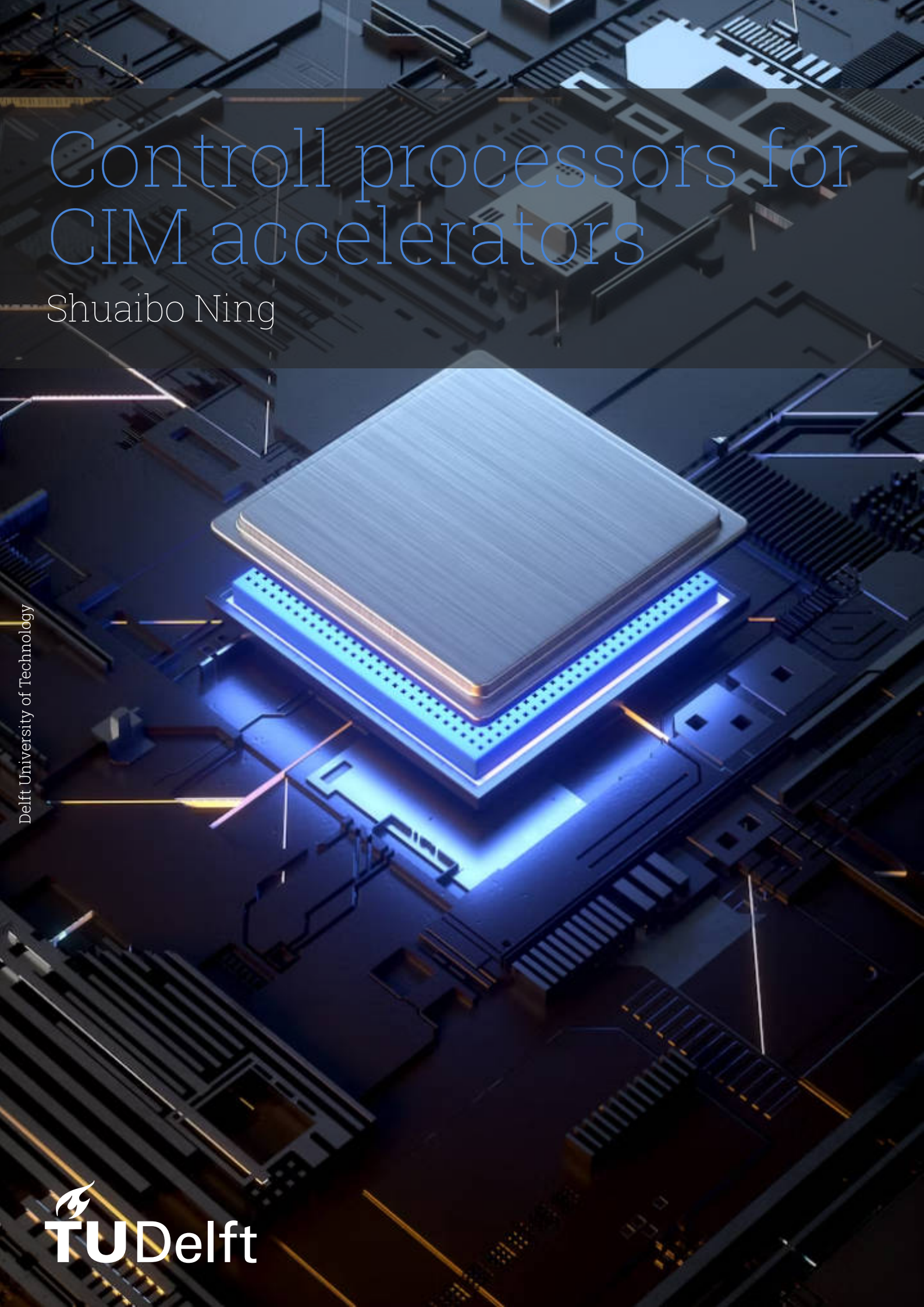
Controll processors for CIM accelerators

Shuaibo Ning



Delft University of Technology

TU Delft

Controll processors for CIM accelerators

by

Shuaibo Ning

Student number:	5761980
Project Duration:	09, 2024 - 09, 2025
Thesis committee:	Prof. G. Gaydadjiev, TU Delft, Supervisor Dr. T. Spyrou, TU Delft, Daily Supervisor Dr. C. Gao, TU Delft
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

Cover:	Central Computer Processors CPU concept (3D rendering, conceptual image) by Getty Images for Unsplash+
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

The completion of this thesis marks not only the culmination of my research efforts but also a journey made possible by the unwavering support, guidance, and inspiration of numerous individuals. It is with deep gratitude that I reflect upon the contributions of those who have accompanied me throughout this endeavor at Delft University of Technology.

I am grateful to my supervisor, Prof. Georgi Gaydadjiev, and my daily supervisor, Dr. Theofilos Spyrou, for their invaluable guidance. Through our weekly meetings, they steadily guided this work and provided constructive feedback essential to its completion.

I am particularly thankful for the technical assistance provided by the laboratory staff, whose expertise and readiness to help were vital to the smooth progression of my experiments.

On a personal note, I owe my deepest thanks to my family and friends. Their unconditional love, understanding, and endless encouragement have been my foundation throughout this challenging yet rewarding chapter.

Lastly, I acknowledge the intellectual and institutional support of Delft University of Technology, which provided an exceptional environment for learning. This thesis is as much a product of their collective influence as it is of my own efforts.

Shuaibo Ning
Delft, November 2025

Abstract

To address the “memory wall” of von Neumann architectures, computation-in-memory (CIM) emerges as a promising paradigm that performs computation directly within memory, thereby enhancing energy efficiency and performance by minimizing data movement. This thesis explores a CIM tile architecture for 1T1R memristor crossbar arrays, supporting fundamental operations including write, read, verification, logic (AND, OR, XOR), and vector-matrix multiplication (VMM). The proposed design integrates key components including input/output buffers, analog interfaces such as Digital-to-Input Modules, Sample&Hold modules, and Analog-to-Digital Converters (ADCs), as well as a comprehensive digital periphery. The digital periphery encompasses the Column Selection, Row Selection, Row Data Selection, and Spare Row Selection modules, Logical Unit, Verification Unit, Read Register, and three-stage Addition Unit dedicated to VMM.

A significant part of our work is the RTL design and gate-level implementation of the tile’s comprehensive digital periphery. As part of this effort, we design and implement a novel Verification Unit that performs offline validation of data written to the crossbar array. This unit detects write errors, initiates selective rewriting of faulty cells, and performs re-verification. If a failure persists, it automatically flags the affected row, disables it for future computations, and triggers remapping to a spare row. Furthermore, we develop analytical models for estimating the area and energy consumption of the digital periphery, based on the characterized gate-level implementation.

The remapping functionality is validated through RTL simulation and the gate-level implementation of the tile’s comprehensive digital periphery is evaluated through post-synthesis simulations using Cadence Genus. We analyze area and energy consumption across key architectural parameters including crossbar size, number of ADCs, operand width, number of spare rows, external bus width, and WDS/RDS instruction data field width. Area analysis shows linear growth with crossbar size and spare rows, non-monotonic relationship with operand width (first decreasing then increasing), increasing trend with ADC count, and inverse relationships with bus width and data field width. Energy analysis reveals that store operations exhibit quadratic scaling with crossbar size and inverse proportionality with bus width; verification, read, and logic operations show quadratic scaling with crossbar size and inverse relationships with ADC count; VMM operations display linear scaling with crossbar size, inverse relationship with ADC count, and inverse proportionality with WDS/RDS instruction data field width. These findings provide valuable guidance for parameter optimization in CIM tile design.

Contents

1 Introduction

The rapid expansion of data-intensive computing encompassing cloud services, databases, scientific simulations, edge analytics, and Artificial Intelligence (AI)-has exposed critical limitations in traditional architectures, particularly in their poor energy efficiency and memory bandwidth limitations. In response, computation-in-memory (CIM) has emerged as a breakthrough solution, overcoming the von Neumann bottleneck by performing computations directly within the memory array, thereby minimizing costly data movement. To further elevate the efficiency and programmability of CIM accelerators, an optimized control processor is essential. This thesis proposes an efficient and flexible controller architecture that orchestrates data flow, manages analog-digital interfaces, and enables programmable computation-in-memory.

This chapter presents the motivation for this thesis by first analyzing the constraints of the traditional von Neumann architecture. It next surveys representative solutions, including cache hierarchies and 3D stacking, which, despite alleviating bandwidth and latency pressures, retain the fundamental separation of compute and memory. Subsequently, the chapter introduces computation-in-memory (CIM) as a promising alternative and outlines its advantages alongside the key technical challenges it faces. Finally, the chapter concludes with a summary of the thesis contributions and an overview of the thesis organization.

1.1. Motivation

With the rapid expansion of data-intensive computing, the “memory wall” in traditional computing systems has become a significant bottleneck for both performance and energy efficiency. In practice, data transfer between main memory and compute units accounts for the majority of system energy, averaging 62.7% across representative Google consumer-device workloads and reaching 77% for tasks such as scrolling a Google Docs page [7]. Meanwhile, the slowdown of Moore’s Law and the end of Dennard scaling mean we can no longer rely on CMOS process shrinkage alone to increase computing power.

Computation-in-memory (CIM) is proposed to address this by performing computations directly within the memory arrays. CIM reduces the transfer overhead between memory and the compute units, thereby minimizing latency and improving both performance and energy efficiency.

1.1.1. The Traditional von Neumann Architecture

The foundation of modern computing systems can be traced back to the computer-architecture concept proposed in 1946 by John von Neumann [26]. The architecture is usually composed of five classic components: input device, output device, memory unit, arithmetic unit, and control unit [26]. Its workflow is essentially centered on the arithmetic unit. Under the coordination of the control unit, instructions are sequentially retrieved from the memory and executed to complete specific tasks. The core characteristics of the von Neumann architecture include:

1. **Single Memory Space**: Both data and program instructions are stored in the same memory space, allowing the CPU to fetch instructions and data from the same memory;

2. **Sequential Execution**: Instructions are executed sequentially, meaning the CPU processes one instruction at a time in the order they are stored in memory, unless directed otherwise by control flow instructions (like jumps or branches);
3. **Bus System**: A system of buses (data, address, and control buses) facilitates communication between the CPU, memory, and input/output devices. This allows data and instructions to be transferred efficiently;
4. **Flexibility**: The architecture is flexible, allowing for a wide range of programming languages and applications. This adaptability has contributed to its longevity in computer design.

This architecture connects the processor and memory via a single bus to fetch instructions and data from the memory, achieving great flexibility for general-purpose computing. However, with explosive growth in computational demands—especially in high-performance computing (HPC) and data-intensive applications like deep learning—the inherent limitations of the von Neumann design have emerged:

- **Memory wall**: The term “memory wall” was introduced in 1995 by William A. Wulf and Sally A. McKee in a brief, controversial piece titled “Hitting the Memory Wall: Implications of the Obvious,” published in ACM SIGARCH Computer Architecture News [39]. It denotes the growing gap between processor speed and memory bandwidth. This disparity limits the overall system performance, as the processor spends more time waiting for data from memory, leading to a bottleneck;
- **Power consumption**: Frequent data movement between the processor and memory consumes vast amounts of energy. Its energy consumption and latency have become major bottlenecks for energy efficiency and throughput, especially in large-scale data or AI computing scenarios [21];
- **Sequential fetch–execute model limits parallelism**: Instructions and data share a single path, so we can not fetch the next instruction while loading data for the current one [5], capping achievable instruction-level parallelism.

Collectively known as the “von Neumann bottleneck”, the above issues severely constrain system performance in fields such as artificial intelligence. In deep learning, particularly in neural-network models, there is often a large number of parameters, and each inference requires moving substantial volumes of weight data from external memory into on-chip caches/registers for MAC operations, making traditional architectures highly inefficient.

1.1.2. Addressing the von Neumann Bottleneck: Emerging Solutions

To address the von Neumann bottleneck, researchers and industry practitioners have proposed a range of solutions. Here are four classic solutions:

1. **Cache Memory**: Implementing multiple levels of cache memory (L1, L2, L3) between the CPU and the computer’s main memory. These caches provide much lower latency and higher bandwidth than main memory and, by exploiting temporal and spatial locality, keep frequently used data and instructions close to the processor core, thereby reducing data-movement distance and average memory access time. This narrows the processor–memory performance gap and improves performance and energy efficiency of the system [33];
2. **Prefetching Techniques**: Prefetching is a technique that hides latency by loading data or instructions into caches before they are needed, based on predictions of a program’s future memory access patterns [22]. It can be done either in hardware or software. Hardware-based prefetching uses dedicated mechanisms that observe ongoing access patterns, identify the next few elements the program might need, and prefetch them into the processor cache in advance [27]. Software-based prefetching is carried out by the compiler, which analyzes code behavior during compilation and inserts specific ”prefetch” instructions into the final program code [28];
3. **Out-of-order execution**: Out-of-order execution is a performance optimization technique that allows CPUs to dynamically rearrange the sequence of instructions, executing them as soon as the required data and computational resources are available, rather than strictly respecting the binary program order;

4. **3D Stacking**: Three-dimensional (3D) integration stacks memory directly atop the processor, shortening interconnect paths and enabling ultra-wide, highly parallel interfaces; This increases the memory bandwidth and reduces the latency between the processor and memory, thereby improving overall processor performance and memory-access efficiency [24].

Although these solutions mitigate the "von Neumann bottleneck," they do not address its root cause: the architectural separation of compute and memory sub-systems. Recently, attention has turned to computation-in-memory (CIM), a technology that tackles the memory bottleneck by performing operations within memory arrays. Next, we will introduce the concept of CIM, its advantages, and outline the key technical challenges.

1.1.3. Computation-in-Memory and Technical Advantages and Challenges

Computation-in-Memory (CIM) is an architecture where the data operations are performed directly inside the data memory [15]. Because computation occurs where the data resides rather than transferring data to the processor, this technique significantly reduces frequent data transfers between processors and storage, thereby improving energy efficiency and performance when moving data between the processor and memory [36]. The advantages of CIM architecture include the following:

1. **Reduced energy consumption**: CIM's greatest energy advantage comes from reducing the massive data movement between processors and external memory in traditional architectures. By performing computations directly within the memory unit, this technique eliminates the energy-intensive and time-consuming data movement [17];
2. **High bandwidth parallel processing capabilities**: Unlike traditional architectures, where throughput is limited by the number of computing units and memory bandwidth, CIM utilizes the parallel nature of crossbar arrays to perform massive multiply-accumulate (MAC) operations simultaneously. In analog CIM designs, each bit-line/word-line essentially acts as an independent computing channel [29]. Hundreds or even thousands of these channels can execute MAC operations in parallel within a single clock cycle;
3. **Improved area efficiency**: By compactly integrating computational circuitry with memory cells, CIM significantly reduces the silicon area occupied by the buses, caches, and register arrays required for data movement in traditional architectures. Many designs embed analog accumulation circuits or digital post-accumulation structures directly alongside standard SRAM or RRAM macros, allowing the memory array to perform both storage and computation functions.

In summary, the CIM represents a fundamental shift in architecture that directly addresses the problem of massive data movement between processors and external memory by colocating computation and memory. By performing operations like multiply-accumulate within the memory array itself, CIM offers a highly promising alternative to traditional designs, delivering significant improvements in energy efficiency, parallel processing bandwidth, and silicon area utilization critical for next-generation computing, particularly in data-intensive applications like AI.

Unlike traditional computing, the core computation in CIM occurs within the crossbar array itself, where input voltages interact with memristor conductance to generate output currents that represent computational results. While the potential for significant energy and performance gains offered by CIM is clear, several challenges remain. To realize the promise of CIM, these problems must be explored and resolved. Following are some of the key challenges currently faced by computation-in-memory:

1. **Device-Level Challenges:**
    - *Non-Idealities*: Device-level non-idealities substantially affect performance and reliability. A major challenge is the conductance variation. It refers to the deviation between the programmed resistance of a memristor and its expected target value [31], [19], [44]. Another concern is conductance drift, which means the conductance states of the memristors tend to drift with time [1]. Although memristors can exhibit long retention up to 10 years [34], device degradation can occur as the oxygen-vacancy profile evolves due to oxygen diffusion [35]. Finally, device endurance typically ranges from 10^6 to 10^{12} switching cycles [16], [20], [10], [8], [38]. The limited endurance bounds the number of permissible reprogramming times;

- *Device Modeling*: To enable circuit- and architecture-level studies of CIM based on memristors, accurate compact models of the memristors are needed. These models characterize key physical characteristics—particularly the dynamic switching behavior during SET and RESET operations—and account for experimental variability. Consequently, creating reliable device models for circuit simulations remains an active area of research [6];
- *Device Testing*: The CIM architecture, which involves the memristor crossbar array, differs from the architectures based on the von Neumann computing paradigm. To guarantee high-quality memristor devices, better test solutions are needed that specifically target the actual physics of memristor defects, rather than just defects in the surrounding circuitry. Consequently, developing appropriate testing methods based on their unique defects presents another challenge [13].

2. **Circuit-Level Challenges:**
   - *Sneak Paths*: Unintended current paths within the crossbar array during read/write operations, corrupting data and increasing power consumption;
   - *IR Drop*: Non-ideal wire resistance causes significant voltage drops across large arrays, leading to non-uniform programming and readout errors, especially at the array periphery;
   - *Analog-to-Digital Converter (ADC) Overhead*: Converting analog computation results (currents/voltages) to digital values for further processing consumes significant area and power, often becoming the bottleneck. Reducing ADC precision hurts accuracy.
3. **Architecture & System-Level Challenges:**
   - *Instruction Set & Control*: CIM tiles comprise crossbar arrays and peripheral circuits (e.g., ADC/DAC). At the architecture level, a main challenge is control: choosing between fixed-function or ISA-based programmability [2, 43], defining how the components in the CIM tile are controlled, and how much flexibility the control system requires. A well-designed control can reduce the data movement and improve efficiency;
   - *System Integration & Interfaces*: System-level integration is a key consideration, which involves enabling communication between interconnected CIM tiles within the system and designing the necessary interfaces for the tiles to communicate with outside system.
4. **Application-Level Challenges:**
   - *Application Identification*: A main challenge is that not all applications are naturally suitable for CIM acceleration. The limited set of operations supported by CIM architectures (e.g., logic, addition, multiplication within the memory array). Therefore, analyzing and identifying which specific applications have computational patterns that match CIM acceleration is essential.

While CIM faces challenges across multiple levels from device level to application level, this thesis focuses primarily on addressing the architecture-level and device-level challenges.

1.2. Thesis Contributions

The contributions of this thesis are related to the peripheral digital circuits for CIM, which make the memristor crossbar array reconfigurable. The key contributions are as follows:

- **Implementation and Validation of Digital Periphery RTL Design:** We implement and validate the complete RTL and gate-level digital periphery for the CIM tile architecture using Verilog. This provides a reconfigurable control framework that addresses architecture-level programmability challenges;
- **Parameterized Area and Energy Models for Design Space Exploration:** We develop comprehensive parameterized area and energy models for the digital periphery circuits, providing a foundation for other researchers to perform systematic design space exploration and architectural optimization;
- **Verification and Remapping Function:** We implement a Verification Unit that detects write errors, selectively rewrites faulty cells in the crossbar array, and, in the case of persistent failures,

disables the affected row and remaps it to a spare row. This provides a solution to address reliability challenges at the device level.

1.3. Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 provides background on memristor device theory, crossbar structures, power estimation on digital circuits, an overview of CIM tile architectures, and the state-of-the-art in CIM designs setting the foundation for the rest of the work.

Chapter 3 presents the CIM tile architecture and ISA: it details the tile's core operations (write, read, verify, compute), the instruction set architecture, and the pipelining scheme for parallel execution.

Chapter 4 describes the hardware implementation, including the synchronous FIFO buffers, the column selection module, the row selection module, the row data selection module, the spare row selection module, the tile controller, verification unit, logical unit, and a three-stage addition unit. This chapter also outlines the system's interface with key analog components, explaining the function of the Digital Input Module (DIM) and a shared ADC solution that enables multiple columns to share a single ADC.

Chapter 5 presents the area models for each digital periphery module and the instruction level energy models for each digital periphery module.

Chapter 6 presents comprehensive experimental results and analysis, beginning with RTL-level validation of the remapping functionality, followed by area and energy evaluation based on post-synthesis gate-level simulations using Cadence Genus under a defined baseline configuration, including area overhead analysis and energy consumption examination across different operations.

Chapter 7 concludes the thesis and presents potential directions for the future work.

2

Background

This chapter introduces the technical background of memristor-based computing-in-memory (CIM). As a foundation, we first explain the basic operating principles of memristor devices and their core operations-the write and read processes. We then detail the key hardware architecture that interconnects these devices to form computation arrays: the crossbar array. A generic CIM tile architecture is then introduced, as well as the fundamentals of power consumption. Finally, the chapter surveys the current state of designs of memristor-based CIM across.

2.1. Memristor Device Theory

Before 1970, three basic circuit elements were known and implemented: resistance (R), capacitance (C), and inductance (L). Each of these elements correlates to two of the four fundamental electrical quantities (charge, current, voltage, and magnetic flux):

1. **Resistance (R)**: Characterizes the relationship between voltage change and current change, i.e.

$$dv = R\,di$$

2. **Capacitance (C)**: Characterizes the relationship between charge and voltage, i.e.

$$dq = C\,dv$$

3. **Inductance (L)**: Characterizes the relationship between magnetic flux and current, i.e.

$$d\phi = L\,di$$

As shown in Figure 2.1, these three components cover the correspondence between three pairs of basic electrical quantities. In 1971, Leon Chua theoretically predicted the properties of a fourth basic element, which fills the gap between the three elements mentioned above: magnetic flux (ϕ) and charge (q), $d\phi = Mdq$. He named this element "Memristor" [12]. It was not until 2008 that Strukov et al. from HP Labs experimentally confirmed the memristive effect on TiO_2 films, marking the leap of memristors from theoretical concepts to practical devices [32]. Equation 2.1 mathematically characterizes the memristor's behavior.

$$M(q) = \frac{d\Phi}{dq} \;\Rightarrow\; M\big(q(t)\big) = \frac{d\Phi/dt}{dq/dt} = \frac{V(t)}{I(t)}. \tag{2.1}$$

Figure 2.2 shows the structure of memristive device. A memristive device consists of a top and a bottom electrode with a resistive-switching metal-oxide layer in between. By applying an external voltage across the two electrodes, the resulting electric field modulates the distribution of oxygen vacancies in the switching layer, thereby changing the shape and continuity of conductive filaments and, consequently, the effective resistance of the device. According to the switching behavior, memristive devices are categorized as unipolar device shown in Figure 2.3(a) and bipolar device shown in Figure 2.3(b)

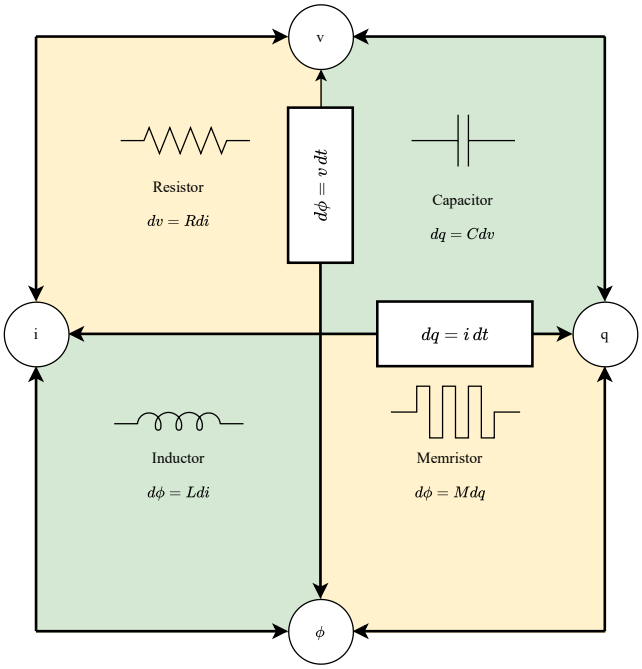


Figure 2.1: Four fundamental circuit elements and the relation between the four elements.

: in unipolar devices the switching depends only on the amplitude of the applied voltage and is independent of its polarity, whereas in bipolar devices the switching direction is determined by the voltage polarity [37].

The operation of a memristive device consists of write and read processes. The write process includes: (1) Forming, where a one-time high voltage is applied to nucleate a conductive filament in the switching layer, turning the device from its initial highly insulating state into a low-resistance state (LRS); (2) SET, where a voltage with the same polarity as forming promotes further filament growth or bridging, switching the device from a high-resistance state (HRS) to LRS; and (3) RESET, where a voltage of the opposite polarity (or a higher amplitude, depending on the device type) ruptures the filament, switching the device from LRS back to HRS. The typical relationship among the voltage magnitudes is

$$|V_{\text{forming}}| > |V_{\text{reset}}| > |V_{\text{set}}|\,. \tag{2.2}$$

A variety of fabrication technologies and materials are employed in the development of memristors. There are three prevalent types:

Resistive Random-Access Memory (ReRAM) is a type of memristor device that uses a metal-insulator-metal stack structure. In bipolar ReRAM, the operating principle is the reversible formation and dissolution of conductive filaments. The filament formation and dissolution are achieved by changing the polarity of the programming voltage (e.g., +2 V). To avoid read disturbance, only a small voltage (e.g., 0.2 V) is applied during reading, and the device's state is read by measuring the current or voltage across it. Programming, however, requires higher voltage/current and longer latency. This enables the device to switch between two stable resistive states: low resistance state (LRS) and high resistance state (HRS) [42].

Phase-change memory (PCM) is based on the attribute of certain materials, such as GeSbTe (GST). Memory switching in these materials is primarily a thermal process. Through Joule heating in the device, the material can switch rapidly and reversibly between a high-resistance amorphous state and a low-resistance crystalline state; once the heat is removed, the material retains its phase. Reading is performed with a small, nondestructive bias to sense the cell resistance, and the resistance contrast between the two phases is about 2–3 orders of magnitude, enabling reliable state discrimination [4].

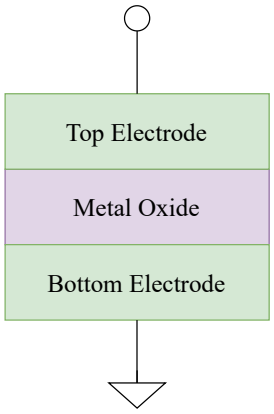
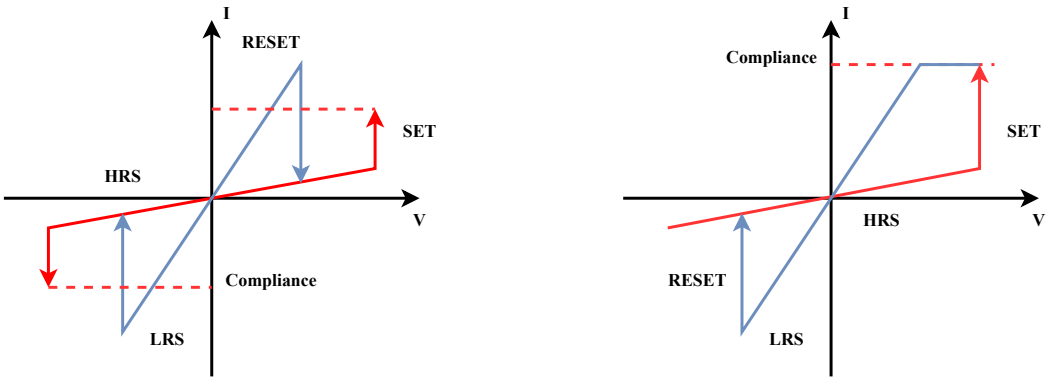


Figure 2.2: Structure of Memristor.



(a) Unipolar switching

(b) Bipolat switching

Figure 2.3: Memristor switching dynamics for bipolar and unipolar devices

A ferroelectric field-effect transistor (FeFET) is a MOSFET-like device in which a ferroelectric layer is inserted between the gate electrode and the semiconductor channel. The device's conductance is switched by the polarization state of this ferroelectric barrier [23]: an applied electric field can reversibly switch the polarization between multiple stable orientations. Depending on that orientation, the ferroelectric polarization either strengthens or weakens the effective gate field, thereby modulating channel formation. When it reinforces the gate field, a robust channel forms and the device exhibits a low resistance state; when it counteracts the gate field, channel formation is suppressed, yielding a high resistance state. Because these polarization states are non-volatile, the FeFET's stored state can be sensed by measuring the source–drain current.

2.2. Memristor Crossbar Structure

By integrating memristors in a crossbar array, not only data storage can be achieved, but this also enables in-memory computing. There are many structural solutions available, and the following will briefly describe two common structures.

2.2.1. 1R Crossbar Structure

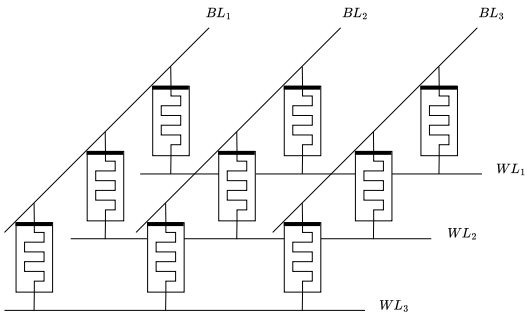


Figure 2.4: 1R Crossbar Structure.

As shown in Figure 2.4, each crosspoint contains a single memristor; the top electrodes are tied along the bit-lines (BLs) and the bottom electrodes along the word-lines (WLs). Ideally, reading a target cell requires selecting its word-line/bit-line pair and applying a small bias. In practice, however, unselected cells create sneak-path currents so that current flows through unintended paths, leading to read/write errors and increased energy consumption [45]. Also, the number of such parasitic paths grows with the crossbar array size. Therefore, mitigation techniques include biasing schemes (e.g., $1/2V$), complementary resistive switching (CRS) cells, or adding a discrete selector to form 1S1R are proposed [18, 30].

2.2.2. 1T1R Crossbar Structure

To suppress sneak paths in the 1R crossbar structure, the 1T1R crossbar structure is introduced to solve this leakage current problem [25]. The 1T1R structure is a unit structure formed by connecting transistors and memristors in series. The crossbar array composed of 1T1R is shown in Figure 2.5. The bottom electrode of the memristor in the crossbar array is connected to the drain of the transistor, and the top electrode of the memristor is connected to the bit line. The gate of the transistor is connected to the word-line, and the source is connected to the source-line. When a memristor in the array needs to be operated, the transistor connected to the memristor is first turned on through the word line, and then the operating voltage is input on the bit line, and the source line is grounded. The 1T1R structure effectively solves the leakage current problem at the cost of area and complexity.

2.3. Generic CIM Tile Architecture

A CIM tile consists of a memristor crossbar, digital and analog peripherals, and an on-tile controller [41], as shown in Figure 2.6. The CIM tile accepts digital data and instructions as inputs from outside; the controller interprets these instructions and coordinates the internal circuits to perform operations such

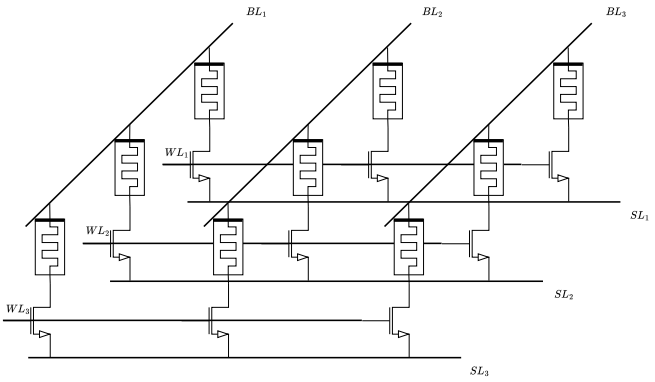


Figure 2.5: 1T1R Crossbar Structure.

as Vector-Matrix-Multiplication (VMM). Data typically flows through four steps: (1) input processing, which processes the external digital inputs (such as decoding) to prepare the input vector for the next step of crossbar computation. This step accepts digital inputs and generates either digital or analog outputs (D2D or D2A); (2) crossbar computation, where the memristor crossbar array carries out the core analog multiply-accumulate (MAC) operations, computing the dot product between the input vector and the stored weight matrix, with the result represented as the analog currents. Its inputs can be digital or analog, and its outputs are analog (D2A or A2A); (3) sensing, which samples and digitizes the crossbar analog outputs (A2D); and (4) output processing, which performs digital post-processing on the digitized results to produce the final digital outputs (D2D) according to different functions. Prior work often focuses on one specific circuit per stage to build application-specific accelerators. To approach a generic CIM tile design, the goal is to support a broad set of primitive functions at each stage that the controller can compose, via the dataflow, into richer logic and arithmetic capabilities within the generic CIM tile.

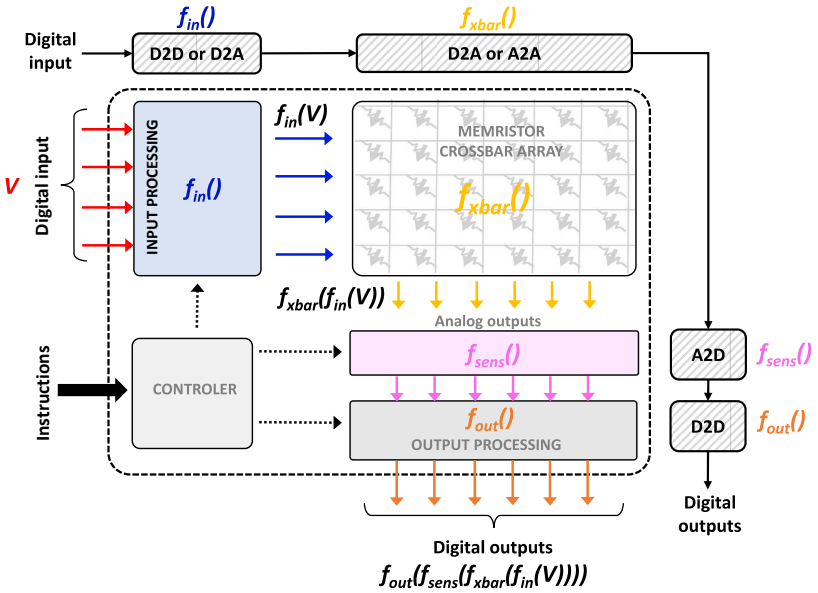


Figure 2.6: Generic CIM Tile four steps [41]: (1) input processing, (2) crossbar computation, (3) sensing, (4) output processing.

2.4. Power Consumption Fundamentals

Power consumption is the rate at which an electronic device or circuit uses electrical energy. It is quantified by the basic relation:

$$P = IV \tag{2.3}$$

where P denotes power, V the voltage, and I the current. Because power is a key metric for evaluating chip performance, accurate assessment of circuit power is essential. In integrated circuit design, power consumption is primarily categorized into static power (leakage power in steady state) and dynamic power (switching power and internal power during state transitions). The following sections will provide a detailed introduction to these two types.

2.4.1. Static Power (Leakage Power)

Static power is the power consumed when a device is not switching. In digital circuits it mainly arises from various leakage currents. The dominant leakage mechanisms are subthreshold leakage, gate leakage, Gate Induced Drain Leakage (IGIDL), and Reverse Bias Junction Leakage (IREV). The subthreshold leakage is the current flowing when the transistor is supposed to be off. The gate leakage is Current flowing directly from the gate through the oxide to the substrate due to gate oxide tunneling and hot carrier injection. The Junction leakage occurs when the source or drain diffusion region is at a different potential than the substrate. Junction leakage is typically very small compared to other leakage currents. The Reverse Bias Junction Leakage is Caused by minority carrier drift and the generation of electron/hole pairs in the depletion region. The Leakage power can be expressed in the general form $P = VI$, and more specifically as

$$P_{\text{static}} = V_{DD} \left(I_{\text{sub}} + I_{\text{gate}} + I_{\text{junction}} + I_{\text{GIDL}} \right). \tag{2.4}$$

Because each standard cell's leakage is tabulated in the library file, the static power under specified PVT conditions can be obtained by summing the leakage of all cells (including macros):

$$P_{\text{total}} = P_{\text{leakage},1} + P_{\text{leakage},2} + \cdots + P_{\text{leakage},N}. \tag{2.5}$$

2.4.2. Internal Power (Short-Circuit Power)

Internal power is also called short-circuit power because it mainly arises from short-circuit conduction. When the input signal switches, the transition cannot be instantaneous, so the PMOS and NMOS cannot always be one off and the other on. For a brief interval, they conduct simultaneously, creating a path from the supply V_{DD} to ground V_{SS}, which produces a short-circuit current, as shown in Figure 2.7, the inverter schematic:

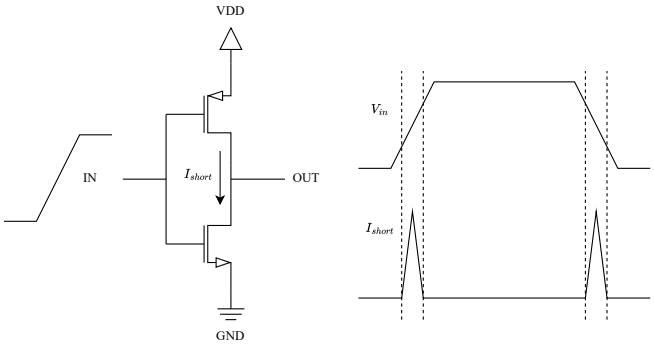


Figure 2.7: Internal Power of Inverter

For each standard cell, internal power is tabulated in the Library file via tables parameterized by input slew and output load. Therefore, under specified PVT, V_{DD}, and activity information (VCD/SAIF), the internal power is obtained by summing across all cells (including macros):

$$P_{\text{int,total}} = \sum_{i=1}^{N} P_{\text{internal},i} = \sum_{i=1}^{N} \alpha_i \, f \, E_{\text{internal},i} \, .$$

where f is the clock frequency and α is the activity factor.

2.4.3. Switching Power

Switching power (also called transition power, a component of dynamic power) is the power consumed when the circuit charges or discharges the load capacitance at an output node during a signal transition. Taking a CMOS inverter as an example shown in the Figure 2.8: when the input toggles from $0 \rightarrow 1$ or $1 \rightarrow 0$, the load capacitance C_{load} is charged by V_{DD} or discharged through the NMOS, so switching power is ocurred.

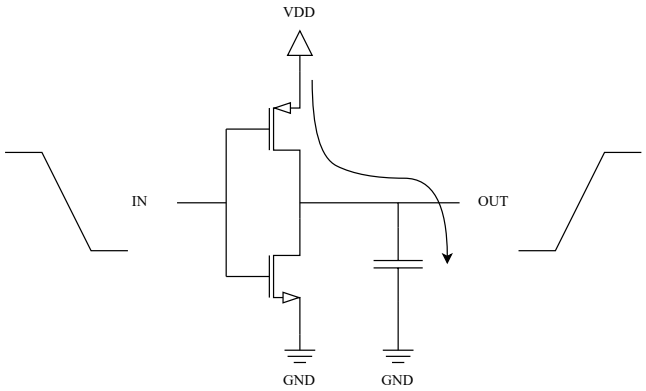


Figure 2.8: Switching Power of Inverter

The switching power can be expressed as following:

$$P_{\text{switch}} = \tfrac{1}{2}\, V_{DD}^2\, C_{\text{load}}\, T_r$$

Here T_r is the toggle rate (transitions/second, counting all transitions). A common alternative form is

$$P_{\text{switch}} = \alpha\, C_{\text{load}}\, V_{DD}^2\, f,$$

where f is the clock frequency and α is the activity factor (average number of $0 \rightarrow 1$ transitions per clock period. The effective load capacitance usually includes gate, interconnect, and diffusion components:

$$C_{\text{load}} = C_{\text{pin}} + C_{\text{wire}} + C_{\text{diff}}\,.$$

Under specified PVT, the switching power can be calculated with the simulated or statistical activity information (VCD/SAIF):

$$P_{\text{switch,total}} = \sum_i \alpha_i\, C_{\text{load},i}\, V_{DD}^2\, f\,.$$

2.5. Overview Of State-Of-The-Art Designs

In this section, we introduce representative state-of-the-art CIM designs, which can be broadly categorized into generic and application-specific designs. The former focuses on programmability, while the latter aims at solving specific challenges to achieve extreme efficiency for particular tasks.

A prominent example of a generic, programmable design is the Programmable Ultra-efficient Memristor-based Accelerator (PUMA) [3]. Building on memristor crossbar arrays, PUMA incorporates general-purpose execution units to significantly broaden functionality, enabling efficient acceleration of a wide range of machine-learning (ML) inference tasks. Through a specialized instruction set architecture (ISA), the design delivers microarchitectural innovations that preserve the high energy efficiency of CIM while providing strong programmability.

To support programming on PUMA, the work also develops a companion compiler that translates high-level code into the PUMA ISA. The compiler partitions the computational graph and optimizes instruction scheduling and register allocation, generating efficient code for large, complex workloads that run on thousands of spatial cores.

For comprehensive evaluation of performance and energy, the authors build a detailed architecture simulator integrating functional, timing, and power models of PUMA's key components. Results show that, at 1 GHz and at the 32 nm technology node, PUMA achieves an area efficiency of 577 GOPS/s/mm^2 and an energy efficiency of 837 GOPS/s/W. PUMA delivers up to 2,446× improvement in inference performance and up to 66× reduction in latency compared with state-of-the-art GPUs across diverse ML applications—including image recognition, machine translation, and language modeling—with model sizes ranging from 5 million to 800 million synapses. Compared with the application-specific memristor-based accelerator ISAAC, PUMA has 20.7% lower power efficiency and 29.2% lower area efficiency due to the overhead of programmability.

Similarly, the in-memory data parallel processor proposed in [14] introduces a programmable, general-purpose architecture for CIM using ReRAM. It proposes a new set of in-memory instructions that function as complete operations executed on ReRAM crossbars. These instructions operate on a Single Instruction, Multiple Data (SIMD) execution model, unlocking massive data parallelism, with the crossbars natively supporting operations like multiplication, addition, and subtraction. To facilitate programming for this novel architecture, the work develops a compilation framework that takes TensorFlow input and generates code for the in-memory processor. This integrated approach yields remarkable performance, demonstrating a 7.5x speedup over a multi-core CPU server for Parsec applications and a 763x speedup over a server-class GPU for Rodinia benchmarks.

In contrast, application-specific designs are optimized to accelerate specific applications, such as convolutional neural network.

A notable example is ISAAC [29], which explores an in-situ processing approach using memristor crossbar arrays that both store weights and perform analog dot-product operations. To enhance efficiency, it employs a pipelining mechanism where crossbars are allocated per layer and eDRAM buffers aggregate data between these pipeline stages, effectively reducing the inter-layer buffer requirements. The work further proposes data-encoding techniques to mitigate ADC overhead and integrates necessary digital peripherals. Through a careful design-space exploration, ISAAC achieves a balanced design, demonstrating superior performance over the DaDianNao architecture with 14.8x higher throughput, 5.5x better energy efficiency, and 7.5x greater computational density on a suite of CNN and DNN workloads.

Another prominent application-specific design is PRIME [11], a ReRAM-based processing-in-memory (PIM) design. It partitions crossbar arrays into normal memory subarrays and reconfigurable "full-function" subarrays that can switch between memory and compute modes. In compute mode, the ReRAM crossbar executes analog VMM and convolution; reused peripherals (write drivers as DACs, sense amplifiers as ADCs) plus minimal logic provide activations and pooling, so common MLP/CNN layers are supported. A controller and buffer subarrays manage configuration and dataflow. Across the evaluated ML benchmarks, PRIME reports about 2360× higher performance and 895× better energy efficiency than a state-of-the-art neural processing unit design [9].

2.6. Summary

This chapter introduced the technical background and related work of memristor-based CIM. At the device level, we reviewed memristor physics, write/read processes, and bipolar vs. unipolar switching. At the array level, we compared 1R and 1T1R crossbars and discussed sneak-path issues and mitigation. All subsequent architectural and circuit assumptions build on a 1T1R crossbar array, adopted to suppress sneak-path currents while retaining high parallelism. At the architecture level, we described the generic CIM tile and its four processing steps: input processing, crossbar computation, sensing, and output processing. The proposed CIM tile structure in Chapter 3 is based on this generic CIM tile architecture. In addition, we summarized power consumption fundamentals: static (leakage), internal (short-circuit), and switching power. The energy model used throughout the thesis is derived from the power consumption fundamentals.

3

CIM Tile Structure and ISA

This chapter presents the architectural framework of the CIM tile. Section 3.1 presents an overview of the tile's structure and its four core functions (Write, Read, Verification, and Computation). Our overall architecture, as well as the Write, Read, and Computation functions, are based on Mahdi's prior work [41]. Building on that foundation, we extend the Verification function by introducing a remapping mechanism that improves reliability. Section 3.2 describes the instruction set architecture (ISA), which uses a subset of instructions from Mahdi's work [41] and enable dynamic reconfiguration of CIM operations via programmable instructions. Section 3.3 introduces a pipelining mechanism that partitions the tile's functional blocks into multiple stages to enhance throughput via overlapped execution and reduce idle time; this pipeline is adapted from Mahdi's design [41] with several modifications. The specific implementation details of the CIM Tile architecture are discussed in Chapter 4.

3.1. CIM Tile Structure

This section presents an overview of the proposed CIM tile and introduces four functions: Write function, Read function, Verification function, and Computation function.

3.1.1. CIM Tile Overview

Figure 3.1 illustrates the overall structure of the CIM tile, comprising four buffers, digital modules, analog modules, a controller, and a memristor crossbar array, along with the associated control signals and data signals. The core functions supported by this tile can be classified into four types: i) data writing; ii) reading; iii) data verification; and the last iv) computation. While computing, the CIM tile can perform VMM as well as the basic Boolean logic operations: AND, OR, and XOR. These logic operations can be applied to any selected pair of rows over the entire array. Next, we will focus on these four functions to explain the components of the tile architecture and its key modules in detail (the specific analysis of the control signals will be discussed in subsequent chapters).

3.1.2. Write Function

To perform the reading, verification, and any computation functions, the memristor crossbar array requires that the operand data (e.g., weights) be programmed into the target cells in advance. In the proposed CIM tile, the memristor crossbar uses the 1T1R architecture. To ensure that data can be accurately written to the target cell in the crossbar array, the Write Data Buffer, the Column Selection module, Row Selection module, and Row Data Selection module (a multiplexer) operate sequentially. First, the Write Data Buffer provides the data to be written to the memristor crossbar array. Next, the Column Selection module determines which specific Bit-Line Digital Input Modules (DIMs) to activate; these DIMs convert digital inputs into crossbar-level operating voltages. Finally, the Row Selection module selects which Gate DIMs and Source DIMs are activated and supplies the data for Gate DIMs' conversion, while the Row Data Selection module supplies the data for Source DIMs' conversion. By activating the correct rows and columns, this process confines the write to the selected cells without affecting other cells in the crossbar array, thereby preventing unintended accesses that would otherwise

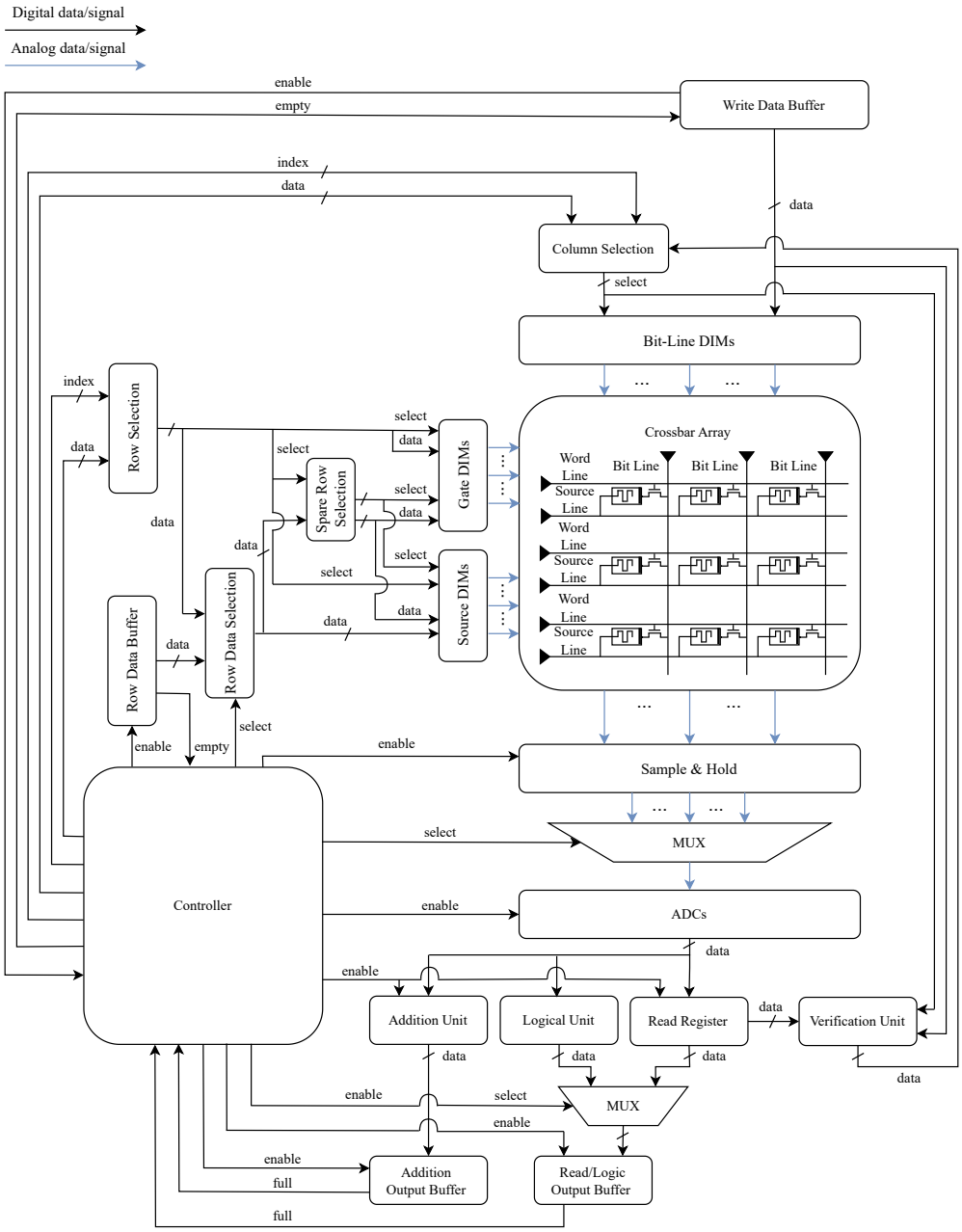


Figure 3.1: The structure of CIM Tile

cause incorrect operation.

3.1.3. Read Function

The read function is designed to read data from a specific row in the crossbar. Unlike the write operation, the read operation does not require column selection. However, the Row Selection module is still needed to activate the target wordlines and to supply the operand vector to the Row Data Selection module. The Source DIMs then convert the required operands into the prescribed read voltage(s) and drive the source lines accordingly. The Sample & Hold Module captures the crossbar's analog outputs, which are subsequently digitized by the ADCs; The resulting digital data is first stored in Read Register and is ultimately transferred to the Output Buffer.

3.1.4. Verification Function

The CIM tile performs a verification function immediately after a write operation to verify write correctness to enable accurate computation. Once the write operation completes, the Sample & Hold module captures the row's analog outputs, and the ADCs digitize the outputs. The Verification Unit then compares the digitized result directly against the just-programmed data. In this way, there is no need to reload the Row Selection module and buffer the write data elsewhere, minimizing extra instructions and energy overhead. If a mismatch is detected, mitigation techniques—rewrite and remap—are applied; detailed strategies are presented next.

As shown in the Figure 3.2, if the comparison succeeds, execution proceeds normally. Otherwise, the tile sets the `remap_flag` and rewrites only the erroneous cells and re-verifies. If the second verification still fails, the controller disables the row, clears the `remap_flag` and checks for an available spare row for remapping. If a spare row is available, the tile writes the data to the spare row. If no spare row is available, the controller sets the `no_spare_row_flag` signal to indicate that no additional spares remain.

3.1.5. Computation Function

For the computation function, there are two cases. (1) Logic operations (AND/OR/XOR): the Row Selection module selects the target Gate DIMs and Source DIMs. It supplies the gate-activation data to the Gate DIMs, which convert it into the corresponding wordline voltage(s) to enable the selected rows. In parallel, it provides the operand vector to the Row Data Selection module, which forwards the selected operands to the Source DIMs. (2) VMM: the Row Selection module still selects and activates the target wordlines (via Gate DIMs), but the operand vector comes from the Row Data Buffer; the Row Data Selection module fetches these operands from the buffer and delivers them to the Source DIMs. In both cases, the Source DIMs convert the selected operands into the prescribed source-line voltages to bias the array, the resulting bitline currents are captured by the Sample & Hold module, the ADCs digitize the outputs, and the digitized data are routed to the Logical Unit (for AND/OR/XOR) or to the Addition Unit (for VMM) to produce the final result.

3.2. CIM Tile Instruction Set Architecture

To efficiently perform the functions introduced in section 3.1, the tile must address several key challenges: selecting the appropriate functions, activating the relevant modules, controlling their execution sequence, and managing the inter-module data interfaces. Therefore, a dedicated instruction set is developed to address these challenges. The instruction set not only enables the correct execution of the functions, but also ensures that the overall tile runs with high efficiency. The instruction set is presented in Table 3.1, and the following parts provide detailed explanations for each instruction.

- **WDsh:** The Write Data Shift (`WDsh`) instruction drives the Write Data Buffer to transfer its data into the Bit-Line DIMs;
- **RDsh:** The Row Data Shift instruction drives the Row Data Buffer to transfer its data into the Source DIMs;
- **WDS:** The Write Data Selection (`WDS`) instruction selects which Bit-Line DIMs-equivalently, bit lines, as each bit line corresponds to one Bit-Line DIM-are enabled. It comprises three fields: an opcode that identifies the Write Data Selection operation; a block index that selects which group of bit lines to address (the memristor crossbar's bit lines are partitioned into blocks to accommodate

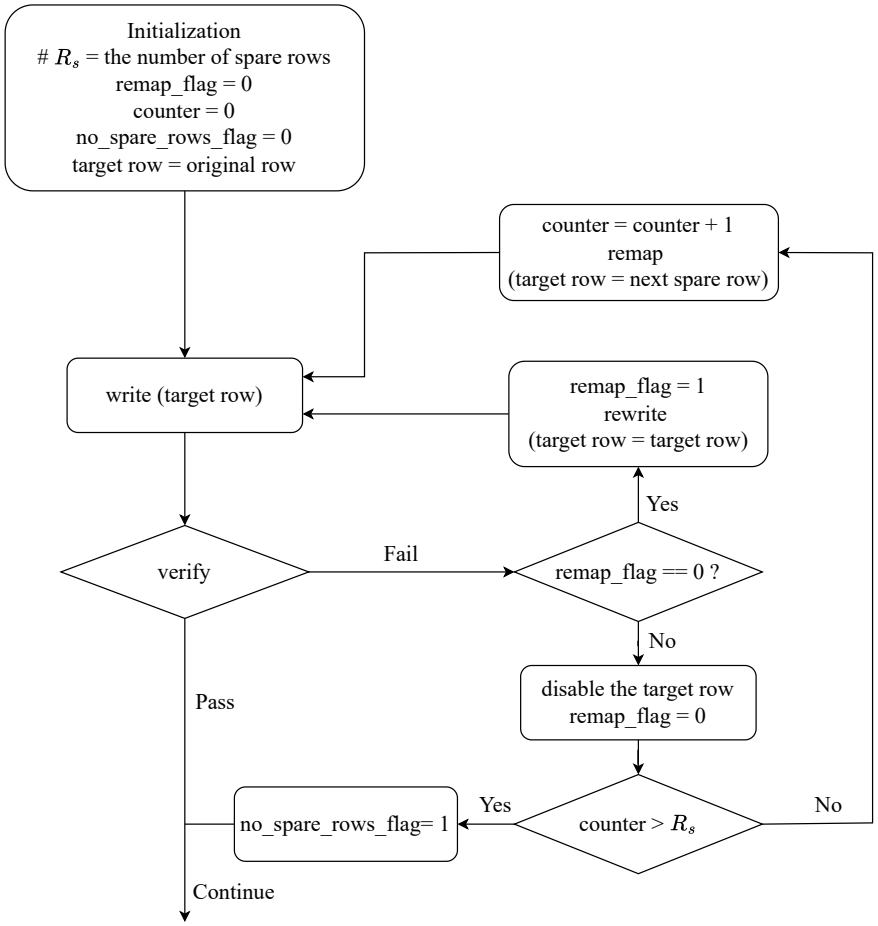


Figure 3.2: Verification Flow

Table 3.1: The Instruction Set Architecture

Opcode	Field1	Field2	Function description
WDsh			Shift the data out of the Write Data Buffer.
RDsh			Shift the data out of the Row Data Buffer
WDS	Index	data	select the specified Bit-Line DIMs
WDSs			select all the Bit-Line DIMs
RDS	Index	data	select the specified Gate DIMs and Source DIMs
RDSs			select all the Gate DIMs and Source DIMs
FS	Index		Select the function
DoA			Do the function on the crossbar array
DoS			Sample the outputs from the crossbar array
CSR	Index	data	Enable the ADC and choose the specified column to be digitized.
jal	Address		Save the current PC into the back register and jump to ”Address”.
jr			Jump to the PC stored in the back register.
BNE			Branch to the PC stored in the branch register.
LS			Indicate the last section of rows to be read.
IADD			Activate the third stage of the addition unit.
CP			Write logic/addition unit outputs to the Output Buffer.

bandwidth limitations of the instruction); and the data in the Field2 specifies exactly which bit Lines within the chosen block to activate;

- **WDSs:** The Write Data Selection Set (`WDSs`) instruction selects all the bit lines to activate;
- **RDS:** The Row Data Selection (`RDS`) instruction selects which Gate DIMs and Source DIMs to activate. We use “word line” and “Gate DIM” interchangeably, and “source line” and “Source DIM” interchangeably, since each word line and source line has a one-to-one corresponding DIM. It comprises three fields: an opcode that identifies the Row Data Selection operation; a block index that selects which group of word lines and source lines to address (the memristor crossbar’s word lines and source lines are partitioned into blocks to accommodate bandwidth limitations of the instruction); and the data in the Field2 specifies exactly which word lines and source lines within the chosen block to activate;
- **RDSs:** The Row Data Set (`RDSs`) instruction selects all the word lines and source lines to activate;
- **FS:** The Function Selection (`FS`) instruction configures the CIM tile to perform a specific function by loading the desired function selected by the index in the instruction into the controller;
- **DoA:** The Do Array (`DoA`) instruction is the execution trigger for all CIM tile functions on the memristor crossbar. After the DIMs have been set up by the instructions (`WDS`/`WDSs` for columns and `RDS`/`RDSs` for rows) and function mode has been chosen by the previous `FS` instruction, `DoA` will activate the DIMs to provide specific voltages (whether the voltages represent writing, reading, or computation functions) to the crossbar array’s source, word, and bit lines. Then the crossbar array performs the corresponding analog operation within the crossbar array;
- **DoS:** The Do Sample (`DoS`) instruction is used to enable the Sample & Hold module, which captures the analog outputs from the crossbar array in its storage units. After the result is safely stored in the Sample & Hold module, the crossbar array is free to do the next operation;
- **CSR:** The Column Select Read (`CSR`) instruction configures which crossbar columns are read and which ADC channels are enabled. To match the limited number of ADCs and reduce area, the array’s columns are partitioned into equal-sized blocks; The `CSR` instruction selects the same intra-block column across all blocks and routes those columns to the ADC inputs. `CSR` has three fields: an opcode identifying the operation, a block index specifying the selected intra-block column, and data that enables the corresponding ADC channel(s) for digitization;
- **jal, jr:** The `jal` instruction saves the current program counter (PC) to the back register and jumps to ”Address”, effectively calling a routine; the companion `jr` instruction returns by jumping to the PC stored in the back register. Together, `jal`/`jr` provide a lightweight call/return mechanism with minimal overhead;
- **BNE:** The Branch Not Equal (`BNE`) instruction conditionally branches to the address stored in the branch register when the compared values differ. If verification passes, execution continues sequentially. If the two values are not equal, indicating that the memristor did not program correctly, the `BNE` instruction branches to the PC stored in the branch register to rewrite the erroneous cells in the crossbar array. If the second verification still fails, the controller flags the row and disables its Digital Input Module (DIM), excluding that row from subsequent computations. This verification mechanism compensates for memristor non-idealities without requiring extra buffering or control logic;
- **LS, IADD:** The Last Section (`LS`) instruction and Immediate Addition (`IADD`) instruction are used for the addition unit, which is discussed in section 4.11. As for the limited resolution of ADC, summing across more rows must be done in multiple phases. Otherwise, it will exceed the maximum level. The `LS` instruction is used to indicate the final phase in this segmented accumulation. For instance, in a 16×16 crossbar array where each cell holds ’1’ and the ADC resolution is four, the 16 rows are read in four phases of four rows each. When the last phase is to be read, issuing the `LS` instruction indicates the addition unit to complete the sum for that column;

  In case of limited levels in a memristor cell within the crossbar array, a number cannot be represented by a single cell. Multi-bit values must be distributed across multiple cells in a single row, requiring multiple passes to read and sum those bits. The `IADD` instruction signals the final pass in this accumulation process. For example, in a 16×16 crossbar array where each cell has two

levels, a 4-bit number is stored across four cells in four columns of a row. This 4-bit number in binary is read bit by bit from each column and summed sequentially. On the final pass, issuing the `IADD` instruction indicates the addition unit to output the complete 4-bit sum;

- **CP:** The Copy (`CP`) instruction transfers the outputs of the Logical Unit, Read Register, or Addition Unit to their corresponding output buffers (the Read/Logic Output Buffer or the Addition Output Buffer).

The ISA is implemented with two dedicated decoders (Decoder1 and Decoder2), with the instruction set partitioned according to the pipeline stage. Decoder 1 is responsible for instructions in the Configuration and In-Memory Computation stages, which include `WDsh`, `RDsh`, `WDS`, `WDSs`, `RDS`, `RDSs`, `FS`, `DoA`. Decoder 2 manages the Read, Digital Post-Processing & Verification, and Output stages, processing instructions including `DoS`, `CSR`, `jal`, `jr`, `BNE`, `LS`, `IADD`, `CP`.

3.3. Pipelining

In order to improve computing efficiency, this CIM tile adopts a multi-stage parallel execution architecture, which decomposes the functions into multiple stages that can be pipelined. As shown in Figure 3.3 (different colors distinguish different stages), the architecture is divided into five key stages, each of which can independently work with its specific instructions.

1. Configuration stage (yellow dashed region): Set up the Write Data Buffer, the Column Selection module ,the Row Data Buffer, the Row Selection module, and the Row Data Selection module. Preparing the digital data used for Bit-Line DIMs, Source DIMs, and Gate DIMs.
2. In-Memory Computation stage (red dashed region): Perform the selected function in the analog crossbar array.
3. Read stage (green dashed region): Read the analog outputs from the crossbar array and digitize them.
4. Digital Post-Processing & Verification stage (purple dashed region): Do the addition, logical operation, or verification operation according to the selected function. These three units work independently.
5. Output stage (pink dashed region): Copy the results from the addition unit or logical unit to the Output Buffer.

3.4. Summary

This chapter defines the proposed CIM tile's structure, instruction set, and execution flow. The tile integrates the following key components: four buffers (Write Data Buffer, Row Data Buffer, Read/Logic Output Buffer, and Addition Output Buffer), digital control modules (Row/Column/Spare Row Selection, Row Data Selection), analog peripherals (DIMs, Sample & Hold, ADCs), computational units (Addition, Logical, Verification, Read Register), and the core memristor crossbar array, and a controller. The proposed CIM Tile supports write, read, verification, and computation, the latter covering Boolean logic (AND/OR/XOR) and VMM. To orchestrate these operations, a compact instruction set is introduced. Finally, the chapter presents a five-stage pipeline (Configuration, Execution, Read, Computation, and Output) that enables parallelism and reduces hardware idle time.

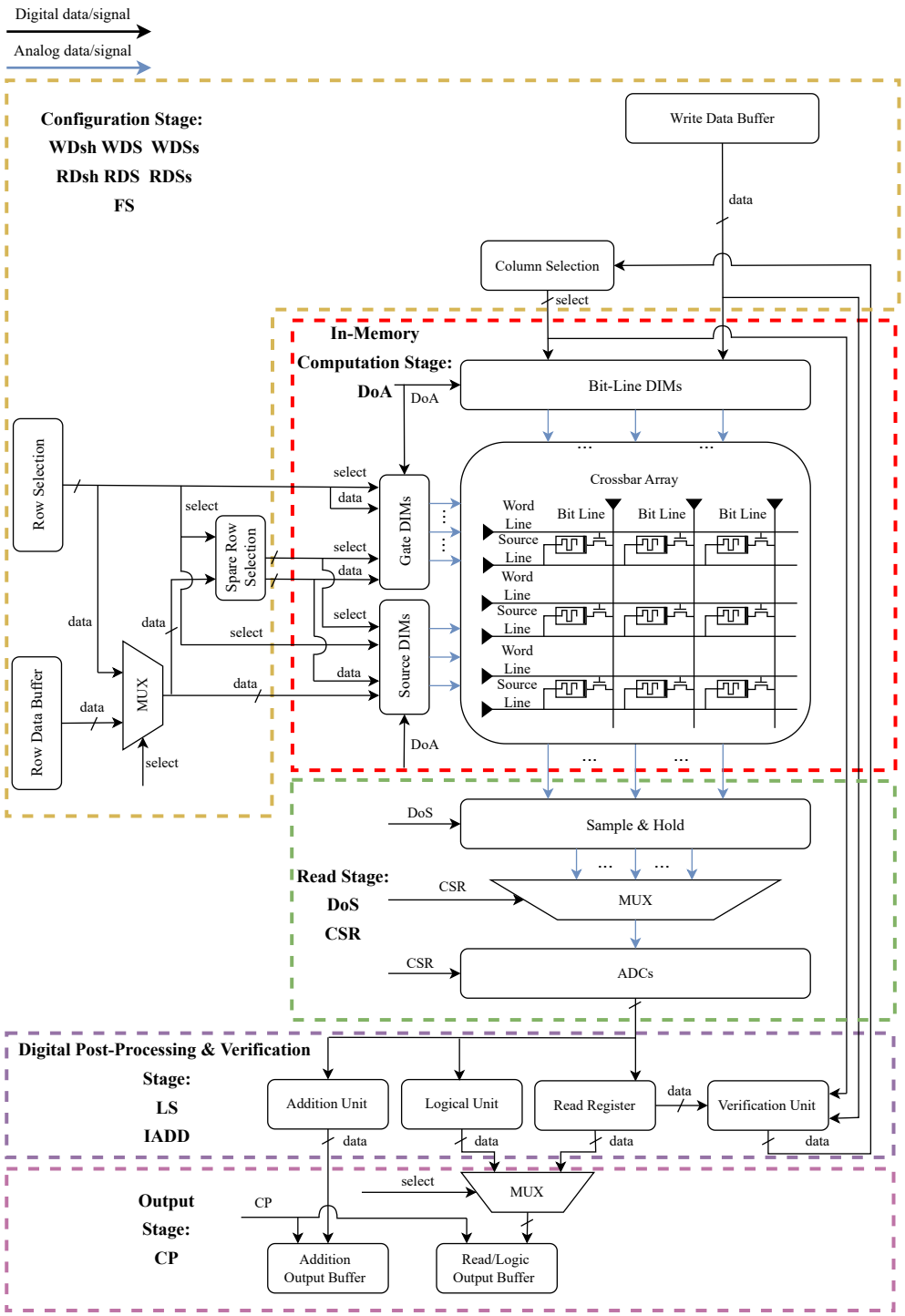


Figure 3.3: The relationship between the Write Data Buffer and its two interfaces

4

Implementation of the CIM Tile

This chapter delves into the detailed hardware architecture and implementation of the CIM tile's core digital periphery. We begin by describing the input/output buffers, which serve as intermediate storage between the external interface and the CIM tile. Subsequently, the chapter introduces the tile controller with two decoders for instruction translation and a stall detection unit for hazard prevention. The implementation of key selection modules—Column Selection, Row Selection, Row Data Selection, and Spare Row Selection—is then thoroughly examined. This is followed by a functional description of how key peripheral modules-namely the Digital Input Modules (DIMs), the Sample & Hold (SH) module, and Analog-to-Digital Converters (ADCs)-operate within the CIM tile. Furthermore, the chapter presents a dedicated Verification Unit for ensuring write reliability, a Logical Unit, and a three-stage Addition Unit designed to compute VMM results.

4.1. Buffer

As a computing unit that needs to interact with external devices, the CIM-tile requires a well-designed interface to receive input data from outside and send out computation results. To accomplish this, two input buffers (a write data buffer and a row data buffer) and two output buffers (one for storing results from the Logical Unit, and another for storing results from the Addition Unit) are implemented as the I/O interface between the CIM-tile and external devices. All four buffers use the same synchronous FIFO (First-In-First-Out) structure, with different parameter configurations tailored to their specific roles. This allows the CIM-tile to operate as an independent module without relying on an external controller to understand its internal details. Additionally, external devices can operate independently and remain unaffected by the CIM-tile's internal instruction execution state, improving the system's maintainability and scalability. The buffer design also enhances parallel processing by enabling data pipeline processing—while the CIM-tile processes current data, external devices can prepare the next batch of data at the same time.

4.1.1. Write Data Buffer

As shown in Figure 4.1, the Write Data Buffer serves as intermediate storage between the external interface and the CIM tile. The buffer receives data from outside and supplies it to the Bit Line DIMs during write operations.

The key architectural parameter of the Write Data Buffer is its width (buffer width), which is determined by two factors: the width of the input bus (connecting the external interface to the Write Data Buffer) and the number of crossbar columns. Two primary scenarios dictate how this width is determined.

In the first scenario, the input bus width matches the number of crossbar columns. This enables an entire row of data to be transferred into a single buffer chunk in one cycle. Consequently, the data can be written from the buffer to the Bit Line DIMs in a single operation. To support this, the buffer width is set equal to the number of crossbar columns. However, this scenario assumes a wide bus, which may not be practical in real implementations where bus width is often constrained.

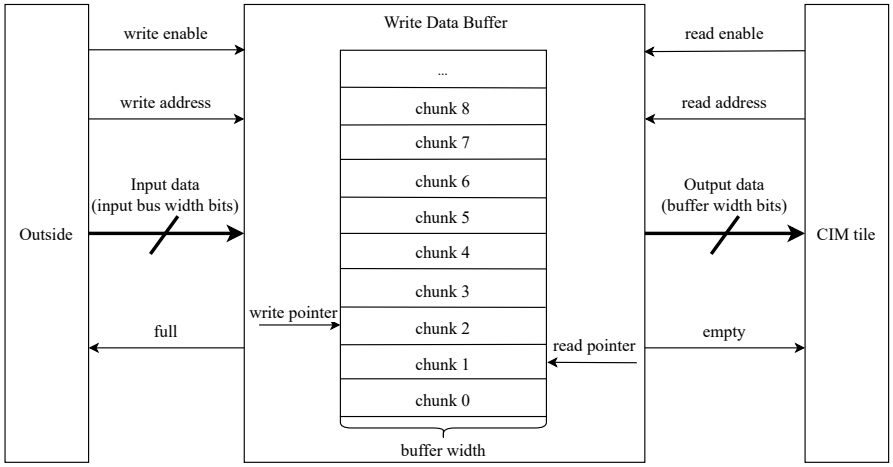


Figure 4.1: The relationship between the Write Data Buffer and its two interfaces

A more realistic scenario occurs when the input bus width is smaller than the number of crossbar columns. Two buffer-width configurations are considered:

The first configuration sets the buffer width equal to the number of columns. Each buffer chunk can hold an entire row of data, but filling it requires multiple bus transfers. Once filled, the data of an entire row can be shifted out in a single read cycle.

The second configuration sets the buffer width equal to the input bus width. Multiple narrower chunks collectively store one row of data. Reading out a full row requires as many cycles as there are chunks.

Using an illustrative 16-bit input bus width and 32 columns, both configurations require three cycles from an empty state: the first configuration needs two cycles to fill the full-row chunk and a third to shift it out; the second configuration writes the first half in cycle 1, writes the second half while shifting the first in cycle 2, then shifts the second half in cycle 3. However, once a whole row is buffered, the first configuration outputs the entire row in a single cycle, whereas the second requires two cycles, resulting in higher steady-state efficiency for the first approach. Furthermore, fewer transfers mean fewer `WDsh` instructions and reduced instruction memory.

Considering processing efficiency and instruction memory, we adopt the first configuration, setting the buffer width equal to the number of crossbar columns.

4.1.2. Row Data Buffer

As noted in the study of M.Z. Zahedi [41], the Row Data buffer temporarily holds one operand and then feeds it into the crossbar one bit at a time for computation operations, since the DIM drivers on each row usually support a single bit resolution (in this thesis, we assume the DIM resolution is 1-bit). Therefore, the multi-bit operands must be split into individual vectors. All least significant bits form one bit vector, all next-least significant bits form another vector, and so on up to the most significant bit. These vectors are sent in sequence from LSB to MSB, as illustrated in Figure 4.2.

The Row Data Buffer has two key architectural parameters: the buffer width and the buffer depth. The buffer width is set equal to the number of crossbar rows, enabling an entire bit vector to be sent to the crossbar array in a single cycle. The buffer depth is determined by the width of the input operand to the crossbar. The buffer depth is determined by the bit-width of the input operand to the crossbar. Considering area constraints, the buffer depth is set to be equal to the operand bit-width.

4.1.3. Output Buffer

There are two output buffers serving as the primary interface for transferring results from the CIM tile to the external system: the Read/Logic Output Buffer and the Addition Output Buffer.

The Read/Logic Output Buffer stores outputs from the Read Register (from read operations) and the

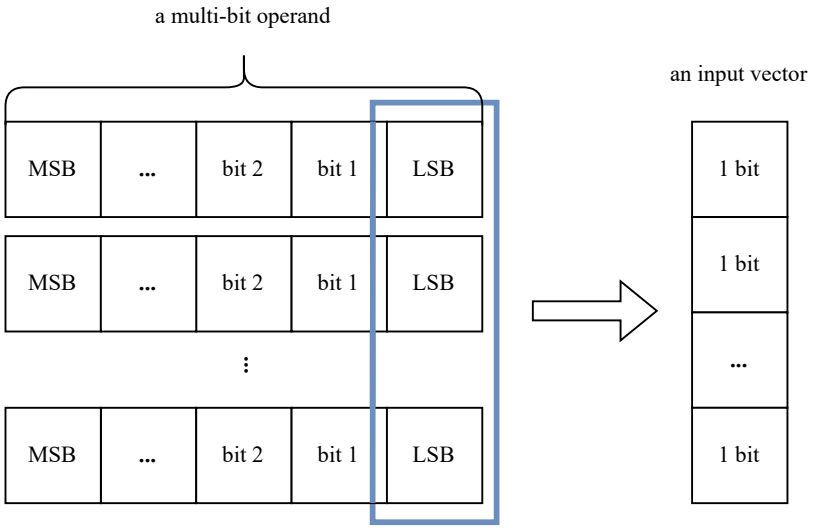


Figure 4.2: Sequential Feeding for Multi Bit Operands

Logical Unit (from logic operations). Since a read operation retrieves data from an entire row, the output from the Read Register has a width equal to the number of crossbar columns. Similarly, logic operations produce a single-bit result per column. Consequently, the width of this buffer is defined by the number of columns in the crossbar array.

The Addition Output Buffer stores results from the Addition Unit (from VMM operations). Its width is determined by the dimensions of the crossbar array and the operand bit-width, specifically as a function of the number of rows, columns, and the operand bit-width.

4.2. Tile Controller

This section describes the design of the tile controller, whose primary function is to execute instructions from the instruction memories. Figure 4.3 shows an overview of the tile controller. It consists of three main components: Decoder1, which decodes instructions for the configuration and execution stages; Decoder2, which handles decoding for the read, computation, and output stages; and a stall detection unit (Stall Detection), which manages pipeline hazards to ensure correctness and timing safety. (This section does not specify particular instruction encodings.)

4.2.1. Decoder1

This section describes the operation of the Decoder1, which receives and decodes instructions from the Instruction Memory1. Programs in this memory follow a structured sequence in which each operation begins with an `FS` instruction and terminates with a `DoA` instruction. First, Decoder1 translates the opcode field of the current instruction into internal opcode flags (e.g., `WDsh_op`, `RDsh_op`, `DoA_op`). When the `stall_1` signal, which is used to pause instruction execution in Decoder1, from the stall-detection unit equals '0', Decoder1 generates specific enable signals (e.g., `WDsh_E`, `RDsh_E`, `DoA_E`) based on these internal opcode flags. These enable signals are used to enable their corresponding functional modules. Furthermore, for instructions such as WDS and RDS, Decoder1 also produces associated index and data signals, such as `WDS_index`, `WDS_data`, `RDS_index`, and `RDS_data`.

For the `FS` instruction, Decoder1 generates the `CS_clear` and `RS_clear` signals based on its index field, which are sent to the Column Selection module and Row Selection module to clear the outputs from the previous operation. Following the execution of the `FS` instruction, Decoder1 produces a set of operation selection signals (`store_s1`, `read_s1`, `vrf_s1`, `AND_s1`, `OR_s1`, `XOR_s1`, and `VMM_s1`), which identify the current operation type and enable the DIMs to generate appropriate voltage values during the execution stage. Additionally, the `VMM_s1` signal controls data selection in the Row Data Selection

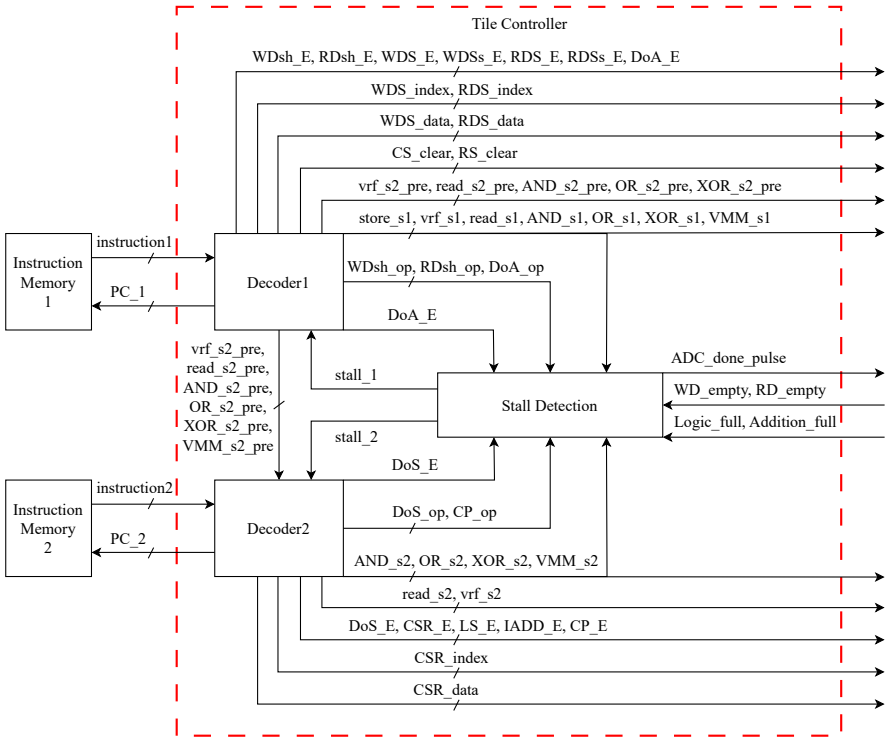


Figure 4.3: The Structure of the Tile Controller

module.

After executing a `DoA` instruction, the Decoder1 generates a set of pre-operation selection signals (`vrf_s2_pre`, `read_s2_pre`, `AND_s2_pre`, `OR_s2_pre`, `XOR_s2_pre`, and `VMM_s2_pre`) based on the current operation selection signals (`vrf_s1`, `read_s1`, `AND_s1`, `OR_s1`, `XOR_s1`, and `VMM_s1`). These pre-operation signals are necessary because Decoder2's instruction set does not include `FS`; thus, they perform the role of the `FS` instruction in Decoder2.

Among these signals, `vrf_s2_pre` and `read_s2_pre` are used to clear previous outputs in the Verification Unit, Read Register, respectively. The pre-operation selection signals, `AND_s2_pre`, `OR_s2_pre`, `XOR_s2_pre`, serve the same purpose specifically for the Logical Unit. The clearing operation is triggered when a unit receives the `DoS_E` signal from Decoder2 and its corresponding pre-operation selection signal equals '1'. It should be noted that the Addition Unit employs a separate clearing mechanism, which will be detailed in the section 4.11.

Decoder1 also manages the program counter, which holds the address for Instruction Memory1. During normal execution, when the `PC_1` signal is '0', the address `PC_1` increments sequentially. However, if a verification operation fails (`vrf_fail` = '1'), Decoder1 loads a branch address from the `decoder1_branch_vrf` registers. These registers are specifically designed to store the address of the DoA instruction that corresponds to the location of the DoA instruction within the store operation. This mechanism allows the controller to resume precisely at the write operation when verification fails, thereby enabling conditional control flow.

4.2.2. Decoder2

This section describes the operation of Decoder2, which receives and decodes instructions from Instruction Memory2. Programs in this memory follow a structured sequence in which each operation begins with a `DoS` instruction. First, Decoder2 translates the opcode field of the current instruction into internal opcode flags (e.g., `DoS_op`). When the `stall_2` signal, which is used to pause instruction execution in Decoder2, from the stall-detection unit equals '0', Decoder2 generates specific enable signals

(e.g., DoS_E) based on these internal opcode flags. These enable signals are used to enable their corresponding functional modules. For the instruction CSR, Decoder2 also produces associated index and data signals, such as CSR_index, CSR_data.

When processing a DoS_E instruction, Decoder2 utilizes the pre-operation selection signals received from Decoder1 to generate the operation selection signals for the computation and output stages in the pipeline. These operation selection signals are then used to route data and configure the appropriate functional modules for execution.

Decoder2 also manages its own program counter. During normal execution, when the stall_2 signal is '0', the address PC_2 increments sequentially. However, in case of a verification failure (vrf_fail = '1'), Decoder2 loads a branch address from the decoder2_branch_vrf registers, which are specifically used to store the address of the DoS instruction within the verification operation.

Furthermore, Decoder2 supports jal and jr instructions for subroutine handling. When a jal instruction is executed, PC_2 jumps to a target address generated by a dedicated adder, and the address of the next instruction following jal is saved into the back_up registers. When a jr instruction is subsequently executed, PC_2 is restored from the back_up registers, allowing the program to return to the point of origin after the subroutine call.

4.2.3. Stall Detection

The Stall Detection module identifies control hazards and, according to explicit hazard rules, stalls the two decoders to ensure pipeline correctness and timing safety. Concretely, it asserts Decoder1_stall, stall_2 under the following conditions:

- It asserts stall_1 to prevent Decoder1 from issuing WDsh_E/RDsh_E when the corresponding Write Data Buffer or Row Data Buffer is empty;
- It asserts stall_1 to block Decoder1 from issuing a new DoA_E until the analog outputs produced in the execution stage have been captured by the S&H module;
- It asserts stall_1 to hold Decoder1 while the CIM tile is performing verification;
- It asserts stall_2 to prevent Decoder2 from issuing DoS_E until new analog outputs are available from the execution stage;
- It asserts stall_2 to stall the Decoder2: if one buffer is full, Decoder2 stalls only the next CP instruction targeting that buffer, while CP instructions directed to the other buffer proceed.

Because analog components (crossbar array, S&H module, ADCs) do not inherently emit digital done pulses, the Stall Detection module includes dedicated, parameterizable counters that synthesize completion pulses to align the digital control with analog timing:

- During DoA, subsequent DoA instruction is blocked until the counter dedicated to the crossbar array asserts its completion pulse;.
- During DoS, the signal stall_2 is asserted and Decoder2 is held until the S&H counter produces its completion pulse;
- During CSR, stall_2 remains asserted until the ADC counter emits its completion pulse. When the ADC counter finishes, the module also generates an ADC_done_pulse to enable computation stage modules to sample the ADCs' outputs.

Together, these mechanisms guarantee forward progress while preventing timing violations and coordinating synchronous control with the intrinsic delays of the analog path.

4.3. Column Selection Module

This section introduces the implementation of the Column Selection module, which selects columns under the WDS and WDSs instructions. The module comprises three parts: a one-hot decoder, a set of column-selection blocks, and column mask backup registers.

The first part is a one-hot decoder that decodes the index of WDS instruction (WDS_index). As shown in Figure 4.4, the m-bit WDS_index is fed into the decoder, which generates an n-bit block-enable vector.

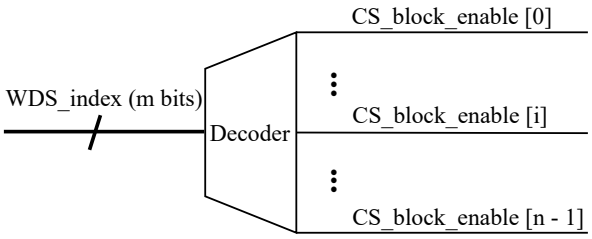


Figure 4.4: Implementation of the One-hot Decoder of the Column Selection Module

Each enable bit connects to its corresponding column-selection block introduced next. The relationship among the parameters is:

$$n = 2^m. \tag{4.1}$$

For example, with $m = 3$ and $W_{\text{data}} = 6$ (the width of the data field in the `WDS`/`RDS` instruction), if the `WDS_index` is `001`, the decoder outputs the 8-bit enable vector `00000010`, which activates only the second column-selection block (`CS_block_enable[1] = '1'`). Each block thus contains $W_{\text{data}} = 6$ columns.

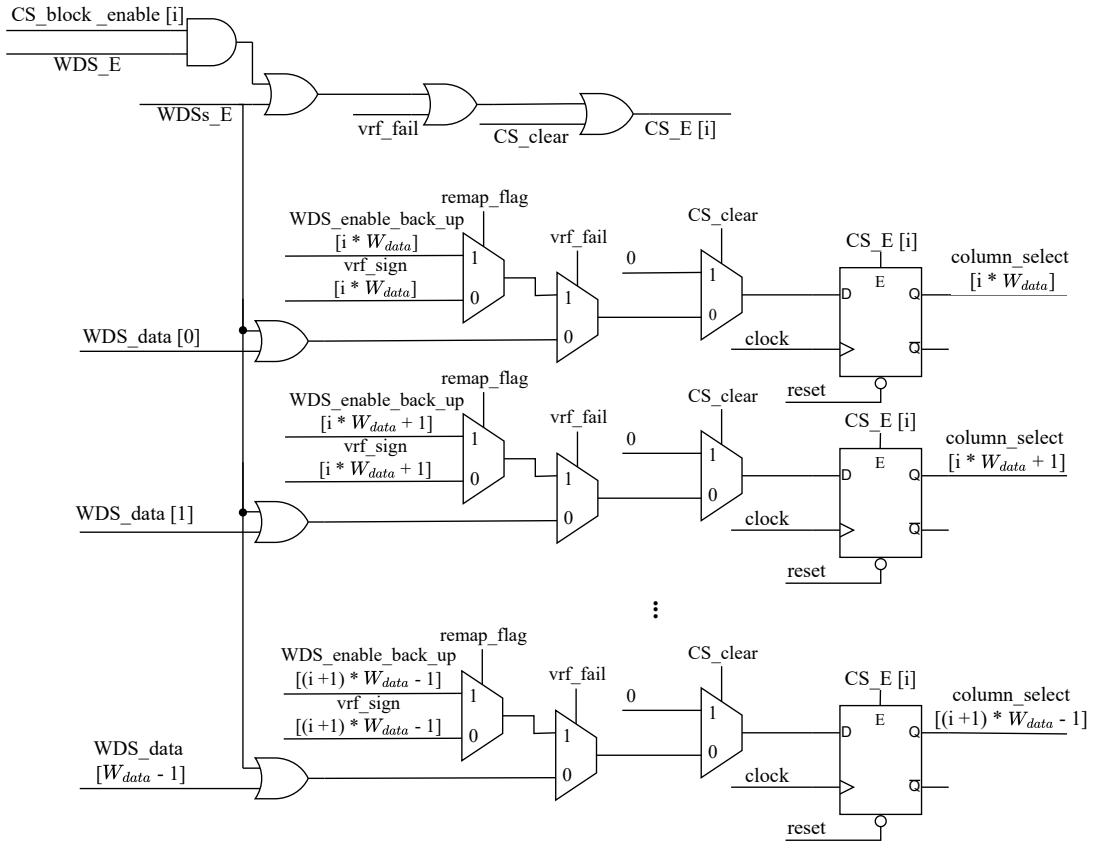


Figure 4.5: Implementation of the Column-Select Block

The second part consists of the column-selection blocks, which select columns within each block. Figure 4.5 shows the schematic of a block. Each block contains W_{data} registers with `reset` and `enable` con-

trol. The input to each register is determined by `CS_clear`, `vrf_fail`, `remap_flag`, and the `WDS_data`. The combinational logic for each block's `CS_E` implements the following Boolean expression:

$$CS\_E[i] = CS\_block\_enable[i] \wedge WDS\_E[i] \vee WDSs\_E \vee vrf\_fail \vee CS\_clear, \tag{4.2}$$

where

$CS\_E[i]$	the enable signal of the i^{th} column-selection block's registers,
$CS\_block\_enable[i]$	the i^{th} column-selection block enable signal from the one-hot decoder,
WDS_E	the enable signal generated by the WDS instruction,
$WDSs\_E$	the enable signal generated by the WDSs instruction,
vrf_fail	the verificarion result from the Verification Unit,
CS_clear	clear signal generated by the `FS` instruction.

At the start of each write operation, `CS_clear` is asserted, resetting all registers in the Column Selection module to zero. This removes any residual values from previous writes and ensures correctness for the current operation. When a `WDSs` instruction is issued, the `WDSs_E` signal is set to 1. All register inputs are forced to 1 and all registers are enabled; consequently, all `column_select` signals become 1 (select all columns). When a `WDS` instruction is issued, the `WDS_E` signal is set to 1. Only the registers in the block selected by the one-hot enables are updated, and that block's `column_select` signals are written with the incoming `WDS_data`; the `column_select` signals in the other blocks remain unchanged.

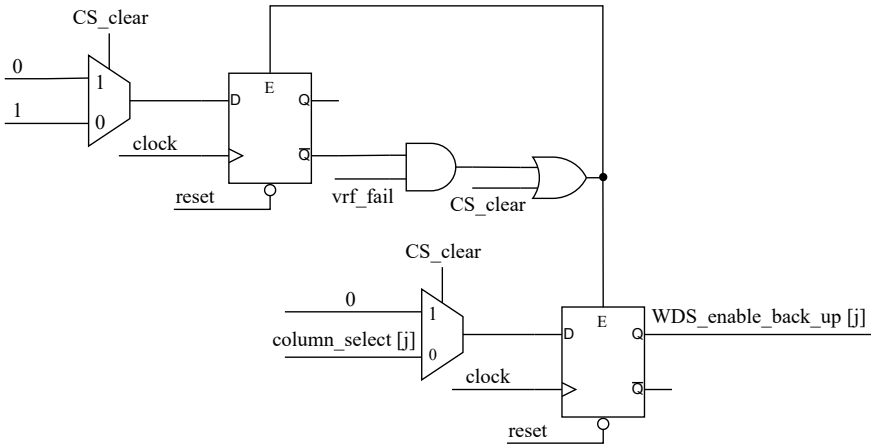


Figure 4.6: Implementation of the Per-Column Mask Register

During verification, the first assertion of `vrf_fail` triggers the design shown in Figure 4.6 to capture the current column-selection mask (`column_select`) into the `WDS_enable_backup` registers. This stored mask enables the same column access pattern to be reapplied during subsequent remapping to a spare row.

When the `vrf_fail` is asserted, this tile responds differently based on the `remap_flag`. If `remap_flag` is 0, all relevant registers are updated using the per-column verification mask (`vrf_sign`) supplied by the Verification Unit, where 1 indicates a column that was incorrectly written. In this case, the tile rewrites only the marked columns in the original row, preserving columns that were correctly programmed.

If `remap_flag` is 1, the rewrite is considered to have failed again and the target row is deemed unreliable. The tile then disables the faulty row and initiates data remapping to a spare row. To support this process, the previously saved column-selection mask is restored from the `WDS_enable_backup` registers, and all relevant registers are updated from these backup values. This ensures the data is accurately written in the new spare row.

4.4. Row Selection and Row Data Selection

This section describes the implementation of the Row Selection module and the Row Data Selection module. The Row Selection module and the Row Data Selection module implement the `RDS` and `RDSs` instructions. The Row Selection module generates `row_select` signals based on the instructions, and the Row Data Selection module outputs the *row_select_data* signals depending on the operation executed. Section 4.4.1 introduces the detailed implementation of the Row Selection module, and section 4.4.2 shows the implementation of the Row Data Selection module.

4.4.1. Row Selection

The Row Selection module comprises three parts: a one-hot decoder, a set of row-selection blocks, and a bank of registers that generates the `row_disable` signals for deactivating specified rows.

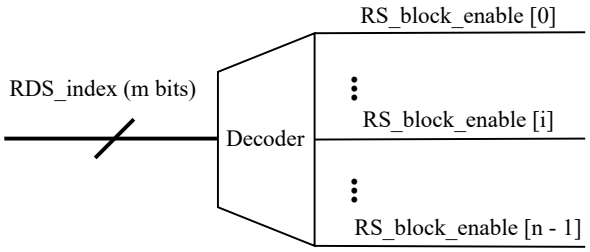


Figure 4.7: Implementation of the One-hot Decoder of the Row Selection Module

The one-hot decoder shown in Figure 4.7 converts the m-bit `RDS_index` into an n-bit `RS_block_enable` signals. Each `RS_block_enable` signal drives one row-selection block, and each row-selection block comprises W_{data} registers.

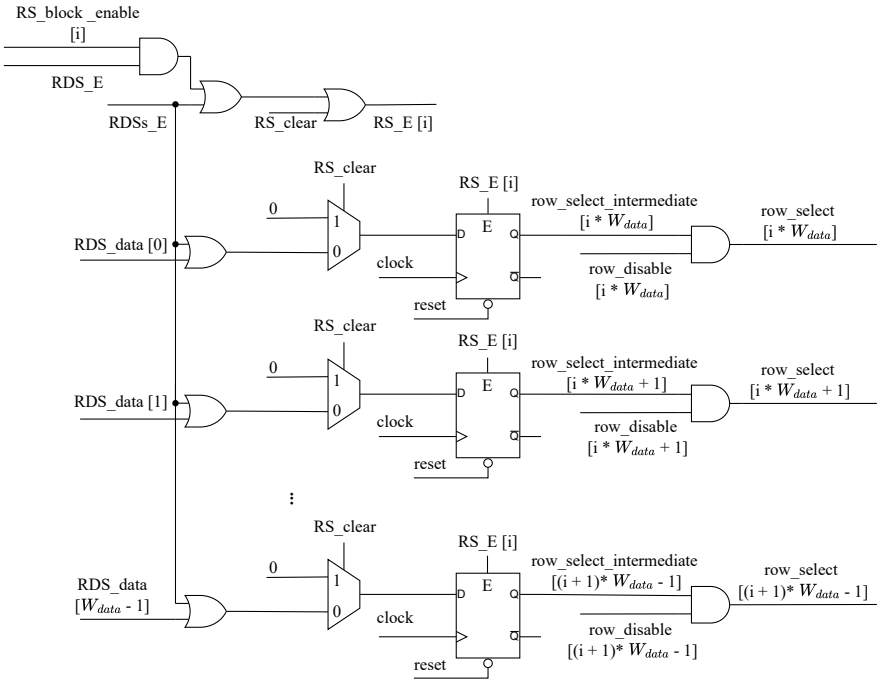


Figure 4.8: Implementation of the Row-Selection Block

The second part is the row-selection block, which either selects rows within the block enabled by the `RS_block_enable` signal from the decoder or selects all rows as commanded by the `RDSs` instruction.

Figure 4.8 shows the schematic of a row-selection block. Each block contains W_{data} registers with `reset` and `enable` controls. Each register's input is determined by the `RS_clear` signal (generated by the `FS` instruction) and by the `RDS_data`. The combinational logic that generates the block `RS_E` implements the following Boolean expression:

$$RS\_E[i] = RS\_block\_enable[i] \wedge RDS\_E[i] \vee RDSs\_E \vee RS\_clear, \tag{4.3}$$

where

$RS\_E[i]$	the enable signal of the i^{th} row-selection block's registers,
$RS\_block\_enable[i]$	the i^{th} row-selection block enable signal from the one-hot decoder,
RDS_E	the enable signal generated by the RDS instruction,
$RDSs\_E$	the enable signal generated by the RDSs instruction,
RS_clear	clear signal generated by the `FS` instruction.

At the start of each write, read, logic, or VMM operation, `RS_clear` is asserted to clear the registers in all row-selection blocks, removing any residual values from previous operations and ensuring correctness for the current operation.

When a `RDSs` instruction is executed, the `RDSs_E` forces every register input to 1 and enables all registers in all row-selection blocks, driving all `row_select_intermediate` signals high. When a `RDS` instruction is executed, `RDS_E` is asserted. In this case, the W_{data}-bit `RDS_data` are presented to every block, but only the registers in the row-selection block selected by the `RS_block_enable` signals actually sample the data and update their outputs; other blocks retain their values.

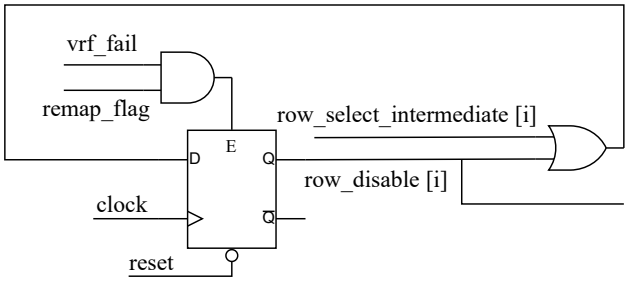


Figure 4.9: Implementation of Row Disable Selection

The third part is the row disable selection logic (Figure 4.9), which disables rows that exhibit errors. When the `remap_flag` is '1' and the `vrf_fail` is also '1' (that is, a rewrite was attempted but verification still failed), the `row_disable` signal is updated to disable the target row.

4.4.2. Row Data Selection

Figure 4.10 depicts the implementation of the Row Data Selection module. The `row_select_data` signals driven to the Source DIMs depend on the operation: for store, read, logic, and verification (VRF) operations, it equals the `row_select_intermediate` signals produced by the Row Selection module; for VMM operation, it takes the value of `row_data` provided by the Row Data Buffer.

4.5. Spare Row Selection

The Spare Row Selection module consists of a spare-row counter (Figure 4.11) and a bank of selectors. For each spare row i, a spare-row selector (Figure 4.13) generates its `spare_row_select` signal indicating whether that spare row is activated, and a spare-row data selector (Figure 4.13) supplies the corresponding data to the Source DIMs.

The Spare Row Counter generates the R_s-bit `spare_row_sel` signal-where R_s denotes the total number of spare rows-to select the targeted Spare Row Selector. Each time a spare row is used, the counter increments to select the next available spare row in sequence.

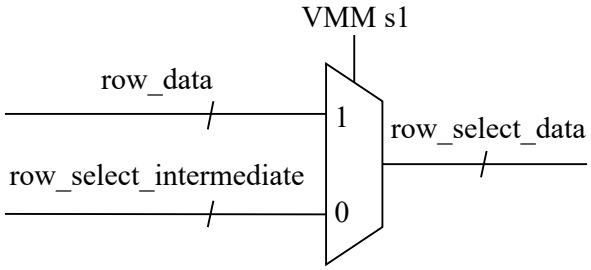


Figure 4.10: Implementation of Row Data Selection

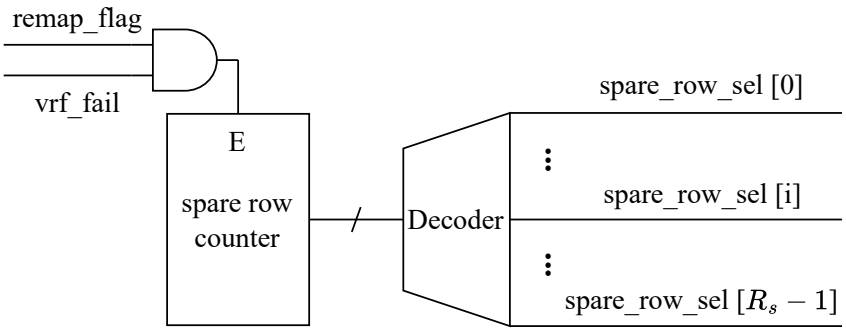


Figure 4.11: Implementation of the Spare-Row Counter

The spare-row selector is illustrated in Figure 4.12. Each selector contains R per-row registers, where R is the number of original rows (excluding spare rows). The selector operates in two scenarios described below.

In the first scenario, the R_s-bit `spare_row_sel` signal selects the corresponding spare-row selector, enabling all the registers within this spare-row selector. The R-bit `spare_row_select_pre` signal is loaded from the R-bit `row_select_intermediate` signal, which is generated by the Row Selection module. Since each write operation selects only a single row, the `row_select_intermediate` signal is a one-hot code, with a single '1' denoting the selected original row. Therefore, the `spare_row_select_pre` signal is likewise one-hot. This mechanism ensures that a spare row is selected if and only if its corresponding faulty original row is selected in the Row Selection module, establishing a precise one-to-one remapping.

The second scenario occurs when a spare row that has been remapped is found to be faulty. In this case, the tile disables the faulty spare row and initiates a new remapping to the next available one. The disabling mechanism works by clearing the corresponding register in the spare-row selector.

For example, suppose faulty spare row i was remapped from original row j. In this case, the signal `spare_row_select_pre[j]` remains '1'. Since the `row_select_intermediate[j]` signal, which reflects the currently selected row in the Row Selection module, is also '1', the resulting `row_masked[j]` signal becomes '1'. This signal drives the input multiplexer of the j-th register in the selector, forcing its D input to '0' instead of `row_select_intermediate[j]`. When a new remap operation starts, this register is updated, clearing the stored value and thereby disabling the faulty spare row i.

The spare-row data selector, implemented as shown in Figure 4.13, routes the data from the Row Data Selection module to the Source DIMs. Because the `spare_row_select_pre` signal of a remapped spare row's spare-row selector is one-hot, only the data from the original row that this spare row replaces is gated through.

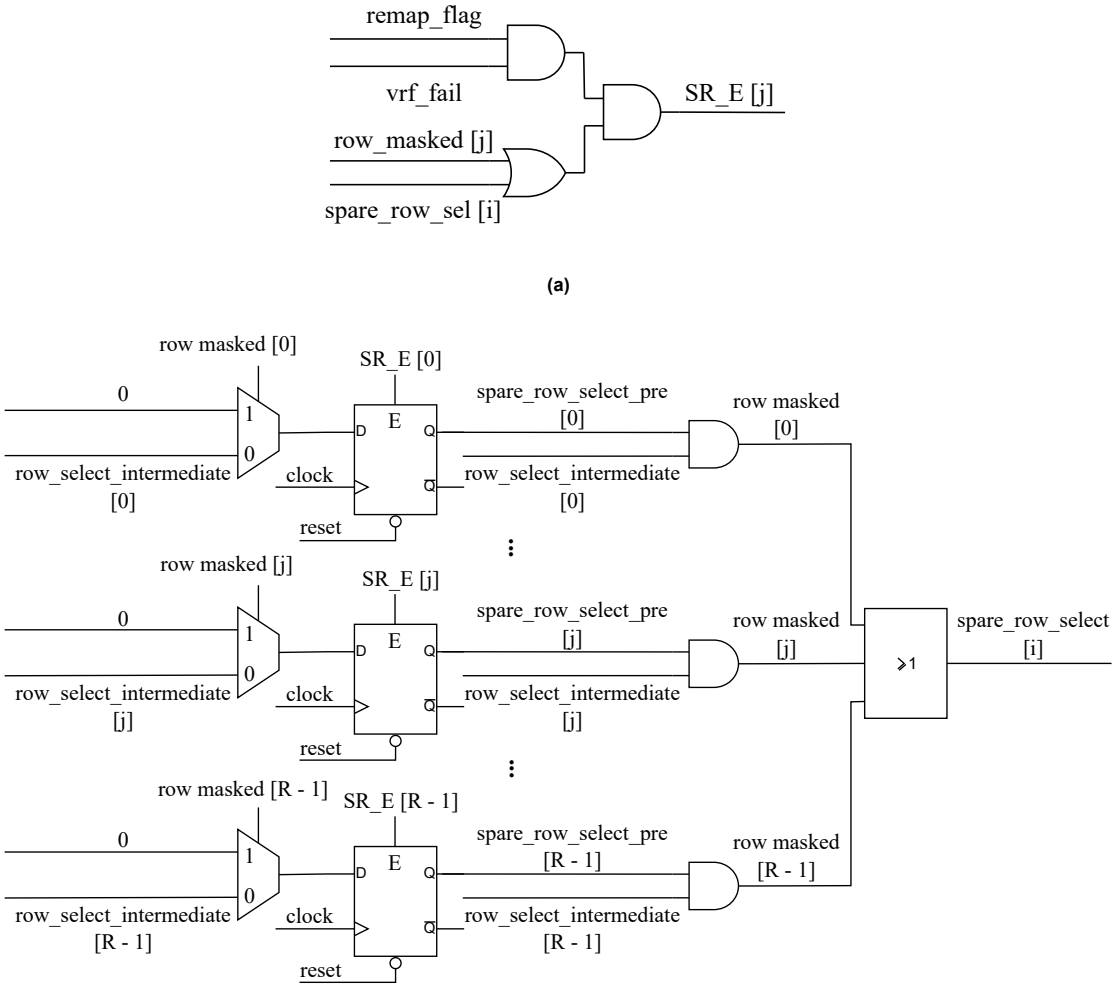


Figure 4.12: (a) Enable Logic of the Registers in the Spare-Row Selector (b) Spare-Row Select Signal Generation Logic

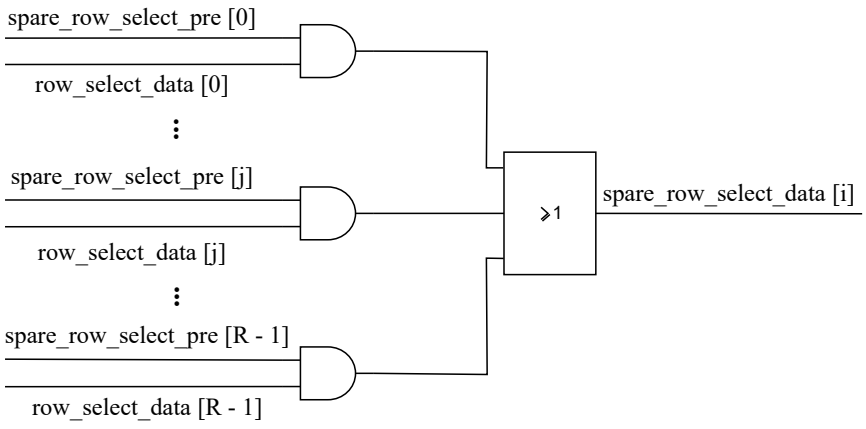


Figure 4.13: Implementation of the Spare-Row Data Selector

4.6. Digital Input Module

In the CIM-Tile architecture, the operating voltages of the crossbar are usually different from the power supply voltage of the digital logic circuit, so a dedicated interface circuit is required to achieve the conversion from the digital domain to the analog domain. This work is completed by the Digital Input Module (DIM), which converts the voltages of the digital modules into the analog operating voltages needed by the crossbar array.

As shown in Figure 3.3, there are three kinds of DIMs required for the crossbar array to complete the operations. These DIMs are activated when executing the `DoA` instruction. The word line DIM is used to control the conduction state of the transistors in a row, thereby realizing the row selection function of the array. When the DIM signal is high (logic '1'), the transistors of the corresponding row are turned on; when the DIM signal is low (logic '0'), the transistors are turned off. For the write operation, applying a voltage difference between the source line and the bit line changes the resistance state of the memristor. For other operations, such as read, verification, logic, or VMM operations, the bit line DIMs do not work. The source line DIMs provide the voltages to complete these operations on the crossbar array.

4.7. Sample & Hold Module

In this CIM Tile structure, the sample & hold (SH) module is placed immediately downstream of the crossbar analog outputs and acts as a buffer for the subsequent ADCs. The SH module is activated upon receiving the `DoS_E` enable signal from the Decoder2. Since the ADCs acquire the held analog results from the SH module, a new operation can be performed on the crossbar array during digitization. This module improves efficiency by allowing parallel processing.

4.8. Analog to Digital Converter

In the CIM Tile structure, Analog to Digital Converters (ADCs) are needed to convert the analog results into the digital domain. However, due to the power consumption and size constraints of ADCs, it is not possible to allocate one ADC per column. Therefore, a shared ADC solution is employed, allowing multiple columns to share a single ADC. The detailed implementation is shown in Figure 4.14. This approach involves a multiplexer and ADCs controlled by the `CSR` instruction. The crossbar array columns are grouped, with each group assigned to a dedicated ADC. The multiplexer selects the target column within each group based on the `CSR_index`. The `CSR_data` selects the corresponding ADC to perform the conversion.

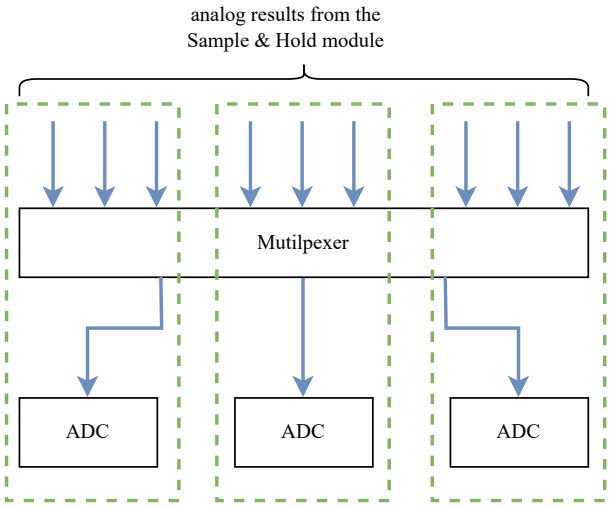


Figure 4.14: Example of a shared-ADC solution

For instance, consider a crossbar array with nine columns (numbered 0–8) and three ADC channels

(ADC0–ADC2). The columns are grouped into three groups of three columns each, with one ADC assigned to each group as follows:

- **Group 0 (ADC0):** Columns 0, 1, 2;
- **Group 1 (ADC1):** Columns 3, 4, 5;
- **Group 2 (ADC2):** Columns 6, 7, 8.

In this case, the `CSR` index is set to '01', meaning each group selects its second column (Group 0 selects Column 1, Group 1 selects Column 4, Group 2 selects Column 7). The CSR mask data is set to '011', which disables ADC0 and enables ADC1 and ADC2. As a result, the multiplexer routes Column 1 to ADC0, Column 4 to ADC1, and Column 7 to ADC2. Since ADC0 is disabled, Column 1 is ignored. Only ADC1 converts Column 4, and ADC2 converts Column 7; all other columns remain inactive in this case.

4.9. Verification Unit

As mentioned in the study by Yang [40], the memristors have non-ideal properties due to programming inaccuracies and device imperfections from fabrication. These issues can cause errors or mismatches in the computed results. Therefore, it is important for the verification unit to check the correctness of the value written in the crossbar array.

The Verification Unit consists of two main components: the verification logic and the remap flag generation logic.

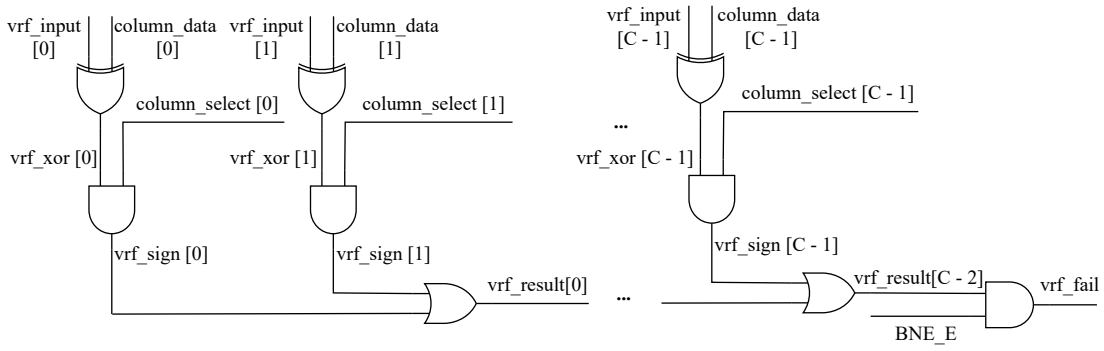
Figure 4.15(a) shows the detailed implementation of the verification logic. The write verification logic compares the crossbar output (stored in the Read Register) with the expected data from the Write Data Buffer. A mismatch in any bit triggers the `vrf_sign` signal to '1'. This comparison is implemented using an XOR gate. To ensure only the columns that were written to are verified, the XOR output is gated with the `column_select` signal from the Column Selection module through an AND gate. An OR chain is used to propagate any error detection, aggregating the `vrf_sign` signals from all columns. When a BNE instruction is executed, the `BNE_E` pulse samples the final result in this chain, `vrf_result[C-2]`, to generate the `vrf_fail` signal, indicating a verification failure.

The remap flag generation logic, illustrated in Figure 4.15(b), operates as follows: the remap flag is cleared at the start of every write operation. If a verification fails for the first time, a rewrite operation is triggered, and the remap flag is set to '1'. If the rewrite also fails, a remap operation is initiated, and the remap flag is cleared.

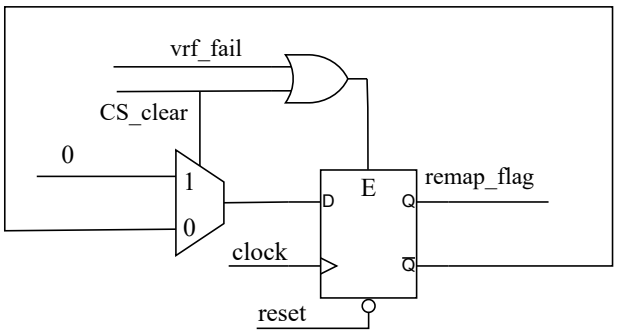
4.10. Logical Unit

During an in-memory logic operation on two selected rows, the crossbar produces analog results, which are digitized by the ADCs. However, this digital values still requires interpretation to become definitive logic states. This is the function of the Logical Unit, which judges the digital value to output the final logic '1' or '0'.

The implementation of the Logical Unit is illustrated in Figure 4.16. As logical operations are executed on two selected rows, each column produces a single analog signal. This signal is quantized into a digital value with three possible outputs: 0, 1, or 2, which correspond to the resistance states of the two memristor cells: both in high-resistance, one in high-resistance and the other in low-resistance, and both in low-resistance, respectively. Since a 1-bit value can only represent two states, a 2-bit representation is required to encode all three outputs. Based on this representation, the Logical Unit implements the three logic operations as follows: for the AND operation, a result of '1' is generated only when the output is '10' (both cells in low-resistance state), for which the higher bit is directly selected as the result; for the OR operation, a result of '1' is produced for outputs '01' or '10' (both or at least one cell in low-resistance state), performing a logical OR operation on the two bits of the output; for the XOR operation, a result of '1' corresponds to output '01' (differing resistance states), for which the lower bit is directly selected as the result. The final output for each column is then selected by the operation-selection signals (`AND_s2`, `OR_s2`, `XOR_s2`) from these computed results.



(a)



(b)

Figure 4.15: (a) Implementation of the Verification Logic (b) Remap Flag Generation Logic

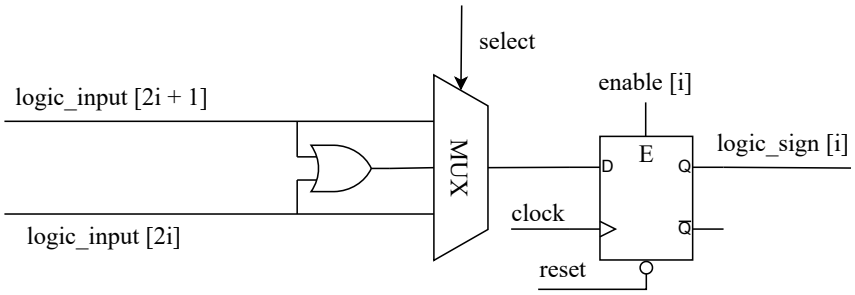


Figure 4.16: Implementation of Logical Unit per column

4.11. Addition Unit

This section introduces an efficient addition unit as a digital peripheral for the memristor crossbar, which can perform an unsigned VMM operation. The design of this addition unit is divided into the following three stages:

1. Analog accumulation stage: This stage accumulates the results from a column, which is used for addressing the limited resolution of ADCs;
2. Multi-column sliding addition stage: This stage sums intermediate results across columns from the analog accumulation stage using a sliding method to address the limited number of resistance levels in each memristor. In this thesis, we assume each memristor stores a single bit;
3. Input vector segment addition stage: This stage combines the partial results from each sub-vector computation through weighted accumulation to generate the final output (Due to the limited DIM resolutions, the input data is partitioned into multiple sub-vectors for sequential array processing).

Figure 4.17(a) shows an example implementation of a 3-bit VMM operation. Each matrix weight is represented at 3-bit resolution and mapped onto the crossbar array. For example,

$$d \longrightarrow \{d_2,\, d_1,\, d_0\},$$

where d_k (for $k = 0, 1, 2$) denotes the k-th bit of d (LSB at $k = 0$, MSB at $k = 2$), and each d_k is programmed into its corresponding position.

The 3-element vector $\{a, b, c\}$ serves as the input vector and is decomposed into three binary sub-vectors:

$$\{a, b, c\} \;\longrightarrow\; \big\{\{a_0, b_0, c_0\},\; \{a_1, b_1, c_1\},\; \{a_2, b_2, c_2\}\big\},$$

where each sub-vector is composed of the bits from the same significance position across all three elements(a, b, c).

Figure 4.17(b) uses the calculation of output element z_0 as an example. It provides a detailed view of where the weights that contribute to the calculation of z_0 are mapped on the crossbar array, as well as the input sequence of the sub-vectors.

4.11.1. First Stage: Analog Accumulation Stage

An ideal scenario for the first stage occurs when the ADC has sufficient resolution to convert the entire output of a bitline in a single step, as exemplified in Figure 4.18. Given that both the input vector elements and the weights are binary, the sum of products for any column (e.g $a_0d_0 + b_0g_0 + c_0j_0$, $a_0d_1 + b_0g_1 + c_0j_1$, $a_0d_2 + b_0g_2 + c_0j_2$) is guaranteed to be an integer between 0 and 3. The minimum ADC resolution in this example required is therefore 2-bit, as it can represent the full range of possible outcomes from 0 to 3. Under this ideal condition, VMM operation on the memristor crossbar array can be performed with maximum parallelism: all source lines are activated simultaneously, enabling each input vector element to be multiplied in parallel with its corresponding row weights, and the resulting column currents are digitized in one ADC operation per column.

However, this ideal scenario hinges on the ADC having sufficient resolution to represent the full range of the summed current. A practical limitation arises when the ADC resolution is too low; the number of available quantization levels becomes insufficient to represent all possible values of the entire column's summed current. Figure 4.19 shows the worst case where the ADC resolution is limited to 1 bit. Given that a 1-bit ADC performs only a single-threshold quantization, it cannot resolve the summed result from multiple cells in a column. To address this problem, the first stage employs a partitioning strategy: the weight array is divided into multiple sections (illustrated in blue, green, and red) that are processed sequentially. The full computation is completed over three sequential phases. In each phase, the process involves: (1) activating wordlines to load elements of the input vector, (2) performing column-by-column current sampling, and (3) generating accumulated outputs. The detailed computational flow proceeds as follows:

1. First Phase (Blue section):

- The first row's wordline is activated, and input a_0 is multiplied with weights $\{d_2, d_1, d_0\}$;

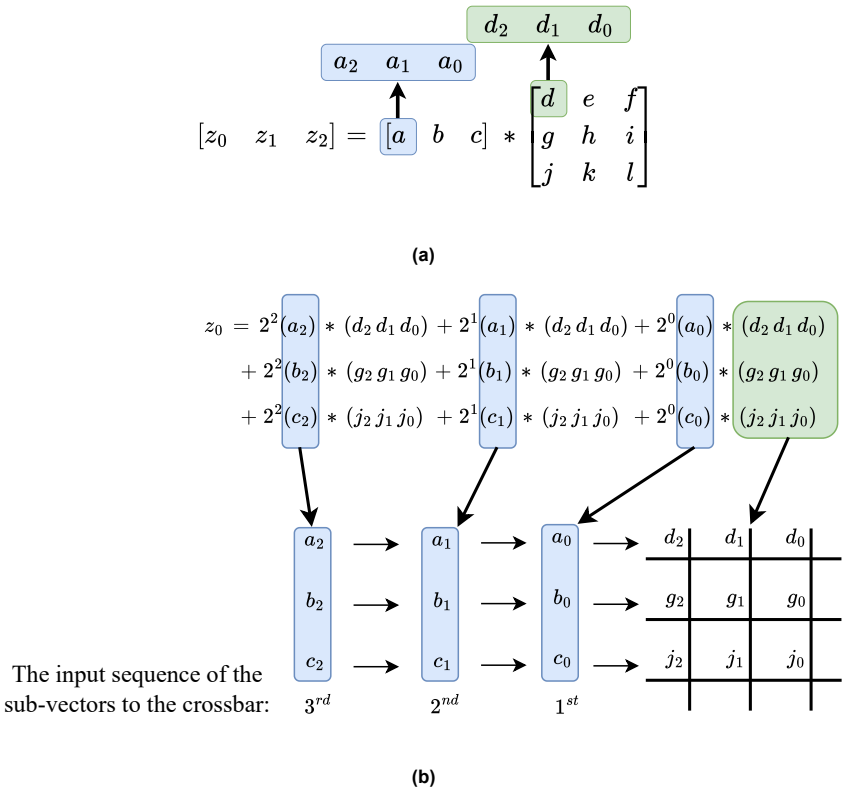


Figure 4.17: (a) An example of VMM operation with 3-bit operands (b) mapping of data to the crossbar as well as the input sequence of the sub-vectors

- The ADC sequentially samples the 1-bit product from each column in the sequence of BL1, BL2, and BL3;
- A demultiplexer routes the outputs $\{a_0d_2, a_0d_1, a_0d_0\}$ to their corresponding intermediate-result registers (R_2, R_1, R_0), completing the initial data loading.

2. Second Phase (Green section):

- The second row's wordline is activated, and input b_0 is multiplied with weights $\{g_2, g_1, g_0\}$;
- The ADC samples the new 1-bit products along BL1, BL2, BL3 in sequence. Each sampled value is then accumulated with its corresponding intermediate result register value;
- The data of the registers is updated to $\{a_0d_2 + b_0g_2,\ a_0d_1 + b_0g_1,\ a_0d_0 + b_0g_0\}$.

3. Third Phase (Red section):

- The last row's wordline is activated, and input c_0 is multiplied with weights $\{j_2, j_1, j_0\}$;
- The ADC samples $\{c_0j_2, c_0j_1, c_0j_0\}$ in sequence, which are added to the intermediate-result register values;
- Unlike previous phases, this phase outputs the complete results to the first stage output register (R_{stage1}) starting from the least significant position: first writing $(a_0d_0+b_0g_0+c_0j_0)$ to the register, followed by $(a_0d_1+b_0g_1+c_0j_1)$, and finally $(a_0d_2+b_0g_2+c_0j_2)$. Also, the intermediate registers are cleared in sequence for the next computation cycle.

In the hardware implementation, the `CSR`, `LS`, and `IADD` instructions work collaboratively to execute the analog accumulation stage. The `CSR` instruction selects which intermediate result register is updated. The selected register's current value is added to the ADC output, and the resulting sum is written back to the same register. The `LS` instruction sets a Last section flag upon execution. In each subsequent

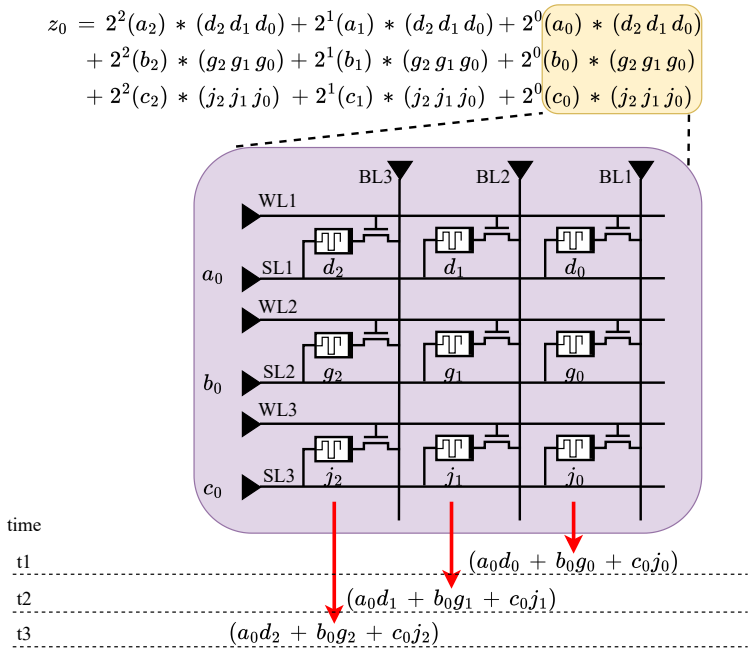


Figure 4.18: Ideal scenario of the analog accumulation stage

CSR instruction following the LS instruction, the adder output is routed to the first stage output register (R_{stage1}), while the corresponding intermediate result register is cleared. The IADD instruction clears R_{stage1} to ensure correct computation for the next operand, as a single ADC may be responsible for processing multiple operands.

4.11.2. Second Stage: Multi-column sliding addition stage

In the first stage of the addition unit, the output represents the result of the computation between the input sub-vector and each column of the weight matrix. Since each column corresponds to a different bit position (e.g., column 1 for the least significant bit, column 2 for the next, and so on), these results must be combined with weights corresponding to powers of two. To achieve this weighted summation efficiently, this stage employs a sliding method: only the higher bits (all bits except the least significant bit) of the previous step's result are retained for further addition, while the least significant bit remains unchanged and is directly placed in its corresponding position in the output of this stage.

Figure 4.20 shows a detailed example of the second stage, and the computational flow proceeds as follows:

- Add the result of the BL1 to the temporary register $R_{\text{stage2 temp}}$. The least significant bit (LSB) of the sum is placed at the first bit position of the output, while the higher bits are stored in $R_{\text{stage2 temp}}$;
- Add the result of the BL2 to $R_{\text{stage2 temp}}$. The LSB of the sum is placed at the second bit position of the output, while the higher bits update the $R_{\text{stage2 temp}}$;
- Add the result of the BL3 to $R_{\text{stage2 temp}}$. The complete sum is placed directly into the remaining output positions.

In the hardware implementation, the CSR, LS, and IADD instructions work collaboratively to perform the weighted accumulation. The execution of the LS instruction initiates this stage. Following this, each subsequent CSR instruction stores the higher bits (excluding the least significant bit) of the current adder output into the temporary register ($R_{\text{stage2 temp}}$), while the LSB is directly mapped to the corresponding position in the output of this stage. The IADD instruction transfers the adder output directly to the output of the second stage and clears $R_{\text{stage2 temp}}$, preparing it for the next weighted accumulation.

$$z_0 = 2^2(a_2) * (d_2\, d_1\, d_0) + 2^1(a_1) * (d_2\, d_1\, d_0) + 2^0(a_0) * (d_2\, d_1\, d_0)$$
$$+ 2^2(b_2) * (g_2\, g_1\, g_0) + 2^1(b_1) * (g_2\, g_1\, g_0) + 2^0(b_0) * (g_2\, g_1\, g_0)$$
$$+ 2^2(c_2) * (j_2\, j_1\, j_0) + 2^1(c_1) * (j_2\, j_1\, j_0) + 2^0(c_0) * (j_2\, j_1\, j_0)$$

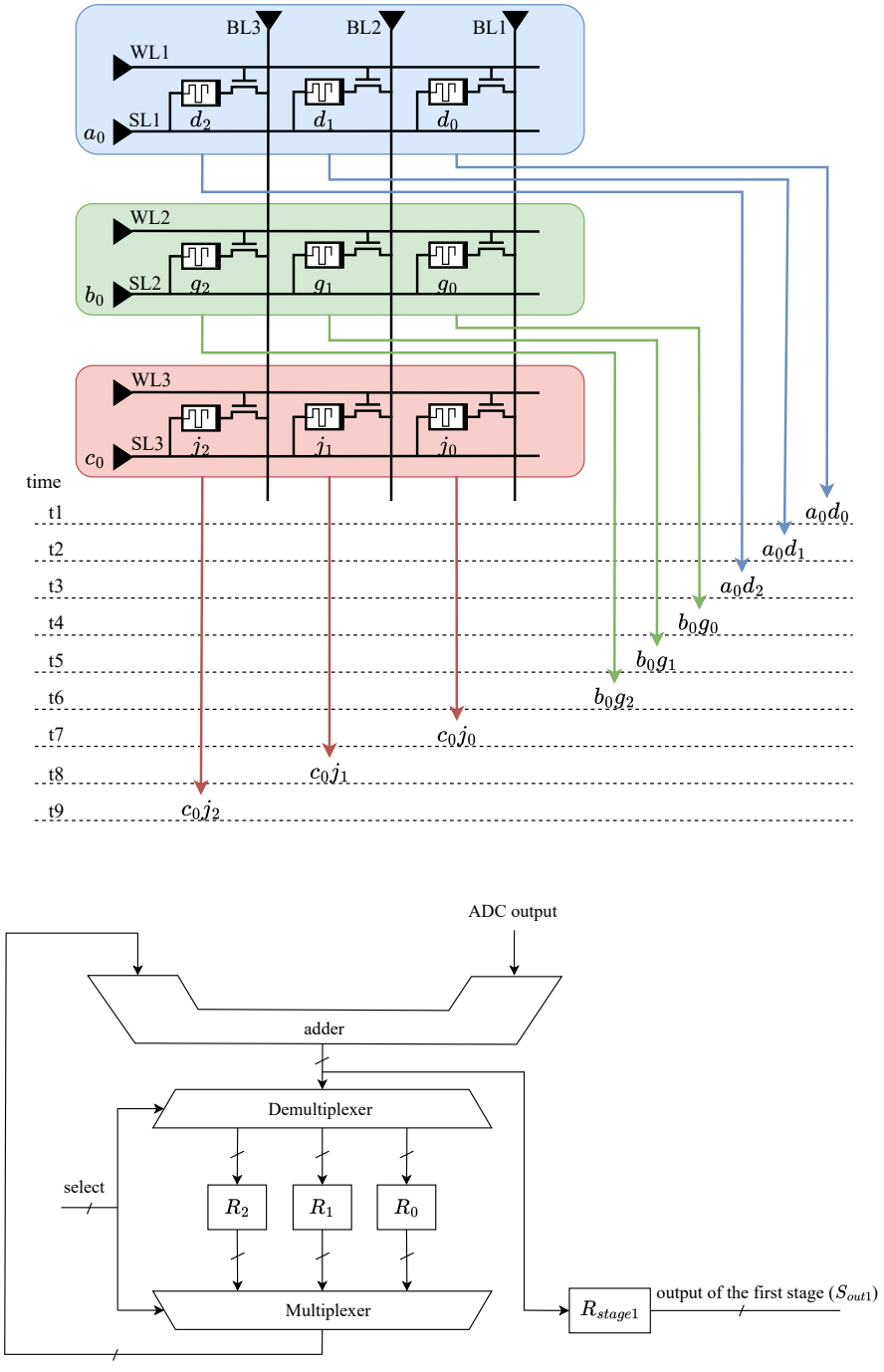


Figure 4.19: Example of the Analog Accumulation Stage

$$z_0 = 2^2(a_2) * (d_2\, d_1\, d_0) + 2^1(a_1) * (d_2\, d_1\, d_0) + 2^0(a_0) * (d_2\, d_1\, d_0)$$
$$+ 2^2(b_2) * (g_2\, g_1\, g_0) + 2^1(b_1) * (g_2\, g_1\, g_0) + 2^0(b_0) * (g_2\, g_1\, g_0)$$
$$+ 2^2(c_2) * (j_2\, j_1\, j_0) + 2^1(c_1) * (j_2\, j_1\, j_0) + 2^0(c_0) * (j_2\, j_1\, j_0)$$

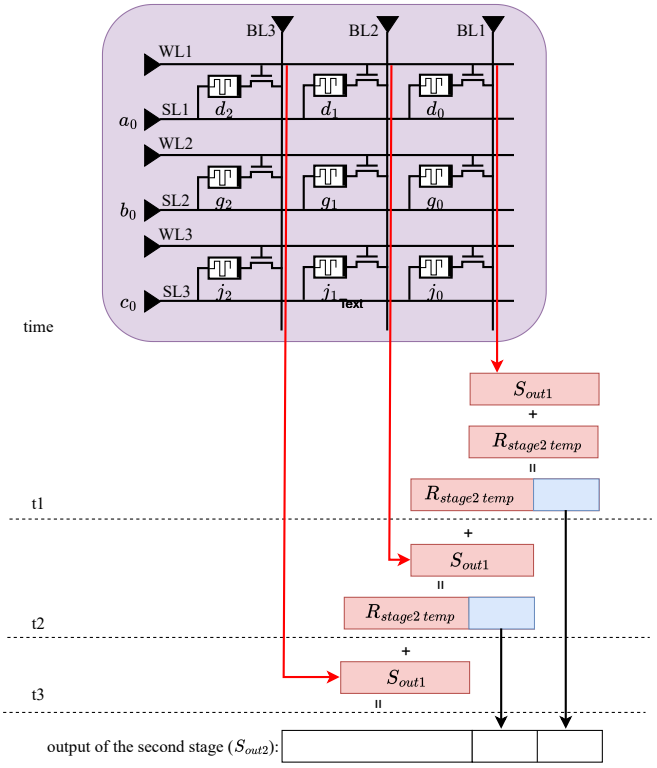


Figure 4.20: Example of the Multi-column sliding addition stage

4.11.3. Third Stage: Input sub-vector addition stage

As mentioned before, the DIM's finite resolution necessitates splitting the input vector into several sub-vectors. Therefore, each output from the second stage represents the result of multiplying one input sub-vector with the weight matrix. To obtain the final VMM outcome, the third stage performs a weighted accumulation of all the results generated in the second stage. Similar to the second stage, the third stage also employs the sliding method to carry out the weighted accumulation.

An example of this process is provided in Figure 4.21. The computation proceeds iteratively: the first sub-vector $\{a_0, b_0, c_0\}$ is input and processed through the first and second stages, yielding an initial partial result (corresponding to the yellow-highlighted box in the figure). This result is then fed into the third stage as the starting point for accumulation. Subsequently, the next sub-vector $\{a_1, b_1, c_1\}$ is input and processed, producing a new partial result (corresponding to the blue-highlighted box). The third stage then performs a weighted accumulation between this new result (blue) and the previous value (yellow), generating an updated accumulated value. Finally, the sub-vector $\{a_2, b_2, c_c\}$ is input and processed, yielding a new partial result (corresponding to the green-highlighted box). The third stage performs a final weighted accumulation between this new partial result (green) and the previously accumulated value, producing the latest accumulated result, which is the final result z_0. The detailed hardware operations proceed as follows:

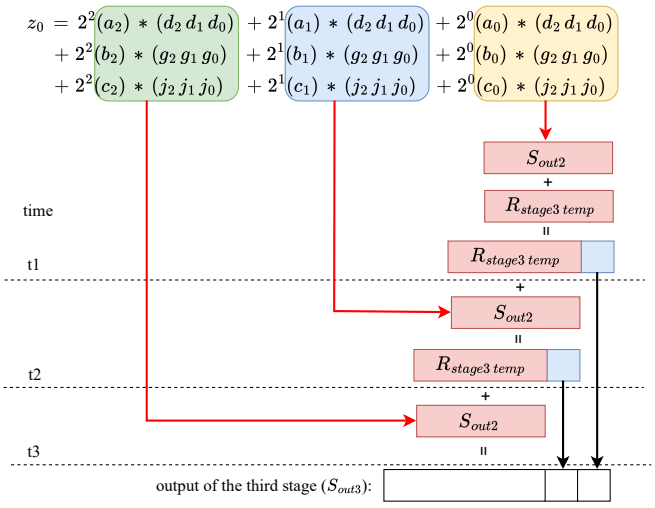


Figure 4.21: Example of the Input sub-vector addition stage

- Add the second stage output (S_{out2}) corresponding to the first sub-vector (yellow box) to the temporary register $R_{\text{stage3 temp}}$. The LSB of the sum is placed at the first bit position of the output, while the higher bits are stored in $R_{\text{stage3 temp}}$;
- Add the second stage output (S_{out2}) corresponding to the second sub-vector (blue box) to the temporary register $R_{\text{stage3 temp}}$. The LSB of the sum is placed at the second bit position of the output, while the higher bits update the $R_{\text{stage3 temp}}$;
- Add the second stage output (S_{out2}) corresponding to the third sub-vector (green box) to the temporary register $R_{\text{stage3 temp}}$. The complete sum is placed directly into the remaining output positions.

In the third stage, the `IADD` instruction accumulates the output from the second stage into the temporary register $R_{\text{stage3 temp}}$, with the LSB of the sum placed at the corresponding output position and the higher bits updating the register for the computation of the next input sub-vector. When processing the final input sub-vector, the complete sum is placed directly into the remaining output positions. The `CP` instruction clears this register to prepare for the new computation.

4.12. Summary

This chapter provides a comprehensive overview of the CIM tile's digital periphery implementation. The design employs a pipelined architecture to efficiently manage data flow and computation.

The chapter first introduces the input and output buffers, explaining the configuration of their critical parameters. It then describes the tile controller, which employs two decoders to translate instructions from separate memory streams and a stall detection unit to manage pipeline hazards.

The implementation of the Column Selection, Row Selection, and Row Data Selection modules is presented, detailing how the Column Selection and Row Selection modules perform their respective column and row addressing, and how the Row Data Selection module routes the data to the Source DIMs. The implementation of the Spare Row Selection is also described, explaining its mechanism for completing the remap operation.

The functions of DIMs, the SH module, and ADCs are also introduced. The role of the DIMs is to convert digital signals to the appropriate analog voltage levels required by the crossbar. The SH module captures analog outputs from the crossbar, enabling new operations to be performed during ADC conversion. We also introduce a shared ADC solution, where one ADC serves multiple columns to address area and power constraints.

The architecture of the Verification Unit and its operation in validating write operations are explained. Furthermore, a Logical Unit is presented to execute Boolean operations (AND, OR, XOR) based on the resistance states of two selected rows. A three-stage Addition Unit is also introduced to perform the VMM operation. The three stages comprise the analog accumulation stage to address the limited ADC resolution, the multi-column sliding addition stage to address the limited number of resistance levels, and the input vector segment addition stage to address the limited DAC resolutions.

Collectively, these components form the CIM tile architecture, enabling write, read, verification, logical operations, and VMM operation.

5

Area and Energy Models

This chapter presents the analytical models for area and energy tailored to our gate-level digital periphery circuits. The area model provides module-level parametric expressions that relate architectural parameters to cost. In contrast, the energy model is instruction-level: for each instruction, we enumerate the modules it activates and attribute energy accordingly, combining their switching, internal, and leakage energy.

5.1. Model Setup

This section specifies the technology/library corner for our digital-periphery models, enumerates the primitive cells available to the design, and defines the basic architectural parameters used throughout the analysis presented.

5.1.1. Standard-Cell Library and PVT Corner

All cells are instantiated from the Liberty standard-cell library `slow_vdd1v0`. The implementation and analysis are performed at the PVT (Process, Voltage, Temperature) corner labeled `PVT_0P9V_125C`, which corresponds to a supply voltage of 0.9 V and a temperature of 125°C. This corner represents a worst-case scenario for timing and power analysis. Under this scenario, Cadence Genus estimates cell area and energy by utilizing the library's timing and internal power tables in conjunction with the provided switching activity (from SAIF/VCD files), the leakage power estimation is obtained directly from the library data.

5.1.2. Cell Primitives

`INVX1` Inverter, $y = \neg a$.

`AND2X1` 2-input AND, $y = a \wedge b$.

`OR2X1` 2-input OR, $y = a \vee b$.

`XOR2X1` 2-input XOR, $y = a \oplus b$.

`XNOR2X1` 2-input XNOR (logical equivalence), $y = \overline{a \oplus b}$.

`AND3X1` 3-input AND, $y = a \wedge b \wedge c$.

`OR3X1` 3-input OR, $y = a \vee b \vee c$.

`NOR3X1` 3-input NOR, $y = \overline{a \vee b \vee c}$.

`AND4X1` 4-input AND, $y = a \wedge b \wedge c \wedge d$.

`OR4X1` 4-input OR, $y = a \vee b \vee c \vee d$.

`MX2X1` 2:1 multiplexer (select s): $y = (\neg s)\, d_0 \ \vee \ s\, d_1$.

`ADDFX1` 1-bit full adder:

$$S = a \oplus b \oplus \mathrm{CI}, \qquad \mathrm{CO} = (a \wedge b) \vee (a \wedge \mathrm{CI}) \vee (b \wedge \mathrm{CI}).$$

`DFFRX1` Rising-edge register with asynchronous active-low reset function:

$$Q^+ = \begin{cases} 0, & RN = 0, \\ D, & RN = 1 \text{ and the rising edge of } \textit{clk}, \\ Q, & \text{otherwise.} \end{cases}$$

`DFFSHQX1` Rising-edge register with asynchronous active-low set function:

$$Q^+ = \begin{cases} 1, & SN = 0, \\ D, & SN = 1 \text{ and the rising edge of } \textit{clk}, \\ Q, & \text{otherwise.} \end{cases}$$

`EDFFTRX1` Rising-edge register with enable and asynchronous active-low reset functions:

$$Q^+ = \begin{cases} 0, & RN = 0, \\ D, & RN = 1 \text{ and } EN = 1 \text{ and the rising edge of } \textit{clk}, \\ Q, & \text{otherwise.} \end{cases}$$

Notation: CI—carry-in *(only for ADDFX1)*; CO—carry-out *(only for ADDFX1)*; D—data input; Q/Q^+—current/next state; RN—asynchronous active-low reset; SN—asynchronous active-low set; EN—data-enable (load-enable) input *(only for EDFFTRX1)*.

5.1.3. Baseline Architectural Assumptions

In the crossbar array, each memristor cell stores one binary bit. The depths of the Write Data Buffer, the Logic Output Buffer, and the Addition Output Buffer are two. We use the following symbols to describe the architectural parameters used in the models.

$W_{\textbf{bus}}$ Width of the bus between the outside and the tile; units: bits. Range: $1 \le W_{\text{bus}} \le 64,\ W_{\text{bus}} \in \mathbb{Z}$ (assumed).

$W_{\textbf{op}}$ Width of the opcode field in the instruction; 4 bits.

$W_{\textbf{data}}$ Width of the Mask field for WDS/RDS instructions; units: bits. $1 \le W_{\text{data}} \le 32,\ W_{\text{data}} \in \mathbb{Z}$ (assumed).

$W_{\textbf{operand}}$ Bit-width of the operand; units: bits. We assume $W_{\text{operand}} > 1$ because, for $W_{\text{operand}} = 1$, the addition unit simplifies structurally: the second stage (multi-column sliding-addition stage) and the third stage (input-vector segment-addition stage) are not required (see Section 4.11). Accordingly, all area/energy formulas in this chapter are developed for $W_{\text{operand}} > 1$. Range: $2 \le W_{\text{operand}} \le 64,\ W_{\text{operand}} \in \mathbb{Z}$ (assumed).

$W_{\textbf{PC1}}$ Depth of the instruction memory 1;

$W_{\textbf{PC2}}$ Depth of the instruction memory 2;

$T_{\textbf{crossbar}}$ the number of clock cycles required to complete one crossbar array operation;

$T_{\textbf{SH}}$ the number of clock cycles for one execution of the SH module;

$T_{\textbf{ADC}}$ the number of clock cycles for one ADC conversion;

C Number of columns in the crossbar array; units: count. Range: $2 \le C \le 2048,\ C \in \mathbb{Z}$ (assumed).

R Number of rows in the crossbar array (spare rows excluded); units: count. Range: $2 \le R \le 2048,\ R \in \mathbb{Z}$ (assumed).

R_s Number of spare rows reserved for remapping; units: count. Range: $1 \le R_{\text{s}} \le 512,\ R_{\text{s}} \in \mathbb{Z}$ (assumed).

$N_{\textbf{ADC}}$ Number of ADC channels in the periphery; units: count. Range: $1 \le N_{\text{ADC}} \le 2048,\ N_{\text{ADC}} \in \mathbb{Z}$ (assumed).

5.2. Area Model

This section presents the area model for each module in our gate-level design. We partition the design into eleven modules, as presented in Section 4: Write Data Buffer, Row Data Buffer, Column Selection module, Row Selection module, Spare Row Selection module, Tile Controller, Write Verification Unit, Logic Unit, Addition Unit, Logic Output Buffer, and Addition Output Buffer. For each module, we provide a closed-form area expression obtained by summing standard-cell areas weighted by their instance counts. We denote area by A. For example, $A_{\text{Write Data Buffer}}$ is the area of the Write Data Buffer module, and $A_{\texttt{INVX1}}$ is the area of one `INVX1` cell.

Write Data Buffer.

$$\begin{aligned} A_{\text{Write Data Buffer}} = (2C+7)\,A_{\text{AND2X1}} + \tfrac{2C}{W_{\text{bus}}}\,A_{\text{AND3X1}} + C\,A_{\text{OR2X1}} + 2\,A_{\text{XNOR2X1}} \\ + A_{\text{XOR2X1}} + 4\,A_{\text{ADDFX1}} + (4+3C)\,A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.1}$$

Row Data Buffer.

$$\begin{aligned} A_{\text{Row Data Buffer}} = 3\,A_{\text{INVX1}} + \Big((R+2)W_{\text{operand}} + 7\Big)\,A_{\text{AND2X1}} + \tfrac{R\,W_{\text{operand}}}{W_{\text{bus}}}\,A_{\text{AND3X1}} \\ + (R + W_{\text{operand}} + 2)\,A_{\text{OR2X1}} + A_{\text{XNOR2X1}} + A_{\text{XOR2X1}} \\ + 2\,A_{\text{DFFSHQX1}} + \Big(R\,(W_{\text{operand}}+1) + 2W_{\text{operand}} + 2\Big)\,A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.2}$$

Column Selection.

$$\begin{aligned} A_{\text{Column Selection}} = (\lceil \log_2 \tfrac{C}{W_{\text{data}}} \rceil + 2)\,A_{\text{INVX1}} + \Big(3C + W_{\text{data}} + \tfrac{C}{W_{\text{data}}}(\lceil \log_2 \tfrac{C}{W_{\text{data}}} \rceil + 1) + 3\Big)\,A_{\text{AND2X1}} \\ + \Big(\tfrac{C}{W_{\text{data}}} + 1\Big)\,A_{\text{OR2X1}} + A_{\text{OR3X1}} + C\,A_{\text{OR4X1}} + (2C+1)\,A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.3}$$

Row Selection.

$$\begin{aligned} A_{\text{Row Selection}} = \lceil \log_2 \tfrac{R}{W_{\text{data}}} \rceil\,A_{\text{INVX1}} + \big(3R + W_{\text{data}} + \tfrac{R}{W_{\text{data}}}(\lceil \log_2 \tfrac{R}{W_{\text{data}}} \rceil + 1) + 1\big)\,A_{\text{AND2X1}} \\ + \big(2R + W_{\text{data}} + 1\big)\,A_{\text{OR2X1}} + \big(1 + \tfrac{R}{W_{\text{data}}}\big)\,A_{\text{OR3X1}} + (2R)\,A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.4}$$

Spare Row Selection.

$$\begin{aligned} A_{\text{Spare Row Selection}} = \big(R_s(1+3R) + 2\big)\,A_{\text{AND2X1}} + (3R_sR - 2R_s)\,A_{\text{OR2X1}} + (RR_s)\,A_{\text{XOR2X1}} \\ + \big(R_s(R+1) + 1\big)\,A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.5}$$

Read Register.

$$A_{\text{Read Register}} = (2C + N_{ADC} + 1)\,A_{\text{AND2X1}} + C\,A_{\text{AND3X1}} + (C+2)\,A_{\text{OR2X1}} + C\,A_{\text{EDFFTRX1}}. \tag{5.6}$$

Verification Unit.

$$A_{\text{Write Verification}} = (2C+2)\,A_{\text{AND2X1}} + C\,A_{\text{OR2X1}} + C\,A_{\text{XOR2X1}} + A_{\text{EDFFTRX1}}. \tag{5.7}$$

Logical Unit.

$$A_{\text{Logical Unit}} = (5C+1)\,A_{\text{AND2X1}} + C\,A_{\text{AND3X1}} + 2C\,A_{\text{OR2X1}} + (C+1)\,A_{\text{OR3X1}} + C\,A_{\text{EDFFTRX1}}. \tag{5.8}$$

Tile Controller.

$$\begin{aligned}A_{\text{Tile Controller}} = &\left(\lceil \log_2 \tfrac{C}{N_{\text{ADC}}} \rceil + W_{\text{PC2}} + 21\right) A_{\text{INVX1}} \\ &+ \left(4N_{\text{ADC}} + \lceil \log_2 \tfrac{C}{N_{\text{ADC}}} \rceil + 7W_{\text{PC2}} + \tfrac{C}{N_{\text{ADC}}}(\lceil \log_2 \tfrac{C}{N_{\text{ADC}}} \rceil - 1) + 36\right) A_{\text{AND2X1}} \\ &+ 3\, A_{\text{AND3X1}} + 28\, A_{\text{AND4X1}} + 16\, A_{\text{OR2X1}} + A_{\text{OR3X1}} + (W_{\text{PC2}} + 3)\, A_{\text{OR4X1}} \\ &+ A_{\text{NOR3X1}} + (W_{\text{PC1}} + 2W_{\text{PC2}})\, A_{\text{ADDFX1}} + W_{\text{PC1}}\, A_{\text{MX2X1}} \\ &+ (T_{\text{crossbar}} + T_{\text{SH}} + T_{\text{ADC}})\, A_{\text{DFFRX1}} \\ &+ \left(2W_{\text{PC1}} + 3W_{\text{PC2}} + \lceil \log_2 \tfrac{C}{N_{\text{ADC}}} \rceil + 20\right) A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.9}$$

Read/Logic Output Buffer.

$$\begin{aligned}A_{\text{Logic Output Buffer}} = &(2C + 9)\, A_{\text{AND2X1}} + 2C\, A_{\text{OR2X1}} \\ &+ 2\, A_{\text{XOR2X1}} + A_{\text{XNOR2X1}} + (3C + 4)\, A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.10}$$

Addition Output Buffer.

$$\begin{aligned}A_{\text{Addition Output Buffer}} = &\left(8 + 2\left\lfloor \tfrac{C}{W_{\text{operand}}} \right\rfloor \left(\lceil \log_2(R+1) \rceil + 2W_{\text{operand}}\right)\right) A_{\text{AND2X1}} \\ &+ \left(\left\lfloor \tfrac{C}{W_{\text{operand}}} \right\rfloor \left(\lceil \log_2(R+1) \rceil + 2W_{\text{operand}}\right)\right) A_{\text{OR2X1}} \\ &+ 2\, A_{\text{XOR2X1}} + A_{\text{XNOR2X1}} \\ &+ \left(4 + 3\left\lfloor \tfrac{C}{W_{\text{operand}}} \right\rfloor \left(\lceil \log_2(R+1) \rceil + 2W_{\text{operand}}\right)\right) A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.11}$$

Addition Unit.

$$\begin{aligned}A_{\text{Addition Unit}} = &\, 3\, A_{\text{INVX1}} \\ &+ \left(\left(C + \left\lfloor \tfrac{C}{W_{\text{operand}}} \right\rfloor + 3N_{\text{ADC}}\right) \lceil \log_2(R+1) \rceil + N_{\text{ADC}}(1 + W_{\text{operand}}) + C + \left\lfloor \tfrac{2C}{W_{\text{operand}}} \right\rfloor\right) A_{\text{AND2X1}} \\ &+ \left(C + N_{\text{ADC}} \lceil \log_2(R+1) \rceil\right) A_{\text{AND3X1}} \\ &+ \left((1 + \lceil \log_2(R+1) \rceil)\left(C + \left\lfloor \tfrac{C}{W_{\text{operand}}} \right\rfloor\right) + 2N_{\text{ADC}}\left(1 - \lceil \log_2(R+1) \rceil\right) - N_{\text{ADC}} W_{\text{operand}}\right) A_{\text{OR2X1}} \\ &+ \left(N_{\text{ADC}}\left(3\lceil \log_2(R+1) \rceil + W_{\text{operand}}\right)\right) A_{\text{ADDFX1}} \\ &+ \left(\left(C + 2N_{\text{ADC}} + \left\lfloor \tfrac{C}{W_{\text{operand}}} \right\rfloor\right) \log_2(R+1) + (2C + \left\lfloor \tfrac{C}{W_{\text{operand}}} \right\rfloor + N_{ADC} W_{\text{operand}})\right) A_{\text{EDFFTRX1}}. \end{aligned} \tag{5.12}$$

5.3. Energy Model

This section presents instruction-based energy models of our gate-level design in tabular form. Each table reports the numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies, illustrating how the per-instruction energy is computed. Owing to space limitations, we include only a representative subset of tables here as examples; the remaining tables and all required parameters are provided in the Appendix.

5.3.1. Column Selection Energy Model

The Column Selection module is active when executing the FS instruction, WDSs instruction, WDS instruction, and BNE instruction. Here are the parameters that are used in the energy model of the Column Selection module.

n_1: the number of '1' in the WDS enable registers; Range: $0 \leq n_1 \leq C,\ n_1 \in \mathbb{Z}$.

n_2: the number of changing bits in column enable outputs; Range: $0 \leq n_2 \leq C,\ n_2 \in \mathbb{Z}$.

n_3: the number of changing bits in WDS data inputs which are from the data field of WDS instruction; Range: $0 \leq n_3 \leq W_{\text{data}},\ n_3 \in \mathbb{Z}$.

n_4: the number of '1' in verification sign inputs. A value of 1 indicates a write–verify mismatch between the programmed value and the target value; Range: $0 \leq n_4 \leq C,\ n_4 \in \mathbb{Z}$.

C_{DIM}: the input capacitance of the DIMs; We assume 50fF.

For the FS instruction (store function), Table 5.1 lists the number of active gates and registers, the associated load capacitances, and the corresponding internal energies.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR2X1	2	$0.4 + 0.4C$	1	0.592
2	INVX1	2	0.2	1	0.232
3	EDFFTRX1	1	0.2	n_1	/
4	OR3X1	2	$0.2C/W_{\text{data}}$	1	0.626
5	OR2X1	2	$0.4W_{\text{data}}$	C/W_{data}	0.594
6	EDFFTRX1	1	$0.4 + C_{DIM}$	n_2	/
register internal energy					
7	EDFFTRX1	2	only EN changes	$2C$	1.56
8	EDFFTRX1	2	only Q changes	$n_1 + n_2$	1.92

Table 5.1: Column Selection module's Energy model Table for FS (store).

Using Table 5.1, we illustrate how to compute the switching energy and internal energy with this table.

$$E_{\text{switch}}^{\text{FS(store)}} = \tfrac{1}{2}\, V_{DD}^2 \sum_{\text{entries}} \big(\text{activity} \times \text{capacitance} \times \text{number}\big), \quad \text{with } V_{DD} = 0.9 \text{ V}$$

$$= \tfrac{1}{2}\,(0.9)^2 \Big[\underbrace{2\,(0.4 + 0.4C) \cdot 1}_{\text{entry \#1}} + \underbrace{2\,(0.2) \cdot 1}_{\text{entry \#2}} + \underbrace{1\,(0.2) \cdot n_1}_{\text{entry \#3}} + \underbrace{2\left(\frac{0.2\,C}{W_{\text{data}}}\right) \cdot 1}_{\text{entry \#4}}$$

$$+ \underbrace{2\,(0.4\,W_{\text{data}}) \cdot \left(\frac{C}{W_{\text{data}}}\right)}_{\text{entry \#5}} + \underbrace{1\,(0.4 + C_0) \cdot n_2}_{\text{entry \#6}} \Big] \text{ fJ}$$

$$E_{\text{internal}}^{\text{FS(store)}} = \sum_{\text{entries}} \big(\text{activity} \times \text{internal energy} \times \text{number}\big)$$

$$= \Big[\underbrace{2 \cdot 0.592 \cdot 1}_{\text{entry \#1}} + \underbrace{2 \cdot 0.232 \cdot 1}_{\text{entry \#2}} + \underbrace{2 \cdot 0.626 \cdot 1}_{\text{entry \#4}}$$

$$+ \underbrace{2 \cdot 0.594 \cdot \left(\frac{C}{W_{\text{data}}}\right)}_{\text{entry \#5}} + \underbrace{2 \cdot 1.56 \cdot 2C}_{\text{entry \#7}}$$

$$+ \underbrace{2 \cdot 1.92 \cdot (n_1 + n_2)}_{\text{entry \#8}} \Big] \text{ fJ}$$

In addition to the data-dependent terms above, registers dissipate internal energy on every clock transition, even if the instruction is not executed (no clock gating). Let N_{reg} be the number of clocked registers considered and N_{cyc} the number of clock cycles. With per-edge internal energy $E_{\text{clk}} = 0.705$ fJ, the clock-driven term is

$$E_{\text{internal,clk}} = N_{\text{reg}} \cdot E_{\text{clk}} \cdot 2 \cdot N_{\text{cyc}},$$

where the factor 2 accounts for rising and falling edges per cycle. The total internal energy is then

$$E_{\text{internal,total}} = E_{\text{internal}}^{\text{FS(store)}} + N_{\text{reg}} \cdot 0.705 \cdot 2 \cdot N_{\text{cyc}} \text{ fJ}.$$

For the leakage energy, it is calculated as the sum of leakage energy from all gates in the module. For each gate, its leakage energy equals the gate leakage power multiplied by the execution time T:

$$E_{\text{leakage}} = \sum_{\text{all gates}} (P_{\text{leakage,gate}} \times T),$$

where $P_{\text{leakage,gate}}$ is the leakage power of an individual gate obtained from the standard cell library, and T is the total execution time of the instruction.

For the WDSs and WDS instructions, Tables A.2 and A.3 report the numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies. For BNE when the verification fails, the same quantities are listed in Table A.4.

5.3.2. Row Selection & Spare Row Selection Energy Model

The Row Selection module and Spare Row Selection module are active when executing the `FS`, `RDSs`, `RDS`, and `BNE` instructions. Table A.5 presents the energy model during the execution of the `FS` instruction. The model is structured into a common Base Case shared across all scenarios and four distinct scenarios, each with specific additions: Scenario 1 applies when the previous operation is store/read/-logic and the current operation selects store/logic/read/verification; Scenario 2 applies when the previous operation is VMM and the current operation selects store/logic/read; Scenario 3 applies when the previous operation is store/read/logic and the current operation selects VMM; and Scenario 4 applies when the previous operation is VMM and the current operation also selects VMM. Table A.6 presents the energy model during the execution of the `RDS` instruction. The model is structured into a common Base Case shared across all scenarios and two distinct scenarios, each with specific additions: Scenario 1 applies when the current operation is store/read/logic; and Scenario 2 applies when the current operation is VMM. For `RDSs` and `RDsh`, the corresponding results are summarized in Tables A.7 and A.8. For `BNE` in the rewrite operation, the results are summarized in Table A.9. Each table reports the numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies.

5.3.3. Read Register Energy Model

The Read Register is active when executing the `DoS` and `CSR` instructions. For `DoS` and `CSR`, the numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies are summarized in Tables A.10, and A.11, respectively.

5.3.4. Verification Unit Energy Model

The Verification Unit is active when executing the `FS` instruction (store operation), `DoS`, `CSR`, and `BNE`. For `FS` (store), `DoS`, and `CSR`, the numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies are summarized in Tables A.12, A.13, and A.14, respectively. For BNE when verification fails, the corresponding quantities are reported in Table A.15.

5.3.5. Logical Unit Energy Model

The Logical Unit is active when executing the `DoS` or `CSR` instructions. For the `DoS` instruction, the numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies are summarized in Tables A.16. For the `CSR` instruction, Table A.17 presents its energy model. The model is structured into a common Base Case shared across all scenarios and two distinct scenarios, each with specific additions: Scenario 1 applies when the current operation is AND/XOR operation; and Scenario 2 applies when the current operation is OR operation.

5.3.6. Tile Controller Energy Model

The Tile Controller energy model is divided into three sub models. Decoder1 energy model, Decoder2 energy model, and SatIl detection energy model.

Decoder1 Energy Model

The Decoder1 is active when executing the `FS`, `WDsh`, `WDS`, `WDSs`, `RDsh`, `RDS`, `RDSs`, `DoA`, and `BNE` instructions. The numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies are summarized per instruction in Tables A.18, A.19, A.20, A.21, A.22,

A.23, and A.24. For DoA in Table A.25, non-store operations (verification, logical, read, VMM) use the base case, while store operations require the base case plus Scenario 1 additions. For BNE when verification fails, the corresponding quantities are reported in Table A.26.

Decoder2 Energy Model

The Decoder2 is active when executing the DoS, CSR, LS, IADD, BNE, JAL, JR, and CP instructions. For DoS in Table A.27, non-verification operations (logical, read, VMM) use the base case, while verification operations require the base case plus Scenario 1 additions. For CSR, LS, IADD, JAL, JR, and CP, the numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies are summarized in Tables A.28, A.29, A.30, A.32, A.33, and A.34, respectively. For BNE, two scenarios are defined based on verification results and spare-row availability, both extending the base case with their respective components: (i) Scenario 1 for cases where verification passes or no spare rows are available, and (ii) Scenario 2 for cases where verification fails and repair actions are executed. The complete energy model is detailed in Table A.31.

Stall Detection Energy Model

The Stall Detection unit is active when executing the DoA, DoS, CSR, and BNE instructions. For DoA, we distinguish three scenarios based on operation type: store operations, verification operations, and other non-store operations (logical, read, and VMM). All scenarios build upon the base case and require their respective additional components, with the complete energy model detailed in Table A.35. For DoS, CSR, and BNE, the numbers of active gates and registers, the associated load capacitances, and the corresponding internal energies are summarized in Tables A.36, A.37, and A.38, respectively.

5.3.7. Addition Unit Energy Model

The Addition Unit is active when executing the CSR, LS, and IADD instructions. For CSR, two scenarios are considered depending on whether the current segment is the last segment of the column: non-last segment scenario (comprising the Base Case and Scenario 1 components) and last segment scenario (comprising the Base Case and Scenario 2 components). The complete energy model with base components and scenario-specific additions is detailed in Table A.39. For IADD, the complete energy model is detailed in Table A.40.

5.3.8. Write Data Buffer Energy Model

The Write Data Buffer is active for external data writes and WDsh instruction execution. As the bus width may be narrower than the buffer width, filling a buffer chunk may require multiple writes. Consequently, two scenarios are defined as shown in Table A.42: the base case, which applies when the write pointer remains unchanged, and the scenario for a changing write pointer, which corresponds to the base case supplemented with the components of Scenario 1. The complete energy model about the WDsh instruction is shown in Table A.43.

5.3.9. Row Data Buffer Energy Model

The Row Data Buffer is active for external data writes and RDsh instruction execution. As the bus width may be narrower than the buffer width, filling a buffer chunk may require multiple writes. For writing external data into the buffer, four operational scenarios are defined based on the value of the write pointer, which ranges from 0 to $W_{operand} - 1$. The base case applies when the write pointer remains unchanged. Scenario 1 happens when the write pointer value ranges from 1 to $W_{operand} - 2$. Scenario 2 and Scenario 3 are triggered when the write pointer equals 0 and $W_{operand} - 1$, respectively. All three scenarios (1, 2, and 3) incorporate the components of the base case in addition to their own unique components. The complete energy model for these scenarios is detailed in Table A.44. For the RDsh instruction, three scenarios are defined by the read pointer value, which also ranges from 0 to $W_{operand} - 1$: the base case applies when the pointer ranges from 1 to $W_{operand} - 2$; Scenario 1 is defined when it equals 0; and Scenario 2 when it equals $W_{operand} - 1$. The corresponding energy model is provided in Table A.45.

5.3.10. Read/Logic Output Buffer Energy Model

The Read/Logic Output Buffer is active during CP instruction execution or external read operations. The energy model for external read operations is shown in Table A.47, while the model for the CP instruction

is detailed in Table A.46.

5.3.11. Addition Output Buffer Energy Model

The Addition Output Buffer is active during `CP` instruction execution or external read operations. The energy model for external read operations is shown in Table A.49, while the model for the `CP` instruction is detailed in Table A.48.

5.4. Summary

This chapter presents analytical models for estimating the area and energy consumption of our gate-level digital peripheral circuitry. The area model provides parametric expressions at the module level, directly linking architectural parameters to hardware costs such as gate counts and register counts. For energy modeling, we adopt an instruction-aware approach: for each instruction type, we identify the specific set of activated modules and compute energy consumption by accumulating their respective contributions from capacitive switching, internal cell activities, and leakage over the instructions. The model is presented in tabular format, detailing the active gates and registers along with their corresponding load capacitances for energy estimation.

6

Experimental Results

This chapter presents the experimental results and analysis for the digital modules of the proposed CIM tile architecture. The remapping functionality validation is conducted at the RTL level. The evaluation of area and energy is based on post-synthesis gate-level simulations performed with Cadence Genus, using our custom gate-level implementation. All experiments are performed under the following baseline configuration: 32×32 crossbar size, external bus width W_{bus} = 8 bits, N_{ADC} = 1, WDS/RDS instruction data field width W_{data} = 4 bits, operand width W_{operand} = 2 bits, number of spare rows R_{s} = 2, W_{PC1} = 10, W_{PC2} = 10, T_{crossbar} = 1, T_{SH} = 1, and T_{ADC} = 1. We first validate the remapping functionality, then analyze the area overhead of the architecture, followed by an examination of its energy consumption for different operations.

6.1. Simulation Results of the Verification Functionality

We validated the verification functionality through RTL simulation in Cadence. To streamline verification and focus on the digital control logic, we employed register-based models to emulate key analog components. The memristor crossbar array was replaced with a register file, where write and read operations were controlled by enable signals from CIM tile operations including write, verification, read, logic, and VMM. Similarly, the SH module and ADCs were modeled using registers that update their values upon receiving “done” pulses from the tile controller.

In this experiment, a persistent cell fault was emulated by programming a data pattern of all ’1’s into the first row while hardwiring its first register to ’0’, deliberately creating a non-writable condition.

As shown in Figure 6.1(a), a store operation executes from 60 ns to 110 ns. At 105 ns, the signal `crossbar_cell[0][31:0]` transitions to ’fffffffe’, indicating that bit [0] was not successfully written to ’1’. A verification phase from 110 ns to 1130 ns follows, with the `vrf_fail` signal asserting at 1125 ns, confirming the write failure. The `vrf_sign[0]` signal being ’1’ precisely locates the error to the cell at position [0]. A rewrite operation is then triggered from 1130 ns to 1150 ns, during which the `remap_flag` is set. A second verification from 1150 ns to 2170 ns confirms the operation still fails at 2160 ns, with `vrf_fail` asserted and `vrf_sign[0]` still high shown in Figure 6.1(b), confirming a persistent fault.

Figure 6.1(b) shows this persistent failure activates the remapping sequence from 2170 ns to 2190 ns. At 2175 ns, the `row_disable[0]` signal is set, logically disabling the faulty first row, while the `row_masked[0][31:0]` signal becomes ’00000001’, indicating that a spare row has been assigned to replace it. By 2185 ns, the `crossbar_cell[32][31:0]` signal is successfully written to ’ffffffff’, confirming the data has been stored in the spare row. A final verification from 2190 ns to 3210 ns shows `vrf_fail` remains low, successfully validating the entire remapping process and system recovery.

6.2. Area Result Analysis

We performed an area analysis based on our gate-level post-synthesis design. The following set of figures quantifies how the area overhead scales with independent variations in: (a) crossbar size, (b)

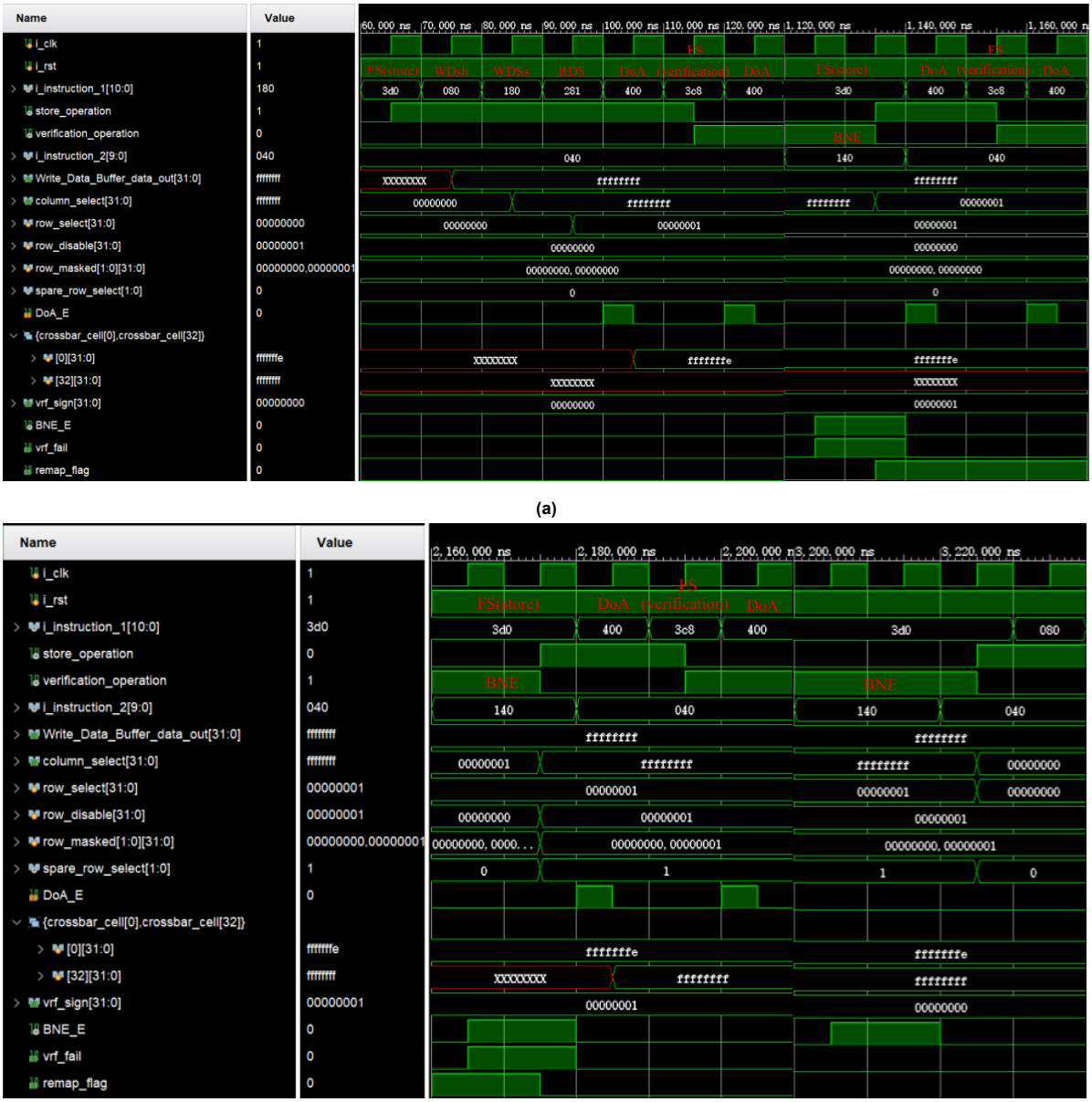


Figure 6.1: Waveforms of the remapping. (a) Initial fault handling: store, verification, and rewrite operations; (b) Row remapping and verification.

number of ADCs, (c) operand width, (d) number of spare rows, (e) the the external bus width, and (f) the WDS/RDS instruction data field width.

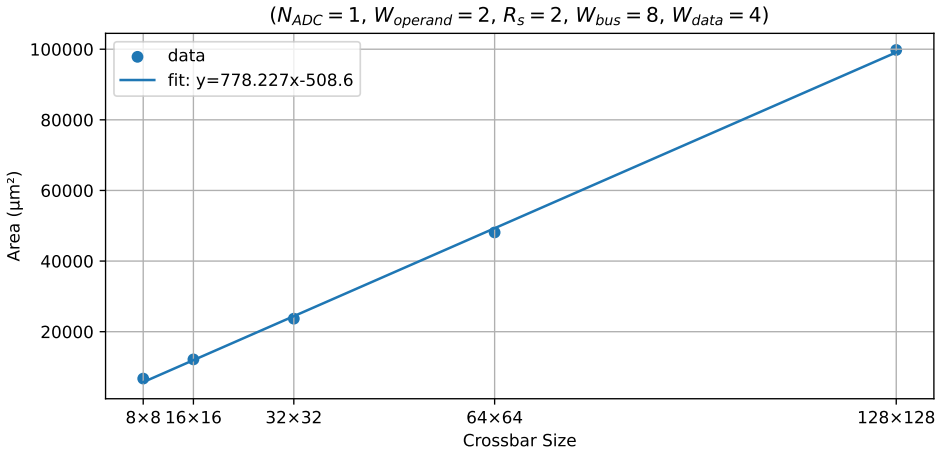


Figure 6.2: Area Overhead of Digital CIM-Tile Components vs. Crossbar size

Figure 6.2 illustrates the area overhead across crossbar sizes from 8×8 to 128×128. The overall area exhibits approximately linear growth, which aligns with the theoretical analysis. The area of the Write Data Buffer, Column Selection, Logical Unit, and Verification Unit all scale linearly with the number of columns. Besides, the area of the Row Data Buffer, Row Selection module, Row Data Selection module, and Spare Row Selection scales linearly with the number of rows. For the addition unit, increasing the number of rows raises the per-column output bit-width. As a result, more registers are needed to store each column's output, and the adder input width increases, requiring larger adders. Increasing the number of columns linearly increases the number of per-column registers. Moreover, the width of the Logic Output Buffer and the Addition Output Buffer increases linearly with the number of columns.

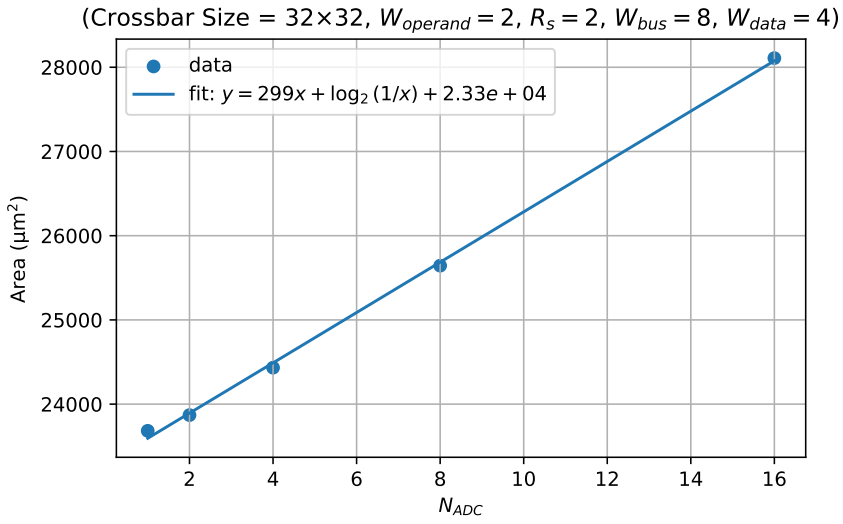


Figure 6.3: Area Overhead of Digital CIM-Tile Components vs. number of ADCs

Figure 6.3 illustrates the area overhead as the number of ADCs N_{ADC} varies from 1 to 16. The number of ADCs mainly affects the Tile Controller and the Addition Units. In our configurations, the total area increases monotonically with the number of ADCs.

For the Tile Controller, with a fixed number of columns (C), increasing the number of ADCs (N_{ADC}) reduces the number of columns per ADC group (C/N_{ADC}). This reduction decreases the bit width required for the index in CSR instructions, since the index is used to select the column within each ADC group. Consequently, the area of the decoder responsible for decoding the indec is reduced. In

addition to the index decoder, the Tile Controller also includes logic for ADC selection. As the number of ADCs increases, the area of this ADC selection logic grows accordingly.

The impact on the three-stage Addition Unit is more complex. In the Analog Accumulation Stage, the total number of registers and adder bit-width is fixed by the crossbar size. However, as N_{ADC} increases, the number of Addition Unit grows, but each unit's multiplexer handles fewer columns. The aggregate multiplexer area decreases because replacing one wide multiplexer with several narrower ones results in a lower total gate count. In the Multi-column Sliding Addition Stage, the register count and adder bit-width per unit are fixed by the number of rows and the operand width; thus, increasing the number of ADCs leads to an approximately linear area increase in this stage. In the Input Vector Segment Addition Stage, the register count and adder bit-width scale with the number of rows, number of columns, and operand bit-width. As N_{ADC} increases, the number of adder in this stage grows proportionally, while the total multiplexer area in this stage decreases. This reduction occurs because each Addition Unit's multiplexer is responsible for fewer operands from the crossbar array.

In summary, increasing the number of ADCs reduces the area of the index decoder for `CSR` instruction and the multiplexer area in the Addition Units. However, these savings are outweighed by the increased area from the ADC selection logic and, most significantly, the linear growth in the number of Addition Units, resulting in an increase in total area.

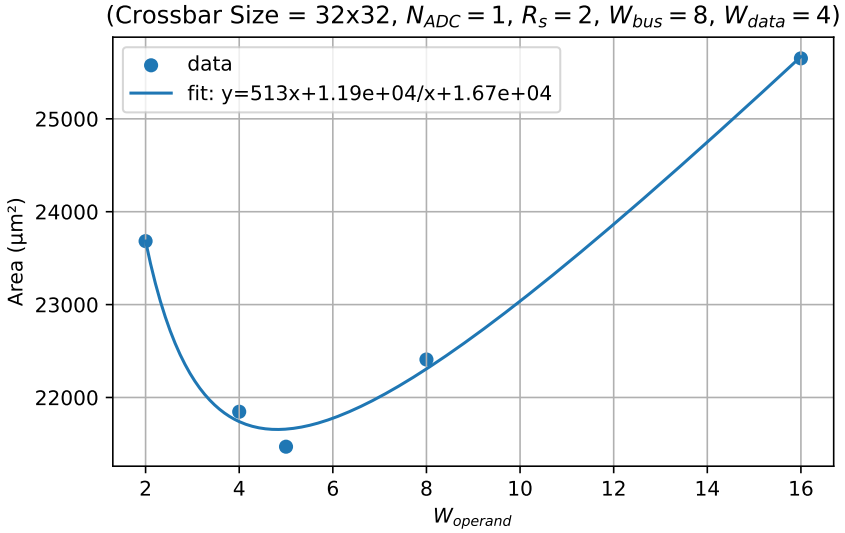


Figure 6.4: Area Overhead of Digital CIM-Tile Components vs. number of Datatype Size

Figure 6.4 reports the area overhead as the operand width W_{operand} varies across specific values (2, 4, 5, 8, and 16). The operand width significantly influences the area characteristics of three key components (the Row Data Buffer, the Addition Output Buffer, and the Addition Units), leading to a non-monotonic trend in total area that first decreases and then increases as the operand width grows.

In the multi-column sliding-addition stage of the Addition Unit, increasing the operand width enlarges both the register count and the adder bit-width. In the input-vector segment addition stage, each Addition Unit covers a fixed number of columns, so the number of operands reduces to $\lfloor C/(N_{\mathrm{ADC}}W_{\mathrm{operand}})\rfloor$ as W_{operand} grows; accordingly, the multiplexer and its selection control logic scale down with W_{operand}, and the total register width $\lfloor C/(N_{\mathrm{ADC}}W_{\mathrm{operand}})\rfloor\left(\log_2(R{+}1){+}2W_{\mathrm{operand}}\right)$ also decreases, whereas the adder width increases with W_{operand}. For the Addition Output Buffer, the width $\lfloor C/W_{\mathrm{operand}}\rfloor\left(\log_2(R{+}1){+}2W_{\mathrm{operand}}\right)$ decreases with the operand width, reducing its area, while the Row Data Buffer area grows approximately linearly because its depth equals the operand width.

Notably, at $W_{\mathrm{operand}} = 5$, the measured area falls below the fitted curve. This deviation arises because the number of operands in the input-vector segment addition stage, given by $\lfloor C/(N_{\mathrm{ADC}}W_{\mathrm{operand}})\rfloor$, must be an integer. With $C = 32$ and $N_{\mathrm{ADC}} = 1$, the floor operation $\lfloor 32/5\rfloor = 6$ results in fewer operands than the theoretical value of 6.4, leading to a greater reduction in hardware resources than predicted by the continuous model. This integer constraint explains the observed area drop at this

specific operand width.

Overall, as the operand width increases, the total area first decreases (dominated by reductions in input-vector segment addition stage multiplexer/selection control logic and the number of registers, together with a narrower Addition Output Buffer) and then increases (dominated by wider adders in the input-vector segment addition stage and multi-column sliding-addition stage, more registers in the multi-column sliding-addition stage, and the Row Data Buffer scaling approximately linearly with the operand width).

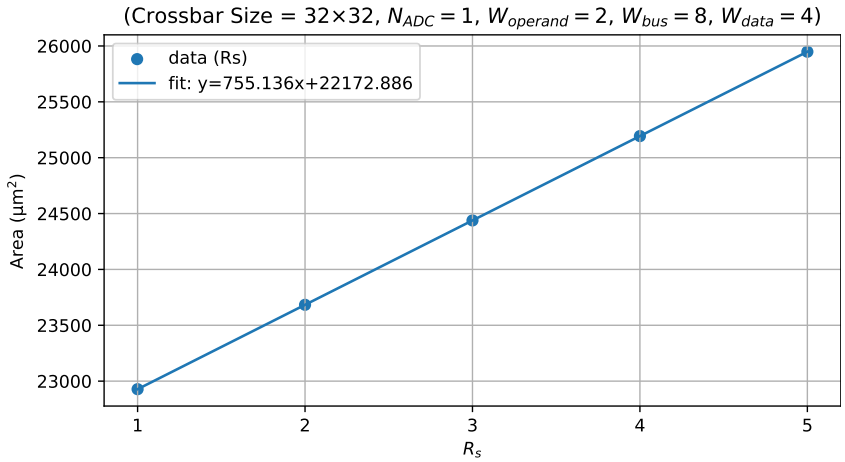


Figure 6.5: Area Overhead of Digital CIM-Tile Components vs. number of spare rows

Figure 6.5 illustrates the area overhead across different numbers of spare rows from 1 to 5. The overall area exhibits approximately linear growth, which aligns with the theoretical analysis. The area of the Spare Row Selection module scales linearly with the number of spare rows, as each additional spare row requires a corresponding set of registers and combinational logic gates for selection control and data management. Therefore, the total area increases linearly with the number of spare rows.

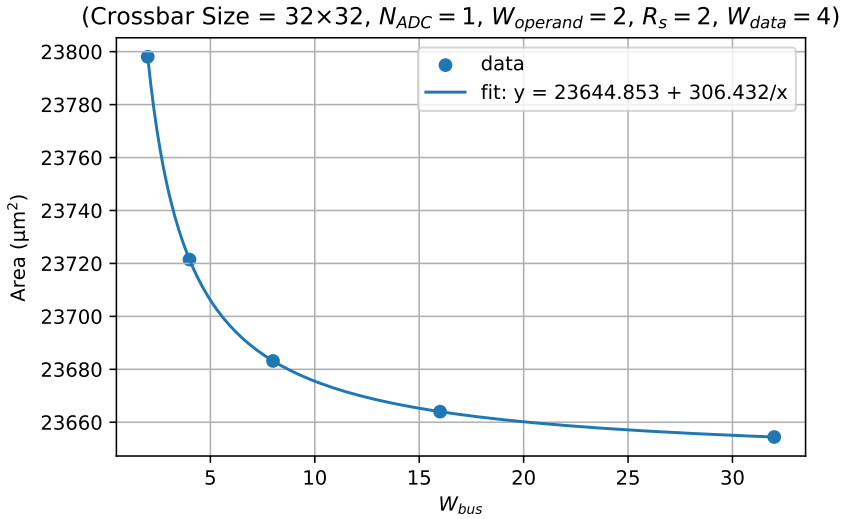


Figure 6.6: Area Overhead of Digital CIM-Tile Components vs. the external bus width

Figure 6.6 illustrates the area overhead across different bus widths W_{bus} from 2 to 32. The overall area decreases as the bus width increases, which aligns with the theoretical analysis. This reduction occurs because a wider bus reduces the number of segments within a buffer chunk that need to be individually selected for writing. The selection logic, which directs the incoming data to the appropriate segment, is thus simplified as the bus width grows, leading to a smaller area. Consequently, the total area decreases with increasing bus width.

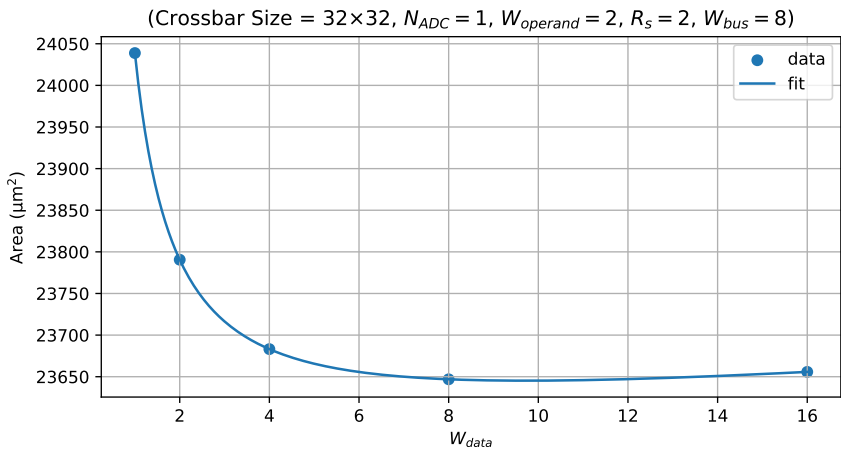


Figure 6.7: Area Overhead of Digital CIM-Tile Components vs. the WDS/RDS instruction data field width

Figure 6.7 illustrates the area overhead across different WDS/RDS data field widths W_{data} from 1 to 16. The overall area exhibits a non-monotonic trend, first decreasing and then increasing as W_{data} grows. This behavior aligns with the theoretical analysis and is attributed to competing factors. The initial reduction is due to two main factors: First, the index width in the `WDS`/`RDS` instructions decreases with larger W_{data}, leading to a smaller decoder area in the Column Selection module and Row Selection module. Second, the number of blocks in both the Column Selection module (C/W_{data}) and the Row Selection module (R/W_{data}) is reduced, thereby simplifying their respective selection logic. However, the WDS/RDS data field processing logic within both the Column Selection module and Row Selection module, which is used for distributing instruction data fields to corresponding modules based on the instructions, exhibits a linear area increase with larger W_{data}. At smaller W_{data} values, the area reduction from simplified decoders and selection logic dominates, resulting in an overall area decrease. As W_{data} continues to increase, the linear growth in data field processing logic becomes the dominant factor, leading to the eventual area increase observed at larger W_{data} values.

6.3. Energy Result Analysis

An energy analysis is performed on our gate-level design using Cadence Genus, examining five distinct operations: store operation, verification operation, read operation, logic operation, and VMM operation. Each operation type is evaluated under its specific parameter variations to characterize energy consumption scaling, while maintaining other parameters at the baseline configuration. The detailed parameter configurations and corresponding results for each operation are presented in the following subsections.

6.3.1. Store Operation Energy Result Analysis

This subsection presents the energy analysis for store operation. The experiment was conducted by writing one row of data into the crossbar array, which involves two sequential phases: (1) loading external data into the Write Data Buffer, and (2) transferring the data from the buffer to the crossbar array. Figure 6.8 shows an example of the instruction sequence used in this experiment.

number	Opcode	Field 1	Field 2
1.	FS	store	
2.	WDSh		
3.	WDSs		
4.	RDS	000	0001
5.	DoA		

Figure 6.8: An example of the instruction sequence for store operation

The energy consumption of store operations is primarily influenced by two parameters: the external bus width W_{bus} and the crossbar size. The following analysis characterizes how energy scales with variations in these two parameters, examining their individual impacts on the overall energy consumption.

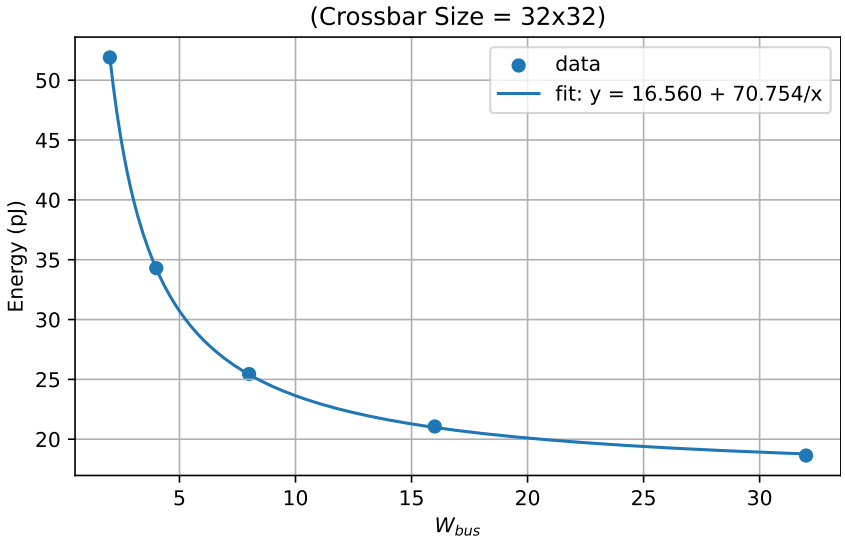


Figure 6.9: Store Operation Energy Overhead of Digital CIM-Tile Components vs. External Bus

Figure 6.9 reports the energy required to execute the same store operation with varying the external bus widths (W_{bus} = 2, 4, 8, 16, and 32). The total energy decreases inversely with the external bus width W_{bus}, primarily because a wider bus reduces the number of write cycles required to load one row of data into the buffer, thereby shortening the execution time. This reduction in clock cycles directly lowers the register internal energy caused by clock toggling, resulting in an overall inverse proportionality between energy and W_{bus}. Additionally, the shortened execution time leads to a proportional reduction in leakage energy, as the total number of components in the digital periphery circuit remains unchanged. Consequently, the total energy exhibits an inverse proportionality to the external bus width.

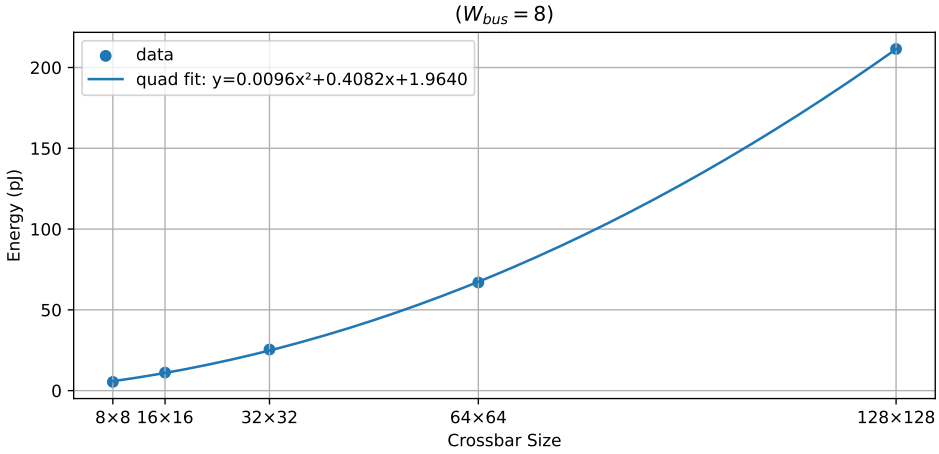


Figure 6.10: Store Operation Energy Overhead of Digital CIM-Tile Components vs. Crossbar Size

Figure 6.10 illustrates the store operation's energy overhead across crossbar sizes from 8×8 to 128×128. The total energy exhibits a quadratic relationship with crossbar size, primarily due to two compounding factors. First, as the crossbar size increases, the number of data elements to be written per row grows linearly. With a fixed $W_{\text{bus}} = 8$, this requires a linearly increasing number of cycles to write data into the Write Data Buffer. Second, the number of registers in the digital periphery increases linearly with crossbar size, leading to a quadratic increase in register internal energy caused by clock

toggling over the extended execution time. Additionally, the number of registers and combinational logic gates involved in the store operation within the Write Data Buffer and Column Selection module also increases linearly with crossbar size, contributing to a linear rise in switching energy. Furthermore, leakage energy exhibits a quadratic increase, as the enlarged crossbar size necessitates both a greater number of circuit components and a longer active duration for the store operation. The combined effect of these factors results in an overall quadratic dependence of total energy on crossbar size.

6.3.2. Verification Operation Energy Result Analysis

This subsection presents the energy analysis for the verification operation. The experiment was conducted by reading the crossbar output and comparing it with the expected data from the Write Data Buffer. Figure 6.11 shows an example of the instruction sequence used in this experiment.

number	Opcode	Field 1	Field 2
1.	FS	verification	
2.	DoA		
3.	DoS		
4.	CSR	00000	1
5.	CSR	00001	1
6-34.	CSR	00010-11110	1
35.	CSR	11111	1
36.	BNE		

Figure 6.11: An example of the instruction sequence for verification operation

The energy consumption of the verification operation is primarily influenced by two parameters: i) the crossbar size; and ii) the number of ADCs N_{ADC}. The following analysis characterizes how energy scales with variations in these two parameters, examining their individual impacts on the overall energy consumption.

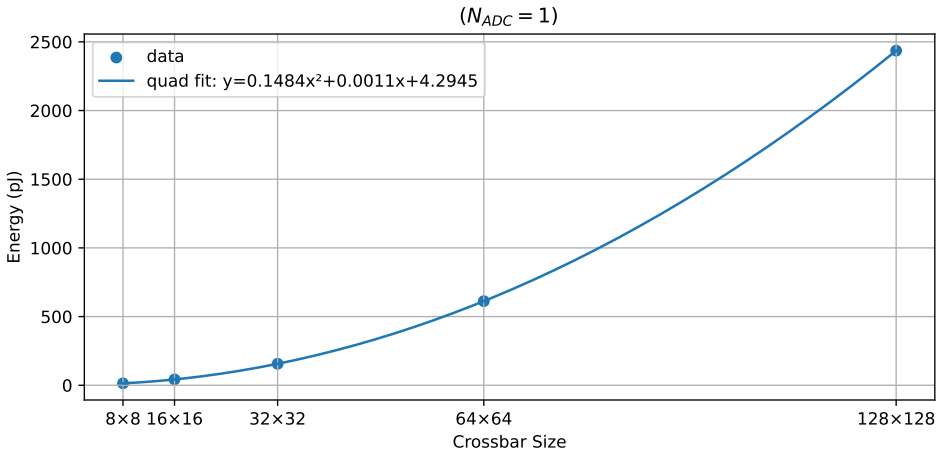


Figure 6.12: Verification Operation Energy Overhead of Digital CIM-Tile Components vs. Crossbar Size

Figure 6.12 illustrates the verification operation's energy overhead across crossbar sizes from 8×8 to 128×128. The total energy of verification operations exhibits a quadratic relationship with crossbar size. This scaling arises from two compounding effects. First, the number of registers and combinational logic gates in the digital periphery increases linearly with crossbar size. Concurrently, with a fixed number of ADCs $N_{\mathrm{ADC}} = 1$, reading the increased number of columns requires a linearly growing number of `CSR` instructions, which extends the execution time. Consequently, the register internal energy caused by clock toggling increases quadratically, resulting from the product of the increased register count and the prolonged execution time. Second, the number of registers and combinational gates actively involved in the verification operation within the verification unit and read registers also

grows linearly with crossbar size. This increase in active components contributes to a linear rise in switching energy. Furthermore, leakage energy increases quadratically due to the combined effect of more circuit components operating over a longer duration. Together, these factors result in an overall quadratic dependence of total energy on crossbar size.

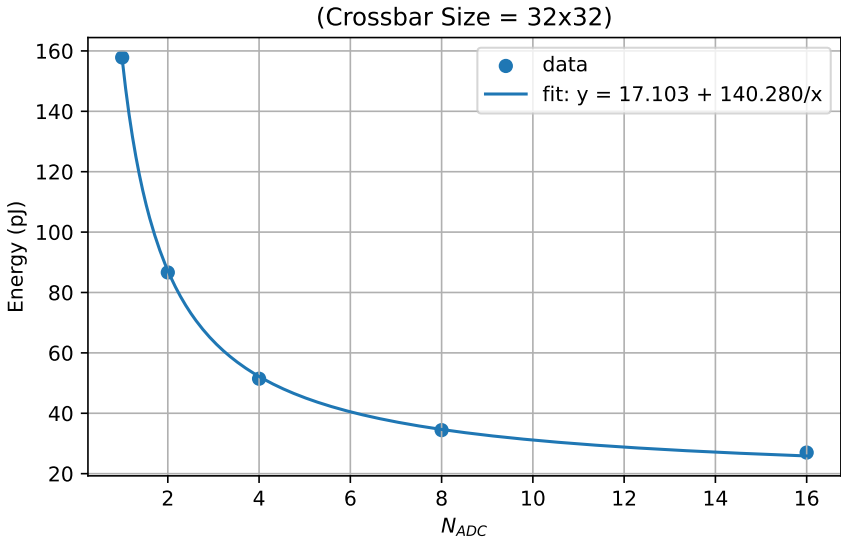


Figure 6.13: Verification Operation Energy Overhead of Digital CIM-Tile Components vs. number of ADCs

Figure 6.13 reports the energy required to execute the verification operation with varying the number of ADCs (N_{ADC} = 1, 2, 4, 8, and 16). The total energy of verification operations decreases with the number of ADCs. This is primarily because a higher ADC count allows each `CSR` instruction to read more columns in parallel, which reduces the total number of `CSR` instructions required and shortens the execution time. The resulting reduction in clock cycles leads to a significant decrease in the register internal energy dominated by clock toggling and the total leakage energy. Although the increased number of ADCs introduces additional registers and combinational logic in the addition units, which slightly raises the internal energy from the extra registers and the leakage energy from the enlarged circuit scale, these increases are negligible compared to the substantial energy savings achieved from the shortened execution time. Consequently, the net effect is a clear decrease in total energy consumption as the number of ADCs grows.

6.3.3. Read Operation Energy Result Analysis

This subsection presents the energy analysis for the read operation. The experiment was conducted by reading the crossbar output directly from the array. Figure 6.14 shows an example of the instruction sequence used in this experiment.

number	Opcode	Field 1	Field 2
1.	FS	read	
2.	RDS	000	0001
3.	DoA		
4.	DoS		
5.	CSR	00000	1
6.	CSR	00001	1
7-35.	CSR	00010-11110	1
36.	CSR	11111	1
37.	CP		

Figure 6.14: An example of the instruction sequence for read operation

The energy consumption of read operation is primarily influenced by two parameters: the crossbar size

and the number of ADCs N_{ADC}. The following analysis characterizes how energy scales with variations in these two parameters, examining their individual impacts on the overall energy consumption.

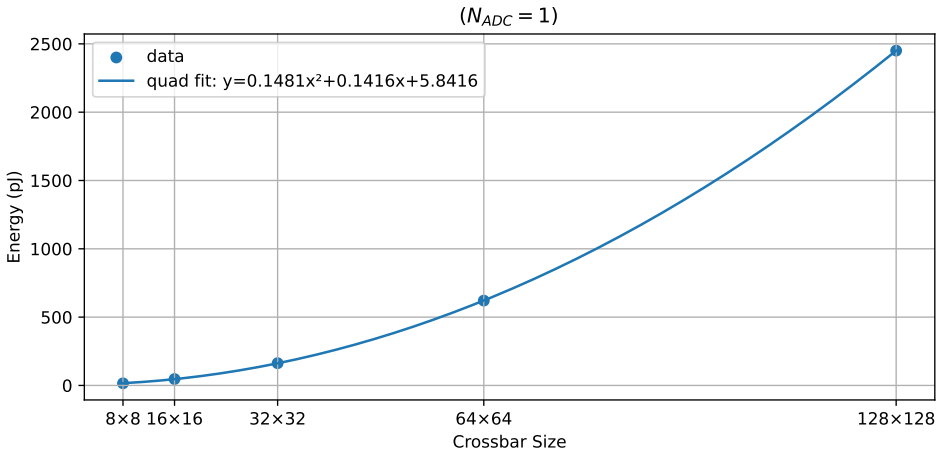


Figure 6.15: Read Operation Energy Overhead of Digital CIM-Tile Components vs. Crossbar Size

Figure 6.15 illustrates the read operation's energy overhead across crossbar sizes from 8×8 to 128×128. The total energy of read operations exhibits a quadratic relationship with crossbar size. This scaling arises from two compounding effects. First, the number of registers and combinational logic gates in the digital periphery increases linearly with crossbar size. Concurrently, with a fixed number of ADCs $N_{\mathrm{ADC}} = 1$, reading the increased number of columns requires a linearly growing number of `CSR` instructions, which extends the execution time. Consequently, the register internal energy caused by clock toggling increases quadratically, resulting from the product of the increased register count and the prolonged execution time. Second, the number of registers and combinational gates actively involved in the read operation within the read registers and Read/Logic Output Buffer also grows linearly with crossbar size. This increase in active components contributes to a linear rise in switching energy. Furthermore, leakage energy increases quadratically due to the combined effect of more circuit components operating over a longer duration. Together, these factors result in an overall quadratic dependence of total energy on crossbar size.

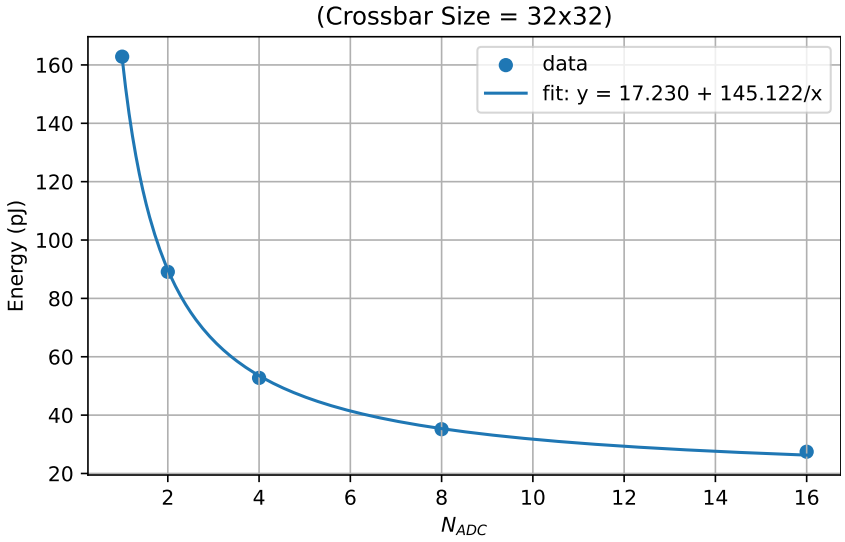


Figure 6.16: Read Operation Energy Overhead of Digital CIM-Tile Components vs. number of ADCs

Figure 6.16 reports the energy required to execute the read operation with varying the number of ADCs (N_{ADC} = 1, 2, 4, 8, and 16). The total energy of read operations decreases with the number of ADCs. This is primarily because a higher ADC count allows each `CSR` instruction to read more columns in

parallel, which reduces the total number of CSR instructions required and shortens the execution time. The resulting reduction in clock cycles leads to a significant decrease in the register internal energy dominated by clock toggling and the total leakage energy. Although the increased number of ADCs introduces additional registers and combinational logic in addition units, which slightly raises the internal energy from the extra registers and the leakage energy from the enlarged circuit scale, these increases are negligible compared to the substantial energy savings achieved from the shortened execution time. Consequently, the net effect is a clear decrease in total energy consumption as the number of ADCs grows.

6.3.4. Logic Operation Energy Result Analysis

This subsection presents the energy analysis for logic operations, including AND, OR, and XOR. Since these operations differ only in the specific FS instruction executed and the corresponding data selection and processing within the Logical Unit of the digital periphery, this experiment uses the AND operation as a representative case. The experiment was conducted by performing logical operations on data read from the crossbar array. Figure 6.17 shows an example of the instruction sequence used for the AND operation experiment.

number	Opcode	Field 1	Field 2
1.	FS	and	
2.	RDS	000	0011
3.	DoA		
4.	DoS		
5.	CSR	00000	1
6.	CSR	00001	1
7-35.	CSR	00010-11110	1
36.	CSR	11111	1
37.	CP		

Figure 6.17: An example of the instruction sequence for AND operation

The AND operation's energy consumption of read operation is primarily influenced by two parameters: the crossbar size and the number of ADCs N_{ADC}. The following analysis characterizes how energy scales with variations in these two parameters, examining their individual impacts on the overall energy consumption.

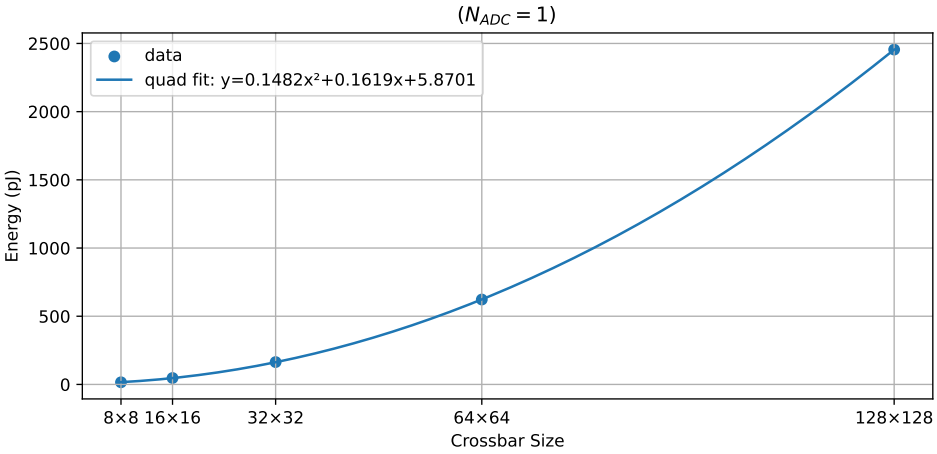


Figure 6.18: AND Operation Energy Overhead of Digital CIM-Tile Components vs. Crossbar Size

A quadratic scaling of total energy with crossbar size is observed, which can be attributed to the interplay of several factors. The expansion of crossbar size necessitates a proportional increase in the number of registers and combinational logic elements within the digital periphery. Under a fixed ADC

configuration ($N_{\mathrm{ADC}} = 1$), this dimensional growth demands a linear increment in CSR instructions to access the additional columns, thereby extending the execution time. The register internal energy, governed by clock toggling, consequently demonstrates quadratic growth as it scales with both the enlarged register population and the extended execution time. Simultaneously, the number of registers and combinational gates actively involved in the AND operation within the Logical Unit and Read/Logic Output Buffer also grows linearly with crossbar size. This increase in active components contributes to a linear rise in switching energy. The compounding influence of more circuit elements operating over longer execution time also produces quadratic growth in leakage energy. The collective contribution of these factors establishes the overall quadratic relationship between total AND operation's energy consumption and crossbar size.

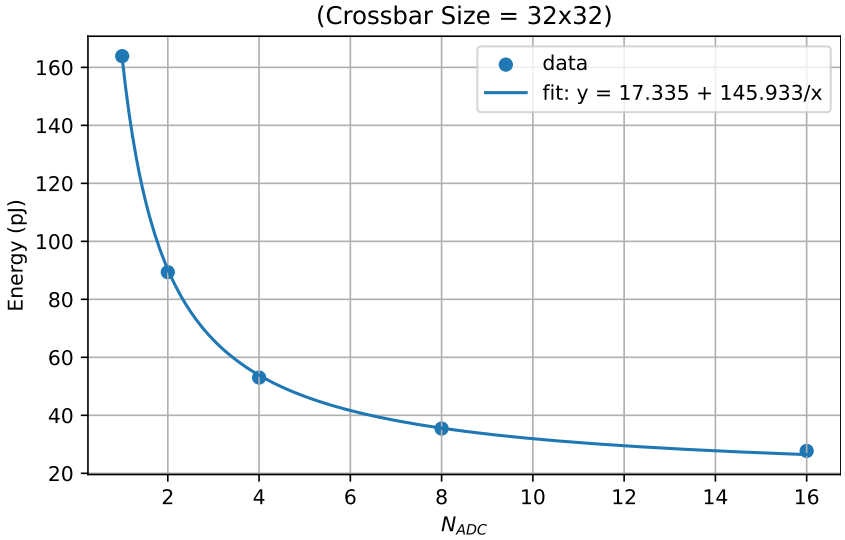


Figure 6.19: AND Operation Energy Overhead of Digital CIM-Tile Components vs. Number of ADCs

Figure 6.19 presents the energy measurements for AND operation under varying ADC counts (N_{ADC} = 1, 2, 4, 8, and 16). It reveals an inverse relationship between total AND operation's energy consumption and the number of ADCs. This trend originates from the enhanced parallelism enabled by additional ADCs, where each CSR instruction can process more columns concurrently. The improved parallelism reduces the total instruction count and shortens the execution timeline. This reduction produces substantial reductions in both register internal energy (clock-toggling dependent) and cumulative leakage energy. While the incorporation of additional ADCs does introduce supplementary registers and combinational logic that modestly elevate both internal and leakage energy components, these increments remain marginal when contrasted with the considerable energy conservation attained through execution time reduction. Consequently, the total AND operation's energy exhibits a clear inverse relationship with the number of ADCs.

6.3.5. VMM Operation Energy Result Analysis

This subsection presents the energy analysis for VMM operation. The energy consumption of VMM operation is influenced by three key parameters: the crossbar size, the number of ADCs N_{ADC}, and the data field width in the RDS instruction W_{data}. The experiment was conducted by reading the crossbar output and processing the digital data from ADCs in the addition units. Figure 6.20 shows an example of the instruction sequence used in this experiment.

Figure 6.21 illustrates the VMM operation's energy consumption across different crossbar sizes. The analysis reveals a linear relationship between energy consumption and crossbar size when executing identical VMM operations with the same input vector and matrix dimensions. This linear scaling can be attributed to the fixed computational workload: since the number of rows selected for the VMM operation remains constant (requiring the same number of RDS instructions), and the number of columns to be read is unchanged (maintaining the same CSR instruction count), the total execution time remains unaffected by crossbar size variations. Consequently, the observed energy increase stems solely from

number	Opcode	Field 1	Field 2
1.	FS	vmm	
2.	RDsh		
3.	RDS	000	1111
4-9.	RDS	001-110	1111
10.	RDS	111	1111
11.	DoA		
12.	DoS		
13.	LS		
14.	CSR	00000	1
15.	CSR	00001	1
16.	IADD		
17-58.	Repeat (CSR, CSR, IADD) 14×, with CSR Field 1 incrementing from 00010 to 11101.		
59.	CSR	11110	1
60.	CSR	11111	1
61.	IADD		
62.	CP		

Figure 6.20: An example of the instruction sequence for VMM operation

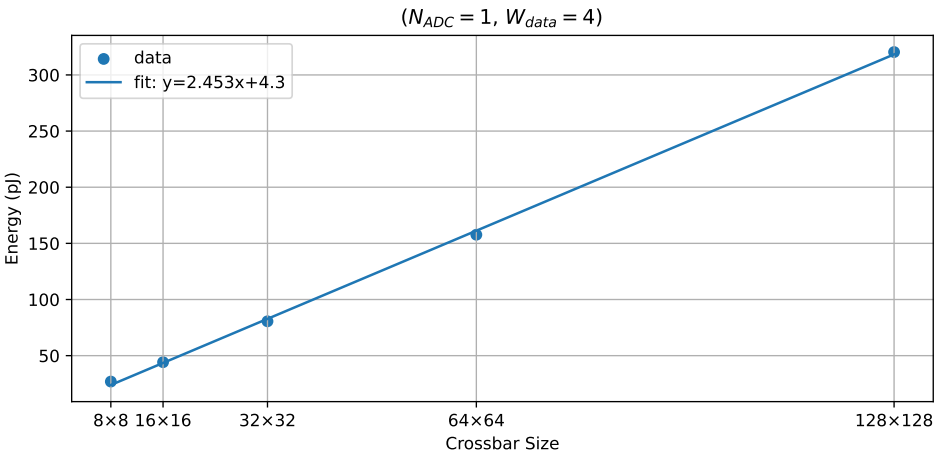


Figure 6.21: VMM Operation Energy Overhead of Digital CIM-Tile Components vs. Crossbar Size

the linear growth in the number of registers and combinational logic gates within the digital periphery. As the crossbar size increases, these additional components contribute proportionally to the overall energy consumption while the operation duration stays constant, resulting in the linear energy scaling characteristic.

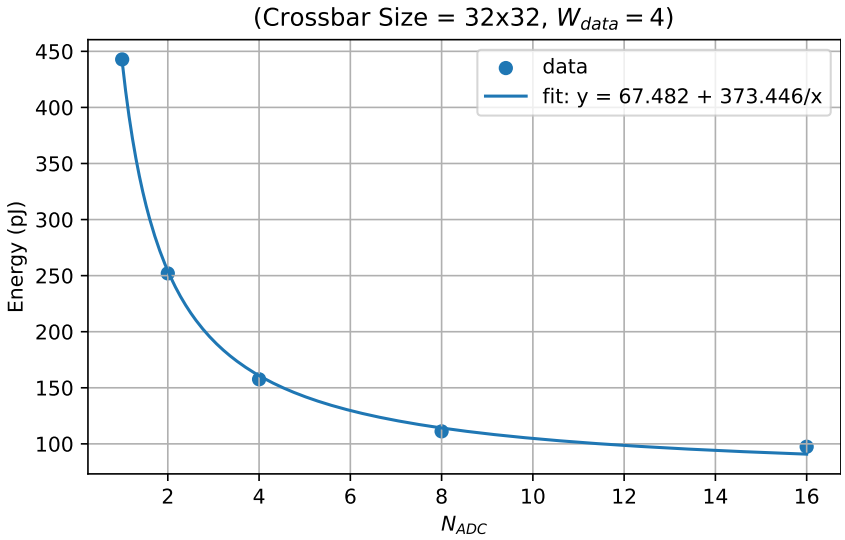


Figure 6.22: VMM Operation Energy Overhead of Digital CIM-Tile Components vs. number of ADCs

Figure 6.22 shows the VMM operation's energy consumption with varying numbers of ADCs. The total energy exhibits an inverse relationship with the ADC count. This is primarily because more ADCs reduce the number of `CSR` instructions required and shorten the execution time. The reduced clock cycles directly decrease the register internal energy dominated by clock toggling. Although additional ADCs introduce extra registers in the addition units, which slightly increases both the internal energy from these additional components and the leakage energy due to the enlarged circuit scale, these increases are negligible compared to the substantial energy savings from the shorter operation time. Consequently, the VMM operation's energy decreases while the number of ADCs increases.

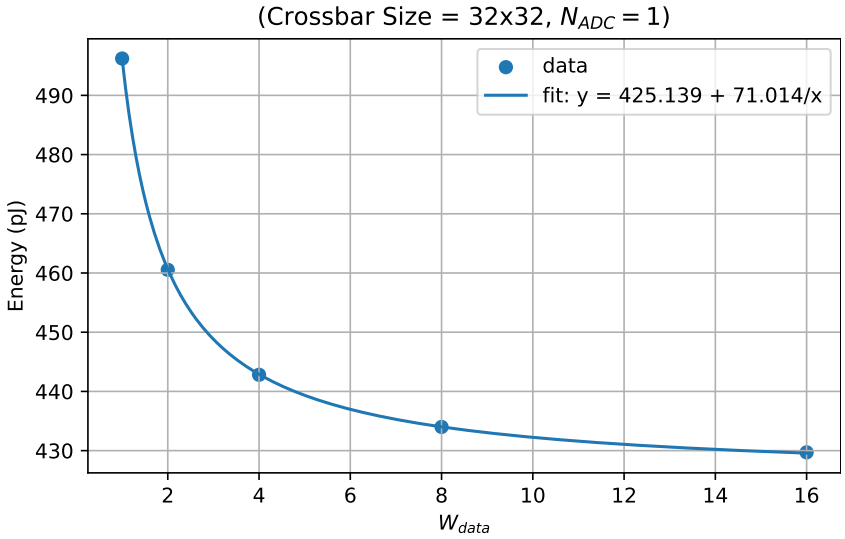


Figure 6.23: VMM Operation Energy Overhead of Digital CIM-Tile Components vs. the data field width in RDS instruction

Figure 6.23 presents the VMM energy consumption with varying operand widths (W_{data}). The analysis assumes the ADC resolution is sufficiently high to require multiple `RDS` instructions for selecting all necessary rows in VMM operation. The results show an inverse relationship between energy consumption and the data field width in `RDS` instruction. As W_{data} increases, each `RDS` instruction can select more

rows, reducing the total number of RDS instructions required to complete the VMM operation. This reduction in instruction count shortens the execution time, thereby decreasing both the register internal energy dominated by clock toggling and the total leakage energy. Consequently, the overall energy consumption decreases as the data field width increases.

6.4. Summary

This chapter has presented a comprehensive experimental evaluation of the digital modules in the proposed CIM tile architecture. The remapping functionality was validated at the RTL level, successfully demonstrating the complete error-handling and row-replacement flow. Further analysis was conducted through post-synthesis gate-level simulations using Cadence Genus, examining the area overhead and energy consumption of these digital modules.

Area Analysis shows different relationships between area overhead and architectural parameters. The total area grows linearly with crossbar size as digital modules scale proportionally with row or column counts. Similarly, area increases linearly with spare rows since each additional spare row requires corresponding registers and combinational logic in the Spare Row Selection module. For ADC count, area increases primarily due to replication of addition units, despite the reduction in index decoder complexity for CSR instruction and smaller multiplexers in each addition unit. In contrast, area decreases with external bus width due to simplified selection logic in the Write Data Buffer, which directs external data to appropriate segments within the chunks in the buffer. Area also decreases with WDS/RDS instruction data field width because of reduced decoder complexity for WDS/RDS instruction decoding and simplified selection logic in both Column and Row Selection modules. For operand width, area exhibits a non-monotonic relationship: it initially decreases due to reduced multiplexer area in the addition units and narrower Addition Output Buffer, but then increases as wider adders in the addition units and the linearly growing Row Data Buffer become dominant.

Energy Analysis characterizes the relationships between energy consumption and specific parameters for each operation type. For store operations, energy consumption decreases inversely with external bus width due to reduced write cycles for loading external data into the buffer, which shortens the execution time and thereby reduces the register internal energy dominated by clock toggling and leakage energy. Meanwhile, store operations show quadratic growth with crossbar size. This quadratic relationship arises because: (1) increased crossbar size linearly raises the number of registers and the cycles needed to write data, leading to quadratic growth in register internal energy from clock toggling; (2) more registers and combinational logic gates in the Write Data Buffer and Column Selection module contribute to linearly increased switching energy; and (3) leakage energy grows quadratically with both more circuit components and longer execution time.

Verification, read, and logic operations share similar relationships with crossbar size and number of ADCs: they all exhibit quadratic energy growth with crossbar size resulting from increased register count, extended execution time from more CSR instructions, and show inverse relationships with ADC count due to enhanced parallelism reducing instruction counts and execution time.

VMM operations displays different characteristics: linear scaling with crossbar size as identical VMM operations maintain fixed RDS and CSR instruction counts and execution time, with energy increase only from additional registers and combinational logic gates raising internal and leakage energy. They also show inverse relationship with ADC count due to parallel processing reducing CSR instructions and execution time, and inverse proportionality with the RDS instruction data field width when ADC resolution is sufficient, as wider RDS instructions reduce total RDS instruction count and shorten operation duration.

7

Conclusions

This thesis has presented the design, implementation, and evaluation of a programmable CIM tile architecture based on 1T1R memristor crossbar arrays, aimed at overcoming the von Neumann bottleneck by enabling computation directly within memory.

We have proposed a complete CIM tile architecture that integrates analog interfaces, a comprehensive digital periphery, a dedicated instruction set, and a five-stage pipeline to support a wide range of operations including write, read, verification, Boolean logic (AND, OR, XOR), and VMM operations. A key contribution is the RTL and gate-level implementation of the digital periphery, which features a novel Verification Unit capable of offline write validation, selective rewriting, and automatic row remapping to spare rows in case of persistent failures. This unit enhances the reliability and fault tolerance of the CIM tile.

Furthermore, based on our own gate-level implementation, this thesis develops analytical module-level area models and instruction-level energy models. These energy models precisely attribute the switching, internal, and leakage energy consumption to the specific digital periphery modules activated by each instruction.

A comprehensive evaluation was conducted on our implementation, encompassing both functional validation and post-synthesis performance analysis. The remapping functionality was verified through RTL simulation. Concurrently, a post-synthesis evaluation of the gate-level implementation was performed. Area Analysis shows that the digital periphery's area scales linearly with crossbar size and the number of spare rows. It exhibits a non-monotonic relationship with operand width: area initially decreases due to a reduction in multiplexer and selection logic within the Addition Unit's input-vector segment stage, before increasing as wider adders in the Addition Units and a linearly scaling Row Data Buffer begin to dominate. Furthermore, area demonstrates an inverse relationship with both the external bus width, which simplifies the Write Data Buffer's segment selection logic, and the WDS/RDS instruction data field width, which reduces the complexity of the one-hot decoders in the Column and Row Selection modules, while showing an increasing trend with ADC count due to the replication of peripheral circuits. Energy Analysis reveals distinct scaling laws for different operations. Store operations exhibit quadratic scaling with crossbar size and inverse proportionality with the external bus width. Verification, read, and logic operations also show quadratic scaling with crossbar size but benefit from an inverse relationship with ADC count. In contrast, VMM operations display linear scaling with crossbar size, an inverse relationship with ADC count, and inverse proportionality with the WDS/RDS instruction data field width. These findings provide concrete guidance for optimizing key architectural parameters in practical CIM tile designs.

In summary, this thesis presents a programmable CIM tile architecture with an integrated verification mechanism. The work delivers a complete RTL and gate-level implementation of the digital periphery, and provides a comprehensive scaling analysis of area and energy consumption. Together, these contributions provide a solid foundation for designing efficient and reliable CIM accelerators.

7.1. Future Work

Based on the solid foundation established in this thesis, several promising directions emerge for future research. The computational capabilities of the architecture can be enhanced by extending support for signed number representations in VMM operations. This advancement would necessitate corresponding modifications to the Addition Unit to handle two's complement arithmetic and may require extensions to the instruction set architecture. Furthermore, exploring support for multi-operand logic operations beyond the current two-row implementation would substantially increase the architecture's functional coverage.

To transition from post-synthesis verification to practical system implementation, the design could be deployed on a Zynq-like SoC platform. This approach would enable comprehensive validation of the instruction pipeline and dataflow management, providing a better assessment of system performance.

To facilitate rapid architectural exploration and software development prior to RTL commitment, the design of the CIM tile can be modeled using SystemC/C++. This high-level model will emulate the tile's behavior, including the controller, digital periphery, and a functional model of the crossbar array. It will serve as a fast validation platform for software toolchains and a good reference for RTL design verification.

References

[1] S. Ambrogio et al. “Reducing the Impact of Phase-Change Memory Conductance Drift on the Inference of large-scale Hardware Neural Networks”. In: *2019 IEEE International Electron Devices Meeting (IEDM)*. 2019, pp. 6.1.1–6.1.4. DOI: 10.1109/IEDM19573.2019.8993482.

[2] Aayush Ankit et al. “PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 715–731. ISBN: 9781450362405. DOI: 10.1145/3297858.3304049. URL: https://doi.org/10.1145/3297858.3304049.

[3] Aayush Ankit et al. “PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference”. In: *CoRR* abs/1901.10351 (2019). arXiv: 1901.10351. URL: http://arxiv.org/abs/1901.10351.

[4] G. Baccarani and E. Gnani. “Memory Devices, Nonvolatile”. In: *Encyclopedia of Condensed Matter Physics*. Ed. by Franco Bassani, Gerald L. Liedl, and Peter Wyder. Oxford: Elsevier, 2005, pp. 324–337. ISBN: 978-0-12-369401-0. DOI: https://doi.org/10.1016/B0-12-369401-9/01167-0. URL: https://www.sciencedirect.com/science/article/pii/B0123694019011670.

[5] John Backus. “Can programming be liberated from the von Neumann style? a functional style and its algebra of programs”. In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579. URL: https://doi.org/10.1145/359576.359579.

[6] Christopher Bengel et al. “Variability-Aware Modeling of Filamentary Oxide-Based Bipolar Resistive Switching Cells Using SPICE Level Compact Models”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 67.12 (2020), pp. 4618–4630. DOI: 10.1109/TCSI.2020.3018502.

[7] Amirali Boroumand et al. “Google workloads for consumer devices: Mitigating data movement bottlenecks”. In: *Proceedings of the twenty-third international conference on architectural support for programming languages and operating systems*. 2018, pp. 316–331.

[8] Umesh Chand et al. “Suppression of endurance degradation by utilizing oxygen plasma treatment in HfO2 resistive switching memory”. In: *Applied Physics Letters* 106 (Apr. 2015), p. 153502. DOI: 10.1063/1.4918679.

[9] Tianshi Chen et al. “DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning”. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 269–284. ISBN: 9781450323055. DOI: 10.1145/2541940.2541967. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2541940.2541967.

[10] Wenbo Chen et al. “Switching characteristics of W/Zr/HfO2/TiN ReRAM devices for multi-level cell non-volatile memory applications”. In: *Semiconductor Science and Technology* 30 (June 2015). DOI: 10.1088/0268-1242/30/7/075002.

[11] Ping Chi et al. “PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA ’16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 27–39. ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.13. URL: https://doi.org/10.1109/ISCA.2016.13.

[12] L. Chua. “Memristor-The missing circuit element”. In: *IEEE Transactions on Circuit Theory* 18.5 (1971), pp. 507–519. DOI: 10.1109/TCT.1971.1083337.

[13] M. Fieback. “Testing RRAM and Computation-in-Memory Devices: Defects, Fault Models, and Test Solutions”. English. Dissertation (TU Delft). Delft University of Technology, 2022. DOI: 10.4233/uuid:71e1cbf8-ce02-4ce8-a09b-883e6f84e996.

[14] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. “In-Memory Data Parallel Processor”. In: 53.2 (Mar. 2018), pp. 1–14. ISSN: 0362-1340. DOI: 10.1145/3296957.3173171. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3296957.3173171.

[15] S. Ghose et al. “Processing-in-memory: A workload-driven perspective”. In: *IBM Journal of Research and Development* 63.6 (2019), 3:1–3:19. DOI: 10.1147/JRD.2019.2934048.

[16] Chung-Wei Hsu et al. “Self-rectifying bipolar TaOx/TiO2 RRAM with superior endurance over 1012 cycles for 3D high-density storage-class memory”. In: *2013 Symposium on VLSI Technology*. 2013, T166–T167.

[17] Daniele Ielmini and H.-S. Philip Wong. “In-memory computing with resistive switching devices”. In: *Nature Electronics* 1 (2018), pp. 333–343. URL: https://api.semanticscholar.org/CorpusID:57248729.

[18] Sungho Kim, Hee-Dong Kim, and Sung-Jin Choi. “Numerical study of read scheme in one-selector one-resistor crossbar array”. In: *Solid-State Electronics* 114 (2015), pp. 80–86. ISSN: 0038-1101. DOI: https://doi.org/10.1016/j.sse.2015.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0038110115002294.

[19] Jung-Hoon Lee et al. “Exploring Cycle-to-Cycle and Device-to-Device Variation Tolerance in MLC Storage-Based Neural Network Training”. In: *IEEE Transactions on Electron Devices* 66.5 (2019), pp. 2172–2178. DOI: 10.1109/TED.2019.2906249.

[20] Myoung-Jae Lee et al. “A Fast, High-Endurance and Scalable Non-Volatile Memory Device Made from Asymmetric Ta2O5−X/TaO2−X Bilayer Structures”. In: *Nature materials* 10 (July 2011), pp. 625–30. DOI: 10.1038/nmat3070.

[21] Bing Li, Bonan Yan, and Hai Li. “An Overview of In-memory Processing with Emerging Non-volatile Memory for Data-intensive Applications”. In: *Proceedings of the 2019 Great Lakes Symposium on VLSI*. GLSVLSI ’19. ACM, May 2019, pp. 381–386. DOI: 10.1145/3299874.3319452. URL: http://dx.doi.org/10.1145/3299874.3319452.

[22] Chunlin Li et al. “Adaptive priority-based cache replacement and prediction-based cache prefetching in edge computing environment”. In: *Journal of Network and Computer Applications* 165 (2020), p. 102715. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2020.102715. URL: https://www.sciencedirect.com/science/article/pii/S1084804520301892.

[23] Qingxuan Li et al. “High-performance ferroelectric field-effect transistors with ultra-thin indium tin oxide channels for flexible and transparent electronics”. In: *Nature Communications* 15 (Mar. 2024). DOI: 10.1038/s41467-024-46878-5.

[24] Gabriel H. Loh. “3D-Stacked Memory Architectures for Multi-core Processors”. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA ’08. USA: IEEE Computer Society, 2008, pp. 453–464. ISBN: 9780769531748. DOI: 10.1109/ISCA.2008.15. URL: https://doi.org/10.1109/ISCA.2008.15.

[25] Harika Manem and Garrett S. Rose. “A read-monitored write circuit for 1T1M multi-level memristor memories”. In: *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. 2011, pp. 2938–2941. DOI: 10.1109/ISCAS.2011.5938207.

[26] J. von Neumann. “First draft of a report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. DOI: 10.1109/85.238389.

[27] Allan Porterfield. “Software Methods for Improvement of Cache Performance on Supercomputer Applications”. PhD thesis. Jan. 1989.

[28] Allan Porterfield. “Software Methods for Improvement of Cache Performance on Supercomputer Applications”. PhD thesis. Jan. 1989.

[29] Ali Shafiee et al. “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 14–26. DOI: 10.1109/ISCA.2016.12.

[30] Lingyun Shi et al. “Research progress on solutions to the sneak path issue in memristor crossbar arrays”. In: *Nanoscale Adv.* 2 (5 2020), pp. 1811–1827. DOI: 10.1039/D0NA00100G. URL: http://dx.doi.org/10.1039/D0NA00100G.

[31] Abhairaj Singh et al. “Accelerating RRAM Testing with a Low-cost Computation-in-Memory based DFT”. In: *2022 IEEE International Test Conference (ITC)*. 2022, pp. 400–409. DOI: 10.1109/ITC50671.2022.00085.

[32] Dmitri B. Strukov et al. “The missing memristor found”. In: *Nature* 453 (2008), pp. 80–83. URL: https://api.semanticscholar.org/CorpusID:4367148.

[33] Chao Su and Qingkai Zeng. “Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures”. In: *Security and Communication Networks* 2021.1 (2021), p. 5559552. DOI: https://doi.org/10.1155/2021/5559552. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2021/5559552. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/5559552.

[34] Z. Wei et al. “Demonstration of high-density ReRAM ensuring 10-year retention at 85°C based on a newly developed reliability model”. In: *2011 International Electron Devices Meeting*. 2011, pp. 31.4.1–31.4.4. DOI: 10.1109/IEDM.2011.6131650.

[35] Z. Wei et al. “Retention Model for High-Density ReRAM”. In: *2012 4th IEEE International Memory Workshop*. 2012, pp. 1–4. DOI: 10.1109/IMW.2012.6213638.

[36] Christopher Wolters et al. “Memory Is All You Need: An Overview of Compute-in-Memory Architectures for Accelerating Large Language Model Inference”. In: *CoRR* abs/2406.08413 (2024). DOI: 10.48550/ARXIV.2406.08413. arXiv: 2406.08413. URL: https://doi.org/10.48550/arXiv.2406.08413.

[37] H.-S. Philip Wong et al. “Metal–Oxide RRAM”. In: *Proceedings of the IEEE* 100.6 (2012), pp. 1951–1970. DOI: 10.1109/JPROC.2012.2190369.

[38] Quantan Wu et al. “Improvement of durability and switching speed by incorporating nanocrystals in the HfO x based resistive random access memory devices”. In: *Applied Physics Letters* 113 (July 2018), p. 023105. DOI: 10.1063/1.5030780.

[39] Wm Wulf and Sally McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *Computer Architecture News* 23 (Jan. 1996).

[40] Xiaoxuan Yang et al. “Research Progress on Memristor: From Synapses to Computing Systems”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 69.5 (2022), pp. 1845–1857. DOI: 10.1109/TCSI.2022.3159153.

[41] M.Z. Zahedi. “Computation-in-memory from application-specific to programmable designs based on memristor devices”. English. Dissertation (TU Delft). Delft University of Technology, 2023. ISBN: 978-94-6366-789-0. DOI: 10.4233/uuid:e0a6d2e3-6ec6-4edc-8e6a-5ae931d7dffb.

[42] Mahdi Zahedi et al. “MNEMOSENE: Tile Architecture and Simulator for Memristor-based Computation-in-memory”. English. In: *ACM Journal on Emerging Technologies in Computing Systems* 18.3 (2022). ISSN: 1550-4832. DOI: 10.1145/3485824.

[43] Mahdi Zahedi et al. “Tile Architecture and Hardware Implementation for Computation-in-Memory”. English. In: *Proceedings - 2021 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2021*. Ed. by Cristina Ceballos. Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI. 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI) ; Conference date: 07-07-2021 Through 09-07-2021. United States: IEEE, 2021, pp. 108–113. ISBN: 978-1-6654-3947-3. DOI: 10.1109/ISVLSI51109.2021.00030.

[44] Yaojun Zhang et al. “STT-RAM Cell Optimization Considering MTJ and CMOS Variations”. In: *IEEE Transactions on Magnetics* 47.10 (2011), pp. 2962–2965. DOI: 10.1109/TMAG.2011.2158810.

[45] M. A. Zidan et al. “Single-Readout High-Density Memristor Crossbar”. In: *Scientific Reports* 6 (2016), p. 18863. DOI: 10.1038/srep18863. URL: https://www.nature.com/articles/srep18863.

A

Appendix

A.1. Energy Model Parameters

n_1: the number of '1' in the `WDS_enable_back_up` signals in the Column Selection module; Range: $0 \leq n_1 \leq C,\ n_1 \in \mathbb{Z}$.

n_2: the number of changing bits in the `column_select` outputs in the Column Selection module; Range: $0 \leq n_2 \leq C,\ n_2 \in \mathbb{Z}$.

n_3: the number of changing bits in the data field of the `WDS` instruction; Range: $0 \leq n_3 \leq W_{\text{data}},\ n_3 \in \mathbb{Z}$.

n_4: the number of '1' in the `vrf_sign` inputs from Verification Unit; Range: $0 \leq n_4 \leq C,\ n_4 \in \mathbb{Z}$.

n_5: the number of changing bits in the `row_select` outputs in the Row Selection module; Range: $0 \leq n_5 \leq R,\ n_5 \in \mathbb{Z}$.

n_6: the number of changing bits in the `row_select_data` outputs in the Row Data Selection module; Range: $0 \leq n_6 \leq R,\ n_6 \in \mathbb{Z}$.

n_7: the number of '1' in outputs of the Row Data Buffer; Range: $0 \leq n_7 \leq R,\ n_7 \in \mathbb{Z}$.

n_8: the number of changing bits in the data field of the `RDS` instruction;; Range: $0 \leq n_8 \leq W_{\text{data}},\ n_8 \in \mathbb{Z}$.

n_9: the number of selected spare rows in the Spare Row Selection module; Range: $0 \leq n_9 \leq R_{\text{s}},\ n_9 \in \mathbb{Z}$.

n_{10}: the number of '1' in the `spare_row_select_data` outputs in the Spare Row Selection module; Range: $0 \leq n_{10} \leq R_{\text{s}},\ n_{10} \in \mathbb{Z}$.

n_{11}: the number of '1' in the data field of the `CSR` instruction; Range: $1 \leq n_{11} \leq N_{\text{ADC}},\ n_{11} \in \mathbb{Z}$.

n_{12}: the number of changing bits at the Read Register input (from the ADC); Range: $0 \leq n_{12} \leq N_{\text{ADC}},\ n_{12} \in \mathbb{Z}$.

n_{13}: the number of changing bits in the output data of the Read Register; Range: $0 \leq n_{13} \leq C,\ n_{13} \in \mathbb{Z}$.

n_{14}: the number of '1' in the outputs from the Write Data Buffer; Range: $0 \leq n_{14} \leq C,\ n_{14} \in \mathbb{Z}$.

n_{15}: the number of changing bits in the vrf_xor signals in the Verification Unit; Range: $0 \leq n_{15} \leq C,\ n_{15} \in \mathbb{Z}$.

n_{16}: the number of changing bits in the vrf_sign signals in the Verification Unit; Range: $0 \leq n_{16} \leq C,\ n_{16} \in \mathbb{Z}$.

n_{17}: the number of changing bits in the vrf_result signals in the Verification Unit; Range: $0 \leq n_{17} \leq C-1,\ n_{17} \in \mathbb{Z}$.

n_{18}: the changing state of the remap_flag of the Verification Unit; Range: $0 \leq n_{18} \leq 1,\ n_{18} \in \mathbb{Z}$.

n_{19}: the number of '1' at the Logical Unit input data (from ADC); Range: $0 \leq n_{19} \leq N_{\mathrm{ADC}},\ n_{19} \in \mathbb{Z}$.

n_{20}: the number of changing bits in the `logic_sign` outputs in the Logical Unit; Range: $0 \leq n_{20} \leq C,\ n_{20} \in \mathbb{Z}$.

n_{21}: the number of changing bits in the opcode field of the input instructions to the Decoder1; Range: $0 \leq n_{21} \leq 3,\ n_{21} \in \mathbb{Z}$.

n_{22}: the number of changing bits in the index field of the WDS instruction;
Range: $0 \leq n_{22} \leq W_{\mathrm{data}},\ n_{22} \in \mathbb{Z}$.

n_{23}: the number of changing bits in the index field of the RDS instruction;
Range: $0 \leq n_{23} \leq W_{\mathrm{data}},\ n_{23} \in \mathbb{Z}$.

n_{24}: the number of '1' in the index field of the FS instruction; Range: $1 \leq n_{24} \leq 4,\ n_{24} \in \mathbb{Z}$.

n_{25}: the number of changing bits in the index field of FS instruction from the previous to the current operation; Range: $0 \leq n_{25} \leq 3,\ n_{25} \in \mathbb{Z}$.

n_{26}: the number of changing bits in the sum outputs of the instruction-memory address adder in Decoder1; Range: $1 \leq n_{26} \leq W_{\mathrm{PC1}},\ n_{26} \in \mathbb{Z}$.

n_{27}: the number of changing bits in the instruction-memory address adder's internal carry chain in the Decoder1; Range: $0 \leq n_{27} \leq (W_{\mathrm{PC1}} - 2),\ n_{27} \in \mathbb{Z}$.

n_{28}: the number of changing bits in the instruction-memory address in the Decoder1; Range: $1 \leq n_{28} \leq W_{\mathrm{PC1}},\ n_{28} \in \mathbb{Z}$.

n_{29}: the number of changing bits in the address stored in the branch registers for verification in the Decoder1; Range: $1 \leq n_{29} \leq W_{\mathrm{PC1}},\ n_{29} \in \mathbb{Z}$.

n_{30}: the number of different bits in the address stored in the branch registers for verification with the sum outputs of the instruction-memory address adder in the Decoder1; Range: $1 \leq n_{30} \leq W_{\mathrm{PC1}},\ n_{30} \in \mathbb{Z}$.

n_{31}: the number of changing bits in the opcode field of the input instructions to the Decoder2; Range: $0 \leq n_{31} \leq 4,\ n_{31} \in \mathbb{Z}$.

n_{32}: the number of changing bits of the index field of the `CSR` instruction;
Range: $1 \leq n_{32} \leq \log_2(C/N_{\mathrm{ADC}}),\ n_{32} \in \mathbb{Z}$.

n_{33}: the number of changing bits of the data field of the CSR instruction; Range: $0 \leq n_{33} \leq N_{\mathrm{ADC}},\ n_{33} \in \mathbb{Z}$.

n_{34}: the number of '1' in the address field of the JAL instruction; Range: $1 \leq n_{34} \leq W_{\mathrm{PC2}},\ n_{34} \in \mathbb{Z}$.

n_{35}: the number of changing bits of instruction-memory address registers' inputs; Range: $1 \leq n_{35} \leq W_{\mathrm{PC2}},\ n_{35} \in \mathbb{Z}$.

n_{36}: the number of changing bits in the instruction-memory address in the Decoder2; Range: $1 \leq n_{36} \leq W_{\mathrm{PC2}},\ n_{36} \in \mathbb{Z}$.

n_{37}: the number of '1' in the instruction-memory address in the Decoder2;
Range: $1 \leq n_{37} \leq W_{\mathrm{PC2}},\ n_{37} \in \mathbb{Z}$.

n_{38}: the number of changing bits in the sum outputs of the instruction-memory address adder in Decoder2; Range: $1 \leq n_{38} \leq W_{\mathrm{PC2}},\ n_{38} \in \mathbb{Z}$.

n_{39}: the number of changing bits in the instruction-memory address adder's internal carry chain in the Decoder2; Range: $0 \leq n_{39} \leq (W_{\mathrm{PC2}} - 2),\ n_{39} \in \mathbb{Z}$.

n_{40}: the number of '1' in the sum outputs of the instruction-memory address adder in Decoder2; Range: $1 \leq n_{40} \leq W_{\mathrm{PC2}},\ n_{40} \in \mathbb{Z}$.

n_{41}: the number of changing bits in the sum outputs of the JAL target address adder in Decoder2; Range: $1 \leq n_{41} \leq W_{\mathrm{PC2}},\ n_{41} \in \mathbb{Z}$.

n_{42}: the number of changing bits in the JAL target address adder's internal carry chain in the Decoder2; Range: $0 \leq n_{42} \leq (W_{\mathrm{PC2}} - 2),\ n_{42} \in \mathbb{Z}$.

n_{43}: the number of changing bits in the address stored in the branch registers for JR instruction in the Decoder2; Range: $1 \leq n_{43} \leq W_{\mathrm{PC2}},\ n_{43} \in \mathbb{Z}$.

n_{44}: the number of '1' in the address stored in the branch registers for JR instruction in the Decoder2; Range: $1 \leq n_{44} \leq W_{\mathrm{PC2}},\ n_{44} \in \mathbb{Z}$.

n_{45}: the number of changing bits in the address stored in the branch registers for verification in the Decoder2; Range: $1 \leq n_{45} \leq W_{\mathrm{PC2}},\ n_{45} \in \mathbb{Z}$.

n_{46}: the number of '1' in the address stored in the branch registers for verification in the Decoder2; Range: $1 \leq n_{46} \leq W_{\mathrm{PC2}},\ n_{46} \in \mathbb{Z}$.

n_{47}: the number of '1' in the ADC Addition Unit inputs; Range: $0 \leq n_{47} \leq \lceil \log_2(R+1) \rceil,\ n_{47} \in \mathbb{Z}$.

n_{48}: the index of the selected column by CSR instruction; Range: $0 \leq n_{48} \leq C/N_{\mathrm{ADC}},\ n_{48} \in \mathbb{Z}$.

n_{49}: the number of '1' in the temporary registers in the Analog Accumulation Stage of the selected column by the CSR instruction; Range: $0 \leq n_{49} \leq \lceil \log_2(R+1) \rceil,\ n_{49} \in \mathbb{Z}$.

n_{50}: the number of changing bits in the temporary registers in the Analog Accumulation Stage; Range: $0 \leq n_{50} \leq \lceil \log_2(R+1) \rceil,\ n_{50} \in \mathbb{Z}$.

n_{51}: the number of '1' in the sum outputs of the adder in the Analog Accumulation Stage; Range: $0 \leq n_{51} \leq \lceil \log_2(R+1) \rceil,\ n_{51} \in \mathbb{Z}$.

n_{52}: the number of '1' in the adder's internal carry chain in the Analog Accumulation Stage; Range: $0 \leq n_{52} \leq (\lceil \log_2(R+1) \rceil - 2),\ n_{52} \in \mathbb{Z}$.

n_{53}: the number of changing bits in outputs of the Analog Accumulation Stage; Range: $0 \leq n_{53} \leq \lceil \log_2(R+1) \rceil,\ n_{53} \in \mathbb{Z}$.

n_{54}: the number of changing bits in the temporary registers in the Multi-column sliding addition stage; Range: $0 \leq n_{54} \leq \lceil \log_2(R+1) \rceil,\ n_{54} \in \mathbb{Z}$.

n_{55}: the number of changing bits in the sum outputs of the adder in the Multi-column sliding addition stage; Range: $0 \leq n_{55} \leq \lceil \log_2(R+1) \rceil,\ n_{55} \in \mathbb{Z}$.

n_{56}: the number of changing bits in the adder's internal carry chain in the Multi-column sliding addition stage; Range: $0 \leq n_{56} \leq (\lceil \log_2(R+1) \rceil - 2),\ n_{56} \in \mathbb{Z}$.

n_{57}: the number of changing bits in outputs of the Multi-column sliding addition stage (indices 0 - W_{operand} - 2); Range: $0 \leq n_{57} \leq (W_{\mathrm{operand}} - 1),\ n_{57} \in \mathbb{Z}$.

n_{58}: the number of '1' in outputs of the Multi-column sliding addition stage; Range: $0 \leq n_{58} \leq (\lceil \log_2(R+1) \rceil + W_{\mathrm{operand}}),\ n_{58} \in \mathbb{Z}$.

n_{59}: the number of '1' in the temporary registers of the selected operand in the Input vector segment addition stage; Range: $0 \leq n_{59} \leq (\lceil \log_2(R+1) \rceil + W_{\mathrm{operand}}),\ n_{59} \in \mathbb{Z}$.

n_{60}: the number of changing bits in the sum outputs of the adder in the Input vector segment addition stage; Range: $0 \leq n_{60} \leq (\lceil \log_2(R+1) \rceil + W_{\mathrm{operand}}),\ n_{60} \in \mathbb{Z}$.

n_{61}: the number of changing bits in the adder's internal carry chain in the Input vector segment addition stage; Range: $0 \leq n_{61} \leq (\lceil \log_2(R+1) \rceil + W_{\mathrm{operand}} - 2),\ n_{61} \in \mathbb{Z}$.

n_{62}: the number of changing bits in outputs of the Input vector segment addition stage; Range: $0 \leq n_{62} \leq (\lceil \log_2(R+1) \rceil + 2W_{\mathrm{operand}}),\ n_{62} \in \mathbb{Z}$.

n_{63}: the number of changing bits in input data of the Write Data Buffer; Range: $0 \leq n_{63} \leq W_{\mathrm{bus}},\ n_{63} \in \mathbb{Z}$.

n_{64}: the number of changing bits in the registers used for buffering in the Write Data Buffer; Range: $0 \leq n_{64} \leq W_{\mathrm{bus}},\ n_{64} \in \mathbb{Z}$.

n_{65}: the number of changing bits in outputs of the Write Data Buffer; Range: $0 \leq n_{65} \leq C,\ n_{65} \in \mathbb{Z}$.

n_{66}: the number of changing bits in the write pointer in the Write Data Buffer; Range: $1 \leq n_{66} \leq 2,\ n_{66} \in \mathbb{Z}$.

n_{67}: the number of changing bits in the read pointer in the Write Data Buffer; Range: $1 \leq n_{67} \leq 2,\ n_{67} \in \mathbb{Z}$.

n_{68}: the number of changing bits in input data of the Row Data Buffer; Range: $0 \leq n_{68} \leq W_{\text{bus}},\ n_{68} \in \mathbb{Z}$.

n_{69}: the number of changing bits in the registers used for buffering in the Row Data Buffer; Range: $0 \leq n_{69} \leq W_{\text{bus}},\ n_{69} \in \mathbb{Z}$.

n_{70}: the number of changing bits in outputs of the Row Data Buffer; Range: $0 \leq n_{70} \leq R,\ n_{70} \in \mathbb{Z}$.

n_{71}: the number of changing bits in the registers used for buffering in the Read/Logic Output Buffer; Range: $0 \leq n_{71} \leq C,\ n_{71} \in \mathbb{Z}$.

n_{72}: the number of '1' in outputs of the Read/Logic Output Buffer; Range: $0 \leq n_{72} \leq C,\ n_{72} \in \mathbb{Z}$.

n_{73}: the number of changing bits in outputs of the Read/Logic Output Buffer; Range: $0 \leq n_{73} \leq C,\ n_{73} \in \mathbb{Z}$.

n_{74}: the number of changing bits in the write pointer in the Read/Logic Output Buffer; Range: $1 \leq n_{74} \leq 2,\ n_{74} \in \mathbb{Z}$.

n_{75}: the number of changing bits in the read pointer in the Read/Logic Output Buffer; Range: $1 \leq n_{75} \leq 2,\ n_{75} \in \mathbb{Z}$.

n_{76}: the number of changing bits in the registers used for buffering in the Addition Output Buffer; Range: $0 \leq n_{76} \leq ((C\lceil \log_2(R+1) \rceil)/W_{\text{operand}} + 2C),\ n_{76} \in \mathbb{Z}$.

n_{77}: the number of '1' in outputs of the Addition Output Buffer; Range: $0 \leq n_{77} \leq ((C\lceil \log_2(R+1) \rceil)/W_{\text{operand}} + 2C),\ n_{77} \in \mathbb{Z}$.

n_{78}: the number of changing bits in outputs of the Addition Output Buffer;
Range: $0 \leq n_{78} \leq ((C\lceil \log_2(R+1) \rceil)/W_{\text{operand}} + 2C),\ n_{78} \in \mathbb{Z}$.

n_{79}: the number of changing bits in the write pointer in the Addition Output Buffer; Range: $1 \leq n_{79} \leq 2,\ n_{79} \in \mathbb{Z}$.

n_{80}: the number of changing bits in the read pointer in the Addition Output Buffer; Range: $1 \leq n_{80} \leq 2,\ n_{80} \in \mathbb{Z}$.

C_{DIM}: the input capacitance of the DIM; We assume 50fF.

C_{memory}: the input capacitance of the instruction memory; We assume 50fF.

$C_{outside}$: the input capacitance of the interface between the CIM Tile and outside; We assume 50fF.

A.2. Enegy Model Table

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR2X1	2	$0.4 + 0.4C$	1	0.592
2	INVX1	2	0.2	1	0.232
3	EDFFTRX1	1	0.2	n_1	/
4	OR3X1	2	$0.2C/W_{\text{data}}$	1	0.626
5	OR2X1	2	$0.4W_{\text{data}}$	C/W_{data}	0.594
6	EDFFTRX1	1	$0.4 + C_{DIM}$	n_2	/
register internal energy					
7	EDFFTRX1	2	only EN changes	$2C$	1.56
8	EDFFTRX1	2	only Q changes	$n_1 + n_2$	1.92

Table A.1: Column Selection module's Energy model Table for FS (store).

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR3X1	2	$0.2C/W_{\text{data}}$	1	
2	OR2X1	2	$0.4W_{\text{data}}$	C/W_{data}	
3	OR4X1	2	0.2	C	
4	EDFFTRX1	1	$0.4 + C_{DIM}$	C	/
register internal energy					
5	EDFFTRX1	2	only D changes	$2C$	1.42
6	EDFFTRX1	2	only EN changes	$2C$	1.56
7	EDFFTRX1	2	only Q changes	C	1.92

Table A.2: Column Selection module's Energy model Table for WDSs.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	2	0.2	1	
2	OR2X1	2	$0.4W_{\text{data}}$	1	
3	OR4X1	2	0.2	n_3C/W_{data}	
4	EDFFTRX1	1	$0.4 + C_{DIM}$	n_2	/
register internal energy					
5	EDFFTRX1	2	only D changes	n_3C/W_{data}	1.42
6	EDFFTRX1	2	only EN changes	W_{data}	1.56
7	EDFFTRX1	2	only Q changes	n_2	1.92

Table A.3: Column Selection module's Energy model Table for WDS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.2	1	
2	AND2X1	2	0.2	$2n_1 + n_4$	
3	AND2X1	2	$0.2C$	2	
4	AND2X1	2	$0.2 + 0.2C$	1	
5	OR2X1	2	$0.4W_{\text{data}}$	C/W_{data}	
6	OR2X1	2	$0.4 + 0.4C$	1	
7	OR3X1	2	$0.2C/W_{\text{data}}$	1	
8	OR4X1	2	0.2	$n_1 + n_4$	
9	EDFFTRX1	1	0.2	n_1	/
10	EDFFTRX1	1	$0.4 + C_{DIM}$	n_2	/
register internal energy					
11	EDFFTRX1	2	only D changes	$2n_1 + n_4$	1.42
12	EDFFTRX1	2	only EN changes	$2C$	1.56
13	EDFFTRX1	2	only Q changes	$n_1 + n_2$	1.92

Table A.4: Column Selection module's Energy model Table for BNE.

#	gates	activity	capacitance (fF)	number	internal energy (fJ)
Base Case (common to all FS scenarios)					
1	AND2X1	1	0.6	n_9	
2	AND2X1	1	C_{DIM}	$n_5 - n_9$	
3	XOR2X1	1	0.2	$n_5 R_s - n_9$	
4	OR3X1	2	$0.4W_{\text{data}}$	R/W_{data}	
5	EDFFTRX1	1	$0.6(R_s + 1)$	n_5	/
Scenario 1: additions to Base					
6	AND2X1	1	0.2	$n_5 + n_9$	
7	OR2X1	1	0.2	$n_5 + 2n_9(\lfloor \log_2(R) \rfloor - 5)$	
8	OR2X1	1	C_{DIM}	$2n_9$	
9	OR2X1	1	$0.2R_s + C_{DIM}$	n_5	
Scenario 2: additions to Base					
10	AND2X1	1	0.2	$n_8 + n_{10}$	
11	OR2X1	1	0.2	$n_5 + n_9(\lfloor \log_2(R) \rfloor - 3)$ $+ n_{10}(\lfloor \log_2(R) \rfloor - 2)$	
12	OR2X1	1	C_{DIM}	$n_9 + n_{10}$	
13	OR2X1	1	$0.2R_s + C_{DIM}$	n_7	
14	OR3X1	1	$0.2R$	1	
Scenario 3: additions to Base					
15	AND2X1	1	0.2	$n_5 + n_7 + n_9$	
16	OR2X1	1	0.2	$n_5 + 2n_9(\lfloor \log_2(R) \rfloor - 5)$	
17	OR2X1	1	C_{DIM}	$2n_9$	
18	OR2X1	1	$0.2R_s + C_{DIM}$	n_6	
19	OR3X1	1	$0.2R$	1	
Scenario 4: additions to Base					
20	AND2X1	1	0.2	n_{10}	
21	OR2X1	1	0.2	$n_5 + 2n_9(\lfloor \log_2(R) \rfloor - 5)$	
22	OR2X1	1	C_{DIM}	$n_9 + n_{10}$	
register internal energy (common to all FS scenarios)					
23	EDFFTRX1	1	only D changes	$n_5(R_s + 1)$	1.42
24	EDFFTRX1	2	only EN changes	R	1.42
25	EDFFTRX1	1	only Q changes	n_5	1.92

Table A.5: Row Selection module's Energy model Table for FS.

#	gates	activity	capacitance (fF)	number	internal energy (fJ)
Base Case (common to all RDS scenarios)					
1	AND2X1	1	0.6	n_9	
2	AND2X1	1	C_{DIM}	$n_5 - n_9$	
3	AND2X1	2	0.2	1	
4	XOR2X1	1	0.2	$n_5 R_s - n_9$	
5	OR3X1	2	$0.4W_{\text{data}}$	1	
6	EDFFTRX1	1	$0.6(R_s + 1)$	n_5	/
Scenario 1: additions to Base					
7	AND2X1	1	0.2	$n_5 + n_9$	
8	OR2X1	1	0.2	$n_5 + 2n_9(\lfloor \log_2(R) \rfloor - 5)$	
9	OR2X1	1	C_{DIM}	$2n_9$	
10	OR2X1	1	$0.2R_s + C_{DIM}$	n_5	
11	OR2X1	2	$0.2R/W_{\text{data}}$	n_8	
Scenario 2: additions to Base					
12	AND2X1	1	0.2	n_{10}	
13	OR2X1	1	0.2	$n_5 + n_9(\lfloor \log_2(R) \rfloor - 3)$ $+ n_{10}(\lfloor \log_2(R) \rfloor - 2)$	
14	OR2X1	1	C_{DIM}	$n_9 + n_{10}$	
15	OR2X1	2	$0.2R/W_{\text{data}}$	n_5	
register internal energy (common to RDS all scenarios)					
16	EDFFTRX1	1	only D changes	$n_5(R_s + 1)$	1.42
17	EDFFTRX1	2	only D changes	$(n_5 R)/W_{\text{data}}$	1.42
18	EDFFTRX1	2	only EN changes	W_{data}	1.42
19	EDFFTRX1	1	only Q changes	n_5	1.92

Table A.6: Row Selection module's Energy model Table for RDS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	0.2	n_{10}	
2	AND2X1	1	0.6	n_9	
3	AND2X1	1	C_{DIM}	$R - n_9$	
4	OR2X1	1	0.2	$R + n_9(\lfloor \log_2(R) \rfloor - 3)$ $+ n_{10}(\lfloor \log_2(R) \rfloor - 2)$	
5	OR2X1	1	C_{DIM}	$n_9 + n_{10}$	
6	OR2X1	2	$0.2R/W_{\text{data}}$	W_{data}	
7	XOR2X1	1	0.2	$n_5 R_s - n_9$	
8	OR3X1	2	$0.4W_{\text{data}}$	R/W_{data}	
9	EDFFTRX1	1	$0.6(R_s + 1)$	n_5	/
register internal energy					
10	EDFFTRX1	1	only D changes	$R(R_s + 1)$	1.42
11	EDFFTRX1	2	only D changes	R	1.42
12	EDFFTRX1	2	only EN changes	R	1.56
13	EDFFTRX1	1	only Q changes	R	1.92

Table A.7: Row Selection module's Energy model Table for RDSs.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	0.2	n_6	
2	OR2X1	1	$0.2R_s + C_{DIM}$	n_6	

Table A.8: Row Selection module's Energy model Table for RDsh.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	C_{DIM}	1	
2	AND2X1	2	$0.4R$	1	
3	OR2X1	2	0.2	1	
4	EDFFTRX1	1	0.4	1	/
register internal energy					
5	EDFFTRX1	2	only EN changes	1	1.56
6	EDFFTRX1	1	only Q changes	1	1.92

Table A.9: Row Selection module's Energy model Table for BNE.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	0.2	n_{13}	
2	AND2X1	1	$0.2C/N_{\mathrm{ADC}}$	n_{12}	
3	AND2X1	2	$0.2C$	1	
4	OR2X1	2	0.4	C	
5	EDFFTRX1	1	0.6	n_{13}	/
register internal energy					
6	EDFFTRX1	1	only D changes	$(n_{12}C)/N_{\mathrm{ADC}} + 2n_{13}$	1.42
7	EDFFTRX1	2	only EN changes	C	1.56
8	EDFFTRX1	1	only Q changes	n_{13}	1.92

Table A.10: Read Register's Energy model Table for DoS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	0.2	n_{13}	
2	AND2X1	1	$0.2C/N_{\mathrm{ADC}}$	n_{12}	
3	OR2X1	2	0.4	n_{10}	
4	AND3X1	2	0.2	n_{10}	
5	EDFFTRX1	1	0.6	n_{13}	/
register internal energy					
6	EDFFTRX1	1	only D changes	$(n_{12}C)/N_{\mathrm{ADC}} + 2n_{13}$	1.42
7	EDFFTRX1	2	only EN changes	n_{10}	1.56
8	EDFFTRX1	1	only Q changes	n_{13}	1.92

Table A.11: Read Register's Energy model Table for CSR.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR2X1	2	0.4	1	
2	EDFFTRX1	1	1.4	n_{18}	/
register internal energy					
3	EDFFTRX1	2	only EN changes	1	1.56
4	EDFFTRX1	1	only Q changes	n_{18}	1.92

Table A.12: Verification Unit's Energy model Table for FS (store).

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	0.4	$n_{14} + n_{16}$	
2	OR2X1	1	0.2	n_{17}	
3	XOR2X1	1	0.2	n_{15}	

Table A.13: Verification Unit's Energy model Table for DoS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	0.4	n_{16}	
2	OR2X1	1	0.2	n_{17}	
3	XOR2X1	1	0.2	n_{15}	

Table A.14: Verification Unit's Energy model Table for CSR.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	2	0.2	1	
2	AND2X1	2	2.6	1	
3	OR2X1	2	0.4	1	
4	EDFFTRX1	1	1.4	1	/
register internal energy					
5	EDFFTRX1	2	only D changes	1	1.42
6	EDFFTRX1	2	only EN changes	1	1.56
7	EDFFTRX1	1	only Q changes	1	1.92

Table A.15: Verification Unit's Energy model Table for BNE.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	0.2	n_{20}	
2	AND2X1	2	$0.2C$	1	
3	OR2X1	2	0.4	C	
4	EDFFTRX1	1	0.2	n_2	/
register internal energy					
5	EDFFTRX1	2	only D changes	$2n_{20}$	1.42
6	EDFFTRX1	2	only EN changes	C	1.56
7	EDFFTRX1	1	only Q changes	n_2	1.92

Table A.16: Logical Unit's Energy model Table for DoS.

#	gates	activity	capacitance (fF)	number	internal energy (fJ)
Base Case (common to all CSR scenarios)					
1	AND2X1	1	0.2	n_{20}	
2	OR2X1	2	0.4	n_{11}	
3	AND3X1	2	1	n_{11}	
4	OR3X1	2	0.2	n_{20}	
5	EDFFTRX1	1	$0.6(R_s + 1)$	n_5	/
Scenario 1: additions to Base					
6	AND3X1	2	0.2	n_{20}	
Scenario 2: additions to Base					
7	OR2X1	2	0.2	n_{20}	
8	AND3X1	2	0.2	n_{11}	
register internal energy (common to all CSR scenarios)					
9	EDFFTRX1	2	only D changes	$n_{11} + 2n_{20}$	1.42
10	EDFFTRX1	2	only EN changes	n_{20}	1.42
11	EDFFTRX1	1	only Q changes	n_{20}	1.92

Table A.17: Logical Unit's Energy model Table for CSR.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{21}	
2	AND2X1	2	$0.4C + 1$	1	
3	AND4X1	2	0.4	1	
4	ADDFX1	1	0.2	n_{26}	
5	ADDFX1	1	0.6	n_{27}	
6	MX2X1	1	0.2	n_{26}	
7	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
register internal energy					
8	EDFFTRX1	1	only D changes	$n_{26} + n_{28}$	1.42
9	EDFFTRX1	1	only Q changes	n_{28}	1.92

Table A.18: Decoder1's Energy model Table for WDsh.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{17}	
2	AND2X1	1	$0.6\lceil log_2(C/W_{\text{data}})\rceil$	n_{22}	
3	AND2X1	1	$0.2C/W_{\text{data}}$	n_3	
4	AND2X1	2	$0.2C/W_{\text{data}}$	1	
5	AND4X1	2	$0.2(C/W_{\text{data}} + W_{\text{data}}) + 0.4$	1	
6	ADDFX1	1	0.2	n_{26}	
7	ADDFX1	1	0.6	n_{27}	
8	MX2X1	1	0.2	n_{26}	
9	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
register internal energy					
10	EDFFTRX1	1	only D changes	$n_{26} + n_{28}$	1.42
11	EDFFTRX1	1	only Q changes	n_{28}	1.92

Table A.19: Decoder1's Energy model Table for WDS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{17}	
2	AND2X1	2	$0.2C + 0.2$	1	
3	AND4X1	2	0.2	1	
4	ADDFX1	1	0.2	n_{26}	
5	ADDFX1	1	0.6	n_{27}	
6	MX2X1	1	0.2	n_{26}	
7	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
register internal energy					
8	EDFFTRX1	1	only D changes	$n_{26} + n_{28}$	1.42
9	EDFFTRX1	1	only Q changes	n_{28}	1.92

Table A.20: Decoder1's Energy model Table for WDSs.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{17}	
2	AND2X1	2	$0.4R + 0.6W_{\text{operand}} + 0.2$	1	
3	AND4X1	2	0.4	1	
4	ADDFX1	1	0.2	n_{26}	
5	ADDFX1	1	0.6	n_{27}	
6	MX2X1	1	0.2	n_{26}	
7	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
register internal energy					
8	EDFFTRX1	1	only D changes	$n_{26} + n_{28}$	1.42
9	EDFFTRX1	1	only Q changes	n_{28}	1.92

Table A.21: Decoder1's Energy model Table for RDsh.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{17}	
2	AND2X1	1	0.2	n_8	
3	AND2X1	1	$0.6\lceil log_2(R/W_{\text{data}})\rceil$	n_{23}	
4	AND2X1	2	$0.2R/W_{\text{data}}$	1	
5	AND4X1	2	$0.2(R/W_{\text{data}} + W_{\text{data}}) + 0.4$	1	
6	ADDFX1	1	0.2	n_{26}	
7	ADDFX1	1	0.6	n_{27}	
8	MX2X1	1	0.2	n_{26}	
9	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
register internal energy					
10	EDFFTRX1	1	only D changes	$n_{26} + n_{28}$	1.42
11	EDFFTRX1	1	only Q changes	n_{28}	1.92

Table A.22: Decoder1's Energy model Table for RDS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{17}	
2	AND2X1	2	$0.2(R/W_{\text{data}} + W_{\text{data}})$	1	
3	AND4X1	2	0.2	1	
4	ADDFX1	1	0.2	n_{26}	
5	ADDFX1	1	0.6	n_{27}	
6	MX2X1	1	0.2	n_{26}	
7	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
register internal energy					
8	EDFFTRX1	1	only D changes	$n_{26} + n_{28}$	1.42
9	EDFFTRX1	1	only Q changes	n_{28}	1.92

Table A.23: Decoder1's Energy model Table for RDSs.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{17}	
2	AND2X1	2	0.6	n_{24}	
3	AND2X1	2	1.4	1	
4	AND4X1	2	1.0	1	
5	ADDFX1	1	0.2	n_{26}	
6	ADDFX1	1	0.6	n_{27}	
7	MX2X1	1	0.2	n_{26}	
8	EDFFTRX1	1	1.0	n_{25}	/
9	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
register internal energy					
10	EDFFTRX1	1	only D changes	$n_{25} + n_{26} + n_{28}$	1.42
11	EDFFTRX1	2	only D changes	n_{25}	1.42
12	EDFFTRX1	2	only EN changes	4	1.56
13	EDFFTRX1	1	only Q changes	$n_{25} + n_{28}$	1.92

Table A.24: Decoder1's Energy model Table for FS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
Base Case (common to all DoA scenarios)					
1	INVX1	1	0.8	n_{17}	
2	AND2X1	2	$2.4 + (R + C)C_0$	1	
3	AND4X1	2	0.8	1	
4	ADDFX1	1	0.2	n_{26}	
5	ADDFX1	1	0.6	n_{27}	
6	MX2X1	1	0.2	n_{26}	
7	EDFFTRX1	1	1.0	n_{25}	/
8	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
Scenario 1: additions to Base					
9	AND2X1	2	$0.4\lceil log_2(W_{\mathrm{PC1}})\rceil$	1	
10	EDFFTRX1	1	0.4	n_{29}	/
register internal energy (common to all DoA scenarios)					
11	EDFFTRX1	1	only D changes	$n_{25} + n_{26} + n_{28}$	1.42
12	EDFFTRX1	2	only EN changes	4	1.56
13	EDFFTRX1	1	only Q changes	$n_{25} + n_{28}$	1.92
Scenario 1: additions to Base					
14	EDFFTRX1	2	only EN changes	$\lceil log_2(W_{\mathrm{PC1}})\rceil$	1.42
15	EDFFTRX1	1	only Q changes	n_{29}	1.92

Table A.25: Decoder1's Energy model Table for DoA.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR2X1	2	0.2	2	
2	OR2X1	2	1.6	1	
3	OR2X1	2	$0.4\lceil log_2(W_{\mathrm{PC1}})\rceil$	1	
4	ADDFX1	1	0.2	n_{26}	
5	ADDFX1	1	0.6	n_{27}	
6	MX2X1	1	0.2	n_{30}	
7	EDFFTRX1	1	1.0	2	/
8	EDFFTRX1	1	$C_{memory} + 1.0$	n_{28}	/
register internal energy					
9	EDFFTRX1	1	only D changes	$n_{26} + n_{28} + n_{30} + 2$	1.42
10	EDFFTRX1	2	only D changes	2	1.42
11	EDFFTRX1	2	only EN changes	$\lceil log_2(W_{\mathrm{PC1}})\rceil + 4$	1.56
12	EDFFTRX1	1	only Q changes	$n_{28} + 2$	1.92

Table A.26: Decoder1's Energy model Table for BNE.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
Base Case (common to all DoS scenarios)					
1	INVX1	1	0.8	n_{31}	
2	AND2X1	1	0.2	n_{38}	
3	AND2X1	1	$0.8N_{\mathrm{ADC}}$	1	
4	AND2X1	2	$3.8 + C_{SH}$	1	
5	OR2X1	2	0.4	1	
6	AND4X1	2	0.8	1	
7	OR4X1	1	0.2	n_{38}	
8	ADDFX1	1	0.4	n_{38}	
9	ADDFX1	1	0.6	n_{39}	
10	EDFFTRX1	1	1.2	n_{25}	/
11	EDFFTRX1	1	$1.2 + C_{memory}$	n_{36}	/
Scenario 1: additions to Base					
12	AND2X1	2	$0.4\lceil log_2(W_{\mathrm{PC2}})\rceil$	1	
13	EDFFTRX1	1	0.2	n_{45}	/
register internal energy (common to all DoS scenarios)					
14	EDFFTRX1	1	only D changes	$n_{36} + 2n_{38}$	1.42
15	EDFFTRX1	2	only EN changes	5	1.56
16	EDFFTRX1	1	only Q changes	$n_{25} + n_{36}$	1.92
Scenario 1: additions to Base					
17	EDFFTRX1	2	only EN changes	$\lceil log_2(W_{\mathrm{PC2}})\rceil$	1.56
18	EDFFTRX1	1	only Q changes	n_{45}	1.92

Table A.27: Decoder2's Energy model Table for DoS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{31}	
2	AND2X1	1	0.2	n_{38}	
3	AND2X1	1	0.2	n_{32}	
4	AND2X1	1	$C_{ADC} + 0.2$	n_{33}	
5	AND2X1	2	$0.4(N_{\mathrm{ADC}} + log_2(C/N_{\mathrm{ADC}} + 1)) + 2C_{ADC}$	1	
6	AND4X1	2	$0.2(N_{\mathrm{ADC}} + log_2(C/N_{\mathrm{ADC}} + 1))$	1	
7	OR4X1	1	0.2	n_{38}	
8	ADDFX1	1	0.4	n_{38}	
9	ADDFX1	1	0.6	n_{39}	
10	EDFFTRX1	1	$1.2 + C_{memory}$	n_{36}	/
register internal energy					
11	EDFFTRX1	1	only D changes	$n_{36} + 2n_{38}$	1.42
12	EDFFTRX1	1	only Q changes	n_{36}	1.92

Table A.28: Decoder2's Energy model Table for CSR.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{31}	
2	AND2X1	1	0.2	n_{38}	
3	OR2X1	2	0.4	1	
4	AND4X1	2	0.4	1	
5	OR4X1	1	0.2	n_{38}	
6	ADDFX1	1	0.4	n_{38}	
7	ADDFX1	1	0.6	n_{39}	
8	EDFFTRX1	2	$0.2N_{\mathrm{ADC}}$	1	/
9	EDFFTRX1	1	$1.2 + C_{memory}$	n_{36}	/
register internal energy					
10	EDFFTRX1	1	only D changes	$n_{36} + 2n_{38}$	1.42
11	EDFFTRX1	2	only D changes	1	1.42
12	EDFFTRX1	2	only EN changes	1	1.56
13	EDFFTRX1	1	only Q changes	n_{36}	1.92
14	EDFFTRX1	2	only Q changes	1	1.92

Table A.29: Decoder2's Energy model Table for LS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{31}	
2	AND2X1	1	0.2	n_{38}	
3	AND2X1	2	$0.8 + 0.2C/W_{\mathrm{operand}}$	1	
4	AND4X1	2	0.2	1	
5	OR4X1	1	0.2	n_{38}	
6	ADDFX1	1	0.4	n_{38}	
7	ADDFX1	1	0.6	n_{39}	
8	EDFFTRX1	1	$1.2 + C_{memory}$	n_{36}	/
register internal energy					
9	EDFFTRX1	1	only D changes	$n_{36} + 2n_{38}$	1.42
10	EDFFTRX1	1	only Q changes	n_{36}	1.92

Table A.30: Decoder2's Energy model Table for IADD.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
Base Case (common to all BNE scenarios)					
1	INVX1	1	0.8	n_{31}	
2	AND2X1	2	0.4	1	
3	AND4X1	2	0.2	1	
4	ADDFX1	1	0.4	n_{38}	
5	ADDFX1	1	0.6	n_{39}	
6	EDFFTRX1	1	$1.2 + C_{memory}$	n_{36}	/
Scenario 1: additions to Base					
7	AND2X1	1	0.2	n_{38}	
8	OR4X1	1	0.2	n_{38}	
Scenario 2: additions to Base					
9	AND2X1	1	0.2	$n_{40} + n_{46}$	
10	NOR3X1	2	$0.2\lceil log_2 W_{\mathrm{PC2}} \rceil$	1	
11	OR4X1	1	0.2	n_{35}	
register internal energy (common to all BNE scenarios)					
12	EDFFTRX1	1	only D changes	$n_{36} + n_{38}$	1.42
13	EDFFTRX1	1	only Q changes	n_{36}	1.92
Scenario 1: additions to Base					
14	EDFFTRX1	1	only D changes	n_{38}	1.42
Scenario 2: additions to Base					
15	EDFFTRX1	1	only D changes	n_{35}	1.42

Table A.31: Decoder2's Energy model Table for BNE.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{31}	
2	INVX1	2	0.2	n_{34}	
3	AND2X1	1	0.2	$n_{40} + n_{41}$	
4	AND2X1	2	0.4	n_{34}	
5	AND2X1	2	0.8	$n_{34} + n_{37}$	
6	OR2X1	2	0.4	1	
7	NOR3X1	2	$0.2W_{\mathrm{PC2}}$	1	
8	AND4X1	2	$0.8 + W_{\mathrm{PC2}}$	1	
9	OR4X1	1	0.2	n_{35}	
10	ADDFX1	1	0.2	n_{41}	
11	ADDFX1	1	0.4	n_{38}	
12	ADDFX1	1	0.6	$n_{39} + n_{42}$	
13	EDFFTRX1	1	0.2	$n_{45} + 1$	/
14	EDFFTRX1	1	$1.2 + C_{memory}$	n_{36}	/
register internal energy					
15	EDFFTRX1	1	only D changes	$n_{35} + n_{38}$	1.42
16	EDFFTRX1	2	only D changes	1	1.42
17	EDFFTRX1	2	only EN changes	$1 + W_{\mathrm{PC2}}$	1.56
18	EDFFTRX1	1	only Q changes	$n_{36} + n_{45} + 1$	1.92

Table A.32: Decoder2's Energy model Table for JAL.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{31}	
2	AND2X1	1	0.2	n_{40}	
3	AND2X1	2	0.2	n_{44}	
4	AND2X1	2	$0.4 + 0.2W_{\mathrm{PC2}}$	1	
5	OR2X1	2	0.4	1	
6	NOR3X1	2	$0.2W_{\mathrm{PC2}}$	1	
7	AND4X1	2	0.4	1	
8	OR4X1	1	0.2	n_{35}	
9	ADDFX1	1	0.4	n_{38}	
10	ADDFX1	1	0.6	n_{39}	
11	EDFFTRX1	1	0.2	1	/
12	EDFFTRX1	1	$1.2 + C_{memory}$	n_{36}	/
register internal energy					
13	EDFFTRX1	1	only D changes	$n_{35} + n_{38}$	1.42
14	EDFFTRX1	2	only EN changes	1	1.56
15	EDFFTRX1	1	only Q changes	$n_{36} + 1$	1.92

Table A.33: Decoder2's Energy model Table for JR.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	INVX1	1	0.8	n_{31}	
2	AND2X1	1	0.2	n_{38}	
3	AND2X1	2	0.4	1	
4	AND2X1	2	$0.4 + 0.2C/W_{\mathrm{operand}}$	1	
5	AND4X1	2	0.6	1	
6	OR4X1	1	0.2	n_{38}	
7	ADDFX1	1	0.4	n_{38}	
8	ADDFX1	1	0.6	n_{39}	
9	EDFFTRX1	1	$1.2 + C_{memory}$	n_{36}	/
register internal energy					
10	EDFFTRX1	1	only D changes	$n_{35} + n_{38}$	1.42
11	EDFFTRX1	1	only Q changes	$n_{36} + 1$	1.92

Table A.34: Decoder2's Energy model Table for CP.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
Base Case (common to all DoA scenarios)					
1	DFFRX1	2	0.2	T_{crossbar}	/
Scenario 1: additions to Base					
2	AND2X1	2	0.2	3	
3	OR2X1	2	0.2	1	
4	OR2X1	4	0.2	1	
5	OR4X1	2	0.4	1	
6	EDFFTRX1	2	0.4	1	/
Scenario 2: additions to Base					
7	AND2X1	2	0.4	1	
8	OR2X1	2	0.4	1	
9	OR4X1	1	0.4	1	
10	EDFFTRX1	1	0.2	1	/
Scenario 3: additions to Base					
11	AND2X1	2	0.4	2	
12	OR2X1	2	0.4	2	
13	EDFFTRX1	1	0.2	2	/
register internal energy (common to all DoA scenarios)					
14	DFFRX1	2	only D changes	T_{crossbar}	1.23
15	DFFRX1	2	only Q changes	T_{crossbar}	1.64
Scenario 1: additions to Base					
16	EDFFTRX1	2	only D changes	1	1.42
17	EDFFTRX1	4	only EN changes	1	1.56
18	EDFFTRX1	2	only Q changes	1	1.92
Scenario 2: additions to Base					
19	EDFFTRX1	2	only D changes	1	1.42
20	EDFFTRX1	2	only EN changes	1	1.56
21	EDFFTRX1	1	only Q changes	1	1.92
Scenario 3: additions to Base					
22	EDFFTRX1	2	only D changes	2	1.42
23	EDFFTRX1	2	only EN changes	2	1.56
24	EDFFTRX1	1	only Q changes	2	1.92

Table A.35: Stall Detection's Energy model Table for DoA.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR2X1	2	0.4	2	
2	OR2X1	4	0.4	1	
3	OR4X1	4	0.4	1	
4	DFFRX1	2	0.2	T_{SH}	/
5	EDFFTRX1	1	0.2	2	/
6	EDFFTRX1	2	0.4	1	/
register internal energy					
7	DFFRX1	2	only D changes	T_{SH}	1.23
8	DFFRX1	2	only Q changes	T_{SH}	1.64
9	EDFFTRX1	2	only D changes	1	1.42
10	EDFFTRX1	2	only EN changes	2	1.56
11	EDFFTRX1	4	only EN changes	1	1.56
12	EDFFTRX1	1	only Q changes	2	1.92
13	EDFFTRX1	2	only Q changes	1	1.92

Table A.36: Stall Detection's Energy model Table for DoS.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR2X1	4	0.4	1	
2	OR4X1	2	0.4	1	
3	DFFRX1	2	0.2	$T_{\mathrm{ADC}} - 1$	/
4	DFFRX1	2	$0.2N_{\mathrm{ADC}}$	1	/
5	EDFFTRX1	2	0.4	1	/
register internal energy					
6	DFFRX1	2	only D changes	T_{ADC}	1.23
7	DFFRX1	2	only Q changes	T_{ADC}	1.64
8	EDFFTRX1	2	only D changes	1	1.42
9	EDFFTRX1	4	only EN changes	1	1.56
10	EDFFTRX1	2	only Q changes	1	1.92

Table A.37: Stall Detection's Energy model Table for CSR.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR2X1	2	0.4	1	
2	EDFFTRX1	1	0.2	1	/
register internal energy					
3	EDFFTRX1	2	only EN changes	1	1.56
4	EDFFTRX1	1	only Q changes	1	1.92

Table A.38: Stall Detection's Energy model Table for BNE.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
Base Case (common to all CSR scenarios)					
1	AND2X1	2	0.2	n_{49}	
2	OR2X1	2	0.2	$n_{49}(C/N_{\text{ADC}} - n_{48} - 1)$	
3	OR2X1	2	0.8	n_{49}	
4	AND3X1	2	0.8	n_{47}	
5	AND3X1	2	$0.6\lceil log_2(R+1)\rceil$	1	
6	ADDFX1	1	0.4		
7	ADDFX1	1	0.6		
Scenario 1: additions to Base					
8	AND2X1	2	$0.2C/N_{\text{ADC}}$	n_{51}	
9	ADDFX1	1	0.4	n_{51}	
10	ADDFX1	1	0.6	n_{52}	
11	EDFFTRX1	1	0.2	n_{50}	/
Scenario 2: additions to Base					
12	AND2X1	1	0.2	n_{55}	
13	AND2X1	2	$0.2 + 0.4W_{\text{operand}}$	1	
14	OR2X1	2	$0.8\lceil log_2(R+1)\rceil$	1	
15	ADDFX1	1	0.4	$n_{51} + n_{55}$	
16	ADDFX1	1	0.6	$n_{52} + n_{56}$	
17	EDFFTRX1	1	0.2	n_{51}	/
18	EDFFTRX1	1	0.4	n_{57}	/
19	EDFFTRX1	1	0.8	$n_{53} + n_{54}$	/
register internal energy (common to all CSR scenarios)					
20	EDFFTRX1	2	only EN changes	$\lceil log_2(R+1)\rceil$	1.56
Scenario 1: additions to Base					
21	EDFFTRX1	2	only D changes	$n_{51}C/N_{\text{ADC}}$	1.42
22	EDFFTRX1	1	only Q changes	n_{50}	1.92
Scenario 2: additions to Base					
23	EDFFTRX1	1	only D changes	$n_{51} + n_{55} + n_{57}$	1.42
24	EDFFTRX1	2	only EN changes	$3\lceil log_2(R+1)\rceil + W_{\text{operand}}$	1.56
25	EDFFTRX1	1	only Q changes	$n_{50} + n_{53} + n_{54} + n_{57}$	1.92

Table A.39: Addition Unit's Energy model Table for CSR.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	1	0.2	$n_{55}+2$	
2	AND2X1	2	0.2	n_{59}	
3	AND2X1	2	0.8	n_{58}	
4	AND2X1	2	$0.2(\lceil log_2(R+1)\rceil + W_{\text{operand}}) + 0.2C/N_{\text{ADC}} + 0.2$	2	
5	OR2X1	2	0.8	n_{59}	
6	OR2X1	2	$0.8\lceil log_2(R+1)\rceil$	1	
7	OR2X1	2	$0.4C/(N_{\text{ADC}}W_{\text{operand}})$	1	
8	OR2X1	2	$0.4(\lceil log_2(R+1)\rceil + W_{\text{operand}})$	1	
9	ADDFX1	1	0.2	n_{60}	
10	ADDFX1	1	0.4	n_{55}	
11	ADDFX1	1	0.6	$n_{56}+n_{61}$	
12	EDFFTRX1	1	0.2	2	/
13	EDFFTRX1	1	0.8	$n_{53}+n_{54}$	/
14	EDFFTRX1	1	$0.2(C+1)$	n_{62}	/
register internal energy					
15	EDFFTRX1	1	only D changes	$n_{55}+n_{57}+n_{60}+ 2n_{62}+2$	1.42
16	EDFFTRX1	2	only EN changes	$3\lceil log_2(R+1)\rceil + 2W_{\text{operand}}$	1.56
17	EDFFTRX1	1	only Q changes	$n_{53}+n_{54}+n_{62}+2$	1.92

Table A.40: Addition Unit's Energy model Table for IADD.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	OR2X1	2	$0.4\lceil log_2(R+1)\rceil + 0.4W_{\text{operand}}$	$C/(N_{\text{ADC}}W_{\text{operand}})$	
2	EDFFTRX1	1	0.6	n_{59}	/
register internal energy					
3	EDFFTRX1	2	only EN changes	$(\lceil log_2(R+1)\rceil + W_{\text{operand}}) * C/(N_{\text{ADC}}W_{\text{operand}})$	1.56
4	EDFFTRX1	1	only Q changes	n_{59}	1.92

Table A.41: Addition Unit's Energy model Table for CP.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
Base Case (common to all writing scenarios)					
1	AND3X1	2	$0.4W_{\text{bus}}$	1	
2	EDFFTRX1	1	0.2	n_{64}	/
Scenario 1: additions to Base					
3	AND2X1	2	0.4	$n_{66}-1$	
4	AND2X1	2	0.6	1	
5	EDFFTRX1	1	1	$n_{66}-1$	/
6	EDFFTRX1	1	$0.6+0.4C/W_{\text{bus}}$	1	/
register internal energy (common to all writing scenarios)					
7	EDFFTRX1	1	only D changes	$2n_{63}C/W_{\text{bus}}$	1.42
8	EDFFTRX1	2	only EN changes	W_{bus}	1.56
9	EDFFTRX1	1	only Q changes	n_{64}	1.92
Scenario 1: additions to Base					
10	EDFFTRX1	1	only D changes	$2n_{63}C/W_{\text{bus}}+n_{66}$	1.42
11	EDFFTRX1	2	only EN changes	$W_{\text{bus}}+n_{66}$	1.56
12	EDFFTRX1	1	only Q changes	$n_{64}+n_{66}$	1.92

Table A.42: Write Data Buffer's Energy model Table for writing.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	2	0.2	n_{14}	
2	AND2X1	2	0.4	$n_{67}-1$	
3	AND2X1	2	$0.2C$	1	
4	OR2X1	1	0.2	n_{65}	
5	EDFFTRX1	1	1.2	n_{67}	/
6	EDFFTRX1	1	$C_{DIM}+0.2$	n_{65}	/
register internal energy					
7	EDFFTRX1	1	only D changes	$n_{65}+n_{67}$	1.42
8	EDFFTRX1	2	only EN changes	$C+n_{67}$	1.56
9	EDFFTRX1	1	only Q changes	$n_{65}+n_{67}$	1.92

Table A.43: Write Data Buffer's Energy model Table for WDsh.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
Base Case (common to all writing scenarios)					
1	AND3X1	2	$0.4W_{\text{bus}}$	1	
2	EDFFTRX1	1	0.2	n_{69}	/
Scenario 1: additions to Base					
3	INVX1	2	0.2	1	
4	AND2X1	2	$0.4W_{\text{operand}} + 0.2$	1	
5	EDFFTRX1	1	$0.2R/W_{\text{bus}} + 0.4$	1	/
Scenario 2: additions to Base					
6	AND2X1	1	0.2	1	
7	OR2X1	1	0.2	1	
8	DFFSHQX1	1	$0.2R/W_{\text{bus}} + 0.6$	1	/
Scenario 3: additions to Base					
9	AND2X1	2	0.6	1	
10	OR2X1	2	0.2	1	
11	DFFSHQX1	1	$0.2R/W_{\text{bus}} + 0.6$	1	/
12	EDFFTRX1	1	0.6	1	/
register internal energy (common to all writing scenarios)					
13	EDFFTRX1	1	only D changes	$(n_{68}W_{\text{operand}}R)/W_{\text{bus}}$	1.42
14	EDFFTRX1	2	only EN changes	W_{bus}	1.56
15	EDFFTRX1	1	only Q changes	n_{69}	1.92
Scenario 1: additions to Base					
16	EDFFTRX1	1	only D changes	1	1.42
17	EDFFTRX1	2	only EN changes	1	1.56
18	EDFFTRX1	1	only Q changes	1	1.92
Scenario 2: additions to Base					
19	DFFSHQX1	1	only D changes	1	1.34
20	DFFSHQX1	1	only Q changes	1	1.76
Scenario 3: additions to Base					
21	DFFSHQX1	1	only D changes	1	1.34
22	DFFSHQX1	1	only Q changes	1	1.76
23	EDFFTRX1	1	only D changes	1	1.42
24	EDFFTRX1	2	only EN changes	1	1.56
25	EDFFTRX1	1	only Q changes	1	1.92

Table A.44: Row Data Buffer's Energy model Table for writing.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
Base Case (common to all writing scenarios)					
1	INVX1	2	0.2	1	
2	AND2X1	2	$0.2R$	1	
3	AND2X1	2	0.2	n_7	
4	OR2X1	1	0.2	n_{70}	
5	EDFFTRX1	1	0.6	1	/
6	EDFFTRX1	1	$0.2R_{\mathrm{s}} + 0.2$	n_{70}	/
Scenario 1: additions to Base					
7	AND2X1	1	0.2	1	
8	OR2X1	1	0.2	1	
9	DFFSHQX1	1	0.8	1	/
Scenario 2: additions to Base					
10	AND2X1	2	0.6	1	
11	OR2X1	2	0.2	1	
12	DFFSHQX1	1	0.8	1	/
13	EDFFTRX1	1	0.6	1	/
register internal energy (common to all writing scenarios)					
14	EDFFTRX1	1	only D changes	$n_{70} + 1$	1.42
15	EDFFTRX1	2	only EN changes	$R + 1$	1.56
16	EDFFTRX1	1	only Q changes	$n_{70} + 1$	1.92
Scenario 1: additions to Base					
17	DFFSHQX1	1	only D changes	1	1.34
18	DFFSHQX1	1	only Q changes	1	1.76
Scenario 2: additions to Base					
19	DFFSHQX1	1	only D changes	1	1.34
20	DFFSHQX1	1	only Q changes	1	1.76
21	EDFFTRX1	1	only D changes	1	1.42
22	EDFFTRX1	2	only EN changes	1	1.56
23	EDFFTRX1	1	only Q changes	1	1.92

Table A.45: Row Data Buffer's Energy model Table for RDsh.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	2	0.4	$n_{74} - 1$	
2	AND2X1	2	1	1	
3	AND2X1	2	$0.4C$	1	
4	EDFFTRX1	1	0.2	n_{71}	/
5	EDFFTRX1	1	1	$n_{74} - 1$	/
6	EDFFTRX1	1	1.2	1	/
register internal energy					
7	EDFFTRX1	1	only D changes	n_{74}	1.42
8	EDFFTRX1	2	only EN changes	$n_{74} + C$	1.56
9	EDFFTRX1	1	only Q changes	$n_{71} + n_{74}$	1.92

Table A.46: Read/Logic Output Buffer's Energy model Table for CP.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	2	0.2	n_{72}	
2	AND2X1	2	0.4	$n_{75}-1$	
3	AND2X1	2	$0.2C$	1	
4	EDFFTRX1	1	1	$n_{75}-1$	/
5	EDFFTRX1	1	1.2	1	/
6	EDFFTRX1	1	$C_{outside}$	n_{73}	/
register internal energy					
7	EDFFTRX1	1	only D changes	n_{75}	1.42
8	EDFFTRX1	2	only D changes	n_{73}	1.42
9	EDFFTRX1	2	only EN changes	$n_{75}+C$	1.42
10	EDFFTRX1	1	only Q changes	$n_{73}+n_{75}$	1.92

Table A.47: Read/Logic Output Buffer's Energy model Table for reading.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	2	0.4	$n_{79}-1$	
3	AND2X1	2	$0.4C$	1	
4	EDFFTRX1	1	0.2	n_{76}	/
5	EDFFTRX1	1	1	$n_{79}-1$	/
6	EDFFTRX1	1	1.2	1	/
register internal energy					
7	EDFFTRX1	1	only D changes	n_{79}	1.42
8	EDFFTRX1	2	only EN changes	$n_{79}+2C$ $+C\lceil log_2(R+1)\rceil)/W_{\text{operand}}$	1.56
9	EDFFTRX1	1	only Q changes	$n_{76}+n_{79}$	1.92

Table A.48: Addition Output Buffer's Energy model Table for CP.

#	gates	activity	capacitance(fF)	number	internal energy(fJ)
1	AND2X1	2	0.2	n_{77}	
2	AND2X1	2	0.4	$n_{80}-1$	
3	AND2X1	2	$0.2C$	1	
4	EDFFTRX1	1	1	$n_{80}-1$	/
5	EDFFTRX1	1	1.2	1	/
6	EDFFTRX1	1	$C_{outside}$	n_{78}	/
register internal energy					
7	EDFFTRX1	1	only D changes	n_{80}	1.42
8	EDFFTRX1	2	only D changes	n_{78}	1.42
9	EDFFTRX1	2	only EN changes	$n_{80}+2C$ $+C\lceil log_2(R+1)\rceil)/W_{\text{operand}}$	1.42
10	EDFFTRX1	1	only Q changes	$n_{78}+n_{80}$	1.92

Table A.49: Addition Output Buffer's Energy model Table for reading.