	0.3V
Vinm	Negative input voltage for the capacitor array	0.25V
Vfbp	Positive feedback voltage	0.41V
Vfbm	Negative feedback voltage	0.19V
Vcmo	Output common-mode voltage	0.5V
Vb	Voltage bias for the operational amplifier	0.75V
Iopa	Current bias for the operational amplifier	2.5μA
Icmp	Current bias for the voltage comparator	2μA

Table 3.2: Summary of the ADC Characteristics for GF1 VVM.

VDD	1.2V
Frequency	2.5MHz
SNR	40.32dB
Power	16.8μW

3.3.3 Test Chip

A test chip in the IBM 65nm CMOS process was constructed in order to test the mixed-signal VVM core shown in Figure 3.22. This core was interfaced to synthesized digital periphery circuits that provided local register files for the weights and the unary encoded inputs. The full chip layout, including custom built pad I/O circuitry cells, can be seen in Figure 3.24. In addition to the VVM core and the periphery circuitry, this figure also shows the annotated I/O port connections for programming and running the VVM core. The weights and the inputs are loaded to the chip through a serial interface using a shift register in order to reduce the number of required pins. This serial interface uses a periphery clock *clk* and enable signals ($en0\_i$ and $en1\_i$)

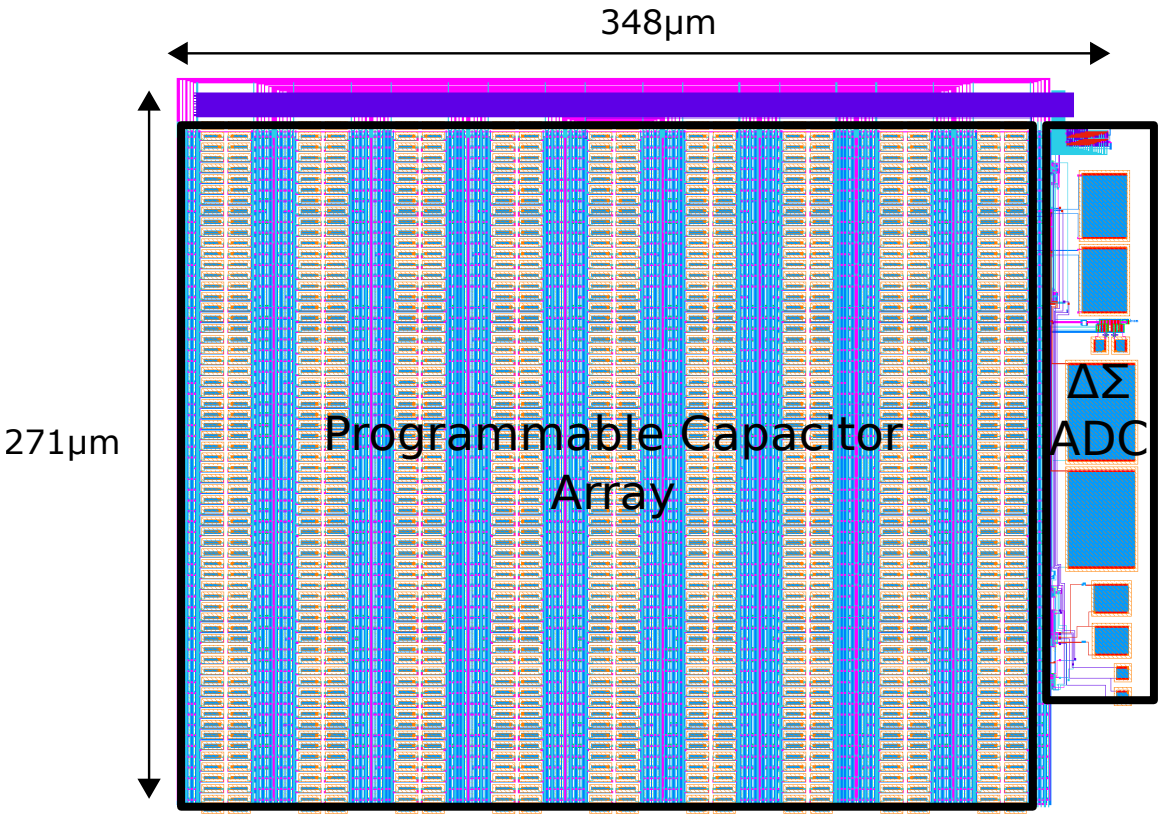


Figure 3.23: Layout of the GF1 VVM Core Architecture.

to push in the bits of the inputs and the weight through a 1-bit data signal d_i. The acknowledge signal ack_i is asserted whenever the input data has been fully loaded. The full set of digital I/O ports for the chip are described in Table 3.3.

The test chip was wirebonded to a 40 dual in-line package (DIP), and then connected to a breakout board that interfaces to a FPGA. Figure 3.25 shows the micrograph of the chip and the test board with the chip. A Spartan-3 FPGA board was used to program the VVM chip. On the board, a programmable clock divider is synthesized for generating the non-overlapping clocks, phi1 and phi2. In addition,

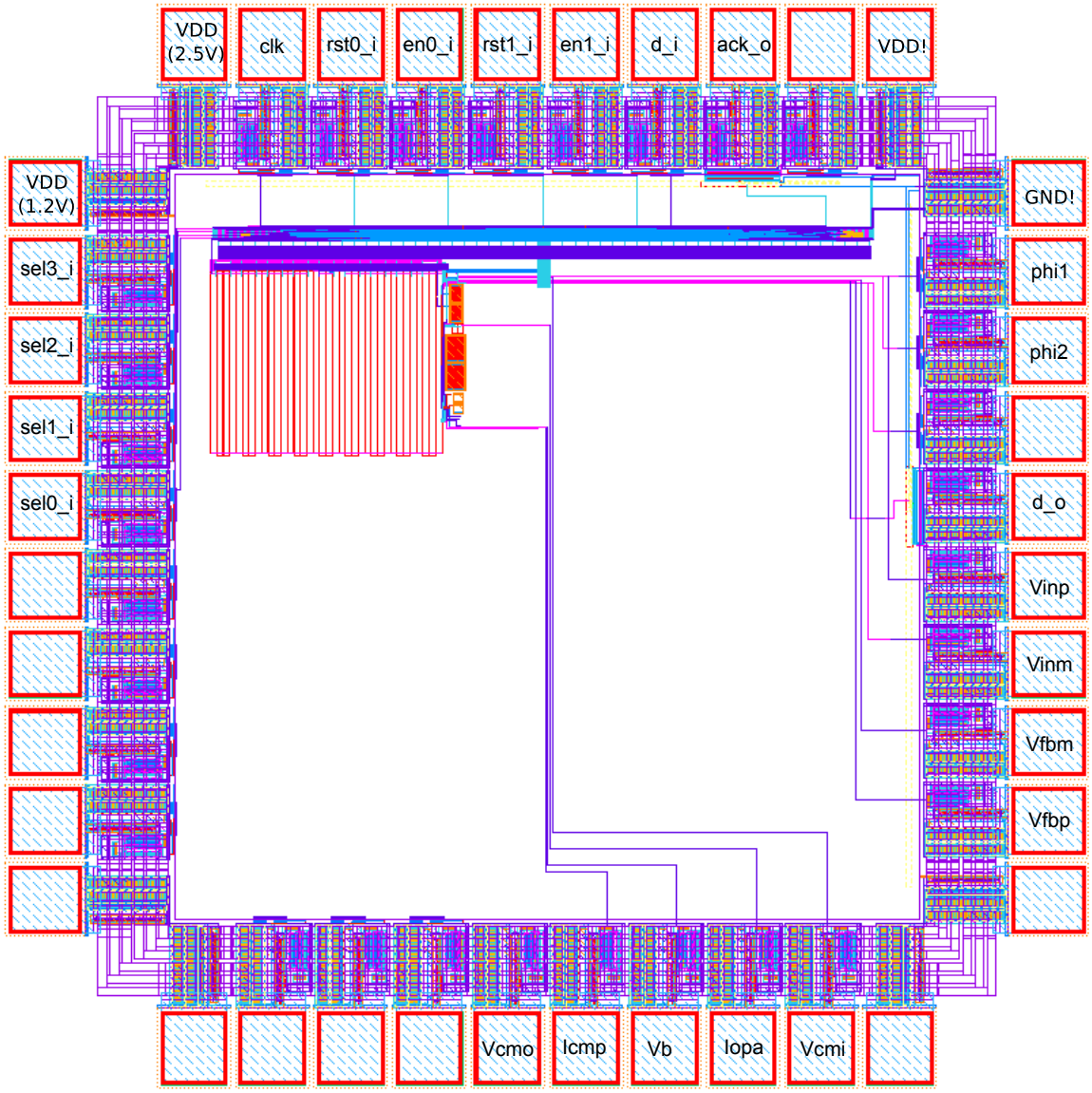


Figure 3.24: Layout of the GF1 VVM Chip.

the input encoders and output counter for decoding are synthesized for the VVM core in the chip. The board is programmed through a USB 2.0 port to a PC using a MATLAB interface. Specifically, the Opal Kelly FrontPanel software interface is used to for sending and receiving test data through the PC to the FPGA and then

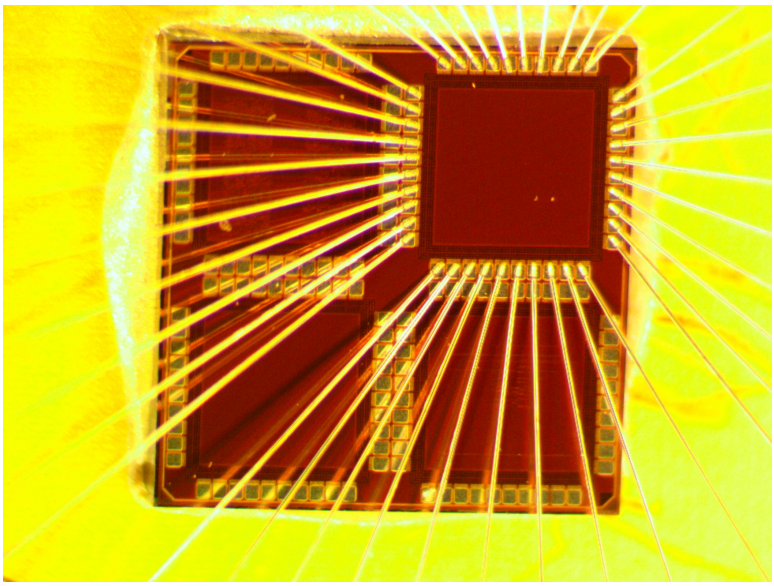
Table 3.3: Digital I/O Ports for the GF1 VVM Chip.

Name	Type	Width	Description
clk	Input	1	Clock signal for periphery circuitry.
phi1	Input	1	VVM core clock signal for first phase.
phi2	Input	1	VVM core clock signal for second phase.
sel_i	Input	4	Select signal for writing to the 9 weights.
rst0_i	Input	1	Reset signal for the weight register files.
rst1_i	Input	1	Reset signal for the input register files.
en0_i	Input	1	Enable signal used for loading the weights.
en1_i	Input	1	Enable signal used for loading the inputs.
d_i	Input	1	Data signal for both serial interfaces..
d_o	Output	1	VVM core output signal.
ack_o	Output	1	Acknowledge signal for the input serial interface..

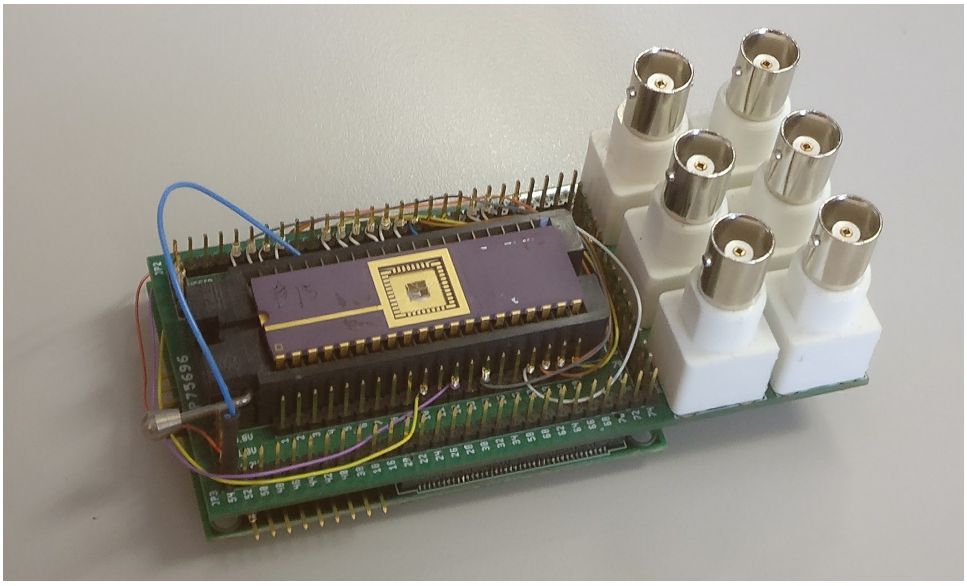
to the chip. Furthermore, the analog biases are programmed through a DAC board connected through the breakout board. Programmed through the FPGA, the DACs sets the values for the 9 analog biases with 12-bit precision and a voltage range of $[0, 1.2]$V.

The packaged VVM chip was tested to measure performance, efficiency, and variability. Operating at 2.5MHz on a 1.2V power supply, the core was successfully tested to compute 6-bit multiply-accumulates (MAC) at 5-bit output precision with a throughput of 350KOPs and an efficiency of 14.6GOP/W (284.4GOP/W for the capacitor array). The measured characteristics of the VVM core can be seen in Table 3.4.

Furthermore, with numerous of applications in signal processing and machine learning, this core was used for an image processing task of DeBayering an image. The results of this task, showing both the Bayer image and the DeBayer image, can



(a) Chip Micrograph



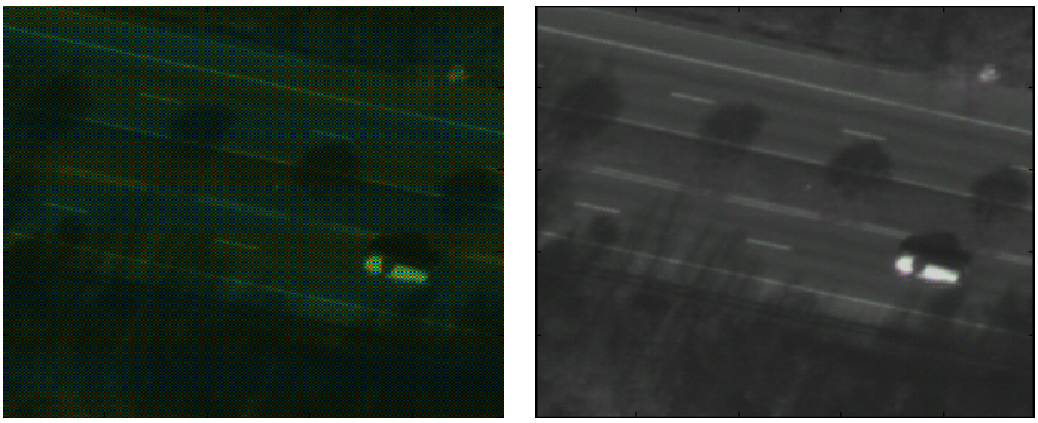
(b) Packaged Chip with Interface Board

Figure 3.25: GF1 VVM Chip Micrograph and Test Setup.

be seen in Figure 3.26.

Table 3.4: Measured Characteristics of the VVM Core.

Technology	65nm CMOS
Die Area	$0.09mm^2$
Operation	6-bit MAC
Precision	5-bit
Supply Voltage	1.2V
Frequency	2.5MHz
Throughput	350KOPs
Power Consumption	$24\mu W$
Energy/Op	68.27pJ (3.5pJ for the array)
Efficiency	14.8GOP/W (284.4GOP/W for the array)



(a) Bayer Image (b) DeBayer Image

Figure 3.26: DeBayering an Image Using the VVM Core.

3.4 GF2 VVM: A 6-bit MAC Processor in 55nm CMOS

Similar to the previous chip, GF1 VVM, the GF2 VVM chip exploits charge-based computing for energy efficient vector-vector multiplication operations. In this second iteration of the mixed-signal design, an alternative capacitor array design based on binary-weighting is explored in effort to cut area and energy cost. Using the

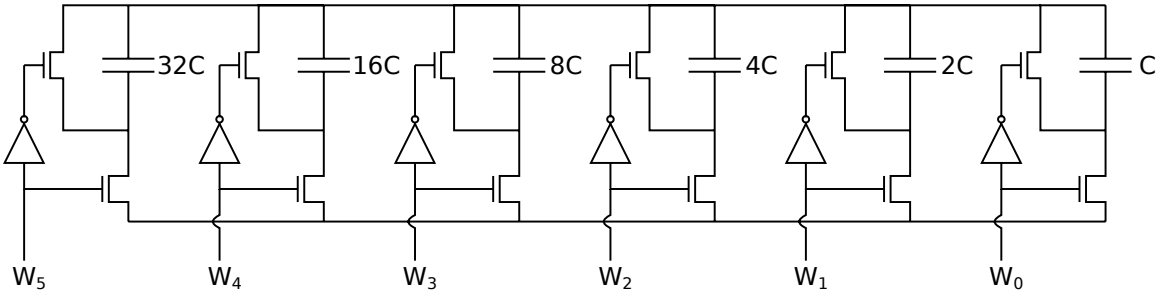


Figure 3.27: Row Slice of the Programmable Capacitor Array for GF2 VVM.

same decoding architecture, the first order $\Sigma\Delta$ modulator, the output charge from the computation is converted back into the digital domain as a PDM signal with output precision varying with time. Furthermore, this design was implemented in the GlobalFoundries 55nm Low-Power Enhanced (LPe) CMOS process.

3.4.1 Binary Weighted Capacitor Array

The capacitor array for the GF2 VVM core is comprised of 9 rows of programmable capacitors for computing vector-vector multiplications qualitatively as charge. Each programmable capacitor is modulated by a 6-bit weight to encode 64 different capacitance value. The circuit for this programmable capacitor is shown in Figure 3.27. By adopting a binary-weighted encoding scheme, the area and energy cost was reduced from the previous design, which employed magnitude comparators for a unary-weighted encoding scheme. Nonetheless, as the modified array uses different sized capacitors, the revised design is more susceptible to inaccuracies due to fabrication and process variations.

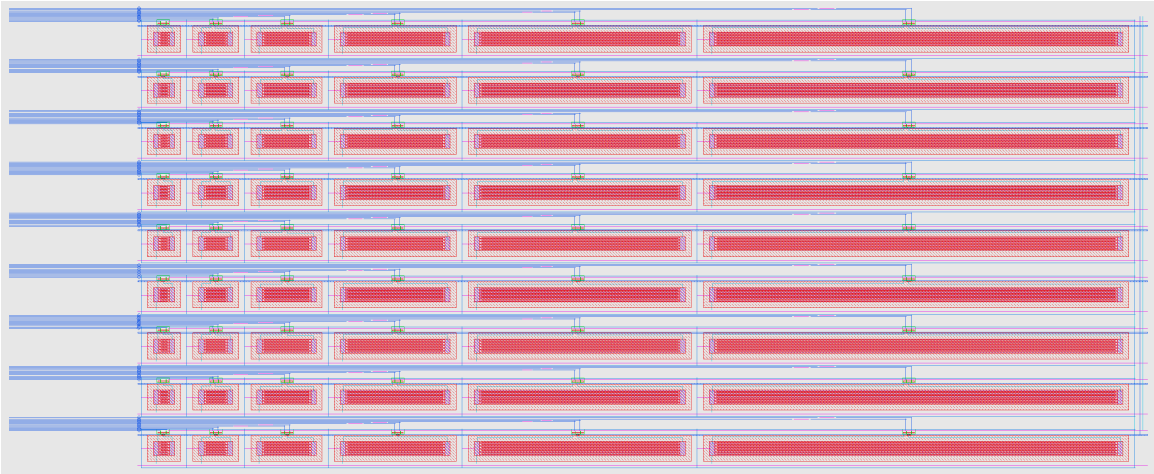


Figure 3.28: Layout for the Binary-Weighted Capacitor Array for GF2 VVM.

Each programmable capacitor is composed of VNCAPs, similar to the GF1 VVM design, and each capacitor is shielded to minimize noise. The layout for the full binary-weighted capacitor array can be seen in Figure 3.28. Based on the noise analysis using Equation 3.11, the capacitor for the least significant bit of the weight was sized for 3.6fF. Collectively, the array encodes 567 different levels ($(2^6 - 1) * 9$) with a maximum capacitance of 321.9fF.

A simulation program with integrated circuit emphasis (SPICE) model of the extracted layout netlist is simulated to measure the output transfer curve characteristics and the energy cost for computing with this circuit. An ideal integrator is used for translating the charge from the implicit product of the weight as capacitance with a fixed voltage bias as a delta voltage on an integrating capacitor. Plots of the output transfer curve with two distinct input voltage biases, 240mV and 60mV, can be seen in Figure 3.29. The integrating capacitor was sized to 1.48pF. Analysis of the transfer

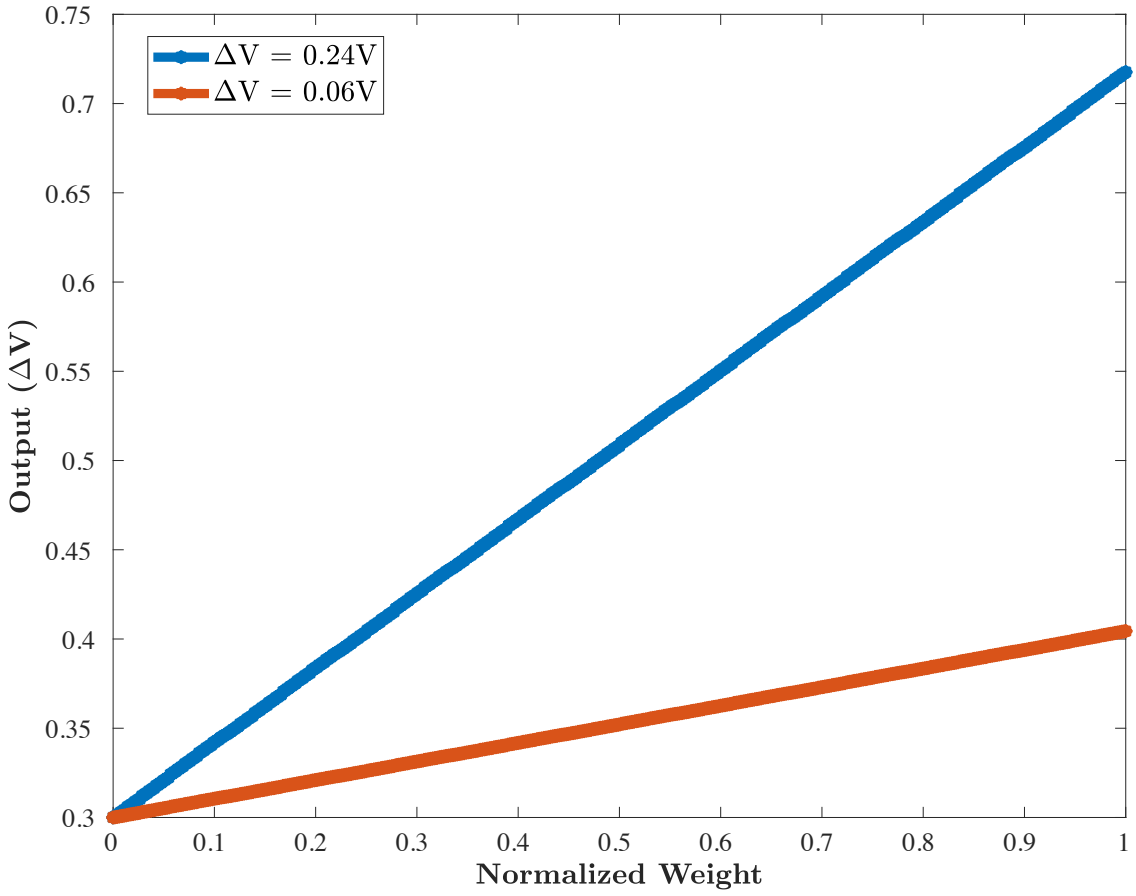


Figure 3.29: Transfer Curve Plots of the Weight (Capacitance) to Output (Voltage) for the GF2 VVM. The transfer curve is generated for two input voltage bias values of 60mV and 240mV. The weight value corresponds to the capacitance of the array normalized to the maximum capacitance.

curve shows that the output has a minimum least significant value of 0.521mV and 0.131mV for a fixed input voltage bias of 240mV and 60mV respectively. The root mean square thermal voltage noise on the integrating capacitor, which is 0.05mV, doesn't exceed these LSB values, and the DNL and INL values from the plots were also below this LSB value.

Moreover, the energy cost for computing with different weight values (capacitance)

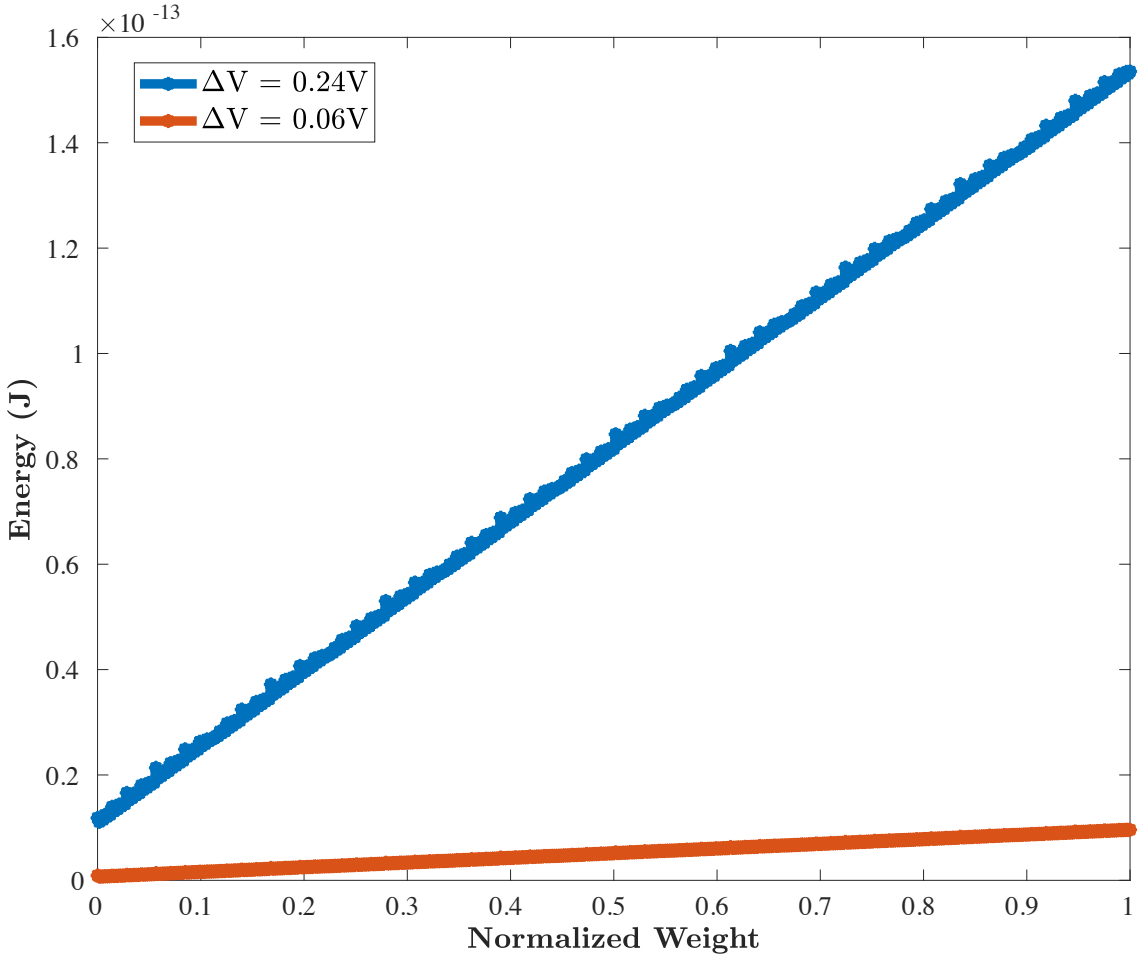


Figure 3.30: Plots of Energy Cost for the Capacitor Array with Different Weights and Input Voltage. The energy is measured with an input voltage bias of 60mV and 240mV for all discrete values of the weight encoded as capacitance on the array.

is also measured from SPICE simulations. The current from the power supplies are integrated onto an other capacitor, and using the Equation 3.36, the energy is measured for the different power supplies. The plots of this simulation can be seen in Figure 3.30. The simulation results show that for both voltage bias values, 240mV and 60mV, the energy cost on the array is on the order of femotoJoules(fJs). Using a 240mV input voltage bias, the energy cost each cycle doesn't exceed 130fJ, and

10fJ for the 60mV input voltage bias. Furthermore, the results also show the energy scaling quadratically with the input voltage bias. Compared to the GF1 VVM unary capacitor array design, the binary-weighted capacitor array uses 4 times less energy; this can mainly be attributed to the simplification of the digital periphery circuitry, as the magnitude comparators are not implemented.

3.4.2 VVM Core Architecture

The VVM core architecture for the GF2 VVM design is composed of the binary-weighted capacitor array and the same $\Sigma\Delta$ modulator ADC from the GF1 VVM design. The full circuit for this core can be seen in Figure 3.31. The ADC has the same specifications listed in Table 3.2. In addition, the analog biases are the same as Table 3.1. The full mixed-signal VVM core was simulated for functional verification, and the final layout for this core can be seen in Figure 3.32. The full mixed-signal core measures $159\mu m$ by $265\mu m$ ($0.042mm^2$) in silicon area. Compared to the GF1 VVM core, the GF2 VVM core uses approximately 2.24 times less area. A comparsion of these two designs can be seen in Figure 3.33.

3.4.3 Test Chip

A test chip is fabricated in GlobalFoundries 55nm CMOS technology in order to test this revised VVM core design. The core is interfaced to synthesized periphery

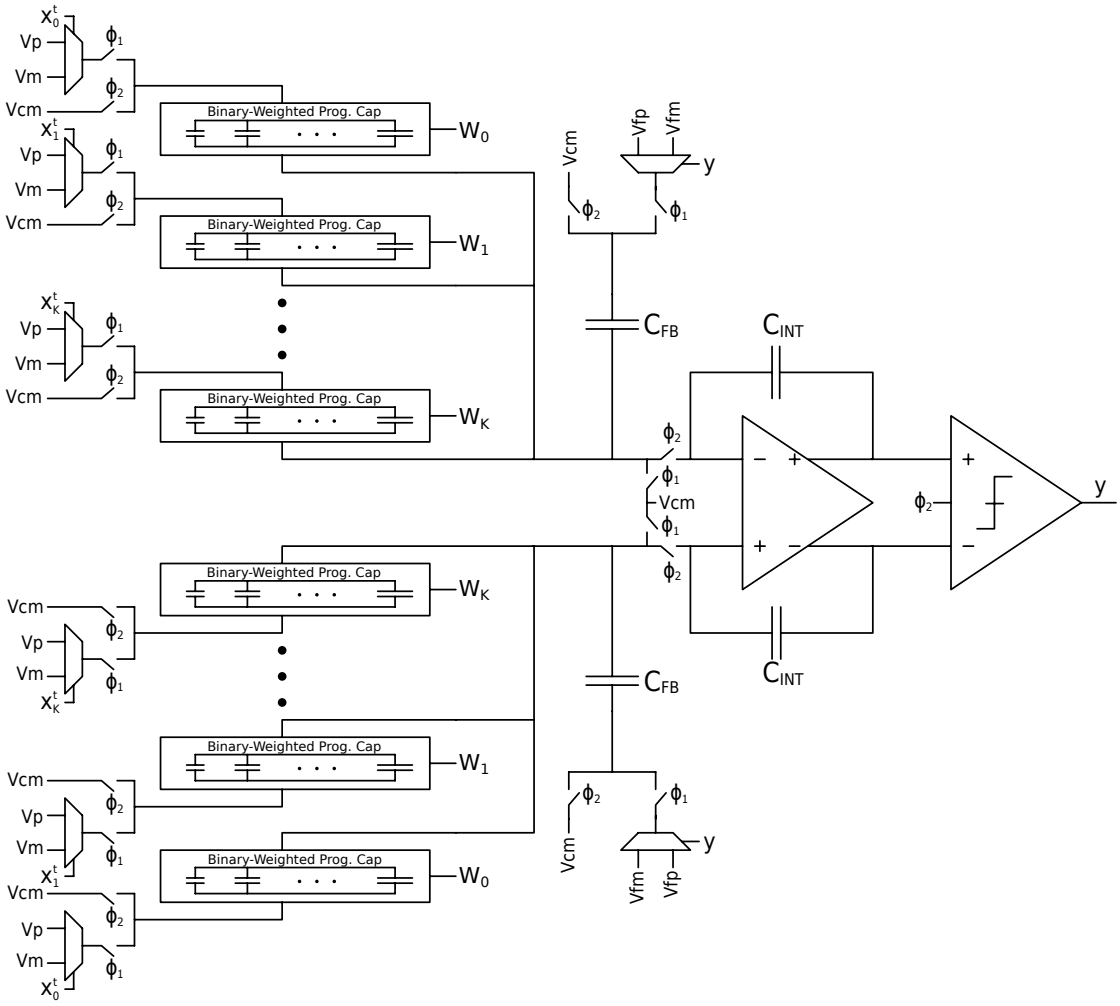


Figure 3.31: GF2 VVM Core Architecture.

circuitry that is used for local data and parameter storage. Similar to the GF1 VVM test chip, data is loaded to and from the chip using a serial protocol with a 1-bit data port because of limited I/O pins. The full chip layout, including custom built pad I/O circuitry cells, can be seen in Figure 3.24. The micrograph of the GF2 VVM chip can be seen in Figure 3.35. Because of top metal fill, the layout of the design can not be easily seen.

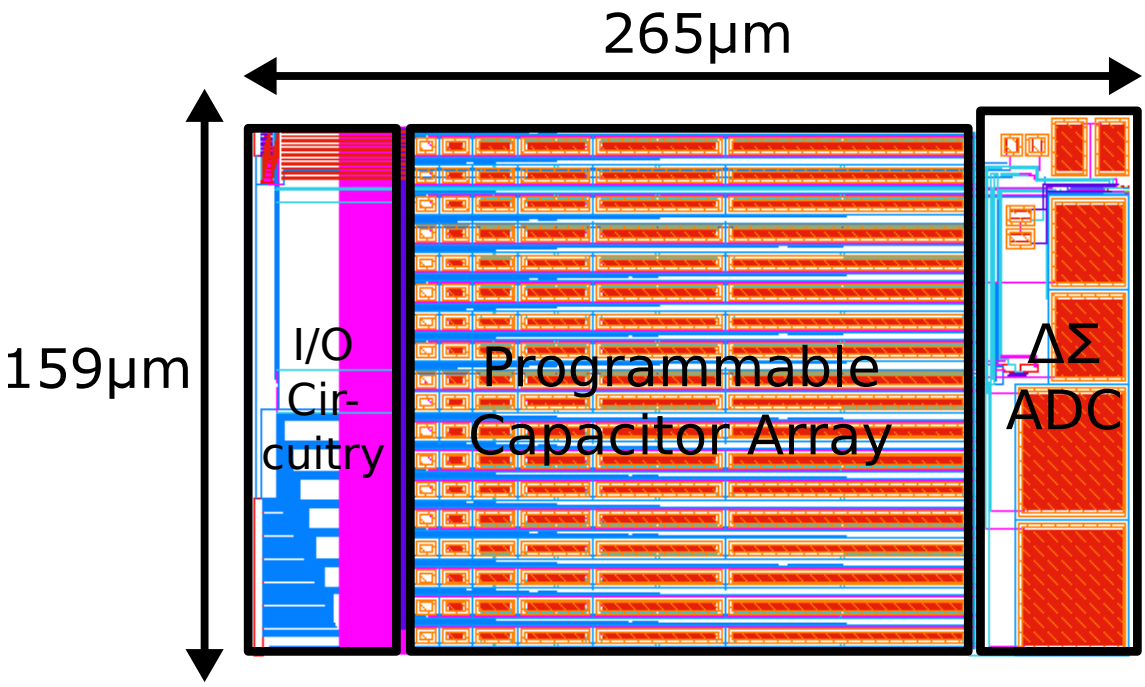


Figure 3.32: Layout of the GF2 VVM Core Architecture.

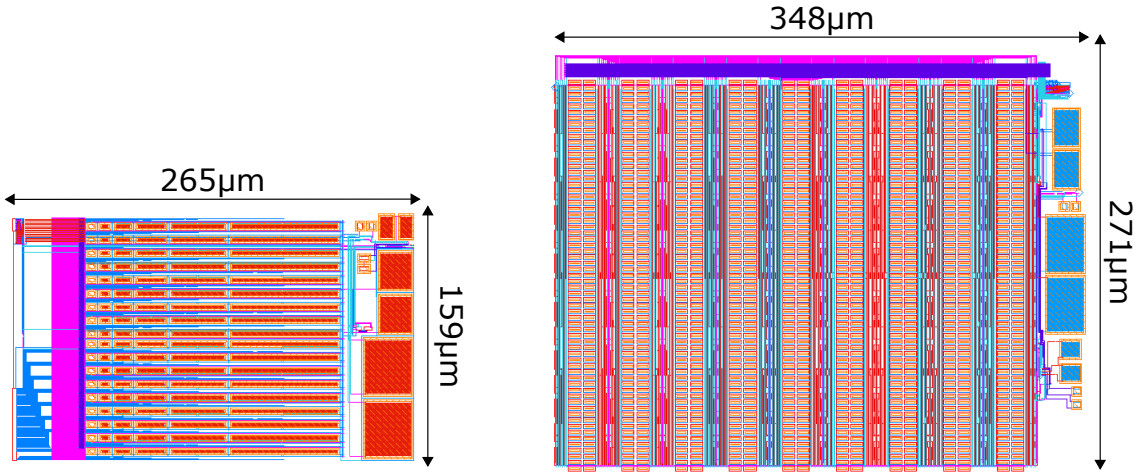


Figure 3.33: Design Comparison of the GF1 VVM Core and the GF2 VVM Core.

Moreover, the digital I/O interface was slightly modified from the previous design. The complete list of the digital I/O ports used for this test chip, as shown in the

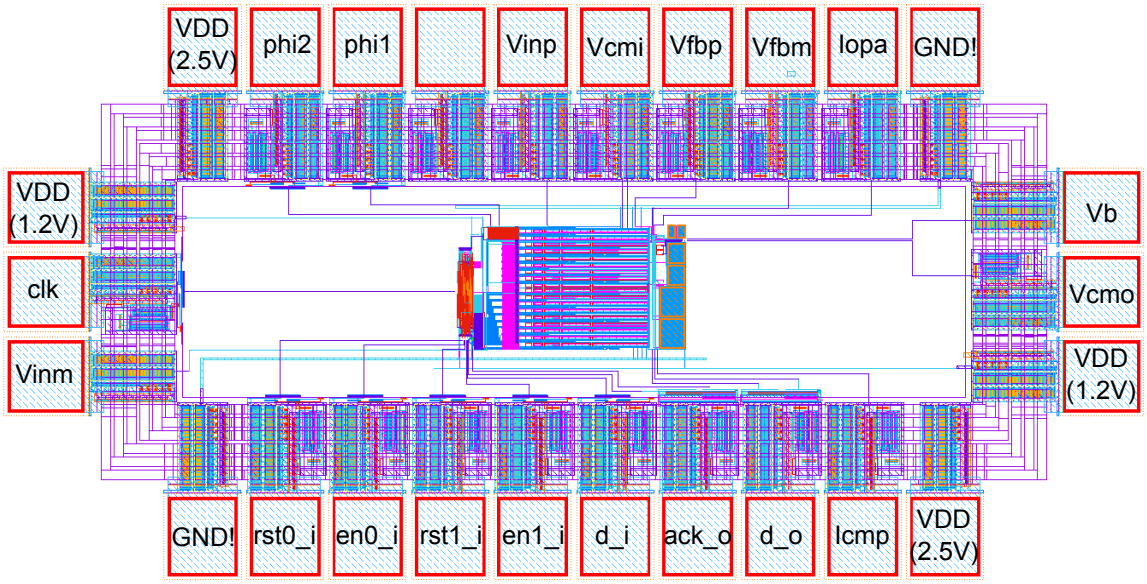


Figure 3.34: Layout of the GF2 VVM Chip.

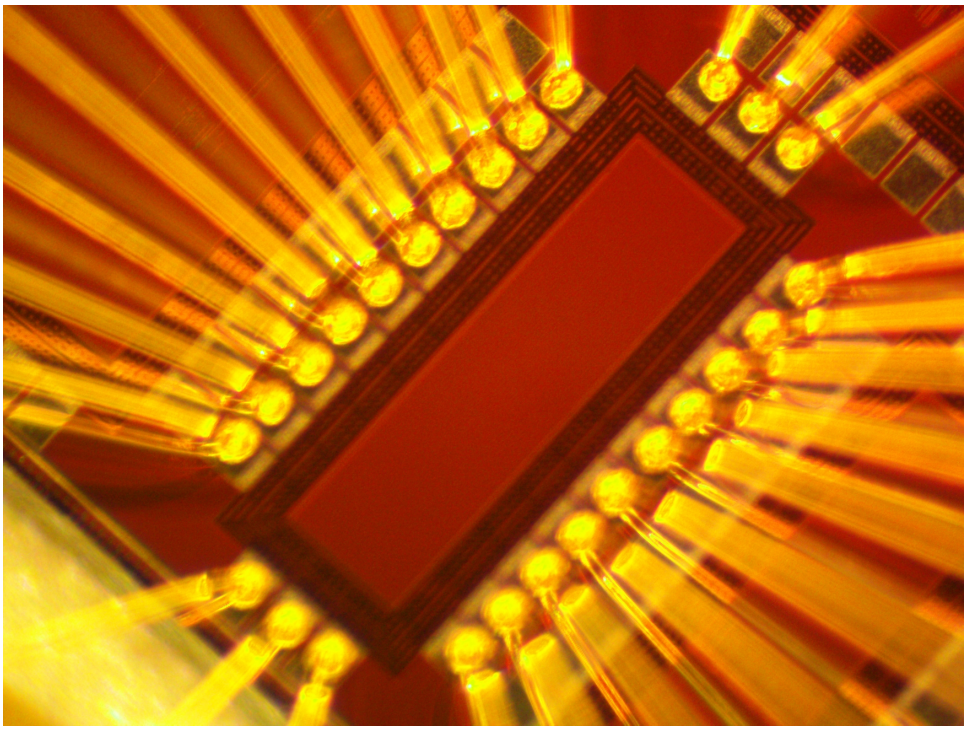


Figure 3.35: GF2 VVM Chip Micrograph.

Table 3.5: Digital I/O Ports for the GF2 VVM Chip.

Name	Type	Width	Description
clk	Input	1	Clock signal for periphery circuitry.
phi1	Input	1	VVM core clock signal for first phase.
phi2	Input	1	VVM core clock signal for second phase.
rst0_i	Input	1	Reset signal for the weight register files.
rst1_i	Input	1	Reset signal for the input register files.
en0_i	Input	1	Enable signal used for loading the weights.
en1_i	Input	1	Enable signal used for loading the inputs.
d_i	Input	1	Data signal for both serial interfaces..
d_o	Output	1	VVM core output signal.
ack_o	Output	1	Acknowledge signal for the input serial interface..

annotated chip layout, is described in Table 3.5. The select port *sel_i* from the GF1 VVM design were removed, and all the weights are programmed sequentially through the serial protocol. These changes not only simplified the periphery circuitry, but also reduced the number of ports used in the design.

Similar to the test setup shown in Figure 3.25, the GF2 VVM chip was wire-bonded to a 40 DIP, and then interfaced to a breakout board, which connects to a DAC board and a Spartan-3 FPGA. This chip was programmed with different test patterns from MATLAB in order to measure performance, computation accuracy, and efficiency. The measured characteristics can be seen in Table 3.6. The measured energy cost per operation for just the array decreased significantly from the GF1 VVM design; however, because the bulk of the energy cost is attributed to the ADC, the overall efficiency of this revised VVM core only improved slightly to 15.2GOP/W from 14.8GOP/W. Nonetheless, the area cost was reduced by 50% from the previous design.

Table 3.6: Measured Characteristics of the GF2 VVM Core.

Technology	55nm CMOS
Die Area	$0.042mm^2$
Operation	6-bit MAC
Precision	5-bit
Supply Voltage	1.2V
Frequency	2.5MHz
Throughput	350KOPs
Power Consumption	$23.1\mu W$
Energy/Op	65.67pJ (0.9pJ for just the array)
Efficiency	15.2GOP/W (1.11TOP/W for just the array)

3.5 GF3 VVM: A 4-bit MAC Multicore Processor in 55nm CMOS

In the third revision of the mixed-signal VVM design, a multicore processor based on the architecture used in the GF1 VVM design (Section 3.3) is implemented in 55nm CMOS. The fabricated test chip is comprised of 192 VVM cores, which can be configured as (192) 4-bit weight cores, (96) 8-bit weight cores, or (64) 12-bit weight cores. Moreover, the multicore processor is designed with flexible output precision, similar to the previous implementations detailed in Section 3.3 and 3.4, and can be adjusted for 9-12-bits of output precision.

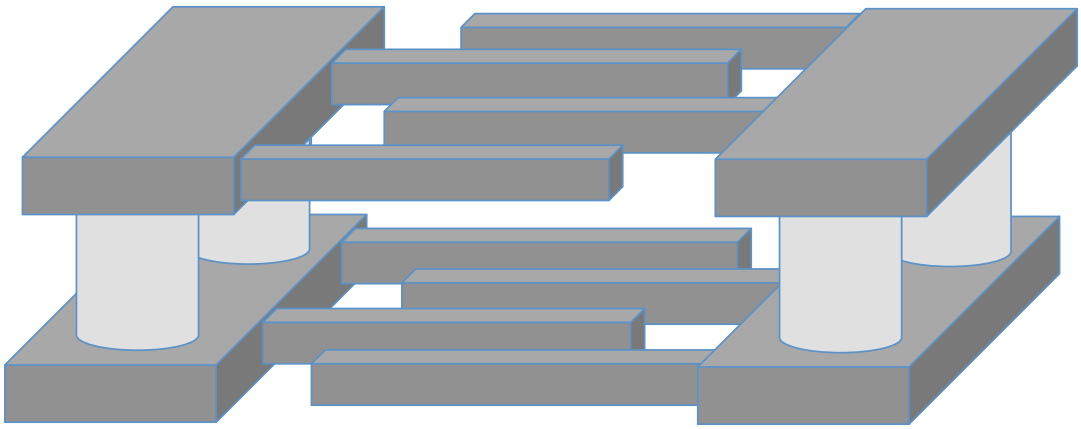


Figure 3.36: 3D View of a Two-Layered APMOM Capacitor.

3.5.1 VVM Core Architecture and Design Improvements

The GF3 VVM core is composed of a unary-encoded 2D capacitor array programmed through (16) 4-bit weights and the same $\Sigma\Delta$ modulator ADC circuit presented in Section 3.2 with characteristics shown in Table 3.2. As opposed to using VNCAPs, custom alternative polarity metal-oxide-metal (APMOM) capacitors are used that alternates the polarity of the metal fingers across the layers. This alternating polarity of interdigitated fingers increases capacitance density by adding top and bottom capacitance along with the side capacitance in the same area. A 3D view of a two layered APMOM capacitor can be seen in Figure 3.36. Exploiting this topology for higher capacitance density, the unit capacitor for the array is laid out as seen in Figure 3.37. This capacitor was designed across 5 metal layers (M2-M6), where the bottom layer (M2) and the top layer (M6) are used for shielding, and a metal

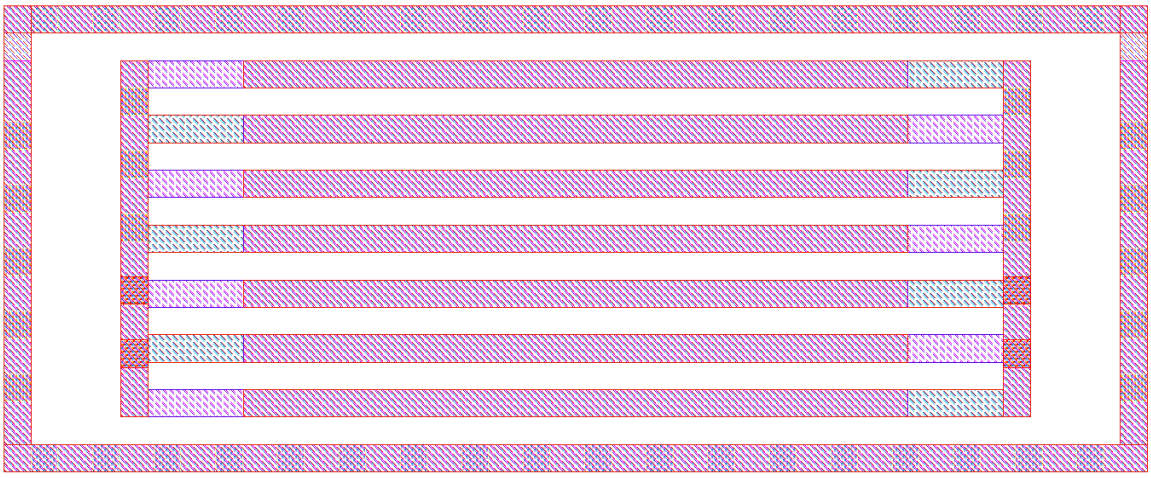


Figure 3.37: Layout of the Unit Capacitor in the GF3 VVM Core.

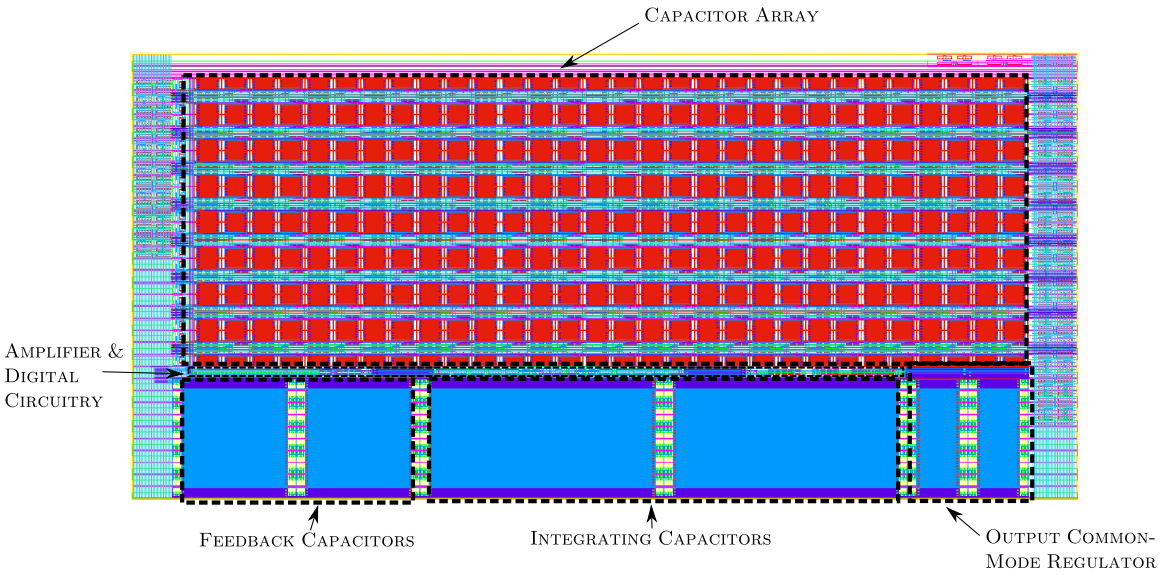


Figure 3.38: Annotated Layout of the GF3 VVM Core.

shielding is added along the perimeter to mitigate parasitic coupling from adjacent cells. The capacitance of the unit capacitor was estimated to 10fF.

The annotated layout for the VVM core, which follows from the circuit displayed in Figure 3.22, can be seen in Figure 3.38. With the modifications in the capacitor

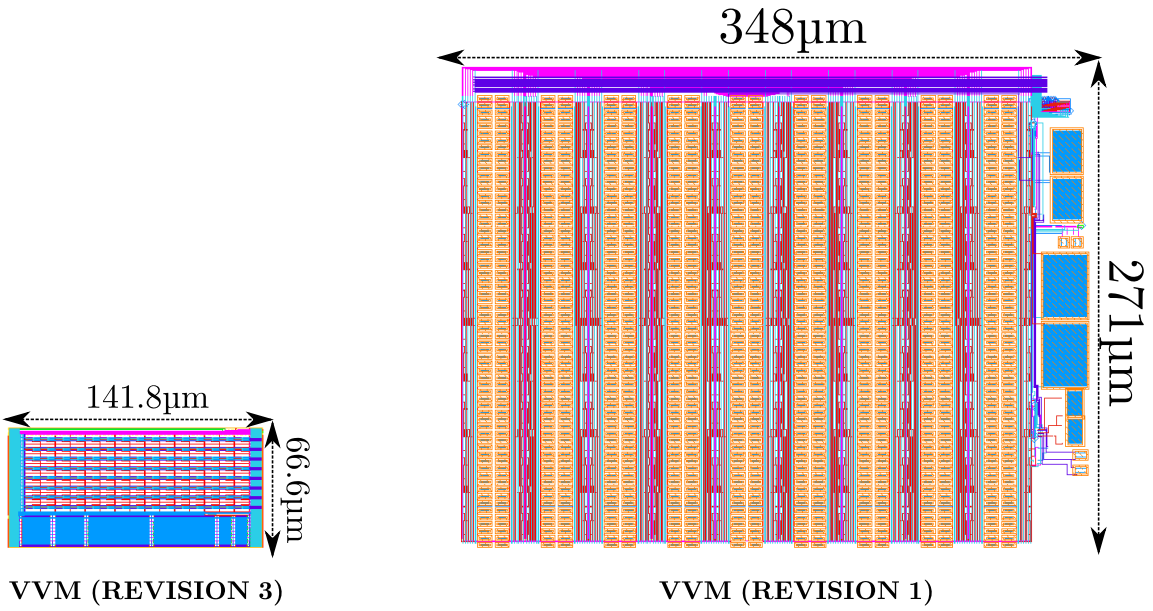


Figure 3.39: Design Comparison of the GF1 VVM Core and the GF3 VVM Core.

Table 3.7: Nominal Bias Values for the GF3 VVM Core.

Name	Description	Nominal Value
V_cm_i	Input common-mode voltage	0.32V
V_inp	Positive input voltage for the capacitor array	0.37V
V_inm	Negative input voltage for the capacitor array	0.27V
V_fbp	Positive feedback voltage	0.64V
V_fbm	Negative feedback voltage	0V
V_cm_o	Output common-mode voltage	0.5V
V_reg	Voltage bias for the operational amplifier	0.75V
I_amp	Current bias for the operational amplifier	2.5μA
I_comp	Current bias for the voltage comparator	2μA

array, the revised VVM core was designed in roughly 10 times less area than the previous GF1 VVM core. This area comparison can be seen in Figure 3.39.

For the revised VVM core, the analog bias values were modified to account for modifications in the input capacitor array, feedback capacitor, and integrating capacitor. The nominal voltage and current bias values are listed in Table 3.7.

Table 3.8: Simulated Characteristics of the GF3 VVM Core.

Technology	55nm CMOS
Operation	4-bit MAC
Supply Voltage	1.2V
Frequency	1MHz
Throughput	1MOPs
Power Consumption	6.88μW
Energy/Op	6.88pJ (66.23fJ on the array)
Efficiency	145.34GOP/W (15.1TOP/W for the array)

Furthermore, the simulated characteristics of the revised VVM core can be seen in Table 3.8. This core was simulated with a 1MHz clock for computing 4-bit MAC operations in 16 clock cycles. In the analog domain, the capacitor array computes the MAC operations with energy costs ranging from 51.83fJ to 80.63fJ. Totally, this operation is executed using the core with an energy cost of 6.88pJ and a total efficiency of 145.34GOP/W. On the array only, this efficiency is measured at approximately 15.1TOP/W.

After the core was designed and functional verified with simulations, timing analysis and physical characterization was done in order to construct a library that can be used for synthesizing and placing and routing the multicore design. This timing analysis included simulations from the parasitic extracted layout to measure relevant timing information, such as input capacitance load and output rise and fall delays relative to the output load and input slew rate, for meeting timing constraints for the top level design. Moreover, physical characterization was done for extracting physical information such as routing paths and blockages, pin details, and more for proper placement and routing of these cores on the top level design.

3.5.2 Multicore Design and Synthesis

The multicore processor of 192 VVM cores is synthesized across three levels of hierarchy — first a **Cell** of 6 VVM cores that are multiplexed for variable weight precision, then a **Cluster** of 4 cells with a total of 24 VVM cores that share periphery logic for data encoding, and finally a **Top** level of 8 standalone clusters for a total 192 VVM cores.

3.5.2.1 Cell Design

The Cell is the first level hierarchy of the multiprocessor design that incorporates 6 VVM cores for computing inner products with different weight precision. By default, this block uses these cores standalone to compute inner products on vectors with 16 elements and a weight precision of 4 bits. Alternatively, these cores can be combined to compute with a weight precision of 8 bits or even 12 bits. For computing with higher weight precision, first the bits of the weights are split into chunks of 4 bits based on their significance, then the weights are distributed to separate VVM cores, and finally the 1-bit PDM outputs are scaled according and added together with synthesized digital logic. The addition and scaling is done with a stream adder that handles scaling the input and also saturation in the 1-bit PDM output with the implementation of additional counters. In order to implement the scaling, the lesser significant input is accumulated on a counter, and once the appropriate count is reached a 1 is added with a 1-bit adder. Furthermore, the overflow or carry-out of

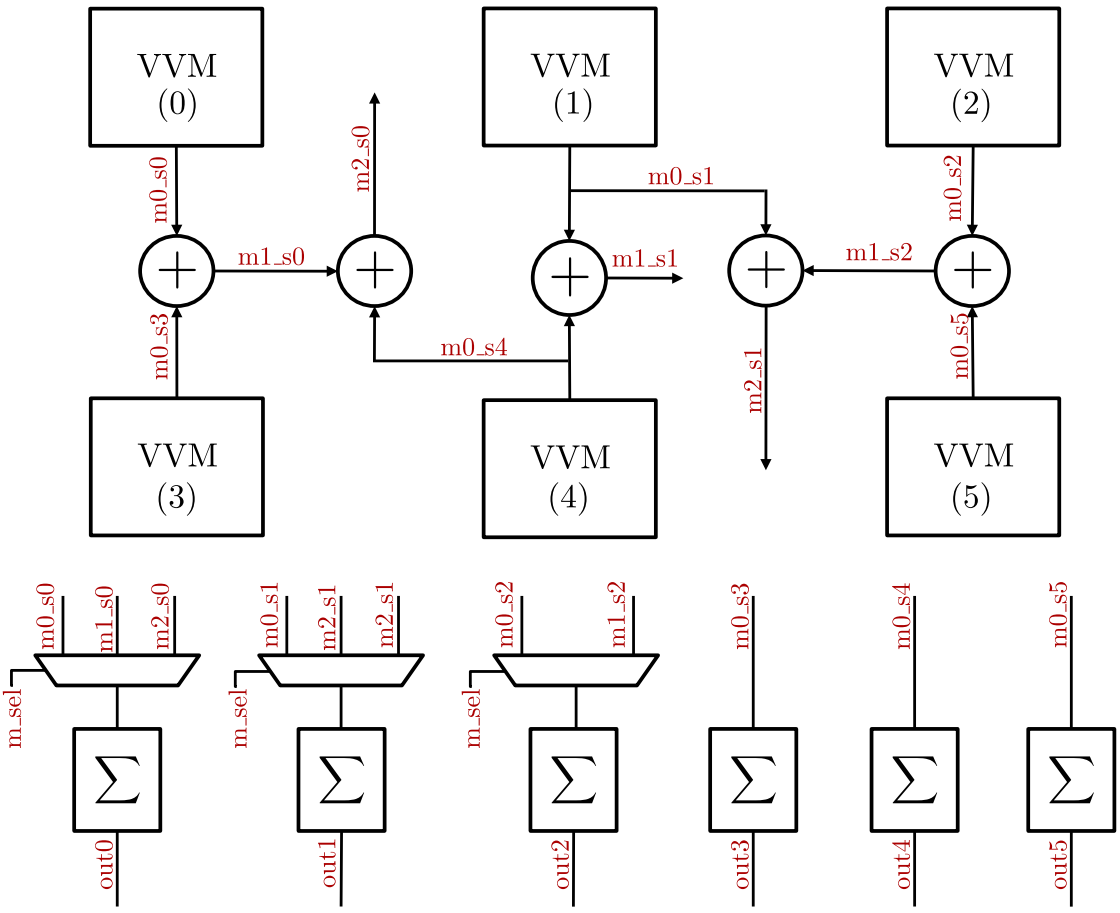


Figure 3.40: Block Diagram of the GF3 VVM Cell Design.

the adder is added to an additional counter to account for output saturation. This stored saturated bit is asserted into the 1-bit output stream when the output is no longer saturated. The final output of this computation is integrated from the output stream with the possible output precision of 9 – 12 bits. A block diagram of this VVM Cell design can be seen in Figure 3.40. Depending on the mode selected, the Cell produces a variant of 11 possible outputs; the output assignment is detailed in Table 3.9.

Table 3.9: GF3 VVM Cell Output Table.

Mode (m_sel)	Weight Precision	Output
0	4 bits	$m0\_s0$ & $m0\_s1$ & $m0\_s2$ & $m0\_s3$ & $m0\_s4$ & $m0\_s5$
1	8 bits	$m1\_s0$ & $m1\_s1$ & $m1\_s2$
2	12 bits	$m2\_s0$ & $m2\_s1$

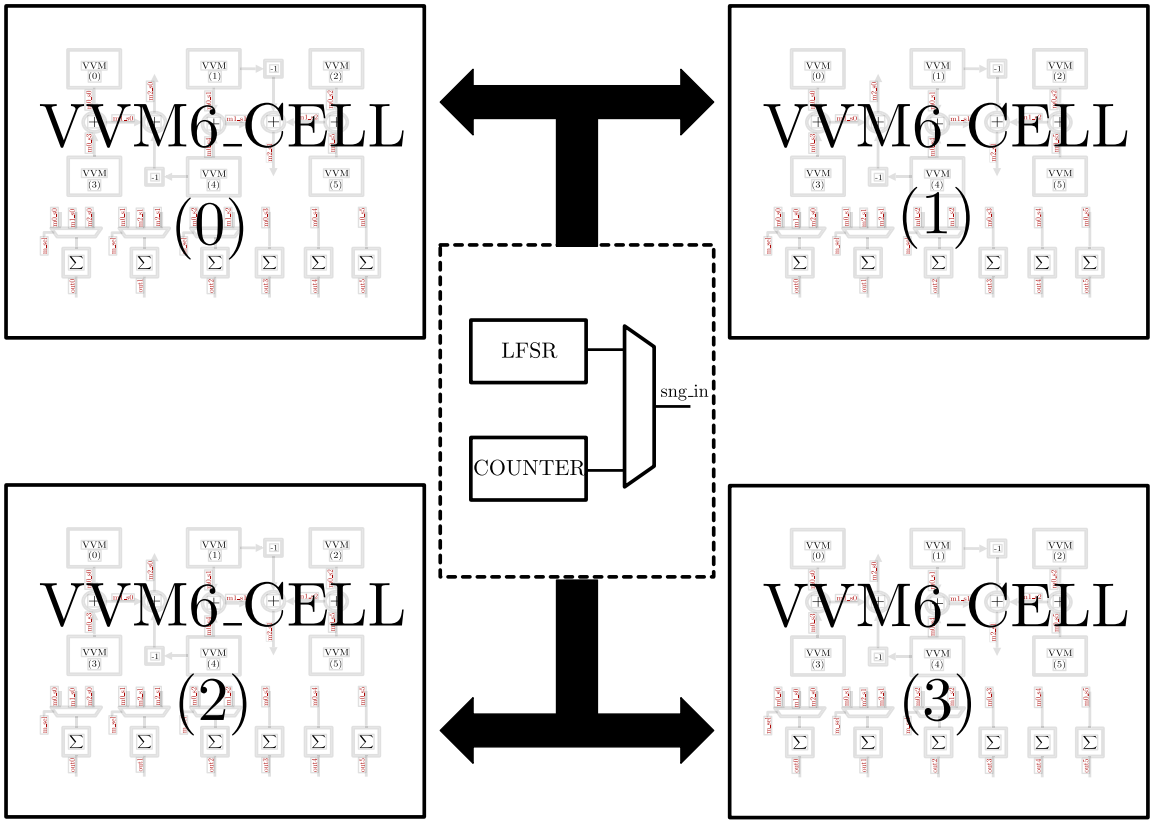


Figure 3.41: Block Diagram of the GF3 VVM Cluster Design.

3.5.2.2 VVM Cluster Design

The next tier of hierarchy for the GF3 VVM design, the Cluster, combines 4 Cells that shares periphery encoding logic. The block diagram of the Cluster can be seen in Figure 3.41. The individual Cell subblocks are identified as $VVM6\_CELL$ blocks

in the diagram, and contain the design detailed in the prior section. A total of 24 VVM cores from 6 Cells are instantiated in this design, and they share a counter and a LFSR, which are used for the unary encoding of the input data. Additionally, each cluster incorporates a simple serial I/O interface for reading/writing with a data widths of 4-bits due to pin limitation on the top level. This interface operates on a separate I/O clock that runs at 100MHz.

3.5.2.3 Top Design

The final tier in the hierarchical GF3 VVM multicore design is the Top level, and in this block, an array of standalone cluster subblocks are placed and routed together with decoder logic for reading, processing, and writing to the individual clusters of VVM cores. This top cell is based on a scalable design that places a generic number of clusters to fit the allotted area. In the GF3 VVM design, this amounted to a total of 8 clusters with a total of 192 VVM cores. A block diagram of the GF3 VVM Top design can be seen in Figure 3.42. The individual Cluster subblocks are identified as $VVM6\_CELL\ CLUSTER$, and each contain 24 VVM cores. As described earlier, each cluster is implemented with a serial I/O interface running on a separate 100MHz clock. These I/O interfaces are coupled together in the top design and the inputs and outputs are read and written to the individual clusters sub-blocks using address ports.

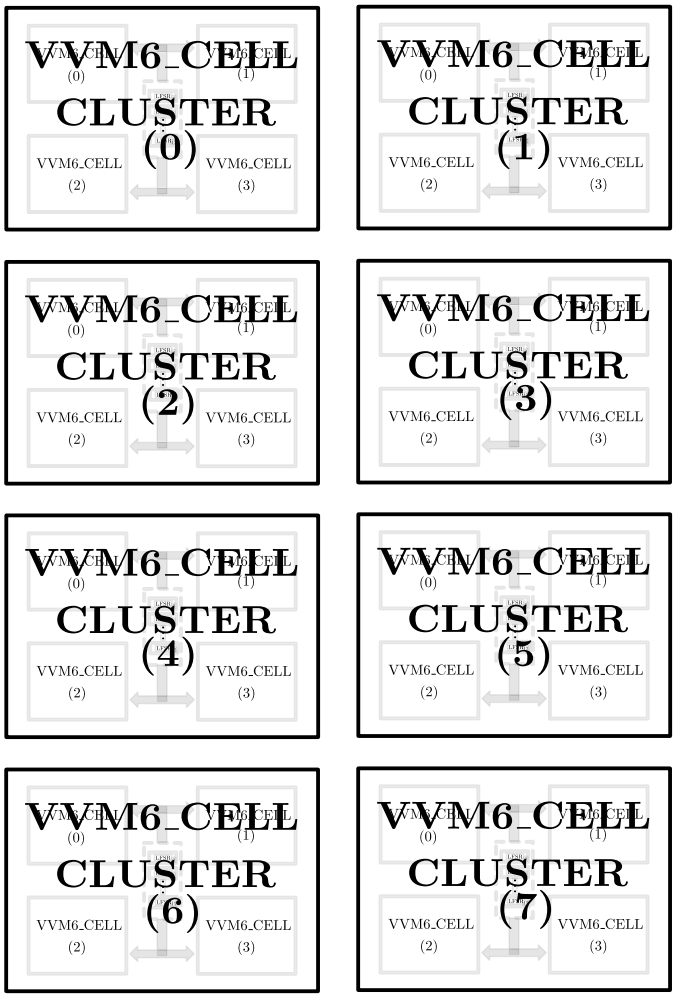
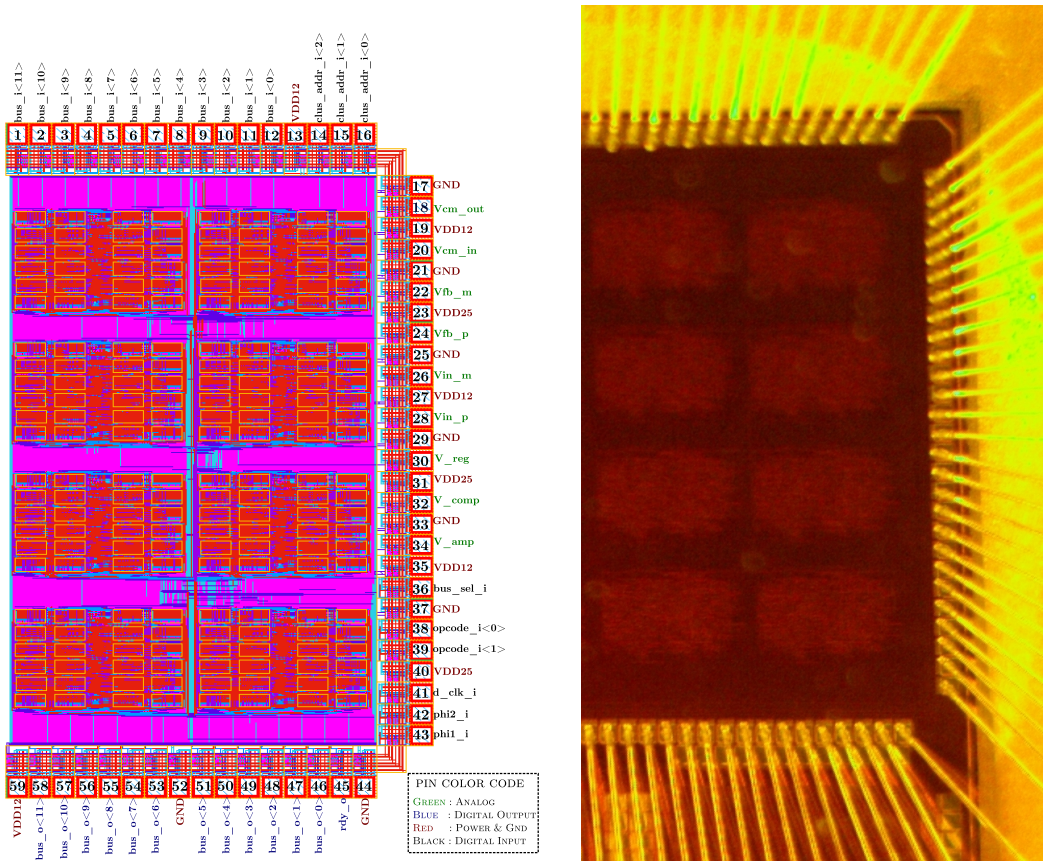


Figure 3.42: Block Diagram of the GF3 VVM Top Design.

3.5.3 Test Chip

The annotated top layout and the micrograph for the GF3 VVM chip along with the pad frame, which was designed and fabricated in the GlobalFoundries 55nm CMOS process, can be seen in Figure 3.43. As described in Section 3.5.2, a total of 192 mixed-signal VVM cores are placed and routed on the top design, which can be programmed through a serial I/O interface to compute variable precision inner



(a) Layout (b) Micrograph

Figure 3.43: Annotated Layout and Micrograph of the GF3 VVM Chip.

products with 4-bit weights, 8-bit weights, and 12-bit weights.

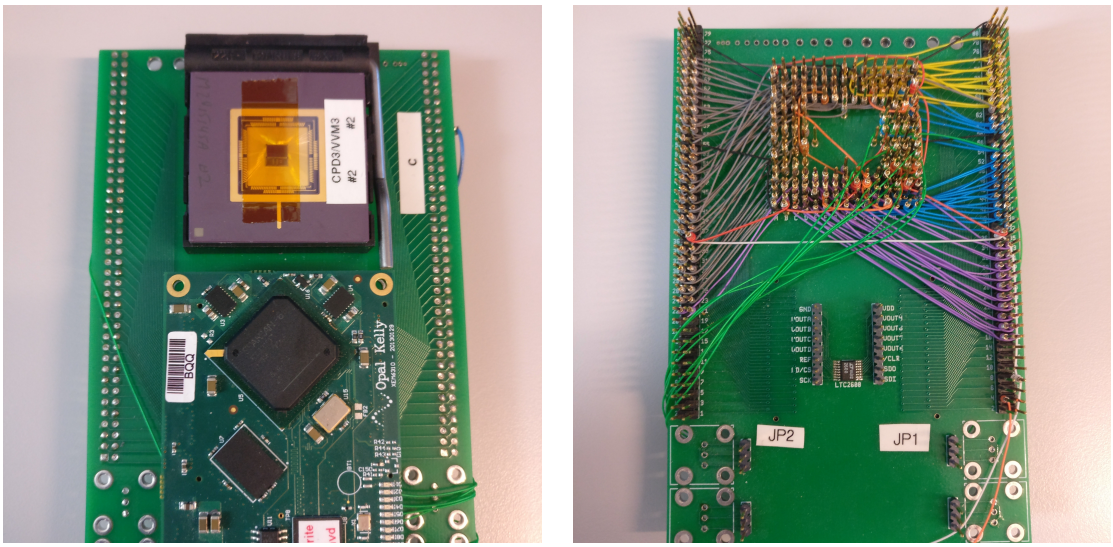
A total of 59 I/O and power pins are used in the test chip for programming and processing with the VVM cores, and they are detailed in Table 3.10. With regards to pin types, POW is power, GND is ground, DI is digital input, DO is digital output, and AI is analog input. Similar to the GF1 VVM and GF2 VVM designs, 9 analog biases are used in this design.

Furthermore, the fabricated test chip was wire-bonded to a 144 DIP and mounted on a custom breakout board that interfaced to a DAC board and Spartan-6 FPGA.

Table 3.10: I/O and Power Ports for the GF3 VVM Chip.

Name	**Type**	**Width**	multicolumn1—c—**Description**
GND	POW	N/A	Ground
VDD12	POW	N/A	1.2V power supply
VDD25	POW	N/A	2.5V power supply
Vcm_out	AI	N/A	Output common-mode voltage for amplifier
Vcm_in	AI	N/A	Input common-mode voltage
Vfb_m	AI	N/A	Negative feedback voltage
Vfb_p	AI	N/A	Positive feedback voltage
Vin_m	AI	N/A	Negative input voltage
Vin_p	AI	N/A	Positive input voltage
V_amp	AI	N/A	Amplifier bias voltage
V_reg	AI	N/A	Voltage bias for regulating output common-mode
V_comp	AI	N/A	Comparator bias voltage
d_clk_i	DI	1	I/O clock signal
phi1_i	DI	1	Non-overlapping clock for the VVM cores
phi2_i	DI	1	Non-overlapping clock for the VVM cores
bus_i	DI	12	Input data bus for control, address, and data signals
clus_addr_i	DI	3	Address signal for selecting between the clusters
opcode_i	DI	2	Opcode for selecting bus functionality
bus_sel_i	DI	1	Selects the output of the output data bus
bus_o	DO	12	Output data bus signal
rdy_o	DO	1	Data output ready signal

Figure 3.44 shows the test board along with the packaged chip mounted in a soldered socket. The 59 pins for the analog biases, power supplies, and digital I/O signals are wire-wrapped from the bonded package to the breakout board. Using the Opal Kelly FrontPanel software interface, the DAC board and the chip can be programmed for testing with the Spartan 6 FPGA.



(a) Front View (b) Back View

Figure 3.44: GF3 VVM Chip Test Setup.

3.5.3.1 Chip Interface

The GF3 VVM chip communicates externally with a serial I/O interface using the following ports:

- 12-bit input data bus
- 2-bit opcode
- 3-bit address
- 12-bit output data bus
- 1-bit bus output select
- 1-bit output ready signal

The input data bus is decoded into control signals, internal block addresses, and input data bits based on the configuration of the 2-bit opcode. The 3-bit address selects one of the 8 clusters to write the opcode and input data bus signals, and

receive the output from. The output data bus transmits the input data bus when the bus output select is low (for debugging purposes), and transmits the post-processed VVM output when the select is high. Finally, the output ready signal is used as a flag that signifies the completion of processing with the VVM cores. The various programming procedures for the chip are outlined below.

- **Reset Procedure**

    1. First select which cluster to reset by setting the 3-bit *clus_addr_i* signal.
    2. Set the opcode to "**00**", which allows the block to register the control signals from the input data bus.
    3. Next, set the input data bus to "**000000000001**", which will put the block in reset mode. (The input data bus has to be held at this value for at least one clock period of the *phi1_i* and *phi2_i*.
    4. Finally, set the input data bus to "**000000000000**" to take the block out of reset mode.

- **Set Mode & Precision Procedure**

    1. First select which cluster to set the mode and precision by setting the 3-bit *clus_addr_i* signal.
    2. Then set the opcode to "**00**", which allows the block to register the control signals from the input address.

3. Finally, set the mode by programming the 3rd and 4th bit of the input data bus, and the precision by programming the 5th and 6th bit of the input data bus. For setting the mode to 1 and the precision to 12bits (precision select of 3), the input data bus is configured as "**000000110100**".

- **Write Weight or Input Value Procedure**

1. First select which cluster to write the weight or input value to by setting the 3-bit *clus_addr_i* signal.
2. Set the opcode to "**01**", to write address signals
3. In the input data bus
   - set the 12th bit to"**1**", which sets the write enable for the weight and input values high.
   - select which of the 4 Cell to write to, by setting the 10th and 9th bit.
   - select which of the 6 VVM cores to write to by setting the 8th-6th bit.
   - select which of 16 pair of weight and input value to set by setting the 5th-2nd bits.
   - select whether to write a weight or an input value with the 1st bit ("**1**" selects weight, while "**0**" selects the input value).
4. Next set the opcode to "**10**", which writes the data signal.
5. Set the input data bus to the weight or input value that you want to write.

6. Repeat step 2 & 3, but set the 12th bit to "**0**", which sets the write enable low.

- **Set Encoder (LFSR or Counter) Procedure**

1. First set which cluster by setting the 3-bit *clus_addr_i* signal.
2. Set the opcode to "**00**" to write the control signals.
3. To use the LFSR for encoding random pulse density modulated (RPDM) unary signals, program the 7th bit in the input data bus as "**1**". To use the counter instead for generating pulse-width modulated (PWM) signals, program the 7th bit in the input data bus as "**0**".

- **Program LFSR (Write Seed) Procedure**

1. First select which cluster to program the LFSR by setting the 3-bit *clus_addr_i* signal.
2. Set the opcode to "**01**", to write address signals
3. In the input data bus
   - set the 12th bit to "**0**", which sets the write enable for the weights and input values low.
   - set the 11th bit to"**1**", which sets the seed write enable high.
   - select which of the 16 LFSR to write the seed to with the 5th-2nd bits.
4. Next set the opcode to "**10**", which writes the data signal.

5. Set the input data bus to the LFSR seed. (The input data bus must be held to this value for at least one clock period of the *phi1_i* and *phi2_i*
6. Repeat step 2 & 3, but set the 11th bit to "**0**", which sets the seed write enable low.

- **Enable VVM Core Processing Procedure**

1. First select which cluster to enable processing on the cores by setting the 3-bit *clus_addr_i* signal.
2. Set the opcode to "**01**", to write address signals
3. For the input data bus
   - make sure the 12th and 11th bit are set to "**0**" which sets both write enable low.
   - select which of the 4 VVM6 cells to start processing data, by setting the 10th and 9th bit.
4. Set the opcode to "**00**", which writes the control signals.
5. Set the 2nd bit in the input data bus to "**1**", which selects the process enable signal, set the mode select and precision select bit, and ensure that 1st bit, which is the reset signal is low.
6. When the *rdy_o* signal goes high for the block that was processing the data (set the address accordingly to check the ready out signal). The output data is ready to be read from the output data bus.

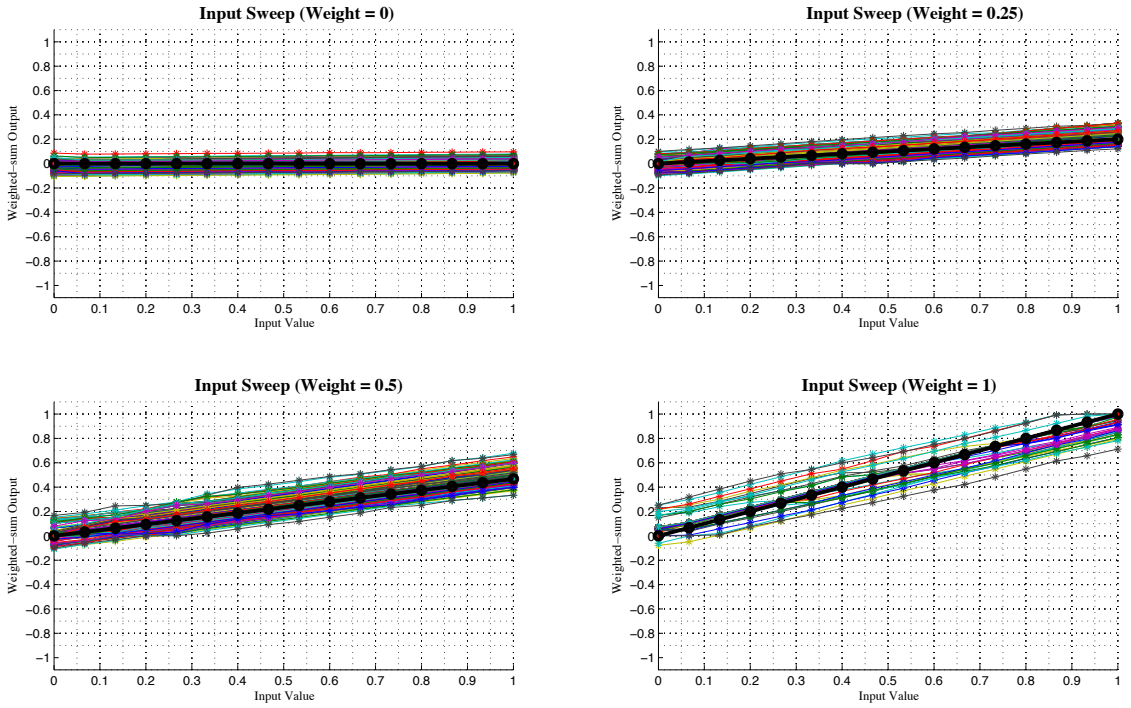


Figure 3.45: Input Sweep for Fixed Weights with the 192 GF3 VVM Cores from a Test Chip. For each plot, each weight is set to a normalized value of 0 (top left), 0.25 (top right), 0.5 (bottom left), and 1 (bottom right). All input are swept from 0 to 1 for each core.

3.5.3.2 Results

The GF3 test chip was tested to validate the multicore mixed-signal architecture, and to also measure computation accuracy and variability across cores. To accomplish this, input and weight test patterns are programmed on the cores, while measuring the outputs. Specifically, the inputs and weights are swept across their respective domain in LSB increments, and all 192 cores are recorded per step. Results showing the core outputs for fixed weight values as the input is being swept can be seen in Figure 3.45. To compensate for mismatch from the cores and variations in the shared

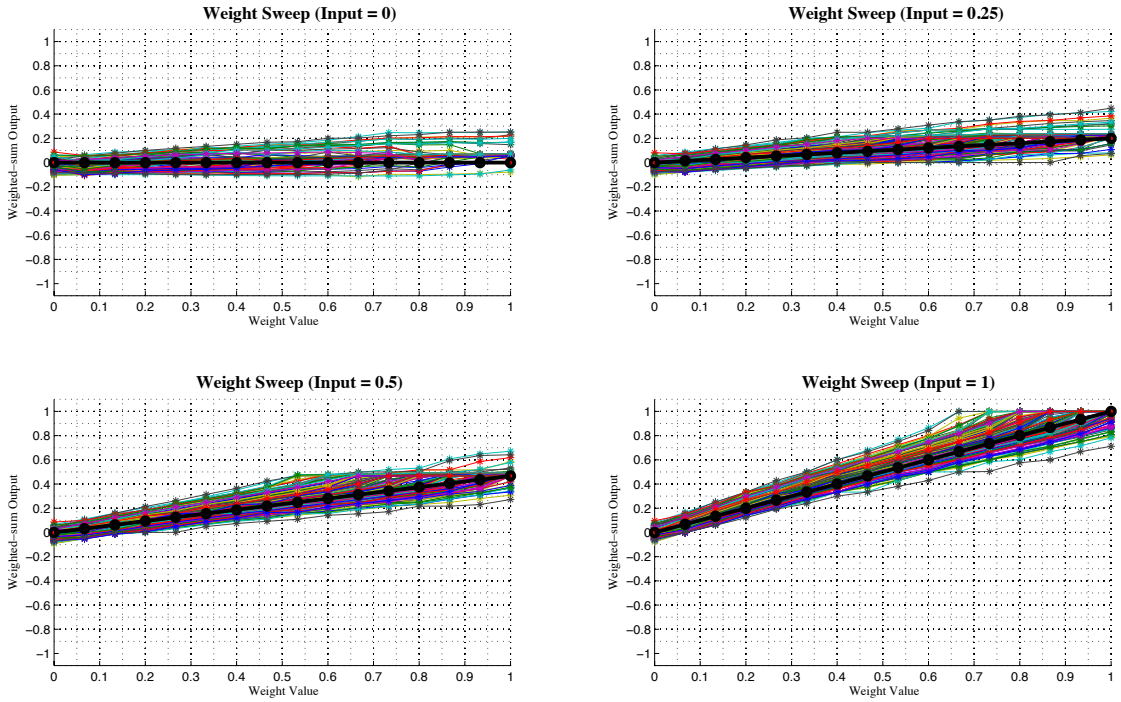


Figure 3.46: Weight Sweep for Fixed Inputs for the 192 GF3 VVM Cores in a Test Chip. For each plot, each input is set to a normalized value of 0 (top left), 0.25 (top right), 0.5 (bottom left), and 1 (bottom right). All input are swept from 0 to 1 for each core.

analog biases from parasitic coupling, each core is calibrated to minimize offset. Even with this calibration, there are noticeable variations in the computation result across the cores.

Furthermore, a plot of the core outputs for fixed input values and variable weight values can be seen in Figure 3.46. Similar to the previous plot, variations are noticeable across the cores even after calibrations. A histogram of the mean absolute error for 192 cores, shown in Figure 3.47, properly highlight these variations. The mean absolute error across cores varies from 0.017 to 0.158 on a normalized output scale of

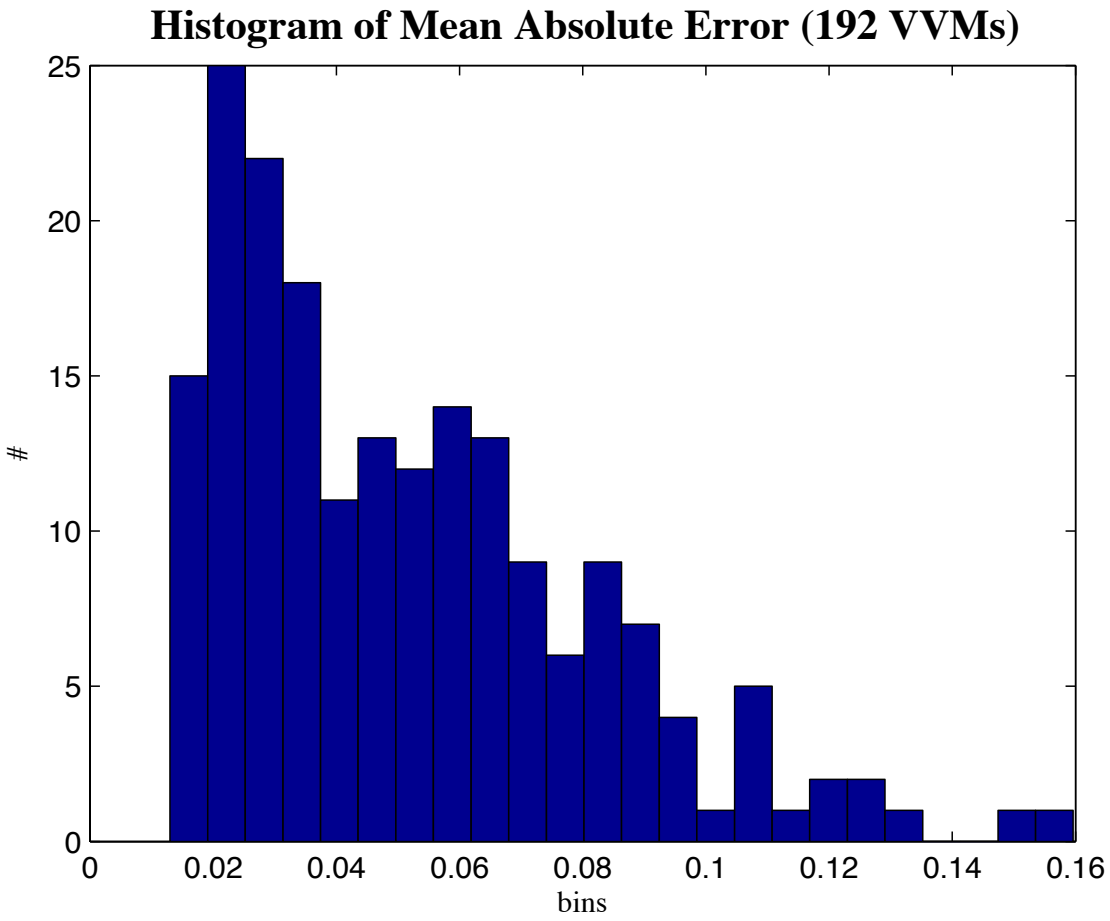


Figure 3.47: A Histogram Plot of the Mean Absolute Error from the 192 GF3 VVM Cores.

[0,1]. More than 100 of the 192 cores can achieve 4-bit output precision in accuracy while computing with 512 unary samples. Accuracy issues in this architecture are primarily attributed to mismatches in the ADC from the integrating and feedback capacitors. Additionally, coupling of the analog biases due to insufficient shielding creates decoding errors, which degrade computation accuracy.

Nonetheless, even with the low precision limitation, this chip has been successfully applied to a image processing task of DeBayering images. Figure 3.48 shows the

Figure 3.48: Image DeBayering through MATLAB and the GF3 VVM Test Chip.

results of this task done through MATLAB with 64-bit floating precision and the GF3 VVM chip done with 512 unary samples.

Measuring the energy efficiency of the core on this test chip was infeasible because of the sharing of the core power supply and the IO circuitry power supply due to limited pads. Nonetheless, the energy efficiency can be extrapolated from the GF1 VVM design, which has an identical architecture.

3.6 GF4 VVM: A 4-bit MAC Multicore Processor in 55nm CMOS

In the fourth iteration of the mixed-signal VVM architecture, a series-parallel programmable capacitor array is explored as an alternative to the full parallel unary and binary capacitor array architecture used in all of the previous designs. The revised core architecture uses the same $\Sigma\Delta$ ADC, and is also designed in the same 55nm process. By exploiting a series-parallel capacitor array topology, the integrating and

feedback capacitance were reduced to minimize energy and area costs. Moreover, the additional functionality of output scaling is integrated into the core with the implementation of programmable feedback capacitors. The fabricated test chip consist of 21 VVM cores that are each capable of computing weighted sums of 16-element vectors with 4-bit weights and variable bit precision input and outputs (maximum of 12 bits).

3.6.1 Series-Parallel Capacitor Array

The series-parallel capacitor array is a composite of 16 binary-encoded programmable capacitors interconnected for summing the individual products of the 4-bit weights and the 1-bit unary samples of the inputs. The programmable capacitor encodes each bit of the weight using a set of serially-connected capacitors, where the number of capacitors connected is scaled exponentially with the bit position. For N capacitors

in series, the equivalent capacitance C_{eq} is deduced as

$$\begin{aligned} C_{eq} &= \frac{Q}{V} \\ C_{eq} &= \frac{Q}{V_1 + V_2 + \dots V_N} \\ \frac{1}{C_{eq}} &= \frac{V_1}{Q} + \frac{V_2}{Q} + \cdots \frac{V_N}{Q} \\ \frac{1}{C_{eq}} &= \frac{1}{C_1} + \frac{1}{C_2} + \cdots \frac{1}{C_N} \\ C_{eq} &= \frac{1}{\frac{1}{C_1} + \frac{1}{C_2} + \cdots \frac{1}{C_N}}, \end{aligned} \tag{3.37}$$

where $V_1, V_2, \dots V_N$ are the voltage potentials on the capacitors, and $C_1, C_2, \dots C_N$ represent the capacitance of the individual capacitors in series.

Furthermore, these serially-connected capacitors are connected in parallel, as shown in Figure 3.49, for encoding the 4-bit weights in the programmable capacitor. The capacitance in this programmable capacitor can be expressed as,

$$C = \left(\frac{1}{\frac{1}{C_0} + \frac{1}{C_1} + \cdots \frac{1}{C_7}}\right) * w_0 + \left(\frac{1}{\frac{1}{C_8} + \frac{1}{C_9} + \cdots \frac{1}{C_{11}}}\right) * w_1 + \left(\frac{1}{\frac{1}{C_{12}} + \frac{1}{C_{13}}}\right) * w_2 + \left(C_{14}\right) * w_3, \tag{3.38}$$

where $C_0, C_1, \dots C_{14}$ are all designed with the same unit capacitance C_u, which means

$$C = \left(\frac{1}{8}C_u\right) * w_0 + \left(\frac{1}{4}C_u\right) * w_1 + \left(\frac{1}{2}C_u\right) * w_2 + \left(C_u\right) * w_3. \tag{3.39}$$

Similar to the GF3 VVM capacitor design, an APMOM topology is also adopted

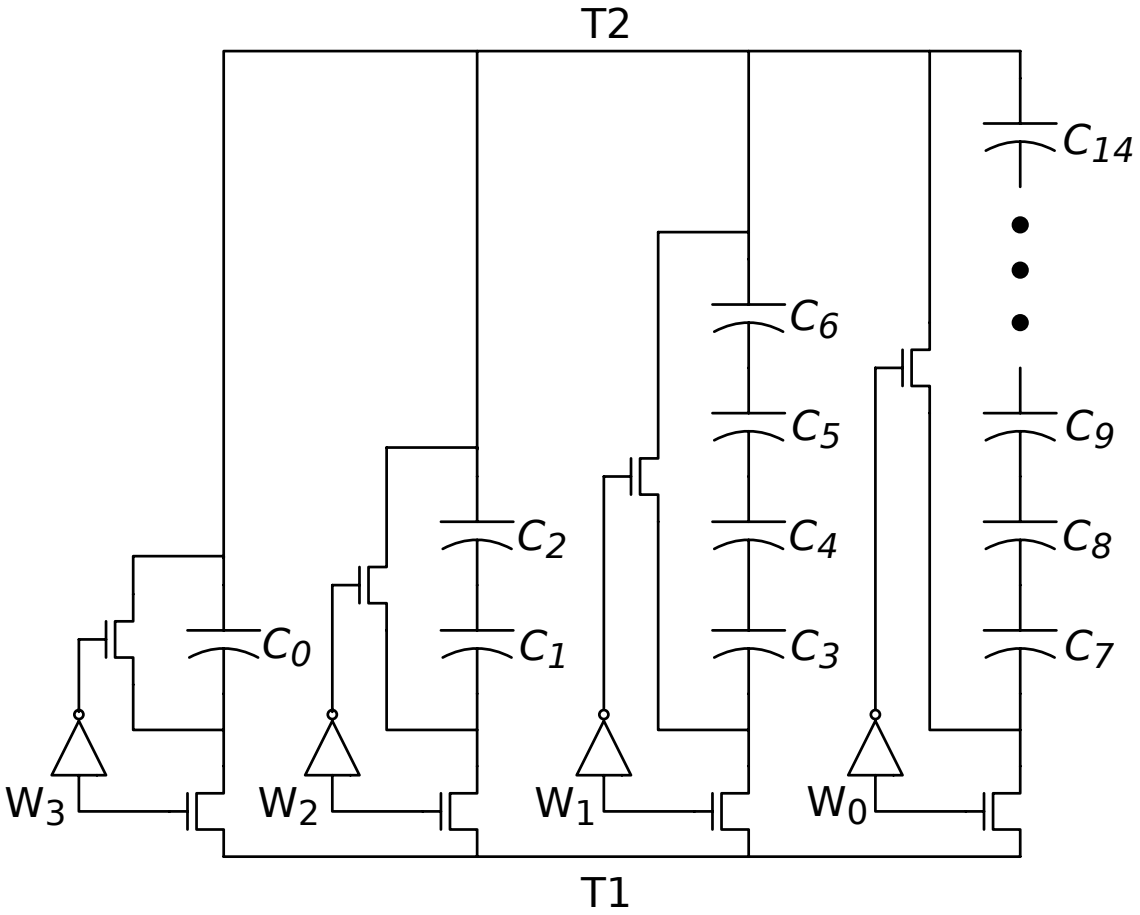


Figure 3.49: Circuit of the 4-bit Programmable Capacitor for the GF4 VVM.

for the unit capacitors in this design. Specifically, a parameterized cell from the process, is used for this capacitor. This unit capacitor was sized for 4fF capacitance, which gives a maximum capacitance of 7.5fF for the series-parallel programmable capacitor.

Furthermore, because of the minimization of the capacitance through serially connecting the unit capacitors, the connecting switches have increased parasitic effects from charge injection. In order to rectify this, the connecting transistors were sized

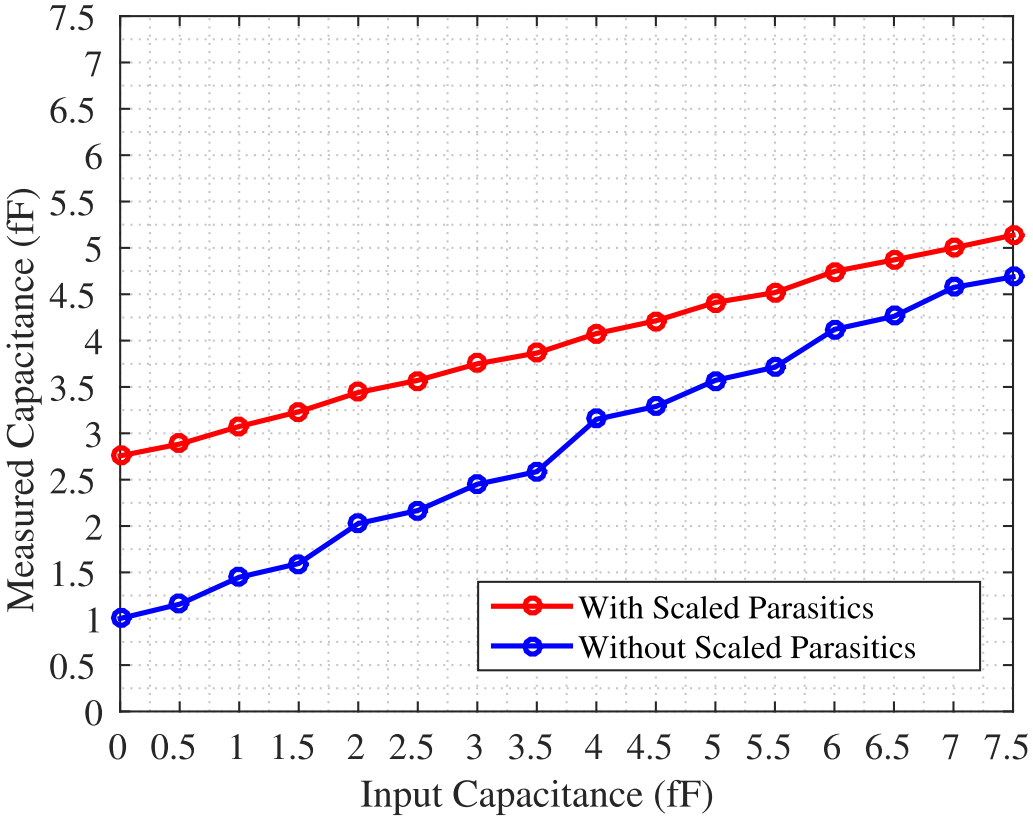


Figure 3.50: Transfer Curve of the Programmable Capacitor for GF4 VVM.

according to the number of capacitors in series. The extracted netlist of this design, with and without the additional transistor scaling to compensate for increased parasitic effects, was simulated to measure the different transfer curves. In these simulations, 10nA over 5ns is sourced to the programmable capacitor for each weight value, and the expected capacitance is evaluated against the measured capacitance. The results of this simulations can be seen in Figure 3.50. In both simulations, the measured capacitance never reaches the minimum and maximum bounds of 0 and 7.5fF respectively because of the parasitic factors. Nonetheless, there is better linear-

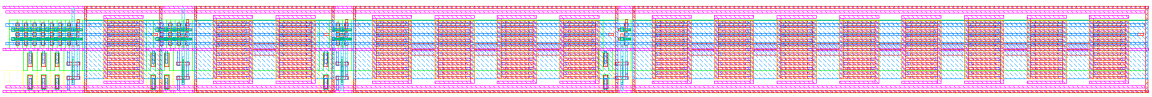


Figure 3.51: Layout of the 4-bit Programmable Capacitor for the GF4 VVM.

ity with the programmable capacitor that incorporates the transistor scaling.

Furthermore, the capacitors were shielded to mitigate parasitic coupling from adjacent digital and analog signals. The layout for the 4-bit programmable capacitor circuit can be seen in Figure 3.51.

3.6.2 Core Architecture

The core architecture for the revised VVM design is largely the same with the exception of a redesigned programmable capacitor array, and the addition of programmable feedback capacitors for output scaling. The schematic for this core circuit can be seen in Figure 3.3, and the annotated layout for this design is shown in Figure 3.52. The VVM core is laid out in 81μm by 120μm silicon area, and has the same interface as the GF3 VVM core with the exception of an additional 2-bit port for programming the feedback capacitors.

In the previous iterations of the VVM core, the feedback capacitors, which are responsible for removing or adding charge depending on the amount of charge accumulated on the integrating capacitors, were fixed to handle the maximum charge that can be accumulated on the input capacitor array. By making this feedback capaci-

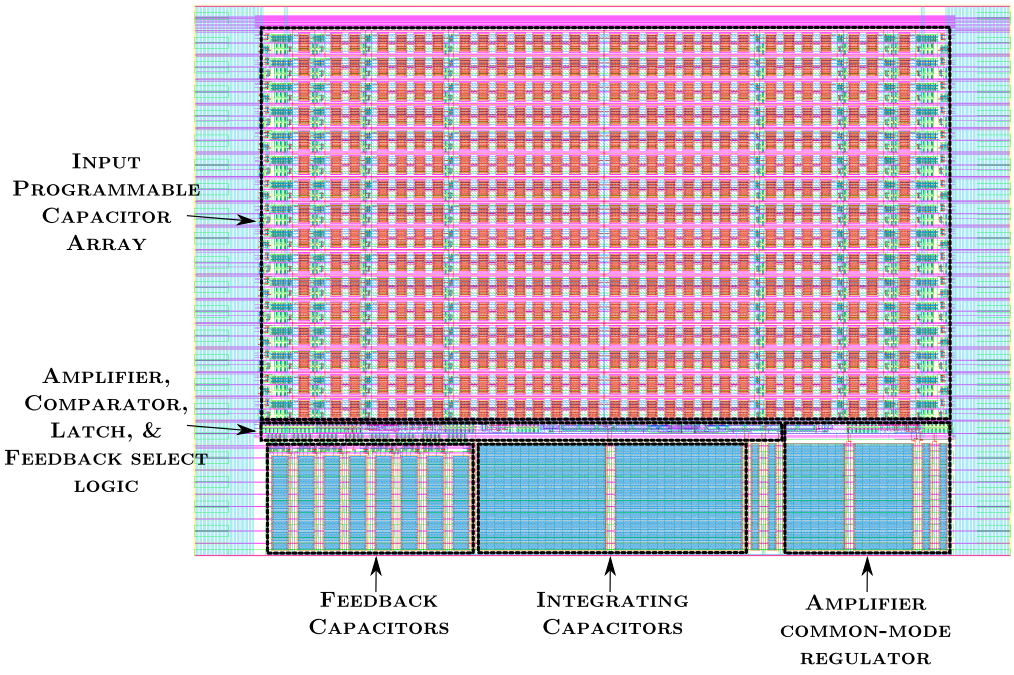


Figure 3.52: Annotated Layout of the GF4 VVM Core.

tance programmable, the feedback charge that is added or removed can be modulated, and used to scale the $\Sigma\Delta$ output. In this design, the feedback capacitors are designed to be able to inject/remove charge equivalent to 1, 2, 3, or 4 times the maximum input charge, which in turn scales the integrated $\Sigma\Delta$ output by a factor of 1, 0.5, 0.33, and 0.25 respectively. A plot of the VVM core computing weighted-sums with different feedback capacitance taken from the spice simulations can be seen in Figure 3.53. As seen in the plot, the running average of the PDM output from the $\Sigma\Delta$ ADC is scaled proportionally based on the feedback capacitance programmed. Incorporating this programmability in the feedback capacitors, allows the revised VVM core to implicitly compute an additional scaling operation.

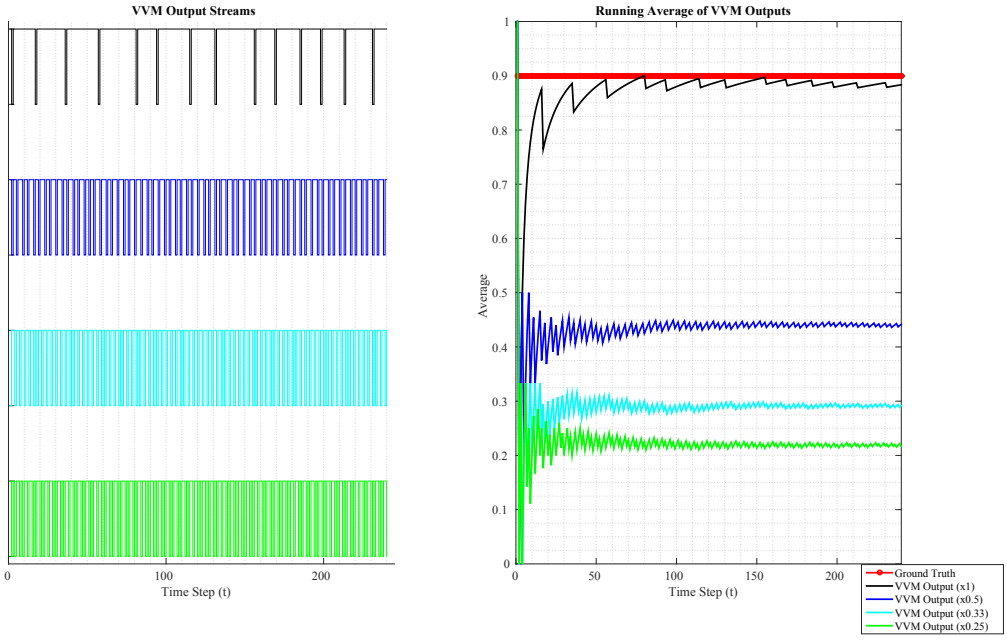


Figure 3.53: The GF4 VVM Core Output with Different Output Scaling. The plot on the left side shows the PDM output from the VVM (bipolar encoding format, where '1' is +1, and '0' is -1), and the right side plot shows the running average of the accumulated output.

For this revised VVM core, the analog bias values were modified to account for changes in the feedback and integrating capacitor. The nominal voltage and current bias values are listed in Table 3.11.

Furthermore, the simulated characteristics of this VVM core can be seen in Table 3.12. This core was simulated with a clock running at 1MHz for computing 4-bit MAC operations in 16 clock cycles. In the analog domain, the capacitor array computes the MAC operations energy costs ranging from 1.95fJ to 4.91fJ. Totally, this operation is executed using the core with an energy cost of 5.31pJ and a total efficiency of 188.32GOP/W. On the array only, this efficiency is measured at approx-

Table 3.11: Nominal Bias Values for the GF4 VVM Core.

Name	Description	Nominal Value
V_cm_i	Input common-mode voltage	0.5V
V_inp	Positive input voltage for the capacitor array	0.685V
V_inm	Negative input voltage for the capacitor array	0.315V
V_fbp	Positive feedback voltage	1.075V
V_fbm	Negative feedback voltage	0.025V
V_cm_o	Output common-mode voltage	0.5V
V_reg	Voltage bias for the operational amplifier	0.75V
I_amp	Current bias for the operational amplifier	1μA
I_comp	Current bias for the voltage comparator	0.5μA

Table 3.12: Simulated Characteristics of the GF4 VVM Core.

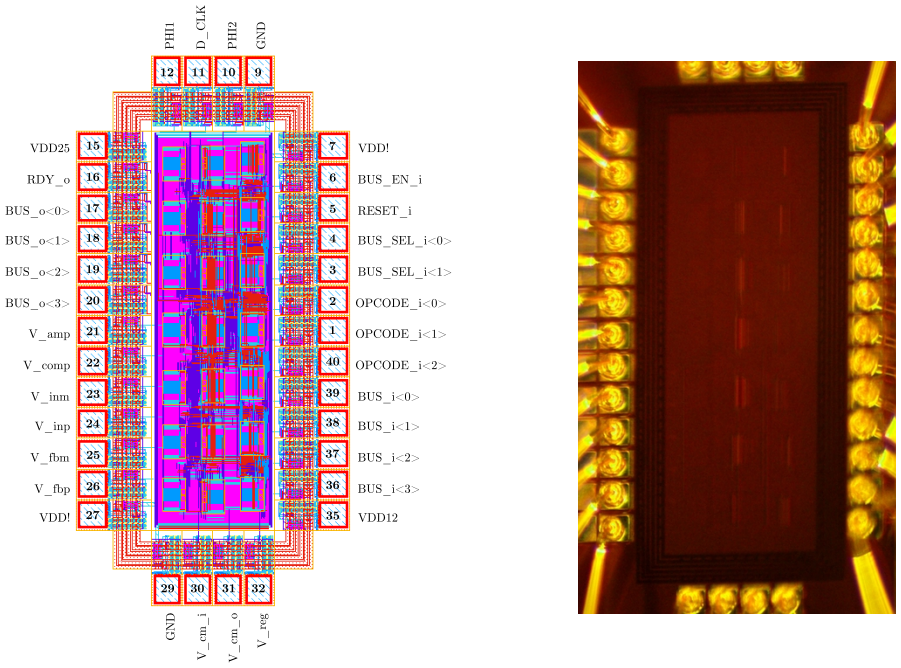
Technology	55nm CMOS
Operation	4-bit MAC
Supply Voltage	1.2V
Frequency	1MHz
Throughput	1MOPs
Power Consumption	5.33μW
Energy/Op	5.31pJ (3.43fJ on the array)
Efficiency	188.32GOP/W (291.55TOP/W for the array)

imately 291.55TOP/W.

3.6.3 Test Chip

A test chip of 21 VVM cores is fabricated in the GlobalFoundries 55nm CMOS process. These cores are laid out across 3 rows and 7 columns, and synthesized with a 3-bit opcode instruction set processor for programming cores and reading and writing to the local registers files. The micrograph and the layout of this test chip, annotated with the I/O and power ports, can be seen in Figure 3.54.

The top design with the 21 VVM cores is rotated 90 degrees in order to fit in



(a) Layout (b) Micrograph

Figure 3.54: Layout and Micrograph of the GF4 VVM Test Chip.

the pad frame. The full test chip including the pad frame is designed in 0.96mm by 1.94mm (1.86mm^2) silicon area.

Moreover, there are 34 ports for this chip — 14 digital inputs, 5 digital outputs, 9 analog inputs, and 6 power and ground ports. These ports are described in detail in Table 3.13. The digital I/O ports are used in programming the instruction set processor for writing, processing, and reading data to and from the VVM cores. In the following subsection, the chip interface is explained in more details.

Table 3.13: I/O and Power Ports for the GF4 VVM Chip.

Name	**Type**	**Width**	multicolumn1—c—**Description**
GND	POW	N/A	Ground
VDD12	POW	N/A	1.2V I/O power supply
VDD25	POW	N/A	2.5V I/O power supply
VDD!	POW	N/A	Core power supply
V_cm_o	AI	N/A	Output common-mode voltage for amplifier
V_cm_i	AI	N/A	Input common-mode voltage
V_fbm	AI	N/A	Negative feedback voltage
V_fbp	AI	N/A	Positive feedback voltage
V_inm	AI	N/A	Negative input voltage
V_inp	AI	N/A	Positive input voltage
V_amp	AI	N/A	Amplifier bias voltage
V_reg	AI	N/A	Voltage bias for regulating output common-mode
V_comp	AI	N/A	Comparator bias voltage
d_clk	DI	1	periphery clock signal
phi1	DI	1	Non-overlapping clock for the VVM cores
phi2	DI	1	Non-overlapping clock for the VVM cores
reset_i	DI	1	Global reset for cores and periphery circuitry
bus_en_i	DI	1	Enable for registering the input data bus
bus_sel_i	DI	2	Select for the 12-bit data buses
opcode_i	DI	3	Opcode for selecting bus functionality
bus_i	DI	4	Input data bus for control, address, and data signals
bus_o	DO	4	Output data bus signal
rdy_o	DO	1	Data output ready signal

3.6.3.1 Chip Interface

The GF4 VVM test chip communicates externally through an instruction-set processor that reads, processes, and writes data through 4-bit data buses. Aside from the clock ports, there are 7 ports used in this processor — 1-bit reset, 1-bit enable, 2-bit bus select, 3-bit opcode, 4-bit input data bus, 4-bit output data bus, and a 1-bit output ready. This processor runs on the *d_clk* clock signal, which was designed to run at 100MHz. The *reset_i* signal is the active-high reset that not only resets all

Table 3.14: Instruction Set for GF4 VVM Test Chip.

Opcode	Description
0	Sets the 2-bit row address (3 rows).
1	Sets the 3-bit column address (7 columns).
2	Sets the address for the weight and input register.
3	Sets the output precision for the VVM cores (range of 5-12bits).
4	Bit 0: Sets the feedback capacitance. Bit 1: Sets the feedback capacitance. Bit 2: Selects the weight or input registers. Bit 3: Sets the mode to run the VVM cores
5	Bit 0: Sets the reset for a VVM core. Bit 1: Write enable for weight and input registers. Bit 2: Write enable for offset registers. Bit 3: Process enable for VVM cores.
6	Writes the 4-bit input to part of the 12-bit data bus

registers, but also puts the processor in the reset state. Moreover, this processor only functions when the *bus_en_i* is asserted. The *bus_sel_i* signal, which is the bus select, programs which 4-bits of the full 12-bit data bus is written and read when the enable is asserted. This 4-bit bus width is adapted for this design due to pin limitations.

The main instructions for this processor is selected by the 3-bit opcode, which is detailed in Table 3.14. The first two instructions are used to select the VVM cores based on the row and the column. The next instruction programs the address for the weight and input register files. After that, the next 3 instructions are used for configuring the cores, which includes setting the output precision , setting the feedback capacitance, writing to respective registers, resetting the cores, and enabling the cores for processing. Additionally, by programming the mode, the output readout can be set to either a direct readout of the $\Sigma\Delta$ outputs or the integrated output from

the counters. The important procedures for this test chip are detailed as follow:

- **Reset VVM Core Procedure**

  1. Set the opcode to **0**, and set the row address on the data bus.
  2. Set the opcode to **1**, and then set the column address on the data bus.
  3. Set the opcode to **5**, and then set the data bus to **"0001"** for at least one clock cycle of the ϕ clock to reset the core.
  4. After reset, set data bus to **"0000"** for another cycle of theϕ to take the core out of reset.

- **Set Feedback Capacitance Procedure**

  1. Set the opcode to **0**, and set the row address on the data bus.
  2. Set the opcode to **1**, and then set the column address on the data bus.
  3. Set the opcode to **4**, and set the feedback capacitance select with the first two bits in the data bus. The feedback capacitance select modifies the output scaling, and the exact mapping is detailed in Table 3.15.

Table 3.15: Feedback Capacitance Select Mapping for the GF4 VVM Design

Feedback Cap. Select	**Scaling Factor**
0	1.00
1	0.50
2	0.33
3	0.25

- **Set Output Precision Procedure**

  1. Set the opcode to **0**, and set the row address on the data bus.
  2. Set the opcode to **1**, and then set the column address on the data bus.
  3. Set the opcode to **3**, and set the output precision select. Table 3.16 shows the mapping of this select to the precision.

Table 3.16: Output Precision Select Mapping for the GF4 VVM Design.

Output Prec. Select	**# of Samples (Precision)**
0	32 (5bits)
1	64 (6bits)
2	128 (7bits)
3	256 (8bits)
4	512 (9bits)
5	1024 (10bits)
6	2048 (11bits)
7	4096 (12bits)

- **Load Weight Procedure**

  1. Set the opcode to **0**, and set the row address on the data bus.
  2. Set the opcode to **1**, and then set the column address on the data bus.
  3. Set the opcode to **2**, and selects which weight to set on the data bus.
  4. Set the opcode to **4**, and set the 3rd bit to **"1"**, while preserving the mode, and the feedback capacitance select values on the data bus.
  5. Next, set the opcode to **5**, and then set the data bus to **"0010"** to set the write bit high.

6. Then, set the opcode to **6**, and then set the 4-bit weight on the data bus.
7. Finally, set the opcode to **5**, and then set the data bus to **"0000"** to set write low.

- **Load Input Procedure**

1. Set the opcode to **0**, and set the row address on the data bus.
2. Set the opcode to **1**, and then set the column address on the data bus.
3. Set the opcode to **2**, and selects which weight to set on the data bus.
4. Set the opcode to **4**, and set the 3rd bit to **"0"**, while preserving the mode, and the feedback capacitance select values on the data bus.
5. Next, set the opcode to **5**, and then set the data bus to **"0010"** to set the write bit high.
6. Then, set the opcode to **6**, and then set the input over 4-bit segments using the data bus and the bus_sel_i signals (**"00"** sets the first 4 bits, **"01"** set the middle 4 bits, and **"10"** for the last 4 bits in the 12-bit data).
7. Next, set the opcode to **5**, and then set the data bus to **"0000"** to set write low.

- **Load Offset Procedure**

1. Set the opcode to **0**, and set the row address on the data bus.
2. Set the opcode to **1**, and then set the column address on the data bus.

3. Set the opcode to **2**, and selects which weight to set on the data bus.

4. Next, set the opcode to **5**, and then set the data bus to **"0100"** to set the write bit high. The write bit must be held high for at least 1 clock cycle of the ϕ clock.

5. Then, set the opcode to **6**, and then set the input over 4-bit segments using the data bus and the *bus_sel_i* signals (**"00"** sets the first 4 bits, **"01"** set the middle 4bits, and **"10"** for the last 4 bits in the 12-bit data). Each 4-bit data loaded must be held for at least 1 clock cycle of the ϕ clock.

6. Next, set the opcode to **5**, and then set the data bus to **"0000"** to set write low. The write bit must be held low for at least 1 clock cycle of the ϕ clock.

- **Enable Processing Procedure**

1. Set the opcode to **0**, and set the row address on the data bus.

2. Set the opcode to **1**, and then set the column address on the data bus.

3. Set the opcode to **5**, and set the process enable as the 4th bit on the data bus.

- **Read Output Procedure**

1. Set the opcode to **0**, and set the row address on the data bus.

2. Set the opcode to **1**, and then set the column address on the data bus.

However, if mode is set to **1**, then skip setting the column, and read outputs for all columns in the selected row on the data bus.

3. If in mode **0** and rdy_o signal is high, then read the output of the selected VVM in 4-bit segments from the output data bus. (Setting bus_sel_i to **"00"** reads the first 4 bits, **"01"** reads the middle 4 bits, and **"10"** reads last 4 bits of the 12-bit output data).

3.7 GF5 VVM: An 8-bit MAC Multicore Processor in 55nm CMOS

In the fifth iteration of the mixed-signal VVM design, an alternative architecture that exploits stochastic data representation and computation is explored. In this design, products are computed temporally in the stochastic domain using 1-bit mixed-signal multipliers . The aggregate of these products as charge is summed implicitly through interconnection of these mixed-signal multiplier. Furthermore, a $\Sigma\Delta$ ADC, similar to the ones implemented in the previous VVM designs, is used in converting this sum of products back into a digital value. By exploiting a stochastic data representation, the energy and area cost for the revised VVM cores were minimized, and the computational efficiency was increased. Not only was this validated through simulations, but also through experimental results from a fabricated 55nm test chip with multiple VVM cores.

3.7.1 Mixed-Signal Stochastic Multiplier

Capitalizing on the strengths of stochastic computation, a mixed-signal circuit is proposed that computes the product of two inputs probabilistically as charge on a capacitor using the circuit shown in Figure 3.1. Analogous to an AND logic gate, this circuit computes the unsigned product of stochastic numbers from the relationship of

$$Y = (WX) \cdot Q_u, \tag{3.40}$$

where the capacitor must be connected through the weight W and a voltage potential through the input X must be asserted for a unit charge Q_u to be stored on the capacitor.

Moreover, this architecture was adapted to signed products with 2's complement input stochastically encoded with a 1-bit bipolar format ('0' = -1 and '1' = +1), and a sign-magnitude weight with a sign and magnitude component signified with S_W and W respectively. Using the circuit shown in Figure 3.55, the product is then computed as

$$Y = W\Big(\overline{(X \oplus S_W)}(Q_u) + (X \oplus S_W)(-Q_u)\Big), \tag{3.41}$$

which simply shows that a unit charge Q_u will be added when the magnitude of the weight is '1', and this charge will be positive if the signed input and the sign of the weight match, otherwise, it will be negative. Similar to the unsigned multiplier, the magnitude of the weight W connects or disconnects the capacitor, but the sign of the

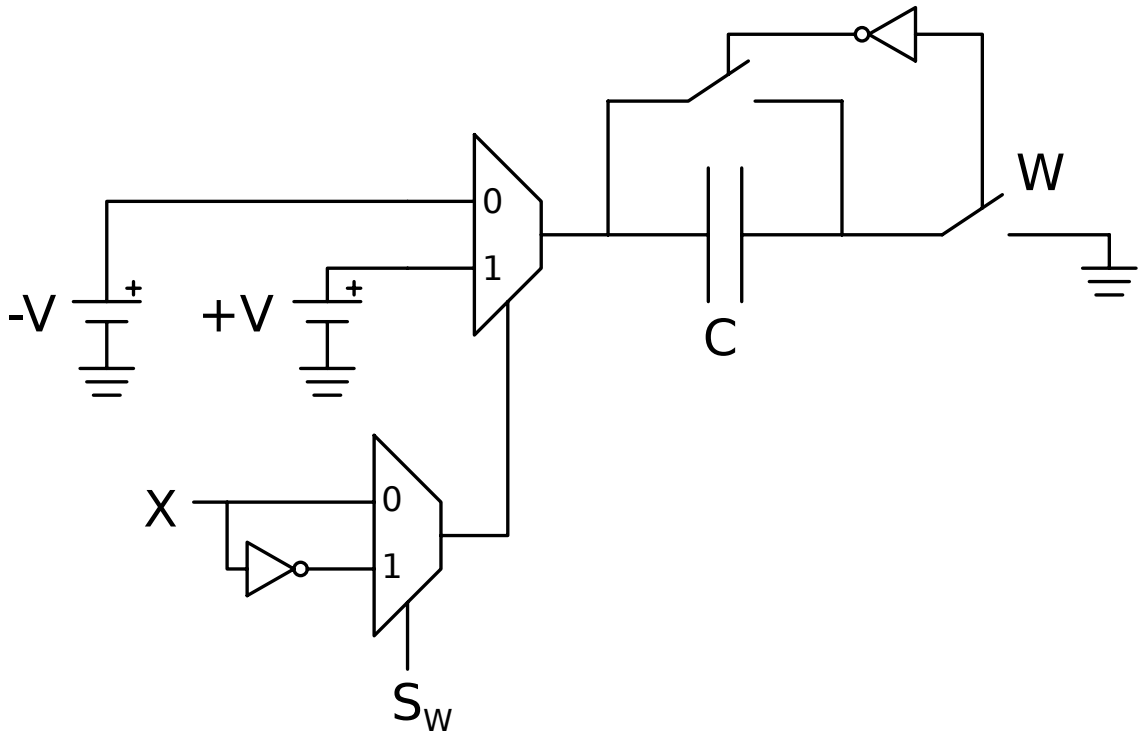


Figure 3.55: Mixed-Signal 1-bit Multiplier for Signed Products.

weight S_W now selects between the non-inverted or inverted input because capacitance can not be negated. Hence by adding an additional multiplexer to the unsigned multiplier a signed multiplier is implemented for four-quadrant multiplication.

The layout of the 1-bit mixed-signal multiplier for signed products in the 55nm process is shown in Figure 3.56. A 3D design topology is adapted for this programmable capacitor cell, where the unit capacitor is designed as interdigitated metal fingers across 3 metals layers (M4 – M6) on top of the routing logic and wires for programming the cell. By vertically integrating the capacitor with this routing and periphery circuitry, this cell was laid out in 3.8μm by 4μm (14.4μm^2) area. The unit

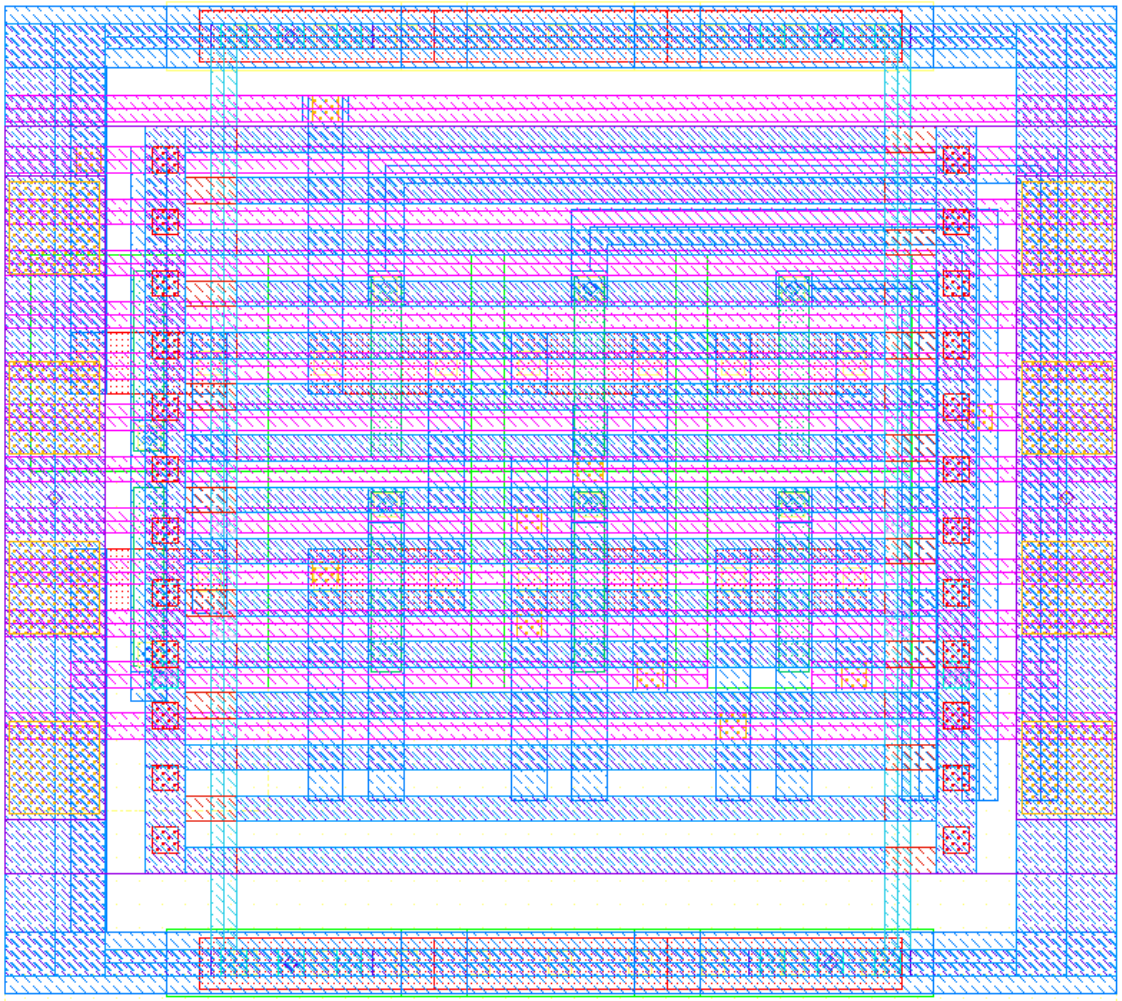


Figure 3.56: Layout of the Mixed-Signal 1-bit Multiplier for Signed Products.

capacitor in this design was sized for 10fF capacitance.

Furthermore, as opposed to connecting one of the terminals to a ground terminal as shown in the multiplier circuit in Figure 3.55, that terminal for each of the multipliers is instead connected together. By interconnecting these plates together, the array of multiplier can then implicitly compute the sum of the product stochastically in time. As opposed to a spacial encoding, the weights are encoded temporally, which reduces

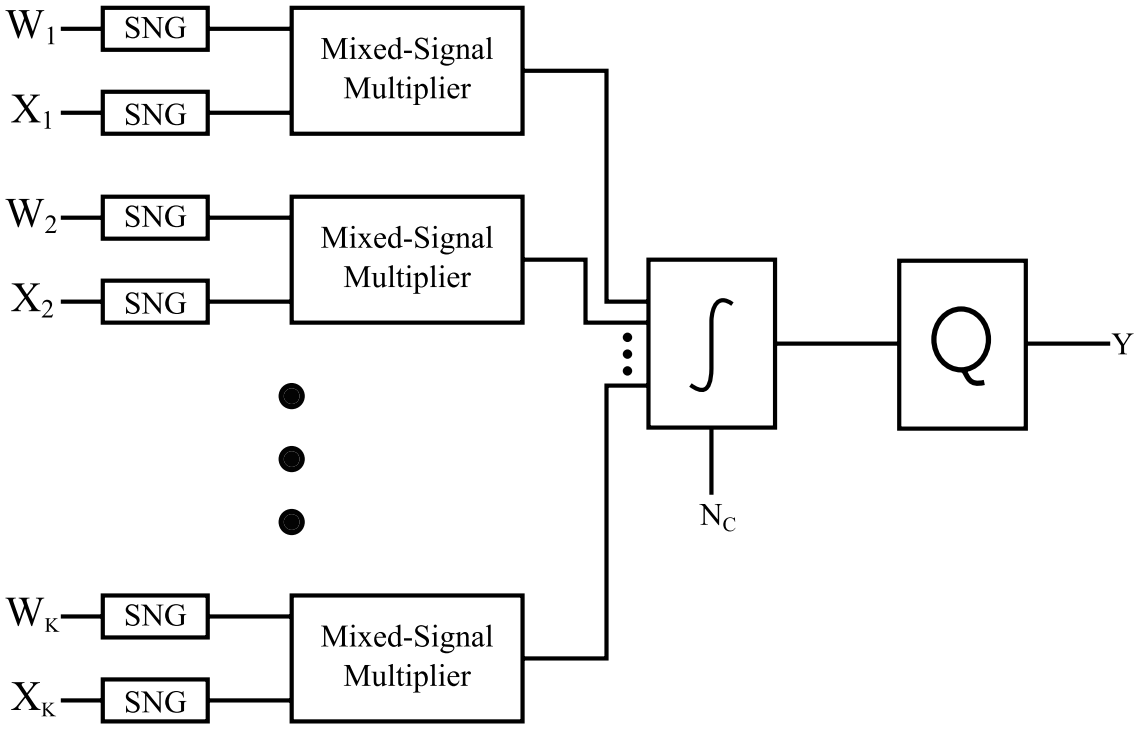


Figure 3.57: Block Diagram of the GF5 VVM Core.

the number of capacitors in the VVM architecture to the number of elements in the individual (weight and input) vectors.

3.7.2 Core Architecture

By cascading a set of mixed-signal stochastic multipliers, the inner product of a weight vector W and an input vector X can be computed as charge, and then decoded to a fixed-point binary output through an ADC. A block diagram of this architecture can be seen in Figure 3.57. Using a stochastic number generator, as shown in Figure 2.3, M-bit weights and inputs W_i and X_i of vectors W and X

respectively, can be encoded as an aggregate of stochastic numbers as

$$X_i = \left(\frac{2^M}{N}\right)\sum_{t=0}^{N-1} x_i^t \quad \text{and} \quad W_i = \left(\frac{2^M}{N}\right)\sum_{t=0}^{N-1} w_i^t, \tag{3.42}$$

where $w_i^{(t)}$ and $x_i^{(t)}$ are stochastic numbers of W_i and X_i respectively sampled at time t. Similar to Equation 3.40, the product, computed as charge $Q^{(t)}$ at time t integrated from the multipliers, is given as

$$Q^{(t)} = \sum_{i=0}^{K-1}(w_i^{(t)} \cdot x_i^{(t)}) \cdot Q_u, \tag{3.43}$$

and the total charge Q integrated for all N time periods is

$$\begin{aligned} Q &= \sum_{t=0}^{N-1} Q^{(t)} \\ Q &= \sum_{t=0}^{N-1}\sum_{i=0}^{K-1}(w_i^{(t)} \cdot x_i^{(t)}) \cdot Q_u \\ Q &= \left(\frac{N}{2^M}\right)\sum_{i=0}^{K-1}(W_i \cdot X_i) \cdot Q_u, \end{aligned} \tag{3.44}$$

where the number of time periods N (integration count) scales the precision after normalizing the output.

Furthermore, using a fully-differential switched-capacitor $\Sigma\Delta$ modulator, the full design is constructed as the architecture shown in Figure 3.58. The mixed-signal multipliers are implemented as circuit shown in Figure 3.55 for implementing signed arithmetic in the stochastic domain. A total of 9 mixed-signal multipliers are incorpo-

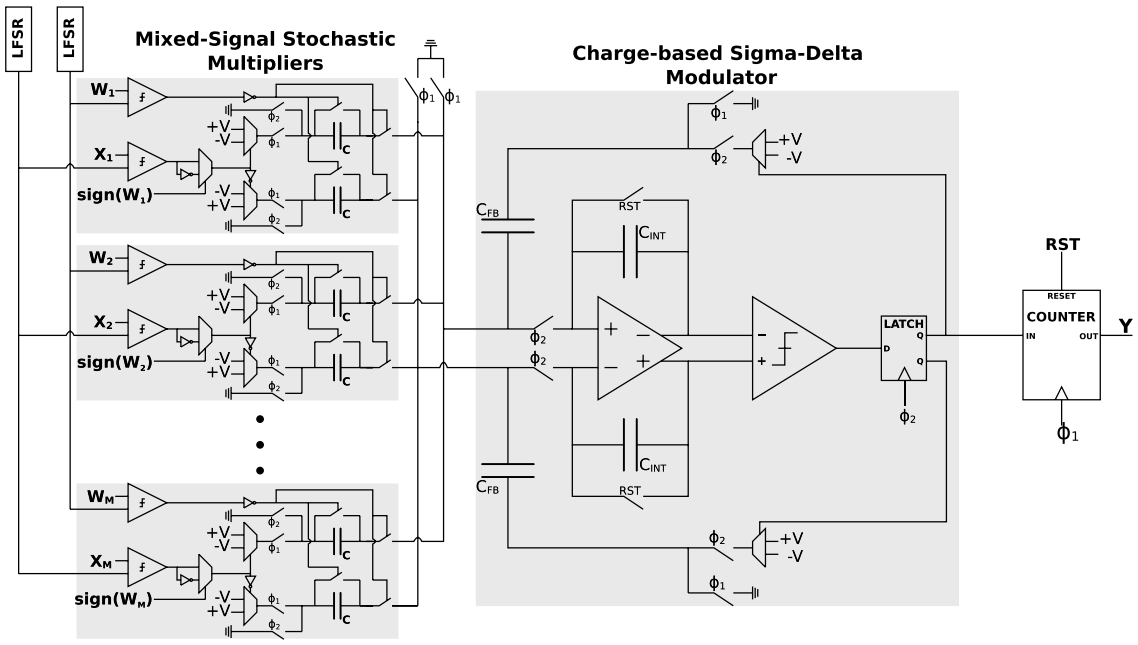


Figure 3.58: GF5 VVM Core Architecture.

rated in this design for doing 9-element vector-vector multiplications. Moreover, since each multiplier are independent of each other within the core, the LFSRs, which are the random number generators used for the stochastic number generation, are shared.

For the charge-to-digital conversion, the feedback capacitors are sized such that feedback charge equates to the maximum charge from the array of mixed-signal multipliers. The integrating capacitors are sized such that the LSB on the output voltage is greater than the V_{rms} due to kTC noise. The output of the $\Sigma\Delta$ ADC, which selects the feedback charge to integrate to the array, is summed with an up-down counter, and the precision of this count is scaled with the integration count N.

The annotated layout of this VVM core can be seen in Figure 3.59. The input capacitors for the mixed-signal multipliers, the feedback capacitors, and the integrating

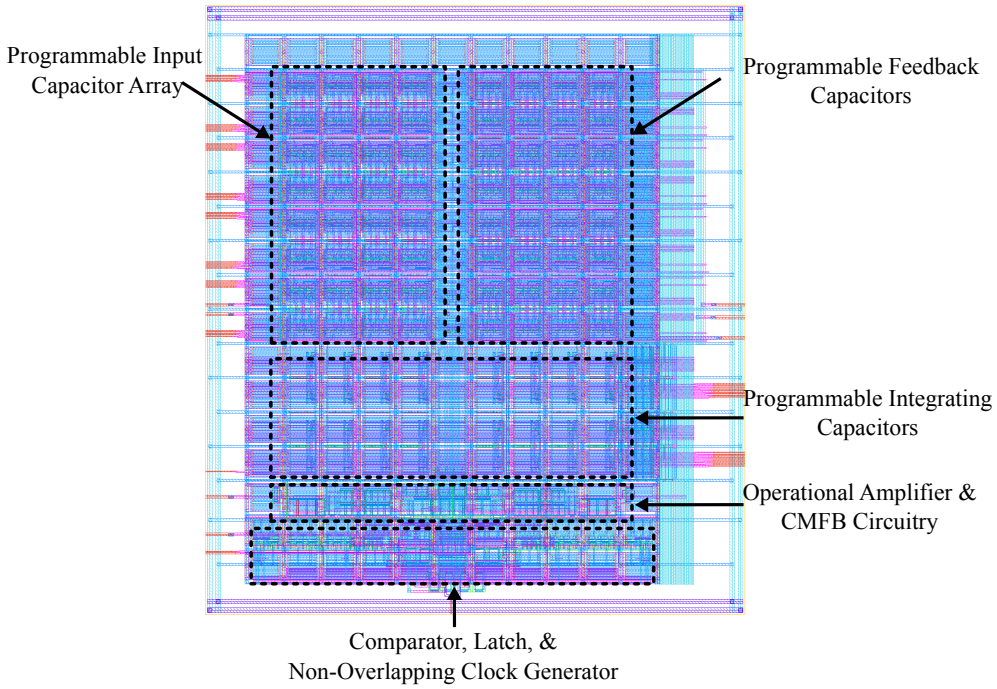


Figure 3.59: Layout of the GF5 VVM Core.

capacitors are designed with the same unit capacitors in order to minimize mismatch in the core. Additionally, the feedback and integrating capacitor are designed as programmable array for computing VVMs with less number of elements. This core, can be configured to compute as few as 1-element VVMs (a single product) to up to 9-element VVMs. The non-overlapping phi1 and phi2 clocks are locally generated on the core with the periphery logic displayed at the bottom of the core. The full VVM core measures 52μm by 64μm (3712μm^2) in area, and compared to the GF4 VVM core, as shown in Figure 3.60, has a 62% area reduction. Not only was the area cost reduced, but the revised core was also designed to handle higher precision computations with up to 8-bit weights and inputs.

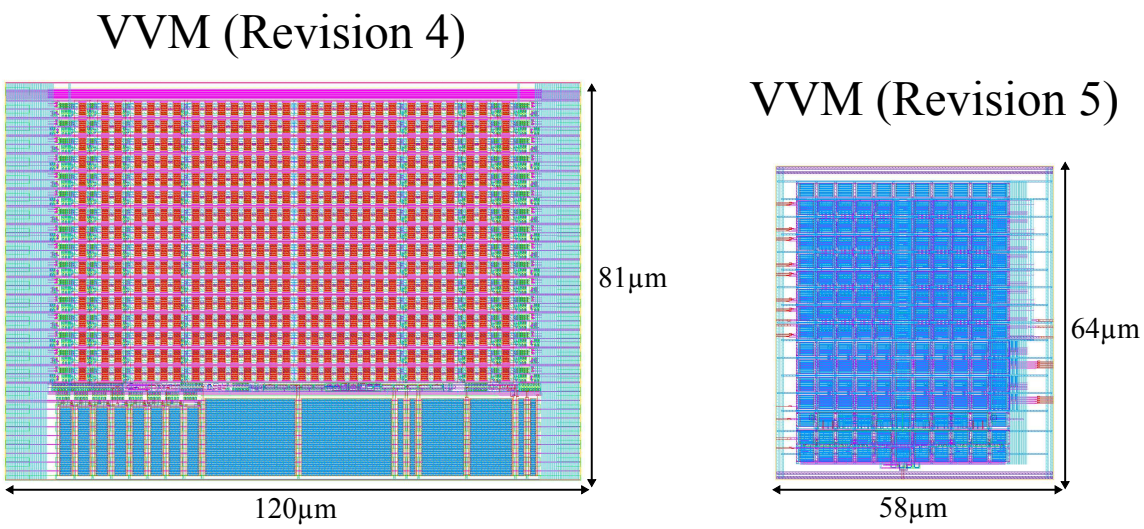


Figure 3.60: Layout Comparison of the GF4 VVM Core and the GF5 VVM Core.

3.7.3 Test Chip

For a test chip fabricated in the GlobalFoundries 55nm CMOS process, 24 GF5 VVM cores were laid out and fabricated for validating this mixed-signal architecture and measuring characteristics. These cores were synthesized with register files for storing the weights and input locally, and counters for integrating the 1-bit $\Sigma\Delta$ outputs from the core. In conjunction to this, a simple 6-instruction processor is implemented in this design for writing, processing, and reading data to and from the cores. Figure 3.61 shows the layout of the top design for the GF5 VVM. The full design with the 24 VVM cores was designed in 0.7mm by 0.7mm (0.49mm^2) area.

The full GF5 test chip is comprised of multiple designs, including GF5 VVM, that are incorporated together in one pad frame. Figure 3.62 shows an annotated layout view of this test chip in the pad frame. In this test chip specifically, there

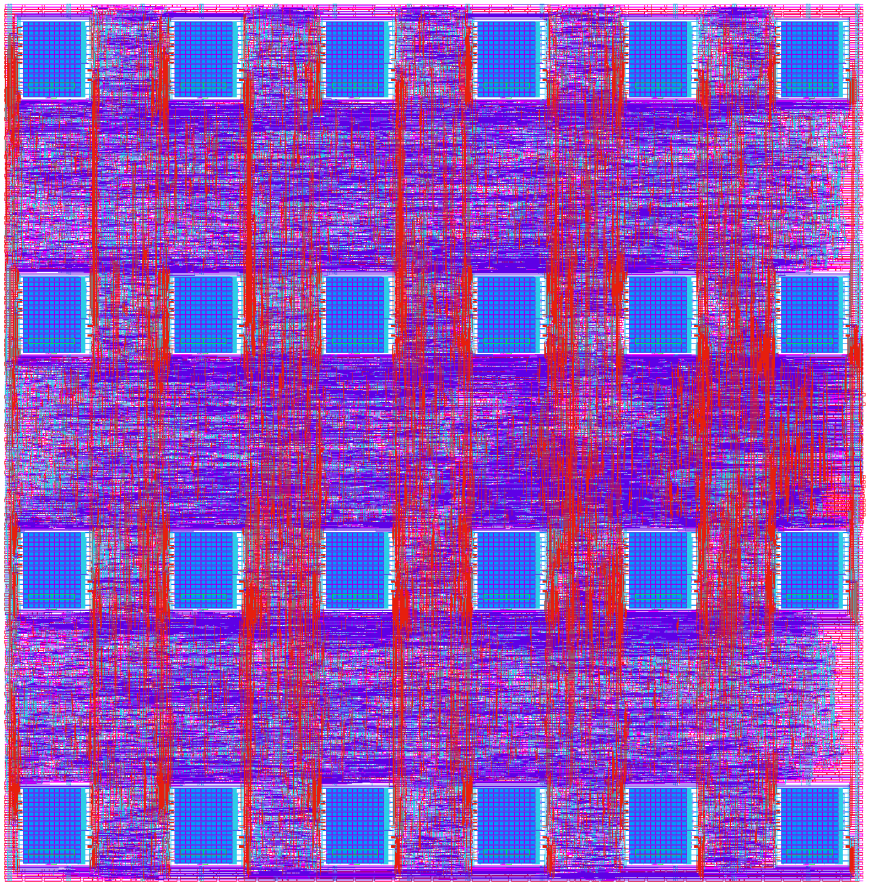


Figure 3.61: Layout of the GF5 VVM Multicore Design.

is a charge injection device (CID) processor, which is charge-based multiprocessor for computing 1-bit products. Additionally, there is an associative memory (ACM) processor, which uses compute-in memory primitives for computing logical operations. There is also a morphological processor (MORPHO), a phase-locked loop (PLL) for clock frequency synthesis, and a mixed-signal implementation of an integrate and fire array transceiver (IFAT). Finally, there is the multicore VVM processors, which has been described extensively in the earlier section. The specific I/O ports for the VVM design and the pin mapping is detailed in Table 3.17.

Table 3.17: I/O and Pin Mapping for the GF5 VVM.

Pin	Port	Type	Description
VSS	VSS	POW	Ground
VDD_I	VDD_I	POW	I/O internal power supply
VDD_E	VDD_E	POW	I/O external power supply
VDD_VVM_GF5	VDD!	POW	GF5 VVM Power Supply
PAD_io_6	V_inp_io	AI	Positive input voltage bias
PAD_io_7	V_inm_io	AI	Negative input voltage bias
PAD_io_8	V_fbp_io	AI	Positive feedback voltage bias
PAD_io_9	V_fbm_io	AI	Negative feedback voltage bias
PAD_io_10	V_cmi_io	AI	Common-mode input voltage bias
PAD_io_11	V_cmo_io	AI	Common-mode output voltage bias
PAD_io_12	V_b_io	AI	Common-mode output regulation voltage bias
PAD_io_15	I_amp_io	AI	Op-amp current bias
PAD_io_16	I_cmp_io	AI	Comparator current bias
PAD_i_0	clk_i	DI	Clock signal
PAD_i_1	rst_i	DI	Reset signal
PAD_i_2	en_i	DI	Enable signal
PAD_i_3	bus_i[0]	DI	Input data bus signal
PAD_i_4	bus_i[1]	DI	Input data bus signal
PAD_i_5	bus_i[2]	DI	Input data bus signal
PAD_i_6	bus_i[3]	DI	Input data bus signal
PAD_i_7	bus_i[4]	DI	Input data bus signal
PAD_i_8	bus_i[5]	DI	Input data bus signal
PAD_i_9	bus_i[6]	DI	Input data bus signal
PAD_i_10	bus_i[7]	DI	Input data bus signal
PAD_i_11	bus_i[8]	DI	Input data bus signal
PAD_i_12	bus_i[9]	DI	Input data bus signal
PAD_i_13	bus_i[10]	DI	Input data bus signal
PAD_i_14	bus_i[11]	DI	Input data bus signal
PAD_i_15	bus_i[12]	DI	Input data bus signal
PAD_i_16	bus_i[13]	DI	Input data bus signal
PAD_i_17	bus_i[14]	DI	Input data bus signal
PAD_i_18	bus_i[15]	DI	Input data bus signal
PAD_i_19	bus_sel_i	DI	Select signal for the output data bus. (VVM output or address pointer)
PAD_i_20	addr_i[0]	DI	Block and data address signal
PAD_i_21	addr_i[1]	DI	Block and data address signal

PAD_i_22	addr_i[2]	DI	Block and data address signal
PAD_i_23	addr_i[3]	DI	Block and data address signal
PAD_i_24	addr_i[4]	DI	Block and data address signal
PAD_i_25	addr_i[5]	DI	Block and data address signal
PAD_i_26	addr_sel_i	DI	Selects between block and data addresses
PAD_i_27	opcode_[0]	DI	Opcode bit for instruction decoding
PAD_i_28	opcode_[1]	DI	Opcode bit for instruction decoding
PAD_i_29	opcode_[2]	DI	Opcode bit for instruction decoding
PAD_o_0	bus_o[0]	DO	Output data bus signal
PAD_o_1	bus_o[1]	DO	Output data bus signal
PAD_o_2	bus_o[2]	DO	Output data bus signal
PAD_o_3	bus_o[3]	DO	Output data bus signal
PAD_o_4	bus_o[4]	DO	Output data bus signal
PAD_o_5	bus_o[5]	DO	Output data bus signal
PAD_o_6	bus_o[6]	DO	Output data bus signal
PAD_o_7	bus_o[7]	DO	Output data bus signal
PAD_o_8	bus_o[8]	DO	Output data bus signal
PAD_o_9	bus_o[9]	DO	Output data bus signal
PAD_o_10	bus_o[10]	DO	Output data bus signal
PAD_o_11	bus_o[11]	DO	Output data bus signal
PAD_o_12	bus_o[12]	DO	Output data bus signal
PAD_o_13	bus_o[13]	DO	Output data bus signal
PAD_o_14	bus_o[14]	DO	Output data bus signal
PAD_o_15	rdy_o	DO	Output ready signal

The VVM cores operate on a frequency-divided clock from a main clock clk_i that runs at 50MHz. The voltage and current biases for the cores are programmed to the nominal values displayed in Table 3.18.

3.7.3.1 Chip Interface

The top design with the 24 VVM cores in the GF5 test chip interface externally through a set of digital ports, which are:

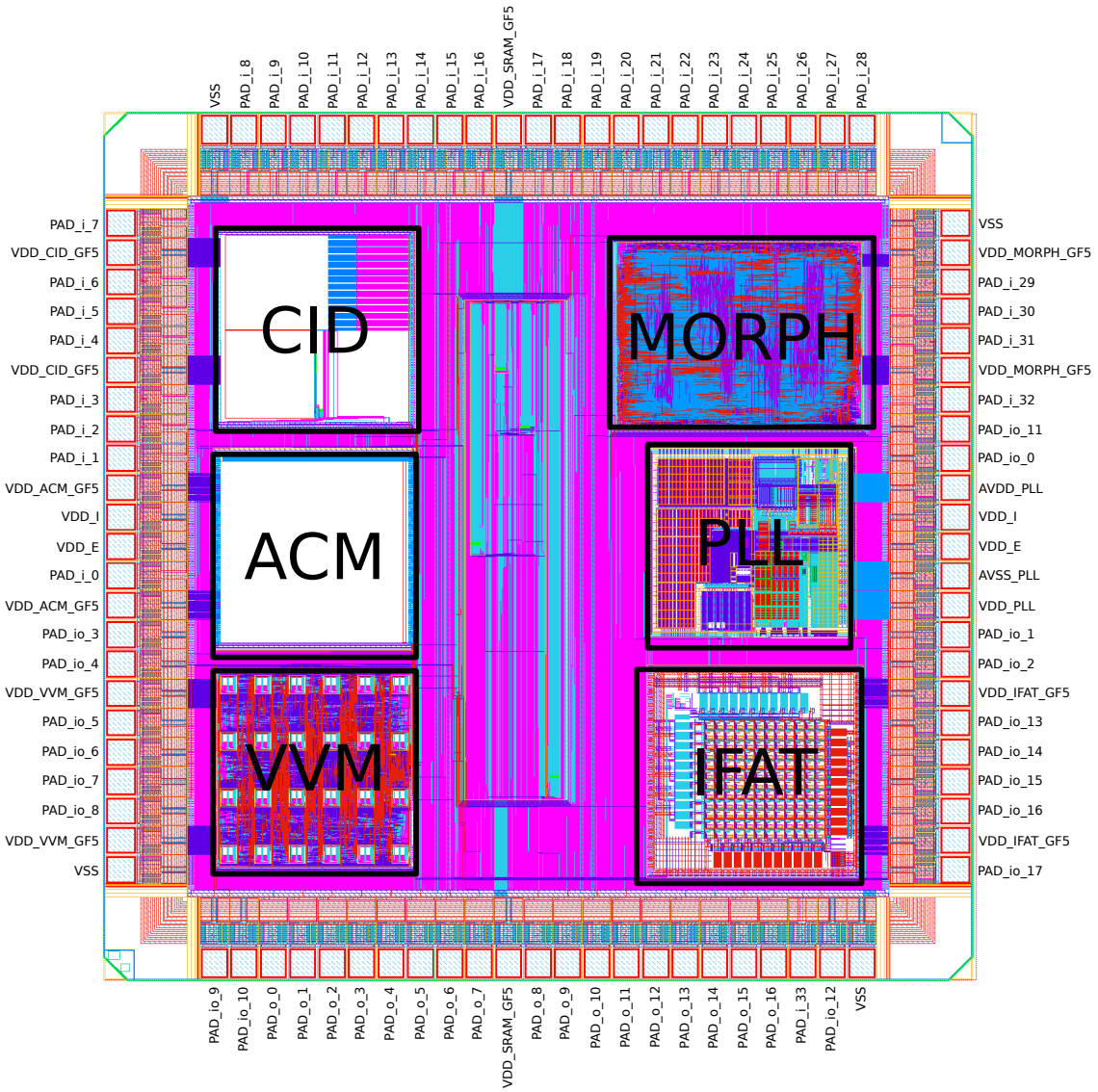


Figure 3.62: Annotated Layout of the GF5 Test Chip.

- a 1-bit reset,
- a 1-bit enable,
- a 1-bit bus select,
- a 1-bit address select,
- a 6-bit address,

Table 3.18: Nominal Bias Values for the GF5 VVM Core.

Name	Nominal Value
VDD	1.2V
V_cmi_io	0.3V
V_inp_io	0.35V
V_inm_io	0.25V
V_fbp_io	0.35V
V_fbm_io	0.25V
V_cmo_io	0.3V
V_b_io	0.53V
I_amp_io	10μA
I_cmp_io	2μA

Table 3.19: Instruction Set for GF4 VVM.

Opcode	Description
0	Sets the control word.
1	Write weight or input.
2	Write sign data for weight.
3	Writes LFSR mask.
4	Writes initial value (offset) for the VVM cores
5	Writes select word for all VVM cores

- a 16-bit input data bus,
- a 16-bit output data bus,
- and a 1-bit output ready.

Using these ports and the instruction set processor, the VVM cores can be configured and data can be loaded to the register files.

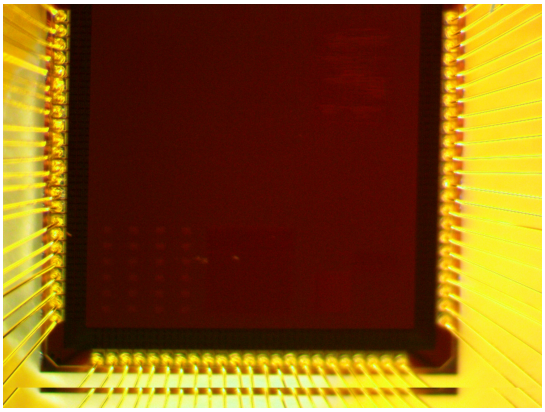
The instruction set processor is comprised of 6 instructions that are detailed in Table 3.19. The control word configures a local core reset rst, a process enable $proc\_en$, an output precision select $prec\_sel$, resets for the LFSRs $lfsr1\_rst$ and $lfsr2\_rst$, and an enable for the LFSRs $lfsr\_en$. The LFSRs, which are used for the stochastic

encoding of the weights and inputs, are loaded with different seeds by asserting the appropriate reset after enabling the shift registers for a fixed number of clock cycles. The LFSRs can be used to generate different maximally length sequences by encoding the appropriate mask using the 4th instruction (opcode = 3). This mask controls which bits of the registers are logically combined with XOR gates for the feedback tap. Additionally, the precision select in the control word selects an output precision ranging from 6 to 13 bits, where the number of unary samples integrated from the VVM cores range from 64 to 8192 samples. Furthermore, the select word is used to program the feedback and integrating capacitance, which is based on the number of elements in the vector for the VVM computation. This select word can select between a scalar product of 1 weight and 1 input to a 9-element vector-vector multiplication.

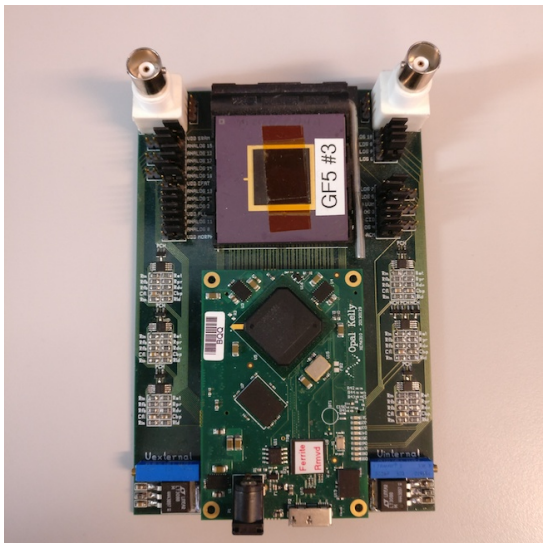
3.7.3.2 Setup

The 3mm by 3mm test chip was wirebonded to a 145 pin grid array (PGA) package and then mounted on a custom PCB board with on board DACs and an interface to Spartan 3 and 6 FPGAs. Figure 3.63 shows the micrograph of the wirebonded chip and the full test board. The test board was designed as a platform for testing any of the individual designs, and the individual power supplies and analog biases can be connected or disconnected using the jumpers shown on the test board.

A Spartan 6310 FPGA is used as a driver for testing the GF5 test chip. On this FPGA, a first in, first out (FIFO) block is synthesized for loading instructions to send



(a) Micrograph



(b) Test Board

Figure 3.63: Chip Micrograph and Test Board for the GF5 Test Chip.

and execute with the chip, and another FIFO block is synthesized for storing output data from the chip. Using the Opal Kelly FrontPanel interface, data is transferred between the FPGA and the PC through a USB port. Furthermore, the behavioral model of the GF5 VVM design was also synthesized and implemented on this FPGA in order to compare experimental results.

3.7.3.3 Results

Both the emulated and the fabricated VVM cores were tested for computing fixed-point arithmetic, specifically 8-bit signed products, with different integration counts N ranging from 64 (6 bits) to 8192 (13 bits). A plot of the normalized absolute error of the computation result from both the emulated and fabricated design evaluated against the computation done in MATLAB with double floating precision is shown in

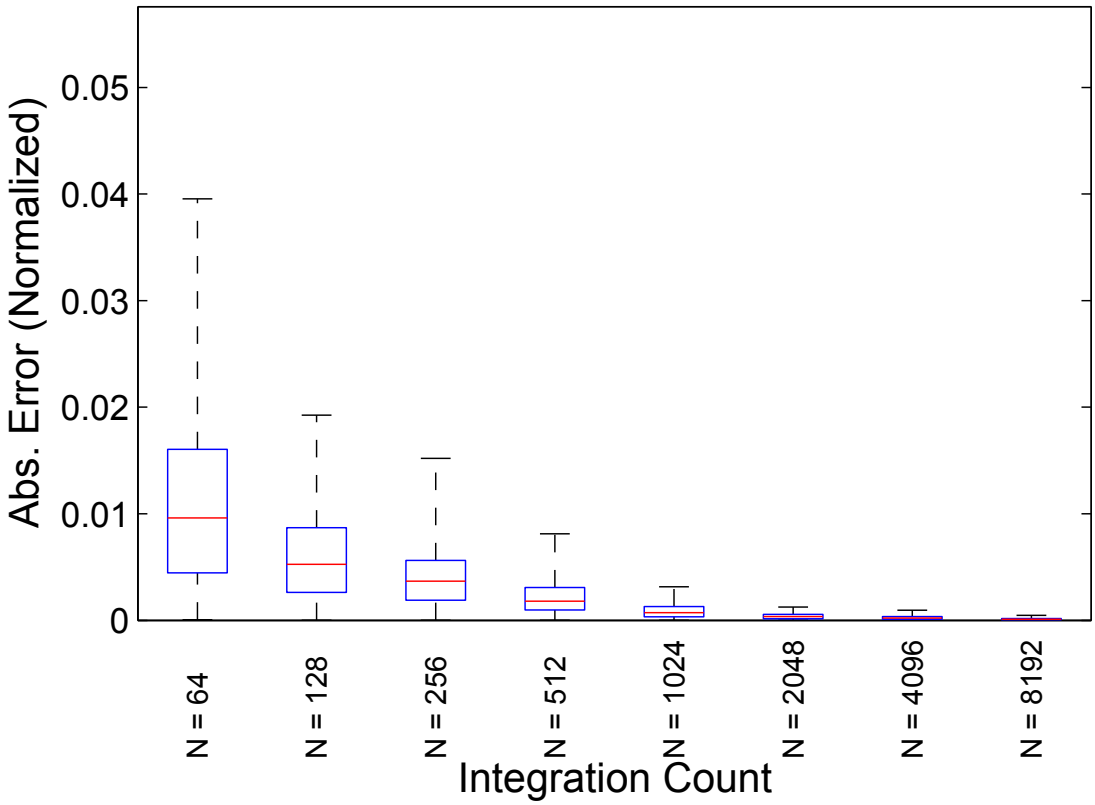


Figure 3.64: Absolute Error Box Plot from a GF5 VVM Core Emulated on the FPGA. The plot show the normalized absolute error of the output from one of the VVM cores from the emulated design in the FPGA compared to the results from computing with double floating point precision in MATLAB. For different integration count, the error is evaluated for all possible input and weight combinations.

Figure 3.64 and 3.65. The results from both the emulated design in the FPGA and the fabricated test chip distinctly show the median and also the max absolute error scaling with the integration count. By exploiting pseudo-random numbers through the LFSRs, as explained in Section 2.2.2, the necessary integration count to achieve M-bit precision in the output is far less than the expected constraint of 2^{2M+2} with true random numbers. The core from the emulated design in the FPGA achieves M-bit

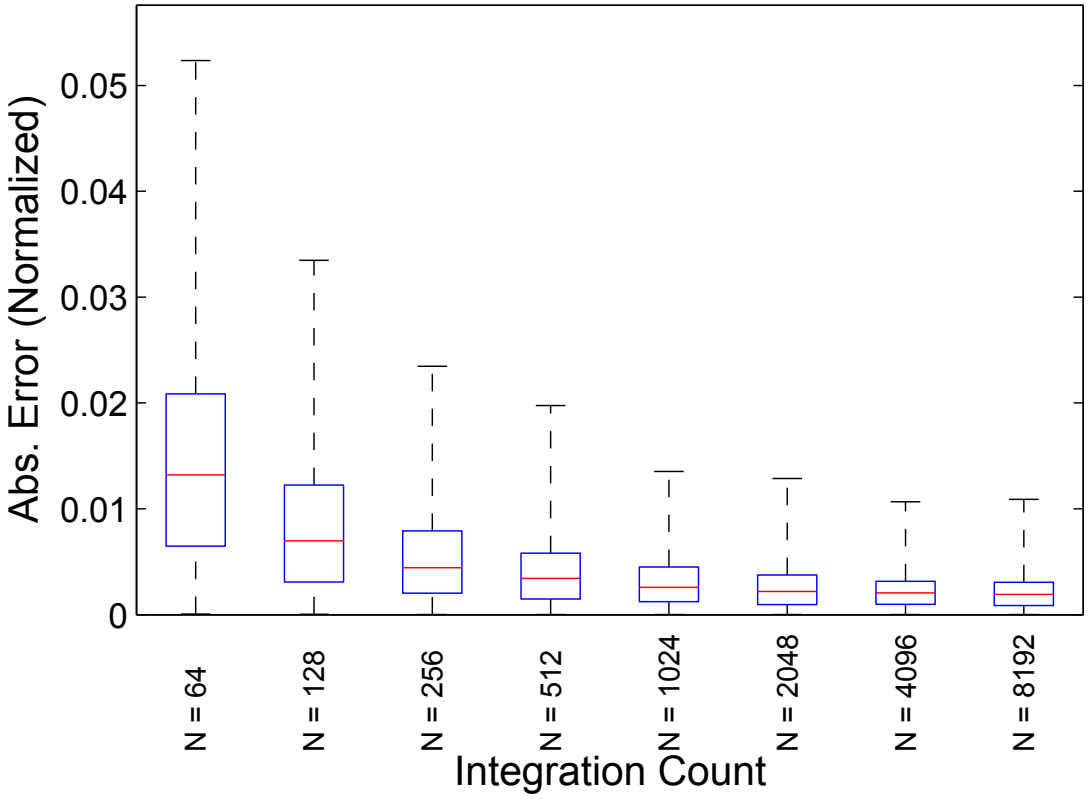


Figure 3.65: Absolute Error Box Plot from GF5 VVM Core on the Chip. The plot show the normalized absolute error of the output from one of the VVM cores from the chip compared to the results from computing with double floating point precision in MATLAB. For different integration count, the error is evaluated for all possible input and weight combinations.

precision averagely with 2^M samples.The chip also shows a similar integration count to error relationship, but plateaus at 8-bit due to noise in the ADC and mismatch on the capacitors.

The surface plots, displayed in Figure 3.66 and 3.67, show the output difference when processing with different integration counts with one of the fabricated VVM cores. In this plot, both 8-bit inputs and weights are swept across the full input and

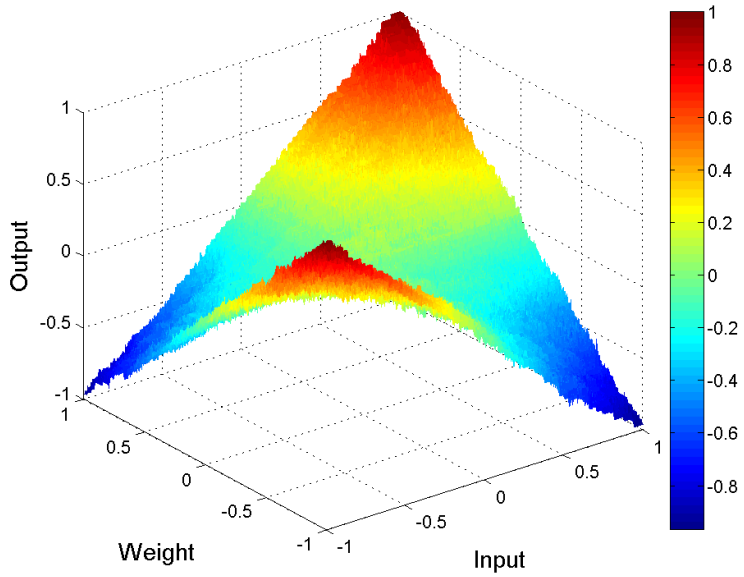


Figure 3.66: Surface Plot from a GF5 VVM Core Processing with an Integration Count of 64. The VVM output is computed for all possible combinations of the 8-bit weight and inputs and the normalized output is plotted.

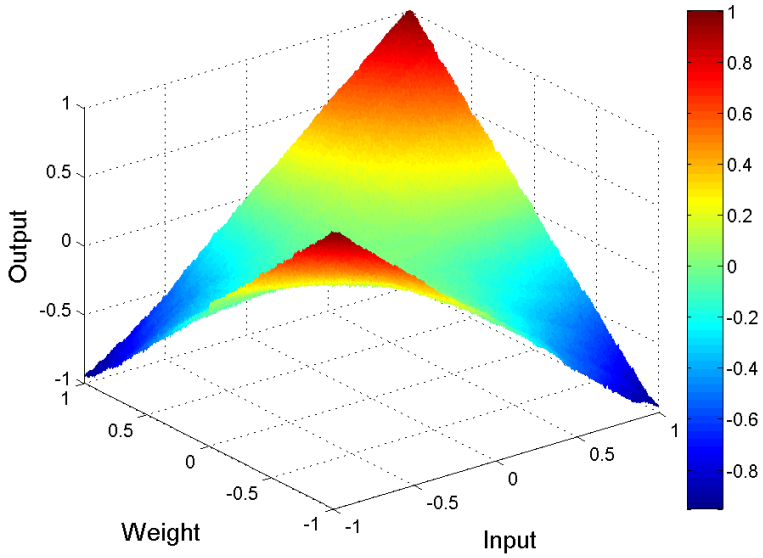


Figure 3.67: Surface Plot from a GF5 VVM Core Processing with an Integration Count of 256. The VVM output is computed for all possible combinations of the 8-bit weight and inputs and the normalized output is plotted.

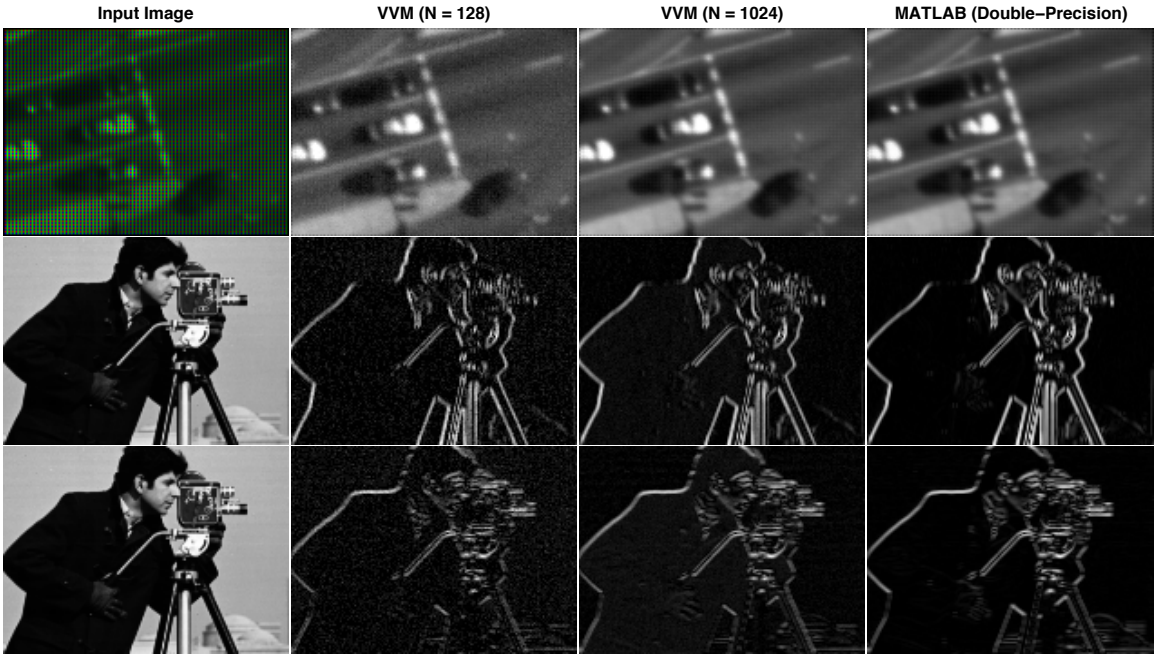


Figure 3.68: Montage of Image Processing Results Using the GF5 VVM Cores at Different Precision. First row shows DeBayering and color transform. Second and third row show vertical and horizontal edge detection respectively.

weight domain and the normalized output is plotted for an integration count of 64 and 256. In both cases the output is discernible, but for an integration count of 256 there is more resolution in the output as expected.

Moreover, this chip was applied for various image processing tasks, and the results can be seen in Figure 3.68. Each row in the image montage represents a different image processing task, and each column represents either the input image or the processed image at different precision. In the first row, the chip is used to DeBayer and color transform an image using a 3 by 3 filter, and in the next two rows the chip was used for vertical and horizontal edge detection separately. The leftmost column is the input image, the next column shows the images processed with an integration count

Table 3.20: Measured Characteristics of GF5 VVM.

Technology	55nm CMOS
Die Area	$0.45mm^2$
Operation	8-bit MAC
Output Precision	$\tilde{8}$bits
Throughput	25MOPs
Supply Voltage	1.2V
Frequency	16.67MHz
Power Consumption	$36\mu W$
Energy/Op	35pJ (570fJ for the array)
Efficiency	28.6GOP/W (1.8TOP/W for the array)

of 128 on the chip, the one after that is the images processed with an integration count of 1024, and the rightmost column is the image processed with double floating point precision.

The measured characteristic and performance of the GF5 VVM cores re detailed in Table 3.20. Operating on a 1.2V power supply at 16.67MHz in $0.45mm^2$ die area, the VVM chip computes 8-bit MACs at a throughput of 25MOP/s with an efficiency of 28.6GOP/W. Comparing to the previous GF4 VVM core, which computes 4-bit MACs, the energy per operation for the GF5 VVM core can be extrapolated to 2.19pJ with an energy cost of 35.63fJ on the array. With regards to efficiency, this core would compute 4-bit MACs at 28.1TOP/W on the array with a total efficiency of 457.1GOP/W on the core.

3.8 Conclusion

By exploiting charge-based computing in the analog domain, alternative architectures for energy efficient vector-vector multiplication are explored in this work. Fundamentally, these proposed architectures implicitly compute products as charge Q through the relationship of $Q = CV$, where one of the input, the weight, is encoded as capacitance C, and the other input is encoded as the V across the capacitor. By connecting multiple capacitors in the array, the inner product of a vector of weights and a vector of inputs can be computed as charge in the analog domain. Importantly, this vector-vector multiplications/inner product can be done efficiently by scaling the capacitance and voltage to the kTC thermal noise. In a 55nm and 65nm process, the energy per operation, where an operation is a multiply and accumulate with 4 or more bits has been scaled to the order of femtoJoules. Furthermore, after computing on the array, a charge-to-digital ADC, specifically a first-order switched-capacitor $\Sigma\Delta$ modulator, is used to decode this charge in time as a PDM output, where the output bit precision scales proportionally with the number of samples in time. In this work, 5 different mixed-signal architectures were designed and fabricated in submicron CMOS processes. These fabricated test chips were successfully tested, and benchmarked for performance and efficiency.

3.8.1 Design Optimization

Through the iterations of the mixed-signal VVM design, improvements were made to minimize inaccuracies due to the ADC and the capacitor array, improve efficiency by minimizing energy cost through the array, and improve performance through multicore system design. In the GF1 VVM design, the inner product is done through a unary-encoded 2D capacitor array composed of unit capacitors to foster good capacitance matching. In the next revision, GF2 VVM, a simpler binary-encoded array topology is used for lower area and energy cost at the cost of greater mismatch from fabrication variation. The VVM core was further simplified in GF3 VVM, where the capacitor array was redesigned to compute with less bits, and the layout was optimized for less area and energy cost. The GF4 VVM design further improved computation efficiency through the implementation of a series-parallel capacitor array that minimized capacitance in the array, and thus cut energy cost. Finally, the GF5 VVM design exploited stochastic computation in the analog domain in order to minimize energy and area cost by reducing the number of capacitors implemented in the core.

In order to contrast computation efficiency across the VVM designs, the designs were scaled to the same computation precision (4-bits), and the energy cost and efficiency were measured. The computation efficiency in the analog domain on the capacitor array for the designs can be seen in the scatter plot in Figure 3.69. Although all the VVM designs achieved more than 1TOP/W for computing 4-bit MAC

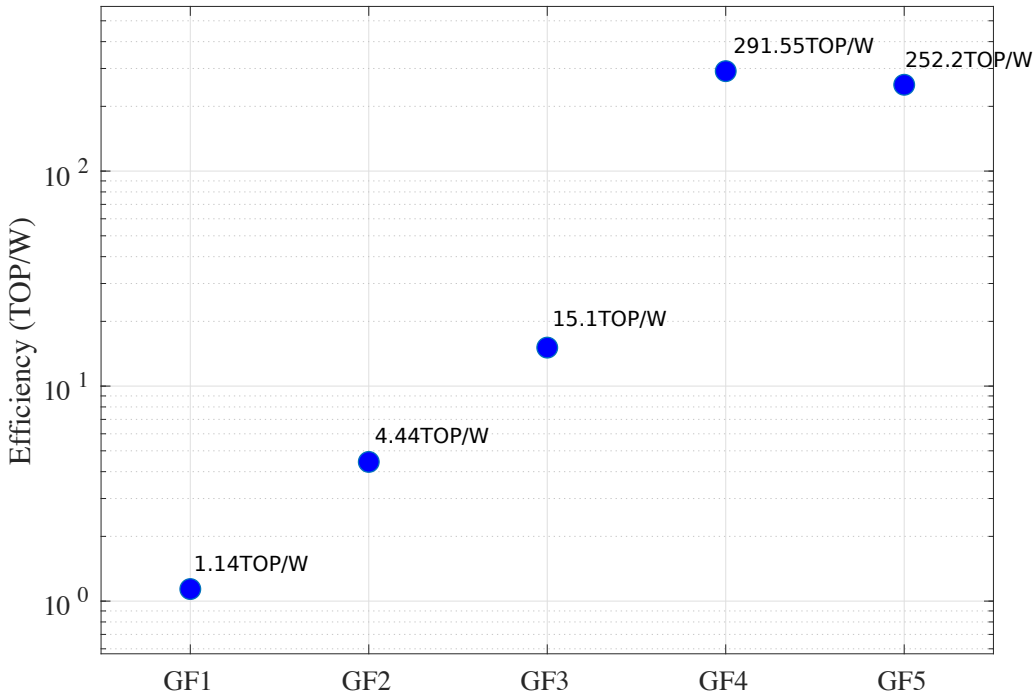


Figure 3.69: Computation Efficiency in the Analog Domain Across VVM Designs for a 4-bit MAC Operation (Simulation).

operations in the simulations, the GF4 VVM design achieved the best efficiency of 291.55TOP/W with the series-parallel capacitor array topology. Nonetheless, the GF5 VVM capacitor array has a more scalable design that can compute with higher weight precision with no added hardware resource, and still achieves an efficiency of 252.2TOP/W.

The computation efficiency for the full VVM cores, which includes the energy cost from decoding with the $\Sigma\Delta$ ADC can be seen in Figure 3.70. With each iteration of the VVM design, the overall efficiency was increased because of the improvements in

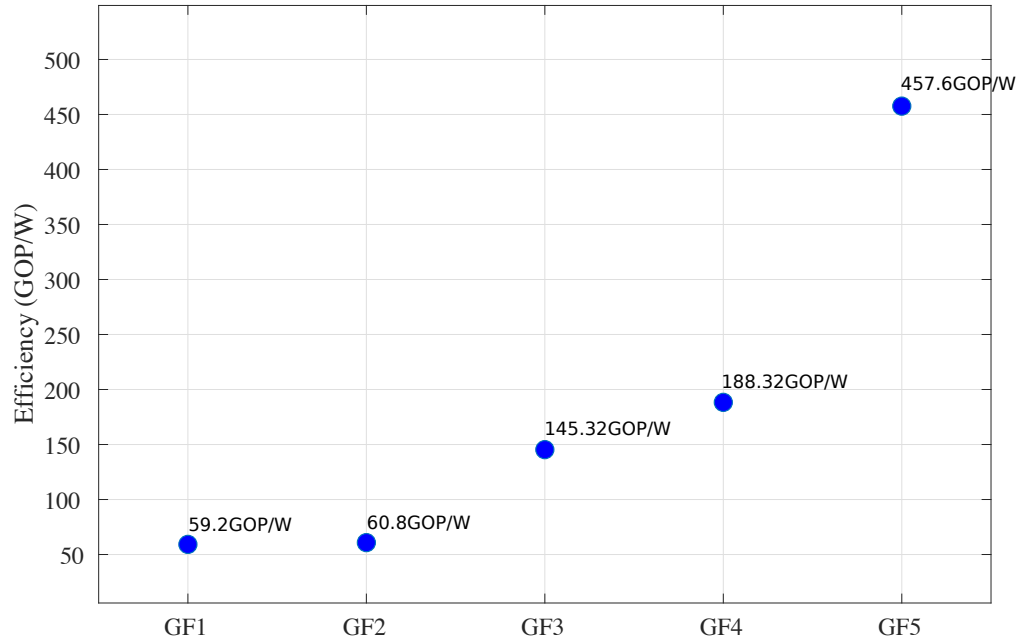


Figure 3.70: Computation Efficiency Across VVM Designs (Simulation).

the capacitor array and ADC. In the final iteration of this design, the efficiency for computing 4-bit MAC operations was measured at 457.6GOP/W.

3.8.2 Comparative Analysis

The mixed-signal approach to computing vector-vector multiplication as charge in the analog domain is an alternative solution to compete with conventional DSPs. In order to compare these mixed-signal architectures to conventional computing approaches, fixed-point vector-vector multiplication units were synthesized and characterized in the same submicron process. Specifically, processors were designed and

Table 3.21: Design Specification and Characteristic of the Synthesized DSP for 4-bit MAC and the GF VVM Core.

	DSP (4-bit MAC)	**GF5 VVM (array)**	**GF5 VVM (total)**
Technology	55nm CMOS	55nm CMOS	55nm CMOS
Area	$2116\mu m^2$	$461\mu m^2$	$3712\mu m^2$
Operation	4-bit MAC	4-bit MAC	4-bit MAC
Energy/Op	54.4fJ	4fJ	2185fJ
Efficiency	18.4TOP/W	252.2TOP/W	0.46TOP/W

synthesized for computing 4-bit MAC and 8-bit MAC operations. Similar to the GF5 VVM design, these processors were tailored for vector-vector multiplications with 9-element vectors.

The design specification and energy measurements from both the DSP, synthesized for 4-bit MAC operations, and the GF5 VVM core can be seen in Table 3.21. The array of mixed-signal multipliers in the GF5 VVM core is more computational efficient than the conventional DSP synthesized in the same process. Not only does it use less energy per operation, it also takes up less area. However, with the addition of the ADC for decoding and the periphery logic for running the mixed-signal VVM core, the full GF5 VVM core has higher energy and area cost than the conventional DSP implemented in this process.

Moreover, the design specification and energy measurements from both the DSP, synthesized for 8-bit MAC operations and the GF5 VVM core can be seen in Table 3.22. Although the GF5 VVM core has better area utilization for scaling computation bit precision, the conventional DSP synthesized in this process achieves better

Table 3.22: Design Specification and Characteristic of the Synthesized DSP for 8-bit MAC and the GF VVM Core.

	DSP (8-bit MAC)	**GF5 VVM (array)**	**GF5 VVM (total)**
Technology	55nm CMOS	55nm CMOS	55nm CMOS
Area	$7026\mu m^2$	$461\mu m^2$	$3712\mu m^2$
Operation	8-bit MAC	8-bit MAC	8-bit MAC
Energy/Op	193fJ	63.4fJ	35000fJ
Efficiency	5.2TOP/W	15.8TOP/W	0.03TOP/W

computation efficiency for higher computation bit precision. Nonetheless, the GF5 VVM core exploits stochastic computing to provide a robust and adaptable computing solution that not only achieves better efficiency at lower bit precision on the array, but also is more error-tolerant because if its innate probabilistic modality of computing.

Each of the different mixed-signal VVM designs highlighted the advantage of charge-based computing in the analog domain for better computational efficiency, and despite the constraining decoding cost with the $\Sigma\Delta$ ADC, these mixed-signal architectures could be optimized to be a competitive alternate to the conventional computing systems. Furthermore, with bandwidth limitation becoming a critical factor in technology scaling, this area efficient mixed-signal architecture approach to computing can be exploited in building more scalable systems that properly leverage computation time and precision.

Chapter 4

A Mixed-Signal Successive Approximation Architecture for Energy-Efficient Fixed Point Arithmetic

In the previous chapter, mixed-signal architectures were presented that exploit computing in the analog domain for energy efficient vector-vector multiplications. Leveraging charge-based computing, these architectures showed how implicitly computing inner products as charge at the thermal noise limit can reduce energy cost. Nonetheless, these gains in energy efficiency have been primarily limited to decoding cost with the analog-to-digital conversion. Thus, an alternative design, inspired from

the successive approximation (SA) analog-to-digital converter (ADC), is explored that combines the computational and decoding structures to minimize energy cost and improve computational efficiency. Specifically, an architecture is proposed that uses a capacitive DAC and a successive approximation register (SAR) for computing multiplication and addition operations at the thermal noise limit.

4.1 Successive Approximation (SA) ADC

As one of the most popular architectures for high speed low-power data conversion, the SA ADC has been widely adapted for data conversion,[64–67] biomedical applications,[60] imaging,[68] sensor networks,[69] and more. Specifically, the charge redistribution SA ADC, which is comprised of a capacitive DAC, a comparator, and some output decoding logic, SA ADCs can be designed for both high resolution and high speed while being implemented in relatively small area. Using a SAR, the SA ADC achieves efficient data conversion by performing a binary search through all possible quantization levels to converge to the correct digital output representing the analog input. The DAC and the comparator work concurrently with the SAR to perform this search, and thus convert from the analog to the digital domain.

A block diagram of the charge redistribution SA ADC can be seen in Figure 4.1. The voltage input Vin in this architecture is converted into a digital value by encoding as charge in the DAC, and then performing a binary search that manipulates this

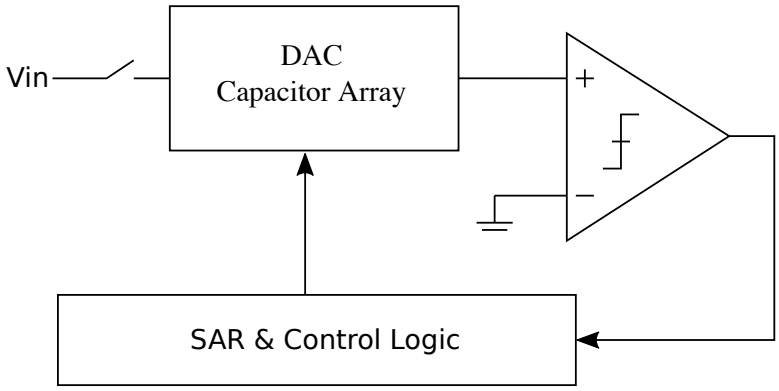


Figure 4.1: Successive Approximation ADC Block Diagram.

charge to deduce the converted output. To achieve this, first the Vin is sampled on the capacitor array. Then, using this capacitor array as a voltage divider, charge quantized to different bit precision is added to the array, which creates a delta voltage that is measured through the comparator. This modulated voltage from the shared plate of the capacitor array V_t is given as

$$V_t = -Vin + \left(D_{(N-1)} \cdot \frac{V_{ref}}{2^1}\right) + \left(D_{(N-2)} \cdot \frac{V_{ref}}{2^2}\right) + \cdots + \left(D_0 \cdot \frac{V_{ref}}{2^N}\right) + V_o, \quad (4.1)$$

where V_{ref} is the reference voltage, which represents the maximum the input voltage, V_o is the offset voltage from the comparator, and $D_{(N-1)}$, $D_{(N-2)}$,...,D_0 are the individuals bits of the SAR which represents the converted output.

The primary limitation of this ADC, is the capacitor matching, as accuracy and precision in this architecture depends on the capacitance mismatch in the DAC. This

capacitance mismatch gives a maximum integral non-linearity (INL) of

$$|INL|_M = \frac{V_{ref} \cdot 2^{(N-1)} \cdot |\Delta C|_M}{2^N \cdot C} = \frac{V_{ref}}{2} \cdot \frac{|\Delta C|_M}{C}, \tag{4.2}$$

where $|\Delta C|_M$ is the maximum delta capacitance from mismatch. In order to achieve N-bit precision, this maximum INL is limited to half an LSB, which corresponds to a bit precision equivalent to

$$N = log_2\left(\frac{C}{|\Delta C|_M}\right). \tag{4.3}$$

Similarly, the maximum differential non-linearity (DNL) is given as

$$|DNL|_M = \frac{V_{ref} \cdot (2^N - 1) \cdot |\Delta C|_M}{2^N \cdot C}, \tag{4.4}$$

and with a bound of half of a LSB for the maximum DNL, the deduced bit precision is similar to Equation 4.3.

Despite these accuracy issues from capacitor mismatch, numerous of SA ADC have been designed that achieve conversion precision of 8-bits,[67,68,70] 10-bits,[64] and even more.[66,69]

4.2 SA Multiply-Add Architecture

4.2.1 Multiply-Add Operation

The proposed mixed-signal architecture, computes a fixed-point multiply-add operation as

$$y = wx + c, \tag{4.5}$$

for a signed weight w, input x, and offset c encoded as signed magnitude. Conventionally, the unsigned product of the magnitude of w and x can be computed as the sum of the partial products deduced from the individual bits of the input and weight; this is illustrated in Table 4.1. The sign of this product can be deduced from simple

Table 4.1: Unsigned Binary Multiplication.

				$\mathbf{w_{n-1}}$	$\cdots$	$\mathbf{w_1}$	$\mathbf{w_0}$
			$\times$	$\mathbf{x_{n-1}}$	$\cdots$	$\mathbf{x_1}$	$\mathbf{x_0}$
				$\mathbf{p_{(n-1,0)}}$	$\cdots$	$\mathbf{p_{(1,0)}}$	$\mathbf{p_{(0,0)}}$
			$\mathbf{p_{(n-1,1)}}$	$\cdots$	$\mathbf{p_{(1,1)}}$	$\mathbf{p_{(0,1)}}$	
				$\vdots$			
$+$	$\mathbf{p_{(n-1,n-1)}}$	$\cdots$	$\mathbf{p_{(1,n-1)}}$	$\mathbf{p_{(0,n-1)}}$			
$\mathbf{y_{2n-1}}$			$\cdots$			$\mathbf{y_1}$	$\mathbf{y_0}$

logic of the MSB of the weight and input, and then the full result from the product can be added to offset c to derive the output of this operation.

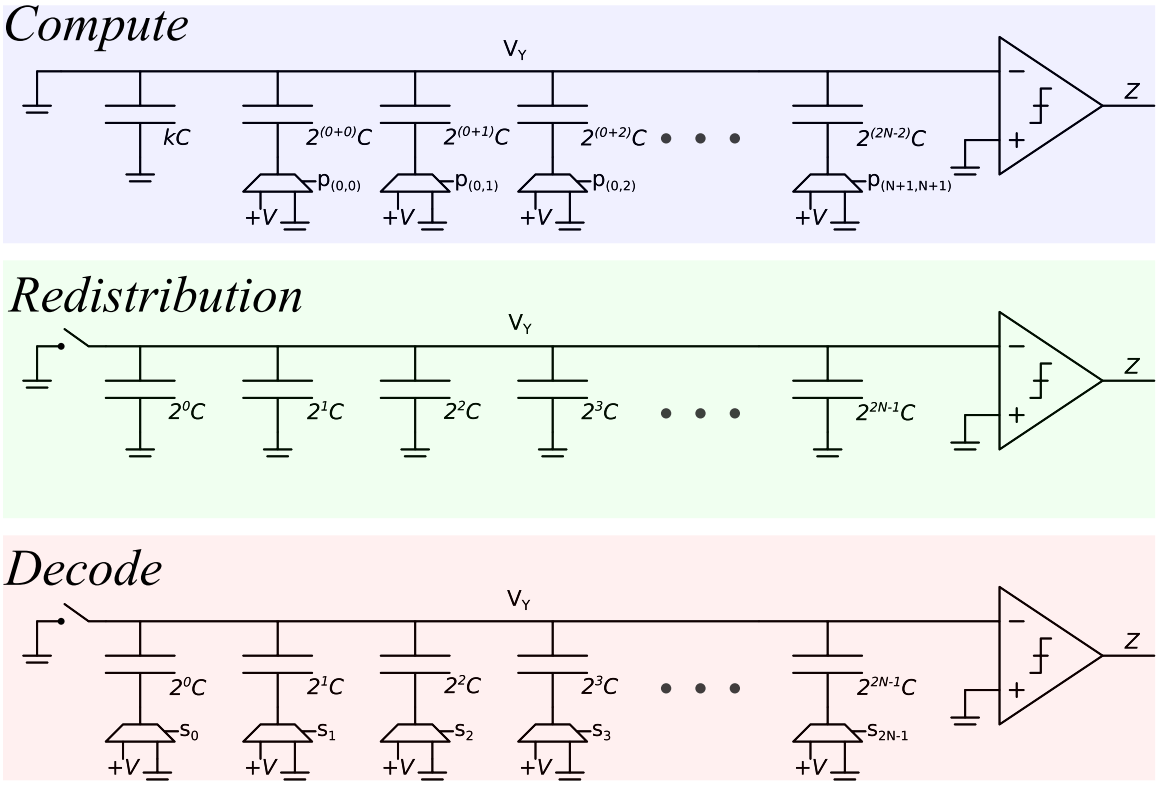


Figure 4.2: Processing flow for SA Architecture.

4.2.2 Mixed-Signal Processing Flow

As opposed to using a cascade of full adders for summing the partial products, a programmable capacitor array is used to sum the scaled partial product bits as charge. Qualitatively computing in the analog domain, allows for scaling towards the thermal noise limit, which can be exploited for more energy efficient processing. Moreover, this charge can then be converted very efficiently using a successive approximation-based binary search. Integrating these methodologies together, a mixed-signal multiplier can be designed that operates as the flow illustrated in Figure 4.2. This flow is comprised of three different phases — the **Compute** phase, **Redistribution** phase,

and finally the **Decode** phase — for integrating the weighted partial product bits.

Firstly, in the **Compute** phase, a charge equivalent to the sum of the weighted partial product bits is injected into the array of capacitors. The top plates of all capacitors are connected to a common ground node, while the bottom plates are isolated and connected to either a power supply with $+V$ volts or the common ground node depending on the partial product bits. This total charge Q_Y can be expressed as

$$Q_Y = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2^{i+j} p_{(i,j)}(CV), \tag{4.6}$$

where i and j are the indexes of the partial product bits generated from the logical AND operation between the weight and input as shown in Table 4.1, C is the unit capacitance, and V is the voltage potential induced across the capacitors. Thus, by connecting a set of capacitor in parallel using switches, the partial product can be integrated as the total charge Q_Y injected onto this capacitor array. Furthermore, by scaling this voltage and capacitance to the thermal noise (kTC) and mismatch limit, this computation can be done efficiently.

After the first phase, the charge Q_Y is translated into a voltage V_Y in the **Redistribution** phase. In this phase, charge is redistributed evenly across all the capacitors in the array by disconnecting the top plates of the capacitors to ground, and connecting all the bottom plates of the capacitors to ground. Since the charge is conserved

between the Compute and Redistribution phase,

$$Q_Y = C_T V_Y \tag{4.7}$$

$$\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2^{i+k} p_{(i,j)}(CV) = C_T V_Y \tag{4.8}$$

$$V_Y = \left(\frac{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2^{i+k} p_{(i,j)} C}{C_T}\right) V, \tag{4.9}$$

where C_T is the total capacitance of the array when all the capacitor are connected. C_T is sized for the precision of the computation, which is,

$$C_T = (2^N - 1)C \tag{4.10}$$

Thus, the voltage V_Y represents the sum of the weighted partial product bits scaled by the voltage of the power supply V and the capacitance of the array C_T.

Finally, this voltage representing the output of computation is converted back into a digital value in the **Decode** phase. Using a digital control word s_i, the capacitor array is reconfigured from a parallel configuration to a series-parallel configuration in order to perform the binary search to decode the voltage V_Y into the digital word. In the series-parallel configuration, the V_Y is voltage divided according to each bit position in the digital word, and compared to the common ground node indicating the magnitude of that bit for the decoded result. Although the charge from the computation is preserved on the capacitor array, an additional charge is injected into

the array while decoding, which in turns creates a delta voltage; this voltage V_D during this phase is given as

$$V_D = V_Y + \left(\frac{C_D}{C_T}\right)V, \tag{4.11}$$

where C_T is given earlier as $(2^N - 1)C$, C_D is total capacitance of the capacitors with the bottom plate connected to the power supply with voltage V in the capacitor array, and V_Y is initial voltage for this phase that represents the output of the computation. During decoding, C_D changes exponentially at a rate of 2^i according to the control word s_i from most-significant to least-significant bit. Through the binary-search the result of the computation can be decoded in $2N$ time steps, where N is the bit-precision of the weight and the input, and hence the total precision of the resulting product.

Furthermore, by adapting this architecture to a differential topology, the addition operation can be implicitly performed with a voltage offset on the other input of the comparator. The full architecture implementation is described in the following section.

4.2.3 Architecture

The full architecture implementing the fixed-point multiply-add operation can be seen in Figure 4.3. This architecture operates under the 3 phases described in detail

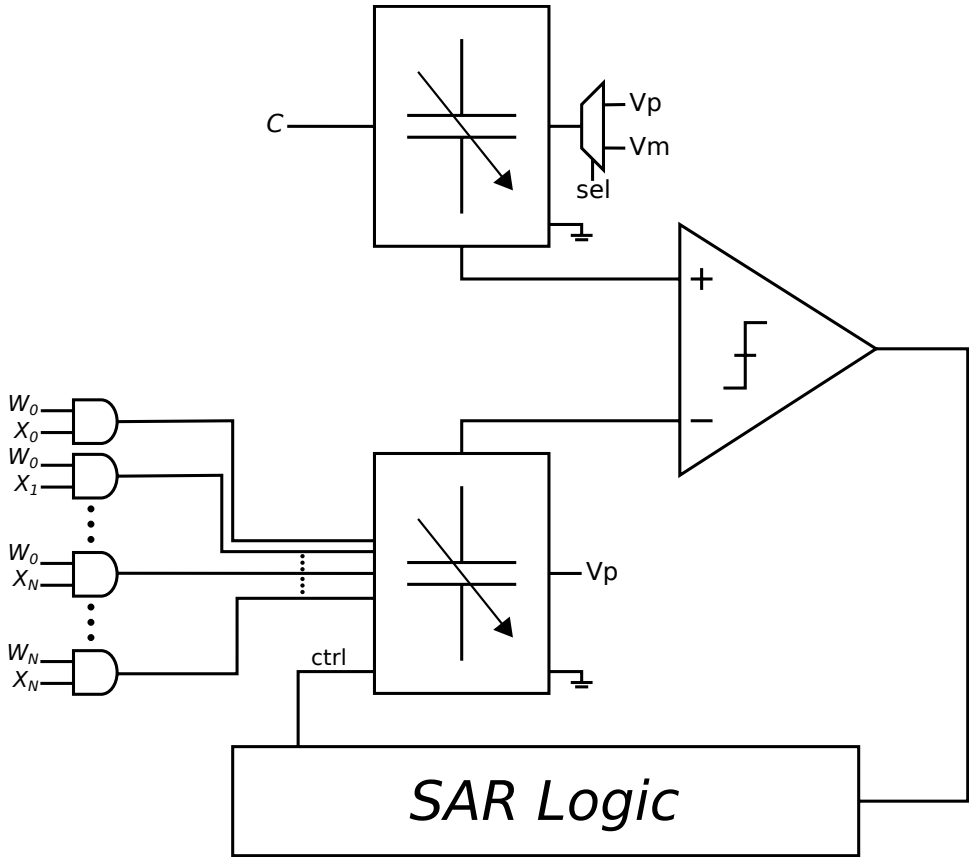


Figure 4.3: SA Architecture for Multiply-Add Operations.

in Section 4.2.2. As shown in the full architecture diagram, the partial product bits are computed from the weight w and the input x, and these bits are then used to program a capacitor array. The connected top plates of this capacitor array connects to the comparator, which is used in decoding the output of the computation from the analog domain. An additional programmable capacitor array is used for generating a voltage offset used during decoding for implementing the addition operation and also calibration if necessary.

Furthermore, the capacitor array for this architecture is shown in Figure 4.4. The 3 phase process is simplified into two phases, where the first phase ϕ_0 does the

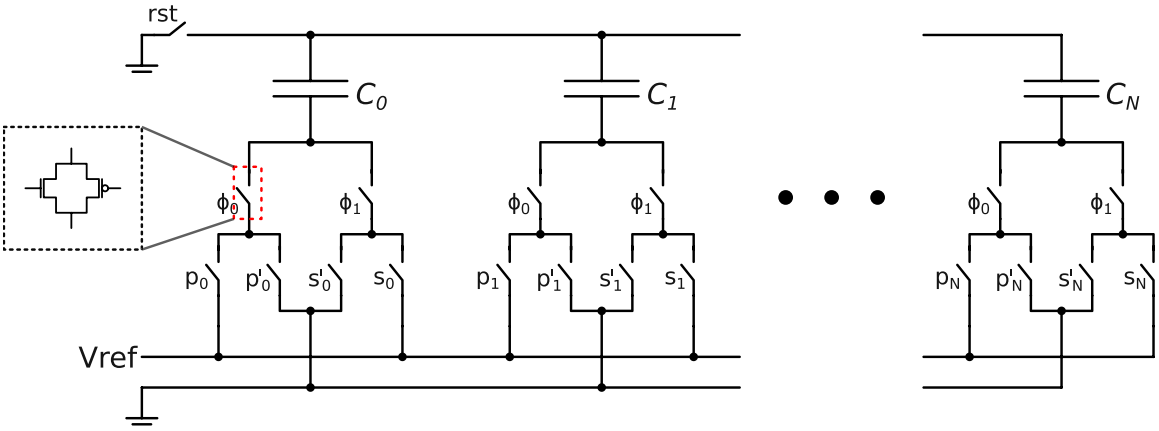


Figure 4.4: Programmable Capacitor Array for SA Architecture.

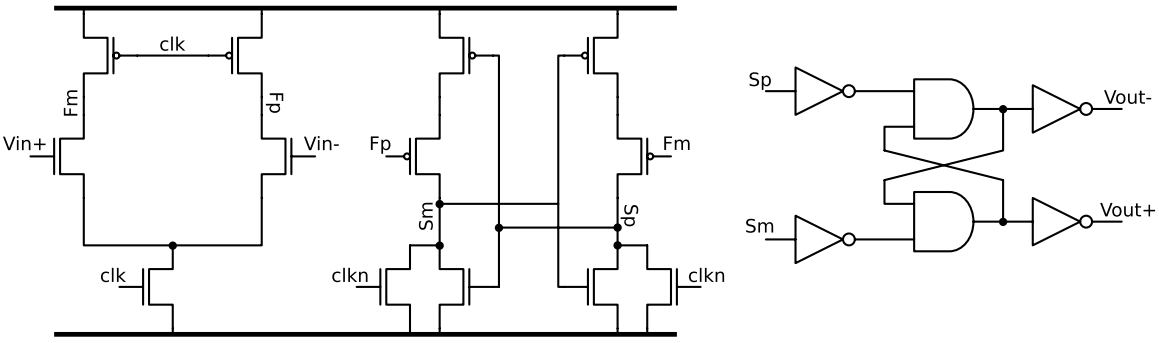


Figure 4.5: Dynamic Comparator and Latch for the SA Architecture.

computation, and the second phase ϕ_1 does the redistribution and the decode step. Analog multiplexers comprised of transmission gates are used to charge and discharge the capacitor to either the reference voltage or ground. Moreover, the *rst* switch is used to reset the top node plate for each processing cycle.

As shown in Figure 4.5, a two-stage dynamic comparator along with a SR latch is used for decoding the analog output. When the *clk* is low (reset), Fp and Fm are set high, and Sp and Sm are reset low. With the outputs of the comparator Sp and Sm set to low, the SR latch is in the latch state, and the values of $Vout+$ and

$Vout-$ are preserved. Then when clk is high (compare), the difference between the inputs to the comparator $Vin+$ and $Vin-$ is amplified and reflected in the difference between Fp and Fm. This amplification from the first stage causes the outputs of the second stage Sp and Sm to pull to the respective rail through positive feedback from the cross-coupled inverters. The comparator outputs set the new value for the latch which is held for a clock cycle.

4.2.4 Computational Efficiency

In order to compare the computational efficiency of this design to that of conventional architectures, circuit simulations along with an energy analysis is done. The mixed-signal SA architecture was designed in a 16nm FinFET process, and this architecture was constructed to compute 8-bit multiply-add operations with 5-bit signed weights and inputs, and 8-bit offset. The capacitor array was designed with 4fF unit capacitors, with an input voltage bias of 50mV (an LSB of 200μV). In addition, a digital implementation of a fixed-point multiplier and adder with the same precision was synthesized in the same process.

Subsequently, the SPICE models of both implemented designs were simulated with different supply voltages and clock rates in order to measure performance and efficiency. With a nominal process supply voltage of 0.8V, the power supply is swept from 0.4-0.8V for both designs, and the total energy from these simulations were measured based on a similar setup as described in Section 3.3.1.2. The results of this

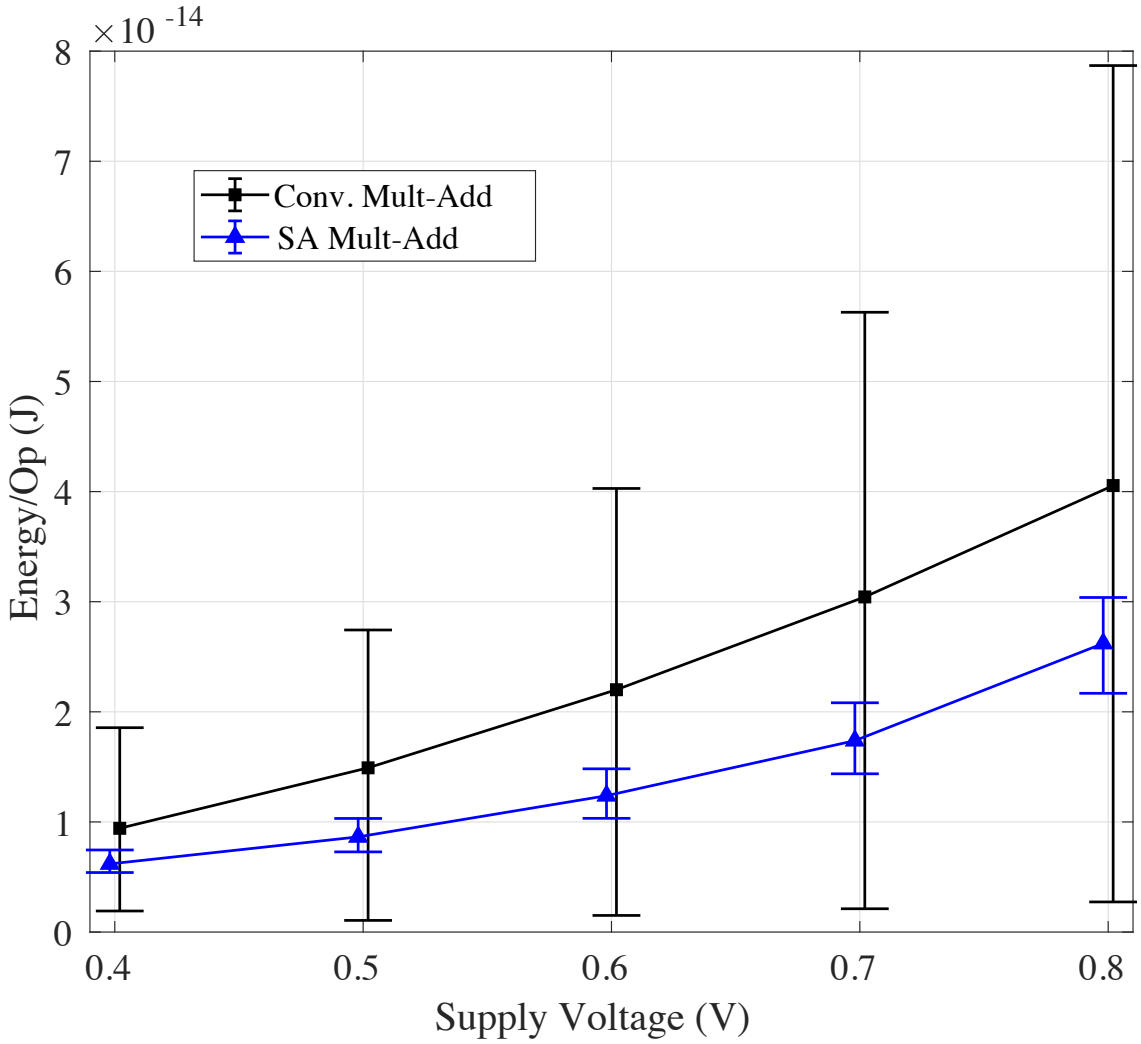


Figure 4.6: Energy Plot across Different Supply Voltages for the Different Multiply-Add Designs.

analysis can be seen in Figure 4.6. As the energy fluctuates with both leakage current and dynamic current from gates switching, numerous of measurements are recorded as the inputs are randomly sampled in measuring the distribution of the energy cost for different supply voltages. The SA-based multiply-add design on average is more energy-efficient, and has lower maximum energy-cost compared to a conventional

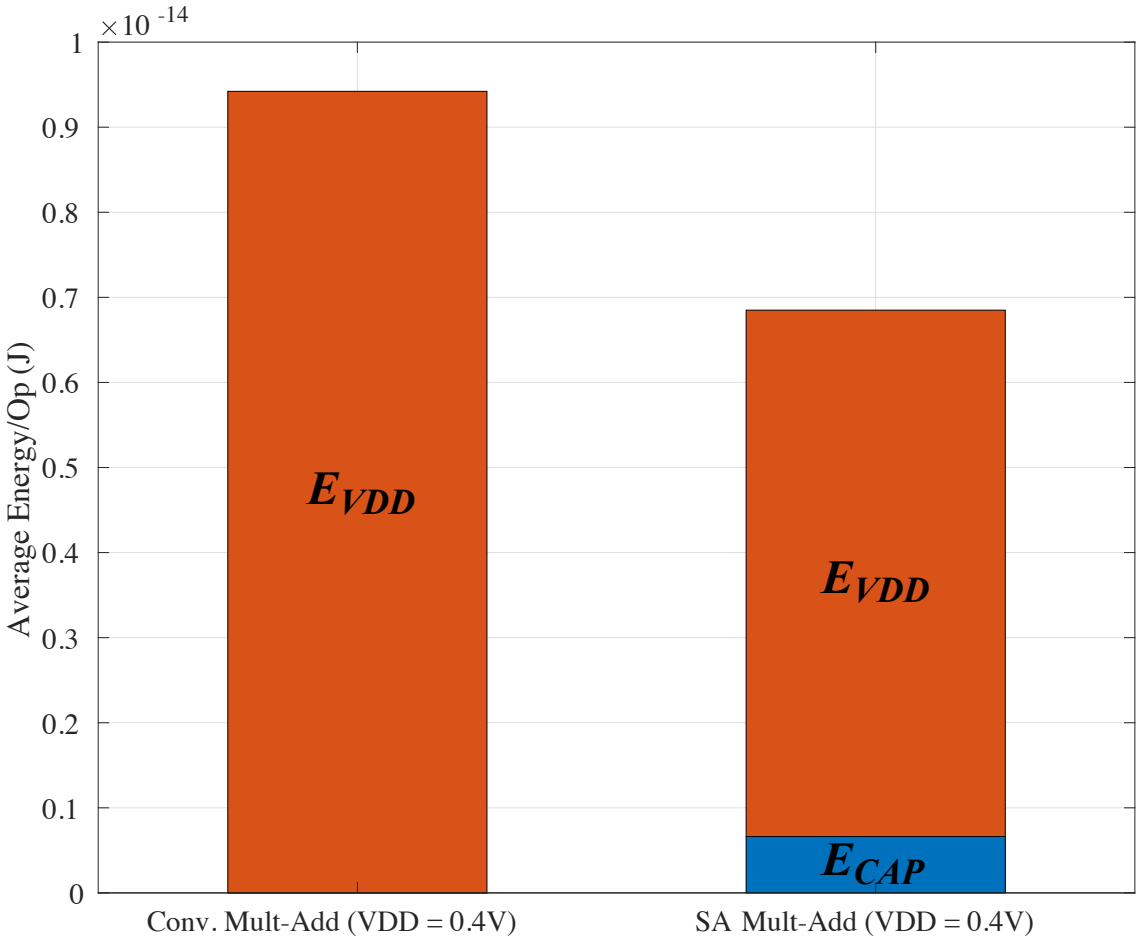


Figure 4.7: Energy Comparison for the Different Multiply-Add Designs at 0.4V Supply Voltage.

implementation.

Moreover, as the energy scales quadratically with the supply voltage, both design achieve their minimum bound of energy cost per operation at 0.4V from this analysis. Comparing these two cases, the plot shown in Figure 4.7 contrasts the average energy cost between these two designs at the aforementioned supply voltage. The conventional multiply-add design uses 9.4fJ averagely per operation, while the SA

Table 4.2: Simulated Characteristic of the SA Mulitply-Add Single Core.

Process	16nm FinFET
Operation	8-bit Multiply-Add
Supply Voltage	0.4V
Clock Frequency	50MHz
Throughput	3.57MOPs
Power	24.5nW
Energy/Op	6.85fJ
Energy Efficiency	146TOPs/W

multiply-add design uses 6.85fJ averagely per operation, which equates to an energy savings of roughly 37%. Furthermore, as majority of the energy cost is attributed to the dynamic comparator, the capacitor array, which computes the operation, uses roughly 0.66fJ for the computation.

The summary of the simulated characteristics of the SA multiply-add single core can be seen in Table4.2. Measurements presented in this table are solely based on SPICE model simulations, and are not fully indicative of the performance and efficiency of a fabricated core. Furthermore, the energy dissipated from the SAR decoding logic is not included in this simulation as this logic can be shared across multiple cores with the exception of the state-holding successive approximation register. Nonetheless, these results show that the proposed SA multiply-add core can compute 8-bit multiply-add operations with an average efficiency 146TOPs/W.

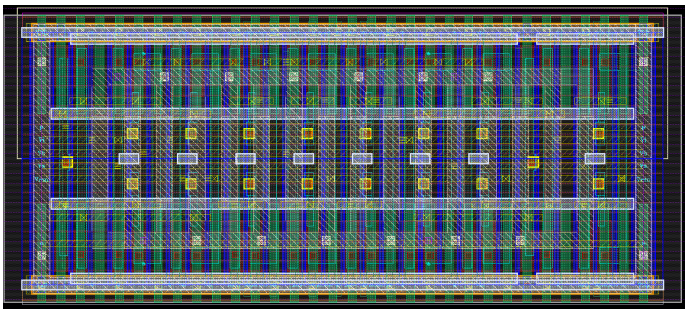


Figure 4.8: Layout of the Unit Cell of the DAC for the SA Multiply-Add Core.

4.3 16nm FinFET Test Chip

A test chip, comprised of multiple SA multiply-add cores, was laid out in a 16nm FinFET process for validating the proposed architecture. The core layout, which was matched to the circuitry described in Section 4.2.3, was optimized for minimal area and energy through custom design validated with parasitic extraction simulations. In order to maximize the efficiency of area utilization, the capacitor array for the DAC was constructed as a 2D grid of cells containing a unit capacitor and the pass gate logic for driving the cell (as seen in Figure 4.4). Similar to the the mixed-signal vector-vector multiplier (VVM) capacitor array discussed in Chapter 3, a metal-oxide-metal (MOM) capacitor design is used for the unit capacitors in the mixed-signal cores. The pass gate logic and the capacitor are laid out jointly in a 3D layout topology, where the MOM capacitors, which were designed across the higher level metals, are underlaid with the periphery circuitry and routing. The layout for the unit cell for the DAC can be seen in Figure 4.8. The MOM capacitor was designed across two metal layers (M4 and M5), and was shielded below and laterally to minimize parasitic

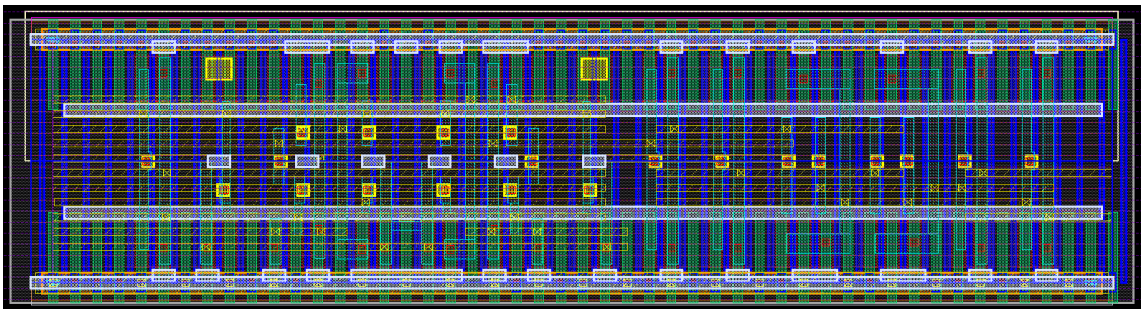


Figure 4.9: Layout of the Dynamic Comparator and Latch for the SA Multiply-Add Core.

coupling from the digital signals and adjacent cells. This cell measures 1.248μm by 2.976μm (3.714μm^2) in area with an approximate unit capacitance of 4fF, which was deduced from the parasitic extraction analysis.

Moreover, the dynamic comparator and latch (circuit shown in Figure 4.5) were laid out in 1.056μm by 4.604μm (4.862μm^2) silicon area, and a netlist, including parasitic nodes, was extracted and simulated to verify functionality and measure performance. The layout for this comparator and latch can be seen in Figure 4.9. The comparator was simulated to run at 1GHz, and a Monte Carlo simulation was done to characterize the comparator offset. A histogram plot of the result of this simulation can be seen in Figure 4.10. The maximum comparator offset with this design with process and mismatch variations didn't exceed 21mV. With the added functionality of calibration in the core, this offset can easily be corrected.

The complete layout of the capacitor array, comparator, latch, and periphery circuits for the SAR multiply-add architecture can be seen in Figure 4.11. The design measures 51μm by 53μm in area (2703μm^2) without the digital periphery circuit

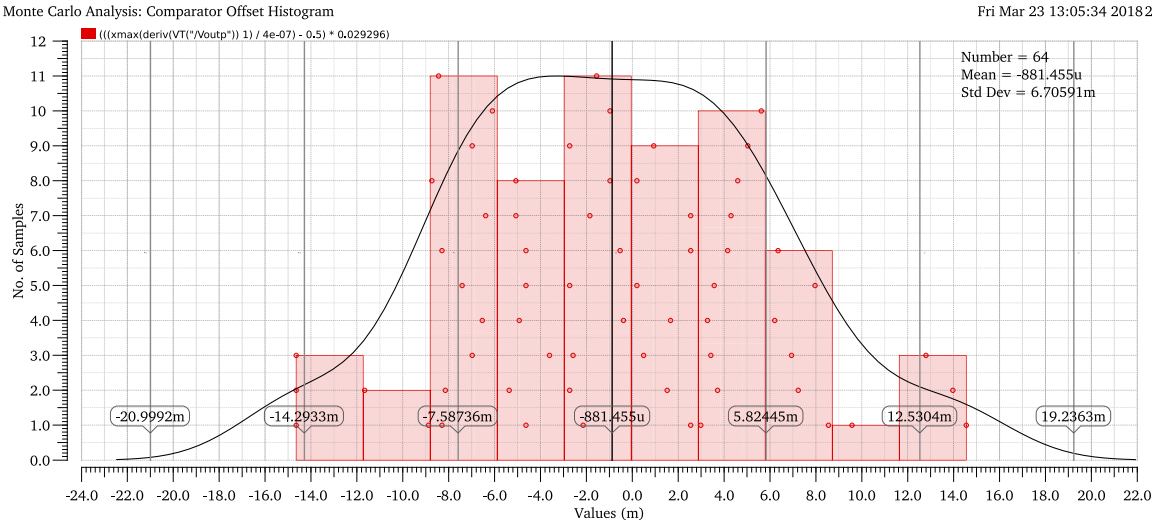


Figure 4.10: Histogram Plot of the Comparator Offset with Fabrication Variation.

(which can be shared across cores), and computes 8-bit products and additions. Comparing to a digital implementation, which would require roughly 96μm^2 of silicon area, this SA-based multiply-add core has higher area cost. Nonetheless, majority of this area is primarily used for the layout of the unit capacitors, as shown in the illustration in Figure 4.12. For the full computation a total of 510 capacitors are used for both capacitor arrays. In order to facilitate the routing and sharing of the periphery logic circuitry for driving the capacitors, the capacitor arrays are arranged in a 2D grid of 36 by 8 unit capacitors Thus, 33 dummy capacitors are incorporated in each capacitor array.

Furthermore, since the periphery logic that drives the cells are shared across capacitors, most of the underlaid circuitry is inactive and unused Figure 4.13 highlights the inactive logic in the core layout. The total active logic area is 271.8μm^2, which

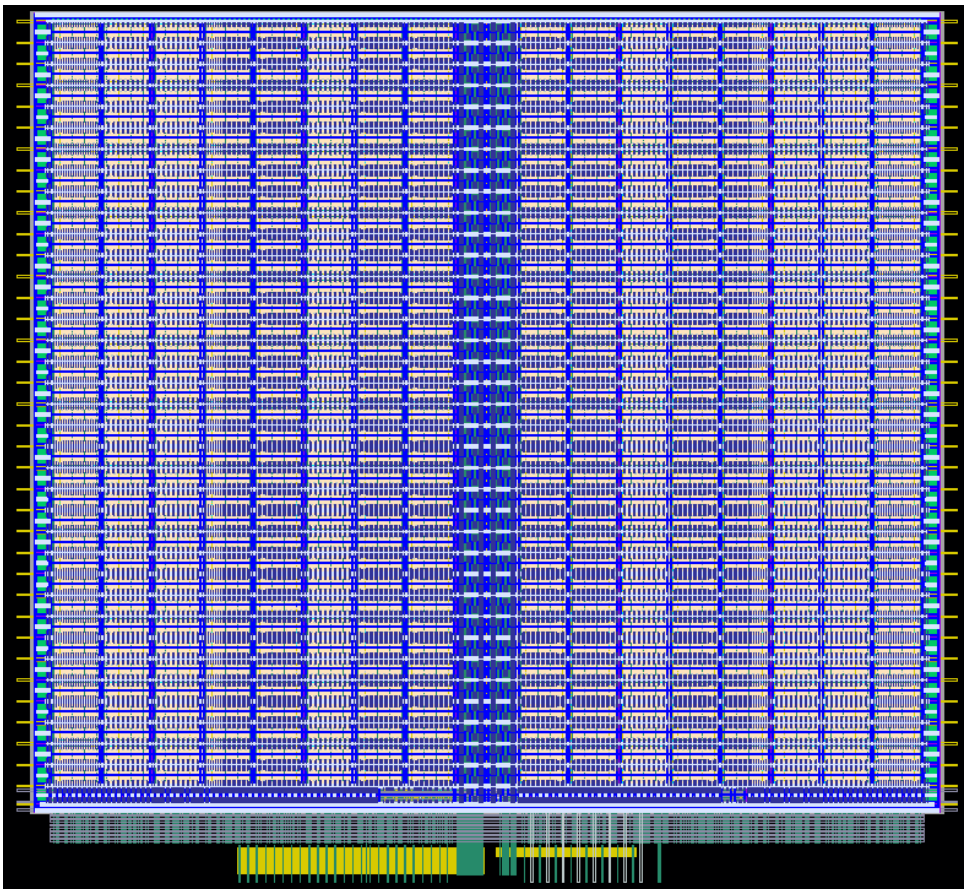


Figure 4.11: Layout of the SA Multiply-Add Core without SAR Decoding Logic.

accounts for approximately 10% of the total area of the core. Furthermore, majority of this active logic used in this core is pass gate logic, which has minimal dynamic power, as cells are not constantly switching between the power rails.

In the test chip, an individual core is placed with synthesized digital periphery circuitry to measure single core performance and efficiency. In addition, an array of 64 multiply-accumulate (MAC) units composed of SA-based multiply-add cores is constructed in this chip. Each processor has a multiply-add unit along with a 16-bit accumulator to do more general-purpose DSP. Although each unit should achieve the

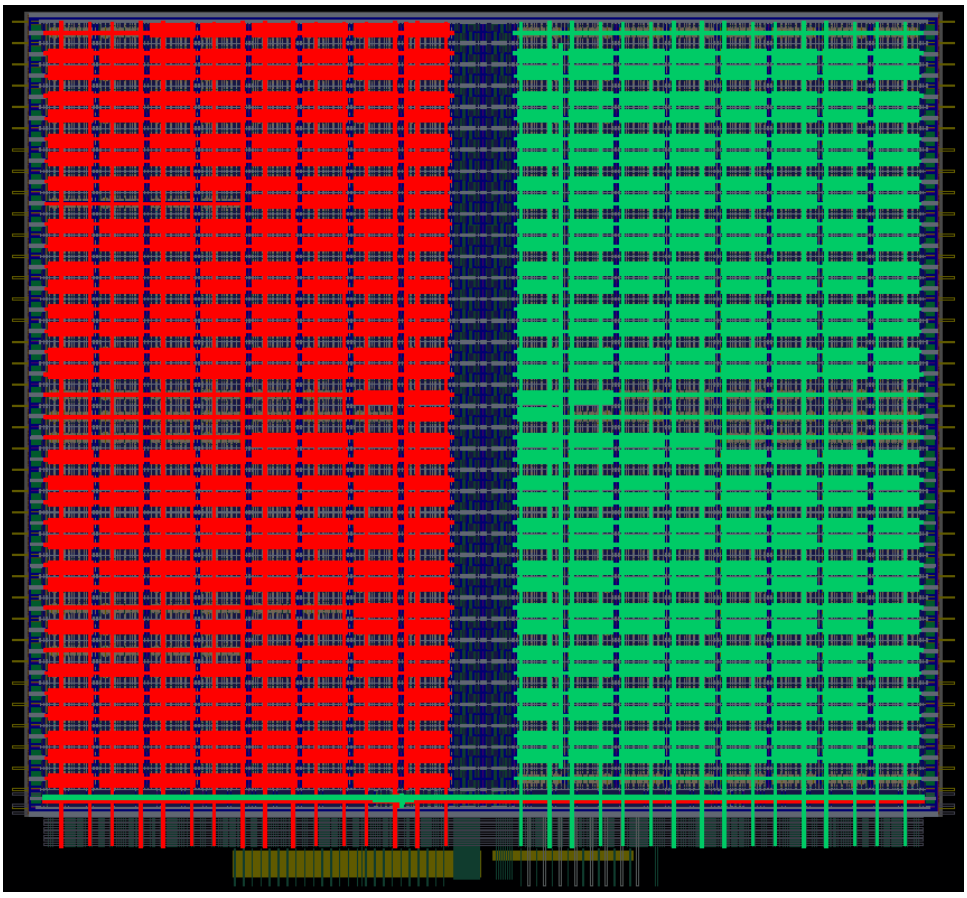


Figure 4.12: Layout View of the SA Multiply-Add Core with the Active Capacitors Highlighted. The main capacitor array for summing the partial product bits is highlighted in red, and the capacitor array for calibration and the additional add operation is in green.

simulated performance as shown in Table 4.2, this is not achievable due to bandwidth constraints from limited number of pins. Moreover, this array is particularly useful for measuring accuracy and performance variations across these mixed-signal processors. The layout of the full test chip can be seen in Figure 4.14. A detailed description of the pins for this test chip can be seen in Table 4.3. The pins are segmented by two blocks — block #1 (B1) is the single core SA multiply-add core and block #2 (B2) is the 64-core MAC processor. With regards to pin types, POW is power, GND is

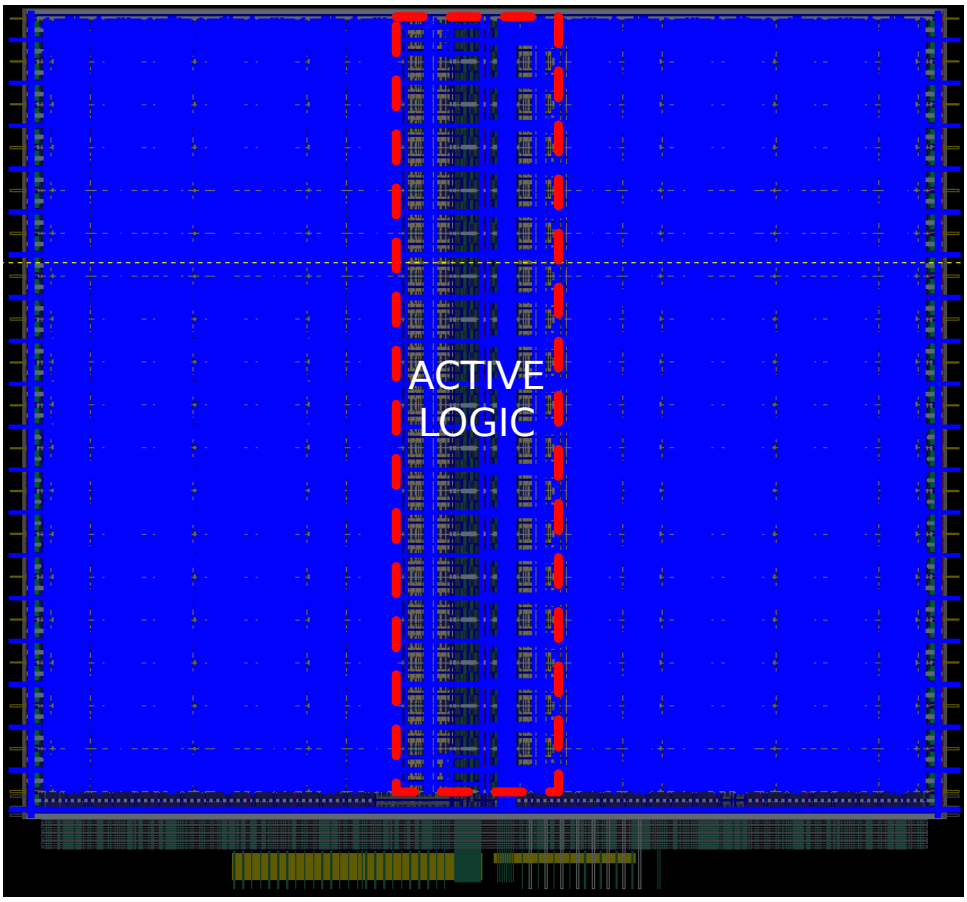


Figure 4.13: Layout View of the SA Multiply-Add Core with the Inactive Logic Highlighted. The layout highlighted in blue is the inactive logic that is underlaid the capacitors.

ground, DI is digital input, DO is digital output, and AI is analog input. The Vp and VDD core ports are separated between the two blocks for power measurement purposes. There are total of 30 pins used in this chip.

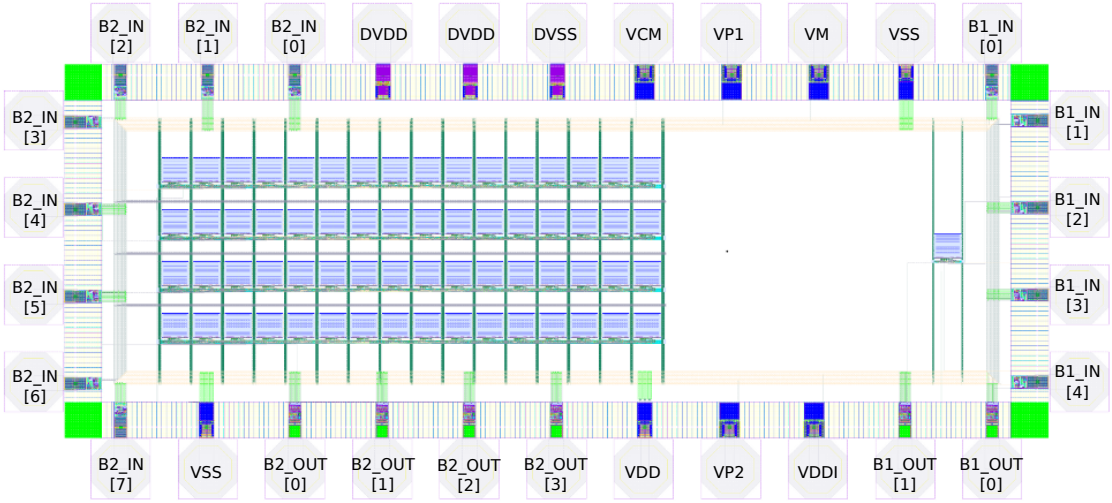


Figure 4.14: Annotated Layout of the 16nm FinFET Test Chip.

4.4 Conclusion

In this chapter, a mixed-signal architecture based on successive-approximation is presented as a energy efficient alternative to conventional digital signal processors for fixed-point multiplications and additions. Based on the widely-popular successive approximation (SA) ADC, this architecture utilizes a programmable capacitor array to compute products and sums, and then decodes into a digital value using a binary-search scheme. Invoking similar charge-based computing primitives as the mixed-signal VVM architectures presented in Chapter 3, this architecture takes a step forward, and combines the analog computational structure with the decoding structure to minimize computation time and energy cost. Designed in a 16nm FinFET process, the architecture was simulated to compute an 8-bit multiply-add operations at 6.85fJ averagely with a 0.4V supply voltage, and an average energy efficiency of

Table 4.3: Table of Pins for the 16nm FinFET Test Chip.

Pin	Core Port	Type	Description
DVDD	N/A	POW	External I/O power supply
VDD	N/A	POW	Internal I/O and core power supply
VDDI	VDD	POW	Power supply for the single core
DVSS	N/A	GND	External I/O ground
VSS	VSS	GND	Internal and core ground
VCM	Vcm	AI	B1 and B2 common-mode voltage bias
VM	Vm	AI	B1 and B2 negative input voltage bias w.r.t. Vcm
VP1	Vp	AI	B1 positive input voltage bias w.r.t. Vcm
VP2	Vp	AI	B2 positive input voltage bias w.r.t. Vcm
B1_IN[0]	clk	DI	B1 clock signal
B1_IN[1]	rst_i	DI	B1 reset signal
B1_IN[2]	en_i	DI	B1 enable signal for the serial I/O interface
B1_IN[3]	ld_i	DI	B1 load signal for the serial I/O interface
B1_IN[4]	d_i	DI	B1 data input signal
B1_OUT[0]	d_o	DO	B1 data output signal
B1_OUT[1]	d_o	DO	B1 done signal
B2_IN[0]	clk	DI	B2 clock signal
B2_IN[1]	rst_i	DI	B2 reset signal
B2_IN[2]	en_i	DI	B2 enable signal for the serial I/O interface
B2_IN[3]	ld_i	DI	B2 load signal for the serial I/O interface
B2_IN[4]	d_i[0]	DI	B2 data input signal
B2_IN[5]	d_i[1]	DI	B2 data input signal
B2_IN[6]	d_i[2]	DI	B2 data input signal
B2_IN[7]	d_i[3]	DI	B2 data input signal
B2_OUT[0]	d_o[0]	DO	B2 data output signal
B2_OUT[1]	d_o[1]	DO	B2 data output signal
B2_OUT[2]	d_o[2]	DO	B2 data output signal
B2_OUT[3]	d_o[3]	DO	B2 data output signal

146TOPs/W. Compared to a conventional digital architecture implementing the same operation with the same simulation specifications, the proposed design used 37% less energy. Furthermore, a test chip comprised of one SA multiply-add core for measuring single core performance and efficiency, and an array of 64 MAC processors, was designed and fabricated in the aforementioned process.

Chapter 5

Heterogeneous Chip Multiprocessor Design

Within the last few decades, transistor scaling, microarchitecture techniques, and cache memories have fueled exponential gains in microprocessor performance.[71] Through these advancements, single core processor performance has improved over three-orders of magnitude, and paved the way to more complex computing systems that can handle more complicated task. Nonetheless, with new challenges of diminished transistor-speed scaling and energy constraint, alternative design practices and approaches hinged on large-scale parallelism and heterogeneity have been essential for realizing recent performance and energy efficiency gains.[72]

Before the advent of chip multiprocessors (CMPs), skepticism of large-scale multiprocessor systems had been elucidated through Amdahl's law,[73] which attributes

performance limitations in a multicore processor to the nonparallel code/algorithm executed on the cores. Formally,

$$Speedup = \frac{1}{(1-f) + \frac{f}{N}}, \tag{5.1}$$

where f is the fraction of the code that is nonparallelizable, and N is the number of cores implemented. Equation 5.1 shows that as the fraction of executed code that can be parallelized approaches 1, the processor speedup tends to the number of cores N, and categorically, that this fraction is critical in the number of cores implemented in the CMP.

Extending Amdahl's law for energy efficient computing, design considerations with regards to number of cores and types of cores becomes even more critical. Symmetric multiprocessors with high single-thread performance can easily lose its energy efficiency as the number of cores increases, and adopting a heterogeneous multicore alternative that integrates specialized cores can yield more energy efficient solution.[74] Moreover, properly exploiting heterogeneity in CMP design, can not only leverage good performance and energy efficiency, but can also foster more generality for hardware reusability.

A comprehensive design methodology for architectural exploration in CMPs, that links parallel processing to traditional very large scale integration (VLSI) metrics, has been done to further extend Amdahl's law.[75] Through a cost function minimization,

a high-level architectural optimization approach is presented and demonstrated for maximizing energy-delay performance for fixed area constraints. Not only does their model analyze parallelism in CMPs, but also incorporates memory area utilization, hierarchy, and communication contention for capturing complex behaviors in CMP system design.

In conjunction to challenges faced for improving multicore performance and efficiency, limited bandwidth scaling for both off-chip and on-chip traffic has also become a critical concern.[76,77] With pin limitations and power constraints, bandwidth scaling has lagged behind the transistor scaling, and the rate of memory traffic generation has exceeded the rate that it can be serviced. Innovations in 3D chip integration,[71,78] such as through silicon via (TSV) chip stacking, have brought increased interconnectivity through the vertical dimension.

In recent years, high performance computing (HPC) and energy efficient systems have been developed to tackle more data-intensive tasks. Specifically, a custom application specific integrated circuits (ASIC) chip, the Tensor Processing Unit (TPU), has been designed for accelerating inference for machine learning applications in data centers.[79] Additionally, different neuromorphic chip multiprocessors (CMPs) and systems,[80–82] have been designed for energy efficient machine learning, multi-media processing, and neural simulation.

In this dissertation, work is done for the design of the 2.5D Nano-Abacus System-on-Chip, which is a heterogeneous multiprocessor chip designed in 55nm CMOS pro-

cess for large scale energy efficient data processing. Starting from the individual processing unit cores to the full system-on-chip with the network-on-chip and the I/O interface, this CMP was designed to efficiently compute, while also achieving good performance and generality. In this chapter, first the full system-on-chip is described. Then, some processing units (PUs) applicable for general-purpose signal processing are detailed along with a PU controller that facilitates distributive computing. After that, a scalable bufferless network-on-chip architecture is discussed with the presentation of a network simulator. Next, high speed pad I/O circuitry used for both a general purpose I/O (GPIO) interface and a 3D DiRAM HUB interface is described. Finally, a mixed-signal accelerator CMP is detailed with a few large-scale applications in image processing and machine learning.

5.1 2.5D Nano-Abacus System-on-Chip

The 2.5D Nano-Abacus SoC is comprised of 3 CMPs, each designed in 14.133μm by 17.466μm silicon area in a 55nm CMOS process, that are interconnected to a high memory bandwidth 3D dis-integrated random access memory (DiRAM) and a Xilinx Zynq 7100 FPGA on a 50mm by 64mm interposer in a 1μm process. This SoC is constructed as a system-in-package (SIP), where the chip modules are mounted onto the interfacing silicon interposer through a flip chip package of controlled collapse chip connections. As opposed to a true 3D chip design, which employs TSV connections

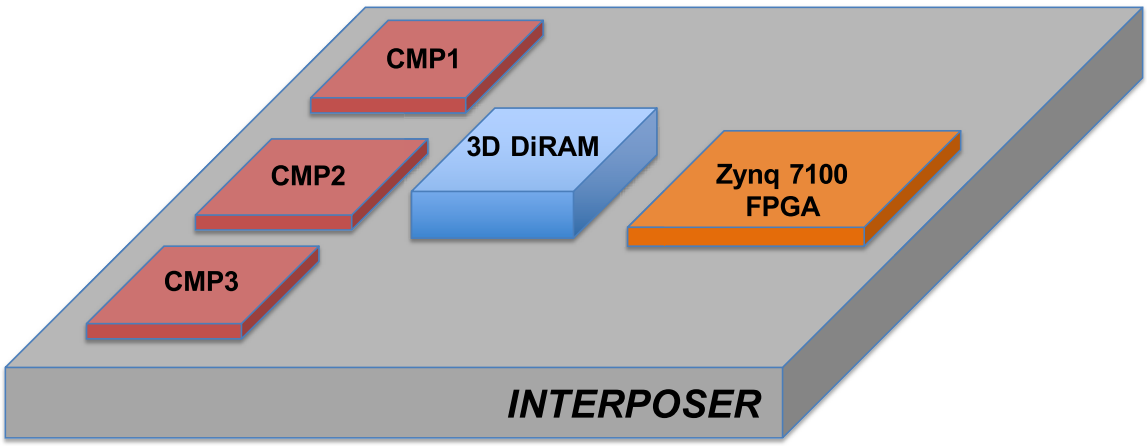


Figure 5.1: 3D View of the 2.5D Nano-Abacus System-on-Chip Design. Three CMPs, a 3D DiRAM, and a Zynq 7100 FPGA are mounted on a silicon interposer.

for multiple die stacking, this SoC is identified as a 2.5D chip that integrates this silicon interposer in a vertical configuration in order to facilitate a multi-chip module (MCM) design. Figure 5.1 shows the 3D view of the 2.5D SoC designed for large scale data processing.

The top view of this SoC with the chip interconnections is shown in Figure 5.2. Each of the 3 CMP and the FPGA have access to the 3D DiRAM through parallel high speed data buses connected through a double-data rate (DDR) PHY interface. Additionally, each CMP communicates to the on board FPGA through a GPIO interface that can be used for CMP configuration or as data ports. The FPGA also directly connects to I/O pads on the interposer (not shown on the diagrams) for external communication. All of the CMP footprints are identical on the interposer in order to promote modularity and system reconfigurability. Furthermore, the interposer stitches four 32mm by 25mm reticles together, and has five metal layers

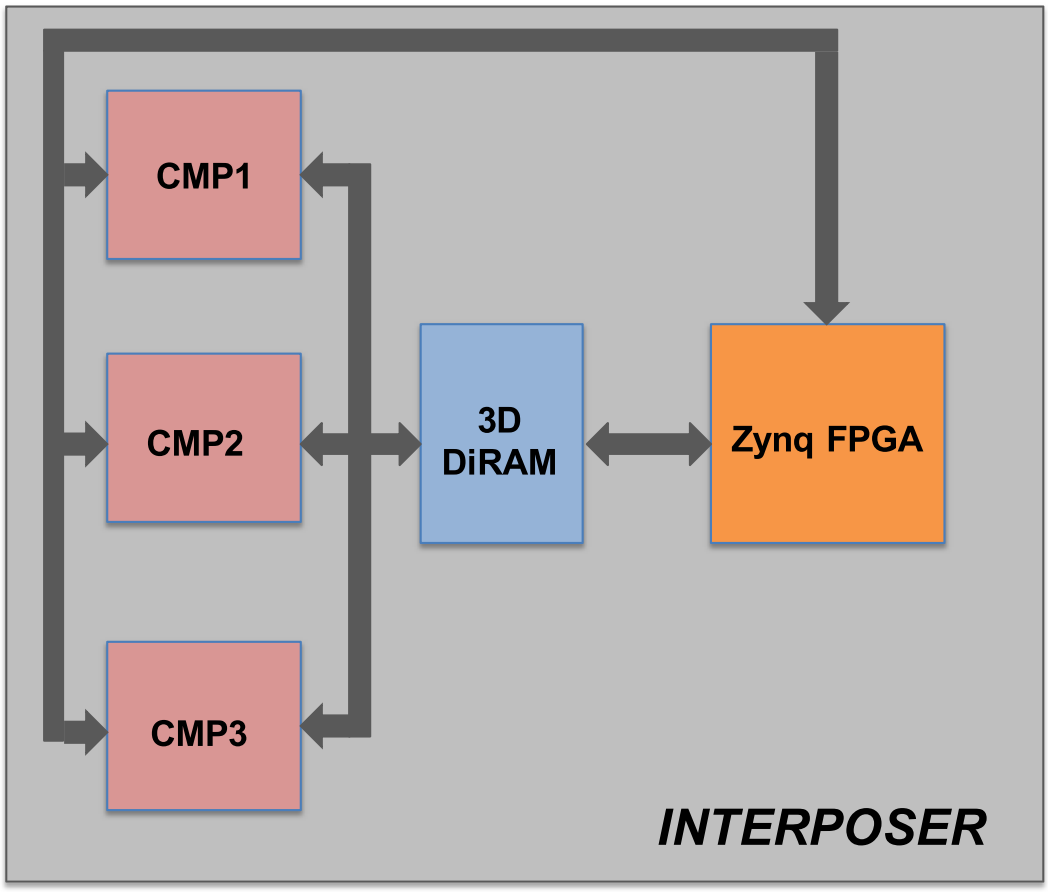


Figure 5.2: Top View of the 2.5D System-on-Chip Design. In addition to the three CMPs, 3D DiRAM, and the Zynq 7100 FPGA, the chip interconnections are also shown.

for routing. Because of the limited routing layers, the chip modules are arranged as shown in Figure 5.1 and 5.2 to maximize power routing efficiency across the different power domains and to minimize routing congestion between modules.

In this 2.5D multi-module chip, the 3D DiRAM, which features a 64-bit DDR interface running at a maximum frequency of 1.6GHz, was supplied by Tezzaron Semiconductors. The Zynq 7100 is a high-end FPGA equipped with ARM and floating point processors that was supplied by Xilinx. Moreover, the CMPs, which were

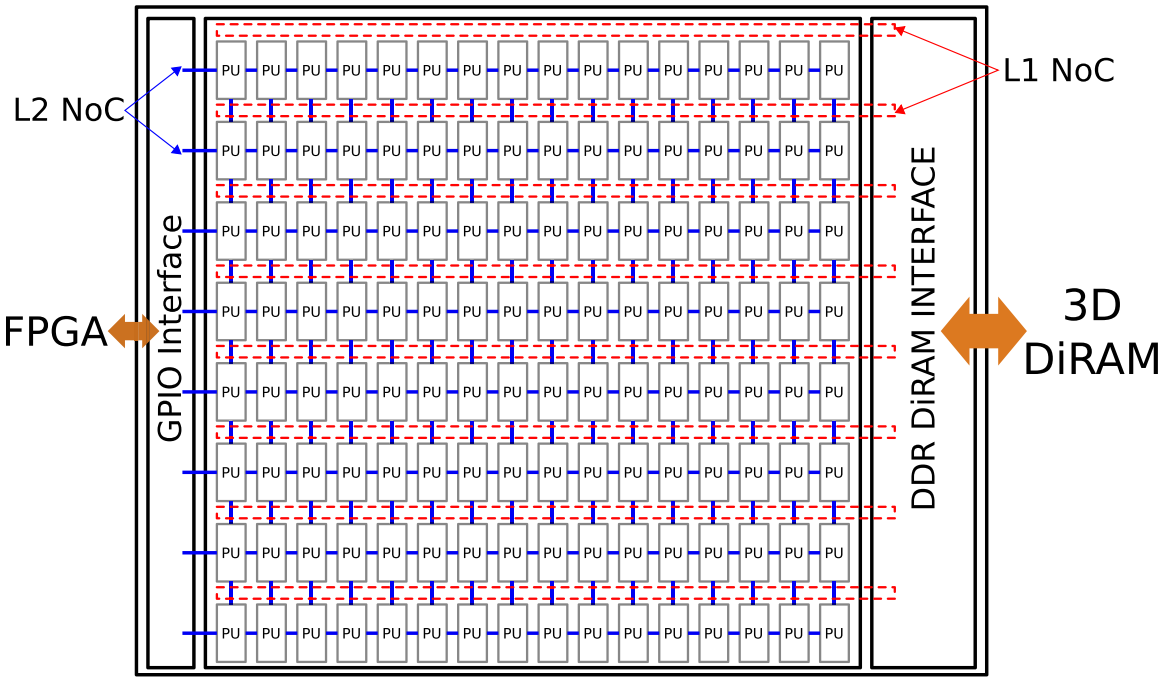


Figure 5.3: Modular Framework for the CMP Design.

specially designed in a 55nm process, are based on the modular framework shown in Figure 5.3. Each CMP is comprised of an array of specialized processing units, a high memory bandwidth interface (DDR DiRAM Interface) to main memory (3D DiRAM), a GPIO interface to the FPGA, a token ring network from the PUs to the DDR DiRAM interface (L1-NoC), and a mesh network (L2-NoC) for intercommunication between PUs and the GPIO interface. In this modular framework, the PUs have identical dimensions and interfaces to the NoCs, and can be interchanged or replaced within the 2.5D Nano-Abacus SoC. Additionally, the PUs have access to separate power and clock domains for optimizing performance and efficiency.

5.2 Processing Units

In the CMPs, the processing units (PUs) are specially designed to not only push performance for data-intensive task through the high bandwidth NoCs, but also promote computational efficiency through architecture and circuit optimization. In this dissertation work, three different PUs were designed — a mixed-signal processor with 128 cores for efficient fixed-point arithmetic, a computation memory controller that can be used for rotational and translational data manipulation, and an auxiliary memory unit that can work in conjunction with other PUs. Furthermore, a PU controller based on a simple instruction set processor is designed to facilitate distributive computing in the CMPs.

5.2.1 PU VVM: A Mixed-Signal Processor for Fixed-Point Arithmetic

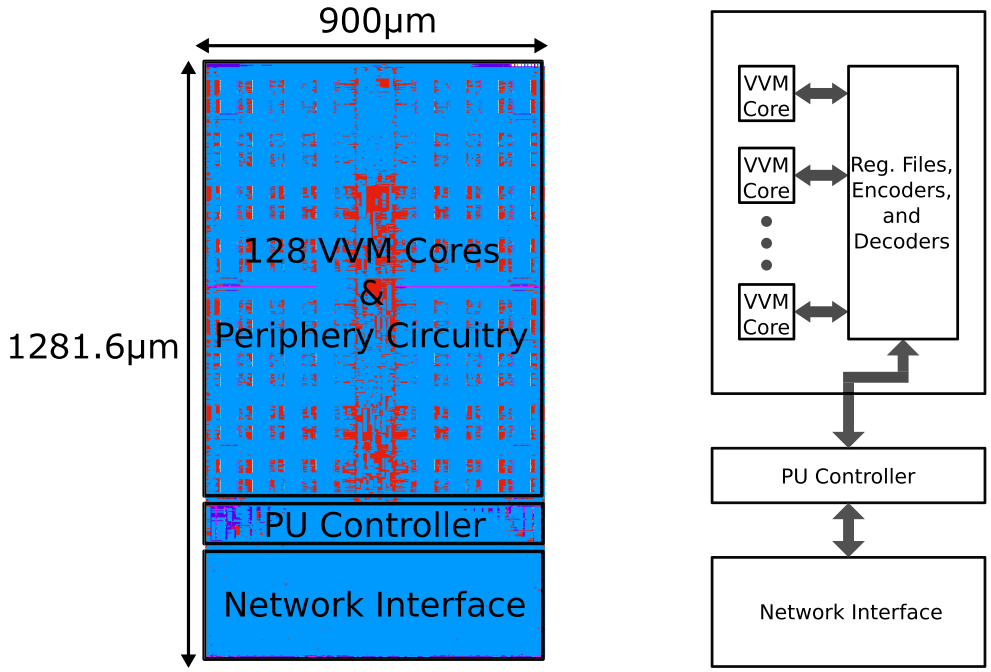
5.2.1.1 Overview

The processing unit vector-vector multiplier (PU VVM) is a mixed-signal accelerator comprised of 128 cores for computing fixed-point arithmetic. Exploiting charge-based computing and stochastic logic, each VVM core computes inner products at the thermal noise limit in the analog domain for minimal energy cost using a capacitor array. Subsequently, a first-order Sigma-Delta ($\Sigma\Delta$) modulator ADC is used to

Table 5.1: Simulated Characteristics of the PU VVM.

Technology	55nm CMOS
Core Area	$1.152mm^2$
Operation	8-bit MAC
Precision	8-bit
Throughput	225MOPs
Efficiency	28.6GOP/W (1.8TOP/W for the array)

decode the computation output back into the digital domain. Each VVM core, which comprises of this capacitor array and the $\Delta\Sigma$ ADC, is interfaced to periphery circuits for local weight and parameter storage, input data encoding, and post-processing for output scaling and thresholding. Furthermore, an instruction set-based processor is implemented as a programmable DMA and processor controller for programming the cores and I/O communication. This design capitalizes on the computational efficiency of the VVM cores for computing 8-bit precision inner products of vectors with up to 9 elements at 1.8TMAC/W on the capacitor array and 28.6GMAC/W totally. Additionally, this mixed-signal processor is capable of computing 225MOP/s, and can be used for numerous of signal processing and machine learning applications including, image filtering, fast-fourier transform (FFT), neural networks, and more. Table 5.1 shows the simulated characteristics of the PU VVM. Additionally, figure 5.4 shows an annotated layout view and an architectural diagram of the PU VVM.



(a) Annotated Layout (b) Architecture

Figure 5.4: Annotated Layout and Architecture of the PU VVM.

5.2.1.2 VVM Core Architecture

The VVM core is largely based on the GF5 VVM design (see Section 3.7, which uses a cascade of mixed-signal multipliers comprised of a capacitor and a few switches to compute and sum products in parallel as charge. This computation is done efficiently by not only computing in the analog domain, but also by computing probablistically using stochastic logic. The resulting charge can be converted back to the digital domain using a switched-capacitor $\Delta\Sigma$ modulator which simultaneously converts while computing on the array to give a trade-off of output precision and computation time.

The full VVM core circuit, which is adopted for signed inner products can be seen

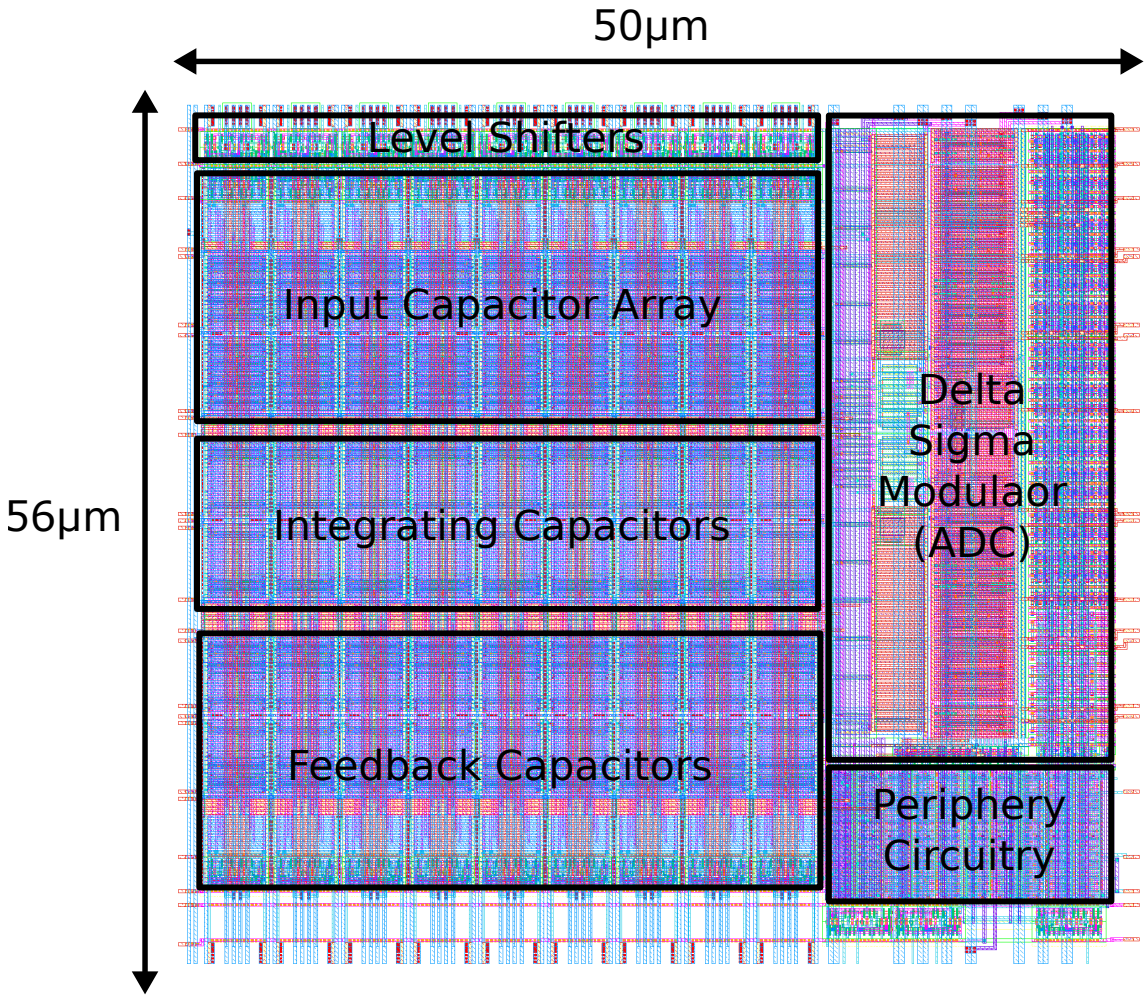


Figure 5.5: Annotated Layout of the PU VVM Core Design.

in the previous section in Figure 3.58. Moreover, the layout was revised to further mitigate parasitic factors from routing, and the annotated layout of the VVM core can be seen in Figure 5.5. The optimized core design, which measures 56μm by 50μm, is more compact than the GF5 VVM core design, which measured 64μm by 58μm. Additionally, level shifters were integrated in this core for lower supply voltage operation.

Furthermore, the nominal bias values for operation and measured characteristic are reported in Table 3.18 and 3.20. Operating on a 1.2V power supply at 16.67MHz in 0.0028mm^2 silicon area, the VVM core computes 8-bit MACs at a throughput of 580KOPs with an efficiency of 28.6GMAC/W.

5.2.1.3 PU VVM Programmability

The PU VVM was designed with the programmability of adjusting the clock rate, active cores, data type representation (unsigned or signed), random number generators, output offset, output integration count, output scaling, and output thresholding. Each of these programmable settings are discussed in more details below.

- ***Clock Rate***: A clock divider based on a 32-bit binary counter is implemented for modulating the VVM core clock rate. The VVM core, which is running on a separate clock domain may need to run slower when scaling the power supply, thus the clock divider is important for scaling the clock rate for optimal performance. Based on a 300MHz main clock, the clock divider can scale the VVM core clock rate from as high as 150MHz to as low as 0.06Hz.

- ***Active Cores***: Each of the 128 VVM cores in the PU VVM can be disabled, which deactivates the clock signals for the core and the counter for integrating the VVM core output. In the case that only a few cores are needed, this functionality helps minimize power consumption from unused cores.

- ***Data Type***: The data type representation of both the weight and input (either signed or unsigned) can be programmed, which modifies the encoders to properly generate the unary samples. This parameter guarantees that the full input domain is used based on the data type.

- ***Random Number Generators***: The 2 random number generators used in the PU VVM, one for the weight encoder and the other for the input encoder, can be modified to have a different starting seed and tap mask. The tap mask is used for configuring which bits of the LFSR are used in the feedback. This allows for different maximal-length LFSR patterns across different bit precision to be used with different seeds.

- ***Output Offset***: A 16-bit signed offset value can be programmed in the output counter for any of the 128 VVM cores. This allows for an implicit addition operation, which can also be useful for VVM core calibration for mitigating fabrication variations.

- ***Output Integration Count***: The count threshold for integrating the VVM output can be programmed for each core. This can be used for an implicit multiplication operation or implementing gain function by scaling the count threshold according.

- ***Output Scaling***: The 16-bit output counter, can be scaled by a power of 2 (for both multiplication and division) through bit shifting.

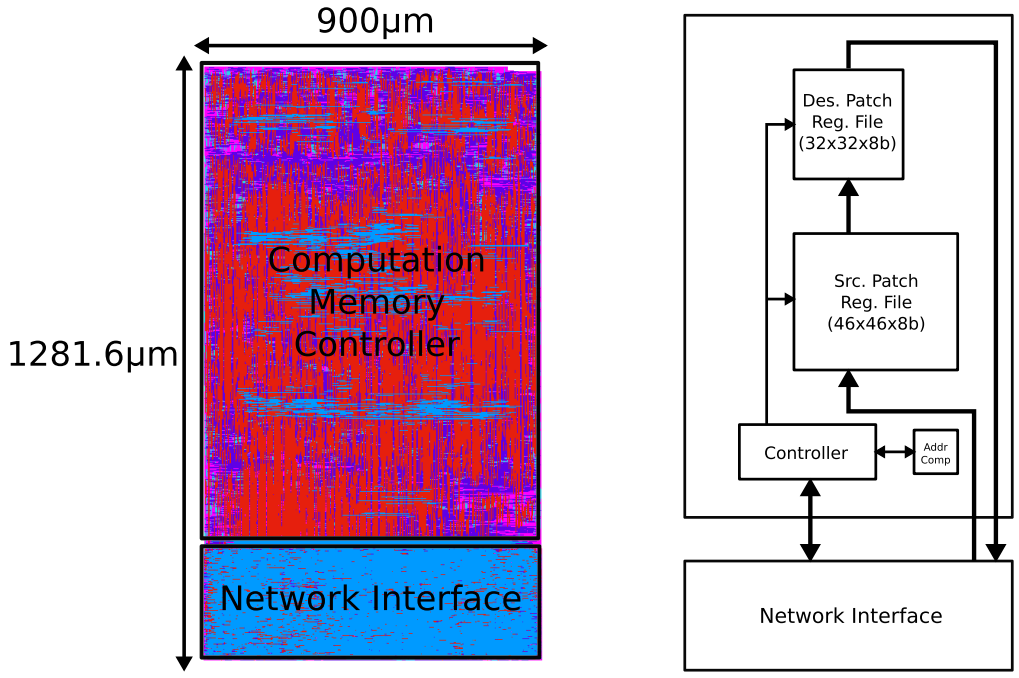
- ***Output Thresholding***: The integrated output on the counters from the VVM cores can be bounded by a programmable minimum and maximum value that is particularly useful for event detection applications.

5.2.2 PU CMC: A Computation Memory Controller

5.2.2.1 Overview

The computational memory controller processing unit (PU CMC) is a processor used for translational and rotational data manipulation through address remapping. Using a programmable rotation and translation matrix, this processor performs a destination-based data remapping, where each destination data point is derived from the source data point. In order to minimize remapping latency from memory fetch and store operations, this processor is designed to remap 2D patches of data with the same rotation and translation matrix per processing cycle. The architecture for this processor, along with the layout of the design can be seen in Figure 5.6.

For a 32 by 32 data patch, the processor computes the source addresses and fetches the bounding source patch of 46 by 46 data points. As depicted in Figure 5.7, this bounding patch encompasses all possible rotational points. The controller in the processor, which is a finite-state machine, is used to initiate the address computations and manage the read and write operations to and from main memory for data remapping.



(a) Annotated Layout (b) Architecture

Figure 5.6: Annotated Layout and Architecture of the PU CMC.

Table 5.2: Simulated Characteristics of the PU CMC.

Technology	55nm CMOS
Core Area	1.15mm^2
Data Precision	8bits
Address Precision	16bits
Supply Voltage	1.2V
Frequency	300MHz
Throughput	61.44MP/s

As shown in Figure 5.8 from post-layout simulations, this processor can be used for image DeWarping to correct for rotation and translational distortions. Furthermore, the design specifications of this processor can be seen in Table 5.2. This unit can produce 1024 rotational and translational remapped 8-bit data points per processing cycle (60KHz) for a total throughput of 61.44 million data points per second.

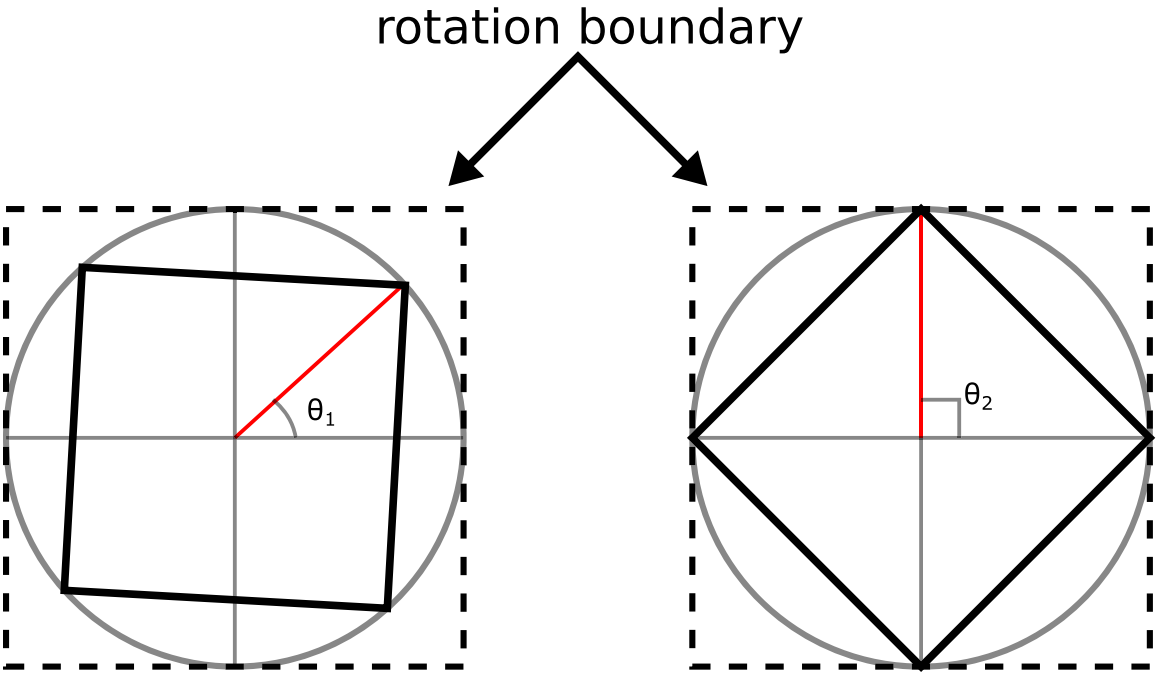


Figure 5.7: Rotational Boundary for Points Enclosed in the Unit Circle.

(a) Warped (b) DeWarped

Figure 5.8: Image DeWarping Using the PU CMC. A rotated and translated image (left) is rotated and centered (right) using the PU CMC.

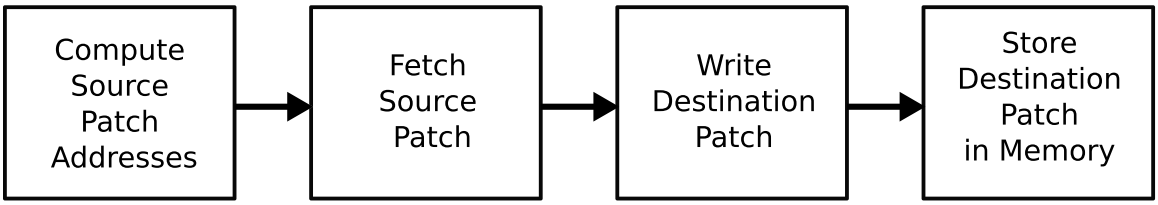


Figure 5.9: Processing Flow for the PU CMC.

Nonetheless, the processing cycle is dependent on the memory fetch and store latency, and can having slower cycles with heavy traffic on the network.

5.2.2.2 Processing Flow

Data remapping with the computational memory controller can be broken down into 4 main processing steps — Compute Source Patch Addresses, Fetch Source Path, Write Destination Patch, and Store Destination Patch in Memory. Figure 5.9 shows this pictorial view of this flow.

During the first stage, **Compute Source Patch Addresses**, the source addresses are computed based on the address rotation and translation matrix. From trigonometry, a point (x_1, y_1) on a plane, will have the relationship

$$\begin{aligned} x_1 &= rcos(\theta_1) \\ y_1 &= rsin(\theta_1) \end{aligned} \tag{5.2}$$

with respect to a radius r and angle θ_1. If that point is then rotated by an angle θ,

the new point (x_2, y_2) will have the relationship

$$\begin{aligned} x_2 &= rcos(\theta_1 + \theta) \\ y_2 &= rsin(\theta_1 + \theta) \end{aligned} \tag{5.3}$$

with respect to the new angle $(\theta_1 + \theta)$. By applying the angle addition identities, the new point can be deduced as

$$\begin{aligned} x_2 &= x_1 cos(\theta) - y_1 sin(\theta) \\ y_2 &= x_1 sin(\theta) + y_1 cos(\theta) \end{aligned} \tag{5.4}$$

Combing this with a translational shift of (x_0, y_0), the address function computed for this processor is given as

$$\begin{aligned} x_2 &= x_1 cos(\theta) - y_1 sin(\theta) + x_0 \\ y_2 &= x_1 sin(\theta) + y_1 cos(\theta) + y_0 \end{aligned} \tag{5.5}$$

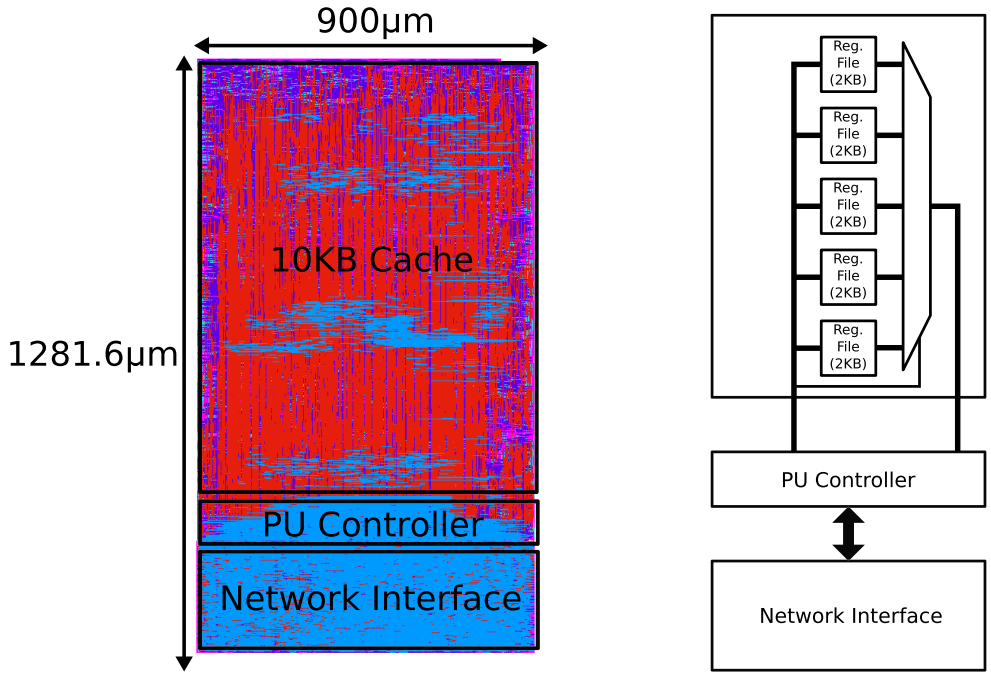
The $sin(\theta)$ and $cos(\theta)$ values are precomputed and stored before each processing cycle along with the translational parameters x_0 and y_0. Thus, the first stage of the computational memory controller then computes this multiply and addition operations for generating the addresses for each 1024 data point. In addition to that, it also keep track of the minimum and maximum values for each dimension to be used for the next stage of fetching the source data patch.

In the next stage, **Fetch Source Patch**, the controller makes a burst of read requests from the main memory in order to populate the source patch register files. This is done simply through looping from the minimum x and y source address to the maximum x and y source address. Following that, the controller then performs the **Write Destination Patch** operation, which cycles through the destination addresses in consecutive ascending order, and assign the data point according to the corresponding source address. Finally, in the **Store Destination Patch in Memory** phase, the destination data points are written back to main memory through a burst of write requests through the network.

5.2.3 PU CACHE: Auxiliary Memory Unit

5.2.3.1 Overview

The cache processing unit (PU CACHE) is an auxiliary buffer/cache unit used for local data storage for the other processing units. A 10KB cache, composed of an aggregate of five 2KB register files, is interfaced to a controller module, which facilitates read and write operations to and from main memory to this cache and other PU modules. The annotated layout view and block architecture for the PU CACHE is shown in Figure 5.10. In order to achieve high speed performance, register files are used as the memory banks for the cache. As shown in the block architecture, a simple cache topology is adapted that shifts the responsibility of cache coherency



(a) Annotated Layout (b) Architecture

Figure 5.10: Annotated Layout and Architecture of the PU CACHE.

to the software level of description with the PU controller. Thus, cache read and write misses can be completely circumvented through proper configuration of the instruction calls for the PU controller of the PU CACHE.

The full cache unit was designed to run at 300MHz with a data bus width of 256 bits. The unit can achieve a read and write throughput of 76.8Gb/s with a maximum read and write latency of 2 clock cycles. The full design specifications can be seen in Table 5.3.

Table 5.3: Simulated Characteristics of the PU CACHE.

Technology	55nm CMOS
Core Area	1.15mm^2
Supply Voltage	1.2V
Frequency	300MHz
Throughput	76.8Gb/s
Read/Write Latency	6.7ns

5.2.3.2 2KB Register File Design

The 2KB register files used in the cache were synthesized from generic VHSIC hardware description language (VHDL) code that exploits register duplication in order to meet timing closures. In the conventional implementation of the output decoder, the read address control signal drives logic that selects the output bits from the rows of the register files. An example of this decoder for 16 row N-bit width register file is shown in Figure 5.11. As the fan-out grows with the data width N and the number of rows, meeting timing closures for place and routing this design becomes more difficult. To avoid this scaling issue, an alternative solution is used that systematically duplicates the read address signals in order to minimize fan-out and maintain good performance for large register files. An example of this alternative decoder design for the same register file in Figure 5.11 can be shown in Figure 5.12. In this implementation, the 16-1 N-bit multiplexer is broken into a multiplexer tree with the select signals driven by duplicated registered read address bits. Specifically, the lower 2 bits of the read address, which drive a higher fan-out than the upper 2 bits, is replicated into 4 duplicates that drive separate 4-1 N-bit multiplexer. Logically, both

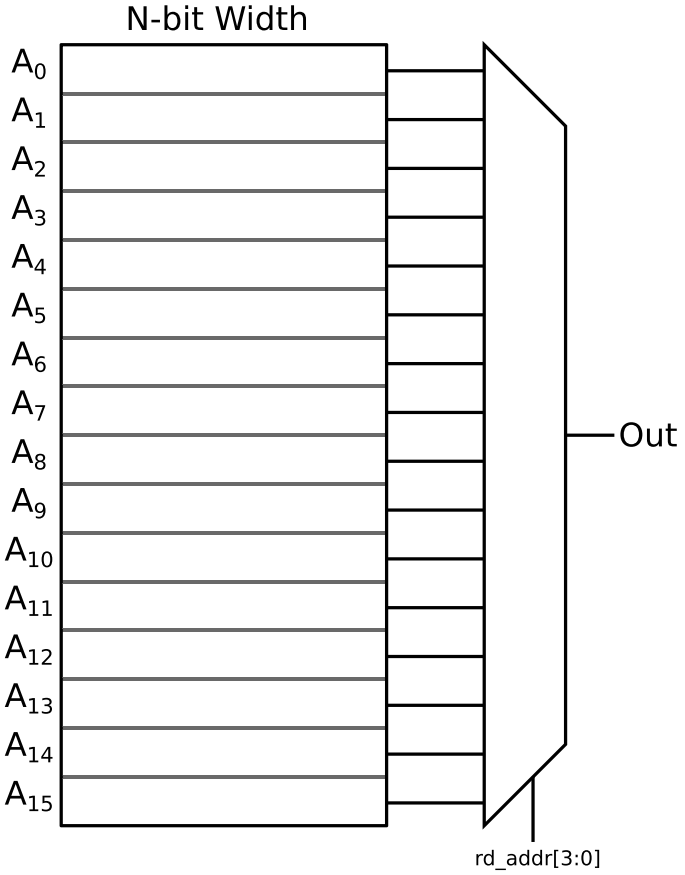


Figure 5.11: Output Decoder for a 16 Row N-bit Width Register File.

decoders are equivalent, but the the duplication of those read address bits spreads the fan-out load, and makes its easier to place and route the design to meet performance. This approach can be generalized to different size register files, where the multiplexer tree and the number of read address duplication are chosen accordingly. The number of stages for the multiplexer tree N_S is given as

$$N_S = log_K(N_R), \tag{5.6}$$

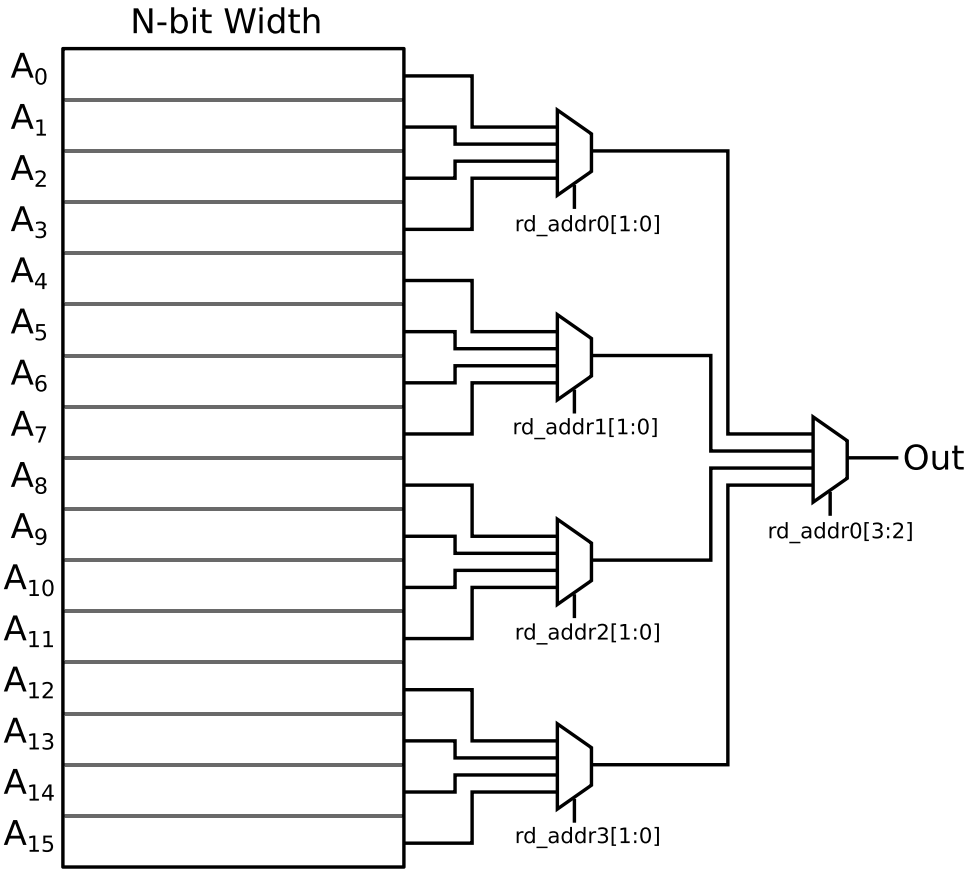


Figure 5.12: Output Decoder for a 16 Row N-bit Width Register File with Read Address Duplication.

where K is the number of rows partitioned in the register file that share a duplicated read address and N_R is the total number of rows. To guarantee a symmetric decoder, N_R should be a power of the base K.

The VHDL code for this 2KB register file is detailed below.

```
entity data_mem is
  generic(
    DEPTH : integer := 64;
    WIDTH : integer := 256
  );
  port(
    clk_i : in std_logic;
    wr_en_i : in std_logic;
```

```
    wr_addr_i : in std_logic_vector(log2(DEPTH)-1 downto 0);
    rd_addr_i : in std_logic_vector(log2(DEPTH)-1 downto 0);
    data_i  : in std_logic_vector(WIDTH-1 downto 0);
    data_o  : out std_logic_vector(WIDTH-1 downto 0)
  );
end data_mem;

architecture arch of data_mem is

constant AWIDTH : integer:= log2(DEPTH);
type phase_address_type_16 is array (0 to 16-1) of
 std_logic_vector(AWIDTH-1 downto 0);

signal rd_addr_reg : phase_address_type_16;

type reg_file_type_16 is array (0 to 16-1) of
 std_logic_vector(WIDTH-1 downto 0);

type reg_file_type_4 is array (0 to 4-1) of
 std_logic_vector(WIDTH-1 downto 0);

signal reg_file_16 : reg_file_type_16;
signal reg_file_4 : reg_file_type_4;

type mem is array (DEPTH-1 downto 0) of
 std_logic_vector(WIDTH-1 downto 0);

signal mem_mod : mem;
signal wr_en_reg : std_logic;
signal wr_addr_reg : std_logic_vector(AWIDTH-1 downto 0);
signal data_reg : std_logic_vector(WIDTH-1 downto 0);

begin

--process for registering inputs and assigning data to the reg file
process(clk_i) is begin
  if(rising_edge(clk_i))then
    wr_addr_reg <= wr_addr_i;
    wr_en_reg <= wr_en_i;
    data_reg <= data_i;
    if(wr_en_reg = '1')then
      mem_mod(to_integer(unsigned(wr_addr_reg))) <= data_reg;
```

```
    end if;
  end if;
end process;

--duplicate read address to minimize fan-out
gen_rd_addr : for ii in 0 to 16-1 generate
  process (clk_i) is begin
    if(rising_edge(clk_i))then
        rd_addr_reg(ii) <= rd_addr_i;
    end if;
  end process;
end generate gen_rd_addr;

--1st stage of data out assignment pipeline
gen_16_out : for ii in 0 to 16-1 generate
  reg_file_16(ii) <= mem_mod(ii*4+
      to_integer(unsigned(rd_addr_reg(ii)(1 downto 0))));
end generate gen_16_out;

--2nd stage of data out assignment pipeline
gen_4_out : for ii in 0 to 4-1 generate
  reg_file_4(ii) <= reg_file_16(ii*4+
      to_integer(unsigned(rd_addr_reg(ii)(3 downto 2))));
end generate gen_4_out;

--final stage of data out assignment pipeline
data_o <= reg_file_4(
    to_integer(unsigned(rd_addr_reg(0)(5 downto 4))));

end arch;
```

In this hardware description, the input control signals — write enable, write address, and data input — are registered to avoid setup time violations from input delay. The read address is duplicated on 16 registers, which are used to address the data word from the 64 rows. The multiplexer tree for the output decoding is designed in 3 stages for groups of 4 rows. This design was synthesized and placed and routed to run at

300MHz in the 55nm process.

5.2.4 PU Controller

Through the high bandwidth interfaces and the numerous of PUs integrated in this SoC design, the CMPs are capable of locally processing copious amount of data. Nonetheless, how these PUs and the interface work in tandem to deliver the high performance is critical in the full design of these CMPs. To address this, a central processing unit (CPU), composed of general-purpose ARM Cortex M0 cores, and a direct memory access (DMA) controller are also integrated in the system design. However, a dire bottleneck arises when this CPU is tasked with micromanaging data transfers and intermediate unit programming for hundreds of PUs on the chip. Furthermore, scaling the number of M0 cores just to address specific memory and unit operations is not only redundant, but also wasteful for area and energy. In this dissertation work, a custom PU controller with a simple, yet adequate instruction set, is designed to facilitate distributive processing so the unit can operate independently and efficiently with the interfaces. As shown in Figure 5.13, the PU controller communicates to the core through the PU core interface, to other PUs and the GPIO interface through the L2 network, and to the 3D DiRAM through the L1 network.

In the following sections, first the instruction set architecture (ISA) is explained. Then, the three different interfaces are described. Finally, applications of the PU controller are detailed along with a few example instruction call flows.

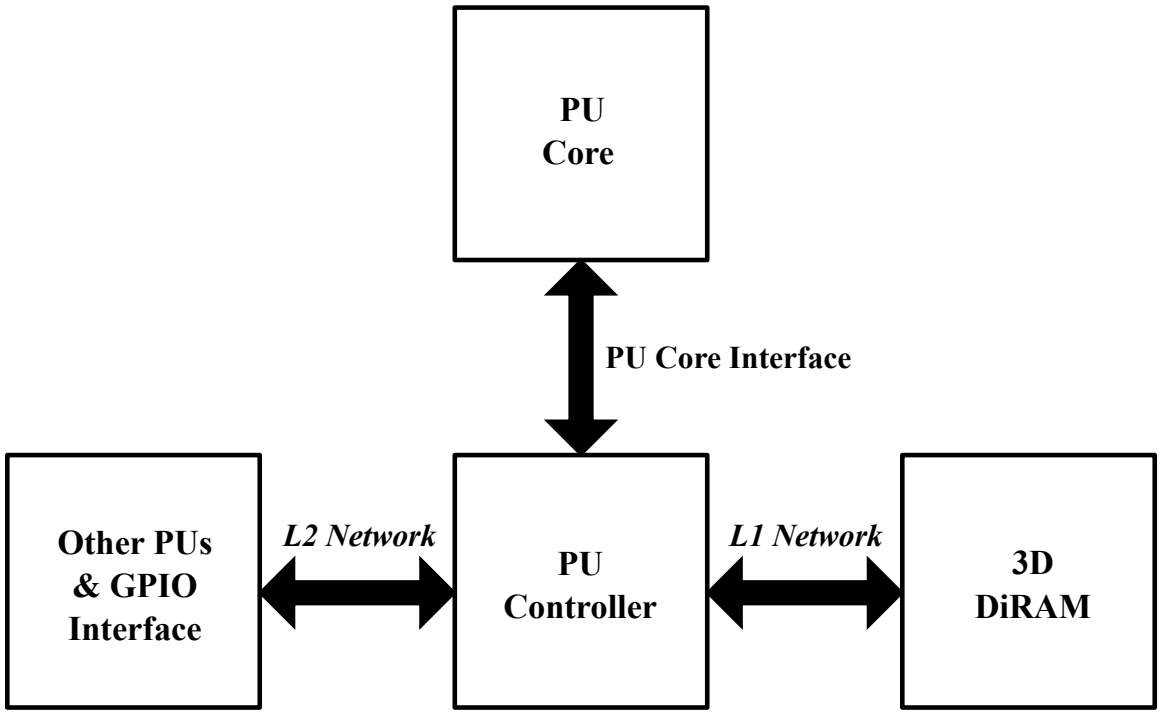


Figure 5.13: Block Diagram of the PU Controller Interface.

5.2.4.1 Architecture

In order to provide a generic interface that is robust for various read/write patterns for the PU cores, an instruction set-based processor is adapted for the PU controller. This processor is composed of an instruction set that works in conjunction with a network and core ports for generic reading and writing through both the token ring (L1) and mesh (L2) network. Based on only 8 unique instructions, the PU controller can perform read and write operations with addresses that span multiple dimensions. Additionally, the controller can perform data manipulation operations that shifts the bytes in a data word for data alignment. There is also a wait instruction, which can be used for synchronization across PUs. Figure 5.14 shows a breakdown of the PU

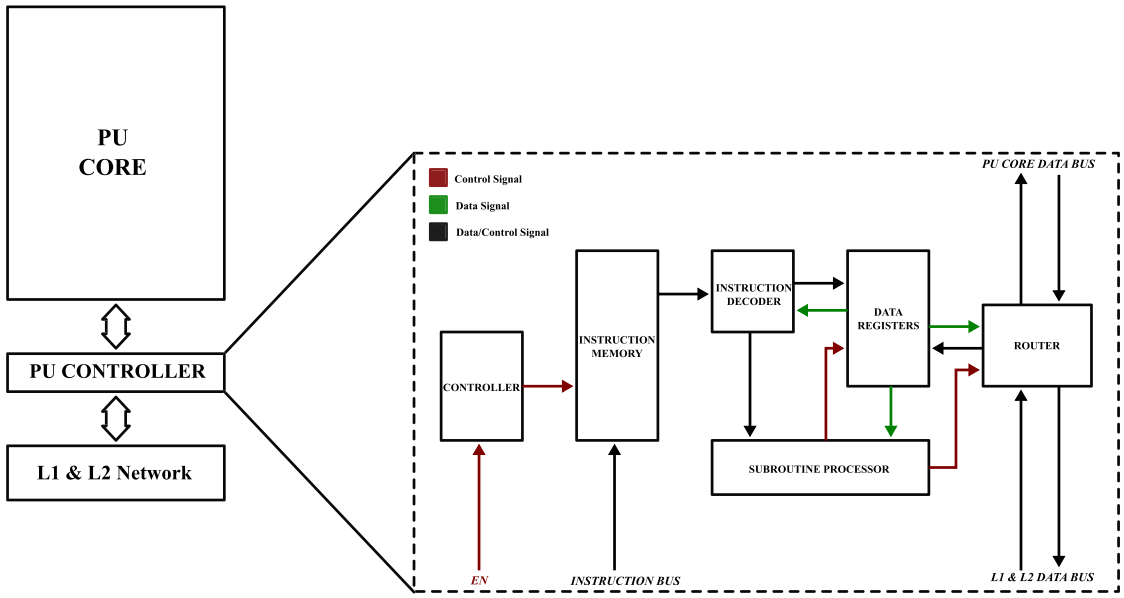


Figure 5.14: PU Controller Architecture.

controller architecture. In this architecture, the controller initiates the read out of instructions stored in the instruction memory. The read out instructions are then decoded with the instruction decoder, and depending on the opcode, either the registers are set or incremented/decremented, instructions are looped, or a subroutine is initiated. Both the subroutine processor and the data registers communicate with the router for reading and writing data from either the main memory (3D DiRAM), other PU cores, or local data registers. A generic design is adapted for the PU controller, where the size of the data register files and the instruction memory can be optimized for a PU.

This PU controller was implemented in the PU VVM described in Section 5.2.1. As seen in Figure 5.15, the controller has less than a 10% area overhead in the full

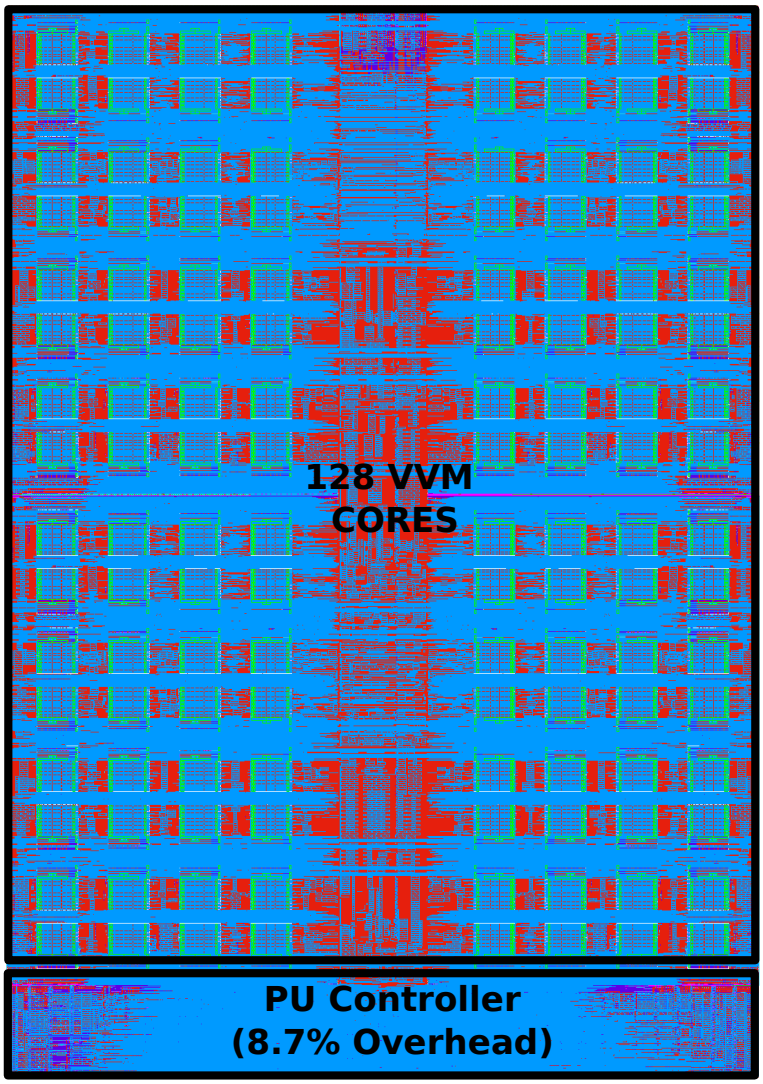


Figure 5.15: Layout of the PU VVM with the PU Controller.

PU VVM design.

Furthermore, the full instruction set for the PU controller can be seen in Table 5.4. The first 3 instructions, *LOAD*, *ADD*, and *LOOP*, are the basic instructions used for setting the data registers and counters for generating the read and write address patterns. Using these instructions, addresses can be generated across multiple dimension. Particularly, this is essential for 2d or 3d image processing. The

Table 5.4: Instruction Set for the PU Controller.

INST.	OPCODE	DESCRIPTION	PARAMETER(s)
LOAD	0	Sets one of the data registers	Address, Data
ADD	1	Adds or subtracts data registers	Address, Data
LOOP	2	Loop to an instruction	Loop Sel, Inst Address
READ	3	Read request to L1 or buffer	Mode
WRITE	4	Writes to L1 or L2	Mode
SHIFT	5	Circular byte shift in buffer	Num Shift
WAIT	6	Stalls instruction calls	Mode, Count
DONE	7	End instruction calls	

next 4 instructions, *READ*, *WRITE*, *SHIFT*, and *WAIT*, are subroutine instructions that execute the main function of this architecture. The final instruction, *DONE*, is used as an identifier for the end of instructions calls. There are total of **24 bits** per each instruction — 3 bits of opcode, and potential 21 bits for setting parameters and registers.

Each instruction is explained in more detail below.

- **LOAD**: This is a basic instruction that sets one of the registers in the PU controller to a value (16-bit) indicated in the instruction word. Currently there are potentially up to 31 unique registers (5 bit address) that can be set with this instruction; they are described in Table 5.5. Registers that are more than 16-bits are segmented into multiple addresses.

- **ADD**: This is another basic instruction that adds a signed value to one of the registers. Using this instruction the address can be incremented or decremented for address generation across dimensions. Since not all the registers listed in Table 5.5 need to be incremented or decremented (e.g the byte mask), fewer

Table 5.5: Registers for the PU Controller.

NAME	#BITS	ADDRESS	DESCRIPTION
l1_rd_addr	28	0 & 1	Main memory read address
l1_wr_addr	28	2 & 3	Main memory write address
l1_addr_inc	28	4 & 5	Increment value for L1 address
l2_addr	8	6	Destination L2 address (row and col)
pu_wr_addr	16	7	PU write address
pu_rd_addr	16	8	PU read address for write operations
byte_mask	32	9 & 10	Byte mask for writes to memory
rd_cnt	16	11	Packets received counter for L1 and L2
wr_sel	2	12	Write select for buffer
loop_cnts	16	16-31	Loop counters (Up to 16)

registers are addressable. Additionally, since the L1 address exceeds 16 bits, the increment value must first be set, and an ADD instruction with reference to the L1 address will add the signed values of the L1 address and the signed increment value. Registers that can be added or subtracted with this instructions include: *l1_addr*, *l2_addr*, *pu_wr_addr*, *pu_rd_addr*, and *pu_ctrl* (arranged in ascending address order)

- **LOOP**: This is essentially a conditional jump instruction that changes the program counter (read pointer) to the specified instruction address as long as the selected counter is not equivalent to 0. Primarily this instruction is used to loop through a set of instructions for tasks such as address generation across multi-dimensions. Each time the loop instruction is executed, the selected counter (*loop counters*) is decremented, and if the count value isn't 0, the read pointer is updated to the address specified in instruction word. If 0, the read pointer is incremented by 1 instead (similar to other instructions). Currently, there are 5

different loop counter that can be selected, and they can be set using the LOAD instruction.

- **READ**: This instruction is used to send a read request to L1 (main memory) or to read data stored in the interface buffer directly to the local cache of the PU core (specified with the *wr_sel* register). Since the L1 network and the PU controller are on two different clock domains, the 4-phase handshaking protocol is used during this subroutine for sending the read request. The interface stalls at this instruction until the request has been acknowledged by the L1-network. Because the subroutine process runs independently of the router, successive read instructions can be called consecutively even though there may be a delay in the arrival of the data. The *WAIT* instruction, described below, can then be used to assure all the data was received before executing additional instructions. Furthermore, the L1 read address and the destination address (PU core register or data buffer) is specified beforehand with the *LOAD* & *ADD* instructions.

- **WRITE**: This instruction can either send a write request to L1 (main memory) or to L2 (M0, FPGA, and other PU cores). Similar to the READ instruction, the interface sends these write request through a 4-phase handshaking protocol, and until the NoC sends an acknowledge back does the interface end this subroutine. Moreover, the source address for the data and the destination address for the write are specified beforehand with the *LOAD* & *ADD* instructions.

- **SHIFT**: The shift is a special instruction utilized with the *READ* instruction for offsetting the data word to be written to the PU core. Since the L1 & L2 data word has a size of 256bits, this instruction is particularly useful for reading smaller size data (e.g 8bits) from the word. Data stored in the interface buffer, which is of size 512bits (2 data words), is circularly shifted-right serially for a certain number of bytes (specified in the instruction word).

- **WAIT**: The wait instruction is used for stalling the execution of future instructions based on a set of conditions. This subroutine operates in 3 modes:

  - *Count Wait* : Waits for a certain number of clock cycles, which is set by a 20-bit counter in the instruction word (used for waiting for the core to finish processing)

  - *Read Wait* : Waits for the PU controller to receive the specified number of packets from L1 and/or L2

  - *M0 Wait* : Waits for a flag from the M0 processor. Additionally, for this wait, an acknowledge is transmitted to the M0 to signal that the PU core is waiting for the M0 to continue.

- **DONE**: This is an idle instruction, which signifies the end of instruction executions for the PU controller. The interface waits for the *start* control flag, a control signal used to start the execution of instructions, to resume again.

Table 5.6: PU I/O Ports to the PU Controller.

NAME	TYPE	#BITS	DESCRIPTION
pu_rst_i	IN	1	PU reset signal
pu_wr_en_i	IN	1	Write enable for data writes
pu_proc_en_i	IN	1	Process enable for the PU core (if necessary)
pu_wr_addr_i	IN	16	PU Write address and/or control for input
pu_rd_addr_i	IN	16	PU Read address and/or control for output
pu_data_i	IN	256	Input data
pu_data_o	OUT	256	Output data

5.2.4.2 PU Core Interface

A PU using the PU controller communicates through a set of ports detailed in Table5.6. The PU core has an input and an output data bus of 256 bits, and the PU controller controls when data is written to the PU core's local memory through a write enable port. A 16-bit write address and read address ports are used for indexing the PU core's local input and output memory respectively. Both address ports are unconstrained and can be used for selecting addresses or even decoding the data on the data bus. There is also a process enable port that can be used to enable processing with the PU core.

5.2.4.3 L1 Network Interface

The PU core communicates to the L1 network through an interface designed in the PU controller. The summary of the ports in this interface is detailed in Table 5.7. Data going to and from main memory through the L1 network to the PU core goes through a 256-bit data bus, which is read and written on the network through

Table 5.7: L1 I/O Ports to the PU Controller.

NAME	TYPE	#BITS	DESCRIPTION
DATA IN			
N1_PU_data_i	IN	256	Input data from the L1 network
N1_PU_addr_i	IN	256	L1 address
N1_PU_tag_addr_i	IN	24	PU write address
N1_PU_req_i	IN	1	Request in signal
N1_PU_ack_o	OUT	1	Acknowledge out signal
DATA OUT			
PU_N1_data_o	OUT	256	Output data
PU_N1_op_o	OUT	2	command (read (01) and write (11))
PU_N1_addr_o	OUT	40	L1 address
PU_N1_tag_addr_o	OUT	24	PU write address
PU_N1_req_o	OUT	1	Request out signal
PU_N1_ack_i	IN	1	Acknowledge in signal

a 4-phase handshaking protocol. For this protocol, the sender makes a request to the receiver, and this request set low when the receiver transmits an acknowledge back to the sender. Accompanying this data is a 16-bit tag that identifies where data coming from main memory is stored in the PU core. Furthermore, the PU controller transmits read and write requests to the main memory, and in order to make this request, a 2-bit command is sent, along with the L1 address for reading/writing, and the previously discussed data and tag data is transmitted through the network.

5.2.4.4 L2 Network Interface

The PU core can communicate to other PUs through the L2 network. Using the PU controller, this is done through a set ports detailed in Table 5.8. Similar to the L1 network interface, data is transmitted to and from a PU core to other PU cores

Table 5.8: L2 I/O Ports to the PU Controller.

NAME	TYPE	#BITS	DESCRIPTION
DATA IN			
N2_PU_is_data_i	IN	1	Configuration or data packet identifier
N2_PU_data_i	IN	256	Input data
N2_ver_addr_i	IN	256	L2 row address
N2_hor_addr_i	IN	256	L2 column address
N2_PU_reg_addr_i	IN	10	Input PU address
N2_PU_reg_part_i	IN	6	Input PU identifier
N2_PU_req_i	IN	1	Request in signal
N2_PU_ack_o	OUT	1	Acknowledge out signal
DATA OUT			
PU_N2_is_data_o	IN	1	Configuration or data packet identifier
PU_N2_data_o	OUT	256	Output data
PU_N2_dest_ver_addr_o	OUT	8	L2 row address
PU_N2_dest_hor_addr_o	OUT	8	L2 column address
PU_N2_reg_addr_o	OUT	10	Output PU address
PU_N2_reg_part_o	OUT	6	Output PU identifier
PU_N2_req_o	OUT	1	Request out signal
PU_N2_ack_i	IN	1	Acknowledge in signal

through a 256-bit bus in the L2 network interface. This bus sends and receives either data packets when $l2\_is\_data = 1$, and sends and receives configuration packets, such as instructions for the data interface processor, when $l2\_is\_data = 0$. The remaining ports for the L2 network interface include, the signals for the 4-phase handshaking protocol for sending and receiving data through the network and address tags that specify the destination of the data being transmitted.

5.2.4.5 Configuration

The PU controller is programmed through configuration packets sent via the L2 network from a configuration unit (CU). The CU, which can be the main CPU or

Table 5.9: Configuration Packet for the PU Controller.

—	**Misc. word**	**Acknowledge word**	**Instruction Word**
[255:192]	[191:128]	[127:64]	[63:0]

Table 5.10: Instruction Word for the PU Controller.

—	**inst**	—	**wr_addr**	**num_inst**	**cu_addr**	**wen**	**en**	**req**	**i_rst**	**set**	**rst**
[63:56]	[55:32]	[31:30]	[29:22]	[21:14]	[13:6]	[5]	[4]	[3]	[2]	[1]	[0]

Table 5.11: Acknowledge Word for the PU Controller.

—	**ver_addr**	—	**hor_addr**	—	**ack3**	**ack2**	**ack1**	**ack0**	**ack_en**
[63:43]	[42:40]	[39:37]	[36:32]	[31:5]	[4]	[3]	[2]	[1]	[0]

even the external FPGA, is responsible for sending these configuration packets, and also managing the acknowledges from the PU. A breakdown of the configuration packet can be seen in Table 5.9. The instruction word in this configuration packet is described in Table 5.10. Also, the acknowledge word in this configuration packet is described in Table 5.11.

Configuration packets with instruction words and acknowledge words are sent between the CU and the PU controller, in order to program that PU to execute a set of memory and processing operations. The protocol for this configuration is detailed below.

1. Firstly, the CU **SENDS** a configuration packet to reset the PU controller, and put it in configuration state by setting *rst* high in the instruction word.

   - **rst = 1** (instruction word)

2. Next, the CU **SENDS** a configuration packet to start the configuration. In the

configuration packet, the set bit in the instruction word is pulled high, and the configuration address *cu_addr* is set so the PU controller knows which address to send the acknowledge packet (this allows for robustness of configuration from different unit). Additionally, the number of instruction *num_inst* that will be sent to the PU controller is set accordingly.

- **set = 1** (instruction word)
- **cu_addr = [CU ver. addr & CU hor. addr]** (instruction word)
- **num_inst = total # instructions** (instruction word)

3. After that, the CU waits to **RECEIVE** an acknowledgement that the PU controller is done with the first configuration phase.

   - **ack_en = 1** (acknowledge word)
   - **ack0 = 1** (acknowledge word)

4. After acknowledgment of the first configuration, the CU **SENDS** all the instructions sequentially to the PU controller.

   - **wen = 1** (instruction word)
   - **wr_addr = (PU instruction memory address)** (instruction word)

- **inst = instruction** (instruction word)

5. Then, the CU waits to **RECEIVE** an acknowledgment that the PU received all the instructions concluding the configuration of the PU controller.

   - **ack_en = 1** (acknowledge word)
   - **ack1 = 1** (acknowledge word)

6. The instructions have been loaded to program, and the PU controller can begin executing instructions. This is done by setting the *en* bit high.

   - **en = 1** (instruction word)

7. While executing the instructions for running the PU, the PU may execute an instruction to synchronize with other PUs while suspending future instruction execution.While interrupted, the PU controller sends an acknowledge packet back to the CU. The PU will continue after it receive a *req* from the CU.

   - **ack_en = 1** (acknowledge word)
   - **ack2 = 1** (acknowledge word)
   - **req = 1** to continue instruction execution (instruction word)

8. Finally, after all instructions are executed, a final acknowledgment is sent from the PU controller back to the CU. The instructions can be reset and ran over again by pulsing *i_rst*.

   - **ack_en = 1** (acknowledge word)
   - **ack3 = 1** (acknowledge word)
   - **i_rst = 1** to restart executing instructions (instruction word)

Moreover, an assembler was written in MATLAB for inferring the machine code executed by the controller from higher-level instruction calls. The code for this can be seen below.

```
function inst_out = inst_call(inst_name,param1,param2)

switch upper(inst_name)
%LOAD (opcode = 0)
%DESCRIPTION : This instructions set one of the
%registers in the interface specified by the address
% below
%   ADDRESS :
%       0 : l1_rd_addr (lsb)
%       1 : l1_rd_addr (msb)
%       2 : l1_wr_addr (lsb)
%       3 : l1_wr_addr (msb)
%       4 : l1_addr_inc (lsb)
%       5 : l1_addr_inc (msb)
%       6 : l2_addr
%       7 : pu_wr_addr
%       8 : pu_rd_addr
%       9 : byte_mask (lsb)
%       10 : byte_mask (msb)
%       11 : rd_cnt (set to 0)
```

```
%       12 : wr_sel (0 or 1 for PU addresses &
%                    2 or 3 for interface bufffer)
%       16 - 31 : reserved for loop counters
    case 'LOAD'
        if(nargin ==3)
            opcode = 0;
            address = param1; %register address
            data = param2; %data value

            %error check
            if(address < 0 || address > 31 ||
                 data < 0 || data > 2^16-1)
                error('Invalid parameters for the LOAD
                 instruction');
            end

            inst_out = opcode + address * 2^3 + data * 2^8;
        else
            error('Invalid number of arguments for the LOAD
             instruction');
        end

%ADD (opcode = 1)
%DESCRIPTION : This instructin adds to one of the registers
%in the interface specified by the address below.
%   ADDRESS :
%       0 : l1_rd_addr
%       1 : l1_wr_addr
%       2 : l2_addr
%       3 : pu_wr_addr
%       4 : pu_rd_addr
    case 'ADD'
        if(nargin ==3)
            opcode = 1;
            address = param1; %register address
            data = param2; %data value (signed)

            %error check
            if(address < 0 || address > 4 || data < -(2^15-1) ||
             data > 2^15-1)
                error('Invalid parameters for the ADD
                 instruction');
```

```
            end
            inst_out = opcode + address * 2^3 +
            bin2dec(dec2twos(data,16)) * 2^8;
        else

            error('Invalid number of arguments for the ADD
            instruction');
        end

%LOOP (opcode = 2)
%DESCRIPTION : A conditional jump instruction that sets the program
%counter if the selected counter (specified  with the address) is
%equivalent to 0
%   ADDRESS :
%       0 - 15 : selects between loop counters
%                       (depending on number of counters
%                       inferred with the NUM_LOOPS parameter
    case 'LOOP'
        if(nargin ==3)
            opcode = 2;
            address = param1; %loop select
            inst_addr = param2; %data value (signed)

            %error check
            if(address < 0 || address > 15 || inst_addr < 0 ||
             inst_addr > 2^16-1)
                error('Invalid parameters for the LOOP instruction');
            end
            inst_out = opcode + address * 2^3 + inst_addr * 2^8;
        else
            error('Invalid number of arguments for the LOOP
             instruction');
        end

%READ (opcode = 3)
%DESCRIPTION : Subroutine instruction that either reads from L1
%or reads from an internal buffer, based on the MODE.
%(Note : pu_wr_addr, wr_sel, and read address register should be
% set before calling this instruction).
%   MODE :
%       0 : read from L1
%       1 : read from internal buffer
```

```
    case 'READ'
        if(nargin>=2)
            opcode = 3;
            mode = param1; % read from L1 or from data buffer

            %error check
            if(mode < 0 || mode > 1)
                error('Invalid parameters for the READ
                instruction');
            end
            inst_out = opcode + mode * 2^3;
        else
            error('Invalid number of arguments for the READ
             instruction');
        end

%WRITE (opcode = 4)
%DESCRIPTION : Subroutine instruction that either writes
% data to L1 or L2
%       0 : writes to L1
%       1 : writes to L2
    case 'WRITE'
        if(nargin>=2)
            opcode = 4;
            mode = param1; % write to L1 or to L2

            %error check
            if(mode < 0 || mode > 1)
                error('Invalid parameters for the WRITE
                instruction');
            end
            inst_out = opcode + mode * 2^3;
        else
            error('Invalid number of arguments for the WRITE
             instruction');
        end

%SHIFT (opcode = 5)
%DESCRIPTION : Instruction for circularly shifting data in
% the 64-byte buffer in the interface. Shifts are done in
% steps of bytes, and the number of shifts are specified
% with the num_shift parameter
```

```
    case 'SHIFT'
        if(nargin>=2)
            opcode = 5;
            num_shift = param1; % number of shifts

            %error check
            if(num_shift  < 0 || num_shift > 63)
                error('Invalid parameters for the SHIFT
                instruction');
            end
            inst_out = opcode + num_shift * 2^3;
        else
            error('Invalid number of arguments for the SHIFT
            instruction');
        end

%WAIT (opcode = 6)
%DESCRIPTION : Instruction for stalling the
%interface and processing unit core. There
%are 4 modes of wait for this instruction
%   MODE :
%       0 : PROCESS WAIT
%       1 : RESET WAIT
%       2 : IDLE WAIT
%       3 : READ WAIT
%       4 : M0 WAIT
    case 'WAIT'
        if(nargin>=3)
            opcode = 6;
            mode = param1; % mode for wait
            count = param2; % count for wait
            %error check
            if(mode == 0)
                if(count < 0 || count > 2^20-1)
                    error('Invalid count value for the count
                    wait instruction');
                end
                inst_out = opcode + count * 2^4;
            elseif(mode == 1)
                if(count < 0 || count > 2^16-1)
                    error('Invalid parameters for the WAIT
                     instruction');
```

```
                end
                inst_out = opcode + 2^3 + 2^6
                + count * 2^8;
            else
                mode = mode - 2;
                if(mode  < 0 || mode > 2 || count < 0
                 || count > 2^16-1)
                    error('Invalid parameters for the
                    WAIT instruction');
                end
                inst_out = opcode + mode * 2^4 + 2^3 +
                 count * 2^8;
            end
        else
            error('Invalid number of arguments for the
            WAIT instruction');
        end

%DONE (opcode = 7)
%DESCRIPTION : Instruction that identifies the
%end of instruction calls.
    case 'DONE'
        opcode = 7;
        inst_out = opcode;
    otherwise
        error('Can not recognize instruction name');
end

end
```

5.2.4.6 Applications

Using the instruction set detailed in Table 5.4, the PU controller can be tailored for an array of different read and write tasks between the the local PU core and main memory or this local PU core and other PUs. Not only can this controller handle multi-dimensional read and write patterns, but can do data alignment through shift

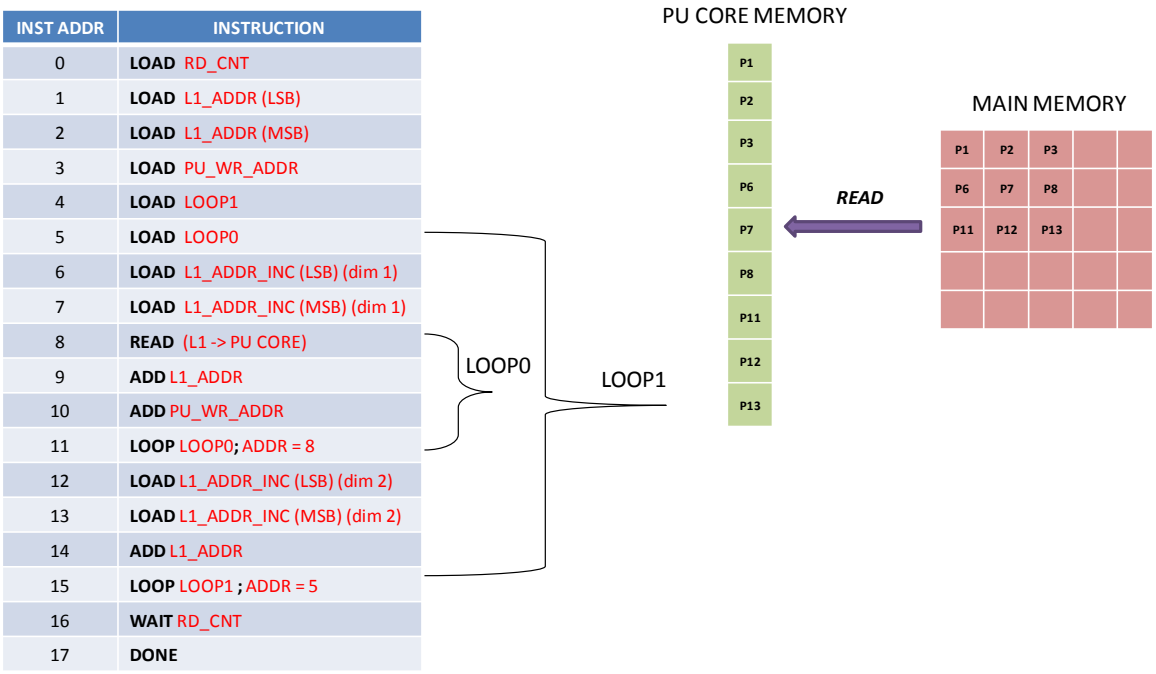


Figure 5.16: 2D Read From Main Memory to the PU Core.

operations on a local buffer, and also a read-modify-write operation that modifies the partial content of a memory word in main memory. In this section, the instruction call flow for executing some important tasks - 2-dimensional read from main memory to PU core, serial read-shift from main memory to PU core, 2-dimensional write from PU core to main memory, a read modify-write from PU core to main memory, and a write from PU core to multiple PU cores through L2 - are detailed.

The instruction call flow for a 2D read from main memory to the PU core memory is illustrated in Figure 5.16. For this instruction call flow, 2D data stored in main memory is read to the PU core memory in 1D format. This is done through 18 instructions, where 2 loop instructions are used to repeat a set of instructions for

INST ADDR	INSTRUCTION
0	**LOAD** L1_ADDR (LSB)
1	**LOAD** L1_ADDR (MSB)
2	**LOAD** PU_WR_ADDR
3	**LOAD** L1_ADDR_INC (LSB) (dim 1)
4	**LOAD** L1_ADDR_INC (MSB) (dim 1)
5	**LOAD** RD_CNT
6	**LOAD** WR_SEL (buffer addr = 0)
7	**READ** (L1 -> BUFFER)
8	**ADD** L1_ADDR
9	**LOAD** WR_SEL (buffer addr = 1)
10	**READ** (L1 -> BUFFER)
11	**WAIT** RD_CNT
12	**LOAD** WR_SEL (PU core)
13	**LOAD** LOOP0
14	**READ** (BUFFER -> PU CORE)
15	**SHIFT** NUM_SHIFT
16	**ADD** PU_WR_ADDR
17	**LOOP** LOOP0; ADDR = 14
18	**DONE**

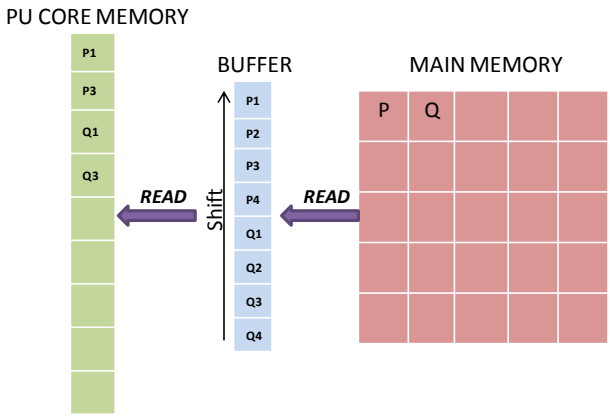


Figure 5.17: 1D Read with Byte Shifts From Main Memory to the PU Core.

making the necessary read requests to memory.

Another example of reading from main memory can be seen in Figure 5.17. In this example instruction call flow, data is read from the main memory to the local register file buffer in the PU controller. This data is then byte shifted and then stored on the PU core. This is done with 19 instructions, where a set of 4 instructions are looped to accomplish this task with the PU controller.

For writing back to main memory, Figure 5.18 shows an example of a read-modify-write operation from a PU core to the main memory. In this instruction call flow, first the data word to perform the partial write to is read. Then, by masking the read in data word with the new data from the PU core with a data mask, the new

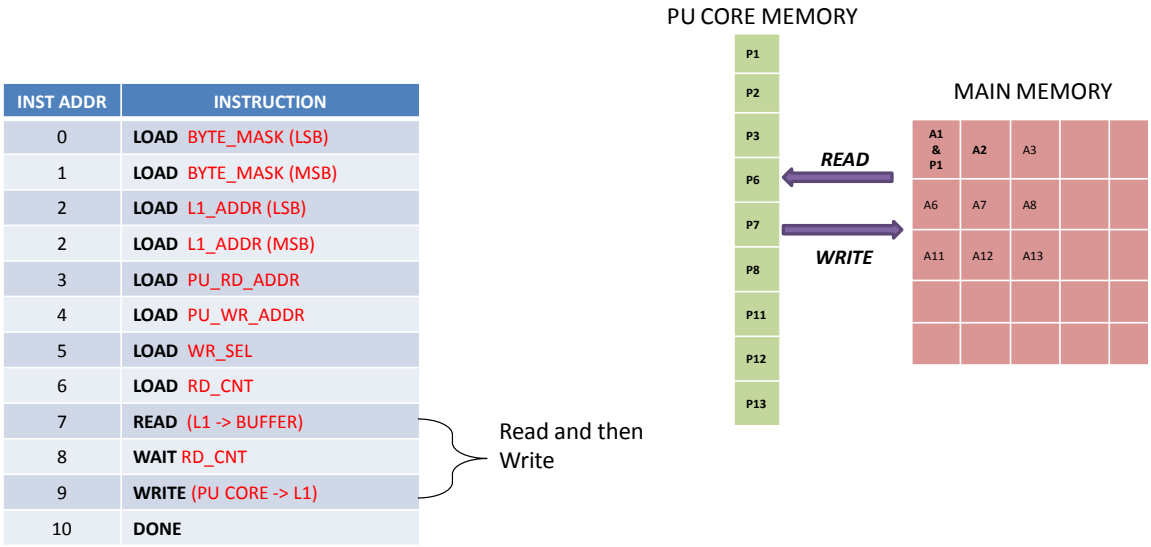


Figure 5.18: Read Modify Write From the PU Core to the Main Memory.

modified data word is then written back to memory. This task is accomplished with 11 instructions, where 3 instructions are looped in order to perform the read-modify-write. This instruction can be adapted to a full write, by simplifying the instruction call to just write the data word directly from the PU core to the specified main memory address.

The final example shows how a PU can write to other PUs using the PU controller. Figure 5.19 shows the instruction call flow for this task tailored to writing to 3 different PUs. In this instruction call flow, *PU CORE(0)* writes data to *PU CORE(1)*, *PU CORE(2)*, and *PU CORE(3)*. This is accomplished with 14 instructions that invoke 2 loops for addresses the PU addresses and the local register addresses in the PUs.

INST ADDR	INSTRUCTION
0	**LOAD** BYTE_MASK (LSB)
1	**LOAD** BYTE_MASK (MSB)
2	**LOAD** PU_RD_ADDR
3	**LOAD** L2_ADDR
4	**LOAD** PU_WR_ADDR
5	**LOAD** L2_ADDR
6	**LOAD** LOOP1
5	**LOAD** LOOP0
6	**WRITE** (PU CORE -> L2)
7	**ADD** PU_RD_ADDR
8	**ADD** PU_WR_ADDR
9	**LOOP** LOOP0; ADDR = 6
10	**ADD** PU_WR_ADDR
11	**ADD** L2_ADDR
12	**LOOP** LOOP1 ; ADDR = 5
13	**DONE**

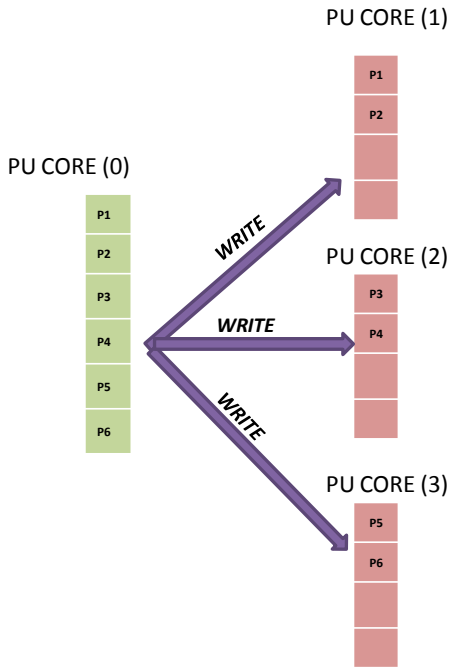


Figure 5.19: 1D Write From the PU Core to another PU Core.

Multiple PUs can be used together coherently for doing a large-scale data processing task. To show this on a smaller scale, 3 VVM processing units (detailed in Section 5.2.1), which all have a PU controller, are combined together for a doing a multi-stage processing task. Figure 5.20 shows this processing flow and the outputs from the simulated PU VVMs. In this processing flow, horizontal and vertical edges are extracted from two separate PU VVM that are executing instructions from their respective PU controllers. The resulting images are written to a 3rd PU VVM which combines the edges into one coherent image. Not only are each of the PU VVM independently executing instructions, but they are also synchronized after processing phase in order to prevent data collision.

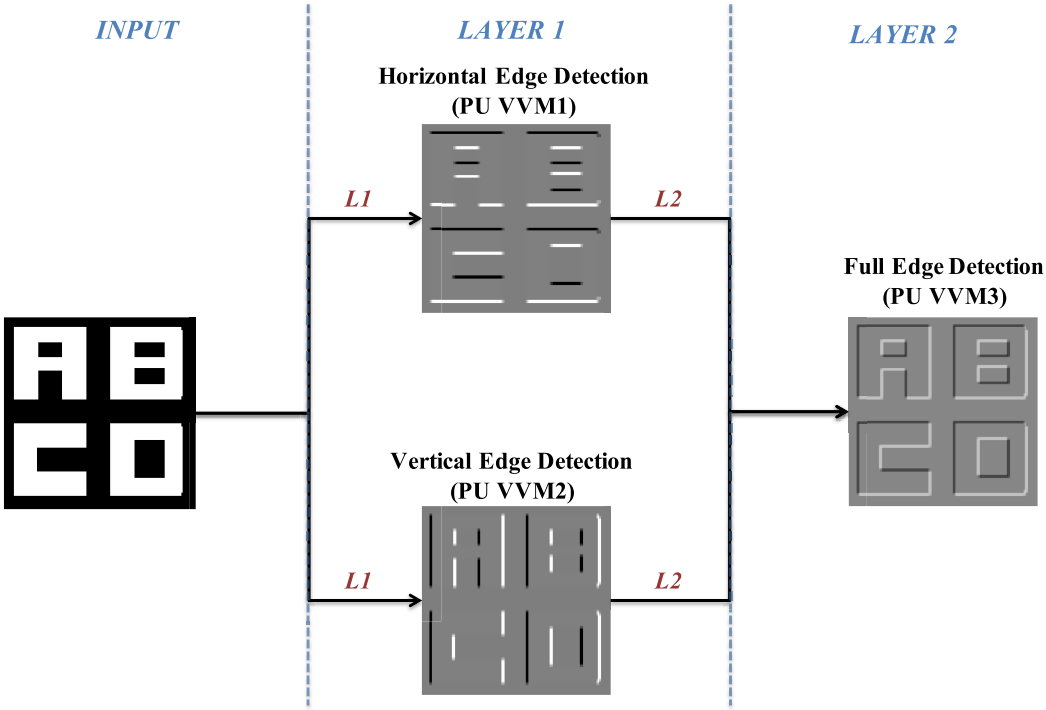


Figure 5.20: Processing Flow for Image Edge Detection with 3 PU VVMs. Horizontal and vertical edges are extracted from two separate PU VVM that are executing instructions from their respective PU controllers. The resulting images are written to a 3rd PU VVM which combines the edges into one coherent image.

Furthermore, this can be applied to large-scale tasks, such as image classification with multi-layered neural networks. It can also be applied to more heterogeneous tasks, where different types of PUs process data coherently and cohesively.

5.3 Network-on-Chip

As CMPs scale to bigger SoCs, node interconnections have become a critical factor in energy consumption, performance, and area. With the future of SoCs hinged on modular heterogeneity for promoting both generality and performance in ap-

plications, energy efficient and scalable solutions for NoCs have become especially paramount.[83,84]

Recent work in network architectures has led to high-speeds asynchronous networks,[85] low voltage hybrid packet/circuit switching networks,[86] bufferless routing networks,[87–89] and more. In the 2.5D Nano-Abacus SoC, a circuit switching network is designed to provide fixed high speed communication paths of the PUs to main memory, and a packet switching bufferless mesh network is designed that provides a robust and efficient communication between PUs. The implemented packet-switching network, not only provides a flexible communication solution that efficiently routes based on data traffic, but is also resilient to network deadlocks and livelocks, and can handle faulty routing channels due to increased fabrication defect with larger networks. In this dissertation work specifically, a design approach is formalized for constructing scalable bufferless networks similar to the one implemented in the 2.5D Nano-Abacus chip. Additionally, a network simulator is designed for prototyping these network models with different constraints.

5.3.1 Scalable Bufferless Network

The proposed network is based on a 2D packet-switching mesh topology, where an array of nodes comprised of a router and an internal unit, which could be a processor, memory block, or ASIC, is arranged in a 2D grid of rows and columns. Each node see potential data traffic from the internal unit and traffic from a north, south, east, and

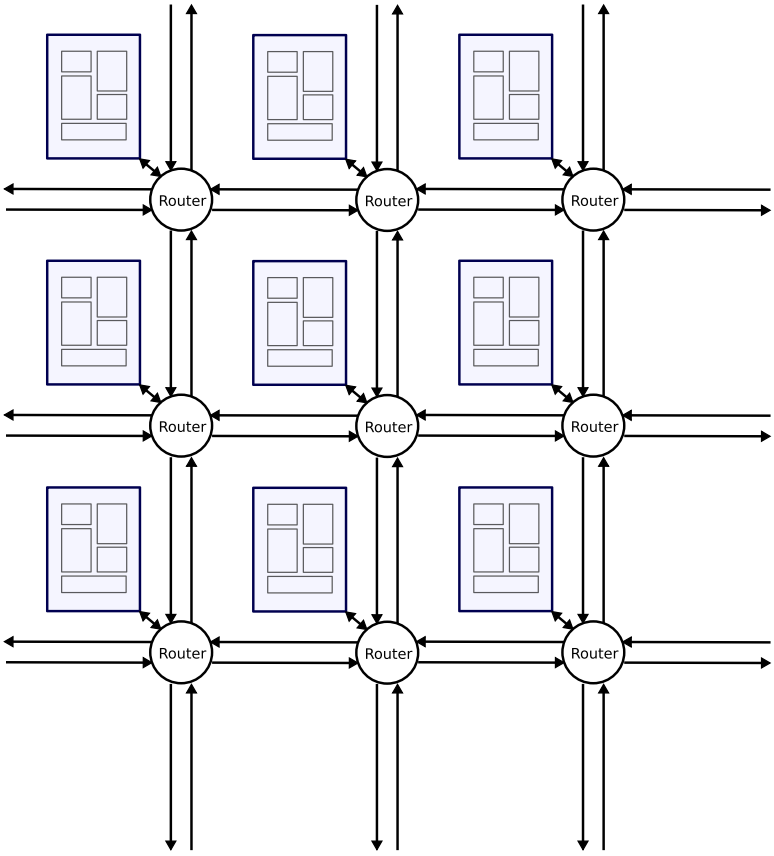


Figure 5.21: A 3 by 3 Mesh Network Comprised of Homogeneous Nodes.

west port. Using these neighboring connections, packets are delivered between nodes and also network input/output ports. The router controls the data traffic control through an adaptive routing table that incorporates deflective routing to avoid the use of buffers (explained in the next section). Consequently, the network is globally synchronous as packets have to routed across each node in a consistent manner, and thus careful consideration is taken for the clock network distribution. An example 3 by 3 mesh network of homogeneous nodes can be seen in Figure 5.21.

Hinged primarily on scalability, this network architecture provides a generic framework for mesh networks of any size, as the complexity of the architecture doesn't scale

with the number of nodes. Additionally, because of the modular and tillable design of this architecture, this network is adaptable for heterogeneous and homogeneous processing and memory cores. Furthermore, as SoC scale to larger sizes, fabrication defects become a more critical concern; a broken link in the network can result in dropped packets and unpredictable system behavior. Thus, a broken link detection scheme, which works in conjunction with the adaptive routing table, is implemented in the router to achieve fabrication defect tolerance for larger SoC.

5.3.1.1 Router

Each router employs a low complexity bufferless routing scheme based on a oldest-first (OF) and most-deflected prioritization that guarantees a deadlock-free and livelock-free network.[87] Using the OF prioritization, the oldest packet is routed to the optimal path, while all other packets are arbitrarily routed. Moreover, since packets can be injected from any node into the network, it is possible for multiple packets with the same time priority in the network to arrive at a node's router. In order to prioritizes packets for this case, a ranking based on number of deflections is used. As the oldest most-deflected packet in the network will be successfully routed, each packet that stays in network long enough will have the opportunity to become the highest priority packet, and thus be routed correctly. This guarantees a livelock-free network, which ensures that a packet will not be routed in a loop indefinitely. Furthermore, a deadlock-free network is also guaranteed because of a buffer-free design; every packet

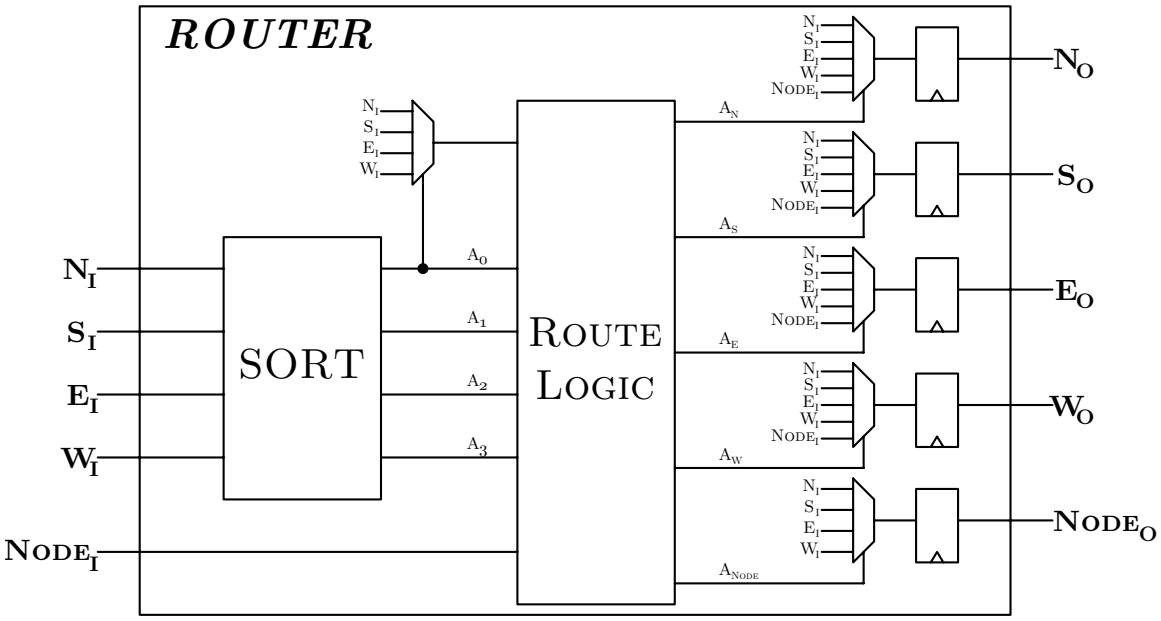


Figure 5.22: Architecture of the Bufferless Router in the Network.

will always route to another link each clock period. This is only possible because the number of outputs links at each router doesn't exceed the number of input links.

A block diagram of the router can be seen in Figure 5.22. As described above, each router potentially receives packets from the north, south, east, west, and the internal unit ports. For each clock period, first the incoming packets are sorted based on the priority scheme detailed earlier. Then, the highest priority packet is allocated to the optimal output link based on a simple routing table implemented in the routing logic block. Since it is possible to have defective links especially with very large NoC implementations, some additional logic is included in the route logic that adaptively tunes the routing table to compensate for this issue. Moreover, after the highest priority packet is allocated, the remaining packets are arbitrarily routed

to minimize routing logic and hence reduce area and energy cost of the router. The routing logic block supplies addresses that are used for selecting the correct packet from the incoming input links to the output links. Furthermore, the internal unit can only inject a packet into the network if there is a free output link.

For this network, a simple bound for the priority counter implementing the oldest-first, and then most deflected priority scheme can be determined. Assuming the network has **M** rows, **N** columns, **K** number of links per router, and no clock delay for an internal unit at a node to accept a packet, then a bound can be deduced as follow. Firstly, for the set of the oldest packets $S0$ in the network, the highest priority packet (oldest and most deflected) of the network P_0 exists in this set, and will have at most a maximum time delay of

$$T_{P_0} = ((M-1) + (N-1)) \tag{5.7}$$

clock cycles, which is the delay for the longest path in the network. The second highest priority packet in this set P_1 can be deflected at most once (when competing with the first highest priority packet), which could mean an additional latency of 2 clock cycles. Furthermore, the third highest priority packet in this set P_2 can at most be deflected twice which could mean an additional latency of 4 clock cycles, and so

on. That is,

$$\begin{aligned} T_{P_1} &= (M + N - 2) + 2 \\ T_{P_2} &= (M + N - 2) + 4 \\ T_{P_{n-1}} &= (M + N - 2) + 2 \times (n - 1) \end{aligned} \tag{5.8}$$

for n packets in the set of oldest packets in the network. Since a node can only generate 1 packet per clock, at most only $(M \times N)$ packets can be in this set of oldest packets. Thus, the maximum possible time it takes to deliver all these packets in the set T_{S0} can be derived as

$$T_{S0} = (M + N - 2) + 2 \times (M \times N - 1) \tag{5.9}$$

clock cycles. Expanding to the set of all packets in the network ranked first by time, and then by most deflected, the least priority packet in the network could at most be deflected $2 \times (M \times N \times K - 1)$ times and have a longest path of $(M + N - 2)$ clock cycles. Thus a simple time bound T_M on the time priority counter for all packets can be deduced as

$$T_M = (M + N - 2) + 2 \times (M \times N \times K - 1) \tag{5.10}$$

For a 4 by 4 network with 4 ports at each nodes and no defective routing links (**M = 4, N = 4, K = 4**), the maximum time counter would be **132**, which would require each packet to have a minimum of **8** bits for the time counter. As this analyzes the

worst case scenario, most packet time delay should be much less than this bound with normal traffic patterns. Moreover, it is possible to deduce a lower bound for the time priority counter if a more rigorous and realistic data traffic analysis is done with this network.

5.3.1.2 Network Protocol

A 4 phase programming protocol is implemented for booting up, assessing, and running the NoC. This protocol allows the NoC to clear all packets in the network, diagnose the status of each links, check which nodes are reachable, and enable nodes to send and receive packets.

- **RESET:** During the RESET phase, which is controlled by a global 1-bit signal (reset), all node links are cleared, and the status of all links are reset to unhealthy. Reset phase ends once the reset signal is deasserted.

- **DIAGNOSE:** After the RESET phase is the DIAGNOSE phase, which is also controlled by a global 1-bit signal (diagnose). In the phase, each node transmits diagnose packets through each of its outputs links to surrounding nodes. When a node receives a diagnose packet through one if its links, it compares it with a programmable diagnose code, which is shared among all the node. If the code stored on the node matches the transmitted code, the link is labeled healthy, and is included as a valid link for the routing table. However, if the code does

not match, then it is recorded as unhealthy/broken, and the link will not be used. The DIAGNOSE phase last for as long as the diagnose signal is asserted.

- **PING:** Next, the PING phase is implemented for assessing which nodes are reachable after diagnosing all links in the network. One of the input ports for the networks transmits a ping packet, which is a configuration packet that elicits a response (acknowledgment packet) from the router, to each node. A destination address identifier is included in each ping packet so the destination node knows which address to send the acknowledgement packet. Additionally, a source address identifier is included in the acknowledgement packet so the configuration unit for this ping protocol can identify the node. Furthermore, a list of reachable nodes can then be compiled from the list of received acknowledgment packet. The PING phase is coordinated by the external configuration unit, which can adapted for different ping process routines.

- **ENABLE:** Finally, after the nodes have been pinged, and the list of reachable nodes have been determined, then the ENABLE phase can be commenced. Similar to the PING phase, an external configuration unit coordinates this phase. Based on this list of reachable nodes, the external unit sends enable packets, which can be used to enable internal unit processor, ASIC, and so on, and also enables the router to send/receive packets from that internal unit, to the nodes. Nodes that are not enabled in the network, will discard packets that

are received through the router. After the ENABLE phase has concluded, the network is ready to run, and can accept data traffic from any valid source.

5.3.2 Network Simulator

In MATLAB, a network simulator with a graphic user interface (GUI) was designed for prototyping the proposed network architecture with different constraints. The network simulator supports generic mesh networks of different sizes, can simulate the network programming protocol discussed in Section 5.3.1.2, and can run the network with different data traffic while measuring network statistics. Additionally, this simulator also incorporates defective/broken links in order to test the robustness of the network.

To illustrate the functionality of this network simulator, a 4 by 4 mesh network with defective links is emulated. Using the GUI, this network was constructed, and can be seen in Figure 5.23. In this example network, only the west port is functional, and there is also a defective internal link in the network. Furthermore, the nodes are outlined in red because they have not pinged yet, and thus are presumed invalid.

After the RESET phase, the network links are diagnosed in the DIAGNOSE phase, and this can be seen in Figure 5.24. As diagnose packets are transmitted through the links, defective links are removed, as seen in the diagram.

Furthermore, after that all nodes are pinged during the PING phase; a snapshot of this process can be seen in Figure 5.25. During this process, ping packets, which

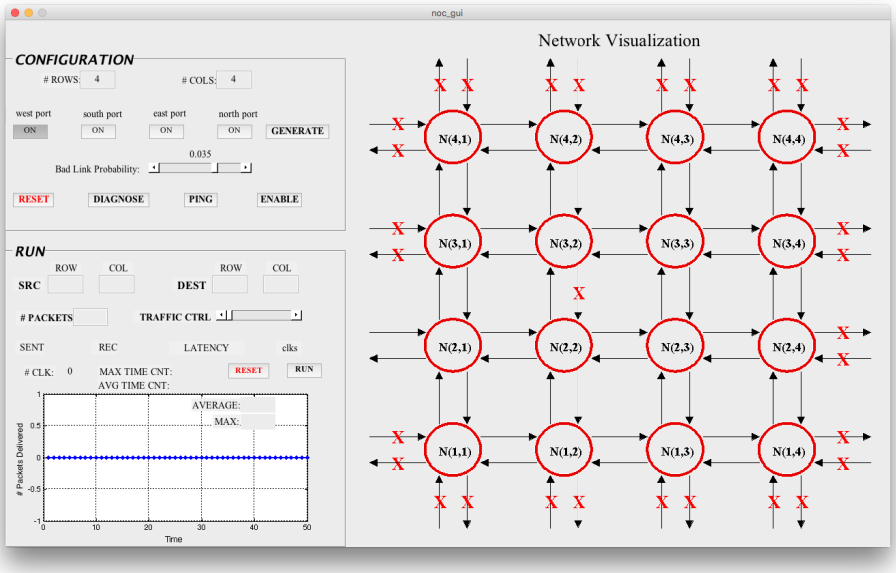


Figure 5.23: Visualization of a 4 by 4 Mesh Network During the RESET Phase. An "X" on the link signifies a defective link. Nodes are outlined red because they have not been pinged.

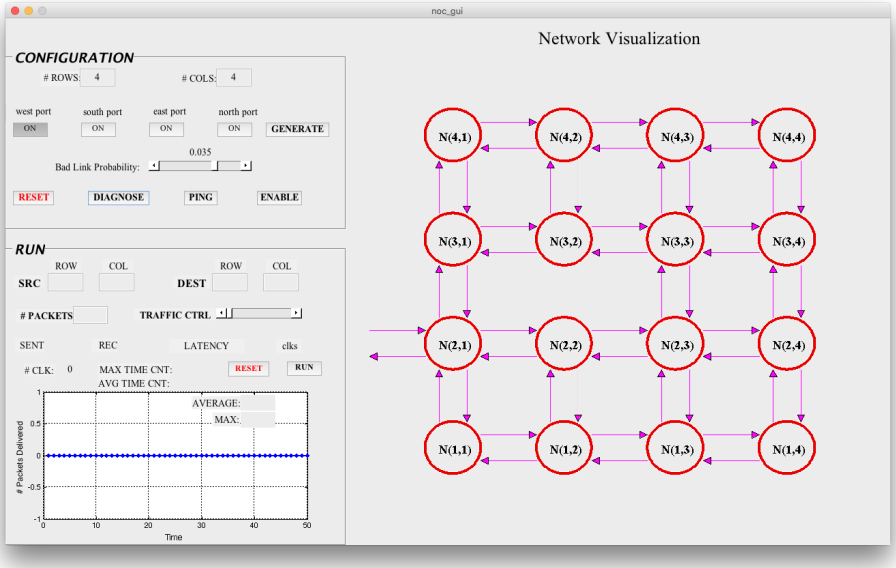


Figure 5.24: Visualization of a 4 by 4 Mesh Network During the DIAGNOSE Phase. A purple link signifies a diagnose packet transmission. Links that have been removed from the network diagram have been identified as defective.

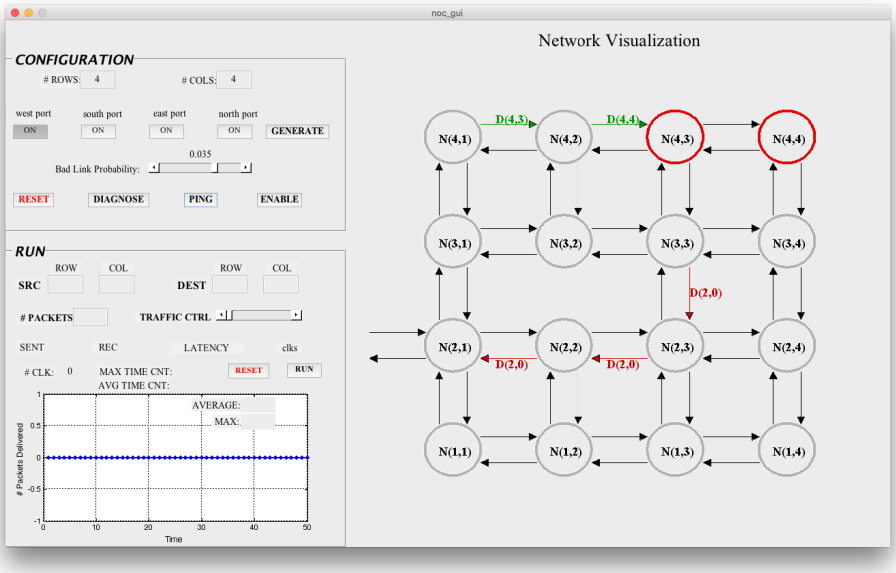


Figure 5.25: Visualization of a 4 by 4 Mesh Network During the PING Phase.A green link signifies a ping packet from the configuration unit, and a red link signifies an acknowledge packet from one of the network nodes. Nodes change from red to gray if it receives a ping packet.

are displayed in green, are transmitted from the configuration unit to the different network nodes, and the receiving nodes sends an acknowledgement packet, visualized as a red link, back to the configuration unit. A network node becomes valid when it receives a ping packet, and the outline of the node changes from red to gray.

The final stage of the network protocol, the ENABLE phase, enables all reachable nodes in the network. Figure 5.26 shows a visualization of the network during this phase. In the phase, the configuration unit sends enable packets, signified as a green link in the diagram, to the reachable nodes in order to enable them. After a node is enabled it can transmit packets generated locally. A node receiving an enable packet first changes green when the packet is received, and then to black, which signifies that

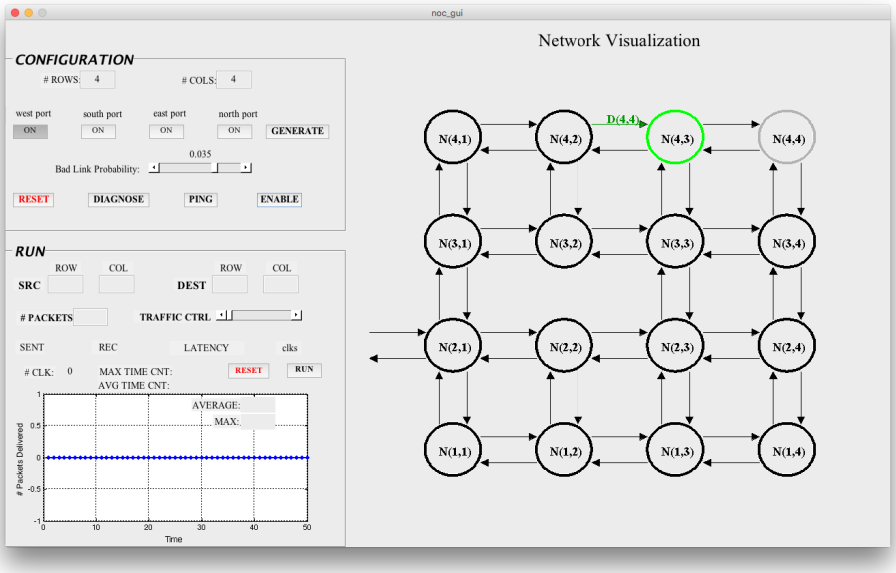


Figure 5.26: Visualization of a 4 by 4 Mesh Network During the ENABLE Phase. A green link signifies an enable packet transmission. A node changes from grey to green when it receives the enable packet, and then to black signify it is enabled.

the node is now enabled.

After the network is programmed through the 4 phase protocol, different data traffic is injected into the network to analyze performance. The traffic is segmented into 2 parts – general traffic, which is shown as yellow links, and data of interest, which is shown as blue links. For the data of interest, the source, destination, and number of packets can be set, and the average latency of packet delivery is measured along with the number sent by the source and received by the receiver. In addition, the average and the maximum number of packets delivered per clock in this network for a given time window is also measured and shown in the GUI. Figure 5.27 and 5.28 show the network running in two different traffic conditions. In both traffic scenarios packets are never dropped, and the network never stalls with a deadlock or livelock.

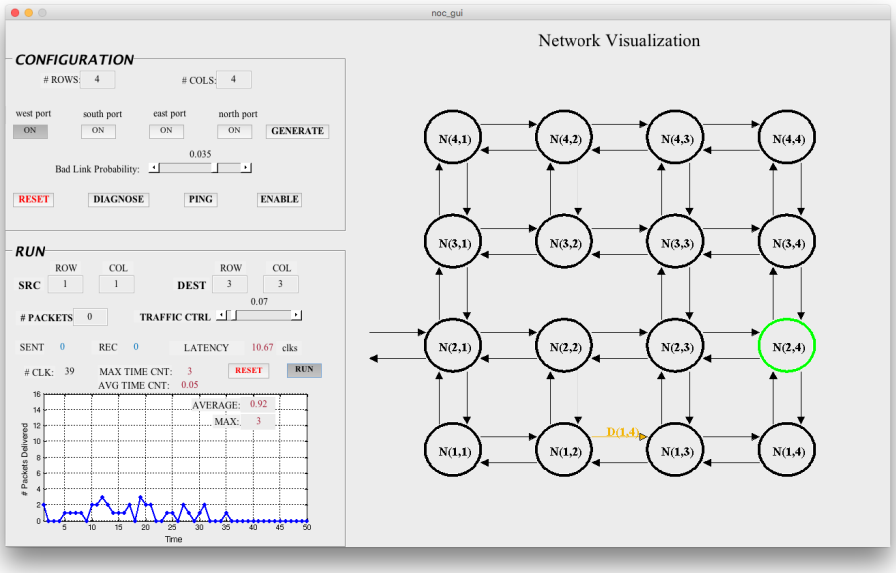


Figure 5.27: Visualization of a 4 by 4 Mesh Network Running with Light Data Traffic. A yellow link signifies a general data packet, while a blue link signifies a packet of interest, where the source and destination are specified on the side panel.

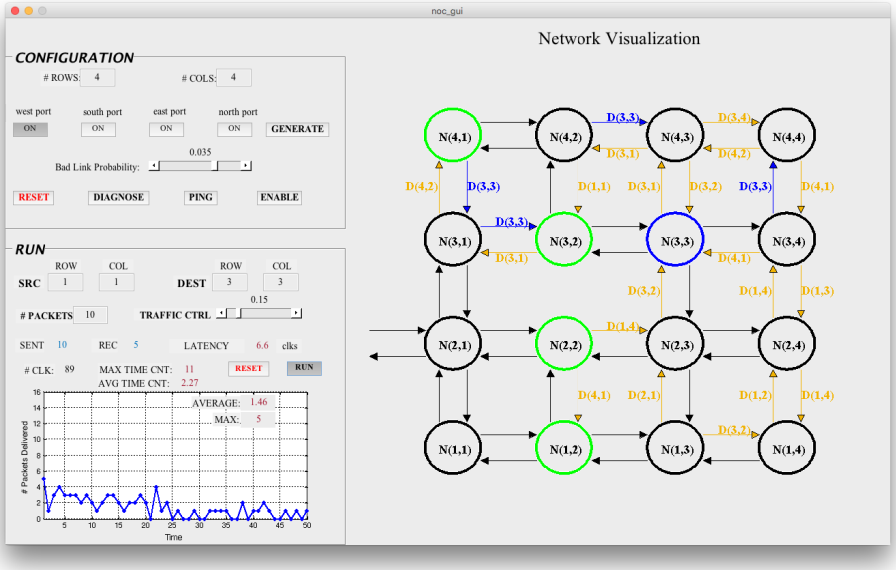


Figure 5.28: Visualization of a 4 by 4 Mesh Network Running with Heavy Data Traffic. A yellow link signifies a general data packet, while a blue link signifies a packet of interest, where the source and destination are specified on the side panel.

Additionally, the maximum time for a packet delivery doesn't come to the computed bound, even in heavy traffic case.

As shown in the detailed example. not only is this network simulator useful for testing the network model, but is also useful for measuring network statistic under different constraints. In this simulator, the average packet delivery latency, maximum time counter, and the network throughput can be measured.

5.4 PAD I/O Circuitry

For the CMPs in the 2.5D Nano-Abacus SoC, a set of custom I/O cells with connecting pads were designed in the 55nm process for power and analog connections, the GPIO interface, and the 3D DiRAM interface. For the GPIO interface, bi-directional I/O pads were designed to run at up to 1GHz on a power supply ranging from 0.7V to 1.2V. For the 3D DiRAM interface, separate digital input and output pads were designed to run at 1GHz on a power supply as low as 0.5V. In addition to this, power and analog pads were made for the various power supplies and analog signals on the chip. Each pad is designed with electrostatic discharge (ESD) protection diodes to prevent high voltages on the transistors. These pads are assembled as shown in Figure 5.29, for each CMP. The MHUB pads are used for the 3D DiRAM interface, and the FPGA I/O pads are used in the GPIO interface. Moreover, in addition to the C4 bumps that are used in the flip chip package for interfacing the CMP to the

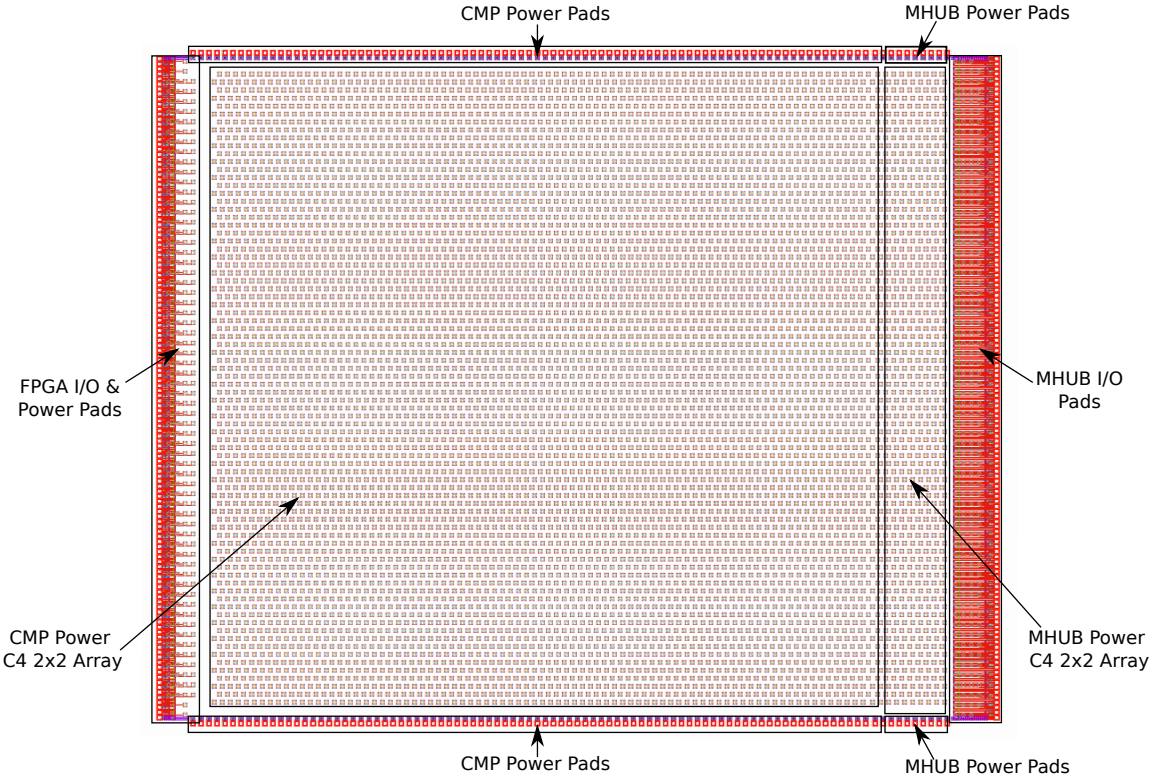


Figure 5.29: Annotated Layout of the CMP Pad Frame.

interposer board, the power, analog, and digital I/O signals are also connected to inline wire bond (WB) pads for the additional option of testing the CMPs on a PGA package. The WB pads are arranged on the perimeter of the CMP chip, while the C4 pads are placed on top of the internal circuitry. In order to minimize resistance to the power supplies and ground, the power pads are replicated numerously across the whole chip. The unique pad cells in this pad frame are listed as:

- MHUB I/O Pad
- FPGA I/O Pad

- Power/Analog Pad
- C4 2x2 Pad Array

With the exception of the C4 2x2 pad array, the pad cells along with post-layout simulations are discussed in the following subsection.

5.4.1 MHUB I/O Pads

The MHUB input and output pads are designed jointly in the same cell, and the circuitry for these pads can be seen in Figure 5.30. The output from the MHUB output pad, D_o, is level-shifted to the correct voltage supply domain (range of 0.7-1.2V), and then buffered out to drive the capacitive load of the WB and C4 pad and external wiring (estimated at 20pF). The input to the CMP from the MHUB input pad, D_i, is buffered and level-shifted to the core voltage supply. Both input and output pads are wired to a set of diodes to minimize current from ESD. The layout for these pads can be seen in Figure 5.31. In this layout, the WB pads are placed on the right side, and the C4 bumps for the input, output, power, and ground signals are placed directly about the I/O circuitry. Majority of the layout is dedicated to the output buffer, which is composed of thousands of transistors in parallel for driving the capacitive load with the given power supply and timing constraints. The pair of theses input and output pads measure 210μm by 954μm in area.

Post-layout simulations were done to measure the performance of the pads for

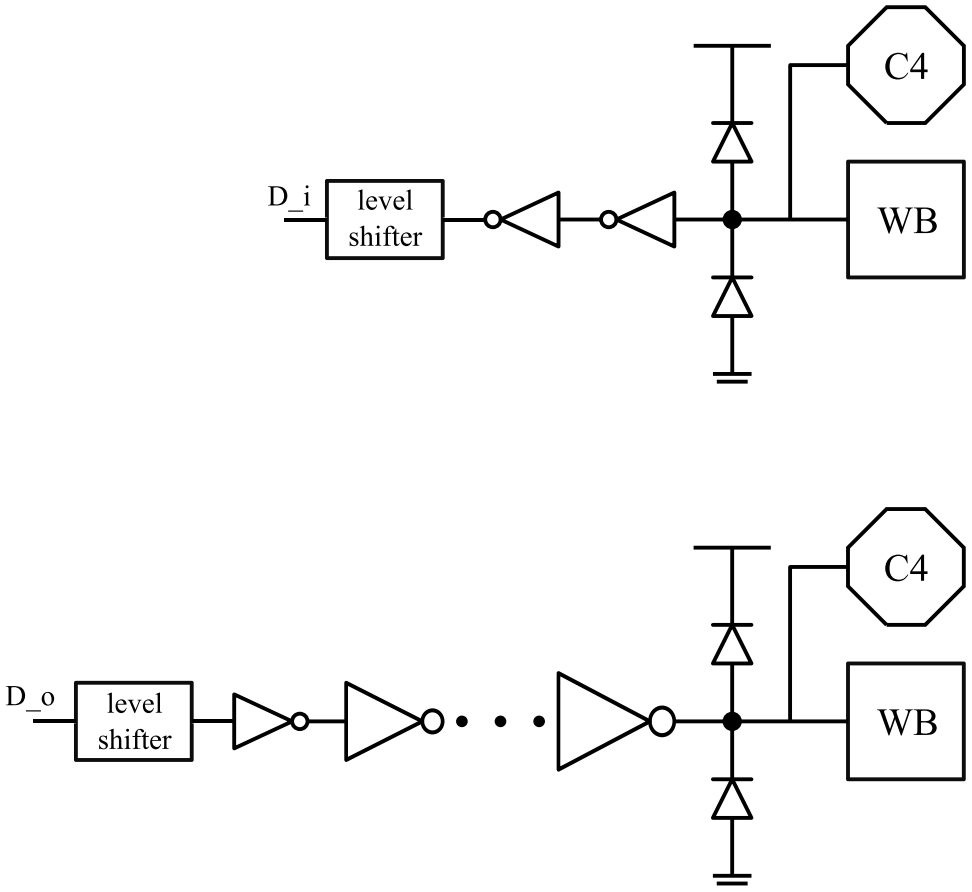


Figure 5.30: MHUB Input and Output Pad Circuitry. Input pad is displayed on the top, while the output pad is displayed on the bottom.

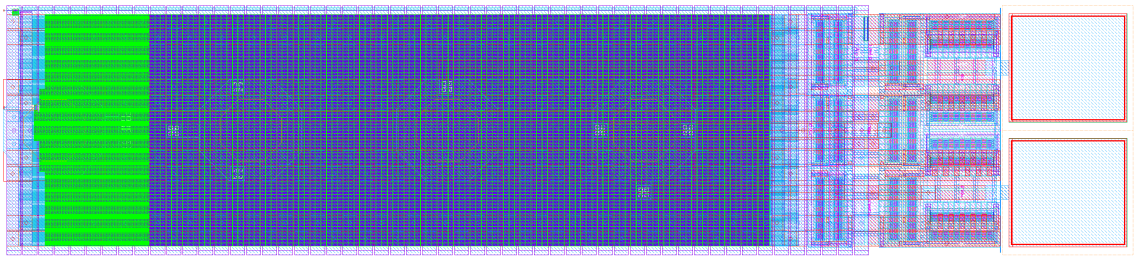


Figure 5.31: Layout of the MHUB Input and Output Pads.

different supply voltages. The results of these simulations can be seen in Figure 5.32.

The chip output, shown in red, is the output from the core which drives the input

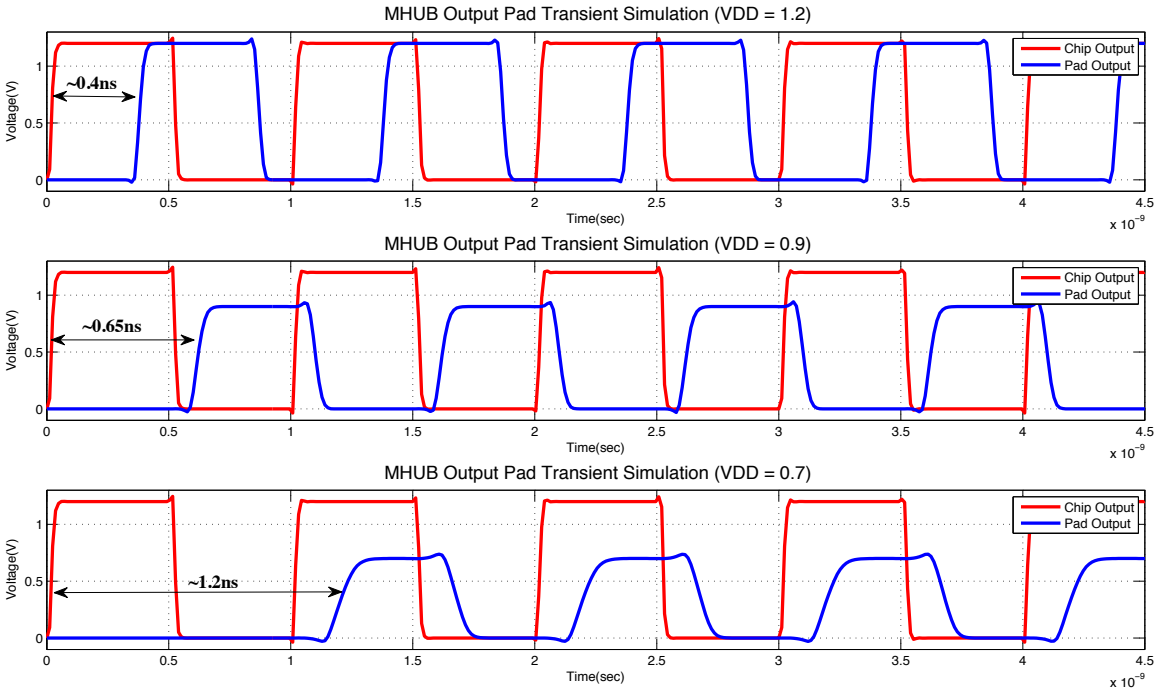


Figure 5.32: Transient Simulation Results of the MHUB Output Pad. The output is plotted for a 1GHz input signal and an external supply voltage of 1.2V (top), 0.9 (middle), and 0.7 (bottom).

of the output buffer in the pad, and the pad output, shown in blue, is the output from the pad. The transient simulations results show that the output pads can run at 1GHz with an external supply of 1.2V down to 0.7V. The latency for these different supply voltages varies from 0.4ns to 1.2ns.

Additionally, the input pad was simulated to ensure that it can run with the same specification. Since the input pad doesn't drive a huge capacitive load like the output pad, the large buffer was not necessary. Furthermore, to determine the maximum frequency the pads can run from the input to output in a closed system, a simulation is carried out with an oscillator configuration of the I/O pads. With the input and

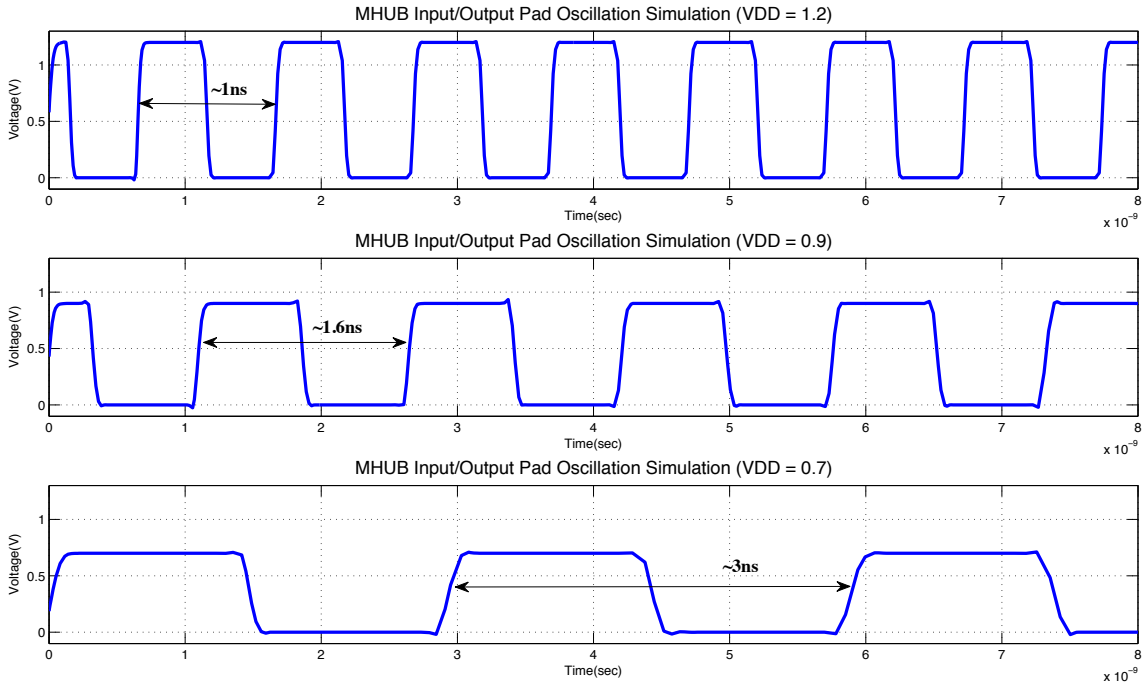


Figure 5.33: Transient Simulation of a Oscillator Configuration of the MHUB Input and Output Pads. The output from the connected pads is plotted for an external supply voltage of 1.2V, 0.9V, and 0.7V.

output pad connected at the WB and C4 pad, the inverted input signal from the input pad is connected to the signal driving the output pad. The period is measured for different external voltage supplies, and this result can be seen in Figure 5.33. In this setup, the latency due to the propagation delay heavily limits the maximum speed measured at the pads. The maximum frequency for the different supply voltages was deduced as: 1GHz at 1.2V, 625MHz at 0.9V, and 333MHz at 0.7V. Nonetheless, when the pads are used independently, both pads can run up to 1GHz even with a supply voltage as low as 0.7V.

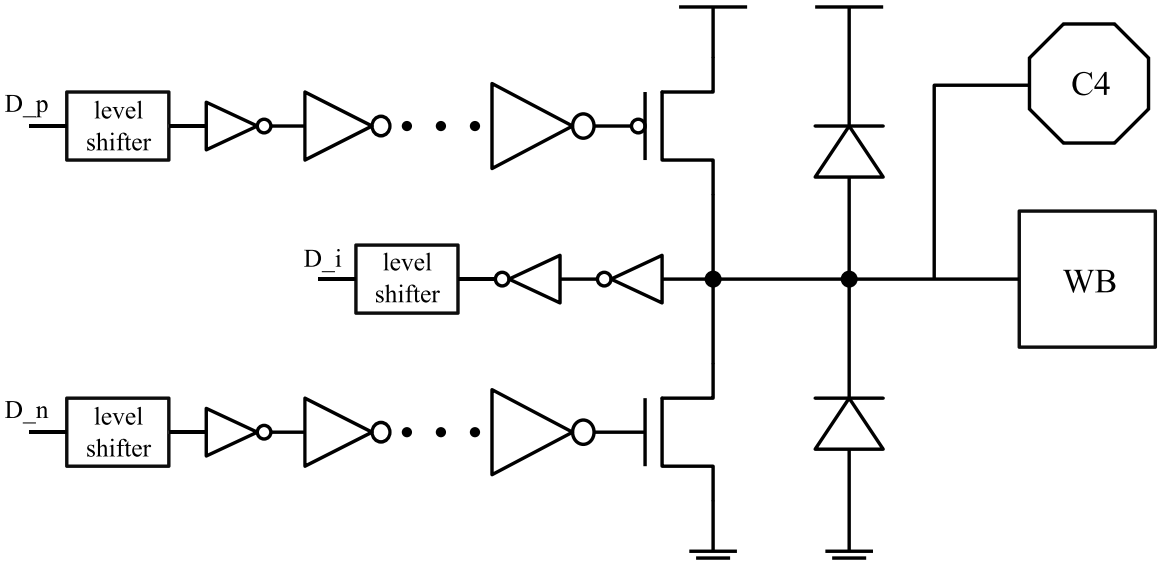


Figure 5.34: FPGA Bi-directional I/O Pad Circuitry.

5.4.2 FPGA I/O Pads

Bi-directional pads were designed for communication between the CMP chip and the Zynq FPGA. The input and output signals were combined together to share WB and C4 pads and ESD diodes. The architecture for this pad can be seen in Figure 5.34. In this pad design, there are two outputs signals from the CMP, D_p and D_n. These signals are internal signals from the CMP that drive the output to the FPGA, and also control the mode of pad. D_i is the input signal to the CMP, and is either driven by D_p and D_n or the WB or C4 pad. Table 5.12 shows the truth table for this circuit. When D_p and D_n are equivalent, the pad is configured as an output pad, and the WB and C4 pad will be driven by the inverted D_p/D_n signal. When D_p is set high and D_n is set low, the pad will be in high impedance, and can be driven by the external WB or C4 as an input pad. Furthermore, this pad can also be used as an

Table 5.12: Truth Table for the FPGA I/O Pad Circuit.

D_p	D_n	D_i
0	0	1
1	0	U
0	1	X
1	1	0

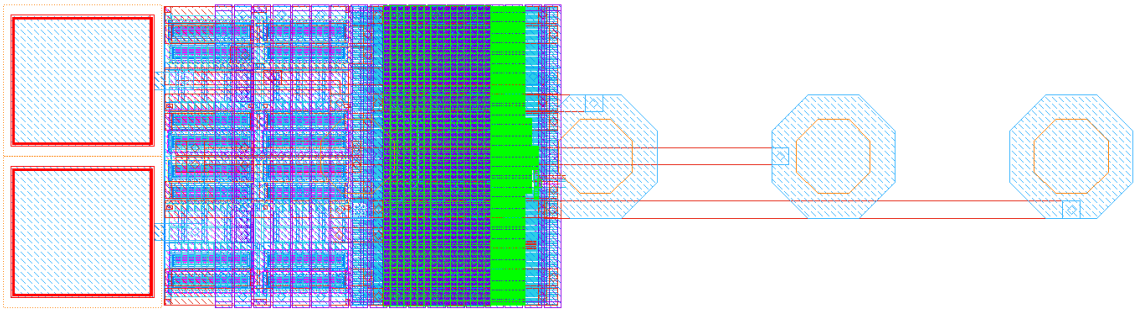


Figure 5.35: Layout of the FPGA I/O Pad.

output pad with just the pull-up network with the inclusion of an external pull down resistor. In this case D_n is fixed low to leave the last stage NFETs off. Conversely, the pad can also be configured as an output pad with just the pull-down network with an external pull-up resistor. In this case D_p is fixed high. If the D_p signal is set low and the D_n signal is set high both transistors for the last stage of the buffer will be on causing an undesirable short circuit.

The layout of this pad can be seen in Figure 5.35. In addition to the C4 pad used for the I/O signal, there are C4 pads placed for the power and ground signals. There is also a WB pad for the I/O signal, and a WB pad for one of the power or ground signal. This pad was designed to match the vertical pitch of the MHUB pads, and the full pad measures 210μm by 392μm in area. Because circuitry can be placed under

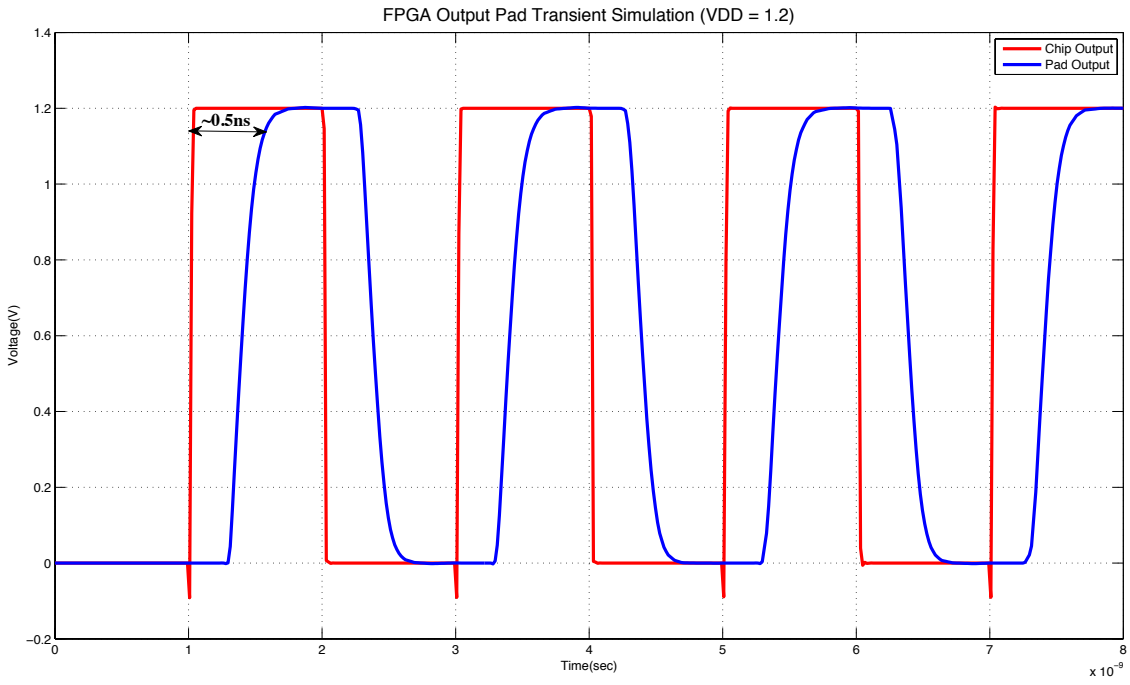


Figure 5.36: Transient Simulation Result of the FPGA I/O Pad Configured as Output. The output is plotted for a 1GHz input signal and an external supply voltage of 1.2V.

the C4 pads, this pad cell can overlap other cells in the chip.

Similar to the MHUB pads, the FPGA I/O pad was simulated to measure performance with different supply voltages. Specifically, this pad was designed to run with a supply voltage between 0.3 to 1.2V. With a 1.2V power supply, the FPGA I/O pad, when configured as an output, can run at 1GHz with a propagation delay of 0.4ns; the simulation results can be seen in Figure 5.36.

Configuring the pad as an oscillator, where the inverted input drives the output in a loop, the maximum frequency of this pad is measured for different voltage supplies. The plot in Figure 5.37 shows the measured frequency for supply voltages across the

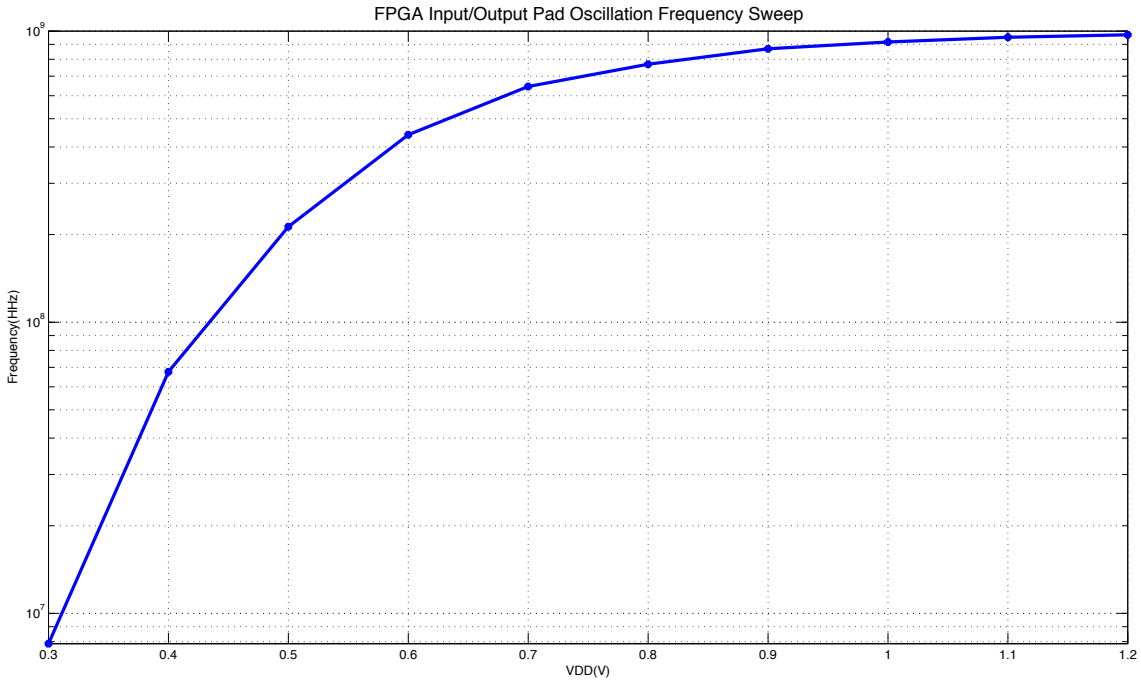


Figure 5.37: Frequency Plot of the FPGA I/O Pad Configured as an Oscillator For Different Supply Voltages

aforementioned range. In this setup, the pads can run at 1GHz with a 1.2V supply voltage, while at 0.3V the pad can run at 8MHz.

5.4.3 Power/Analog Pads

This pad was designed to provide external connections for the power and analog signals for both the CMP and MHUB driver circuitry through the WB and C4 terminals. The layout of this pad can be seen in Figure 5.38. Aside from the bond pads, this design include protection diodes for ESD, and the power and ground wires that make up the pad frame supply rings. Furthermore, these pads are placed on the top

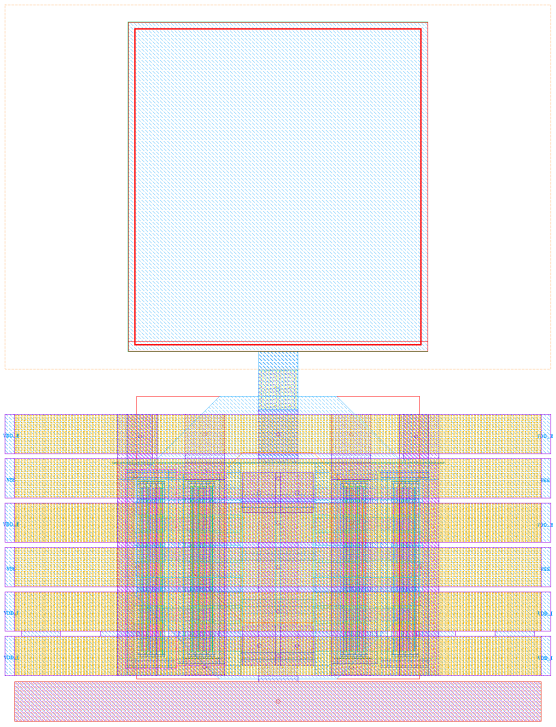


Figure 5.38: Layout of the Power/Analog Pad.

and bottom edges of the pad frame, and measure 217.8 μm by 166μm in area.

5.5 The Yupana CMP: A Heterogeneous Mixed-Signal Accelerator

5.5.1 Overview

The Yupana chip is a heterogeneous chip multiprocessor comprised of mixed-signal processing units for large-scale energy efficient data processing. Fabricated in a 55nm CMOS process, this chip consist of an array of heterogeneous processing units, a high

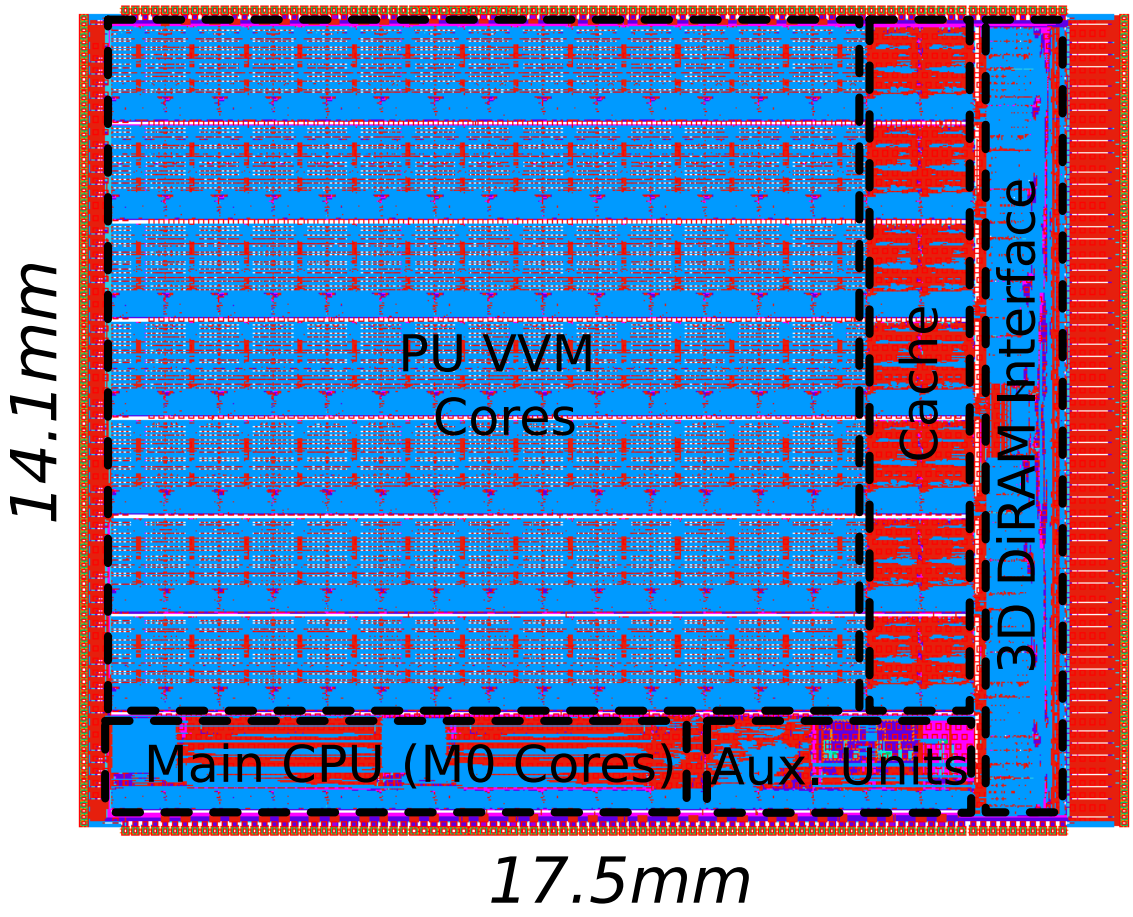


Figure 5.39: Layout of the Yupana CMP.

bandwidth memory interface to a 3D DiRAM with a token-ring network on chip, a switch-packet mesh network, and a general purpose input/output port interface to a Zynq FPGA in 14mm by 17mm silicon area. The layout for this heterogeneous chip multiprocessor can be seen in Figure 5.39. In this CMP, there are 16 by 8 (128) 1.152mm^2 block areas partitioned for PUs. Within that 2D grid, 98 PU VVM cores, each comprised of 128 mixed-signal VVM cores for a total of 12,544 VVM cores, were placed. The PU VVM core, which was described in detail in Section 5.2.1, was

design to exploit charge-based and stochastic computing for computational efficient fixed-point inner products. Collectively, these mixed-signal PUs can compute over 20 billions MACs per second assuming minimal latency from the networks and interfaces. The characteristics for the PU VVM are detailed in table 5.1.

Additionally, 14 PU CACHE blocks, with a total of 140KB of additional memory, is included in this CMP. These cache blocks are additional memory units used as local storage for the other PUs. As detailed in Section 5.2.3, these cache units are designed with a PU controller, and can be used independently for performing data remapping for PUs and even the main memory.

Furthermore, there are 2 ARM Cortex-M0 processors designed in this CMP as the main CPU. There are also auxiliary units, which include: linear and non-linear morphological processors, a computation memory controller (PU CMC), a high precision DSP, clock generators, and DACs (used for generating the analog signals for the mixed-signal units).

5.5.2 Applications

This mixed-signal CMP is primarily tailored towards accelerating large-scale signal processing and machine learning tasks efficiently with minimal energy cost. Through the thousands of VVM cores implemented, this chip can be applied to real-time processing in wide-area motion imagery of giga-pixels of images for information extraction and analysis. Additionally, this CMP can be used for data intensive machine

learning applications such as image classification through deep neural networks. Also with the heterogeneous design that includes the auxiliary units, this chip can also be generalized for other applications including morphological processing, high precision fixed-point arithmetic for general purpose computing, and even image warping or dewarping using the computation memory controller unit (PU CMC).

5.5.2.1 Wide Area Motion Imagery

Advances in optics[90,91] and the proliferation of cheap CMOS image sensors have enabled the creation of commercially available larger tiled image arrays such as the Kestrel and Simera,[92] CorvusEye 1500[93] and Sentinel CA-247[94] with billions of pixels based on essentially what is cell-phone camera technology. Wide area motion imagery (WAMI)[95] from giga-pixel sensor systems is a rapidly growing data resource for civilian and defense applications. These air-borne systems, aboard a moving platform such as a small plane, a UAV or an aerostat, are capable of imaging objects with a resolution of 0.2 to 0.8 meters at a distance of a few kilometers with giga-pixel image sizes and temporal resolution of a few frames per second (3 to 15 fps).[96] Advanced imaging technologies such as analog[97–99] or all digital[100,101] event based cameras can circumvent the challenges of limited frame rates but the latter have not found their way yet into WAMI systems. Hence WAMI processing pipelines rely extensively on motion dynamic information.

Availability of full motion high resolution data over large, city-size, geographical

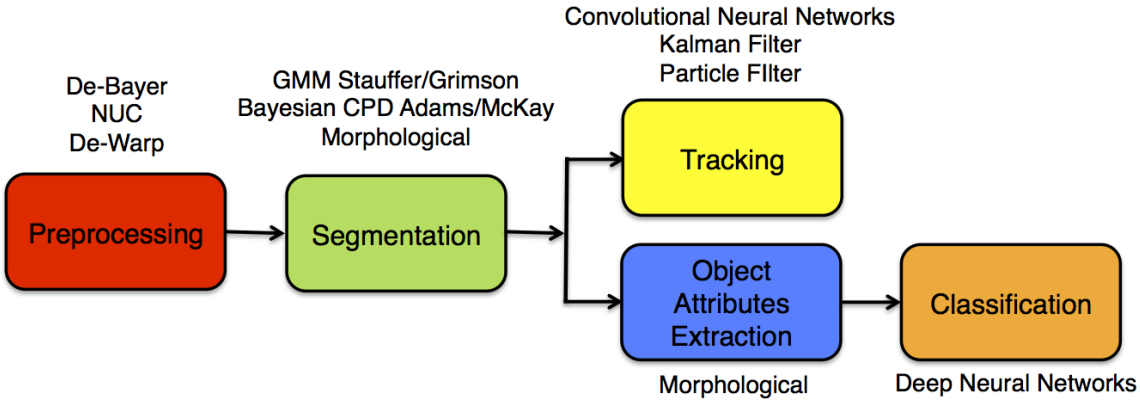


Figure 5.40: Wide Area Motion Imagery Processing Pipeline.

areas, (100 square kilometers) offers unprecedented capabilities for situational awareness. The dynamic nature of the imagery offers insights about actions and patterns of activities that static images do not. Civilian applications of WAMI data allow for the monitoring and intelligent control of traffic across large geographical area and inference of a hierarchy of events and activities that ultimately detail "life-patterns".[102] Additional applications include the coordination of activities in disaster areas and the monitoring of wildlife. Algorithm development for WAMI tasks is facilitated through databases such as CLIF[103] and VIVID[104] and data management standards.[105]

The processing pipeline used for the real-time high velocity wide area motion imagery of giga-pixel sensor systems is shown in Figure 5.40. The processing flow begins with raw pixel values from a camera, and implements DeBayer interpolation, non-uniformity correction, camera motion compensation, background/foreground segmentation, object attributes extraction, object tracking and object classification.

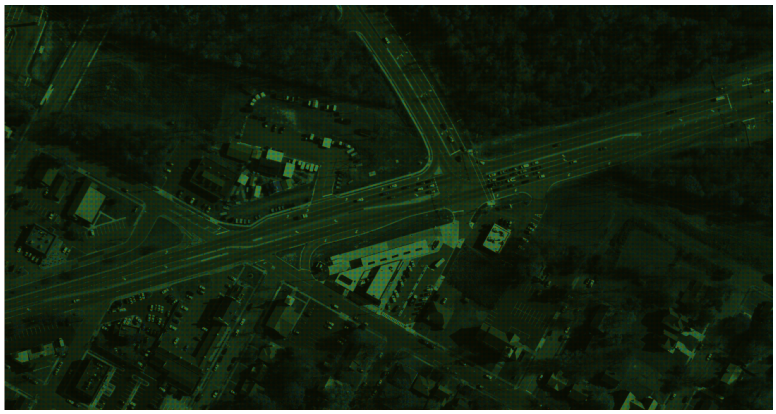
The preprocessing step of the WAMI processing pipeline, as shown in Figure 5.40,

can be done efficiently with the Yupana CMP. Generally, the raw data collected directly from imagers need to be corrected from fabrication variations on the pixel level. To rectify this, a non-uniformity correction (NUC) is done to apply a linear correction with fixed gain and offset values. Additionally, image sensors also use color filter array (CFA) overlaid on top of pixels in order to generate color images. Because all colors cannot be sampled at every pixel location, a CFA with a fixed pattern such as a Bayer filter is used to sub-sample the individual colors. By applying a demosaicing/DeBayer algorithm, the full color image can then be reconstructed from the Bayer pattern image output. Furthermore, a color transform can then be done to convert the color map to a more ideal color encoding for the image processing pipeline. For the preprocessing stage, this CMP will be useful for the NUC, demosaicing, and color transformation of the raw Bayer patterned images into a coherent color image that can be further processed for information extraction and analysis. Figure 5.41 shows a raw 5 mega-pixel image which was preprocessed into a color image, and then color transformed into a grayscale image using the emulated PU on a FPGA.

Using the VVM cores in the CMP, NUC, demosaicing, and color transformation can be done efficiently. For the NUC, which has the operation:

$$y = ax + b, \tag{5.11}$$

where a is the gain, b is the offset, x is the uncorrected pixel, and y is the corrected



(a) Raw Image

(b) Preprocessed Image

Figure 5.41: Wide Area Motion Imagery Preprocessing. A Raw Image is NUC, demosaiced, and color transformed into a grayscale Image.

pixel. The multiplication operation is done on the capacitor array using only the necessary number of capacitors, while the addition operation is done implicitly by loading the offset on the counter of the VVM core.

The demosaicing operation is done on the capacitor array in the VVM cores using fixed weights. For a specific Bayer pattern, such as 'GRBG', which has tiled squares of green, red, blue, and green filters repeated across the pixel array in the imager, a

common filter is used for all color locations. That is,

$$y_i = \frac{(x_1 + x_2 + x_3 + x_4)}{4}, \tag{5.12}$$

where y_i are the colored outputs that are deduced from the average of 4 pixels x_1, x_2, x_3, x_4. The input pixels are chosen based on the pixel location with respect to the Bayer pattern. Since there are either 1, 2, or 4 pixels that are being interpolated for each colored pixel, either each input x_1, x_2, x_3, and x_4 will be unique, or will be duplicated accordingly.

Furthermore, the RGB color image is converted to a grayscale (luminance) image through the RGB-to-YCbCr color transformation. Specifically, the color pixels are interpolated as

$$y_o = 0.299x_r + 0.587x_g + 0.114x_b, \tag{5.13}$$

where x_r is the red pixel, x_g is the green pixel, x_b is the blue pixel, and y_o is corresponding grayscale value. The filter coefficients are quantized to a specific bit precision depending on the required accuracy.

For this image preprocessing, a total of 8 multiplication and 6 addition operations are needed per pixel. In the case of giga-pixel images, the total number of operations to compute at a framerate surpasses a billion. Capable of computing MACs with a throughput of over 20 billions operations per second, this CMP could efficiently be applied to large scale image preprocessing for the WAMI pipeline.

5.5.2.2 Deep Convolutional Neural Networks

Within the past decade, deep convolutional neural networks (DCNNs) have become widely popular in large-scale image classification and recognition tasks.[106,107] They have been applied to huge image databases and repositories, such as ImageNet,[108] with unprecedented success. Nonetheless, because of the structure of these networks and the copious amount of data being processed, large amount of memory is needed to store the parameters and billions of operations are required to run the network. Different techniques have been used to simplify the complexity of these networks including, weight pruning,[109] limiting convolutional filter sizes,[110] network compression through vector quantization or matrix factorization,[111] and more. These algorithmic implementations and system optimization has given rise to DCNN accelerators such as Origami,[112] Eyeriss,[113] and others.[114]

Specifically in this dissertation work, a DCNN, based on a state of the art model,[110] is constructed and trained offline using GPUs, and then implemented for image classification using emulated cores from the Yupana CMP on the CIFAR-10 dataset.[115] This dataset includes 60,000 color images in 10 distinct classes; figure shows a few examples of the images in this dataset. Moreover, the model of the DCNNs is shown in Figure 5.43. This model is composed of 11 layers — an input layer for the 32 by 32 by 3 color images, 6 convolutional rectified linear unit (ReLU) layers with either 64 or 128 (3x3) filters, 2 max pooling layers, 1 average pooling layer, and a fully-connected layer for the classification labels. There are approximately half a million

Figure 5.42: Images from the CIFAR-10 Dataset.

parameters used in this model, which if quantized to 8-bit precision, would require 500KB of memory. During inference, there are total of 15.5 million VVM and add operations needed for the convolutional layers, 17 thousand pooling operations, and 1.28 thousand MAC operations for the full connected layers.

This trained network is evaluated for different bit precision of the data and the parameters during testing. The classification results of the DCNN evaluated on the test dataset of the CIFAR-10 dataset for the different bit precision can be seen in Figure 5.44. Although the network was trained with double floating point precision using GPUs, this model can achieve a high classification accuracy of $\tilde{9}0\%$ with just 5-bit weights and 6-bit data. In fact, notable from the plot, increasing the weight or data precision past 5 bits and 6 bits respectively gives marginal improvements.

The convolutional layers were implemented on the emulated VVM cores for the PU

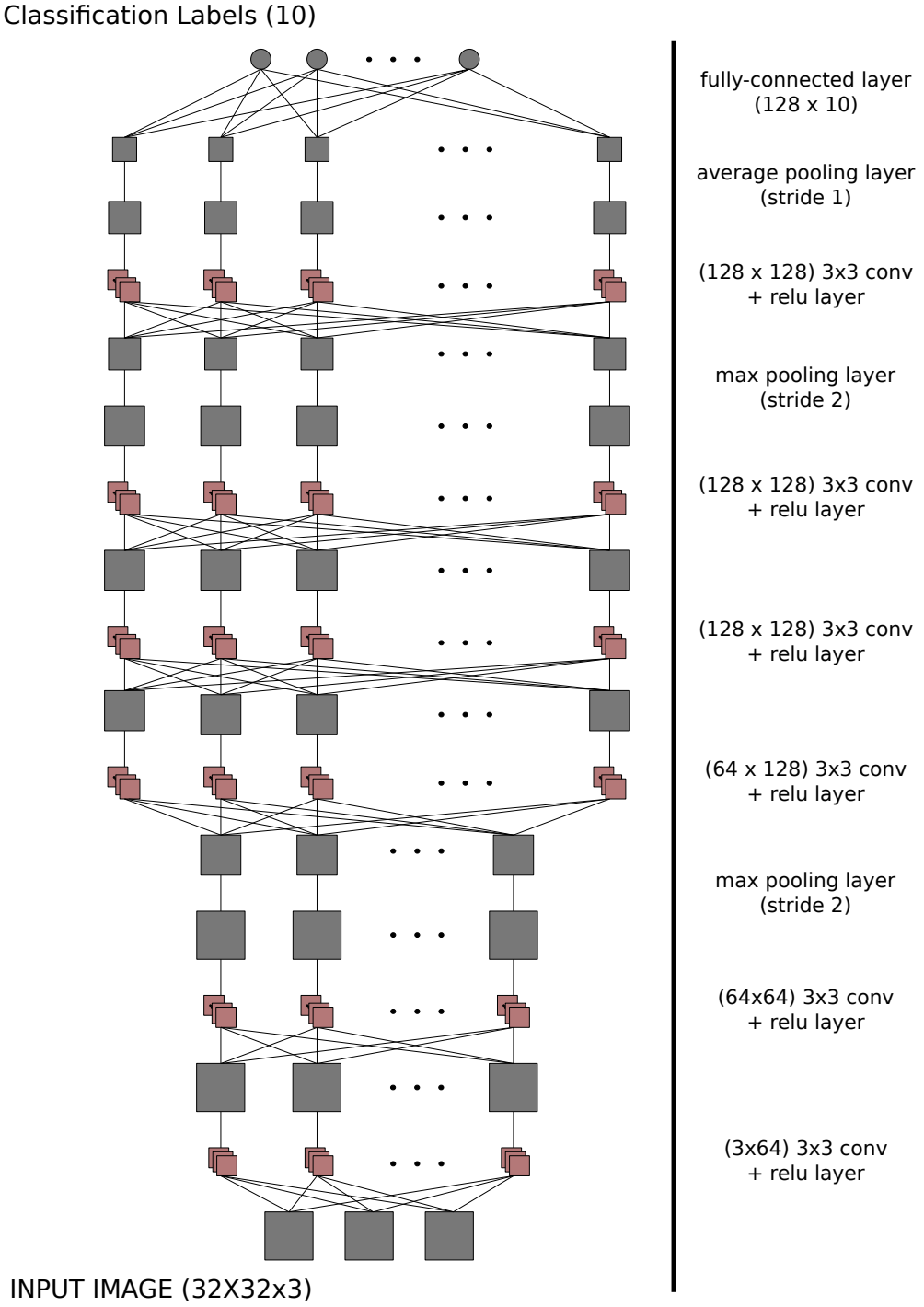


Figure 5.43: A Deep Convolutional Neural Network for the CIFAR-10 Dataset.

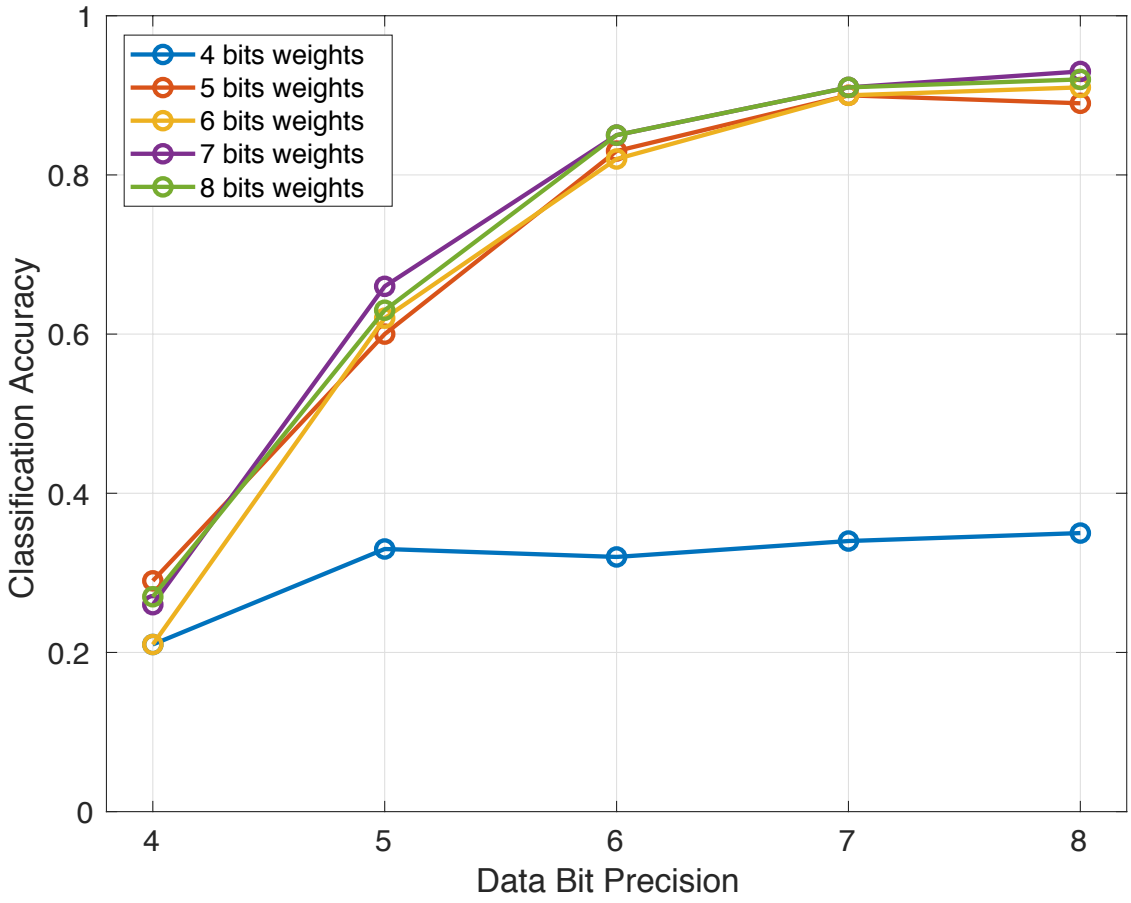


Figure 5.44: Classification Accuracy of the DCNN for Different Data and Weight Precision on the CIFAR-10 Dataset.

VVM. Although the add operation which sums all the intermediate convolved outputs could not be done within the emulated core, all convolutions and ReLU operations were done on this platforms. The inference done with the model using the emulated VVM core achieves a $\tilde{8}8\%$ accuracy when computing with at least 256 samples.

Furthermore, the implementation of deep neural networks can also be useful for the classification step in the WAMI processing pipeline (Figure 5.40). Similar to the example with the CIFAR-10 dataset, a network can be trained offline, and then be

implemented for inference in object classification/recognition using the Yupana CMP.

5.6 Conclusion

The culmination of this dissertation is the multifaceted design of heterogeneous mixed-signal processing units and system modules for a high performance computing and energy efficient 2.5D multiprocessor system-on-chip in a 55nm CMOS process. Formally known as the 2.5D Nano-Abacus SoC, this chip was constructed of 3 CMPs that interface to a high bandwidth 3D DiRAM and a Zynq FPGA through a vertically-integrated interposer board. In this work specifically, the primary focus has been on the different levels of design for the these CMPs from the individual processors to the I/O interfaces.

In order to push performance in these CMPs while also bolstering computational efficiency, special processing units (PUs) were designed. Particularly, a mixed-signal processing unit, the PU VVM, was designed with 128 programmable VVM cores that exploit charge-based and stochastic computing to efficiently compute inner products for a variety of signal processing and machine learning tasks. Laid out in only 1.152mm^2 area, this PU is capable of computing 225 million 8-bit MACs per second with an efficiency of 28.6GOP/W. Additionally, a computational memory controller processor, PU CMC, was designed for rotational and translation data manipulation through address remapping; this PU is capable of doing 61.44 million rotational

and translational operations per second. Moreover, an auxiliary cache unit, the PU CACHE, was designed with 10KB of memory in 1.152mm^2 area for local storage for other PUs.

In addition to the PUs, a specialized instruction set processor, the PU controller, was designed as a DMA controller and a controller for the PU core in order to facilitate distributive computing within the CMP. Composed of only 8 unique instructions, the PU controller can perform multi-dimensional read and write operations across the networks implemented on the chip and also program the PU core to operate synchronously with other units. As seen in the PU VVM, this implemented controller has less than 10% overhead, and has been simulated to run a multi-stage edge detection task (Figure 5.20) with multiple PU VVM cores.

Moreover, other work in this dissertation includes the design of I/O cells for interfacing the input, output, power, and analog signals of the CMP to the periphery blocks, and the design and simulation of a bufferless mesh network useful for large-scale SoCs. The I/O cells, which were used for all the CMPs in the 2.5D Nano-Abacus SoC, were designed and simulated to run at low voltage (as low as 0.3V for the CMP to FPGA I/O interface) and high speed (up to 1GHz). Furthermore, not only was a robust scalable bufferless network presented in this work, a network simulator, designed in MATLAB, was also constructed for prototyping different network models.

These different PUs and system modules were integrated into one of the chips, the Yupana CMP, for large-scale energy efficient data processing. Comprised of 12,544

VVM cores, a high bandwidth memory interface with 2 NoCs, ARM Cortex-M0 cores, and additional auxiliary units in 14mm by 17mm silicon area, this chip was designed to be useful for wide area motion imagery with gigabytes of framed images, large scale inference task with deep convolutional neural networks, and more. Although, the fabricated chip could not be tested due to a costly fault in the final chip assembly, the post-layout simulations have validated the architectural and design approach for this mixed-signal heterogeneous CMP for large-scale efficient data processing.

Appendix A

Mixed-Signal ASIC Design Flow

In this thesis, a bottom-up design approach is used for the construction of the mixed-signal ASIC blocks. Starting with a set of specification, the full block is designed hierarchically to the transistor level. In this hierarchically approach, the individual blocks are designed stand-alone and verified to meet constraints outside the context of the full system. After individual verification, these blocks are then integrated together and verified for the system specification. Moreover, this design flow has been used for the construction of the multiprocessor mixed-signal VVM chips (GF3 VVM, GF4 VVM, GF5 VVM), the PU VVM in the 2.5D Nano-Abacus SoC, the I/O cells, and more.

The outline of this design flow is detailed in Figure A.1. The first phase of the design flow is the custom digital or analog block design. Using the Cadence schematic, layout, and simulation tools, the custom integrated circuits are finalized to the nec-

Figure A.1: Mixed-Signal Design Flow.

essary design specifications. Afterwards, these custom cells are characterized and compiled into a library that can be placed and routed with digital blocks. Next, the digital blocks for the top design, which could include register files, processors, state machines, and more, are synthesized into a gate-level netlist using a standard cell library. This synthesized netlist is then placed and routed with the custom cells for the full top level design, and then verified through the signoff step, which assures that timing, geometric, schematic, and logical verification are met.

In the following sections, each of these steps in the mixed-signal ASIC design flow are discussed in more details.

A.1 Custom Block Design

The first stage of the mixed-signal ASIC design flow is the custom analog and/or digital block design. These custom blocks can be constructed using the design flow shown in Figure A.2.

This iterative design flow is initiated with a behavioral modeling of the custom block to be built. Using a hardware description language (HDL) such as VHDL or

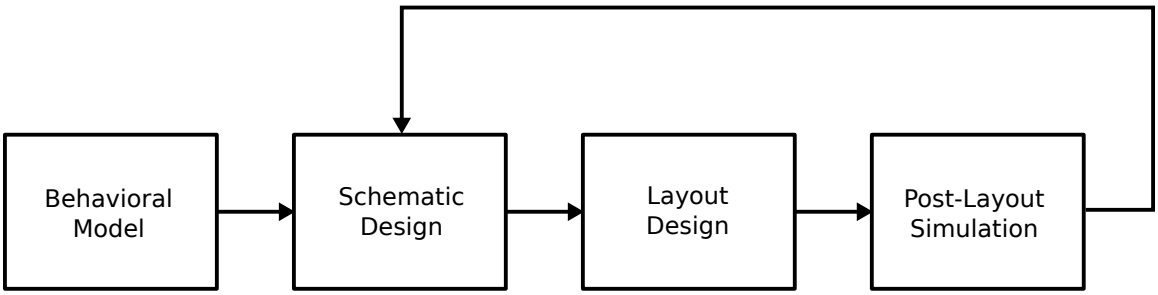


Figure A.2: Custom Block Design Flow.

Verilog-AMS, a functional model of the block is constructed to refine and validate the block specification.

After the behavioral model is finalized and has been simulated to the expected specifications, the circuit schematic is designed in the intended technology process down to the transistor level. In order to ensure that the circuit schematic matches the functionality of the behavioral model, the SPICE netlist of the schematic is simulated in an analog design environment (ADE). In Cadence the Virtuoso ADE can be used in conjunction with the Spectre circuit simulator to run fast and accurate SPICE-level simulations. Additionally, the Virtuoso ADE XL tool can be used to run Monte Carlo simulations for analyzing the block functionality with process and fabrication variations taken into consideration.

Once the schematic has been designed and tested, the circuit layout is then drawn. Generally, a hierarchical bottom-up approach is used for progressively constructing the full layout. Moreover, in order to maintain good signal and power integrity, special consideration is taken for routing; special low resistance routes are given to power and

ground signals to prevent IR drops and electromigration, sensitive analog wires are shielded to minimize cross-talk, and a systematic Manhattan-style routing technique is adapted for efficient wiring of the signals. Once the custom block layout has been completed, the SPICE netlist is extracted and compared with that of the schematic to ensure that the designs match.

Even after matching the layout and schematic of the custom block, it is still imperative to do a post-layout simulation, where parasitic resistive and capacitive components are taken into consideration. Inefficient placement of subblocks and signal routing can have detrimental effects on the functionality and performance of the block. Using the Mentor Graphics Calibre xACT, a parasitic extraction of the layout can be done. The resulting SPICE netlist is simulated using the same tools for simulating the schematic, and if the design doesn't meet the required specification, the process is iterated back to the schematic design stage.

A.2 Library Characterization

After the custom analog and/or digital block has been designed, and the post-layout design has been verified to meet specification, then this block is characterized into a library that be can be used in the next level of hierarchy of the design. In characterizing this custom cell into a library, there are 4 main files that are created:

- Behavioral model description (HDL) file

- Physical abstract description (LEF) file
- Timing description (LIB) file
- Layout description (GDS) file

Each of these files are described in more detail in the following subsections.

A.2.1 Behavioral Model Description (HDL)

The behavioral model description is a high-level hardware description of the custom block, which can be used in system-level simulations of the block with other components. As this is primarily used for functional verification before and after synthesis, this description is constructed as an ideal model without timing annotation, and can be written in the Verilog or VHDL language. Moreover, this HDL file must match the ports of the custom block exactly as this description is also used in the placement and routing of the design.

A.2.2 Physical Abstract Description (LEF)

The library exchange format (LEF) file is an abstract view of the custom block that contains information on pin positions, routing details, place and route (PR) boundary, and metal layer details. This file is used specifically by the electronic design automation (EDA) tool for placing and routing this block, and can be created using the Cadence Abstract Generator software. In this software, first the layout

and logical view are imported, then the abstract block details are extracted, and finally the output LEF file is exported. During the abstract generation, discrepancies between the layout and logical view, incorrect power labeling, and shorting of pins are flagged. Furthermore, additional constraints can be augmented to the LEF such as extra blockages and preferred pin layer extraction and connection.

A.2.3 Timing Description (LIB)

The liberty (LIB) file is a library file that defines the timing, power, noise, input and output, and the process voltage temperature (PVT) characteristics of the custom cell in a particular technology. Generally, this file is deduced from post-layout simulations of the custom block characterized under specific PVT and I/O conditions. Using the Cadence Liberate Characterization tool, the generation of this file for standard cells, I/O, and complex multi-bit cells can be automated. Alternatively, this file can be manually derived from user-based simulations.

This file is structured with 3 unique headers — the library header, the cell header, and the pin header. The library header denotes the library name, the delay models used for relaying timing and power information, and the settings and parameters such as units, threshold point, and PVT values used in the simulation. The cell header is used as an identifier for unique cells within this library; this header describes the cell name, area, leakage power, and footprint (cell type description). Finally, the pin headers encompass all the details for a pin within a cell for the library; this includes

details on the type, direction, function, and capacitance of the pin. For output pins, the maximum output capacitance the pin can drive is included, and the rise/fall delay and the rise /fall transition is defined for given input slew rate and load capacitance. Moreover, the internal power for clock and output pins are also defined in this section.

Furthermore, this LIB file is used during synthesis by the design compiler for timing and power analysis and optimization. Additionally, this file is also used during the place and route stage for the same purpose.

A.2.4 Layout Description (GDS)

The GDS file is a binary file that represents the planar geometric shapes and labels for describing the multi-layered layout hierarchically. This is one of the final deliverable for a chip tapeout used in generating the layer masks for fabrication. The GDS file for the custom cell is used in the top level place and route for generating the final GDS. This file is simply generated within Cadence Virtuoso, by streaming out using the layer map for the targeted technology.

A.3 Digital Design and Synthesis

Once the custom cell library has been established, the next level of hierarchy of the design, which may use these cells, can be synthesized to a gate-level netlist. The design flow to finalize the gate netlist for place and route is illustrated in Figure A.3. The

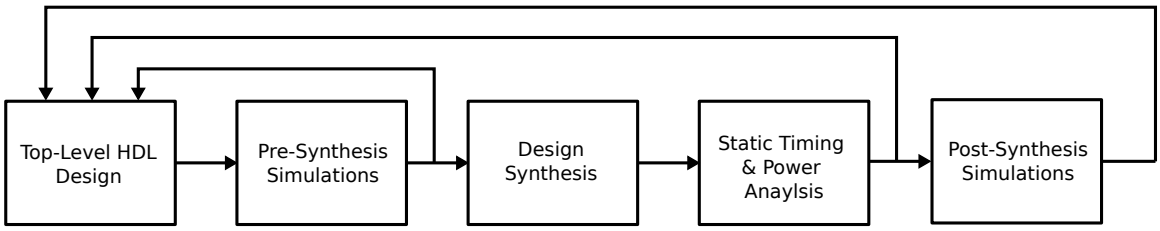


Figure A.3: Digital Design and Synthesis Flow.

process begins with the HDL design. In the hardware description, the synthesizable digital blocks are described with structural and/or behavioral code (either VHDL or Verilog), while the ideal behavioral models from the custom cell library are used for the custom blocks. Afterwards, the block is simulated for functional verification. In this work, this verification is done with the Mentor Graphics ModelSim software tool. If the design doesn't pass verification, it is redesigned, and once it does, it is then synthesized using the Synopsis Design Compiler tool. This synthesis tool, compiles and maps the design to a gate netlist based on the allotted standard cell library. After synthesis, a static timing and power analysis is done to evaluate whether the synthesized block meets expected performance and expectation. Then, a post-synthesis simulation is done with the gate-level netlist to functional verify the design for the place and route stage.

An iterative flow for the design synthesis, where the HDL design is optimized from the pre-synthesis simulations, static timing and power analysis, and post-synthesis simulations, is used. Before the design is placed and routed, the post-synthesis gate-level netlist must not only be functional accurate, but must all meet timing and power

Figure A.4: Place and Route Flow.

constraints.

A.4 Place and Route

After synthesis, the gate-level netlist is placed and routed using the Cadence Innovus tool. The steps of the place and route flow are detailed in Figure A.4. Each of these individual steps are executed through a TCL script in the Cadence Innovus tool.

The place and route process start with floorplanning. During this stage, the die area, I/O area, and core area are set. Additionally, power planning is also done in this stage; the power domains are established and power distribution is laid out through rails, stripes, and rings. For assembling a top level with a pad frame, the I/O cells, pads (C4 or WB), and I/O cell spacers can also be placed in this stage.

After establishing the floorplan, the cells are then placed within the core area. Firstly, the I/O pins, both analog and digital, are placed (generally on the boundary of the core area if the pad frame is not integrated). Then, the custom blocks are placed with considerations to routing congestion and pin locations. Power connections are

made between the custom blocks and the already placed power routes once the custom blocks have been set. Afterwards, the tap cells, which are the bulk connections for the transistors are placed with fixed interval spacing that adhere to the design rule for minimizing latch-up. If necessary, end cap cells are placed after placing taps. Finally, the remaining cells from the gate-level netlist are placed, and then a placement check is done to ensure that all placements are valid. A trial route can be done after the full placement to initially route the design.

Once all the cells are placed, an in-place optimization (IPO) is done. This is an iterative process that extracts the RC model of the placed design, does a static timing analysis for setup and hold timing violations (timing constraints specified in a SDC file), and then reruns the placements with a trial route. To fix timing issues, the tool may move the cells or add inverter and buffer cells for dealing with lengthy paths or problematic capacitive loads. Generally, this stage is over once all timing constraints are met with a valid placement of all cells.

Next, the clock tree for the design is synthesized, placed, and routed. Within the Innovus software, the clock concurrent optimization (CCOpt) tool is used to synthesis trees for clocks constrained for slew, latency, and uncertainty. This process uses both the SDC clock constraints along with the CCOpt configurations for generating the necessary clock tree in the design.

After creating the clock trees, the final stage is the routing and optimization of the design. In this phase, not only does the tool route the full design based on the

routing and timing constraints, but it also fixes all design rule violations (DRV). If routing congestion is an issue, the tool may fail to route without DRV issues, and the whole place and route process may need to be reiterated. Moreover, once the design has been fully routed to the timing constraints without DRV issues, then fillers may be placed to meet design rule checks.

A.5 Signoff

Signoff is the final step in the ASIC design flow for verifying the design before tape-out. The main signoff checks include:

- Design rule check (DRC)
- Formal verification
- Static timing analysis (STA)

These checks ensure that the design is valid for the fabrication process, is functionally accurate, and meets the required performance. The STA check is done within the place and route tool, Innovus, and the design must meet timing under the worse case process, voltage and temperature (PVT) settings. Moreover, the DRC is done on the final GDS streamed into the Virtuoso tool using the Mentor Graphics Calibre DRC software. The last main check, which is the formal verification, is a post-layout simulation done with the Mentor Graphics ModelSim tool to ensure that the final design operates as intended.

In addition to these main checks, there are other important signoff checks that can be done. This includes a schematic verification, LVS, done in the Virtuoso environment, where the post-layout netlist is imported as a schematic and compared with the netlist derived from the GDS. That check can be useful in identifying discrepancies due to inaccuracies in the custom cell library or the top level placement and routing. Other checks include a voltage drop analysis, that measures the integrity of the power grid, and a signal integrity analysis that measures the effects of cross-talk noise on the circuit functionality.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 4, pp. 114–117, 1965.

[2] C. R. Schlottmann and P. E. Hasler, "A Highly Dense, Low Power, Programmable Analog Vector-Matrix Multiplier: The FPAA Implementation," *IEEE Journal of Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 3, pp. 403–411, 2011.

[3] R. Genov and G. Cauwenberghs, "Charge-mode parallel architecture for vector-matrix multiplication," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 48, no. 10, pp. 930–936, Oct. 2001.

[4] R. Genov, "Massively Parallel Mixed-Signal VLSI Kernel Machines," Ph.D. dissertation, Ph.D Dissertation, Johns Hopkins University, Jul. 2002.

[5] R. Karakiewicz, R. Genov, and G. Cauwenberghs, "480-GMACS/mW Resonant Adiabatic Mixed-Signal Processor Array for Charge-Based Pattern Recogni-

tion," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 11, pp. 2573–2584, 2007.

[6] ——, "1.1 TMACS/mW Fine-Grained Stochastic Resonant Charge-Recycling Array Processor," *IEEE Sensors Journal*, vol. 12, no. 4, pp. 785–792, 2012.

[7] B. R. Gaines, "Stochastic Computing Systems," in *Advances in Information Systems Science*, J. F. Tou, Ed. New York: Plenum, 1969, pp. 38–172.

[8] W. Poppelbaum, C. Afuso, and J. Esch, "Stochastic computing elements and systems," in *Proceedings of the 1967 Fall Joint Computer Conference (FJCC)*. ACM, 1967, pp. 635–644.

[9] K. Sanni, T. Figliolia, G. Tognetti, P. Pouliquen, and A. Andreou, "A Charge-Based Architecture for Energy-Efficient Vector-Vector Multiplication in 65nm CMOS," *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2018.

[10] R. Sarpeshkar, "Analog Versus Digital: Extrapolating from Electronics to Neurobiology," *Neural Computation*, vol. 10, no. 7, pp. 1–38, Oct. 1998.

[11] D. J. Brown and C. Reams, "Toward energy-efficient computing," *Communications of the ACM*, vol. 53, no. 3, p. 50, Mar. 2010.

[12] B. J. Hosticka, "Performance comparison of analog and digital circuits," *Proceedings of the IEEE*, vol. 73, no. 1, pp. 25–29, Jan. 1985.

[13] P. M. Furth and A. G. Andreou, "Bit-energy comparison of discrete and continuous signal representations at the circuit level," *Physics of Computation Conference*, pp. 1–9, Nov. 1996.

[14] P. Li and D. Lilja, "Using stochastic computing to implement digital image processing algorithms," in *Proceedings of the 29th IEEE International Conference on Computer Design (ICCD)*, 2011, pp. 154–161.

[15] ——, "A low power fault-tolerance architecture for the kernel density estimation based image segmentation algorithm," in *Proceedings of the 2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2011, pp. 161–168.

[16] W. Qian and M. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," in *Proceedings of the 45th ACM/EDAC/IEEE Design Automation Conference (DAC'08)*, 2008, pp. 648–653.

[17] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. Lilja, "An Architecture for Fault-Tolerant Computation with Stochastic Logic," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 93–105, 2011.

[18] T. Hammadou, M. Nilson, A. Bermak, and P. Ogunbona, "A 96 × 64 intelligent digital pixel array with extended binary stochastic arithmetic," in *Proceedings of the 2003 International Symposium on Circuits and Systems*, May 2003.

BIBLIOGRAPHY

[19] A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," *ACM Transactions on Embedded Computing Systems (TECS)*, pp. 1–25, 2012.

[20] A. Alaghi, C. Li, and J. P. Hayes, "Stochastic circuits for real-time image-processing applications," in *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC'13)*. ACM, 2013, p. 136.

[21] J. L. Molin, T. Figliolia, K. Sanni, I. Doxas, A. G. Andreou, and R. Etienne-Cummings, "FPGA emulation of a spike-based, stochastic system for real-time image dewarping," in *Proceedings of the 58th Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2015, pp. 1–4.

[22] S. Toral, J. Quero, and L. Franquelo, "Stochastic pulse coded arithmetic," in *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2000, pp. 599–602.

[23] P. Mars and H. Mclean, "High-speed matrix inversion by stochastic computer," *IET Electronics Letters*, vol. 12, no. 18, pp. 457–459, 1976.

[24] S. Sharifi Tehrani, S. Mannor, and W. J. Gross, "Fully Parallel Stochastic LDPC Decoders," *IEEE Transactions on Signal Processing*, vol. 56, no. 11, pp. 5692–5703, 2008.

[25] B. D. Brown and H. C. Card, "Stochastic neural computation I: Computational

elements," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 891–905, Sep. 2001.

[26] ——, "Stochastic neural computation II: Soft competitive learning," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 906–920, Sep. 2001.

[27] K. Sanni, G. Garreau, J. L. Molin, and A. G. Andreou, "FPGA Implementation of a Deep Belief Network Architecture for Character Recognition Using Stochastic Computation," in *Proceedings of the 49th Annual Conference on Information Sciences and Systems (CISS)*, Feb. 2015, pp. 1–5.

[28] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition." *Neural Computation*, vol. 22, no. 12, pp. 3207–3220, Dec. 2010.

[29] V. Nair and G. E. Hinton, "3D Object Recognition with Deep Belief Nets," in *Advances in Neural Information Processing Systems 22 (NIPS-2009)*, 2009, pp. 1–9.

[30] A.-r. Mohamed, G. E. Dahl, and G. E. Hinton, "Acoustic Modeling Using Deep Belief Networks," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 1, pp. 14–22, 2012.

[31] F. Seide, G. Li, and D. Yu, "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks." in *2011 International Conference*

on Speech Communication and Technology (INTERSPEECH), 2011, pp. 437–440.

[32] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks." *Science*, vol. 313, no. 5786, pp. 504–507, Jul. 2006.

[33] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for scientific computing conference (SciPy)*, 2010.

[34] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML-09)*. ACM, Jun. 2009.

[35] I. Marian, R. Reilly, and D. Mackey, "Efficient event-driven simulation of spiking neural networks," in *Proceedings of the 3rd WSEAS international conference on neural networks and applications*. MIT Press Cambridge, MA, 2002.

[36] A. Delorme and S. J. Thorpe, "SpikeNET: an event-driven simulation package for modelling large networks of spiking neurons," *Network: Computation in Neural Systems*, vol. 14, no. 4, pp. 613–627, Nov. 2003.

[37] C. J. Lobb, Z. Chao, R. M. Fujimoto, and S. M. Potter, "Parallel event-driven neural network simulations using the Hodgkin-Huxley neuron model," in *Work-*

shop on Principles of Advanced and Distributed Simulation (PADS 2005), 2005, pp. 16–25.

[38] P. O'Connor, D. Neil, S.-C. Liu, T. Delbruck, and M. Pfeiffer, "Real-time classification and sensor fusion with a spiking deep belief network," *Frontiers in Neuroscience*, vol. 7, 2013.

[39] D. Neil and S.-C. Liu, "Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, Dec. 2014.

[40] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[41] Y. Freund and D. Haussler, "Unsupervised learningof distributions on binary vectors using two layer networks," in *Advances in Neural Information Processing Systems 4 (NIPS-1991)*, 1991, pp. 1–8.

[42] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems 19 (NIPS-2006)*. MIT; 1998, 2007, p. 153.

[43] M. A. Carreira-Perpinan and G. E. Hinton, "On contrastive divergence learning," in *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*. Citeseer, 2005, pp. 33–40.

BIBLIOGRAPHY

[44] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[45] A. H. S. Lai and N. H. C. Yung, "A Fast and Accurate Scoreboard Algorithm for Estimating Stationary Backgrounds in an Image Sequence," *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, pp. 241–244, Jun. 1998.

[46] M. H. Sigari, N. Mozayani, and H. R. Pourreza, "Fuzzy Running Average and Fuzzy Background Subtraction: Concepts and Application," *International Journal of Computer Science and Network Security*, vol. 8, no. 2, pp. 1–6, Feb. 2008.

[47] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *Proceedings of the 1999 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Oct. 1999, pp. 1–7.

[48] R. P. Adams and D. J. MacKay, "Bayesian Online Changepoint Detection," *arXiv.org*, Oct. 2007.

[49] D. Culibrk, O. Marques, D. Socek, H. Kalva, and B. Furht, "Neural Network Approach to Background Modeling for Video Object Segmentation," *IEEE Transactions on Neural Networks*, vol. 18, no. 6, pp. 1614–1627, Nov. 2007.

BIBLIOGRAPHY

[50] A. Sobral and A. Vacavant, "A comprehensive review of background subtraction algorithms evaluated with synthetic and real videos ," *Computer Vision and Image Understanding*, vol. 122, no. C, pp. 1614–1627, May 2014.

[51] R. Manohar, "Comparing Stochastic and Deterministic Computing," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2015.

[52] T. Figliolia and A. G. Andreou, "Reconfigurable Stochastic Computation Architecture for Bayesian Online Change Point Detection," *Microprocessors and Microsystems*, pp. 1–27, Feb. 2017.

[53] P. O. Pouliquen, A. G. Andreou, and K. Strohben, "Winner-Takes-All Associative Memory: A Hamming Distance Vector Quantizer," *Analog Integrated Circuits and Signal Processing*, vol. 13, no. 1-2, May 1997.

[54] S. Pavan, R. Schreier, and G. C. Temes, *Understanding Delta-Sigma Data Converters*, 2nd ed. Wiley-IEEE Press, Oct. 2017.

[55] H. Y. Yang and R. Sarpeshkar, "A time-based energy-efficient analog-to-digital converter," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 8, pp. 1590–1601, Jul. 2005.

[56] Y. Chae and G. Han, "Low Voltage, Low Power, Inverter-Based Switched-Capacitor Delta-Sigma Modulator," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 2, pp. 458–472, Jan. 2009.

[57] M. Dessouky and A. Kaiser, "Very Low-Voltage Digital-Audio Modulatorwith 88-dB Dynamic Range Using Local Switch Bootstrapping," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 3, pp. 349–355, Mar. 2001.

[58] L. Yao, M. S. J. Steyaert, and W. Sansen, "A 1-V 140-/spl mu/W 88-dB audio sigma-delta modulator in 90-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 11, pp. 1809–1818, Oct. 2004.

[59] J. Goes, N. Paulino, H. Pinto, R. Monteiro, B. Vaz, and A. Garcao, "Low-power low-voltage CMOS A/D sigma-delta modulator for bio-potential signals driven by a single-phase scheme," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 52, no. 12, pp. 2595–2604, 2005.

[60] E. López-Morillo, R. G. Carvajal, F. Munoz, H. El Gmili, A. Lopez-Martin, J. Ramirez-Angulo, and E. Rodriguez-Villegas, "A 1.2-V 140-nW 10-bit Sigma–Delta Modulator for Electroencephalogram Applications," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 2, no. 3, pp. 223–230, Oct. 2008.

[61] C. Mayr, J. Partzsch, M. Noack, S. Hänzsche, S. Scholze, S. Höppner, G. Ellguth, and R. Schueffny, "A Biological-Realtime Neuromorphic System in 28 nm CMOS Using Low-Leakage Switched Capacitor Circuits," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 10, no. 1, pp. 243–254, Feb. 2016.

[62] M. Dessouky and A. Kaiser, "Very low-voltage fully differential amplifier for

switched-capacitor applications," in *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2000, pp. 441–444 vol.5.

[63] M. Noack, J. Partzsch, C. G. Mayr, and S. Hänzsche, "Switched-capacitor realization of presynaptic short-term-plasticity and stop-learning synapses in 28nm CMOS," *Frontiers in Neuroscience*, pp. 1–14, Jan. 2015.

[64] Y. Wu, X. Cheng, and X. Zeng, "A 960μW 10-bit 70-MS/s SAR ADC with an energy-efficient capacitor-switching scheme," *Microelectronics Journal*, vol. 44, no. 12, Dec. 2013.

[65] K.-P. Pun, L. Sun, and B. Li, "Unit capacitor array based SAR ADC," *Microelectronics Reliability*, vol. 53, no. 3, pp. 505–508, Mar. 2013.

[66] P. Harpe, E. Cantatore, and A. van Roermund, "A 10b/12b 40 kS/s SAR ADC With Data-Driven Noise Reduction Achieving up to 10.1b ENOB at 2.2 fJ/Conversion-Step," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 12, pp. 3011–3018, Dec. 2013.

[67] L. Kull, D. Luu, C. Menolfi, M. Braendli, P. A. Francese, T. Morf, M. Kossel, A. Cevrero, I. Ozkaya, and T. Toifl, "A 24-to-72GS/s 8b Time-Interleaved SAR ADC with 2.0-to-3.3pJ/conversion and ¿30dB SNDR at Nyquist in 14nm CMOS FinFET," *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 358–360, Mar. 2018.

[68] R. Özgün, J. H. Lin, F. Tejada, P. O. Pouliquen, and A. G. Andreou, "A low-power 8-bit SAR ADC for a QCIF image sensor," in *Proceedings of the 2011 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2011, pp. 841–844.

[69] N. Verma and A. P. Chandrakasan, "An Ultra Low Energy 12-bit Rate-Resolution Scalable SAR ADC for Wireless Sensor Nodes," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 6, pp. 1196–1205, Jun. 2007.

[70] Y.-K. Chang, C.-S. Wang, and C.-K. Wang, "A 8-bit 500-KS/s Low Power SAR ADC for Bio- Medical Applications," *2007 IEEE Asian Solid State Circuits Conference (ASSCC'07)*, pp. 228–231, Jun. 2007.

[71] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, May 2011.

[72] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *IEEE Computer*, vol. 38, no. 11, pp. 32–38, Nov. 2005.

[73] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings AFIPS Spring Joint Computer Conference*, 1967, pp. 483–485.

[74] H. W. Dong and H. Lee, "Extending Amdahl's Law for energy-efficient com-

puting in the many-core era," *IEEE Computer*, vol. 41, no. 12, pp. 24–31, Dec. 2008.

[75] A. S. Cassidy and A. G. Andreou, "Beyond Amdahl's Law: an objective function that links multiprocessor performance gains to delay and energy," *IEEE Transactions on Computers*, vol. 61, no. 8, pp. 1110–1126, Aug. 2012.

[76] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th Annual Design Automation Conference (DAC)*, Jun. 2007, pp. 746–749.

[77] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Jun. 2009.

[78] P. G. Emma and E. Kursun, "Is 3D chip technology the next growth engine for performance improvement?" *IBM Journal Of Research And Development*, vol. 52, no. 6, pp. 541–552, 2008.

[79] N. P. Jouppi, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*, Jun. 2017, pp. 1–17.

[80] S. Furber, F. Galluppi, S. Temple, and L. Plena, "The SpiNNaker Project," *Proceedings of the IEEE*, pp. 1–17, 2014.

[81] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.

[82] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Feb. 2018.

[83] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *Proceedings of the 2002 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2002, pp. 105–112.

[84] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the 38th Annual Design Automation Conference (DAC)*, 2001, pp. 684–689.

[85] J. J. H. Pontes, M. T. Moreira, F. G. Moraes, and N. L. V. Calazans, "Hermes-

AA: A 65nm asynchronous NoC router with adaptive routing," in *Proceedings 2010 System on Chip Conference (SOCC'10)*, 2010, pp. 493–498.

[86] G. Chen, M. A. Anders, H. Kaul, S. K. Satpathy, S. K. Mathew, S. K. Hsu, A. Agarwal, R. K. Krishnamurthy, V. De, and S. Borkar, "A 340 mV-to-0.9 V 20.2 Tb/s Source-Synchronous Hybrid Packet/Circuit-Switched 16x16 Network-on-Chip in 22 nm Tri-Gate CMOS," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 59–67, 2015.

[87] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, 2009.

[88] C. Fallin, C. Craik, and O. Mutlu, "CHIPPER: A low-complexity bufferless deflection router," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 144–155.

[89] C. Fallin, G. Nazario, X. Yu, and K. Chang, "Bufferless and Minimally-Buffered Deflection Routing," in *Routing Algorithms in Networks-on-Chip*, M. Palesi and M. Daneshtalab, Eds. Springer Science and Business Media, 2014, pp. 241–275.

[90] D. J. Brady, M. E. Gehm, R. A. Stack, D. L. Marks, D. S. Kittle, D. R. Golish, E. M. Vera, and S. D. Feller, "Multiscale gigapixel photography," *Nature*, vol. 486, no. 7403, pp. 386–389, Jun. 2012.

BIBLIOGRAPHY

[91] D. Brady. (2014, Feb.) AWARE2 Multiscale Gigapixel Camera. [Online]. Available: http://disp.duke.edu/projects/AWARE/

[92] Logos-Technologies, "Multi-Sensor, Wide-Area Persistent Surveillance," pp. 1–2, Sep. 2015.

[93] Harris-Corporation, "CorvusEye 1500," pp. 1–4, 2015.

[94] UTC-AerospaceSystems, "ISR Systems," pp. 1–7, Nov. 2015.

[95] R. Porter, A. Fraser, and D. Hush, "Wide-Area Motion Imagery," *IEEE Signal Processing Magazine*, vol. 27, no. 5, pp. 56–65, Sep. 2010.

[96] Logos-Technologies, "Simera: Lightweight, Wide-Area Persistent Surveillance Sensor for Aerostats," pp. 1–2, Sep. 2015.

[97] E. Culurciello, R. Etienne-Cummings, and K. A. Boahen, "A biomorphic digital image sensor," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 2, pp. 281–294, Feb. 2003.

[98] E. Culurciello and A. G. Andreou, "3D integrated sensors in silicon-on-sapphire CMOS," in *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2006.

[99] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128x128 120dB 15us Latency Asynchronous Temporal Contrast Vision Sensor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 2, pp. 566–576, 2008.

[100] C. G. Rizk, P. O. Pouliquen, and A. G. Andreou, "Flexible Readout and Integration Sensor (FRIS): new class of imaging sensor arrays optimized for air and missile defense," *Johns Hopkins APL Technical Digest*, vol. 28, no. 3, pp. 252–253, Jan. 2010.

[101] J. H. Lin, P. O. Pouliquen, A. G. Andreou, A. C. Goldberg, and C. G. Rizk, "Flexible readout and integration sensors (FRIS): a bio-inspired, system-on-chip, event based readout architecture," in *Proceedings of SPIE: Infrared Technology and Applications XXXVIII Conference*, May 2012, pp. 8353–1N.

[102] C. Stauffer and W. E. L. Grimson, "Learning patterns of activity using real-time tracking," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 747–757, 2000.

[103] SDMS. (2006) Columbus Large Image Format (CLIF-2006) Dataset.

[104] R. T. Collins, X. Zhou, and S. K. Teh, "An open source tracking testbed and evaluation web site," in *IEEE International Workshop on Performance Evaluation of Tracking and Surveillance (PETS 2005)*, 2005.

[105] R. Thakkar, "A primer for dissemination services for Wide Aray Motion Imagery," Tech. Rep. OCG 12-077r1, Dec. 2012.

[106] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with

deep convolutional neural networks," in *Advances in Neural Information Processing Systems 28 (NIPS-2015)*, 2012.

[107] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv.org*, p. 1556, Sep. 2014.

[108] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2009, pp. 248–255.

[109] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," *arXiv.org*, Nov. 2016.

[110] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," *arXiv.org*, Dec. 2014.

[111] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing Deep Convolutional Networks using Vector Quantization," *arXiv.org*, Dec. 2014.

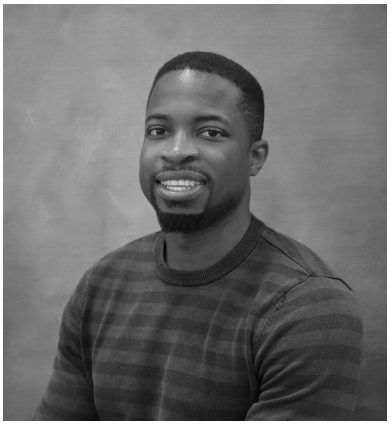
[112] L. Cavigelli and L. Benini, "Origami: A 803 GOp/s/W Convolutional Network Accelerator," *arXiv.org*, Dec. 2015.

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

[114] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems," in *2016 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2016, pp. 264–265.

[115] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Ph.D. dissertation, Ph.D. Dissertation, University of Toronto, Apr. 2009.

Vita



Kayode Sanni received a B. S. degree in Computer Engineering from the University of Maryland, Baltimore County (UMBC) in 2012. While at UMBC, he also received the Meyerhoff Scholarship, and participated in summer research internships at Johns Hopkins University, University of Michigan, and Georgia Institute of Technology. He enrolled in the Electrical & Computer Engineering Ph.D. program at Johns Hopkins University in 2012, where he completed a M. S. degree in 2014. He received the Louis. M. Brown Engineering fellowship along with the JHU ECE departmental fellowship during 2012-2013. His research focuses on mixed-signal VLSI and system design for heterogeneous chip multiprocessors, and his dissertation work has contributed to the DARPA-sponsored Unconventional Processing of Signals for Intelligent Data Exploitation (UPSIDE) project.