In/near-Memory for MAC and Matrix Computations

Dissertation submitted in partial fulfillment of the requirements for the award of the degree of

Master of Technology

by

Durgam Saiprasad

(Roll No. 24m1210)

Supervisor :

Prof. Sachin B. Patkar

Co-supervisor :

Prof. Pradeep R. Nair

Prof. Veeresh Deshpande

Solid State Devices

High Performance Computing Laboratory

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Mumbai - 400076, India

October, 2025

Declaration

I declare that this written submission represents my ideas in my own words. Where others' ideas and words have been included, I have adequately cited and referenced the original source. I declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated, or falsified any idea/data/-fact/source in my submission. I understand that any violation of the above will cause disciplinary action by the Institute and can also evoke penal action from the source which has thus not been properly cited or from whom proper permission has not been taken when needed.

.....

Durgam Saiprasad
Roll No.: 24m1210
Date:
Place: IIT Bombay

Abstract

As machine learning workloads shift to edge devices, the demand for efficient, low-latency hardware accelerators has grown significantly. This dissertation addresses the critical bottleneck of data movement in traditional architectures by exploring a processor-in-memory (PIM) approach. First, we implement and evaluate SPAR-2, a programmable SIMD processor array overlay, on a Xilinx Pynq FPGA to accelerate matrix-vector multiplication. The architecture's performance is analyzed, highlighting the resource efficiency of utilizing dedicated Block RAMs (BRAMs) over distributed logic for on-chip storage.

Second, we reviewed CIM paper, designed controller and decoder for the CIM which does MAC operation.Next as a foundational step toward true in-memory computation, this work presents a successful device-level simulation of a Resistive RAM (RRAM) cell. Using the JART Verilog-A model in the Cadence Spectre environment, the core I-V characteristics and resistance-switching behavior of the RRAM device are validated. This research demonstrates a practical FPGA-based accelerator for current ML tasks and establishes the viability of RRAM technology for future architectures where computation occurs directly within the memory fabric, promising substantial improvements in performance and energy efficiency for edge AI.

Contents

List of Figures

Chapter 1

Introduction

The rapid expansion of artificial intelligence (AI) and machine learning (ML) workloads has created a growing demand for high-speed, energy-efficient computing platforms. Conventional von Neumann architectures suffer from the "memory wall" problem, where frequent data movement between the processor and memory dominates both energy and latency. Compute-In-Memory (CIM) architectures offer an elegant solution by performing arithmetic operations, such as Multiply-and-Accumulate (MAC), directly within the memory array, thereby minimizing data transfer and improving overall computational efficiency.

Chapter 2

Literature Review

This chapter presents a brief review of the research works that form the foundation of the present study.

SPAR-2: A SIMD Processor Array for Machine Learning in IoT Devices (2020)

Suhail Basalama and Atiyehsadat Panahi proposed **SPAR-2**, a SIMD processor array optimized for low-power machine learning applications in IoT devices. The architecture emphasizes parallel matrix-vector computation and efficient data handling for sparse workloads. Based on the concepts discussed in this paper, I have implemented the **spar matrix-vector multiplication** operation as explained in detail in Chapters 3 and 4 of this thesis.

3D MIGCIM: A Material-Device-Circuit Co-Design for 3D Monolithic Integrated 4T2C ITO Gain Cell Compute-In-Memory on 2 nm GAA CMOS

The 3D MIGCIM work introduces a monolithically integrated Compute-In-Memory (CIM) architecture combining Indium Tin Oxide (ITO) 4T2C eDRAM cells with 2 nm GAA CMOS logic. The paper demonstrates how ternary weight storage and analog MAC operations can be efficiently implemented within the memory array.

I have reviewed this paper in detail and designed a **controller and decoder** inspired by its architecture. These circuits generate the necessary control, timing, and input signals for the CIM array and are described comprehensively in Chapter 5. The design ensures correct sequencing of write, compute, and read phases, enabling efficient matrix-vector multiplication using the proposed memory-centric approach.

Chapter 3

SPAR-2 Architecture

3.1 Introduction

The rapid migration of machine learning (ML) workloads from centralized data centers to distributed IoT edge devices has created a demand for high-performance yet resource-efficient hardware. Traditional CMOS-based accelerators such as GPUs and ASICs offer high throughput, but they are not always well-suited for FPGA-based edge computing, where lower clock frequencies and limited gate densities restrict performance. A key bottleneck in ML inference is the time spent waiting for weights and intermediate data to move between memory and compute units—sometimes accounting for more than half of the overall latency.

To overcome these challenges, researchers have explored *Processing-in-Memory (PIM)* and *Computational RAM (C-RAM)* architectures, which tightly couple processing elements with memory to reduce communication overhead. Building on this concept, **SPAR-2** (SIMD Processor Array) is introduced as a second-generation programmable overlay architecture that leverages the heterogeneous resources of modern FPGAs to efficiently accelerate diverse ML models including MLPs, RNNs, and CNNs. Unlike fixed-function accelerators, SPAR-2 is programmable, allowing developers to map multiple classes of ML algorithms without redesigning the hardware.

3.2 SPAR-2 Architecture

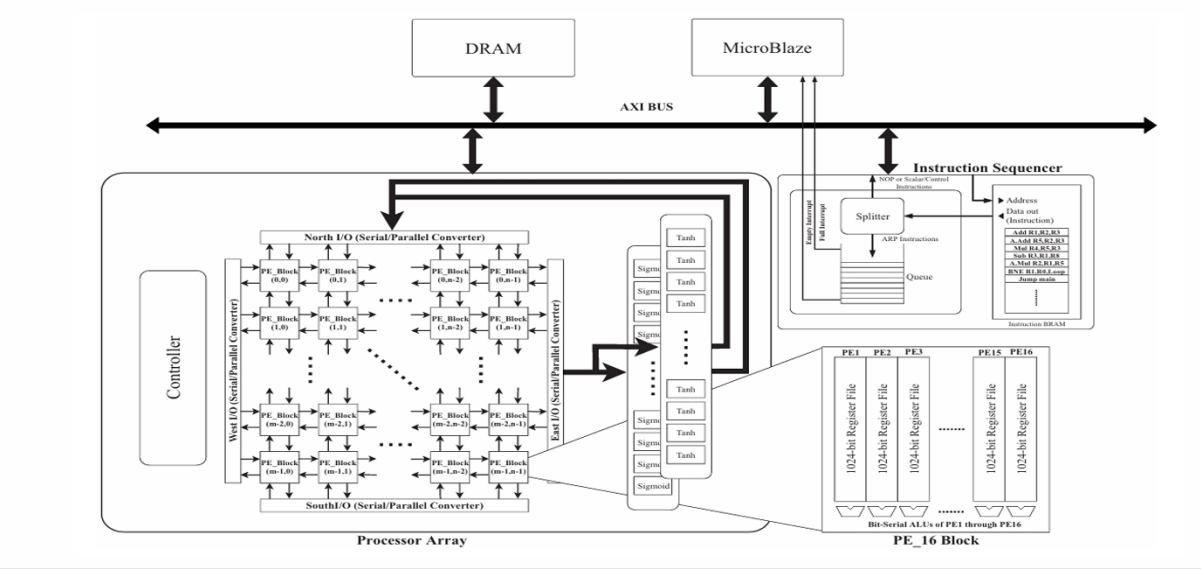

Figure 3.1: SPAR Architecture

SPAR-2 is designed as a *SIMD (Single Instruction, Multiple Data)* processor array overlay for FPGAs, aimed at reducing inference latency and improving on-chip resource utilization. Unlike fixed-function accelerators, SPAR-2 provides a flexible, programmable infrastructure that supports a wide range of machine learning models such as MLPs, RNNs, and CNNs. The architecture is built around three key components:

Processing-in-Memory (PIM) Tiles

The core building block of SPAR-2 is the *Processing-in-Memory (PIM) tile*, which integrates computation and memory into a single unit. The main characteristics are:

- Each PIM tile consists of a 4×4 array of Processing Elements (PEs), i.e., 16 PEs per tile.
- PEs are *bit-serial ALUs*, optimized for dense packing on FPGAs.
- Each PE has a local register file implemented using distributed Block RAMs (BRAMs).
- Data is stored in a **column-major format**, enabling all 16 PEs to access operands concurrently from a single BRAM.
- Supports multiple operand bit-widths (4, 8, 16, 32-bit). Lower precision increases the number of registers per PE, improving storage efficiency.
- The column-major mapping allows concurrent read/write across PEs without memory access conflicts.
- PIM tiles act as scalable units that can be combined to form large 2-D arrays of PEs.

SIMD Execution Model

The SPAR-2 overlay adopts the SIMD paradigm, where a single instruction is broadcast across many PEs. Supported operations include:

- Arithmetic: addition, subtraction, and multiplication.
- Data movement: directional moves (`move N`, `move S`, `move E`, `move W`) across the array.
- High-level macros: matrix-vector multiplication, vector addition, and element-wise multiplication.

This instruction set is optimized for machine learning workloads, allowing efficient execution of linear algebra operations common to neural networks.

Support for Machine Learning Functions

SPAR-2 includes efficient implementations of non-linear activation functions, such as *sigmoid* and *tanh*, using low-cost approximation techniques. These functions are placed at the array boundaries and operate in parallel with the PEs. The system also features:

- **Controller:** A finite state machine that decodes SPAR-2 or MicroBlaze instructions and generates control signals for ALUs, BRAMs, and activation units.
- **I/O Buffers:** Parallel-to-serial and serial-to-parallel converters at the array edges that manage data movement between external modules and the PEs.

3.2.1 Matrix-Vector Multiplication in SPAR-2

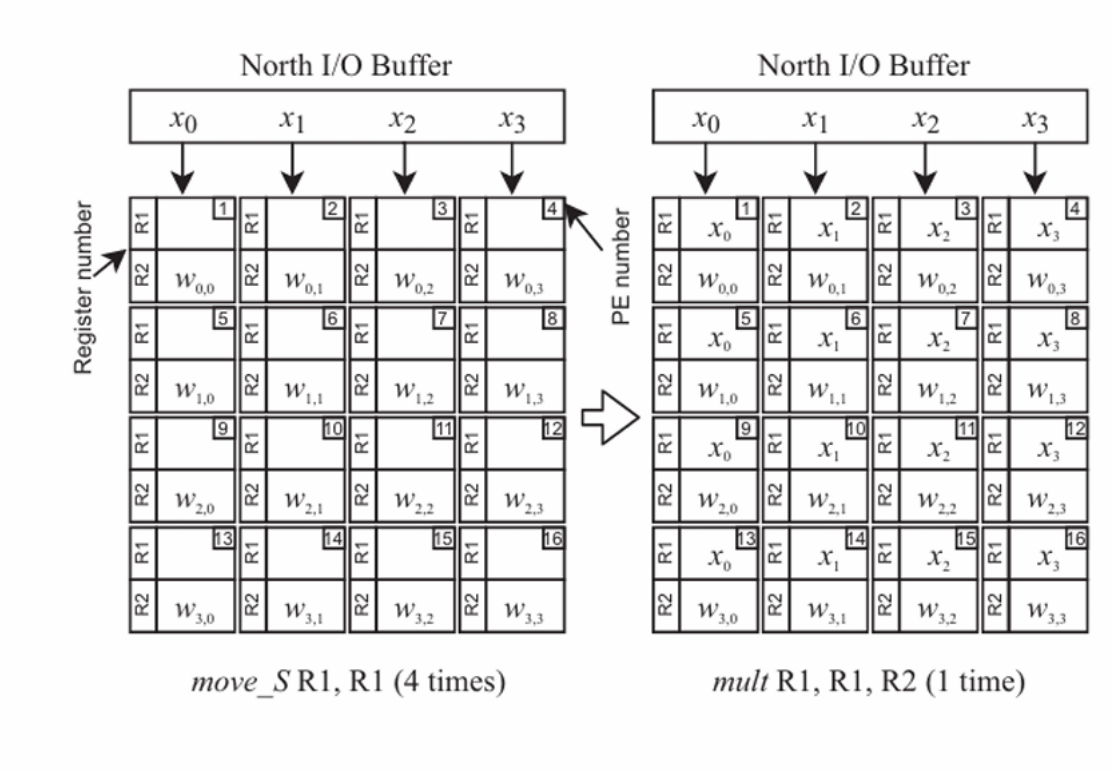

Figure 3.2: Matrix Multiplication step 1 ,2

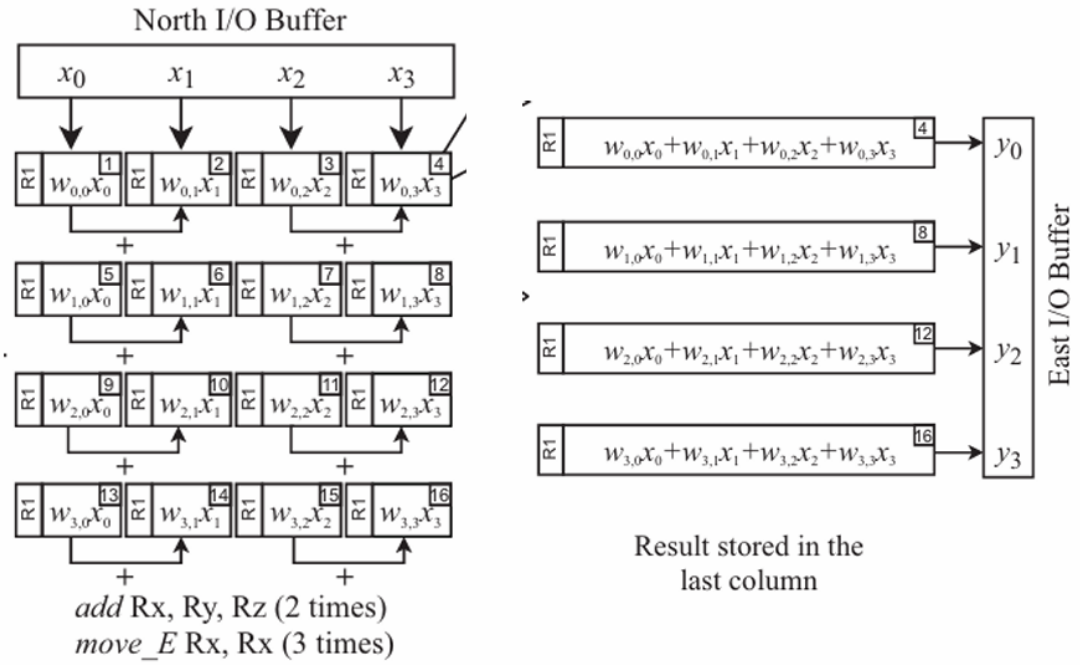

Figure 3.3: Matrix multiplication step 3,4

Matrix-vector multiplication is a fundamental operation in many machine learning workloads, and SPAR-2 is optimized to execute it efficiently. The process is structured as follows:

- **Data Mapping:** The weight matrix W is distributed across the BRAMs of the 2-D PE array. The input vector X is loaded into the first column of PEs and replicated along the columns using the `move S` instruction.
- **Parallel Multiplication:** Each PE performs bit-serial multiplication between an element of W and the corresponding element of X. All partial products are generated simultaneously across the array in a single SIMD instruction.
- **Partial Sum Reduction:** The partial products are accumulated using a binary reduction tree.
  - Additions are performed with the `add` instruction.
  - Intermediate results are shifted across PEs using `move E` or `move W`.
- **Latency Considerations:**
  - For an n-element vector, reduction requires $\log_2(n)$ addition steps and $(n-1)$ move operations.
  - Each addition takes $2m$ cycles and each move takes m cycles, where m is the bit-width.
- **Output Storage:** The final accumulated result of each row of $W \times X$ is stored in the output buffer located at the array boundary. These results are then available for activation functions or subsequent computations.

This mapping strategy allows SPAR-2 to exploit its large number of PEs and concurrent memory access, significantly reducing the latency of matrix-vector multiplication compared to traditional FPGA-based implementations.

$$\text{WxX} = \text{Y} \Rightarrow \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & w_{0,3} \\ w_{1,0} & w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,0} & w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,0} & w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} w_{0,0}x_0+w_{0,1}x_1+w_{0,2}x_2+w_{0,3}x_3 \\ w_{1,0}x_0+w_{1,1}x_1+w_{1,2}x_2+w_{1,3}x_3 \\ w_{2,0}x_0+w_{2,1}x_1+w_{2,2}x_2+w_{2,3}x_3 \\ w_{3,0}x_0+w_{3,1}x_1+w_{3,2}x_2+w_{3,3}x_3 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Figure 3.4: Matrix Multiplication

3.3 Implementation on Pynq Board

3.3.1 Matrix-Vector Multiplication Flow

The matrix-vector multiplication was carried out using a 4×4 processing element (PE) array. The steps of computation are as follows:

- **Weight Initialization:** The 4×4 weight matrix

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

  is written into the PEs before the start of computation.

- **Input Vector Feeding:** The input vector

$$\begin{bmatrix} 4 \\ 3 \\ 1 \\ 2 \end{bmatrix}$$

  is passed from the North IO buffer into the PE array. Each element of the vector is supplied column-wise in synchronization with the controller.

- **Computation:** Each PE multiplies the stored weight with the corresponding input element. The partial results are passed across the array, and additions are performed as directed by the controller.

- **Final Output:** After completion of all multiply and add operations, the final output vector obtained is:

$$\begin{bmatrix} 1F \\ 0A \\ 0A \\ 0A \end{bmatrix}$$

  where values are represented in hexadecimal format.

3.3.2 RTL design in Vivado

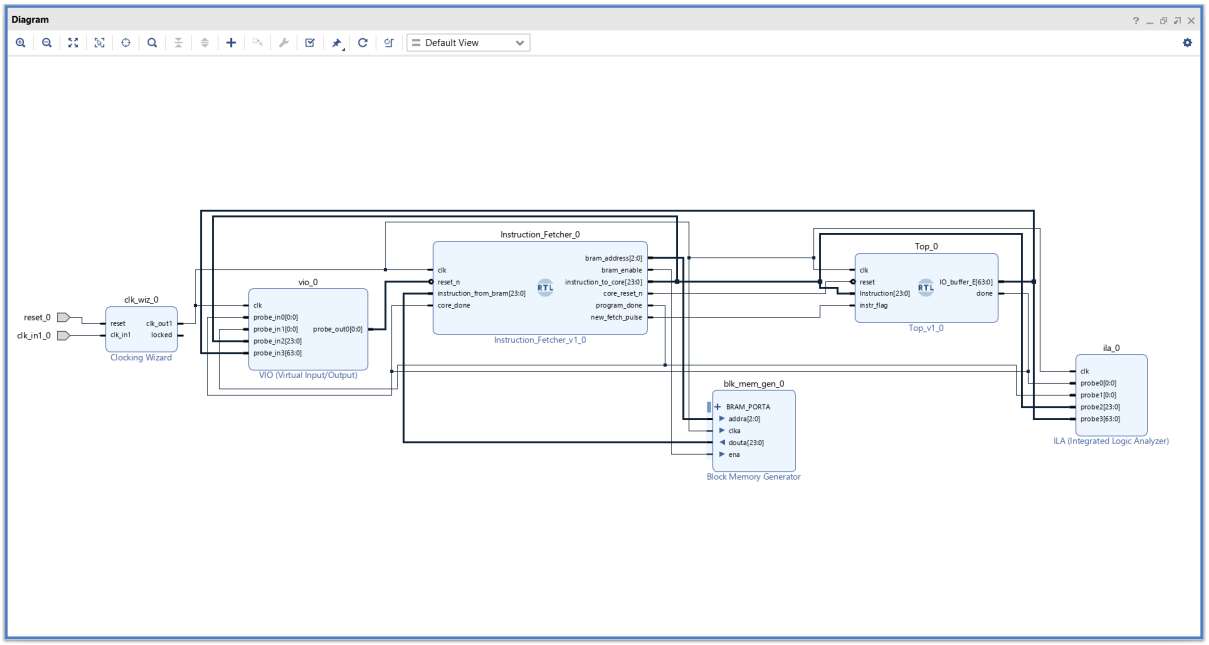

Figure 3.5: Block Diagram

Instruction Fetch and Execution Flow

The system's operation is controlled by the `Instruction_Fetcher_0` module, which coordinates instruction delivery from the memory to the computation core. The execution flow follows these steps:

- The `Instruction_Fetcher_0` module generates an address on its `bram_address` port to read from the `blk_mem_gen_0` (BRAM).
- The BRAM places the corresponding instruction onto its output port, which is read by the fetcher via the `instruction_from_bram` input.
- The `Instruction_Fetcher_0` then passes this instruction to the `Top_0` module (the matrix computation core) using the `instruction_to_core` bus.
- The `Top_0` module executes the received instruction. Once it completes the computation for that single instruction, it asserts the `done` signal high.
- The `Instruction_Fetcher_0` monitors this `done`. When it detects the high signal, proceeds to fetch the next instruction from the BRAM.
- This fetch-execute loop repeats for all instructions. When the entire sequence of instructions (the program) is finished, the `Instruction_Fetcher_0` module asserts the final `program_done` signal.

3.4 Top module description

3.4.1 Verilog Modules

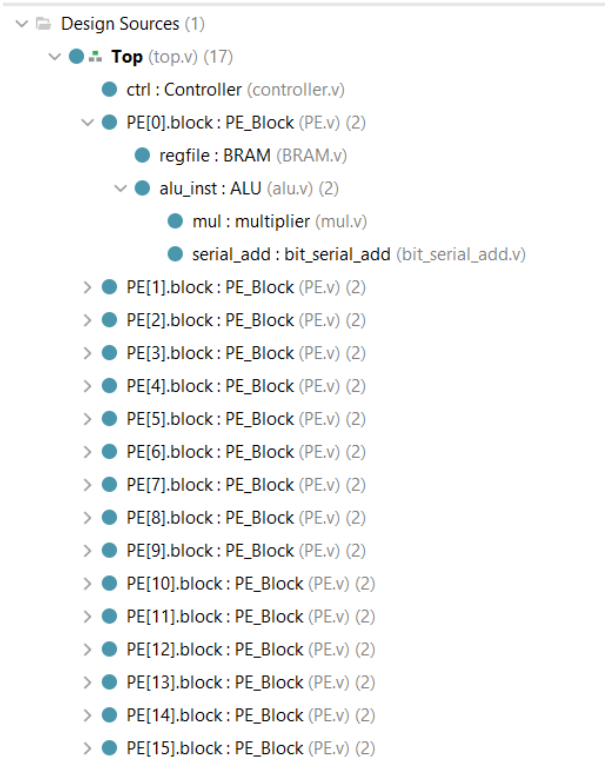

Figure 3.6: Module Hirarchy

The top-level module of the design is `Top` (`top.v`), which integrates all submodules required for computation and control. The hierarchy consists of three major components — the **Controller**, multiple **Processing Elements (PEs)**, and within each PE, local computational and storage modules.

Top Module (`top.v`)

The `Top` module serves as the system-level integration point. It connects:

- The **Controller** (`ctrl`)
- An array of **Processing Elements** (`PE[0]` to `PE[15]`)

forming a 16-PE parallel architecture. It manages the overall data flow, instruction broadcast, and synchronization among all PEs.

Controller (`controller.v`)

The `Controller` module is responsible for:

- Decoding incoming instructions
- Generating control signals for all PEs
- Managing operation sequencing such as read, write, computation (ALU operations), and data transfers

This module acts as the "brain" of the system, directing how each PE executes its assigned operation.

Processing Element (PE) Array (`PE.v`)

Each PE (`PE[0]` to `PE[15]`) represents a self-contained computation block. Inside each **PE_Block**, the following submodules are instantiated:

(a) Register File (BRAM) — `regfile : BRAM (BRAM.v)`

- Acts as local memory for the PE
- Stores input operands, intermediate results, and output data
- Implemented using on-chip **Block RAM** for efficient access

(b) Arithmetic Logic Unit (ALU) — `alu_inst : ALU (alu.v)`

- Performs all arithmetic and logic operations for the PE
- Internally composed of two key submodules:
  - **Multiplier** (`mul : multiplier (mul.v)`) — Implements a **Booth's multiplier** for signed multiplication, ensuring efficient computation of partial products
  - **Bit-Serial Adder** (`serial_add : bit_serial_add (bit_serial_add.v)`) — Performs addition in a bit-serial manner, reducing hardware complexity while maintaining functional accuracy

Each PE thus combines local storage and arithmetic capability, enabling distributed computation and parallel processing.

3.4.2 Radix-2 Booth's Multiplication Algorithm

Booth's algorithm is an efficient technique for multiplying signed binary numbers. The Radix-2 version examines one bit of the multiplier at a time along with an additional bookkeeping bit. It reduces the number of addition and subtraction operations when there are consecutive 1s in the multiplier.

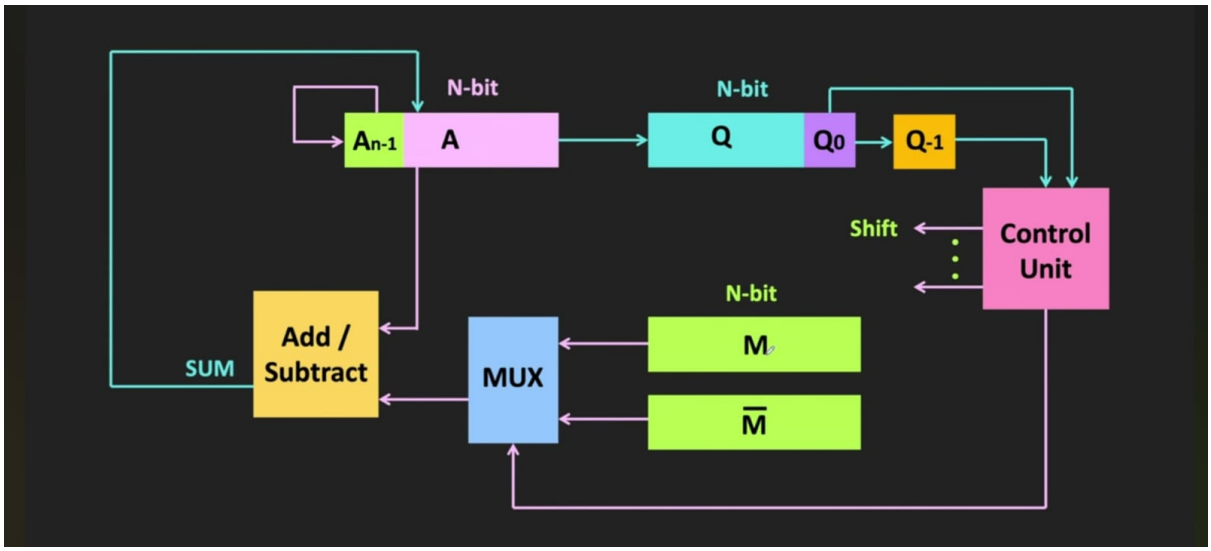

Figure 3.7: Block Diagram

Algorithm Overview:

1. Initialize the accumulator $A = 0$, the multiplier register Q with the multiplier, and $Q_{-1} = 0$ (extra bit for Booth decision). Let M be the multiplicand. The bit-width is n.

2. For n iterations, do:

   (a) Examine the two least significant bits $\{Q_0, Q_{-1}\}$:
   - 01: $A \leftarrow A + M$ (add multiplicand)
   - 10: $A \leftarrow A - M$ (subtract multiplicand)
   - 00 or 11: A unchanged (no operation)

   (b) Perform an arithmetic right shift of the combined register $\{A, Q, Q_{-1}\}$ by 1 bit. The MSB of A is replicated to preserve the sign.

   (c) Repeat for the next bit.

3. After n iterations, the concatenation $\{A, Q\}$ holds the $2n$-bit signed product.

Key Points:

- Radix-2 handles one multiplier bit per iteration, requiring n cycles for n-bit operands.
- The extra bit Q_{-1} allows detection of transitions in the multiplier, enabling optimized addition/subtraction.
- Arithmetic right shifts preserve the sign of negative numbers in two's complement representation.

Example: Multiplying 3×5:

- Initialize: $A = 0$, $Q = 00000101$, $Q_{-1} = 0$, $M = 00000011$.
- Perform 8 iterations (for 8-bit operands) according to the above rules.
- Final product: $\{A, Q\} = 0000000000001111 = 15$.

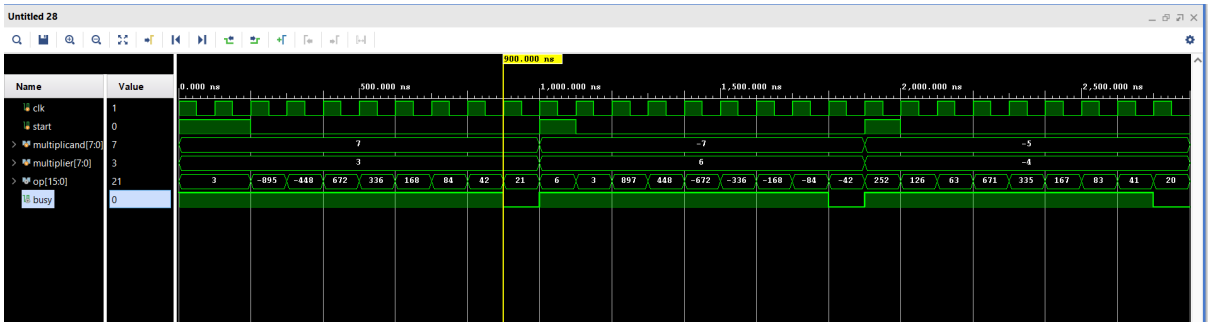

Figure 3.8: Waveform

Chapter 4

Results

4.1 Using BRAMs

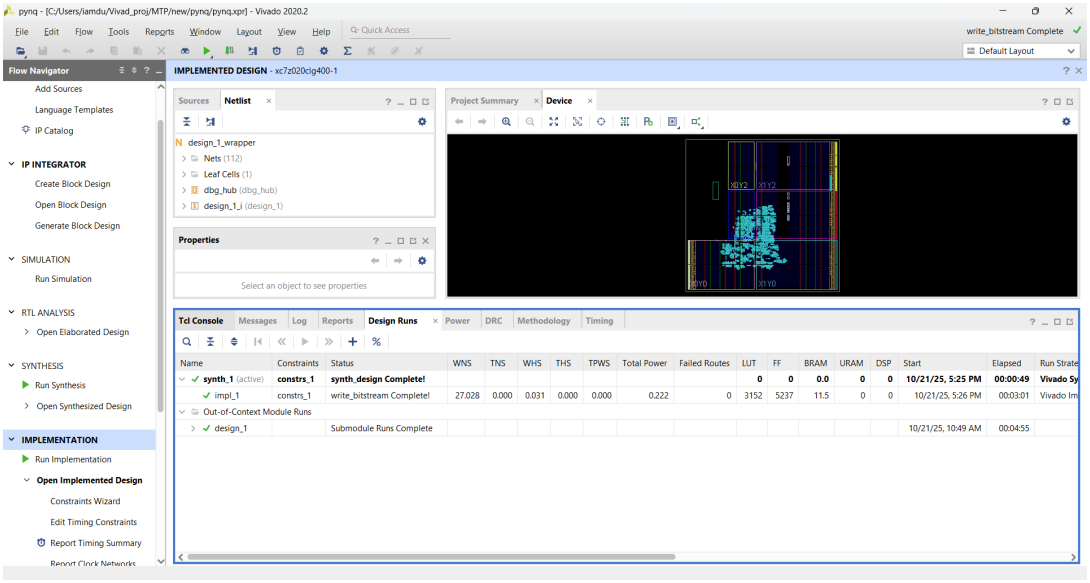

Figure 4.1: Resource utilisation

Implementation and Resource Utilization

The design was successfully implemented for the Zynq-7000 target device, `xc7z020clg400-1`. The post-implementation timing results are strong, with no setup or hold violations, indicated by a positive Worst Negative Slack (WNS) of +27.028ns.

The resource utilization for the `impl_1` run is as follows:

- **LUT (Look-Up Tables):** 3152
- **FF (Flip-Flops):** 5237
- **BRAM (Block RAM):** 11.5
- **DSP Slices:** 0
- **Total Power (Estimated):** 0.222W

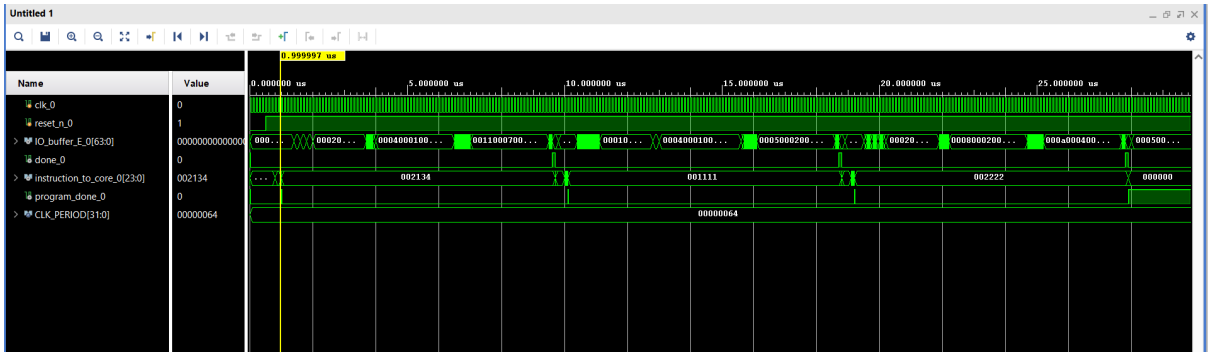

Figure 4.2: Post Implementation timing waveform

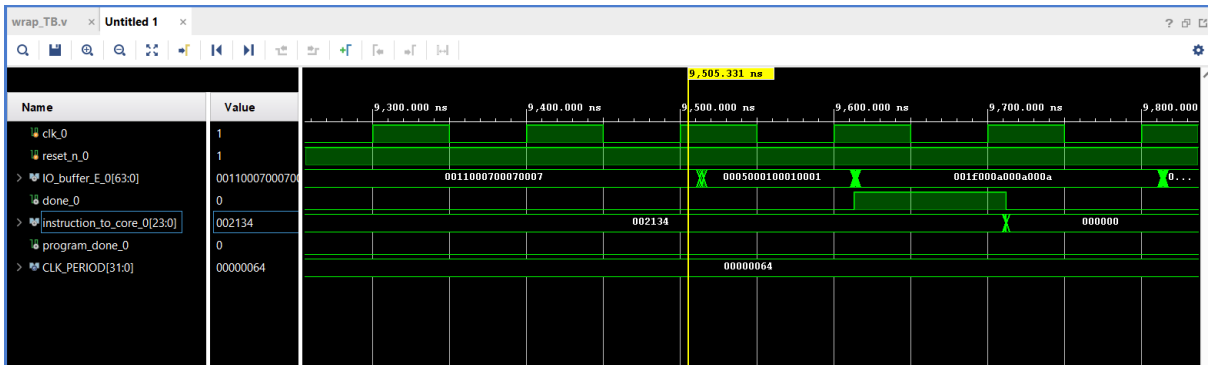

Figure 4.3: Zoomed in

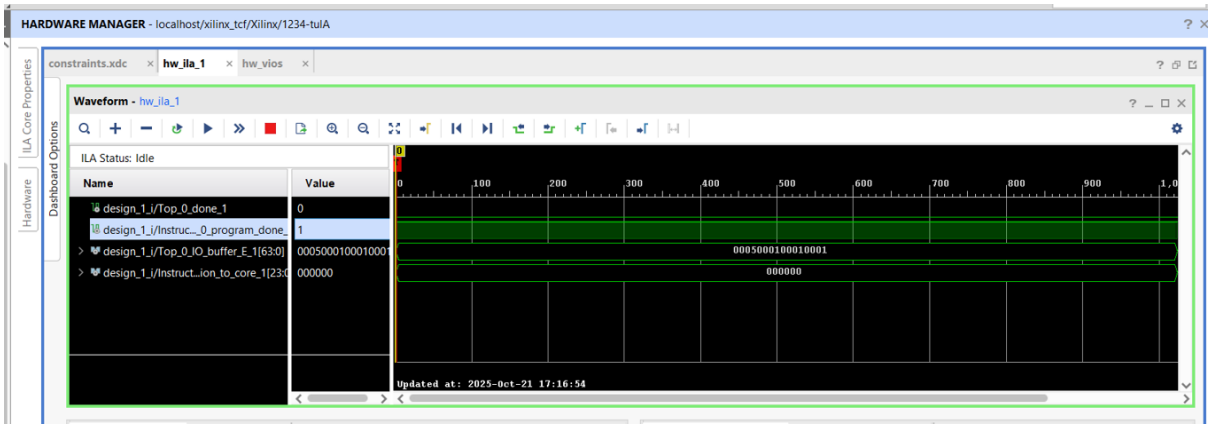

Figure 4.4: On pynq board

4.2 Using LUTs and Flip Flops

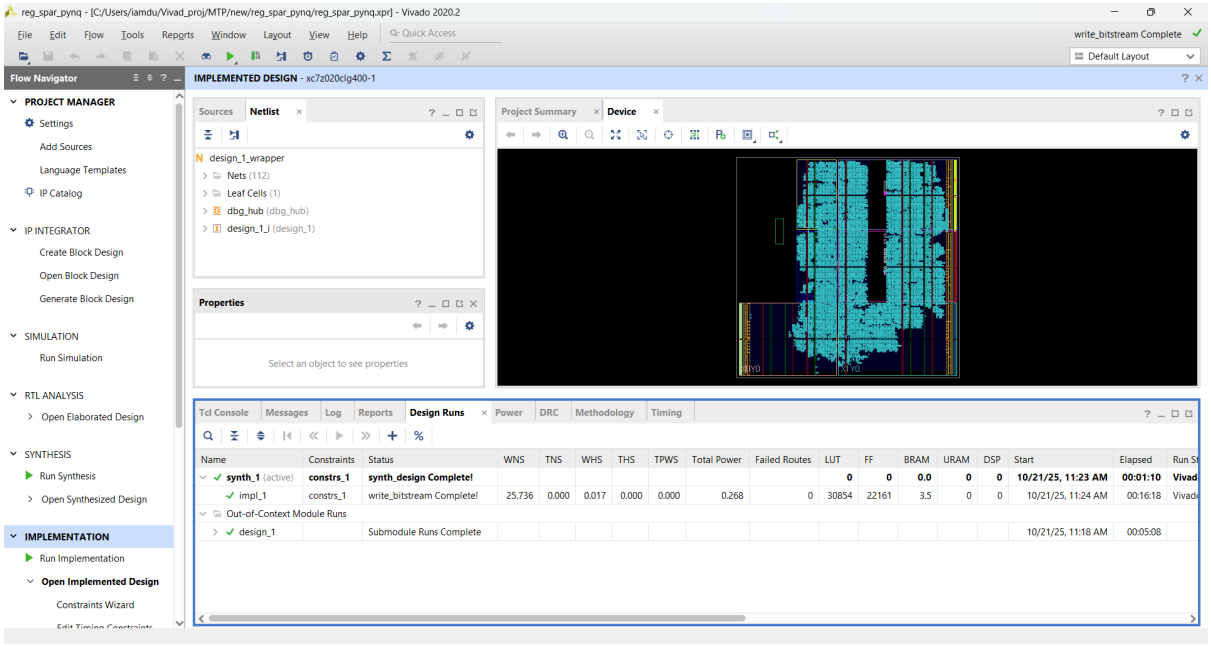

Figure 4.5: Resource utilisation

Implementation Results and Resource Utilization

The design's `impl_1` run has successfully completed implementation, and the bitstream has been generated for the `xc7z020clg400-1` device. All timing constraints have been met, which is confirmed by a positive Worst Negative Slack (WNS) of +25.736ns and a Worst Hold Slack (WHS) of +0.017ns.

The post-implementation resource utilization is as follows:

- **LUT (Look-Up Tables):** 30854 (58.0%)
- **FF (Flip-Flops):** 22161 (20.8%)
- **BRAM (Block RAM):** 3.5 (2.5%)
- **DSP Slices:** 0 (0.0%)
- **Total Power (Estimated):** 0.268W

4.3 Comparision

Feature	Using BRAM	Using Register File (Flip-Flops)
Area	**Very Low.** Uses the 10.5 dedicated BRAM blocks.	**Extremely High.** Will use thousands of LUTs and Flip-Flops.
Speed	Fast, but has a fixed read latency.	Can be slightly faster, as it's built from general-purpose logic.
Power	**Low.** BRAMs are highly optimized for power.	**High.** A large number of active flip-flops consumes much more power.
Scalability	**Excellent.** You can build very large memories.	**Poor.** You will quickly run out of logic resources on the FPGA.

Figure 4.6: Comparision

Storage Implementation Observation

From the implementation results, it is evident that utilizing the dedicated on-chip **BRAM** (Block RAM) is a highly efficient design choice. This approach conserves a significant amount of logic resources (LUTs and FFs) compared to synthesizing the memory as a large register file, directly contributing to a more optimized and scalable design.

4.4 Future Work

The current design serves as a foundational step for a more advanced architecture. The next phase of development will focus on transitioning from conventional memory to an in-memory computation paradigm.

- **Current Design:** The architecture presently uses on-chip **Block RAM (BRAM)** for storing weights and intermediate data.
- **Future Goal:** The next phase of development will replace BRAM with **Resistive RAM (ReRAM)** to enable true **in-memory matrix computation**, performing calculations directly within the memory fabric.

Chapter 5

Computing In Memory

5.1 Introduction

In this work, a novel three-dimensional (3D) monolithic integrated Compute-In-Memory system, termed **3D MIGCIM**, is proposed. The architecture employs an **Indium Tin Oxide (ITO)** based 4T2C embedded DRAM (eDRAM) gain-cell array integrated in the back-end-of-line (BEOL) over a **2 nm Gate-All-Around (GAA) CMOS** logic layer in the front-end-of-line (FEOL). This co-integration achieves high compute density and ultra-low energy consumption.

The proposed **4T2C eDRAM CIM cell** enables mixed-signal MAC operations using ternary weights $(-1, 0, +1)$ and binary inputs, while peripheral CMOS logic performs efficient read/write control and accumulation. A material-device-circuit co-design methodology is adopted to optimize the ITO transistor characteristics, ensuring compatibility with scaled CMOS technology and robust circuit-level performance. Post-layout simulations of a 64×128 CIM macro demonstrate an exceptional energy efficiency of 280.52 TOPS/W and compute density of 70.96 TOPS/mm^2, showcasing the potential of monolithic 3D ITO-CMOS integration for next-generation AI accelerators.

5.2 ITO eDRAM CIM Macro Architecture

Figure below illustrates the proposed **ITO eDRAM Compute-In-Memory (CIM) architecture**, which performs parallel multiply-and-accumulate (MAC) operations for neural network workloads. The left portion of the figure represents a fully-connected neural network (FCNN) layer, while the right side shows its hardware mapping onto the ITO-based 4T2C eDRAM CIM macro. Each neuron output $Z[i]$ corresponds to the dot product between an input vector $X[j]$ and a stored ternary weight matrix W_{ij}.

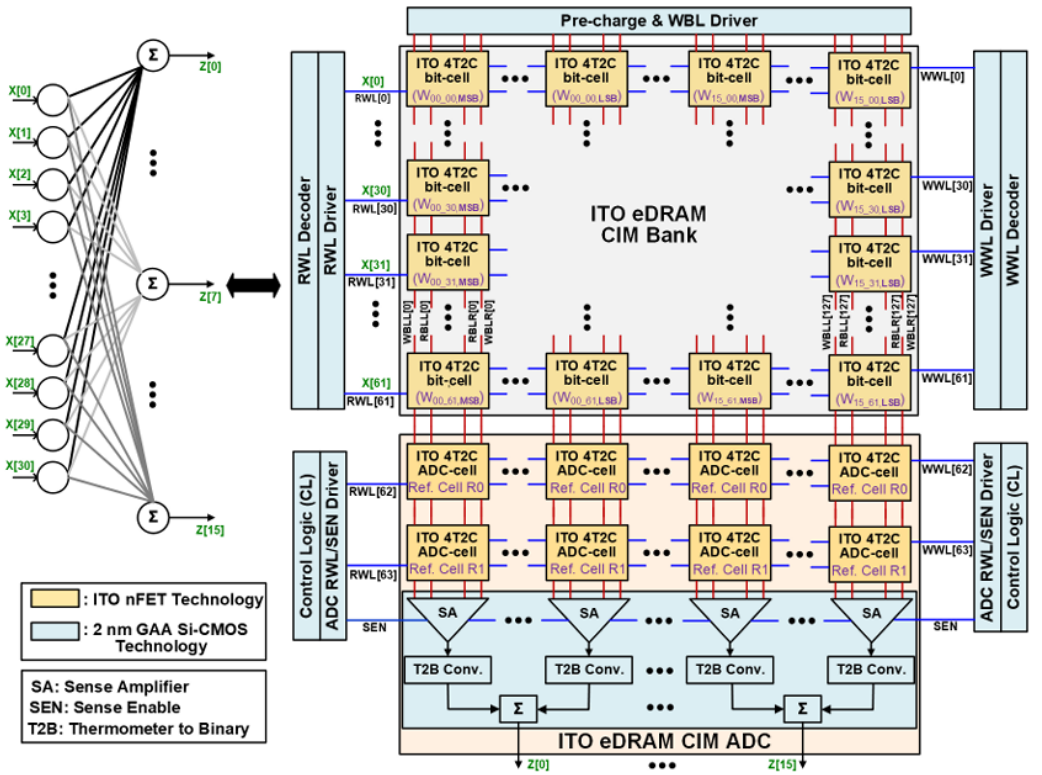

Fig. 16. Translation of a fully connected neural network layer (left) multiply and accumulate (MAC) operation to a equiavlent ITO eDRAM CIM macro (right) and peripheral logic to perform 8b×8b operation between input X[i] and stored ternary weight $W_{i,j}$ (i: variable for outputs; j: variable for inputs) distributed across 8 columns.

Figure 5.1: Translation of a fully-connected neural network layer to an equivalent ITO eDRAM CIM macro with peripheral decoders and control logic performing 8b×8b MAC operations.

5.2.1 Architecture Overview

The proposed CIM macro consists of two main components:

1. **ITO 4T2C eDRAM CIM Bank** — fabricated in the Back-End-of-Line (BEOL) using Indium Tin Oxide (ITO) nFETs.
2. **Peripheral CMOS Logic** — implemented in 2 nm Gate-All-Around (GAA) Si-CMOS technology and integrated in the Front-End-of-Line (FEOL).

This 3D monolithic integration allows the compute array and all decoders, sense amplifiers, and control logic to be vertically stacked, significantly improving compute density and reducing interconnect parasitics.

5.2.2 Functional Blocks

- **4T2C eDRAM Bitcell:** Each bitcell stores a ternary weight (-1, 0, or $+1$) using charge on two storage capacitors. The cell performs a binary input (0 or 1) and ternary weight multiplication through voltage modulation on the bitlines.
- **WWL and RWL Decoders:** The *Write Wordline (WWL)* decoder activates selected rows for writing ternary weights into the array. The *Read Wordline (RWL)* decoder applies binary inputs $X[j]$ as voltage pulses during computation.
- **Pre-Charge and WBL Drivers:** These circuits initialize the bitlines before each compute cycle by charging them to a reference voltage.
- **Column ADC:** Each column includes two dedicated ADC reference cells and a Sense Amplifier (SA). The SA detects the voltage difference (ΔV) between the left and right bitlines (RBLL, RBLR), which corresponds to the analog sum of products across a column.
- **Thermometer-to-Binary (T2B) Converter:** The SA output generates a 32-bit thermometer code, which the T2B converter translates into a compact binary output for digital processing.
- **Control Logic (FSM):** A finite-state machine coordinates all operations — writing weights, applying inputs, enabling sense amplifiers, and accumulating partial results using adder and shifter logic.

5.2.3 Operation Principle

During inference, binary input pulses ($X[j]$) are applied through the RWL decoder. Depending on the stored ternary weight in each 4T2C cell:

- If $W_{ij} = +1$, the right bitline (RBLR) discharges.
- If $W_{ij} = -1$, the left bitline (RBLL) discharges.
- If $W_{ij} = 0$, no discharge occurs.

The cumulative voltage difference ($V_{RBLL} - V_{RBLR}$) across a column represents the analog **MAC sum** for that column. This analog value is then digitized by the ADC and combined across columns by the peripheral adder logic to generate the final neuron outputs $Z[i]$.

5.2.4 Summary

This mixed-signal CIM macro effectively performs **8b×8b multiply-and-accumulate operations** in parallel across 128 columns and 64 rows, achieving high energy efficiency and computational throughput. By stacking the ITO eDRAM array above CMOS logic, the design provides a compact, low-power solution for edge AI and neural network acceleration.

5.3 MAC Operation

5.3.1 MAC Operation with Positive Weights

When a **positive weight** ($W_{ij} = +1$) is stored in the 4T2C eDRAM cell, the right storage node (V_{SNR}) holds a high voltage ('H'), while the left storage node (V_{SNL}) is kept low ('L'). During computation, an active-low pulse applied on the corresponding *Read Word Line (RWL)* enables the access transistors in the bitcell.

As a result, the **right bitline (RBLR)** begins to discharge through the conducting ITO transistor, whereas the **left bitline (RBLL)** remains unchanged. The amount of discharge (ΔV) depends on the stored voltage at V_{SNR} and the access transistor strength.

Across multiple cells in the same column, these discharges accumulate on the column bitline capacitance. Therefore, for all cells storing positive weights and receiving an active input pulse, the RBLR voltage decreases proportionally, representing a cumulative positive contribution to the overall dot product.

The voltage differential between the two bitlines,

$$\Delta V = (V_{RBLL} - V_{RBLR}),$$

is then sensed by the *Sense Amplifier (SA)*. For positive-weight-dominant columns, this ΔV is negative, and the SA output becomes logic '1', indicating a positive sum contribution. The *Thermometer-to-Binary (T2B)* converter then encodes the SA output over multiple ADC cycles into a binary value corresponding to the total number of positive weight activations in that column.

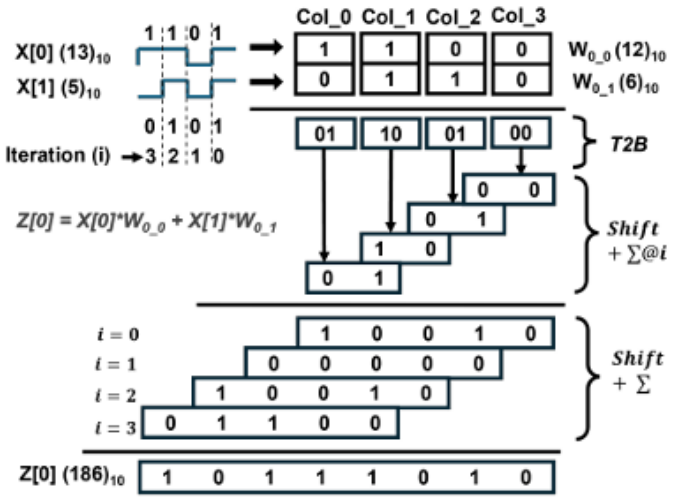

Figure 5.2: MAC

5.3.2 MAC Operation with Negative Weights

In contrast, when a **negative weight** ($W_{ij} = -1$) is stored, the left storage node (V_{SNL}) is written high ('H'), while the right storage node (V_{SNR}) is written low ('L'). During the active-low RWL pulse, the **left bitline (RBLL)** discharges through the active ITO transistor, while the right bitline remains unchanged.

This causes the voltage difference between the bitlines to become positive, i.e.,

$$\Delta V = (V_{RBLL} - V_{RBLR}) > 0,$$

which corresponds to a **negative contribution** to the MAC result. The Sense Amplifier detects this polarity reversal and outputs a logic '0', indicating that more negative weights have been activated in the current operation.

Similar to the positive case, the analog accumulation on RBLL and RBLR represents the total weighted sum of all cells in that column. The embedded column ADC converts this analog differential voltage into a signed digital value through the SA and T2B stages. The *Control Logic* and *Adder/Shifter* units then accumulate these results across all columns to generate the neuron output $Z[i]$.

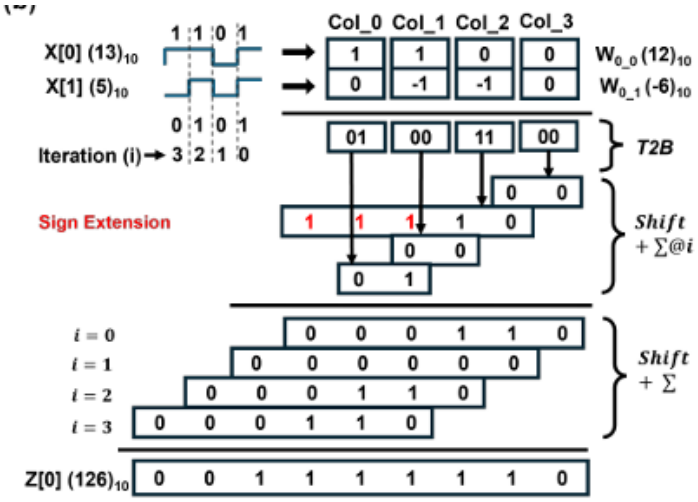

Figure 5.3: MAC

In summary, the ITO eDRAM CIM macro performs parallel **1b×ternary (1, 0, +1)** multiplications, where:

- Positive weights discharge the right bitline (+ contribution),
- Negative weights discharge the left bitline (− contribution),
- Zero weights cause no discharge (neutral contribution).

This mixed-signal accumulation mechanism enables energy-efficient analog MAC computation directly within the memory array.

5.4 Controller and Decoder Design

In the proposed architecture, the operation of the ITO eDRAM Compute-In-Memory (CIM) array is governed by a dedicated **controller** and associated **decoder circuits**, which I have designed to generate the necessary control and data signals for the CIM operation. These blocks are implemented in 2 nm Gate-All-Around (GAA) CMOS technology and are placed in the Front-End-of-Line (FEOL) layer beneath the ITO-based eDRAM array.

5.4.1 Decoder Functionality

Two primary decoders are used to access the CIM array:

- **Write Wordline (WWL) Decoder:** The WWL decoder activates a specific row in the array during the *write phase* to program the ternary weights (-1, 0, $+1$) into the 4T2C bitcells. Each decoder output drives the WWL signal corresponding to one row, enabling the storage nodes (V_{SNL}, V_{SNR}) to be charged or discharged via the write bitlines (WBLL, WBLR). This determines whether the cell stores a positive, negative, or zero weight.

- **Read Wordline (RWL) Decoder:** During the *computation phase*, the RWL decoder generates active-low pulses on selected rows corresponding to the binary input vector $X[j]$. These pulses serve as activation signals to the memory array, effectively applying the input data directly to the cells for the multiply-and-accumulate (MAC) operation. The decoder ensures that all rows receiving a logic '1' input get simultaneous RWL activation, enabling highly parallel computation across the array.

5.4.2 Controller Functionality

The **controller** coordinates all major operational stages of the CIM macro, including write, hold, read (MAC), analog-to-digital conversion (ADC), and accumulation. It is implemented as a **finite-state machine (FSM)** that generates synchronized enable, select, and timing signals for both the decoders and peripheral logic.

The sequence of operation controlled by the FSM is as follows:

1. **Write Phase:** The controller enables the WWL decoder and WBL drivers to store ternary weights in the selected cells.

2. **Hold Phase:** The controller sets the WWL and RWL lines to bias conditions that minimize leakage, ensuring long data retention.

3. **Compute (MAC) Phase:** The controller enables the RWL decoder to apply input pulses $X[j]$ to the selected rows, initiating parallel bitline discharges corresponding to the stored weights.

4. **ADC and Readout Phase:** The controller triggers the Sense Amplifier (SA), Thermometer-to-Binary (T2B) converter, and Adder/Shifter logic to digitize and accumulate the analog MAC results.

5. **Idle/Reset Phase:** Once computation is complete, all control lines are reset, and pre-charge drivers are enabled for the next computation cycle.

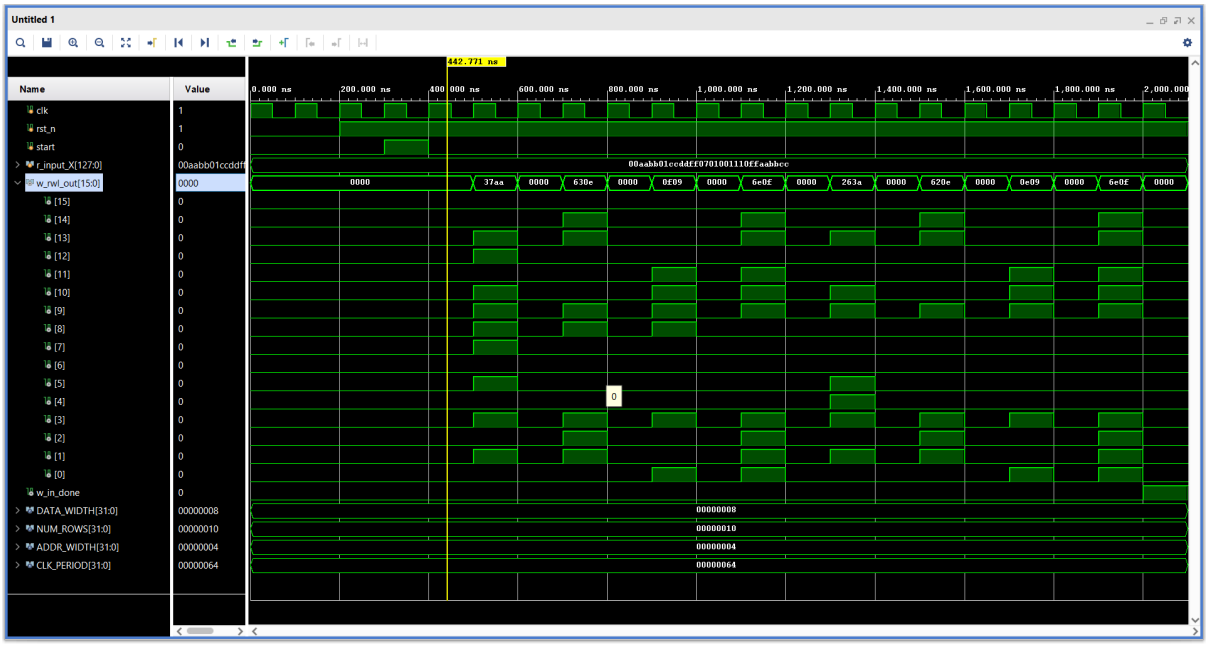

Figure 5.4: Waveform

Chapter 6

RRAM Simulation

6.1 Working Principle

Resistive Random Access Memory (RRAM) is a two-terminal non-volatile memory technology that stores data by switching between high-resistance (HRS) and low-resistance states (LRS). These states correspond to logic ‘0‘ and ‘1‘, respectively.The operation of RRAM is based on the formation and rupture of a conductive filament (CF) within a dielectric layer—typically a transition metal oxide like HfO_2.

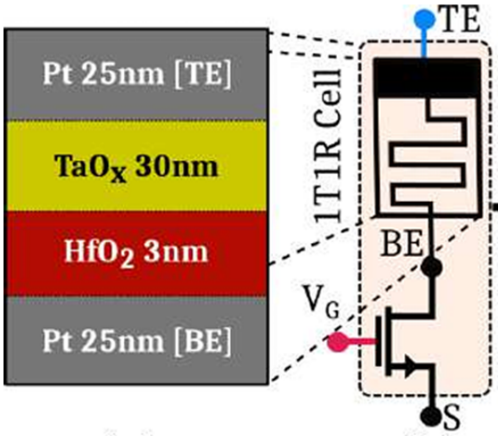

Figure 6.1: RRAM

- **SET Operation (Write ‘1‘):** A positive voltage is applied across the RRAM cell, causing oxygen vacancies or metal ions to migrate and form a conductive filament between the electrodes. This switches the device to a low-resistance state (LRS).
- **RESET Operation (Write ‘0‘):** A reverse voltage dissolves or ruptures the filament, switching the device back to a high-resistance state (HRS).
- **Read Operation:** A small voltage is applied (below the switching threshold) to sense the current and determine the resistance state of the cell.

Filament Formation

The filament typically consists of a chain of oxygen vacancies or metal atoms that span the insulating oxide layer. The switching mechanism is often governed by one of the following:

- Electrochemical Metallization (ECM)
- Valence Change Mechanism (VCM)
- Thermochemical Mechanism (TCM)

In HfO_2-based RRAM, the VCM is dominant, where oxygen ion migration causes the formation and rupture of the conductive path.

6.2 Simulation

- I have referred paper [4] and utilized the **JART memristor model**, a Verilog-A implementation, for the device-level simulation.
- Successfully simulated the core electrical I-V (current-voltage) characteristics of the RRAM cell within the **Cadence Spectre** circuit design environment.

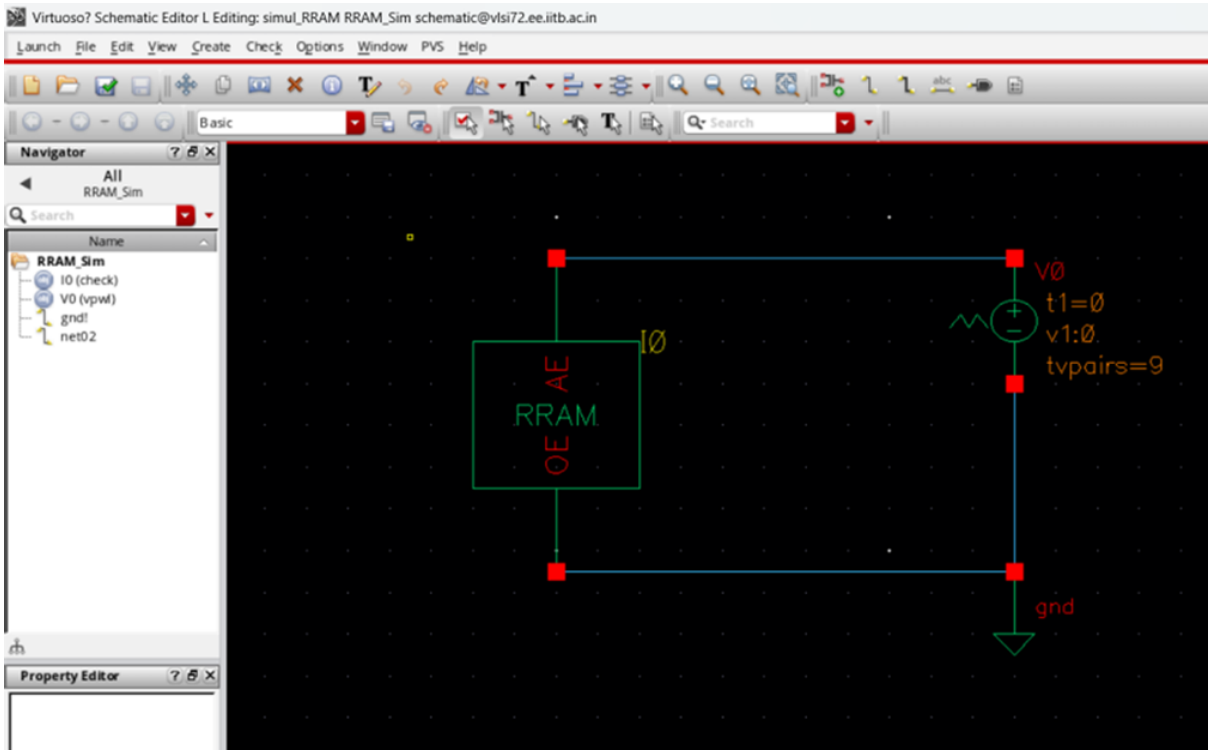

Figure 6.2: Setup

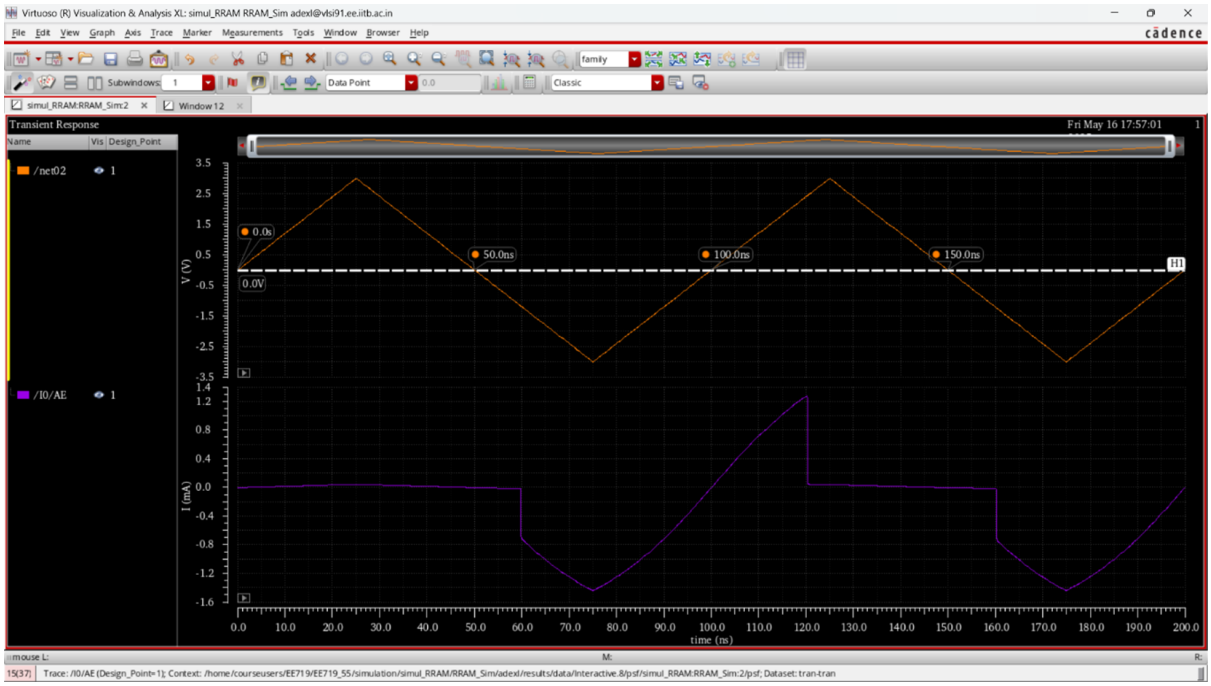

Figure 6.3: Waveform

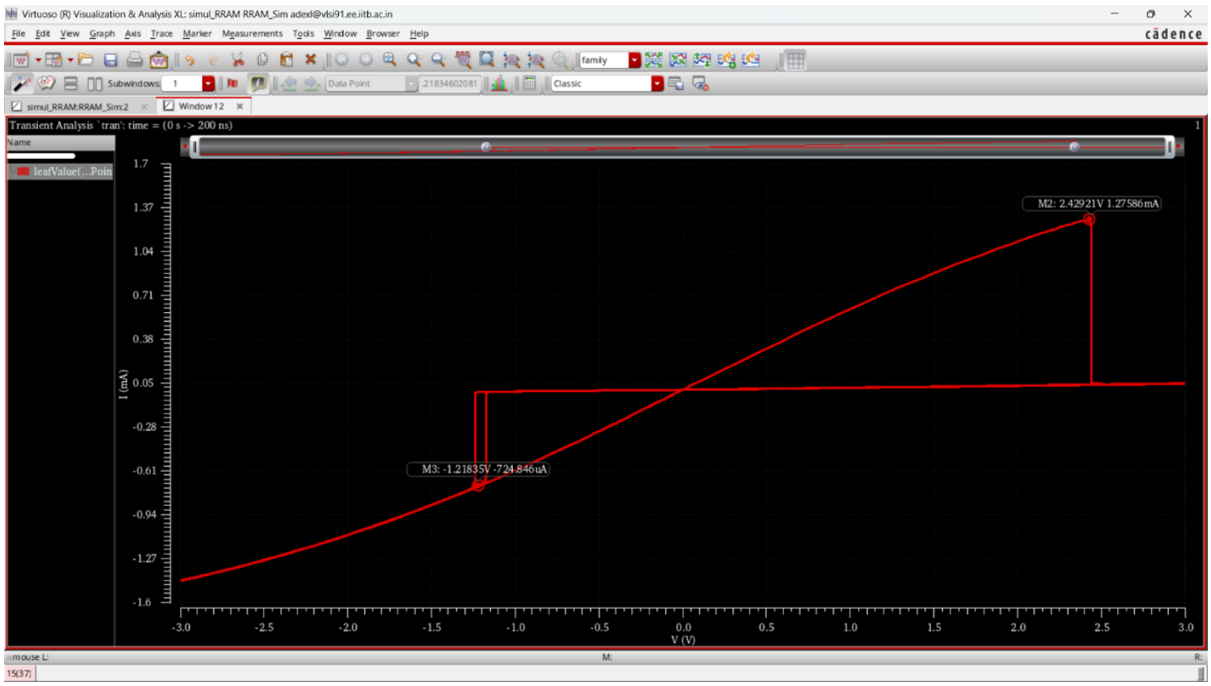

Figure 6.4: IV Characteristics

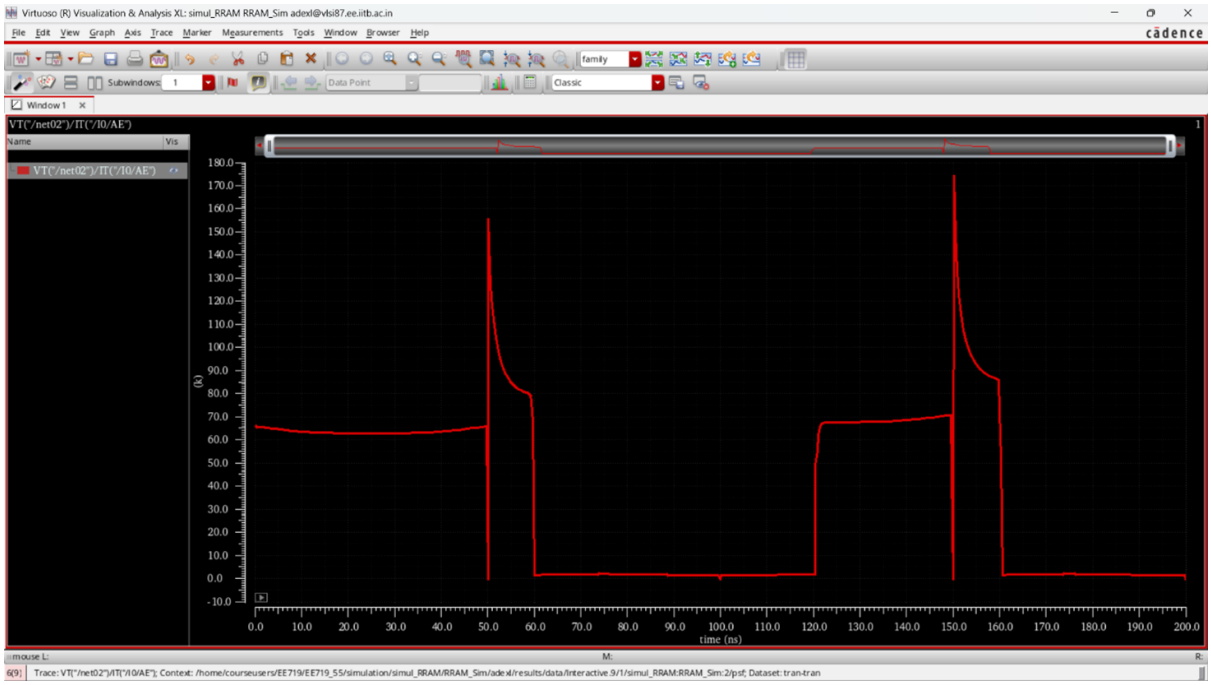

Figure 6.5: Resistance of RRAM

Chapter 7

SRAM based PIM Compiler

7.1 Introduction

Processing-in-Memory (PIM) integrates computation within memory arrays to minimize data transfers between processor and memory, overcoming the von Neumann bottleneck. **Bit-serial PIM** performs Boolean and arithmetic operations across bit positions, enabling compact and programmable computation inside SRAM or ReRAM.

Existing PIM compilers often treat the memory as a single SIMD engine, causing underutilization and communication overhead. The proposed **PIM Logic Compiler (PIMLC)** addresses this through:

- Automatic conversion of Boolean networks (AOIG, MIG, XMG) into PIM instruction streams.
- A *Workload-Resource-Aware Scheduling (WRAS)* method optimizing mapping and latency.

PIMLC achieves up to 15.55× (SRAM-PIM) and 19.03× (ReRAM-PIM) speedup, bridging high-level logic synthesis and low-level PIM execution.

7.2 Memory Organization in PIM

PIM extends standard memory hierarchy with in-situ computing capability, enabling bitwise logic operations within arrays.

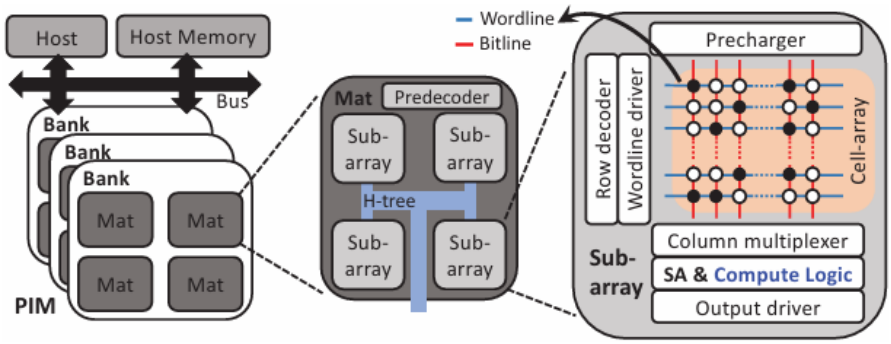

Figure 7.1: Memory organization with PIM capability.

A PIM system consists of:

- **Bank:** Highest level memory division interfacing with I/O.
- **Mat:** Contains several sub-arrays and shared peripherals.
- **Sub-array:** Smallest compute unit performing bitline logic operations, functioning as a SIMD engine.

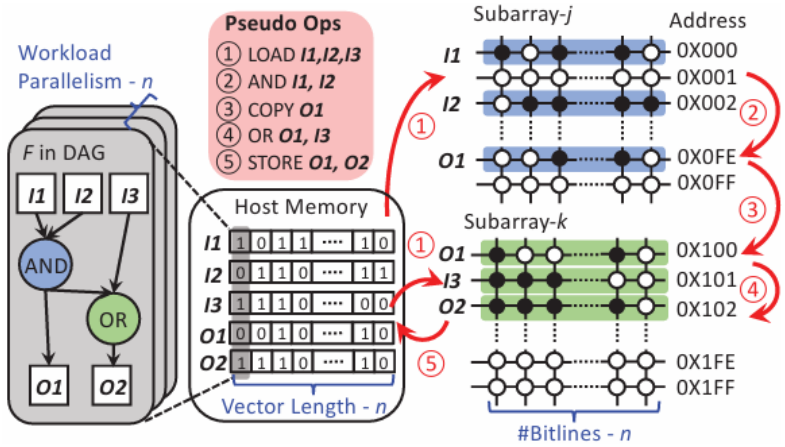

Figure 7.2: Sub-array bitline computation.

Computation occurs along bitlines where multiple rows are activated to perform logic such as AND, OR, or XOR. Results are sensed and written back to destination rows.

PIM instructions are of two types:

- **Computation:** Operate within sub-arrays (e.g., `AND, OR, XOR, MAJ`).

- **Communication:** Move data (`COPY, LOAD, STORE`) across sub-arrays or to/from host memory.

Category	Instruction Format	Opcode Set
Computation	`opcode,src1,src2,src3,dst`	{and,or,maj,xor,and3,or3,xor3,inv}
Communication	`opcode,src,dst`	{copy}
	`opcode,src,dst`	{load,store}

Figure 7.3: Simplified PIM instruction set architecture (ISA).

Operand locality is essential — all data for a computation must reside in the same sub-array. Multiple sub-arrays can work in parallel, supporting fine-grained SIMD/MIMD execution.

7.3 PIMLC Framework

PIMLC is an ahead-of-time compiler that maps Boolean functions to bit-serial PIM hardware. It consists of three key components:

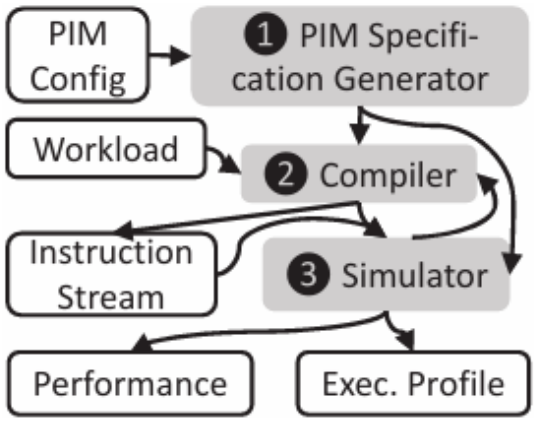

Figure 7.4: Overview of PIMLC framework.

1. **SpecGen:** Generates hardware specifications—banks, mats, sub-arrays, bitline/-wordline counts, latency, and energy models—using tools like CACTI or NVSim.

2. **Logic Compiler:** Converts Boolean DAGs (AOIG, MIG, XMG) into optimized PIM instructions using WRAS scheduling, which considers workload structure, data locality, and sub-array resources.

3. **Performance Simulator:** Estimates execution latency, energy, and throughput, providing feedback for compiler optimization.

Workflow Summary: Input Boolean function $F \rightarrow$ SpecGen generates hardware model $\rightarrow$ Compiler maps and schedules operations $\rightarrow$ Simulator evaluates performance $\rightarrow$ Output is an optimized instruction stream for SRAM/ReRAM-based PIM.

Through this co-design flow, PIMLC efficiently maps high-level logic to in-memory hardware, achieving high throughput and energy efficiency.

7.4 Compilation of Arithmetic Netlists using PIMLC

Two synthesized Verilog netlists — a 16-bit adder (`add16.v`) and an 8-bit multiplier (`mul8.v`) — were compiled using the PIMLC framework targeting an SRAM-based PIM configuration.

Adder (add16.v)

- The `add16.v` file describes a 16-bit combinational addition circuit represented entirely with two-input logic gates such as `AND`, `OR`, `XOR`, and `INV`.
- PIMLC translated the gate-level netlist into a Boolean DAG and generated an optimized instruction stream using the *Workload-Resource-Aware Scheduling (WRAS)* policy.
- The compiler efficiently mapped operations onto available SRAM sub-arrays, minimizing inter-subarray `COPY` instructions and preserving operand locality.
- The resulting bit-serial instruction sequence executed correctly under the integrated simulator, demonstrating effective scheduling and low communication overhead.

Multiplier (mul8.v)

- The `mul8.v` file implements an 8-bit combinational multiplication circuit with a larger Boolean DAG and higher logic depth due to partial-product generation.
- PIMLC automatically partitioned the workload into multiple computation stages, inserting `LOAD` and `STORE` instructions where intermediate data exceeded sub-array capacity.
- Parallel `AND` and `XOR`/`MAJ` operations were distributed across sub-arrays, ensuring balanced utilization and reduced latency.
- Simulation confirmed functional correctness and efficient execution under the SRAM configuration.

Summary

- Both designs were successfully compiled into bit-serial instruction streams tailored for the SRAM-based PIM.
- PIMLC handled variations in circuit size and complexity automatically, validating its effectiveness for logic synthesis and scheduling of arithmetic workloads.

References

[1] Suhail Basalama, Atiyehsadat Panahi, 2020,SPAR-2: A SIMD Processor Array for Machine Learning in IoT Devices

[2] https://www.techrxiv.org/users/968359/articles/1343070-3d-migcim-a-material-device-circuit-co-design-for-3d-monolithic-integrated-4t2c-ito-gain-cell-compute-in-memory-on-2-nm-gaa-cmos

[3] SukhmeetKaur, Suman,2013,Implementation of Modified Booth Algorithm (Radix 4) and its Comparison with Booth Algorithm (Radix-2).

[4] https://www.fz-juelich.de/en/pgi/pgi-7/research-groups-1/ag-menzel/jart-model

[5] User Guide for the JART VCM v1b Compact Model. Available online: `https://www.emrl.de/JART-files/User_Guide_for_the_JART_VCM_v1_Compact_Model.pdf`.