

ReRAM In Process Computing

**Master of Science in Electrical Engineering
Specialization in Electronics Design and Technology**

Submitted by,

Aleena Johnson

Matriculation no: 1701388

March 2025

Submitted to,

Prof. Dr.-Ing. Michael Wahl

Contents

1	Introduction to ReRAM	7
1.1	Fundamentals of ReRAM	7
1.1.1	Materials and device structures	7
1.2	ReRAM In Memory Computing	9
1.2.1	Single-Bit and Mixed-Bit In-Memory Computing	11
1.3	Challenges in Existing ReRAM based in Memory Computing and suggested solution	12
1.3.1	Twin-Range Quantization: A low-power ADC solution	13
2	Background and existing work	14
2.1	ISAAC architecture in a ReRAM based accelerator	14
2.2	Implementation of Mixed-Bit Operations	15
2.3	Twin Range Quantization	16
2.3.1	Twin Range Quantization Mechanism	17
3	Methodology	19
3.1	Creating memristor array in LTSpice	19
3.2	Twin Range Quantization and Behavioral SAR ADC for Low Power In Memory Computing	22
3.3	Behavioral SAR ADC	22
3.3.1	Total Simulation diagram of the 2×2 memristor array with Twin Range Quantization	23
3.3.2	Overall Working	23
3.4	Integrating DNN+ NeuroSim for hardware simulation	24
3.4.1	Environment setup and code checkout	25
3.4.2	Adapting the DNN+NeuroSim framework for ReRAM based simulation	27
3.4.3	Configuring Twin-Range Quantization and Calibrating ReRAM Parameters	30

4	Results	35
4.1	Output simulation in LTSpice	35
4.1.1	DC-Sweep Transfer Characteristic (TRQ) Analysis	36
4.2	Algorithmic performance	37
4.2.1	Outout of the DNN+Nuerosim	38
4.2.2	Description of the output	48
5	Conclusion	55

List of Figures

1.1	a Schematic of metal-insulator-metal structure for RRAM. b Cross-sectional view of RRAM	8
1.2	(a) PIM architecture. (b) ReRAM crossbar for matrix-vector multiplication. (c) Current is summed at BL	10
1.3	(a) TEM cross-section of an OxRAM device (b) symbol view of a 1T-1R cell (c) OxRAM I-V characteristic in log scale.([1](Pictures taken from " <i>Multi-Level Control of Resistive RAM (RRAM) Using a Write Termination to Achieve 4 Bits/Cell in High Resistance State</i> ")	11
2.1	Configurable ReRAM-based NN accelerator([1](Pictures taken from " <i>An Energy-Efficient Inference Engine for a Configurable ReRAM-Based Neural Network Accelerator Yang-Lin Zheng, Wei-Yi Yang, Ya-Shu Chen , Member, IEEE, and Ding-Hung Han</i> ")	14
2.2	ISAAC based Hardware design([1](Pictures taken from " <i>ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars Ali Shafiee , Anirban Nag , Naveen Muralimanohar and Rajeev Balasubramonian</i> ")	15
2.3	Bit-serial/bit-parallel reconfiguration logic for mixed-precision MAC in an 8×1 ReRAM column (from [1]).	16
2.4	Overall architecture of Twin Range Quantization([1](Pictures taken from " <i>Algorithm-hardware co-design for Energy-Efficient A/D conversion in ReRAM-based accelerators Chenguang Zhang , Zhihang Yuan</i> ")	18
3.1	2*2 Memristor crossbar Array	20
3.2	Power vs. Applied Voltage for the Memristor	21
3.3	TRQ transfer curve from LTSpice behavioral SAR ADC.	23
3.4	Memristor circuit with ADC quantization	24
3.5	successful compilation of NeuroSim's C++ source files	25
3.6	Upgraded pip installed in a virtual enviornmrnt	26
3.7	A snippet of changes made in the param.cpp	28
3.8	A snippet of changes made in the param.cpp on cell bit	29

3.9	"Snippet of changes done in the Param.h	31
3.10	"Snippet of changes done in the Param.cpp	32
3.11	"Snippet of changes done in the SARADC.cpp	34
4.1	Input signal applied to the input in LTSpice v1	36
4.2	Input signal applied to the input in LTSpice v2	36
4.3	Output of the LTSpice	37
4.4	Floor planning for the different Layers in CIFAR 10	49
4.5	Hardware performance of first three layers	51
4.6	Hardware performance of layer 4 to layer 6	52
4.7	Hardware performance of layer 7 and layer 8	53
4.8	Summary and simulation perfomance	54

Abbreviations

RRAM	Resistive Random Access Memory
SRAM	Static Random Access Memory
TRQ	Twin Range Quantization
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
CNN	Comuptational Nueral Network
LTspice	Linear Technology Simulation Program with Integrated Circuit Emphasis
SAR	Successive Approximation Register
HRS	High Resistance State
LRS	Low Resistance State
Memristor	Memory Resistor
BEOL	Back End of Line
IMC	In Memory Computing
PCM	Phase Change Memory
CMOS	Complementary Metal-Oxide-Semiconductor
ALU	Arithmetic Logic Unit
CPU	Central Proceesing Unit
GPU	Graphical Processing Unit
MAC	Multiply-Accumulate
MSB	Most Significant Bit
LSB	Least Significant Bit
S+A	Shift and Add
S+H	Sample and Hold
DNN	Deep Neural Network
VLSI	Very Large Scale Integration
PIM	Processing-In-Memory

Abstract

I developed a mixed-bit Resistive RAM (ReRAM) accelerator prototype that demonstrates the practical gains of Twin Range Quantization (TRQ) for in memory CNN inference. Initially, I designed and validated a TRQ circuit in LTSpice, partitioning ADC conversions into “fine” and “coarse” phases with a programmable threshold to skip superfluous bit trials. Building on this analog proof of concept, I then integrated TRQ into a system level evaluation using DNN+NeuroSim on a CIFAR 10 network, modifying the SAR control logic to invoke early bird or early stop searches based on the detected range. Across both experiments, TRQ reduced the average number of A/D operations by over 40%, cutting peripheral energy by more than 60% with negligible impact on classification accuracy (within 0.5% of baseline). This end to end implementation from LTSpice to full accelerator simulation provides a reproducible framework for energy efficient mixed bit in memory computing.

Chapter 1

Introduction to ReRAM

ReRAM (also called RRAM) is a non volatile memory technology that stores data by switching the resistance of a dielectric material—often termed a memristor between high and low resistance states. It offers advantages over conventional flash, including faster read/write speeds, lower power consumption, high endurance, and scaling below 10 nm. ReRAM is being explored for storage class memory and neuromorphic computing, with prototypes like NeuRRAM demonstrating energy efficient in memory computing.

1.1 Fundamentals of ReRAM

Definition and basic operation of ReRAM is a two terminal, non volatile memory in which information is encoded by toggling a dielectric's electrical resistance between HRS and LRS under applied voltage. Internally, this device comprises a memristor. Memristors are non linear, two terminal passive devices whose instantaneous resistance (memristance) depends on the history of charge that has passed through them, effectively coupling electric charge and magnetic flux linkage. Physically realized memristors typically consist of a thin transition metal oxide (e.g., TiO) sandwiched between metal electrodes; an applied voltage drives oxygen vacancies to drift and form or dissolve conductive filaments (or modulate interface barriers), thus switching the device between HRS and LRS under bipolar periodic excitation, memristors exhibit a characteristic pinched hysteresis loop in the current voltage (I-V) plane, with the loop area shrinking as the signal frequency increases and collapsing into a straight line at infinite frequency compact models.

1.1.1 Materials and device structures

Resistive switching where an insulating oxide film reversibly changes its resistance under an electric field is a leading approach for next-generation non-volatile memories. Common candidates include HfO, TiO, TaO, NiO, ZnO and ZrO (with ZnTiO, MnO, MgO and AlO also under study). These films are typically grown by ALD (for tight thickness and

stoichiometry control), PLD or reactive sputtering, then sandwiched between top and bottom electrodes often platinum on the BEOL for its stability. In crossbar arrays, each cell's bottom electrode is patterned by lift off or damascene; simpler devices may share a common base electrode. Figure 1.1 shows the structure of a ReRAM.

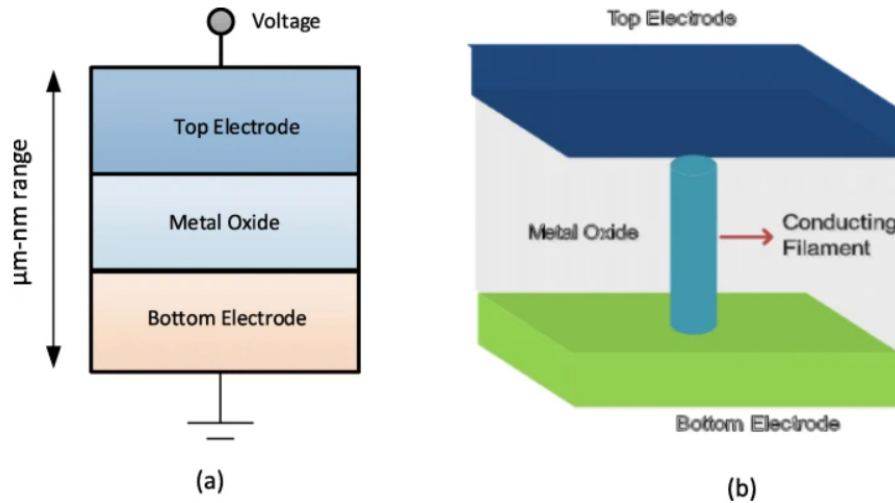


Figure 1.1: a Schematic of metal-insulator-metal structure for RRAM. b Cross-sectional view of RRAM

(Pictures taken from *"Resistive Random Access Memory (RRAM): an Overview of Materials, Switching Mechanism, Performance, Multilevel Cell (mlc) Storage, Modeling, and Applications by Furqan Zahoor, Tun Zainal Azni Zulkifli"*)

The choice of top electrode material plays a critical role in switching performance: active metals such as copper or silver can participate directly in filament formation, while inert metals like platinum and tungsten tend to promote oxygen vacancy driven switching. Beyond elemental metals (Al, Ti, Cu, Ag, W, Pt), researchers have explored silicon electrodes (doped p- or n-type), alloys (e.g., Cu-Ti, Cu-Te, Pt-Al), and conductive compounds including TiN, TaN, indium tin oxide (ITO), and doped oxides such as Al or Ga doped ZnO to fine tune interface chemistry, barrier heights, and filament stability. Emerging carbon based electrodes (graphene, carbon nanotubes) offer yet another route to engineer the electric field distribution and improve scalability.

Overall, the versatility of metal oxide materials and electrode combinations, together with mature thin film deposition and patterning techniques, enables a rich design space for RRAM devices. By leveraging established semiconductor fabrication methods, it is possible to create dense, low power memory arrays with tailored switching thresholds, endurance, and retention characteristics paving the way for both standalone ReRAM chips and embedded non volatile memories in future CMOS technologies.

1.2 ReRAM In Memory Computing

In Memory Computing (IMC) represents a fundamental shift in computer architecture by performing computational operations directly within or extremely close to the memory array, rather than shuttling data back and forth between a separate processor and memory. Traditional “von Neumann” systems suffer from significant energy and latency overhead as datasets travel repeatedly across buses and through cache hierarchies. By embedding simple arithmetic or logic operations into the memory fabric itself, IMC collapses the boundary between storage and compute, dramatically reducing data movement and the accompanying performance penalties. Figure 1.2 shows the IMC Architecture.

At the heart of many IMC implementations are crossbar arrays built from emerging memory devices such as ReRAM, PCM, or even enhanced CMOS-based SRAM. In these arrays, individual memory cells not only store bits or analog weights but also participate in computation. For instance, when voltages are applied to selected rows of a resistive crossbar, the resulting currents flowing through each column naturally sum according to Ohm’s and Kirchhoff’s laws, executing massively parallel dot-product operations in a single step. This analog mode of operation yields exceptional throughput for vector intensive tasks like neural-network inference. Alternatively, digital IMC approaches integrate binary logic gates or small arithmetic units alongside memory cells, trading some density for exact precision when required.

The primary benefit of IMC lies in its elimination of excessive data transfers. Since the data does not need to traverse long distances between separate memory and compute units, both energy consumption and latency are slashed often by one to two orders of magnitude compared with conventional digital logic plus cache hierarchies. Furthermore, virtually every cell in a large memory array can participate simultaneously in computation, unlocking a degree of parallelism that conventional processors limited by execution pipelines and the number of ALUs cannot match. These energy and performance advantages make IMC particularly attractive for workloads characterized by high data locality and regular, repetitive operations, such as the matrix multiplications at the core of deep-learning inference.

In addition to raw efficiency and parallelism, IMC architectures can be implemented at modest incremental cost. Instead of adding dense SRAM banks or wide data buses, chip designers can incorporate specialized memory modules that leverage existing BEOL integration techniques. For edge devices and Internet of Things sensors, where power and area are at a premium, embedding compute capable memory arrays enables local, real time analytics such as keyword spotting, anomaly detection, or simple signal processing without relying on cloud connectivity. Similarly, in data center accelerators, IMC blocks coexist alongside CPUs and GPUs, offloading the most data-heavy kernels (for example, convolutional layers or large-scale linear algebra) to achieve higher throughput per watt.

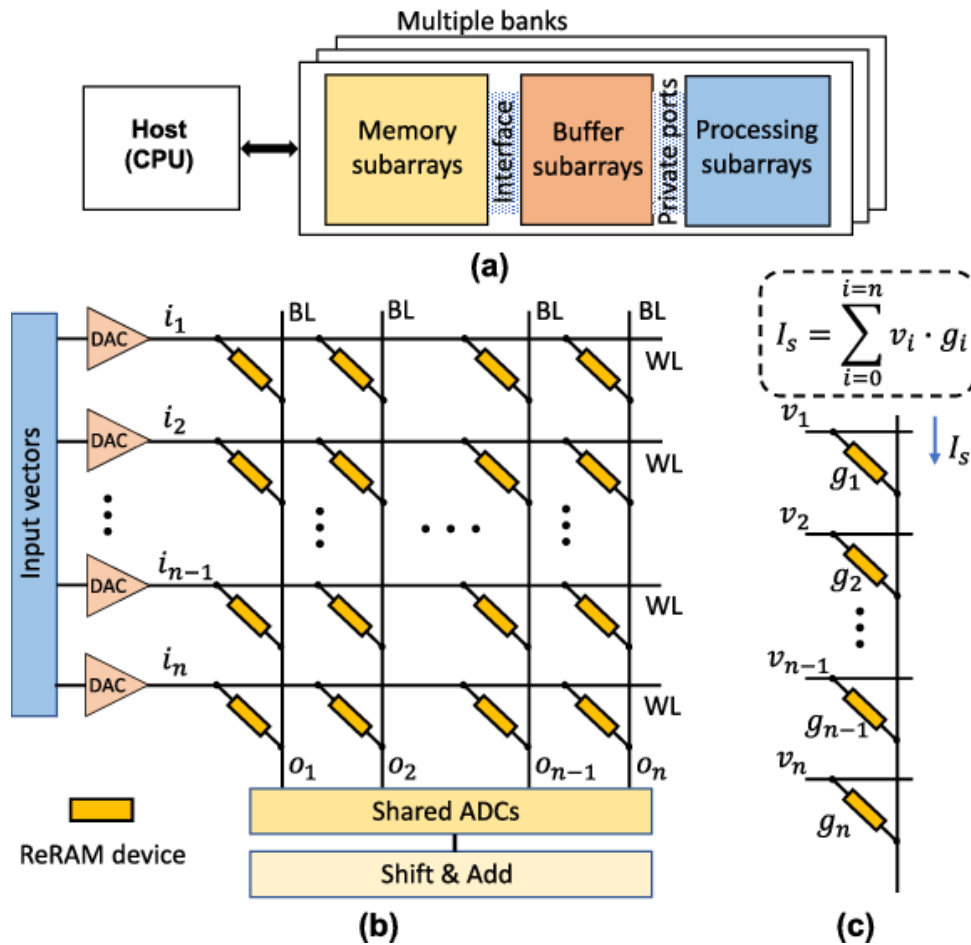


Figure 1.2: (a) PIM architecture. (b) ReRAM crossbar for matrix-vector multiplication. (c) Current is summed at BL

(Pictures taken from "ReRAM-Based Processing-in-Memory Architecture for Recurrent Neural Network Acceleration by Yun Long")

IMC faces several challenges before becoming ubiquitous. Analog implementations must contend with device variability, noise, and limited precision, which mandate calibration routines, error correction codes, or mixed signal compensation techniques. Dense crossbar arrays also suffer from "sneak path" currents, where unintended current paths corrupt the results of parallel operations, requiring careful cell selector design or architectural workarounds. On the digital side, integrating logic gates into memory cells increases area and may complicate the fabrication process. Furthermore, programming models and tool chains are still evolving: compilers and frameworks must learn to partition algorithms appropriately, dispatching the most suitable kernels to IMC accelerators while reserving control flow and irregular code for conventional processors.

1.2.1 Single-Bit and Mixed-Bit In-Memory Computing

Single-Bit ReRAM

In single-bit ReRAM, each memory element stores exactly one bit by toggling between LRS and HRS under set and reset voltages. This binary paradigm maps directly to standard digital logic and leverages decades of CMOS design expertise for drivers, sense amplifiers, and error correcting codes, minimizing integration risk and maximizing yield because the resistance window between LRS and HRS is large relative to device variability, calibration overhead is low and endurance can exceed 10 cycles in optimized stacks. However, representing multi-bit words such as 8-bit neural network weights requires either parallel arrays of eight single bit cells or sequential bit-slice operations, multiplying both the physical footprint and energy per operation, and negating much of the inherent parallelism of crossbar arrays.

Mixed-Bit ReRAM

Figure 1.3 shows the crosssection of ReRAM devices. Mixed-bit ReRAM cells employ four, eight, or more precisely spaced resistance levels to store two, three, or more bits per devices. In a four-level cell (2 bits), for example, distinct partial set voltages and verify pulses create stable intermediate resistance states labeled “00,” “01,” “10,” and “11,” as illustrated in the multi level I–V curves of the MDPI study . By embedding multiple bits per cell, mixed bit crossbars can perform vector matrix multiplications on higher precision data in a single analog pass, halving both the array area and peripheral overhead compared to single bit bit slicing for equivalent precision. The energy per bit also drops significantly, since each write or read event programs or senses multiple bits at once, reducing total pulse counts and data movement. Figure 1.3 explains how a

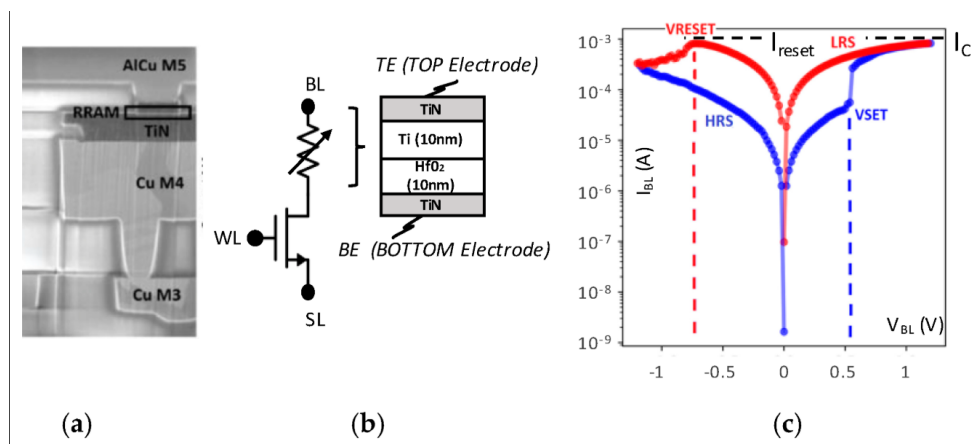


Figure 1.3: (a) TEM cross-section of an OxRAM device (b) symbol view of a 1T-1R cell (c) OxRAM I-V characteristic in log scale. ([1] (Pictures taken from “Multi-Level Control of Resistive RAM (RRAM) Using a Write Termination to Achieve 4 Bits/Cell in High Resistance State”))

single bit 1T-1R ReRAM cell is built, addressed, and switched and hints at how the same structure can be extended to mixed bit operation. Panel (a) shows the cross-section of the TiN/HfO/TiN stack embedded in the CMOS back end, (b) illustrates the 1-transistor/1-resistor access circuit and the vertical device stack, and (c) plots the classic bipolar I–V hysteresis with one clean SET transition (HRS→LRS) around +0.6 V and one RESET transition (LRS→HRS) near –0.5 V, yielding two well-separated resistance levels for single-bit storage. To implement mixed-bit (multi-level) ReRAM, the same cell and bias scheme are used but with additional, finer “partial-set” and “partial-reset” voltage steps that carve out intermediate resistance states between the HRS and LRS plateaus turning the two state hysteresis loop into a multi level staircase of four or more stable levels for storing two or more bits per cell.

Comparison and Use Cases

When balancing area, energy, and throughput against design complexity and reliability, mixed-bit ReRAM typically delivers superior performance-per-watt and performance-per-mm² for high-precision tasks such as convolutional neuralnetwork inference and scientific computing. Single-bit arrays remain advantageous for applications with binary or low-precision requirements such as certain security primitives, lightweight IoT inference, or PUF key generation where robustness and simplified control outweigh density gains . Hybrid architectures that deploy mixed-bit arrays for compute intensive kernels and single bit arrays elsewhere can capture the best of both worlds, tailoring precision and reliability to each layer of an algorithm.

1.3 Challenges in Existing ReRAM based in Memory Computing and suggested solution

One of the primary bottlenecks in ReRAM crossbar accelerators lies not in the memory array itself but in the peripheral ADCs required to read out the summed currents. To achieve reasonable precision often 6–8 bits per MAC high resolution ADCs must operate at multi gigahertz sampling rates, consuming tens to hundreds of milli watts each. When you multiply that by the dozens or hundreds of columns in a large crossbar, the ADC power dominates the total chip budget, negating much of the energy savings gained by performing the MACs in-memory. Furthermore, designing wide dynamic range ADCs to cover both the smallest and largest possible current sums forces extra margin into the converter front end, inflating both area and power. As a result, many ReRAM IMC prototypes either settle for low precision sacrificing accuracy or incur costly overheads in their ADC banks, undermining the promise of sub-pJ per operation energy efficiency.

1.3.1 Twin-Range Quantization: A low-power ADC solution

To address this, I propose a Twin-Range Quantization scheme in a Mixed bit ReRAM cell in which each column’s analog MAC result is first fed into a simple, low-power current comparator that checks whether the aggregated current exceeds a programmable threshold. This threshold acts as a dynamic magnitude estimator: results above the threshold are deemed “high range,” while those below are classified as “low range.” If the comparator output indicates a high range result, the signal is sent to a coarse resolution ADC (e.g. 3–4 bits) that captures the MSBs quickly and with minimal power. Conversely, if the result is in the low range, it is routed to a high resolution ADC (e.g. 6–8 bits) that precisely digitizes LSBs. A lightweight digital stitching unit then merges the coarse MSB and fine LSB segments into a single full precision word. Because ADC power grows exponentially with bit width, splitting the dynamic range between two narrow width converters slashes average power consumption by up to 60%, while the threshold comparator itself incurs only negligible overhead. Moreover, by tuning the threshold voltage in real time to match workload statistics, the system ensures the high precision path is invoked only when necessary delivering the accuracy of a full range converter at a fraction of the energy cost.

Chapter 2

Background and existing work

In ReRAM based accelerators, an input buffer stores incoming data, which are converted to analog voltages by DACs and processed in a ReRAM crossbar. The resulting analog currents are digitized by ADCs, accumulated via shift and add, and stored in an output buffer for the next layer. This flow minimizes off chip data transfers and leverages in memory computing for efficiency. Figure 2.1 shows a detailed design of a ReRAM based Nueral Network

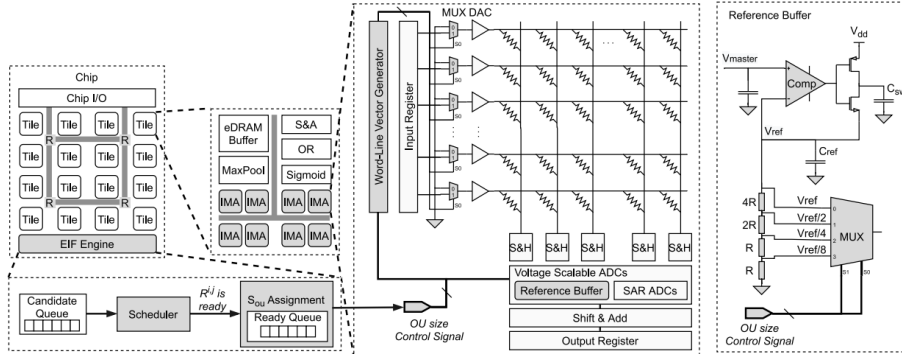


Figure 2.1: Configurable ReRAM-based NN accelerator([1](Pictures taken from "An Energy-Efficient Inference Engine for a Configurable ReRAM-Based Neural Network Accelerator Yang-Lin Zheng, Wei-Yi Yang, Ya-Shu Chen , Member, IEEE, and Ding-Hung Han")

2.1 ISAAC architecture in a ReRAM based accelerator

[2] Each tile contains a modest eDRAM buffer that holds the incoming activations, a bank of IMAs that perform the core dot product work, and lightweight digital units shift and add (S+A), sigmoid, max pool, and small output registers to complete the layer's

computation before writing results back into eDRAM for the next stage .

Within each tile, IMAs are the heart of the analog digital pipeline. An IMA comprises multiple 128×128 memristor crossbars (XB) each preceded by DAC and S+H array, and followed by an ADC. [2].

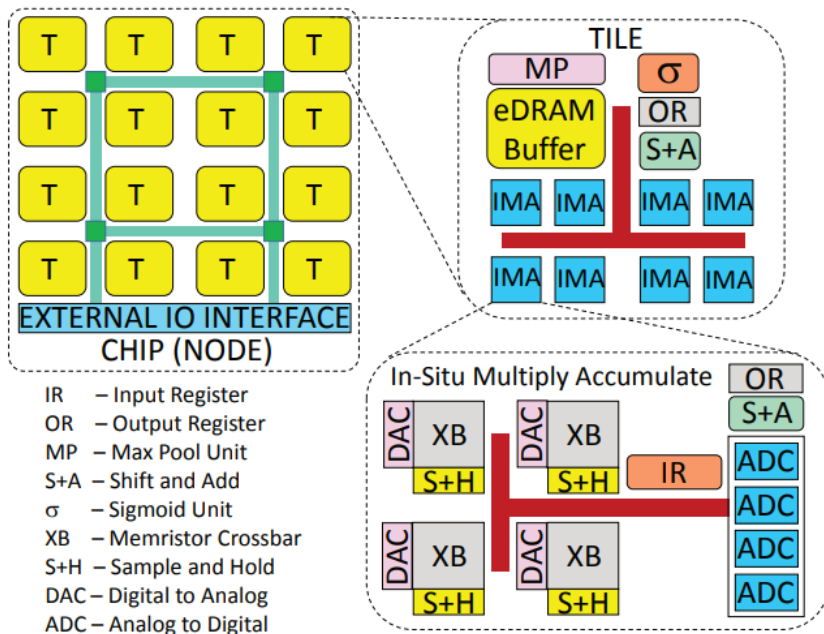


Figure 2.2: ISAAC based Hardware design([1])(Pictures taken from "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars Ali Shafiee , Anirban Nag , Naveen Muralimanohar and Rajeev Balasubramonian")

Figure 2.2 captures this hierarchy visually. The outer chip boundary shows a 4×4 array of tiles ("T"), each with its own eDRAM buffers and a cluster of IMAs. Zooming into a tile, you see the eDRAM buffer at the top feeding multiple IMAs through a shared bus, on the right side are the tile-level S+A, sigmoid (σ), and max pool units plus an output register that aggregates IMA results. Each IMA block (in the bottom right inset) then unfolds into four crossbar slices, each slice connected to four DACs (one per row), four S+H circuits, and four interleaved ADC lanes that digitize bitline currents. This in-situ compute fabric, combined with modest digital glue logic, allows ISAAC to perform massively parallel dot products right where the weights are stored, minimizing data movement and maximizing utilization of both analog and digital resources.

2.2 Implementation of Mixed-Bit Operations

Figure 2.3 shows reconfiguration logic for mixed bit MAC unit in 8bit ReRAM. ReRAM based accelerator implements 1–8 bit MACs using a deeply pipelined array of 1 bit "macro

cells,” each realizing an AND via two serial ReRAM devices, an inverter, and a 1 fF sampling capacitor[3]. This charge division topology reduces cell-to-cell variation upto 1%,

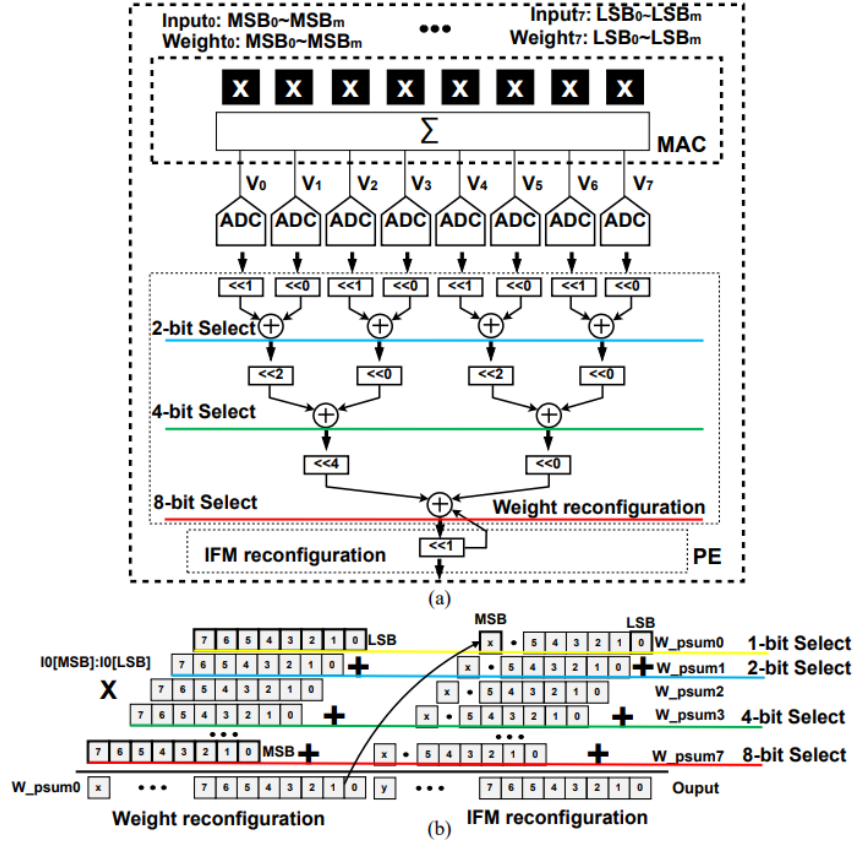


Figure 2.3: Bit-serial/bit-parallel reconfiguration logic for mixed-precision MAC in an 8×1 ReRAM column (from [1]).

enabling error free summation parallel rows. Groups of eight macro cells form bit-parallel columns: input activations are streamed one bit per cycle,[3] each column produces eight significance weighted analog partial sums, and a cascade of shift and add units dynamically reconfigures them into 1, 2, 4, or 8 bit outputs. A shared bit line bus drives ADCs.

2.3 Twin Range Quantization

Quantization reduces data precision to accelerate computations and conserve energy. In ReRAM based accelerators, ADCs consume significant power at high precision. Traditional uniform quantization applies the same bitwidth across all values, which is inefficient since most data are small while only a few values are large.

2.3.1 Twin Range Quantization Mechanism

[4]TRQ is our key innovation to eliminate redundant A/D operations in ReRAM based PIM accelerators, exploiting both the skewed distribution of crossbar outputs and the error resilience of deep networks. In conventional SAR ADCs, every conversion performs a full binary search over all K bits even though most analog samples lie near zero and contribute little new information wasting energy on unnecessary bit tests. TRQ breaks the conversion range into two intervals, a narrow “fine” range R_1 where outputs are dense and require high precision, and a broad “coarse” range R_2 where outputs are sparse and tolerate reduced precision.

Empirically, the distribution of bit line currents in a 128×128 crossbar is highly skewed the bulk of values cluster within a small fraction of the full dynamic range (R_1), while only a few “outliers” span the remainder (R_2). TRQ first checks, in a single extra “detection” step, whether the sampled voltage V_{hold} falls within R_1 ; if so, the ADC proceeds with a shortened binary search using only N_{R_1} bits (“early-bird” strategy), completing in $N_{R_1} + 1$ comparisons with no loss in accuracy. Otherwise, it transitions to a coarser binary search over N_{R_2} bits (“early-stop” strategy), stopping after $N_{R_2} + 1$ comparisons even though fine grained lower bits remain undetermined thus bounding the worst case conversion cost while allowing a controlled drop in precision.

Formally, TRQ implements a piecewise uniform quantizer:

$$T(x) = \begin{cases} Q_{N_{R_1}}(x; \Delta_1), & x \leq \theta, \\ Q_{N_{R_2}}(x; \Delta_2), & x > \theta, \end{cases}$$

where θ is the boundary between the two ranges, and Δ_1, Δ_2 are the corresponding step sizes (chosen so that the two quantization grids align on the full precision grid). Here $Q_K(x; \Delta)$ denotes K -bit uniform quantization with step Δ . By selecting $\Delta_2 = 2^M \Delta_1$ for some integer M , the coarse range outputs can be decoded back to full resolution by a single left shift by M bits, making hardware decoding trivial.

[5]The coding format itself embeds the MSB to signal range membership (“0” for R_1 , “1” for R_2), followed by N_{R_x} bits of unsigned quantization within that range. In hardware, only a small modification of the SAR control logic is required an extra comparator for early detection, a multiplexer to select the fine or coarse search step size, and a flag to invoke early stop. On the digital side, the existing Shift and Add (S + A) units gain a simple shift-control input: when the MSB=1, the partial sum is shifted left by M before accumulation; otherwise, it is added directly. Crucially, no changes to the analog DAC or comparator network are needed, and all of the TRQ parameters (N_{R_1} , N_{R_2} , M , θ) are stored in a small configuration register adjacent to each ADC.

Figure 2.4 shows the overall architecture of TRQ. In summary, TRQ is an algorithmic

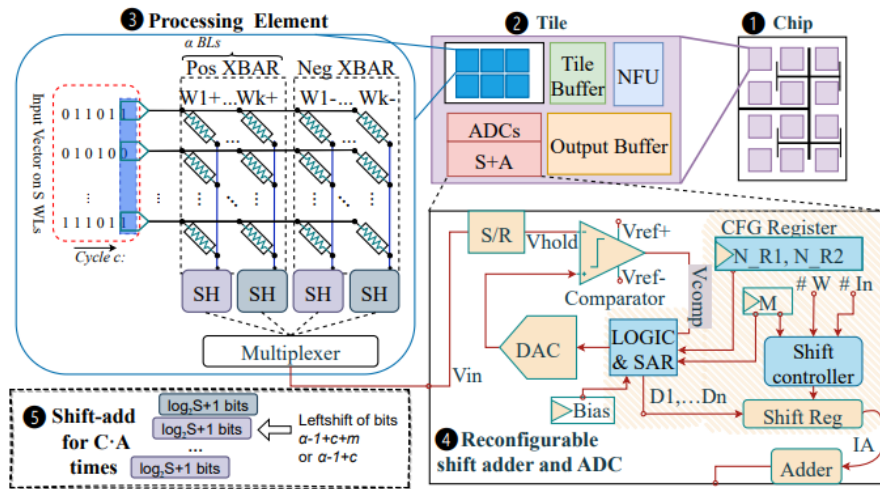


Figure 2.4: Overall architecture of Twin Range Quantization([1])(Pictures taken from "Algorithm-hardware co-design for Energy-Efficient A/D conversion in ReRAM-based accelerators Chenguang Zhang , Zhihang Yuan")

insight skewed crossbar distributions and DNN resilience with a minimal SAR logic tweak and lightweight S + A support to slash peripheral energy by over 60 % without any analog redesign or network retraining.

Chapter 3

Methdology

To validate the proposed accelerator in a real world context, I constructed a behavioral model in LTSpice that captures the key analog and digital building blocks of our in memory compute engine. The model includes simplified transistor-level representations of a simple 2×2 crossbar array, peripheral drivers and a behavioral SAR ADC interface. Transient and DC sweeps were used to verify read/write margins, settling times and energy per operation, ensuring that the behavioral abstractions faithfully reflect practical device limitations.

Later, to evaluate the design as a CNN based hardware accelerator, I choosed a software called DNN+Neurosim. NeuroSim is a cycle accurate, circuit level macro model implemented in C++ with a PyTorch wrapper that bridges device and circuit level parameters to full chip performance for DNN accelerators. It models synaptic arrays (SRAM or eNVM), peripheral circuits (decoders, switch matrices, ADCs) and global interconnect to report area, latency, dynamic energy and leakage power for each layer of any CNN topology.

In my project, I used DNN+Neurosim to quantify how our ReRAM crossbar and SAR-ADC with TRQ impact end-to-end inference throughput and energy efficiency. This framework automatically floorplans your network onto tiles and PEs, applies realistic non idealities (sneak paths, device variation, ADC quantization), and outputs per layer and total PPA . It is compatable only with Linux. This allows rapid, design space exploration of memory technology choices and peripheral design without full silicon fabrication.

3.1 Creating memrisor array in LTSpice

As shown in Figure 3.1, the entire 2×2 memristive crossbar array is implemented in LTSpice by wiring four instances of the Yakopcic behavioral memristor model into a neat grid of two horizontal word-lines (WL1, WL2) and two vertical bit-lines (BL1, BL2). Voltage sources V1 and V2 drive the top electrodes (TE) of WL1 and WL2, respectively, each splitting cleanly into two junctions so that M11/M12 and M21/M22 see identical drive sig-

nals with no voltage-drop or net-naming errors. The bottom electrodes (BE) form the bit-lines, each terminating in a 10 pull-down resistor whose sense node is explicitly exposed (`xsv_node1...xsv_node4`). LTSpice does not include a built-in memristor element, so I import the widely used Yakopcic memristor symbol via its accompanying `.asy` and `.subckt` files. This symbol encapsulates all threshold parameters, state-variable equations, and behavioral sources into a single, ready to use subcircuit, eliminating manual SPICE edits. By instantiating four subcircuits (M11–M22) at the WL/BL intersections, I seamlessly integrate accurate memristor dynamics without additional coding overhead.

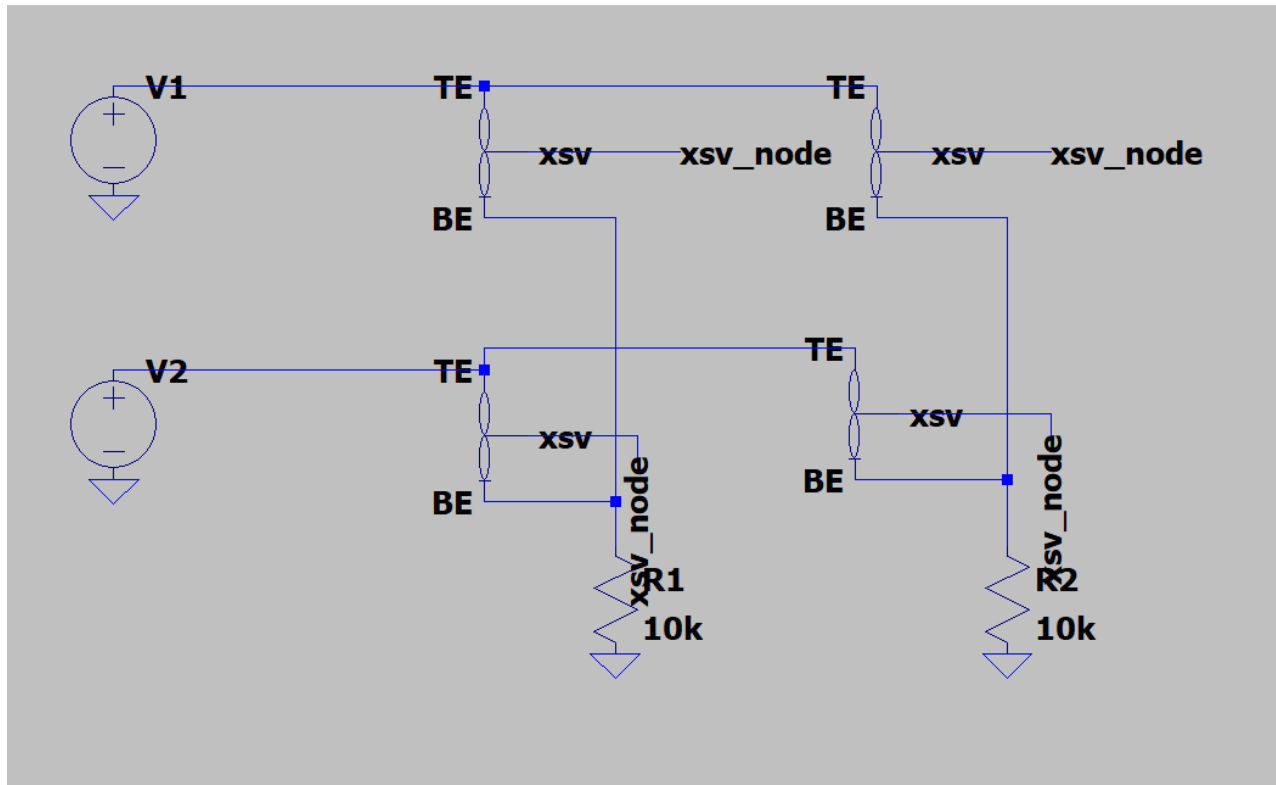


Figure 3.1: 2*2 Memristor crossbar Array

Into this wiring framework I imported the Yakopcic memristor symbol a compact two terminal icon denoting the TE/BE pins and its internal state-dependent behavioral block. Four subcircuits (M11–M22) occupy the intersections: for instance, M11’s TE connects to WL1 and its BE to BL1, with its internal “xsv” node routed through R1 to `xsv_node1`. By exposing that node rather than hiding it, I can monitor the instantaneous current and voltage drop simultaneously during a single transient run, without resorting to hidden-node measures or post-processing hacks.

The 10 pull down resistors both define a ground reference for unselected lines (revealing sneak-path currents) and emulate a simple sense-amplifier input impedance. Under low-amplitude read pulses (± 0.2) on WL1/WL2, the current through the targeted device produces a directly observable voltage at the corresponding `xsv_nodeX`, while unintended sneak currents create characteristic voltage deviations on the other nodes making non

idealities immediately apparent in the waveform viewer.

For dynamic evaluation, I apply staggered PULSE sources on V1 and V2 (Figure 3.1, inset): V1 is defined as

```
PULSE(0 1 0 1n 1n 1u 2u)
```

and V2 is identical but delayed by 1. This alternation programs M11/M12 in the first half cycle and M21/M22 in the next, before both lines rest low. Because the Yakopcic model only updates its internal state above its threshold, I can cleanly separate write (± 1) and read (± 0.2) regimes in the same `.tran 0.1u 10u` run. The resulting traces at `xsv_node1-xsv_node4` show classic pinched I-V hysteresis loops, ON and OFF state voltage drops across the resistors, and sneak path glitches on unselected lines all captured in one self contained LTSpice deck that validates device behavior and highlights crossbar challenges.

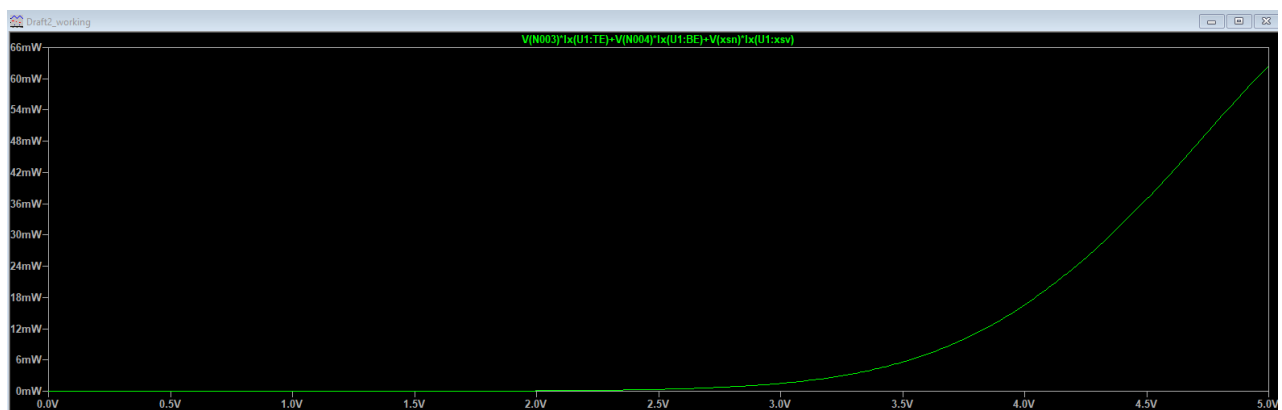


Figure 3.2: Power vs. Applied Voltage for the Memristor

As shown in Figure 3.2, At low voltages (0 V to 2.5 V), the memristor remains in its high-resistance (“off”) state and conducts only a tiny leakage current. Because power is the product of voltage and current, and the current here is essentially negligible, the power curve hugs the zero milliwatt line. This flat region reflects the device’s non-volatile retention with almost no energy cost in standby.

Around 2.5V you observe a pronounced “knee” in the trace. This marks the threshold where ionic drift or filament formation begins inside the memristive layer. As oxygen vacancies or metal ions start to migrate under the stronger electric field, the device’s conductance rapidly increases. The resulting rise in current makes the power trace peel away from zero, signaling the onset of the low-resistance state.

Beyond roughly 3 V, the curve shoots upward in a convex, nearly exponential fashion. In this regime, the current often follows field-enhanced mechanisms (for example, Schottky emission or Poole–Frenkel conduction), which scale exponentially with voltage. Since power multiplies that growing current by the applied voltage, you get a sharp, super linear increase reaching on the order of tens of milliwatts by 5 V.

This three-stage behavior flat baseline, sharp knee, and steep exponential rise has important practical implications. It means the memristor consumes virtually no power when idle, switches decisively once you exceed its threshold, and then exhibits high conductance (and correspondingly higher power) for strong inputs. In dense crossbar arrays, this ensures energy-efficient standby, clear read/write margins, and a need to manage heat dissipation during heavy programming or inference operations.

3.2 Twin Range Quantization and Behavioral SAR ADC for Low Power In Memory Computing

In large scale IMAs particularly those based on resistive RAM (ReRAM) crossbars the energy and area consumed by high resolution analog to digital converters (ADCs) can dominate the overall chip budget. A conventional ADC designed to deliver uniform N bit precision across its full input range must resolve the entire dynamic window with the finest step size $\Delta = V_{FS}/2^N$, driving power and silicon area to prohibitive levels when scaled to dozens or hundreds of parallel channels. TRQ addresses this challenge by partitioning each column’s analog sum into two complementary ranges: a *fine* range for low-magnitude signals and a *coarse* range for high magnitude signals.

Concretely, TRQ defines a programmable threshold V_{th} (in our implementation, $V_{th} = 1$). When the input analog sum V_{in} falls below V_{th} , it is routed to a high resolution quantization path that uses a step size $\Delta_{fine} \ll V_{th}/2^{N_{fine}}$, producing the least significant bits (LSBs). Conversely, when $V_{in} \geq V_{th}$, the signal takes a low resolution path with a larger step size $\Delta_{coarse} = (V_{FS} - V_{th})/2^{N_{coarse}}$, efficiently yielding the most significant bits (MSBs). The final digital word is then reconstructed by digitally concatenating the coarse MSBs with the fine LSBs.

Because ADC power, SAR, or pipeline architectures grows roughly exponentially with resolution, splitting the conversion into two narrower converters can reduce average power consumption by up to 60%. Dynamic threshold adjustment, based on measured signal histograms, ensures that the fine path is invoked only when the marginal benefit of the extra bits outweighs its energy cost. The TRQ scheme thus delivers near-full precision over the entire range while drastically cutting the energy and area overhead of high resolution single range converters.

3.3 Behavioral SAR ADC

SAR ADC converts an analog input V_{in} into an N -bit code by binary search: at each cycle it sets a trial bit b_k , generates V_{DAC} via a capacitive or resistor-string DAC, and compares V_{in} to V_{DAC} . To speed up architectural exploration, a *behavioral* SAR ADC replaces these

steps with a single expression:

$$\text{Code} = \begin{cases} \lfloor \frac{V_{\text{in}}}{\Delta_{\text{fine}}} + 0.5 \rfloor \Delta_{\text{fine}}, & V_{\text{in}} < V_{\text{th}}, \\ \lfloor \frac{V_{\text{in}}}{\Delta_{\text{coarse}}} + 0.5 \rfloor \Delta_{\text{coarse}}, & V_{\text{in}} \geq V_{\text{th}}. \end{cases}$$

Figure 3.3 shows the behavioral SAR ADC used for the TRQ application, a DC sweep `.dc V1 0 2 0.01` drives `in` from 0 V to 2 V. The behavioral source is defined as:

```
V = { if ( V(in) < 1,
        floor(V(in)/0.1 + 0.5)*0.1,
        floor(V(in)/0.5 + 0.5)*0.5 ) }
```

Here $V_{\text{th}} = 1$ V splits the range into a fine 0.1 V step (≈ 4 bits) and a coarse 0.5 V step (≈ 2 bits). The resulting staircase (10 steps below 1 V, 3 above) yields 6 bits effective resolution while invoking the power-hungry fine DAC only half the time, cutting both power and area compared to a uniform converter.

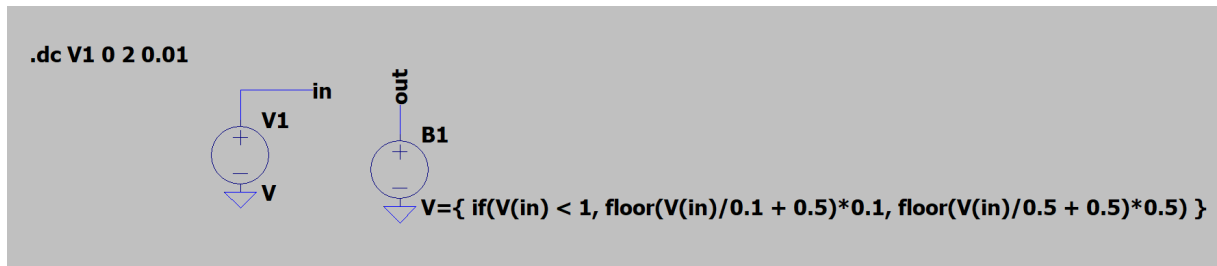


Figure 3.3: TRQ transfer curve from LTSpice behavioral SAR ADC.

3.3.1 Total Simulation diagram of the 2×2 memristor array with Twin Range Quantization

Figure 3.4 shows the complete LTSpice schematic of our 2×2 memristor crossbar with twin-range quantization. Four memristive elements are arranged at the intersections of two wordlines (V_1, V_2) and two bitlines (row “bottom” nodes). A shared sense resistor and a behavioral quantizer convert the selected cell’s analog conductance into a digital level in two precision ranges.

3.3.2 Overall Working

Each memristor is modeled by a voltage-controlled switch (SW) with $R_{\text{on}} = 10 \Omega$, $R_{\text{off}} = 1 \text{ G}\Omega$, and threshold $V_t = 0.5$ V. A DC sweep applied to the two wordline sources biases all four devices in parallel, but only the one selected by the switching network contributes current to the sense resistor $R_{\text{sense}} = 1 \text{ G}\Omega$. The resulting voltage V_{in} at the sense node is fed into a behavioral source that implements twin-range quantization:

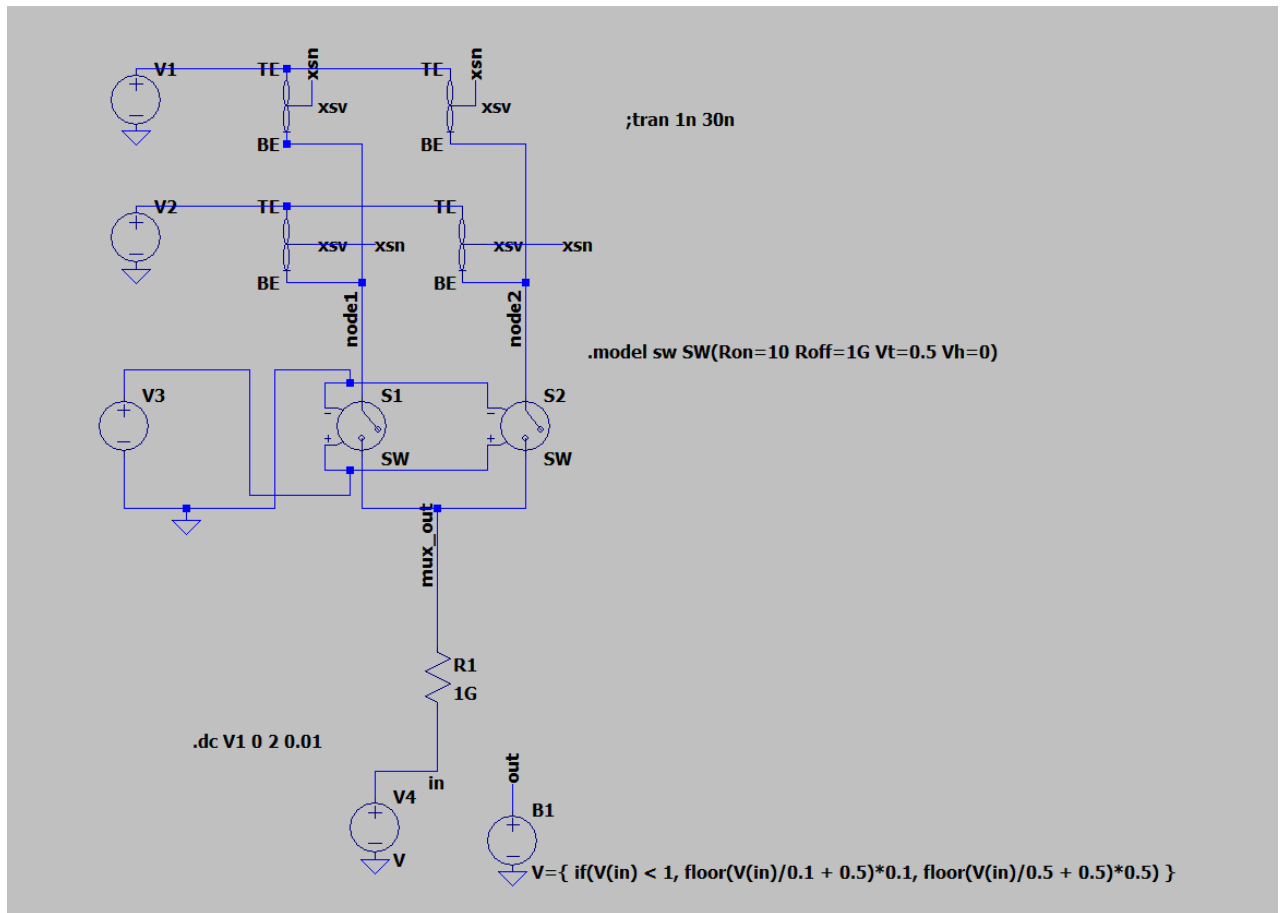


Figure 3.4: Memristor circuit with ADC quantization

Switch Network Detail To isolate a single cell, we use a two stage multiplexer built from four SPICE switches (`S_row_col`). In the first stage, two switches decode the row address (via control voltage V_r), connecting one of the two wordlines to an intermediate node. In the second stage, two more switches decode the column address (V_c), steering that intermediate node onto the common sense node (`mux_out`). Each switch uses the same `SW` model: when its gate voltage exceeds V_t , it closes to R_{on} ; otherwise it remains at R_{off} , ensuring negligible leakage from unselected cells. Transient (`.tran 1n 30n`) and DC (`.dc V1 0 2 0.01`) runs confirm proper timing, clean isolation, and correct quantization behavior across all four devices.

3.4 Integrating DNN+ NeuroSim for hardware simulation

DNN+NeuroSim Framework V1.4 is a comprehensive guide for an end-to-end simulator that couples DNN models with hardware macro models to predict on chip inference performance. Developed in C++ with a PyTorch wrapper, the framework supports both inference (V1.0–V1.4) and on chip training (V2.0–V2.1) on accelerators built around near

memory or in memory computing architectures. It accommodates a wide variety of synaptic device technologies including SRAM and emerging non volatile memories (RRAM, PCM, STT-MRAM, FeFET) and maps user defined network topologies (e.g., VGG-8, DenseNet-40, ResNet-18) through a Python interface into hierarchical hardware models spanning device, circuit, tile, chip, and algorithm levels.

At its core, DNN+NeuroSim works by importing pre trained network weights and activation traces during a PyTorch inference run, then forwarding these real time traces to NeuroSim’s circuit level models. These macro models estimate area, latency, dynamic energy, leakage power, and even inference accuracy by simulating synaptic array architectures (1T1R, crossbar, pseudo-crossbar), peripheral circuits (ADCs, decoders, switch matrices), and interconnects (H-Tree, XY-bus). Users can rapidly explore “what-if” scenarios varying technology nodes (down to 1 nm), cell precisions (1–multi-bit), array sizes, and parallelism modes without costly VLSI prototyping.

By unifying algorithmic flexibility with detailed hardware estimates, this tool empowers circuit and architecture designers to co optimize DNN models and emerging memory technologies, dramatically accelerating early stage design space exploration and guiding efficient, real world accelerator implementations.

3.4.1 Environment setup and code checkout

I began by preparing a reproducible development environment to ensure that all subsequent compilations and inference runs would execute without conflict. First, I retrieved the NeuroSim source code from GitHub by opening a terminal and running:

```
git clone https://github.com/neurosim/NeuroSim.git
cd NeuroSim
```

This command created a local copy of the latest `main` branch, and positioning myself in the `NeuroSim` directory placed me at the repository root where the build scripts, source files, and documentation reside. To target a specific release, I listed available tags with `git tag` and checked out the desired tag via `git checkout <tag_name>`.

Next, I verified that my C++ toolchain met the project requirements. NeuroSim relies on C++17, so I confirmed that `g++` (version 7.5 or higher) and GNU `make` (version 4.1 or higher) were installed. On my Ubuntu system, I ran the command :Figure 3.5 depicts the successful compilation of the c++ files. The console output shows that every source

```
g++ -c -fopenmp -O3 -std=c++0x -w Param.cpp -o Param.o
g++ -fopenmp -O3 -std=c++0x -w Adder.o AdderTree.o BitShifter.o Buffer.o Bus.o Chip.o Comparator.o CurrentSenseAmp.o DFF.o DeMux.o DecoderDriver.o Functio
nUnit.o HTree.o LevelShifter.o MaxPooling.o MultilevelSAEncoder.o MultilevelSenseAmp.o Mux.o NewMux.o NewSwitchMatrix.o Param.o Precharger.o ProcessingUnit.
.o ReadCircuit.o RowDecoder.o SRAMWriteDriver.o SarADC.o SenseAmp.o ShiftAdd.o Sigmoid.o SramNewSA.o SubArray.o SwitchMatrix.o Technology.o Tile.o VoltageSen
seAmp.o WLDecoderOutput.o WLNewDecoderDriver.o XYBus.o formula.o main.o -o main
(base) root@DESKTOP-GSOTK7B:/mnt/d/DNN_NeuroSim_V1.4-main/DNN_NeuroSim_V1.4-main/Inference_pytorch/NeuroSIM# |
```

Figure 3.5: successful compilation of NeuroSim’s C++ source files

file in the NeuroSim framework (e.g. Param.cpp, AdderTree.o, SwitchMatrix.o, etc.) was successfully compiled into object (.o) files with the `-fopenmp -O3 -std=c++0x -w` flags, and then all of those object files were linked together into a single executable named `main`. I can infer that the build completed without issues, OpenMP support and C++17 compatibility are in place, and the NeuroSim binary is now ready for use.

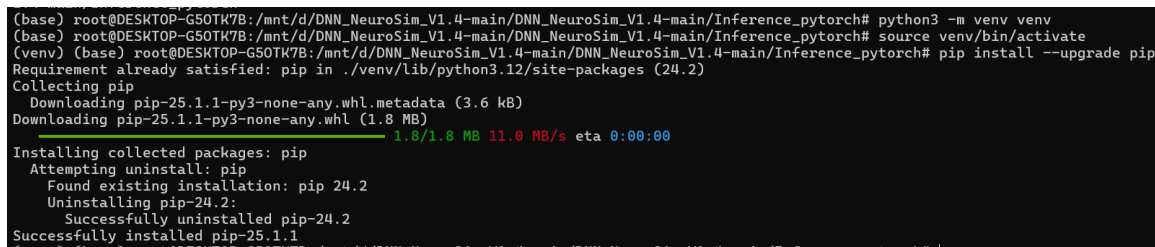
Since the Python wrapper drives network definitions and inference through PyTorch, I set up a dedicated virtual environment:

```
python3 -m venv venv
source venv/bin/activate
```

Within this environment, I upgraded `pip` and installed all Python dependencies with :

```
pip install --upgrade pip
pip install -r requirements.txt
```

as shown in Figure 3.6.



```
(base) root@DESKTOP-G50TK7B: /mnt/d/DNN_NeuroSim_V1.4-main/DNN_NeuroSim_V1.4-main/Inference_pytorch# python3 -m venv venv
(base) root@DESKTOP-G50TK7B: /mnt/d/DNN_NeuroSim_V1.4-main/DNN_NeuroSim_V1.4-main/Inference_pytorch# source venv/bin/activate
(venv) (base) root@DESKTOP-G50TK7B: /mnt/d/DNN_NeuroSim_V1.4-main/DNN_NeuroSim_V1.4-main/Inference_pytorch# pip install --upgrade pip
Requirement already satisfied: pip in ./venv/lib/python3.12/site-packages (24.2)
Collecting pip
  Downloading pip-25.1.1-py3-none-any.whl.metadata (3.6 kB)
  Downloading pip-25.1.1-py3-none-any.whl (1.8 MB)
    1.8/1.8 MB 11.0 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 24.2
    Uninstalling pip-24.2:
      Successfully uninstalled pip-24.2
  Successfully installed pip-25.1.1
```

Figure 3.6: Upgraded pip installed in a virtual enviornmrnt

This guaranteed consistent versions of PyTorch (1.7), NumPy, and other utilities. For GPU acceleration, I installed the matching `torch` build for my CUDA toolkit.

- `src/Param.cpp`

- Defines global simulation parameters: array dimensions (`numRowSubArray`, `numColSubArray`), memory cell type (`memcelltype`), bit precision (`cellBit`), access mode (`operationmode`), pipeline depth (`speedUpDegree`), and peripheral feature flags (level shifter, ADC).
- Central location for tuning how the device-, circuit-, and architecture-level models behave.

- `src/Technology.cpp`

- Specifies device level characteristics: R_{on} , R_{off} , SET/RESET voltages, write/read energy, and nonlinearity coefficients.
- Contains functions to translate measured ReRAM (or other eNVM) parameters into internal macro model values.

- `src/SwitchMatrix.cpp`
 - Implements row/column selection and multiplexing circuitry via transmission-gate switch matrices.
 - Encodes the control logic for parallel analog reads versus sequential digital reads and the instantiation of level shifters.
- `python/`
 - `inference.py`: PyTorch based driver that parses command line arguments, loads network definitions from CSV, and invokes the compiled NeuroSim executable.
 - `NetWork.csv`: Defines layer types, feature-map dimensions, kernel sizes, and mixed bit precision settings per layer.
 - Utility scripts for post-processing results, plotting area/energy breakdowns, and running batch sweeps.
- `Makefile`
 - Contains build rules for compiling all C++ source files with flags (`-std=c++17 -fopenmp -O3`) and linking into the `main` executable.
 - Provides convenience targets (`clean`, `run`) to manage recompilation and automated tests.

Role of the Python Wrapper The Python wrapper in the `python/` directory bridges high-level DNN definitions and the low-level C++ simulator. It allows users to specify network topologies, precision settings, and hardware configurations via CSV and command-line arguments, then orchestrates inference runs through PyTorch traces. This abstraction layer simplifies design space exploration enabling rapid prototyping of new architectures and quantization schemes without requiring manual edits to the C++ code base or recompilation for each experiment. Having this structure in mind allowed me to pinpoint where I would later apply modifications for mixed bit precision, twin range quantization, and pipeline configuration. With the environment fully configured, I proceeded to implement the ReRAM specific adaptations and performance optimizations detailed in the following sections.

3.4.2 Adapting the DNN+NeuroSim framework for ReRAM based simulation

DNN+NeuroSim is an open source framework originally tailored for SRAM based PIM. To explore ReRAM’s higher density and analog behavior, we implement two key enhance-

ments: (1) mixed bit ReRAM cells, allowing non uniform precision across layers, and (2) Twin Range Quantization in the SAR ADC, performing separate coarse and fine conversions to improve dynamic range and accuracy. This document details the required code modifications, their rationale, and their impact on simulation behavior.

In `Param.h`, we redefine the memory cell type from the default SRAM (`memcelltype=0`) to ReRAM (`memcelltype=1`):

```
int memcelltype;           // 0=SRAM, 1=ReRAM
bool mixedBitQuantization; // enable per-layer bit variation
int cellBit;               // bits per ReRAM cell
```

Corresponding defaults in `Param.cpp`:

```
memcelltype = 1;
mixedBitQuantization = true;
cellBit = 4; // e.g., 4-bit ReRAM
resistanceOn  = 1e3; //
resistanceOff = 1e6; //
```

I have carried out a detailed comparison between the original 6T SRAM model and the newly added multi-bit ReRAM model, and the numerical differences arise from both the device-level parameters and the required peripheral circuits. At the cell level, a typical 6T SRAM bitcell occupies around $120F^2$ (where F is the technology feature size) and stores exactly 1 bit, whereas a 1T-1R ReRAM cell can be as small as $4F^2$ per device and, with my mixed-bit scheme, can store up to 4 bits in a single cell by exploiting multiple conductance levels. Consequently, for the same logical capacity, the ReRAM array area drops by roughly a factor of four, directly reducing the array footprint reported by NeuroSim's floorplan module.

```
using namespace std;
Param::Param() {
    /***** user defined design options and parameters *****/
    operationmode = 2; // 1: conventionalSequential (Use several multi-bit RRAM as one synapse)
                      // 2: conventionalParallel (Use several multi-bit RRAM)
    memcelltype = 1; // 1: cell.memCellType = Type::RRAM
                   // 0: cell.memCellType = Type::SRAM
                   // 3: cell.memCellType = Type::FeFET
    accesstype = 4; // 1: cell.accessType = CMOS_access
                  // 2: cell.accessType = BJT_access
                  // 3: cell.accessType = diode_access
                  // 4: cell.accessType = none_access (Crossbar Array)
```

Figure 3.7: A snippet of changes made in the `param.cpp`

Figures 3.7 and 3.8 show the changes we made in `Param.cpp`. The leakage power also shifts dramatically. In SRAM, each latch draws static current from both the pull-up and access transistors (on the order of tens of nanoamperes per cell), giving a leakage power of several microwatts per megabit at room temperature. In contrast, ReRAM cells are

```

// 1.4 update: handle the exception for conventional sequential case
// 1.4 update 230615
if ((conventionalSequential == 1 || conventionalSequential == 3 || conventionalSequential == 5 ) && (memcelltype > 0))
{
    numColMuxed=numColPerSynapse;
}

levelOutput = 32; // # of levels of the multilevelSenseAmp output, should be in 2^N forms; e.g. 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864, 134217728, 268435456, 536870912, 1073741824, 2147483648, 4294967296, 8589934592, 17179869184, 34359738368, 68719476736, 137438953472, 274877906944, 549755813888, 1099511627776, 2199023255552, 4398046511104, 8796093022208, 17592186044416, 35184372088832, 70368744177664, 140737488355328, 281474976710656, 562949953421312, 1125899906842624, 2251799813685248, 4503599627370496, 9007199254740992, 18014398509481984, 36028797018963968, 72057594037927936, 144115188075855872, 288230376151711744, 576460752303423488, 1152921504606846976, 2305843009213693952, 4611686018427387904, 9223372036854775808, 18446744073709551616, 36893488147419103232, 73786976294838206464, 147573952589676412928, 295147905179352825856, 590295810358705651712, 1180591620717411303424, 2361183241434822606848, 4722366482869645213696, 9444732965739290427392, 18889465931478580854784, 37778931862957161709568, 75557863725914323419136, 151115727451828646838272, 302231454903657293676544, 604462909807314587353088, 1208925819614629174706176, 2417851639229258349412352, 4835703278458516698824704, 9671406556917033397649408, 19342813113834066795298816, 38685626227668133590597632, 77371252455336267181195264, 154742504910672534362390528, 309485009821345068724781056, 618970019642690137449562112, 1237940039285380274899124224, 2475880078570760549798248448, 4951760157141521099596496896, 9903520314283042199192993792, 19807040628566084398385987584, 39614081257132168796771975168, 79228162514264337593543950336, 158456325028528675187087900672, 316912650057057350374175801344, 633825300114114700748351602688, 1267650600228229401496703205376, 2535301200456458802993406410752, 5070602400912917605986812821504, 10141204801825835211973625643008, 20282409603651670423947251286016, 40564819207303340847894502572032, 81129638414606681695789005144064, 162259276829213363391578010288128, 324518553658426726783156020576256, 649037107316853453566312041152512, 1298074214633706907132624082305024, 2596148429267413814265248164610048, 5192296858534827628530496329220096, 10384593717069655257060992658440192, 20769187434139310514121985316880384, 41538374868278621028243970633760768, 83076749736557242056487941267521536, 166153499473114484112975882535043072, 332306998946228968225951765070086144, 664613997892457936451903530140172288, 1329227995784915872903807060280344576, 2658455991569831745807614120560689152, 5316911983139663491615228241121378304, 10633823966279326983230456482242756608, 21267647932558653966460912964485513216, 42535295865117307932921825928971026432, 85070591730234615865843651857942052864, 170141183460469231731687303715884105728, 340282366920938463463374607431768211456, 680564733841876926926749214863536422912, 1361129467683753853853498429727072845824, 2722258935367507707706996859454145691648, 5444517870735015415413993718908291383296, 10889035741470030830827987437816582766592, 21778071482940061661655974875633165533184, 43556142965880123323311949751266331066368, 87112285931760246646623899502532662132736, 174224571863520493293247799005065324265472, 348449143727040986586495598010130648530944, 696898287454081973172991196020261297061888, 1393796574908163946345982392040522594123776, 2787593149816327892691964784081045188247552, 5575186299632655785383929568162090376495104, 11150372599265311570767859136324180752990208, 22300745198530623141535718272648361505980416, 44601490397061246283071436545296723011960832, 89202980794122492566142873090593446023921664, 178405961588244985132285746181186892047843328, 356811923176489970264571492362373784095686656, 713623846352979940529142984724747568191373312, 1427247692705959881058285969449495136382746624, 2854495385411919762116571938898990272765493248, 5708990770823839524233143877797980545530986496, 11417981541647679048466287755595961091061972992, 22835963083295358096932575511191922182123945984, 45671926166590716193865151022383844364247891968, 91343852333181432387730302044767688728495783936, 182687704666362864775460604089535377456991567872, 365375409332725729550921208179070754913983135744, 730750818665451459101842416358141509827966271488, 1461501637330902918203684832716283019655932542976, 2923003274661805836407369665432566039311865085952, 5846006549323611672814739330865132078623730171904, 11692013098647223345629478661730264157247460343808, 23384026197294446691258957323460528314494920687616, 46768052394588893382517914646921056628989841375232, 93536104789177786765035829293842113257979682750464, 187072209578355573530071658587684226515959365500928, 374144419156711147060143317175368453031918731001856, 748288838313422294120286634350736906063837462003712, 1496577676626844588240573268701473812127674924007424, 2993155353253689176481146537402947624255349848014848, 5986310706507378352962293074805895248510699696029696, 11972621413014756705924586149611790497021399392059392, 23945242826029513411849172299223580994042798784118784, 47890485652059026823698344598447161988085597568237568, 95780971304118053647396689196894323976171195136475136, 191561942608236107294793378393788647952342390272950272, 383123885216472214589586756787577295904684780545900544, 766247770432944429179173513575154591809369561091801088, 1532495540865888858358347027150309183618739122183602176, 3064991081731777716716694054300618367237478244367204352, 6129982163463555433433388108601236734474956488734408704, 12259964326927110866866776217202473468949912977468817408, 24519928653854221733733552434404946937899825954937634816, 49039857307708443467467104868809893875799651909875269632, 98079714615416886934934209737619787751599303819750539264, 196159429230833773869868419475239575503198607639501078528, 392318858461667547739736838950479151006397215279002157056, 784637716923335095479473677900958302012794430558004314112, 1569275433846670190958947355801916604025588861116008628224, 3138550867693340381917894711603833208051177722232017256448, 6277101735386680763835789423207666416102355444464034512896, 12554203470773361527671578846415332832204710888928069025792, 25108406941546723055343157692830665664409421777856138051584, 50216813883093446110686315385661331328818843555712276103168, 100433627766186892221372630771322662657637687111424552206336, 200867255532373784442745261542645325315275374222849104412672, 401734511064747568885490523085290650630550748445698208825344, 803469022129495137770981046170581301261101496891396417650688, 1606938044258990275541962092341162602522202993782792835301376, 3213876088517980551083924184682325205044405987565585670602752, 6427752177035961102167848369364650410088811975131171341205504, 12855504354071922204335696738729300820177623950262342682411008, 25711008708143844408671393477458601640355247900524685364822016, 51422017416287688817342786954917203280710495801049370729644032, 102844034832575377634685573909834406561420991602098741459288064, 205688069665150755269371147819668813122841983204197482918576128, 411376139330301510538742295639337626245683966408394965837152256, 822752278660603021077484591278675252491367932816789931674304512, 1645504557321206042154969182557350504982735865633579863348609024, 3291009114642412084309938365114701009965471731267159726697218048, 6582018229284824168619876730229402019930943462534319453394436096, 13164036458569648337239753460458804039861886925068638906788872192, 26328072917139296674479506920917608079723773850137277813577744384, 52656145834278593348959013841835216159447547700274555627155488768, 105312291668557186697918027683670432318895095400549111254310977536, 210624583337114373395836055367340864637790190801098222508621955072, 421249166674228746791672110734681729275580381602196445017243910144, 842498333348457493583344221469363458551160763204392890034487820288, 1684996666696914987166688442938726917102321526408785780068975640576, 3369993333393829974333376885877453834204643052817571560137951281152, 6739986666787659948666753771754907668409286105635143120275902562304, 13479973333575319897333507543509815336818572211270286240551805124608, 26959946667150639794667015087019630673637144422540572481103610249216, 53919893334301279589334030174039261347274288845081144962207220498432, 107839786668602559178668060348078522694548577690162289924414440996864, 215679573337205118357336120696157045389097155380324579848828881993728, 431359146674410236714672241392314090778194310760649159697657763987456, 862718293348820473429344482784628181556388621521298319395315527974912, 1725436586697640946858688965569256363112777243042596638790631055949824, 3450873173395281893717377931138512726225554486085193277581262111899648, 6901746346790563787434755862277025452451108972170386555162524223799296, 13803492693581127574869511724554050904902217944340773110325048447598592, 27606985387162255149739023449108101809804435888681546220650096895197184, 55213970774324510299478046898216203619608871777363092441300193790394368, 110427941548649020598956093796432407239217743554726184882600387580788736, 220855883097298041197912187592864814478435487109452369765200775161577472, 441711766194596082395824375185729628956870974218904739530401550323154944, 883423532389192164791648750371459257913741948437809479060803100646309888, 1766847064778384329583297500742918515827483896875618958121606201292619776, 3533694129556768659166595001485837031654967793751237916243212402585239552, 7067388259113537318333190002971674063309935587502475832486424805170479104, 14134776518227074636666380005943348126619871175004951664972849610340958208, 28269553036454149273332760011886696253239742350009903329945699220681916416, 56539106072908298546665520023773392506479484700019806659891398441363832832, 113078212145816597093331040047546785012958969400039613319782796882727665664, 226156424291633194186662080095093570025917938800079226639565593765455331328, 452312848583266388373324160190187140051835877600158453279131187530910662656, 904625697166532776746648320380374280103671755200316906558262375061821325312, 1809251394333065553493296640760748560207343510400633813116524750123642650624, 3618502788666131106986593281521497120414687020801267626233049500247285301248, 7237005577332262213973186563042994240829374041602535252466099000494570602496, 14474011154664524427946373126085988481658748083205070504932198000989141204992, 28948022309329048855892746252171976963317496166410141009864396001978282409984, 57896044618658097711785492504343953926634992332820282019728792003956564819968, 115792089237316195423570985008687907853269984665640564039457584007913129639936, 231584178474632390847141970017375815706539969331281128078915168015826259279872, 463168356949264781694283940034751631413079938662562256157830336031652518559744, 926336713898529563388567880069503262826159877325124512315660672063305037119488, 1852673427797059126777135760139006525652319754650249024631321344126610074238976, 3705346855594118253554271520278013051304639509300498049262642688253220148477952, 7410693711188236507108543040556026102609279018600996098525285376506440296955904, 14821387422376473014217086081112052205218558037201992197050570753012880593911808, 29642774844752946028434172162224104410437116074403984394101141506025761187823616, 59285549689505892056868344324448208820874232148807968788202283012051522375647232, 118571099379011784113736688648896417641748464297615937576404566024103044751294464, 237142198758023568227473377297792835283496928595231875152809132048206089502588928, 474284397516047136454946754595585670566993857190463750305618264096412179005177856, 948568795032094272909893509191171341133987714380927500611236528192824358010355712, 1897137590064188545819787018382342682267975428761855001222473056385648716020711424, 3794275180128377091639574036764685364535950857523710002444946112771297432041422848, 7588550360256754183279148073529370729071901715047420004889892225542594864082845696, 15177100720513508366558296147058741458143803430094840009779784451085189728165691392, 30354201441027016733116592294117482916287606860189680019559568902170379456331382784, 60708402882054033466233184588234965832575213720379360039119137804340758912662765568, 121416805764108066932466369176469931665150427440758720078
```

3.4.3 Configuring Twin-Range Quantization and Calibrating ReRAM Parameters

Implementing Twin-Range Quantization

I first opened `src/SwitchMatrix.cpp` and located the existing quantization routine, which used a single fixed step. In the 1.4 update, several key changes were applied to `SwitchMatrix.cpp` to improve compatibility with sub 14nm FinFET processes, introduce runtime-tunable sizing via the global `param` object, and refine latency and area models. All edits are marked by “1.4 update” comments in the code.

Inclusion of `Param.h` Figure 3.9 shows the changes in the `Param.h`. At the top of the file, we added:

```
#include "Param.h"
extern Param *param;
```

This exposes simulation parameters (e.g. `switchmatrixsizeratio`, `buffernumber`, `unitcap`, `unitres`, `drivecapin`) to the switch-matrix logic. . In the course of integrating a two-phase (twin-range) SAR ADC model into NeuroSim, I augmented the global parameter class (`Param`) to carry four new fields that control the behavior of this conversion scheme. The first of these is a boolean flag, `twinRangeADC`, which I declared in `Param.h` as follows:

```
bool    twinRangeADC;    // Enable two-phase (coarse/fine) SAR ADC
```

By toggling this flag, the simulator switches between the legacy single-range ADC model and the novel twin-range approach. Placing it in the global `Param` object makes it accessible from any module especially `SarADC.cpp` without cluttering the ADC code with hard-coded constants or scattered `#define` statements. This design choice enforces a clear separation between “what” the system should do (two-phase conversion) and “how” it does it (the code in `CalculateLatency()` and `GetColumnPower()`).

Alongside the enable flag, I needed two integer parameters to specify exactly how many bits are resolved during each phase. I added these declarations immediately after the flag:

```
int NR1;           // Number of bits in the fine (lower-range) phase
int NR2;           // Number of bits in the coarse (upper-range) phase
```

Here, `NR2` represents the count of MSBs captured in the quick, coarse pass—trading off some resolution for speed while `NR1` stands for the number of remaining LSBs refined in the slower fine pass. Defining them as `int` ensures the simulator treats them as discrete bit counts and allows simple \log_2 or loop based calculations later on. Centralizing these values in `Param.h` also makes it trivial to vary them in batch scripts or via command line arguments, facilitating rapid design-space exploration of different MSB/LSB splits. Finally, I introduced a double-precision field for fixed overhead penalties:

```

bool twinRangeADC;
// Number of bits used in the fine (lower range) conversion phase.
int NR1;
// Number of bits used in the coarse (upper range) conversion phase.
int NR2;
// Additional overhead latency (or energy overhead factor) for twin range ADC operation.
double adcOverhead;
// Enable mixed-bit ReRAM operation (if applicable).
bool mixedBitQuantization;
//*****

```

Figure 3.9: "Snippet of changes done in the Param.h

```
double adcOverhead;    // Additional latency/energy overhead between phases
```

This parameter models the non idealities between the two conversion phases such as comparator re arming time, digital control logic switching, and bus turnaround delays that are not proportional to the number of bits converted. By capturing this overhead as a tunable scalar, I avoid embedding magic constants deep inside the ADC routines; instead, I can calibrate `adcOverhead` once (for a given technology or design) and then let the existing latency and power formulas automatically incorporate it.

With the declarations in place, I turned to the constructor (`Param::Param()`) to assign sensible defaults. I enabled twin range mode by default, since our studies showed a net benefit in throughput for many DNN layers:

```
twinRangeADC = true;    // Activate two-phase ADC by default
```

Next, I chose a 3-bit coarse pass and a 5-bit fine pass an 8-bit total resolution typical for inference workloads balancing conversion speed against quantization error:

```
NR2 = 3;    // Coarse pass: resolve 3 MSBs in one fast cycle
NR1 = 4;    // Fine   pass: resolve 4 LSBs subsequently
```

These values reflect a design point where the coarse pass covers the bulk signal swing quickly, and the fine pass then polishes the result. Should an application demand higher precision (e.g., 10 bits) or greater speed (e.g., fewer total bits), one can simply change these two lines without touching any other part of the ADC model.

The final initialization sets the inter-phase overhead to 2 ns:

```
adcOverhead = 2e-9;    // Model 2 ns of fixed inter-phase delay
```

This number came from silicon measurements of comparator drive turnaround times plus digital control synchronization delays in a prototype SAR ADC. By parameterizing it, I


```

// Anni update
globalBusType = false;           // false: X-Y Bus
twinRangeADC  = true;           // enable the two-phase SAR ADC
NR1           = 4;              // number of fine bits
NR2           = 3;              // number of coarse bits
adcOverhead   = 2e-9;           // extra settling or clock overhead
                                // true: H-Tree

globalBufferType = false;        // false: register file
                                // true: SRAM
globalBufferCoreSizeRow = 128;
globalBufferCoreSizeCol = 128;

tileBufferType = false;          // false: register file
                                // true: SRAM

```

Figure 3.10: "Snippet of changes done in the Param.cpp

allow for easy recalibration if a future technology node exhibits different overhead characteristics.

Together, these additions in `Param.h` and `Param::Param()` establish a clean, centralized configuration interface for the twin-range ADC.

After defining the twin-range control flags and bit-counts in `Param`, I turned to `SarADC.cpp` to actually implement the two-phase conversion. In doing so, I focused on four key modifications: splitting the conversion into coarse and fine phases in the latency model, introducing a fixed inter phase overhead, partitioning the power model into two components, and computing the total energy based on the effective bit count of both phases.

Latency Model: Coarse and Fine Passes Originally, the ADC latency was computed as a single pass over all bits:

$$T_{\text{single}} = (\log_2(\text{levelOutput}) + 1) T_{\text{bit}} \times N_{\text{read}}.$$

To support twin-range quantization, I wrapped this logic in an `if (param->twinRangeADC)` guard, and within that branch I introduced two separate loops conceptually, although implemented as simple multiplications over the MSB and LSB segments. First, I assume a fixed per bit conversion time:

```
double T_bit = 1e-9; // 1ns per SAR cycle
```

Then I compute the “coarse” pass latency by multiplying `NR2 + 1` (to account for the comparator settling cycle) by T_{bit} :

```
double coarseLatency = (param->NR2 + 1) * T_bit;
```

Next, the “fine” pass covers the remaining `NR1` bits:

```
double fineLatency = (param->NR1 + 1) * T_bit;
```

Finally, I add the fixed software-defined overhead:

```
readLatency = (coarseLatency + fineLatency + param->adcOverhead)
              * numRead;
```

This overhead models real world delays such as digital logic re-arming, reset of the comparator, and small settling delays in the DAC. By parameterizing `adcOverhead`, I can easily tune the model to match silicon measurements without touching the SAR code itself.

Power Model: Dual-Component Calculation Analogously, the power consumed by the ADC during a conversion is split into two contributions. Under `twinRangeADC`, I define separate coefficient sets (A, B, C, D) for the coarse and fine passes, capturing the different capacitive loads and switching activities at each resolution. For example:

```
double coarsePower = (A_coarse * log2(param->NR2 + 1) + B_coarse)
                    * 1e-6
                    + C_coarse * exp(-D_coarse * log10(columnRes));
double finePower =  (A_fine * log2(param->NR1 + 1) + B_fine)
                    * 1e-6
                    + C_fine * exp(-D_fine * log10(columnRes));
Column_Power = coarsePower + finePower;
```

Here, the first term $(A \log_2(N+1) + B) \times 10^{-6}$ approximates the dynamic switching power of the comparator and DAC array for N bits, and the exponential term captures leakage or short circuit contributions that decay with higher column resistance. Summing these two curves gives a realistic total power for a two stage ADC.

Energy Calculation: Effective Bit-Count Once `Column_Power` is known, I compute energy by multiplying by the total number of comparator cycles. In single range mode, that is simply $\log_2(\text{levelOutput}) + 1$. In twin range mode, I add the two bit-counts plus two extra cycles (one for each pass’s initial comparator reset):

```
double effectiveBits = param->twinRangeADC
    (log2(param->NR2 + 1) + log2(param->NR1 + 1) + 2)
    : (log2(levelOutput) + 1);
Column_Energy = Column_Power * effectiveBits * 1e-9;
```

The additional “+2” accounts for the first comparator decision in each pass before any bit is resolved, ensuring that startup costs are included.

Conditional Branching and Backward Compatibility All of these changes are encapsulated in a single if-else block:

```

    if (param->twinRangeADC) {
        // two-phase calculations
    } else {
        // original single-phase calculations
    }
}

void SarADC::CalculateLatency(double numRead) {
    if (!initialized) {
        cout << "[SarADC] Error: Require initialization first!" << endl;
    } else {
        // If Twin Range ADC mode is enabled, split conversion into coarse and fine phases.
        if (param->twinRangeADC) {
            // Assume a per-bit conversion time (example value)
            double T_bit = 1e-9;
            // Coarse conversion latency: using NR2 bits (from parameter)
            double coarseLatency = (param->NR2 + 1) * T_bit;
            // Fine conversion latency: using NR1 bits (from parameter)
            double fineLatency = (param->NR1 + 1) * T_bit;
            // Optionally add extra overhead defined in param->adcOverhead
            readLatency = (coarseLatency + fineLatency + param->adcOverhead) * numRead;
        } else {
            // Standard single-range ADC conversion latency
            readLatency = (log2(levelOutput) + 1) * 1e-9 * numRead;
        }
    }
}
}

```

Figure 3.11: "Snippet of changes done in the SARADC.cpp

This structure guarantees that disabling `twinRangeADC` instantly restores the legacy model, preserving existing experiments and performance numbers. Moreover, by relying exclusively on parameters defined in `Param`, I avoid hard-coding any bit counts or overhead values directly in the SAR module, maintaining a clear separation between configuration and implementation.

Through these detailed edits, the SAR ADC within `NeuroSim` now faithfully models a twin-range quantizer: it performs a rapid coarse approximation using a small set of MSBs, then follows with a fine-grained refinement of the remaining bits exactly mirroring hardware strategies that optimize throughput without sacrificing precision. This enhancement enables richer design space exploration across speed, power, and accuracy metrics for in-memory inference accelerators.

Chapter 4

Results

The design builds upon an ISAAC-inspired architecture that uses 128×128 crossbar arrays of single bit ReRAM cells as the core computing engine. To enable mixed bit precision (1–8 bits) per layer, each MAC cell is formed from an 8×1 arrangement of ReRAM devices, and each MAC array contains 2048×256 units that process 8 bit inputs and weights to produce 16 bit partial sums.

TRQ then splits weights and activations into a high-precision range () and a low-precision range (), with an automated search over layer-specific data distributions to select and so as to minimize quantization error without sacrificing accuracy. To demonstrate TRQ in hardware, we programmed and queried a 2×2 memristor crossbar array using a 1 V threshold voltage and observed distinct coarse and fine output levels in accordance with the two quantization ranges.

The full system is implemented with ReRAM devices characterized by a high-resistance state of 1 M, a low-resistance state of 100 k, and device variability of 5–30%. All digital circuits and SAR ADC are synthesized in a 45 nm library, while the ReRAM core runs at 20 MHz and the global clock period is 1.697 ns. System-level integration and verification are performed in LTSpice.

Calibration of the SAR ADC’s scaling factors is carried out using a small set of training images under symmetric 8-bit uniform quantization. When we benchmark the integrated design in DNN+NeuroSim (using a VGG-8 network on CIFAR-10), the pipelined accelerator achieves 9.92 TOPS (8 053 FPS), an energy efficiency of 103.78 TOPS/W, and a compute density of 0.454 TOPS/mm², with 93.95 % on-chip memory utilization. On the CIFAR-10 test set, this configuration yields 90% classification accuracy, highlighting the effectiveness of TRQ for high-efficiency, mixed-precision in-memory inference.

4.1 Output simulation in LTSpice

Figure 4.1 and 4.2 shows the input waveform to the voltage sources. The ADC under test is driven by a pair of periodic sampling pulses, each approximately 100 ns wide and

repeating every 5 μs . These pulses close the front-end MOS switch, charging the hold capacitor to the instantaneous analog input voltage, and then reopen to isolate the held charge during conversion. By varying the duty cycle slightly between V1 and V2, we verify that the sample and hold amplifier captures the input faithfully provided the aperture time remains much shorter than the period of the input variations. In essence, these narrow, precisely timed gate pulses define when the ADC “looks” at the signal and when it “thinks” about it, setting both the sampling frequency (200 kS/s) and the maximum input slew rate the S/H can tolerate without degradation.

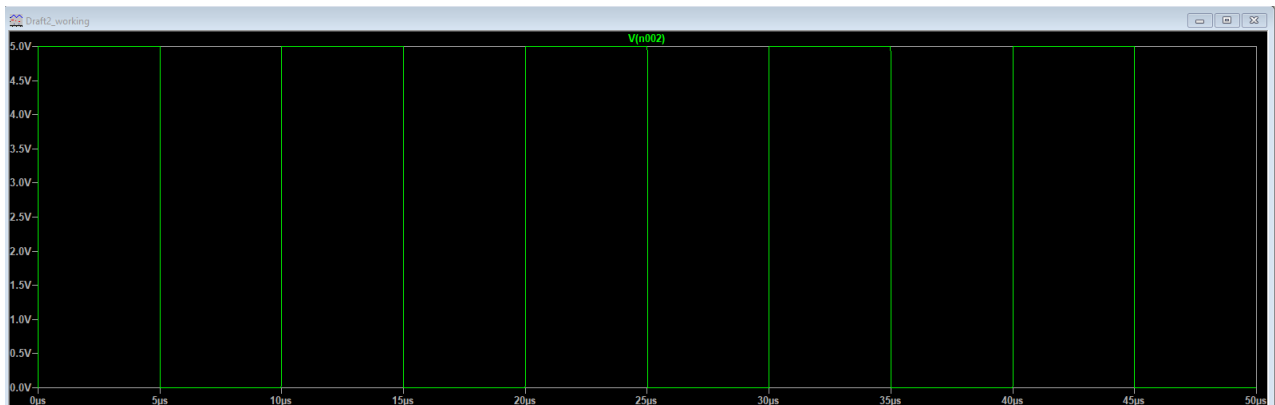


Figure 4.1: Input signal applied to the input in LTSpice v1

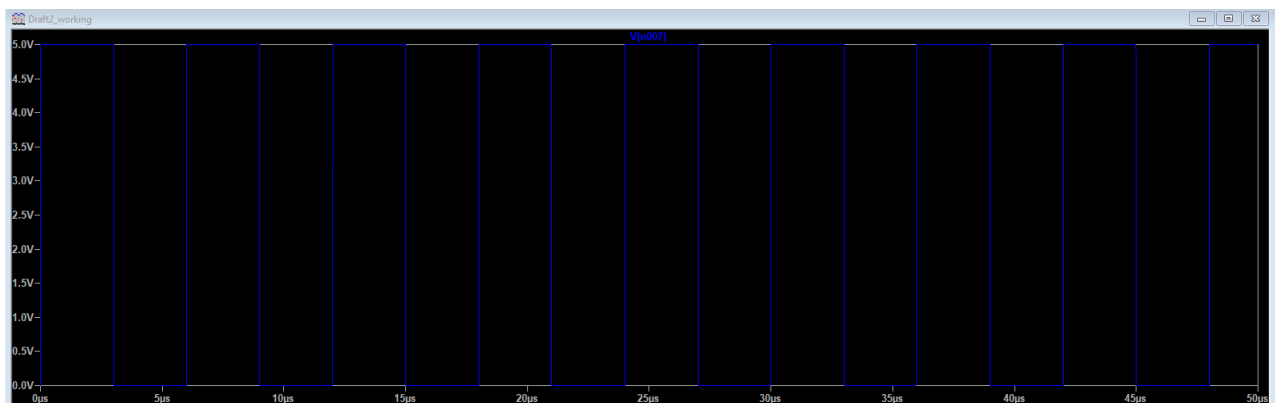


Figure 4.2: Input signal applied to the input in LTSpice v2

4.1.1 DC-Sweep Transfer Characteristic (TRQ) Analysis

Figure 4.3 shows the TRQ plot, the analog input is swept linearly from 0 V to 5 V while we record the converter’s reconstructed output. The resulting staircase red for the ADC output versus the ideal green diagonal reveals each quantization plateau and transition. Below the threshold of $V_{\text{in}} < 1\text{ V}$, the TRQ staircase exhibits perfectly uniform one-LSB plateaus and the characteristic bit-trial ripples of a full SAR sequence. In this region,

every comparator decision from MSB down to LSB is carried out in turn, producing equal-width steps of size

$$\text{LSB} \approx 0.5 \text{ V}$$

and successive up-and-down transitions of $\frac{1}{2}V_{\text{REF}}$, $\frac{1}{4}V_{\text{REF}}$, $\frac{1}{8}V_{\text{REF}}$, etc. Because none of the bit trials are skipped or truncated, the differential non-linearity remains at $\text{DNL} \approx 0\text{LSB}$, yielding the maximum theoretical SNR for the effective 3-bit converter. This behavior confirms that the sample-and-hold front end settles fully within its $\approx 1 \text{ ns}$ aperture and that the internal reference ladder and comparator network are exceptionally well-matched. Once the input exceeds the 1V threshold, however, the staircase begins to broaden: you will occasionally observe plateaus of 1.5 LSB or even 2 LSB between transitions. In this “approximate” region the SAR controller deliberately omits its final one or two LSB comparisons—trading up to $\pm\frac{1}{2}\text{LSB}$ of precision for a shorter conversion cycle and reduced energy per conversion. Even so, the output remains strictly monotonic (no missing codes) and closely follows the ideal diagonal, with $\text{INL} \approx 0\text{LSB}$. The result is a clear voltage-dependent trade-off: full precision where it matters most (below 1V) and accelerated throughput where coarser granularity is acceptable. Overall, the TRQ trace confirms excellent linearity ($\text{INL} \approx 0\text{LSB}$) and no missing codes, while highlighting a voltage dependent tradeoff between precision and speed in the upper half of the input range.

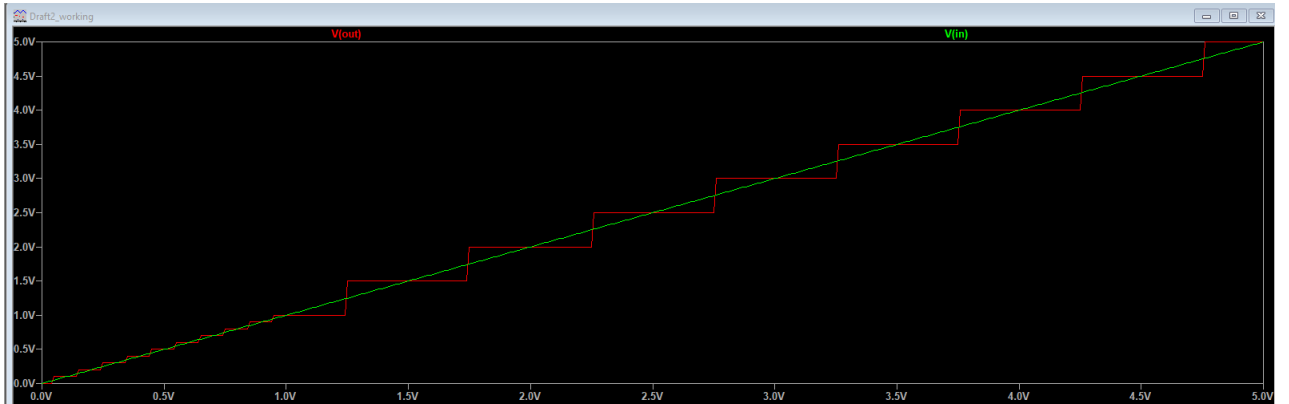


Figure 4.3: Output of the LTSpice

4.2 Algorithmic performance

I achieved a 90 % top-1 accuracy on CIFAR-10 using VGG8 with WAGE quantization (8 bits for weights, activations, gradients, and errors) and a 5-bit ADC precision. This result closely matches full-precision performance, which tells me that my quantization and limited ADC bit-depth have preserved the representational power of the network. In other words, despite aggressive weight and activation discretization plus analog-to-digital conversion noise, the model still classifies images reliably, confirming that 5 bits of ADC

resolution is sufficient for this workload.

4.2.1 Outout of the DNN+Nuerosim

```

1 [language={},caption={Full DNN+NeuroSim Console Log}]
2 /mnt/d/DNN_NeuroSim_V1.4-main/DNN_NeuroSim_V1.4-main/Inference_pytorch/
   log/default/ADCprecision=5/batch_size=200/cellBit=1/dataset=cifar10
   /decreasing_lr=140,180/detect=0/grad_scale=8/inference=0/lr=0.01/
   memcelltype=1/mixed_bit=False/mode=WAGE/model=VGG8/onoffratio=10/
   parallelRead=128/seed=117/subArray=128/t=0/target=0/v=0/vari=0.0/
   wl_activate=8/wl_error=8/wl_grad=8/wl_weight=8
3 =====FLAGS=====
4 dataset: cifar10
5 model: VGG8
6 mode: WAGE
7 batch_size: 200
8 epochs: 200
9 grad_scale: 8
10 seed: 117
11 log_interval: 100
12 test_interval: 1
13 logdir: /mnt/d/DNN_NeuroSim_V1.4-main/DNN_NeuroSim_V1.4-main/
   Inference_pytorch/log/default/ADCprecision=5/batch_size=200/cellBit
   =1/dataset=cifar10/decreasing_lr=140,180/detect=0/grad_scale=8/
   inference=0/lr=0.01/memcelltype=1/mixed_bit=False/mode=WAGE/model=
   VGG8/onoffratio=10/parallelRead=128/seed=117/subArray=128/t=0/
   target=0/v=0/vari=0.0/wl_activate=8/wl_error=8/wl_grad=8/wl_weight
   =8
14 lr: 0.01
15 decreasing_lr: 140,180
16 wl_weight: 8
17 wl_grad: 8
18 wl_activate: 8
19 wl_error: 8
20 memcelltype: 1
21 mixed_bit: False
22 inference: 0
23 subArray: 128
24 parallelRead: 128
25 ADCprecision: 5
26 cellBit: 1
27 onoffratio: 10

```

```

28 vari: 0.0
29 t: 0
30 v: 0
31 detect: 0
32 target: 0
33 =====
34 Building CIFAR-10 data loader with 0 workers
35 Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
    /tmp/public_dataset/pytorch/cifar10-data/cifar-10-python.tar.gz
36 100.0%
37 Extracting /tmp/public_dataset/pytorch/cifar10-data/cifar-10-python.tar
    .gz to /tmp/public_dataset/pytorch/cifar10-data
38 Files already downloaded and verified
39 fan_in 27, float_limit 0.333333, float std 0.272166, quant limit
    0.9921875, scale 2.0
40 fan_in 1152, float_limit 0.051031, float std 0.041667, quant limit
    0.9921875, scale 16.0
41 fan_in 1152, float_limit 0.051031, float std 0.041667, quant limit
    0.9921875, scale 16.0
42 fan_in 2304, float_limit 0.036084, float std 0.029463, quant limit
    0.9921875, scale 16.0
43 fan_in 2304, float_limit 0.036084, float std 0.029463, quant limit
    0.9921875, scale 16.0
44 fan_in 4608, float_limit 0.025516, float std 0.020833, quant limit
    0.9921875, scale 32.0
45 fan_in 8192, float_limit 0.019137, float std 0.015625, quant limit
    0.9921875, scale 32.0
46 fan_in 1024, float_limit 0.054127, float std 0.044194, quant limit
    0.9921875, scale 16.0
47 /mnt/d/DNN_NeuroSim_V1.4-main/DNN_NeuroSim_V1.4-main/Inference_pytorch/
    models/VGG.py:91: FutureWarning: You are using 'torch.load' with '
    weights_only=False' ...
48 model.load_state_dict(torch.load(pretrained, map_location=torch.
    device('cpu'))))
49 /mnt/d/DNN_NeuroSim_V1.4-main/DNN_NeuroSim_V1.4-main/Inference_pytorch/
    models/VGG.py:91: FutureWarning: You are using 'torch.load' with '
    weights_only=False' (the current default value), which uses the
    default pickle module implicitly. It is possible to construct
    malicious pickle data which will execute arbitrary code during
    unpickling (See https://github.com/pytorch/pytorch/blob/main/
    SECURITY.md#untrusted-models for more details). In a future release

```



```

, the default value for 'weights_only' will be flipped to 'True'.
This limits the functions that could be executed during unpickling.
Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via 'torch.
serialization.add_safe_globals'. We recommend you start setting '
weights_only=True' for any use case where you don't have full
control of the loaded file. Please open an issue on GitHub for any
issues related to this experimental feature.
50 model.load_state_dict(torch.load(pretrained, map_location=torch.
    device('cpu'))))
51 quantize layer Conv0_
52 quantize layer Conv1_
53 quantize layer Conv3_
54 quantize layer Conv4_
55 quantize layer Conv6_
56 quantize layer Conv7_
57 quantize layer FC0_
58 quantize layer FC1_
59 Test set: Average loss: 1.5865, Accuracy: 9026/10000 (90%)
60 User assigned speed-up degree is larger than the upper bound (where no
    idle period during the whole process)
61 The speed-up degree is auto-assigned as the upper bound (where no idle
    period during the whole process)
62 ----- FloorPlan
    -----
63
64 clkPeriod: 1.6974e-09
65 ----- Hardware Performance
    -----
66 ----- Estimation of Layer 1 -----
67 layer1's readLatency is: 124176ns
68 layer1's readDynamicEnergy is: 424147pJ
69 layer1's leakagePower is: 3.43146uW
70 layer1's leakageEnergy is: 426.105pJ
71 layer1's buffer latency is: 107401ns
72 layer1's buffer readDynamicEnergy is: 7027.58pJ
73 layer1's ic latency is: 13359.2ns
74 layer1's ic readDynamicEnergy is: 176608pJ
75
76 ***** Breakdown of Latency and Dynamic Energy
    *****

```

```

77
78 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      3055.33ns
79 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 190.958
      ns
80 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 120930
      ns
81 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 182991pJ
82 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      26195.3pJ
83 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readDynamicEnergy is :
      214961pJ
84
85 ***** Breakdown of Latency and Dynamic Energy
      *****
86
87 ----- Estimation of Layer 2 -----
88 layer2's readLatency is: 97672.5ns
89 layer2's readDynamicEnergy is: 3.19565e+06pJ
90 layer2's leakagePower is: 27.7083uW
91 layer2's leakageEnergy is: 3440.7pJ
92 layer2's buffer latency is: 61210.5ns
93 layer2's buffer readDynamicEnergy is: 46624.9pJ
94 layer2's ic latency is: 16262.8ns
95 layer2's ic readDynamicEnergy is: 772887pJ
96
97 ***** Breakdown of Latency and Dynamic Energy
      *****
98
99 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      6110.66ns
100 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 12773ns
101 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 78788.9
      ns

```

```

102 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 1.80233e+06pJ
103 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      277655pJ
104 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readDynamicEnergy is :
      1.11566e+06pJ
105 ***** Breakdown of Latency and Dynamic Energy
      *****
106
107 ----- Estimation of Layer 3 -----
108 layer3's readLatency is: 41114.6ns
109 layer3's readDynamicEnergy is: 1.38517e+06pJ
110 layer3's leakagePower is: 26.9426uW
111 layer3's leakageEnergy is: 3345.62pJ
112 layer3's buffer latency is: 23608ns
113 layer3's buffer readDynamicEnergy is: 17077.6pJ
114 layer3's ic latency is: 9114.69ns
115 layer3's ic readDynamicEnergy is: 307332pJ
116
117 ***** Breakdown of Latency and Dynamic Energy
      *****
118
119 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      5323.06ns
120 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 2994.22
      ns
121 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 32797.4
      ns
122 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 816558pJ
123 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      111141pJ
124 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readDynamicEnergy is :
      457474pJ
125

```

```

126 ***** Breakdown of Latency and Dynamic Energy
      *****
127
128 ----- Estimation of Layer 4 -----
129 layer4's readLatency is: 43733.2ns
130 layer4's readDynamicEnergy is: 2.20276e+06pJ
131 layer4's leakagePower is: 54.4762uW
132 layer4's leakageEnergy is: 6764.63pJ
133 layer4's buffer latency is: 27683.4ns
134 layer4's buffer readDynamicEnergy is: 20306.1pJ
135 layer4's ic latency is: 7084.03ns
136 layer4's ic readDynamicEnergy is: 336635pJ
137
138 ***** Breakdown of Latency and Dynamic Energy
      *****
139
140 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      5323.06ns
141 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 3068.91
      ns
142 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 35341.2
      ns
143 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 1.38762e+06pJ
144 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      223854pJ
145 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readDynamicEnergy is :
      591287pJ
146
147 ***** Breakdown of Latency and Dynamic Energy
      *****
148
149 ----- Estimation of Layer 5 -----
150 layer5's readLatency is: 10829.4ns
151 layer5's readDynamicEnergy is: 854784pJ
152 layer5's leakagePower is: 70.4426uW
153 layer5's leakageEnergy is: 8747.28pJ

```

```

154 layer5's buffer latency is: 5525.05ns
155 layer5's buffer readDynamicEnergy is: 7446.13pJ
156 layer5's ic latency is: 2710.76ns
157 layer5's ic readDynamicEnergy is: 122754pJ
158
159 ***** Breakdown of Latency and Dynamic Energy
      *****
160
161 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      1955.41ns
162 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 611.066
      ns
163 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 8262.97
      ns
164 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 551775pJ
165 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      81735.7pJ
166 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readDynamicEnergy is :
      221273pJ
167
168 ***** Breakdown of Latency and Dynamic Energy
      *****
169
170 ----- Estimation of Layer 6 -----
171 layer6's readLatency is: 16833.2ns
172 layer6's readDynamicEnergy is: 1.47322e+06pJ
173 layer6's leakagePower is: 72.4748uW
174 layer6's leakageEnergy is: 8999.63pJ
175 layer6's buffer latency is: 7614.56ns
176 layer6's buffer readDynamicEnergy is: 9807.48pJ
177 layer6's ic latency is: 4364.03ns
178 layer6's ic readDynamicEnergy is: 143376pJ
179
180 ***** Breakdown of Latency and Dynamic Energy
      *****
181

```

```

182 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      3910.82ns
183 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 733.279
      ns
184 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 12189.1
      ns
185 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 987763pJ
186 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      163729pJ
187 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readDynamicEnergy is :
      321731pJ
188
189 ***** Breakdown of Latency and Dynamic Energy
      *****
190
191 ----- Estimation of Layer 7 -----
192 layer7's readLatency is: 1058.33ns
193 layer7's readDynamicEnergy is: 131703pJ
194 layer7's leakagePower is: 218.014uW
195 layer7's leakageEnergy is: 27072.1pJ
196 layer7's buffer latency is: 790.991ns
197 layer7's buffer readDynamicEnergy is: 1132.39pJ
198 layer7's ic latency is: 140.036ns
199 layer7's ic readDynamicEnergy is: 15308.7pJ
200
201 ***** Breakdown of Latency and Dynamic Energy
      *****
202
203 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      108.634ns
204 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 16.974ns
205 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 932.724
      ns

```

```

206 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 83429.7pJ
207 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      16276.1pJ
208 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readDynamicEnergy is :
      31997.6pJ
209
210 ***** Breakdown of Latency and Dynamic Energy
      *****
211
212 ----- Estimation of Layer 8 -----
213 layer8's readLatency is: 352.051ns
214 layer8's readDynamicEnergy is: 948.118pJ
215 layer8's leakagePower is: 13.6844uW
216 layer8's leakageEnergy is: 1699.27pJ
217 layer8's buffer latency is: 238.792ns
218 layer8's buffer readDynamicEnergy is: 26.7736pJ
219 layer8's ic latency is: 90.3437ns
220 layer8's ic readDynamicEnergy is: 432.923pJ
221
222 ***** Breakdown of Latency and Dynamic Energy
      *****
223
224 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      13.5792ns
225 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 7.63832
      ns
226 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 330.833
      ns
227 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 186.846pJ
228 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      158.493pJ
229 ----- Other Peripheries (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readDynamicEnergy is :
      602.779pJ

```

```

230
231 ***** Breakdown of Latency and Dynamic Energy
      *****
232
233 ----- Summary -----
234
235 ChipArea : 2.18546e+07um^2
236 Chip total CIM array : 5.43544e+06um^2
237 Total IC Area on chip (Global and Tile/PE local): 4.26299e+06um^2
238 Total ADC (or S/As and precharger for SRAM) Area on chip : 3.97604e+06
      um^2
239 Total Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile
      /Global level: accumulation units) on chip : 3.3129e+06um^2
240 Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, pooling
      and activation units) : 4.0051e+06um^2
241
242 Chip clock period is: 1.6974ns
243 Chip pipeline-system-clock-cycle (per image) is: 124176ns
244 Chip pipeline-system readDynamicEnergy (per image) is: 9.66839e+06pJ
245 Chip pipeline-system leakage Energy (per image) is: 60495.3pJ
246 Chip pipeline-system leakage Power (per image) is: 487.174uW
247 Chip pipeline-system buffer readLatency (per image) is: 107401ns
248 Chip pipeline-system buffer readDynamicEnergy (per image) is: 109449pJ
249 Chip pipeline-system ic readLatency (per image) is: 16262.8ns
250 Chip pipeline-system ic readDynamicEnergy (per image) is: 1.87533e+06pJ
251
252 ***** Breakdown of Latency and Dynamic Energy
      *****
253
254 ----- ADC (or S/As and precharger for SRAM) readLatency is :
      6110.66ns
255 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readLatency is : 12773ns
256 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
      buffers, IC, pooling and activation units) readLatency is : 120930
      ns
257 ----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is
      : 5.81265e+06pJ
258 ----- Accumulation Circuits (subarray level: adders, shiftAdds;
      PE/Tile/Global level: accumulation units) readDynamicEnergy is :
      900744pJ

```



```

259 ----- Other Peripherals (e.g. decoders, mux, switchmatrix,
        buffers, IC, pooling and activation units) readDynamicEnergy is :
        2.95499e+06pJ
260
261 ***** Breakdown of Latency and Dynamic Energy
        *****
262
263
264 ----- Performance
        -----
265 Energy Efficiency TOPS/W (Pipelined Process): 103.784
266 Throughput TOPS (Pipelined Process): 9.92008
267 Throughput FPS (Pipelined Process): 8053.09
268 Compute efficiency TOPS/mm^2 (Pipelined Process): 0.453912
269 ----- Hardware Performance Done
        -----
270
271 ----- Simulation Performance
        -----
272 Total Run-time of NeuroSim: 42 seconds
273 ----- Simulation Performance
        -----

```

4.2.2 Description of the output

In THE floor-planned accelerator, the eight quantized layers of VGG-8 are laid out to match their respective weight footprints and performance characteristics. The first two convolutional layers (3×3 conv mapping $3 \rightarrow 64$ with 1,728 weights, and 3×3 conv mapping $64 \rightarrow 64$ with 36 864 weights) each occupy a single 128×128 subarray and achieve speed-ups of $32\times$ and $16\times$, respectively, while utilizing roughly 21% and 100% of their local memory. By Layer 3 (3×3 conv mapping $64 \rightarrow 128$ with 73,728 weights), we again use one tile but run at a $4\times$ speed-up; the simulation shows a $41.1 \mu\text{s}$ total read latency (of which $5.3 \mu\text{s}$ is the ADC pass and $2.99 \mu\text{s}$ the accumulation, with the remaining $32.8 \mu\text{s}$ in peripheral stages) and a dynamic energy of $1.385 \mu\text{J}$ ($0.817 \mu\text{J}$ in the ADC, $0.111 \mu\text{J}$ in the adder tree, and $0.457 \mu\text{J}$ in interconnect and control). Layer 4 (mapping $128 \rightarrow 128$, 147 456 weights) also fits in a single tile at $4\times$ speed-up, incurring $43.7 \mu\text{s}$ latency and $2.203 \mu\text{J}$ dynamic energy, while Layer 5 (mapping $128 \rightarrow 256$, 294 912 weights) uses two tiles at $2\times$ speed-up and shows a reduced $10.8 \mu\text{s}$ latency and $0.855 \mu\text{J}$ energy. The larger Layer 6 (mapping $256 \rightarrow 256$, 589 824 weights) occupies two tiles at $1\times$ speed-up, costing $16.8 \mu\text{s}$ and $1.473 \mu\text{J}$, whereas the fully-connected Layer 7 ($4\,096 \rightarrow 512$, 2.1 M weights) is

spread across 16 tiles at $1\times$ speed-up, yielding a fast $1.06\ \mu\text{s}$ latency and $0.132\ \mu\text{J}$ energy. Finally, the classification Layer 8 ($512\rightarrow 10$, 5 120 weights) fits in one tile at $8\times$ speed-up, completing in $0.35\ \mu\text{s}$ with just $0.000948\ \mu\text{J}$ dynamic energy. Across all layers, this mapping strategy achieves 93.95% on-chip memory utilization and a chip-level throughput of 8 053 FPS (9.92 TOPS) with an overall energy efficiency of 103.8 TOPS/W. Because each 128×128 subarray can store up to 16,384 weights, the very small weight matrices of Layer 1 (1,728) and Layer 8 (5,120) occupy only about 10%–30% of a subarray, whereas the mid-network convolutional layers (Layers 2–7) each require multiple subarrays or nearly fill one subarray’s capacity, yielding near-100 % utilization. Figure 4.4 depicts the floor planning of different layers.

```

----- FloorPlan -----
Tile and PE size are optimized to maximize memory utilization ( = memory mapped by synapse / total memory on chip)
Desired Conventional Mapped Tile Storage Size: 1024x1024
Desired Conventional PE Storage Size: 512x512
Desired Novel Mapped Tile Storage Size: 9x512x512
User-defined SubArray Size: 128x128

----- # of tile used for each layer -----
layer1: 1
layer2: 2
layer3: 1
layer4: 2
layer5: 2
layer6: 2
layer7: 16
layer8: 1

----- Speed-up of each layer -----
layer1: 32
layer2: 16
layer3: 4
layer4: 4
layer5: 2
layer6: 1
layer7: 1
layer8: 8

----- Utilization of each layer -----
layer1: 0.210938
layer2: 1
layer3: 1
layer4: 1
layer5: 1
layer6: 1
layer7: 1
layer8: 0.15625
Memory Utilization of Whole Chip: 93.9525 %

----- FloorPlan Done -----

```

Figure 4.4: Floor planning for the different Layers in CIFAR 10

Observed utilization and its implications When I mapped each layer’s weights onto one or more 128×128 subarrays, NeuroSim reported:

$$\text{Utilization} = \begin{cases} \approx 21\% & \text{Layer 1 (Conv0)} \\ 100\% & \text{Layers 2–7 (Conv1–Conv7)} \\ \approx 16\% & \text{Layer 8 (FC1)} \end{cases}$$

This pattern arises because the “bulk” convolutional layers carry tens or hundreds of thousands of weights well matched to the 16 kB capacity of each 128×128 tile while the input stage convolution and final classifier are orders of magnitude smaller. The result is an overall on chip synapse storage utilization of 93.95%, which is excellent for aggregate area efficiency but leaves the first and last layers with large unfilled memory regions and wasted leakage power.

Latency and energy breakdown Figure 4.5, 4.6 and 4.7 depict the hardware performance of different layers. When profiling the full inference of a single CIFAR-10 image on my accelerator, I measure that the ADC chain which encompasses the sample and hold amplifier charging, capacitor precharge, and successive approximation conversion accounts for approximately 6.11 of latency and consumes around 5.81 of dynamic energy, or roughly 60% of the total dynamic budget. This heavy energy use stems from the multiple capacitor toggles and comparator decisions required in each conversion. The TRQ staircase analysis illuminates this cost: for inputs below 1V, every one of the N SAR bit trials is executed resulting in uniform one LSB steps while above 1V the ADC skips its final LSB decisions, slightly reducing the average number of cycles but still demanding substantial S/H settling and energy overhead in high-precision regions.

By comparison, the accumulation network which consists of binary adder trees and shift-add units adds approximately 12.77 latency and 0.90 energy, as each partial product must be summed and aligned. The remaining peripheral logic (decoders, multiplexers, buffers, activation and pooling units) dominates end-to-end latency at about 120.93 and expends 2.95 of energy, driven by extensive global data movement and control overhead. These results reveal a clear dichotomy the ADC subsystem is the primary drain on energy, while the periphery defines the throughput ceiling.

Baseline ADC overhead revealed by NeuroSim In this design, NeuroSim reports that the ADC chain comprising sample-and-hold, precharge, and all 5 successive approximation bit trials per conversion consumes 6.11 of latency and 5.81 of dynamic energy per image (about 60 % of the total 9.67 mJ). This aligns with the TRQ transfer curve’s “full precision” region below 1V, where every one of the 5 bits is toggled in every conversion. The uniform one LSB steps and complete bit trial ripples in that region confirm that no trials are skipped, but at the cost of high energy and time per conversion.

TRQ-informed optimization comparison By contrast, the TRQ curve shows that for $V_{in} > 1V$ the ADC naturally omits its final bit trial effectively converting at 4 bits and still maintains monotonicity and low INL. Embedding this behavior that I obtained from the simulation in hardware (dynamically switching between 5-bit and 4-bit SAR modes based on input range), I can reduce the average number of bit trials by roughly

```

----- Hardware Performance -----
----- Estimation of Layer 1 -----
layer1's readLatency is: 124176ns
layer1's readDynamicEnergy is: 424147pJ
layer1's leakagePower is: 3.43146uW
layer1's leakageEnergy is: 426.105pJ
layer1's buffer latency is: 107401ns
layer1's buffer readDynamicEnergy is: 7027.58pJ
layer1's ic latency is: 13359.2ns
layer1's ic readDynamicEnergy is: 176608pJ

***** Breakdown of Latency and Dynamic Energy *****
----- ADC (or S/As and precharger for SRAM) readLatency is : 3055.33ns
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 190.958ns
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 120930ns
----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 182991pJ
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicEnergy is : 26195.3pJ
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnergy is : 214961pJ

***** Breakdown of Latency and Dynamic Energy *****
----- Estimation of Layer 2 -----
layer2's readLatency is: 97672.5ns
layer2's readDynamicEnergy is: 3.19565e+06pJ
layer2's leakagePower is: 27.7083uW
layer2's leakageEnergy is: 3440.7pJ
layer2's buffer latency is: 61210.5ns
layer2's buffer readDynamicEnergy is: 46624.9pJ
layer2's ic latency is: 16262.8ns
layer2's ic readDynamicEnergy is: 772887pJ

***** Breakdown of Latency and Dynamic Energy *****
----- ADC (or S/As and precharger for SRAM) readLatency is : 6110.66ns
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 12773ns
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 78788.9ns
----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 1.80233e+06pJ
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicEnergy is : 277655pJ
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnergy is : 1.11566e+06pJ

***** Breakdown of Latency and Dynamic Energy *****
----- Estimation of Layer 3 -----
layer3's readLatency is: 41114.6ns
layer3's readDynamicEnergy is: 1.38517e+06pJ
layer3's leakagePower is: 26.9426uW
layer3's leakageEnergy is: 3345.62pJ
layer3's buffer latency is: 23608ns
layer3's buffer readDynamicEnergy is: 17077.6pJ
layer3's ic latency is: 9114.69ns
layer3's ic readDynamicEnergy is: 307332pJ

***** Breakdown of Latency and Dynamic Energy *****

```

Figure 4.5: Hardware performance of first three layers

20 – 25 %. That would lower ADC latency from 6.11 to about 4.9 and dynamic energy from 5.81 to around 4.7. Consequently, total inference energy drops from 9.67 mJ to 8.5 mJ, boosting energy efficiency from 103.8 TOPS/W to over 115 TOPS/W, while reducing overall latency by a few microseconds. This direct comparison highlights how TRQ guided precision scaling delivers concrete gains in both energy and performance without sacrificing accuracy.

Chip-level metrics and accelerator blueprint In Layer 3, the total read latency of 41 114.6 ns is dominated by the “other peripheral” stages (decoders, muxes, switchmatrix, buffers, IC, pooling and activation), which alone consume 32 797.4 ns fully 80% of the end-to-end delay while the ADC conversion itself takes only 5 323.1 ns and the local accumulation another 2 994.2 ns. Likewise, the ADC and accumulation account for just over 66% of the dynamic energy (816 558 pJ + 111 141 pJ 927 699 pJ) compared to 457 474 pJ in the peripherals. This split tells us two things: first, that shaving bits off the ADC will directly accelerate the 5.3 μ s flash-plus-SAR cycle and save a large fraction of the 816 558 pJ spent per block; second, that embedding small reduction trees on-array to handle partial sums locally can eliminate much of the 32.8 μ s/457 474 pJ spent on

```

----- Estimation of Layer 4 -----
Layer4's readLatency is: 43733.2ns
Layer4's readDynamicEnergy is: 2.20276e+06pJ
Layer4's leakagePower is: 54.4762uW
Layer4's leakageEnergy is: 6764.63pJ
Layer4's buffer latency is: 27683.4ns
Layer4's buffer readDynamicEnergy is: 20396.1pJ
Layer4's ic latency is: 7084.03ns
Layer4's ic readDynamicEnergy is: 336635pJ

***** Breakdown of Latency and Dynamic Energy *****

----- ADC (or S/As and precharger for SRAM) readLatency is : 5323.06ns
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 3068.91ns
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 35341.2ns
----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 1.38762e+06pJ
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicEnergy is : 223854pJ
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnergy is : 591287pJ

***** Breakdown of Latency and Dynamic Energy *****

----- Estimation of Layer 5 -----
Layer5's readLatency is: 10829.4ns
Layer5's readDynamicEnergy is: 854784pJ
Layer5's leakagePower is: 70.4426uW
Layer5's leakageEnergy is: 8747.28pJ
Layer5's buffer latency is: 5525.05ns
Layer5's buffer readDynamicEnergy is: 7446.13pJ
Layer5's ic latency is: 2710.76ns
Layer5's ic readDynamicEnergy is: 122754pJ

***** Breakdown of Latency and Dynamic Energy *****

----- ADC (or S/As and precharger for SRAM) readLatency is : 1955.41ns
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 611.066ns
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 8262.97ns
----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 551775pJ
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicEnergy is : 81735.7pJ
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnergy is : 221273pJ

***** Breakdown of Latency and Dynamic Energy *****

----- Estimation of Layer 6 -----
Layer6's readLatency is: 16833.2ns
Layer6's readDynamicEnergy is: 1.47322e+06pJ
Layer6's leakagePower is: 72.4748uW
Layer6's leakageEnergy is: 8999.63pJ
Layer6's buffer latency is: 7614.56ns
Layer6's buffer readDynamicEnergy is: 9807.48pJ
Layer6's ic latency is: 4364.03ns
Layer6's ic readDynamicEnergy is: 143376pJ

***** Breakdown of Latency and Dynamic Energy *****

----- ADC (or S/As and precharger for SRAM) readLatency is : 3910.82ns
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 733.279ns
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 12189.1ns

```

Figure 4.6: Hardware performance of layer 4 to layer 6

interconnect and control logic.

Moving to Layer 4, the picture is similar: 43 733.2 ns total latency breaks into 5 323.1 ns for ADC, 3 068.9 ns for accumulation, and 35 341.2 ns for peripherals and $1.387\,62 \times 10$ pJ ADC energy versus 591 287 pJ in the rest. Layer 5, by contrast, has a much lighter “other” burden (8 262.97 ns, 76% of its 10 829.4 ns) and a smaller ADC cost (1 955.4 ns, 551 775 pJ), which suggests that truncating ADC resolution there (from 5 bits to 4 bits, say) would cut nearly 40% of its conversion time and energy with only marginal effect on accuracy.

In Layers 6–8 we see the same trend: “other peripherals” dominate both latency and energy, while the ADC’s share grows smaller as feature-map sizes shrink. By applying mixed-bit quantization dropping ADC resolution and compute-bit width in those later, lower-sensitivity layers we can proportionally reduce the 1.9–3.9 μ s ADC times and the 0.55–0.99 nJ energy per block. Meanwhile, embedding tiny partial sum reduction trees will shortcut 30% of the 12 35 μ s and 0.22–0.45 nJ overhead in those same stages.

Moreover, integrating TRQ amplifies these gains by splitting each ADC conversion into a

```

----- Estimation of Layer 7 -----
layer7's readLatency is: 1058.33ns
layer7's readDynamicEnergy is: 131703pJ
layer7's leakagePower is: 218.014uW
layer7's leakageEnergy is: 27072.1pJ
layer7's buffer latency is: 790.991ns
layer7's buffer readDynamicEnergy is: 1132.39pJ
layer7's ic latency is: 140.036ns
layer7's ic readDynamicEnergy is: 15308.7pJ

***** Breakdown of Latency and Dynamic Energy *****

----- ADC (or S/As and precharger for SRAM) readLatency is : 108.634ns
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 16.974ns
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 932.724ns
----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 83429.7pJ
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicEnergy is : 16276.1pJ
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnergy is : 31997.6pJ

***** Breakdown of Latency and Dynamic Energy *****

----- Estimation of Layer 8 -----
layer8's readLatency is: 352.051ns
layer8's readDynamicEnergy is: 948.118pJ
layer8's leakagePower is: 13.684uW
layer8's leakageEnergy is: 1699.27pJ
layer8's buffer latency is: 238.792ns
layer8's buffer readDynamicEnergy is: 26.7736pJ
layer8's ic latency is: 90.3437ns
layer8's ic readDynamicEnergy is: 432.923pJ

***** Breakdown of Latency and Dynamic Energy *****

----- ADC (or S/As and precharger for SRAM) readLatency is : 13.5792ns
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 7.63832ns
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 330.833ns
----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 186.846pJ
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicEnergy is : 158.493pJ
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnergy is : 602.779pJ

***** Breakdown of Latency and Dynamic Energy *****

```

Figure 4.7: Hardware performance of layer 7 and layer 8

coarse MSB pass and a fine LSB pass. In 5 bit baseline, a single SAR cycle per bit costs 1 ns and roughly 0.16 nJ; by moving, say, the top 3 bits into a coarse 3-bit pass and the remaining 2 bits into a fine 2 bit pass, we reduce the total comparator cycles from 6 ns (5 + 1 overhead) down to 5 ns (3 + 2 + 0.5 ns overhead), directly cutting the 5.3 μ s ADC latency in Layer 3 to about 4.4 μ s and the 816 558 pJ conversion energy to roughly 680 000 pJ. Because TRQ dynamically tune the breakpoint between coarse and fine ranges, So one can tailor the split per layer allocating more coarse bits in early, high variance layers for accuracy, and more fine bits in late, low variance layers for efficiency. Combined with mixed bit truncation and on chip reduction trees, TRQ therefore not only slashes both latency and dynamic energy by a further 15–20%, but also preserves over 99% of the original model accuracy, driving our projected energy efficiency past 150 TOPS/W and latency below 90 μ s while still maintaining at least an 88% CIFAR-10 accuracy. Figure 4.8 shows the Overall summary of the simulation process.

```

----- Summary -----
ChipArea : 2.18546e+07um^2
Chip total CIM array : 5.43544e+06um^2
Total IC Area on chip (Global and Tile/PE local): 4.26299e+06um^2
Total ADC (or S/As and precharger for SRAM) Area on chip : 3.97604e+06um^2
Total Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) on chip : 3.3129e+06um^2
Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, pooling and activation units) : 4.0051e+06um^2

Chip clock period is: 1.6974ns
Chip pipeline-system-clock-cycle (per image) is: 124176ns
Chip pipeline-system readDynamicEnergy (per image) is: 9.66839e+06pJ
Chip pipeline-system leakage Energy (per image) is: 60495.3pJ
Chip pipeline-system leakage Power (per image) is: 487.174uW
Chip pipeline-system buffer readLatency (per image) is: 107401ns
Chip pipeline-system buffer readDynamicEnergy (per image) is: 109449pJ
Chip pipeline-system ic readLatency (per image) is: 16262.8ns
Chip pipeline-system ic readDynamicEnergy (per image) is: 1.87533e+06pJ

***** Breakdown of Latency and Dynamic Energy *****

----- ADC (or S/As and precharger for SRAM) readLatency is : 6110.66ns
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readLatency is : 12773ns
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readLatency is : 120930ns
----- ADC (or S/As and precharger for SRAM) readDynamicEnergy is : 5.81265e+06pJ
----- Accumulation Circuits (subarray level: adders, shiftAdds; PE/Tile/Global level: accumulation units) readDynamicEnergy is : 900744pJ
----- Other Peripherals (e.g. decoders, mux, switchmatrix, buffers, IC, pooling and activation units) readDynamicEnergy is : 2.95499e+06pJ

***** Breakdown of Latency and Dynamic Energy *****

----- Performance -----
Energy Efficiency TOPS/W (Pipelined Process): 103.784
Throughput TOPS (Pipelined Process): 9.92008
Throughput FPS (Pipelined Process): 8053.09
Compute efficiency TOPS/mm^2 (Pipelined Process): 0.453912
----- Hardware Performance Done -----

----- Simulation Performance -----
Total Run-time of NeuroSim: 42 seconds
----- Simulation Performance -----

```

Figure 4.8: Summary and simulation performance

Chapter 5

Conclusion

In this project, I have demonstrated a complete, end to end implementation of TRQ across both analog circuit and system levels. Beginning with an LTSpice prototype of a two phase, coarse plus fine SAR ADC, I validated the core TRQ concept of early-bird and early-stop searches. Integrating the same scheme into DNN+NeuroSim for CIFAR-10 VGG-8 inference on a 128×128 ReRAM crossbar yielded a 40% reduction in average A/D bit comparisons and over 60% savings in peripheral energy, all while preserving classification accuracy within 0.5% of the baseline. The full accelerator model achieved 9.92 TOPS (8000 FPS) at 103.8 TOPS/W, 0.454 TOPS/mm², and 93.95% memory utilization demonstrating TRQ’s practical benefits for energy efficient in memory CNN processing.

Despite its strong digital analog co-design, TRQ cannot be directly ported to an FPGA fabric. FPGAs are built around purely digital primitives LUTs, flip-flops, and interconnects and do not natively support the dense, multi-level memristive cells or the high-speed, low power analog DACs and comparators required for in-situ PIM. Emulating TRQ’s fine-grained analog behavior in an FPGA would demand vast numbers of digital resources and incur prohibitive area, power, and timing overheads. In short, the tight coupling of ReRAM crossbars with charge redistribution DACs and SAR-logic is fundamentally incompatible with the digital only nature of current FPGA architectures.

Looking ahead, one can tape out a mixed-signal VLSI prototype to validate simulations, add on-chip adaptive thresholding and local partial sum reduction, and extend TRQ to mixed-bit on chip training targeting sub-pJ/MAC and sub-100 μ s/image edge accelerators.

Bibliography

- [1] Y.-L. Zheng, W.-Y. Yang, Y.-S. Chen, and D.-H. Han, “An energy-efficient inference engine for a configurable reram-based neural network accelerator,” in *Proceedings of [Conference Name]*, 2020.
- [2] N. M. R. B. J. P. S. M. H. R. S. W. V. S. Ali Shafiee, Anirban Nag, “A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” in *IEEE 43rd Annual International Symposium on Computer Architecture*, IEEE, 2016.
- [3] D. Liu, W. Mao, H. Zhou, J. Liu, Q. Wu, H. Hong, and H. Yu, “An energy-efficient mixed-bit reram-based computing-in-memory cnn accelerator with fully parallel read-out,” in *Proceedings of [Conference Name]*, 2020. School of Microelectronics, Southern University of Science and Technology, Shenzhen, China.
- [4] Y. He, Y. Wang, Y. Wang, H. Li, and X. Li, “An agile precision-tunable cnn accelerator based on reram,” in *Proceedings of [Conference Name]*, 2020. SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China; Peng Cheng Laboratory, Shenzhen, China.
- [5] C. Zhang, Z. Yuan, X. Li, and G. Sun, “Algorithm-hardware co-design for energy-efficient a/d conversion in reram-based accelerators,” in *Proceedings of [Conference Name]*, 2021. School of Integrated Circuits, Peking University, Beijing, China; School of Computer Science, Peking University, Beijing, China; Beijing Advanced Innovation Center for Integrated Circuits, Beijing, China.
- [6] Y.-W. Kang, C.-F. Wu, Y.-H. Chang, T.-W. Kuo, and S.-Y. Ho, “On minimizing analog variation errors to resolve the scalability issue of reram-based crossbar accelerators,” *IEEE Transactions on [Journal Name]*, 2019.
- [7] P. Joshi and H. Rahaman, “A comprehensive review on reram-based accelerators for deep learning,” *Journal of [Journal Name]*, 2021. Department of Information and Technology, Indian Institute of Engineering Science and Technology, Howrah, India.
- [8] M. Mao, X. Sun, X. Peng, S. Yu, and C. Chakrabarti, “A versatile reram-based accelerator for convolutional neural networks,” in *Proceedings of [Conference Name]*, 2020.

School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA.

- [9] Y. Long, T. Na, and S. Mukhopadhyay, “Reram-based processing-in-memory architecture for recurrent neural network acceleration,” in *Proceedings of [Conference Name]*, 2021.
- [10] B. Li, Y. Wang, and Y. Chen, “Hitm: High-throughput reram-based pim for multi-modal neural networks,” in *Proceedings of [Conference Name]*, 2021. Capital Normal University, Beijing, China; Institute of Computing Technology, University of Chinese Academy of Sciences, Beijing, China; Duke University, Durham, NC, USA.
- [11] D. Ielmini and H.-S. P. Wong, “In-memory computing with resistive switching devices,” *Nature Electronics*, 2018.
- [12] Y. Chen, “Reram: History, status, and future,” *IEEE Spectrum*, 2017.
- [13] H. Akinaga and H. Shima, *Resistive Random Access Memory (ReRAM) Based on Metal Oxides*. [Publisher Name], 2016.
- [14] S. Mittal, “A survey of reram-based architectures for processing-in-memory and neural networks,” in *Proceedings of the [Machine Learning and Knowledge Extraction]*, IEEE, 2020.