Tiled Matrix Multiplication Unit: Core Hardware for Matrix Computation in Edge AI Systems

by

Chao-Jia Liu (Peter)
Student ID: hdj697

GRADUATION PROJECT REPORT
Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

COMMITTEE MEMBERS:
Dr. Eugene John, Ph.D.
Dr. Wei-Ming Lin, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
Klesse College of Engineering and Integrated Design
Department of Electrical and Computer Engineering
May 2025

ACKNOWLEDGEMENTS

First of all, I want to express my sincere gratitude to the Department of Electrical and Computer Engineering at the University of Texas at San Antonio (UTSA) for providing me with the invaluable opportunity to pursue my master's degree. I am especially thankful for the generous support I received through competitive scholarships, as well as teaching assistant and grader positions, which allowed me to fully focus on my studies and research.

I would like to extend my deepest appreciation to Dr. Eugene John for accepting me as his master's student in my first semester. His well-structured guidance and thoughtful course registration advice played a crucial role in shaping my academic journey. From helping me determine the direction of my graduation project to recommending courses that equipped me with the necessary skills, his support and mentorship were instrumental to my growth and success.

I am also grateful to Dr. Wei-Ming Lin for his insightful feedback on my TPU course project. His advice helped me refine the focus of my work and improve the clarity and persuasiveness of my writing. I especially remember how his suggestions motivated me to replace my hand-drawn figures with professionally crafted diagrams using Adobe Illustrator—marking a significant improvement in how I presented my ideas, even though those figures took a ton of time to complete.

A heartfelt thank-you goes to Dr. Joo Eun (June) Hong, who consistently acknowledged and appreciated my work as a TA and grader. I'm also thankful to the many professors at UTSA who supported me through coursework, projects, and job opportunities. I deeply appreciate everyone who guided me along the way.

I want to express special appreciation to Khanh Nguyen, the best program coordinator I could have asked for. I still remember visiting her office before my first semester even began. Whether my questions were academic or personal, she was always incredibly patient and willing to help however she could. Her kindness and encouragement truly eased my stress and helped me stay focused on my goals.

To Priyabrata Dash—thank you for always being generous with your time and knowledge. I'm especially grateful for that first four-hour conversation where you introduced me to the fundamentals of DNNs and AI, and later shared the idea of implementing a TPU as a research direction. Most importantly, thank you for always

making time to talk through any challenges I faced. Our discussions—whether at the JPL food court or by the Student Union pool table—helped me clarify my design and shape the future direction of my project. Although we didn't get to go skydiving together, I still hope you'll take me to India someday to visit the Taj Mahal!

To my friends at UTSA and in San Antonio—thank you for making this journey so much more memorable. A special shout-out to my Taiwanese group of friends, who were always the best company to unwind with. And Alex, my bro—thank you for always encouraging me to speak up, even when I wasn't confident in my English at first. You always gave me props for stepping out of my comfort zone and practicing with you whenever we hung out—never once losing patience. To Justin—thanks for all the support, from school advice and study tips to those life lessons that really made me start taking responsibility for myself. The cities we explored together were some of the best memories—helped me de-stress and see more of the country. Catch you again soon!

Most importantly, I want to thank my parents for their unconditional love and support—always given without expecting anything in return. I truly could not have completed this degree without you. To my dad—thank you for supporting me financially and reminding me to enjoy life without worrying too much. And to Uncle Vincent—you came to San Antonio on my second day here and supported me throughout these two years. I'll definitely visit you in Houston as often as I can!

To my mom—thank you for always answering my calls, no matter the time. Your emotional support and endless patience in listening to me helped me get back on track whenever I felt unmotivated. Love you, Mom!

To my sister, Celine—your constant encouragement and thoughtful advice pushed me to think more deeply and carefully before making decisions. You helped me gain clarity about my future and guided me to become a better version of myself. Thank you!

To everyone not mentioned by name—you all have my heartfelt thanks. And even to myself: this graduate journey has helped me become a more mature and capable person. Everything I experienced at UTSA will stay with me forever.

Abstract

Tiled Matrix Multiplication Unit: Core Hardware for Matrix Computation in Edge AI Systems

by

Chao-Jia Liu (Peter)
Master of Science in Electrical Engineering
The University of Texas at San Antonio, 2025

Supervising Professor: Eugene John, Ph.D.

Over the past decades, extensive research and development in artificial intelligence (AI) have led to the creation of various AI applications and hardware accelerators. Many leading companies have successfully developed custom AI chips tailored for different workloads, achieving significant breakthroughs in performance and efficiency. Among the key computational operations in AI, General Matrix Multiplication (GEMM) plays a critical role due to its widespread use in computing weighted sums of activations and weights in neural networks. Its mathematical structure and potential for parallel execution make GEMM a fundamental building block for implementing fully connected layers, convolutional operations, and attention mechanisms in many modern AI models.

However, as the focus of AI deployment shifts toward edge computing—including mobile devices and wearables—hardware resource constraints become more prominent due to reduced chip area, limited memory, and lower power budgets. Conventional GEMM implementations, which often involve intensive data movement and demand high memory bandwidth, encounter serious limitations in these constrained environments. To address these challenges, Tiled Matrix Multiplication (TMM) has emerged as an efficient strategy that decomposes large matrix operations into smaller tiles, significantly improving computational efficiency and hardware resource utilization.

This study presents the design, implementation, and evaluation of a Tiled Matrix Multiplication Unit (TMMU) designed to support AI workloads on resource-constrained edge devices. While similar units are already integrated into commercial AI accelerators, their architectural details are typically proprietary. This project explores and reconstructs a TMMU architecture using Verilog, offering a practical

perspective on its structure and operation. The unit supports two modes: basic matrix multiplication and tiled matrix multiplication, and operates using INT8 data format to align with the efficiency requirements of edge computing. Built upon a previously developed 4×4 Tensor Processing Unit (TPU), the design employs a custom CISC instruction set for control. With a 20 ns clock cycle, the TMMU completes a 4×4 matrix multiplication in 3,000 ns (150 clock cycles) and an 8×8 multiplication in approximately 17,000 ns (850 clock cycles).

In addition to the hardware implementation, the project includes simulation results, power estimation using Xilinx Vivado, and floorplanning analysis using Cadence Innovus to evaluate its feasibility for physical realization. By documenting the architectural design, control logic, simulation behavior, and performance characteristics, this work contributes a reference point for those interested in the practical aspects of tiled matrix multiplication in ASIC development for edge AI systems.

TABLE OF CONTENTS

List of Tables

List of Figures

CHAPTER 1

Introduction

In the evolving landscape of hardware acceleration for machine learning, matrix multiplication remains one of the most computationally intensive and power-hungry operations. As neural networks grow deeper and more complex, efficient matrix computation has become a cornerstone of performance optimization in both edge and cloud environments. To address these challenges, custom accelerators such as Tensor Processing Units (TPUs) have emerged, offering architectural innovations that balance parallelism, dataflow control, and memory hierarchy. This graduation project introduces the design and implementation of the Tiled Matrix Multiplication Unit (TMMU)—a hardware accelerator developed using Verilog RTL and designed to support both basic and tiled matrix multiplication through a systolic array architecture.

The chapters in this project reflect the layered approach taken to develop and evaluate the TMMU. From high-level architectural considerations and RTL module development to simulation, power analysis, and physical design floorplanning, each chapter captures a key stage in the system's creation. This progression offers a comprehensive look into how custom digital hardware can be designed, tested, and refined for efficient matrix operations, making the project valuable for students, researchers, and engineers engaged in hardware acceleration.

Chapter 2 lays the theoretical groundwork by explaining the design motivations behind systolic array-based matrix multiplication. It discusses the trade-offs of using tiled computation, data reordering strategies, and the rationale for separating control logic into Level 1 (TPU) and Level 2 (TMMU) layers.

Chapter 3 transitions into the control architecture, presenting detailed finite state machines (FSMs) that govern instruction decoding, data movement, and synchronization. It highlights how the dual-controller setup ensures flexible instruction handling and efficient timing control across pipeline stages.

Chapter 4 focuses on the structural and implementation aspects of the TMMU. It presents block diagrams and selected Verilog code excerpts for each component, including memory modules, FIFOs, data setup units, the MMU, accumulator, and

activation normalization units. By bridging RTL design with architectural intent, this chapter establishes the full hardware blueprint of the TMMU.

Chapter 5 evaluates the performance of the TMMU through cycle-accurate simulations, power analysis, and physical layout via Innovus floorplanning. It explores the functional correctness of the accelerator under various matrix sizes, reports on component-level power consumption, and identifies challenges like multi-driven nets and license-based limitations that affect physical synthesis.

The final section of the Chapter 5 outlines future work and potential enhancements. It details known limitations—such as redundant logic for small matrix sizes and multi-driver conflicts—and proposes architectural extensions, including support for additional activation functions, ASIC fabrication, and scaling the design to support larger matrix dimensions via multi-TMMU integration.

This project seeks to serve as a bridge between academic understanding and practical hardware design. It highlights not just the technical implementation of a matrix accelerator, but also the broader challenges in verification, synthesis, and physical design. Through simulations, power and layout assessments, and detailed documentation, this project offers a transparent and replicable path for others seeking to explore hardware acceleration using systolic arrays and tiled dataflow models.

By connecting foundational architectural principles with practical RTL development, the TMMU project provides a robust platform for experimenting with efficient matrix computation. It equips readers with both the methodology and motivation to advance the field of hardware-based AI acceleration—from experimental student projects to industry-grade chip design.

CHAPTER 2

Background

To better understand the motivation, design choices, and objectives behind this project, this chapter introduces key background topics and related technologies that have influenced the hardware architecture discussed in later chapters. These include the von Neumann architecture [1][2][3][5], systolic arrays [6][7], General Matrix Multiplication (GEMM) [8][9], and Google's Tensor Processing Unit (TPU) [12][13][14].

The chapter begins with an overview of the classic von Neumann architecture, a widely adopted computing model that remains prevalent in modern systems. However, as artificial intelligence (AI) applications continue to grow in complexity and scale, this architecture faces notable limitations—particularly in handling the parallel data movement and computational demands of deep learning workloads. These challenges have motivated the development of Application-Specific Integrated Circuits (ASICs) tailored for AI processing.

One of the foundational concepts in AI-specific hardware design is the systolic architecture, introduced by H. T. Kung in 1982 [6]. This model defines a structured approach to data movement through interconnected processing elements (PEs), enabling high throughput and efficient parallel computation. By synchronizing data flow across a grid of PEs, systolic arrays reduce memory access overhead and minimize latency—two critical factors in accelerating deep learning workloads.

Closely related to this architectural style is General Matrix Multiplication (GEMM), a core computational kernel that underlies many operations in neural networks. GEMM is extensively used in dense linear algebra tasks, including fully connected layers, convolution layers, and attention mechanisms in transformer models. Its regular structure and inherent parallelism make GEMM well-suited for hardware acceleration. In modern AI accelerators such as GPUs, TPUs, and FPGAs, a significant portion of design optimization focuses on GEMM, using strategies like tiling, data reuse, and low-precision formats to improve efficiency.

Finally, the chapter introduces Google's Tensor Processing Unit (TPU) [12], which serves as an important reference for this project. The original TPU paper outlines key

architectural principles and provides a block diagram that has guided many hardware implementations. These concepts collectively inform the design and implementation of the Tiled Matrix Multiplication Unit (TMMU) developed in this work.

2.1 Von Neumann Architecture

The von Neumann architecture, first proposed in the mid-1940s, has had a profound and lasting impact on the evolution of digital computing. The architecture defines a model in which a single memory is used to store both data and program instructions, which are processed sequentially by a central processing unit (CPU). This principle of a shared memory and unified control flow laid the foundation for nearly all general-purpose computers and remains a cornerstone of modern processor design.

Despite its historical significance and continued relevance, the von Neumann architecture presents increasing limitations when applied to data-intensive and parallel computing workloads, particularly in the field of artificial intelligence (AI). As AI models grow in scale and complexity, the architecture's inherent reliance on a centralized memory and serial processing introduces performance bottlenecks that hinder throughput and energy efficiency. This section introduces the basic structure of the von Neumann model and then explores its limitations in the context of AI, setting the stage for more advanced architectural solutions discussed in later chapters.

2.1.1 Introduction to the Von Neumann Architecture

After John von Neumann proposed his architecture in the *First Draft of a Report on the EDVAC* [1] in 1945, computer architecture design has largely been shaped by and optimized around this model. The von Neumann architecture became the foundation for most general-purpose computers and remains highly influential in modern CPU design.

At its core, the von Neumann model consists of a Central Processing Unit (CPU), which includes two key components: the Control Unit and the Arithmetic and Logic Unit (ALU). The Control Unit is responsible for fetching instructions from memory, decoding them, and managing their execution. The ALU, on the other hand, performs all arithmetic and logical operations. The CPU interacts with a shared memory unit, which stores both program instructions and data, as well as an input/output (I/O) interface for communication with external devices. Figure 2.1 illustrates a simplified block diagram of the von Neumann architecture [2]. Table 2.1 summarizes the key components in von Neumann architecture and their functionalities [3].

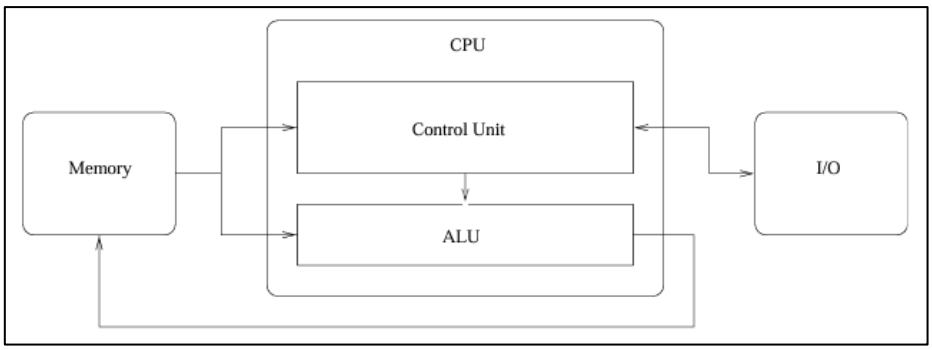


Figure 2.1: The Basic Components of a Computer with a Von Neumann Architecture. [2]

Component	Description
Arithmetic unit	The arithmetic unit can perform basic arithmetic operations
Control unit	The control unit contains a built-in set of machine instructions, and the program counter contains the address of the next instruction to be executed
	This instruction is fetched from memory and executed
	This is the basic fetch-and-execute cycle (Figure 2.2)
Input-output unit	The input and output unit allows the computer to interact with the outside world
Memory	There is a one-dimensional memory that stores all the program instructions and data. These are usually kept in different areas of memory
	The memory may be written to or read from, i.e., it is random-access memory (RAM)
	The program instructions are binary values, and the control unit decodes the binary value to determine the instruction to execute

Table 2.1: Components in Von Neumann Architecture and Their Descriptions [3]

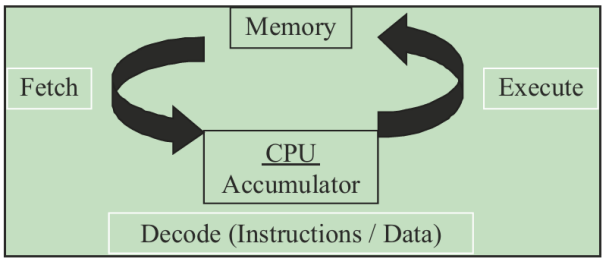


Figure 2.2: Fetch-and-execute Cycle [3]

One of the most distinctive characteristics of this architecture is that both data and instructions share the same memory space and are represented as binary values. During program execution, the control unit fetches values from memory without initially knowing whether the value represents an instruction or data. It then decodes and processes it accordingly. While this unified memory model simplifies hardware design and reduces cost, it also introduces potential drawbacks—such as the risk of unintended instruction execution if memory is corrupted or improperly managed.

To support various computational tasks, CPUs rely on a predefined set of instructions—either Reduced Instruction Set Computing (RISC) or Complex Instruction Set Computing (CISC). This collection of instructions, along with the hardware that implements them, forms what is known as the Instruction Set Architecture (ISA). Ideally, modern processors aim to execute one instruction per clock cycle. In practice, to enhance performance, CPUs employ techniques like pipelining and superscalar execution to handle multiple instructions simultaneously. Today's CPUs can reach clock speeds of around 3.2 GHz, meaning they can perform up to 3.2 billion cycles per second [4]. Earlier generations, by contrast, operated at much lower frequencies—typically measured in megahertz (MHz), or millions of cycles per second [2].

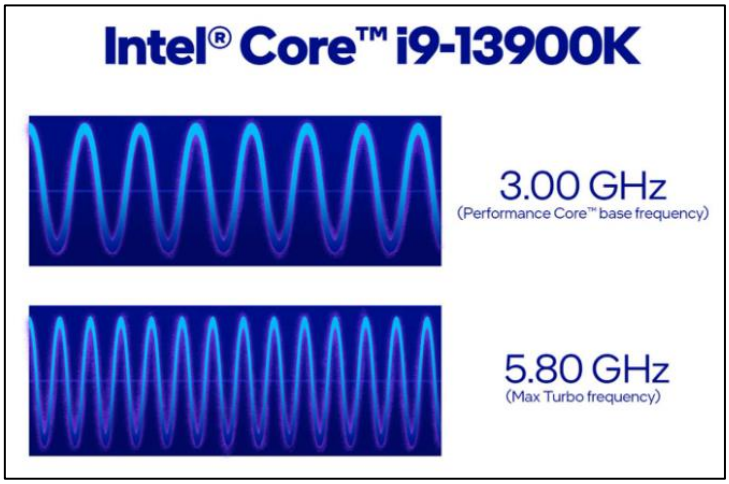


Figure 2.3: The Clock Speeds of Intel Core i9-13900K CPU. [4]

The simplicity, modularity, and broad applicability of the von Neumann architecture have made it the dominant model for general-purpose processing. It provides a clean and scalable framework for implementing control logic, memory management, and sequential instruction execution. For these reasons, its influence persists in nearly every modern computing system—from desktop processors to embedded microcontrollers—despite growing limitations in handling the parallel and high-throughput demands of specialized applications such as artificial intelligence.

2.1.2 Limitations of the Traditional Von Neumann Architecture

Although the von Neumann architecture continues to be widely used in the computing industry due to its straightforward structure for instruction fetching and execution, it has begun to reveal significant limitations—particularly when applied to modern parallel computing and high-throughput workloads, such as those found in state-of-the-art AI systems.

The core issue lies in the limited throughput of the data bus between the memory and the CPU. While modern CPUs can operate at clock speeds in the gigahertz (GHz) range, overall system performance is often constrained by the time it takes to move data through the shared bus. This delay—known as the von Neumann bottleneck—means that the CPU must often idle while waiting for data to be fetched from memory. Figure 2.4 illustrates the speed and power consumption characteristics across different levels of the memory hierarchy [3], further emphasizing this imbalance.

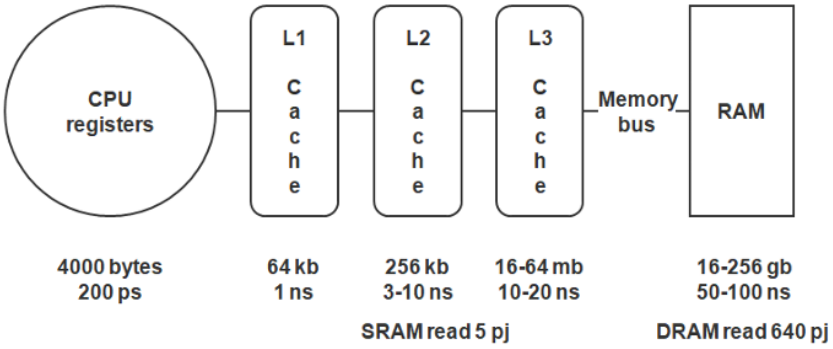


Figure 2.4: Memory Hierarchy, Access Speed and Power Consumption. [3]

As technology evolves toward neural network applications, which demand massive data movement and highly repetitive computations, the limitations of the von Neumann model become more pronounced. Because data must be continuously fetched from memory before processing, the architecture introduces latency that significantly slows down performance. As the amount of data increases and the access range widens, this latency grows substantially—as shown in Figure 2.5 [3].

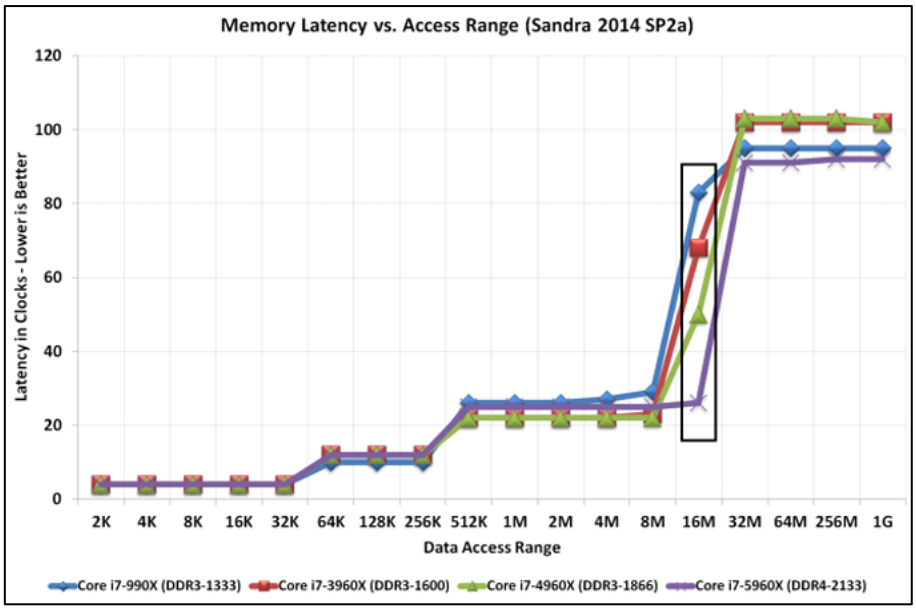


Figure 2.5: Memory Latency Depending on Data Range. [3]

This structural limitation has motivated extensive research into ways to overcome the bottleneck. On one front, alternative memory-centric approaches—such as compute-in-memory (CIM), memristor-based systems, and neuromorphic computing—have been proposed to reduce data transfer delays by processing data closer to where it is stored. On another front, new architectural strategies aim to reduce memory dependency. For instance, modern CPUs incorporate multi-level cache hierarchies to retain frequently accessed data closer to the processor, while GPUs assign large register files to each processing element, enabling thread-level data retention and better parallel task management.

Google's Tensor Processing Units (TPUs) [12] address this bottleneck by integrating a dedicated matrix multiplication unit into the architecture. Each processing element includes its own register space for storing frequently used values such as weights and intermediate results. As a result, TPUs minimize memory access during computation, thereby improving throughput and reducing latency. Further details on TPU design will be discussed in Section 2.4.

2.2 Systolic Architectures

As computing systems evolve to support increasingly parallel and data-intensive workloads—particularly in fields such as artificial intelligence (AI) and signal processing—traditional architectures encounter growing limitations in terms of performance, energy efficiency, and scalability. To address these challenges, systolic

architectures have emerged as a specialized computing model designed for high-throughput and parallel data processing.

By organizing processing elements (PEs) in a regular grid and synchronizing the flow of data between them, systolic arrays enable the efficient execution of repetitive computations with minimal reliance on external memory. This structured, pipelined approach significantly reduces memory bandwidth bottlenecks and improves overall processing efficiency. As a result, systolic architectures have become especially relevant in applications requiring fast and predictable data flow, such as convolutional operations in signal processing and matrix computations in deep learning.

Originally proposed as an alternative to conventional sequential designs, systolic systems continue to influence the architecture of modern domain-specific accelerators. The following sections introduce the fundamental principles of systolic arrays, explore different architectural variations, and examine their application in matrix multiplication—a key operation in many AI workloads.

2.2.1 Introduction to the Systolic Architectures

Systolic architectures are a specialized class of parallel computing designs where data flows rhythmically through a regular network of processing elements (PEs). In these systems, data is not sent back and forth between a central processing unit and memory; instead, it passes from one PE to the next in a coordinated sequence—allowing intermediate results to be computed and propagated in parallel. This design concept was formalized by H. T. Kung [6] in the early 1980s as a general methodology for mapping high-level computations into efficient hardware implementations. Figure 2.6 shows the basic principle of a systolic system [6].

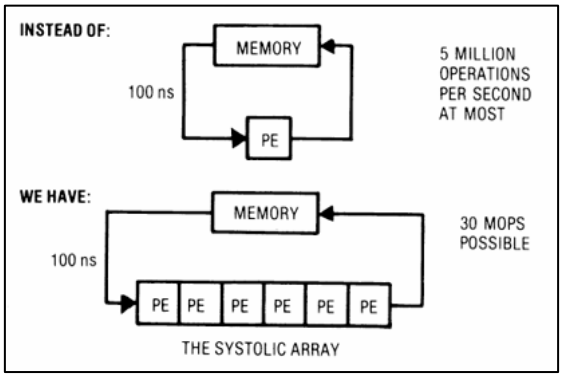


Figure 2.6: Basic Principle of a Systolic System. [6]

The name “systolic” comes from an analogy to the human circulatory system, where blood moves in pulses through a network of vessels. Similarly, in a systolic array, data pulses through the system in a predictable, synchronized fashion. Compared to traditional pipelined architectures where only results move forward, systolic systems support multi-directional data flow, where both inputs and intermediate results can move between PEs at varying speeds and directions.

One of the major strengths of systolic systems is their regularity and modularity. Each processing element performs a simple, repetitive operation and communicates with its immediate neighbors, making the overall system easy to scale and efficient to implement. Architecturally, systolic arrays can be configured in various forms—such as linear, rectangular, triangular, or hexagonal topologies—depending on the computational task and the required level of parallelism.

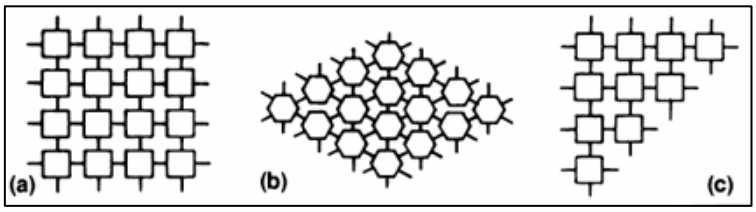


Figure 2.7: Different Styles of Two-dimensional Systolic Arrays. [6]

Because systolic architectures are highly structured, they not only support efficient concurrency, but also reduce memory bandwidth bottlenecks. Data can be reused locally across PEs, which minimizes the need for constant access to external memory—a common limitation in traditional architectures. This makes systolic systems especially well-suited for repetitive, data-flow-driven computations, such as those found in signal processing, deep learning, and matrix operations.

2.2.2 Different types of the Systolic Architectures

Systolic architectures can be implemented in a variety of structural configurations, primarily depending on how data flows between processing elements (PEs). These designs are often selected based on specific application requirements, such as minimizing I/O traffic, maximizing data reuse, increasing parallelism, or simplifying hardware complexity. This section introduces several notable systolic array designs, with a focus on their internal data movement patterns and associated trade-offs.

One well-known approach is illustrated in Figure 2.8 [6]. In this structure, each PE is initialized with a fixed weight value, which remains unchanged during execution. Input

values (x) are streamed vertically through the array, typically from top to bottom, while intermediate results (y) are propagated horizontally from left to right. Each PE performs a multiply-accumulate (MAC) operation using its stored weight and the incoming input, then forwards the result to the next stage. This design enables partial results to accumulate progressively across the array until the final output is available. The simplicity and regularity of this layout make it particularly effective for basic convolutional operations.

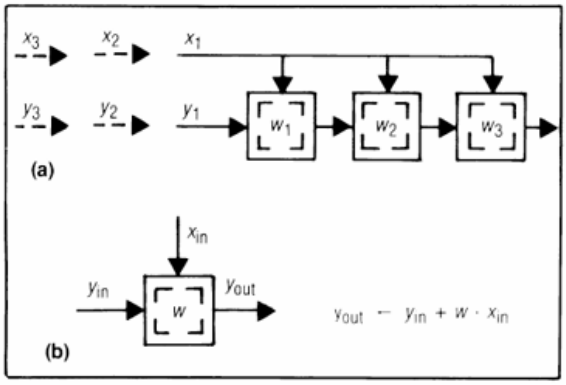


Figure 2.8: Systolic Array with Data Broadcasting: (a) Overall Data Flow, (b) Processing Element Structure. [6]

A contrasting structure provided in Figure 2.9 [6], which avoids forwarding intermediate results across PEs. Here, weights are still fixed within the array, and inputs (x) move from left to right. Instead of propagating partial sums, each PE computes a local result, which is then sent to a centralized fan-in adder for final aggregation. This design simplifies the logic within the array but requires additional global routing for output collection.

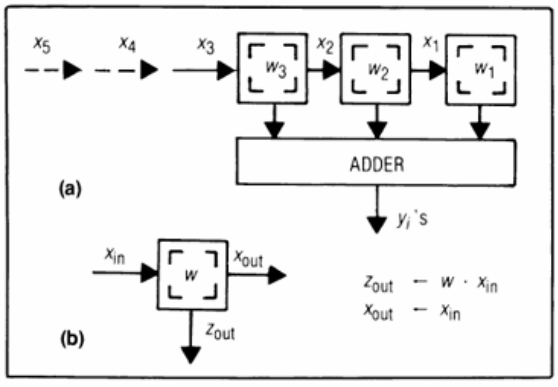


Figure 2.9: Systolic Array with a Fan-in Adder: (a) Overall Data Flow, (b) Processing Element Structure. [6]

Another variation is depicted in Figure 2.10 [6], which adopts a bi-directional flow. Weights are again fixed in each PE, but inputs (x) and partial results (y) travel in opposite directions—inputs move left to right, while results move right to left. Each PE computes a MAC operation and passes the updated y-value backward. This reduces the need for external output connections, as results flow naturally back through the array. However, this design demands precise timing control to manage concurrent bidirectional data movement.

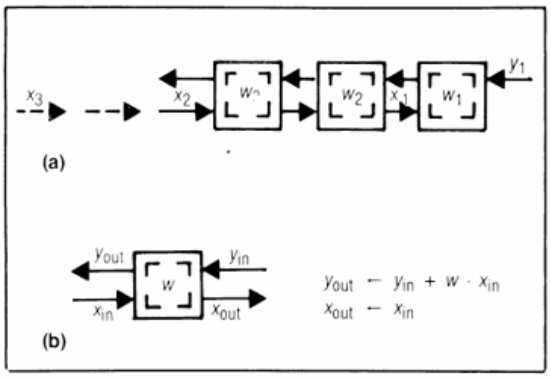


Figure 2.10: Systolic Array with Bi-directional Data Forwarding: (a) Overall Data Flow, (b) Processing Element Structure. [6]

Beyond these foundational structures, other implementations explore alternative strategies to optimize performance and hardware efficiency. One such approach, shown in Figure 2.11 [6], retains partial results within each processing element while shifting weights through the array. This improves MAC unit utilization but makes output collection more complex, as results must be gathered from multiple locations across the array.

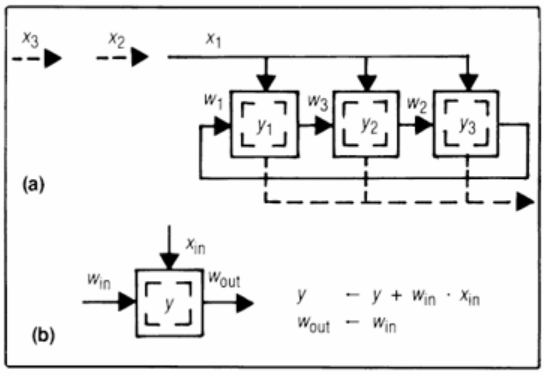


Figure 2.11: Systolic Array with Weight Shifting: (a) Overall Data Flow, (b) Processing Element Structure. [6]

Another design, shown in Figure 2.12 [6], allows both inputs and results to move in the same direction, but at different speeds (e.g., inputs advancing every cycle, results every

two). This ensures that all PEs are consistently active, enhancing throughput. However, the differing speeds add complexity to the control logic and synchronization mechanisms.

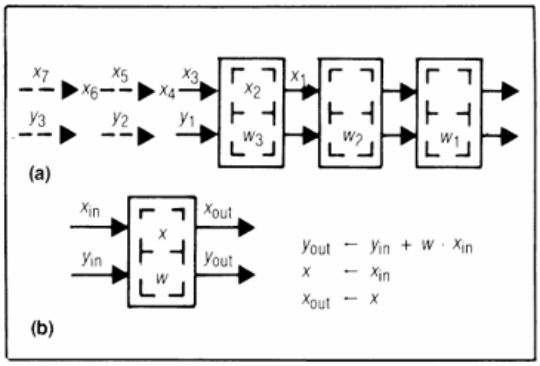


Figure 2.12: Systolic Array with Data Forwarding in Different Speed: (a) Overall Data Flow, (b) Processing Element Structure. [6]

These configurations reflect the flexibility and adaptability of systolic arrays. By selecting an appropriate data flow strategy, designers can tune systolic architectures to meet the unique requirements of their target applications—whether prioritizing speed, area, power efficiency, or simplicity. Each variation offers its own balance between implementation complexity and computational efficiency.

2.2.3 Systolic Array Architecture for Matrix Multiplication

With a well-organized grid of processing elements (PEs), a two-dimensional systolic array can efficiently support operations such as matrix multiplication. Figure 2.13 illustrates a 2D array layout that conceptually represents how a matrix multiplication unit (MMU) can be structured [7]. In this setup, weights are preloaded into each PE—denoted as AD in the figure—while input values (referred to as "search data") are fed into the array in a parallelogram pattern. This ensures that the computations occur in the correct sequence and alignment, producing partial results at the expected intervals.

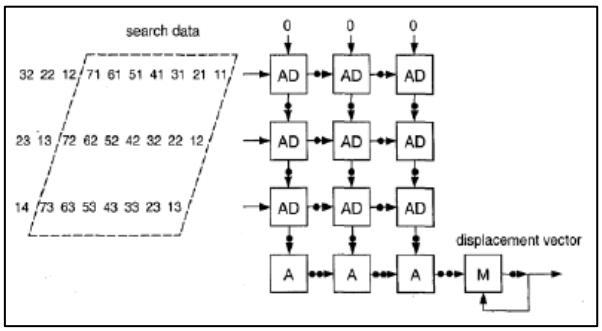


Figure 2.13: Systolic Array Architecture for Matrix Multiplication. [7]

Figure 2.14 presents a block diagram of a single processing element with its associated input and output ports [7]. In this context, A_{ij} represents the activation input (analogous to x in the previous section), and B_{ij} corresponds to the accumulating partial result (analogous to y). While the diagram omits the weight-loading path for clarity, each PE is initialized with its corresponding weight value. When these elements are connected in a modular fashion, they form the complete 2D systolic array, as illustrated in Figure 2.15 [7].

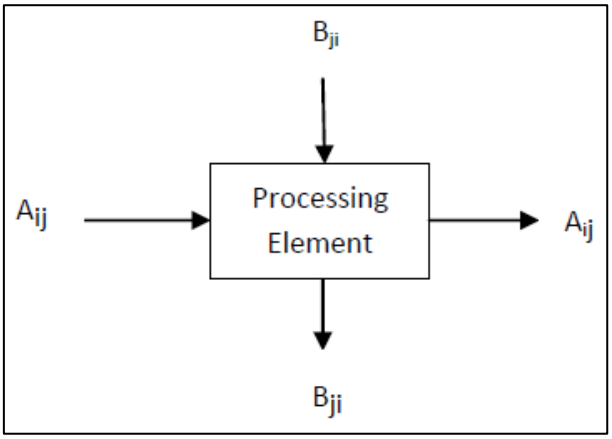


Figure 2.14: PE of Systolic Architecture. [7]

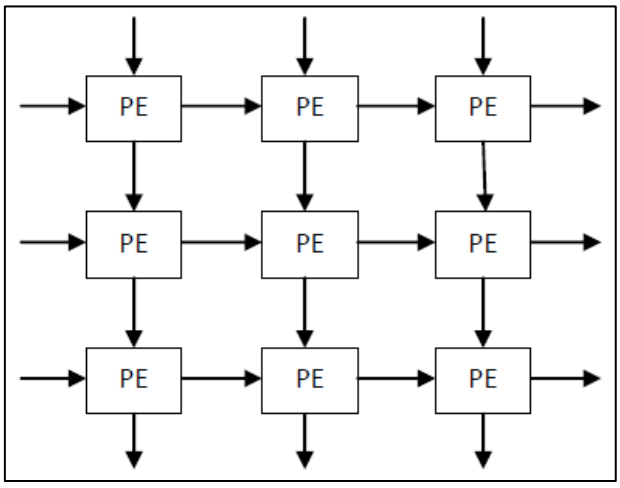


Figure 2.15: Two-dimensional Systolic Array. [7]

To clarify the data movement and timing, Figure 2.16 demonstrates a simple example of 2×2 matrix multiplication using a systolic array. This visual representation helps convey how data propagates through the array and how results are accumulated. Additional hardware components, such as internal registers and interconnections, are also involved in the actual implementation, which will be discussed in more detail in Chapter 4.

$$\vec{a} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} ; \vec{w} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} ; \overrightarrow{Result} = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

$$\overrightarrow{Result} = \begin{bmatrix} a_{11} \times w_{11} + a_{12} \times w_{21} & a_{11} \times w_{12} + a_{12} \times w_{22} \\ a_{21} \times w_{11} + a_{22} \times w_{21} & a_{21} \times w_{12} + a_{22} \times w_{22} \end{bmatrix}$$

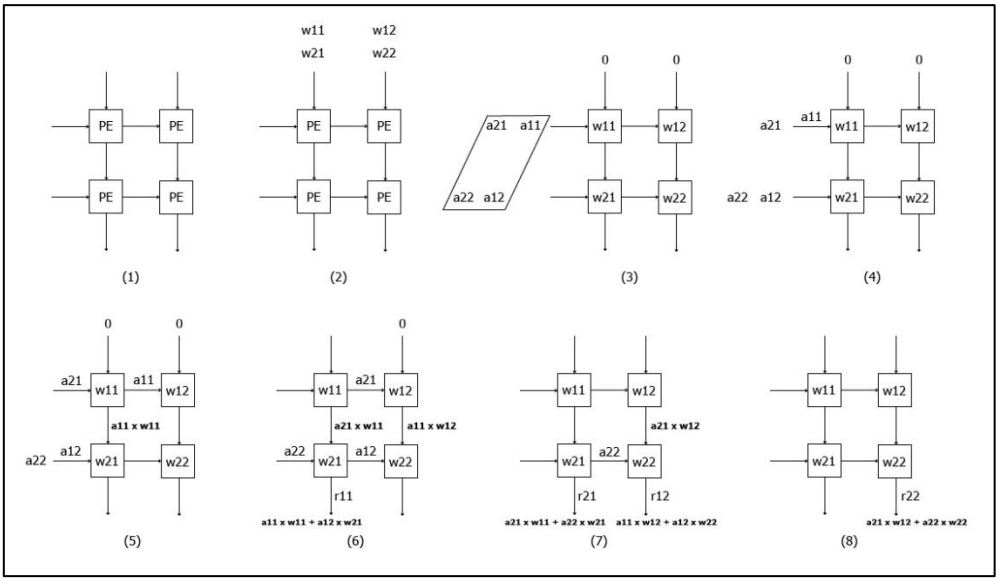


Figure 2.16: An Example of 2×2 Matrix Multiplication Using a Systolic Array, Showing Data Flow Over Time.

2.3 General Matrix Multiplication

As many studies have highlighted [8][9][10][11], General Matrix Multiplication (GEMM) has become a core computational building block in the implementation of artificial intelligence (AI) and deep neural network (DNN) applications. GEMM refers to the mathematical operation of computing the product of two matrices, typically represented as:

$$C = \alpha \cdot A \cdot B + \beta \cdot C$$

where A, B, and C are matrices, and α and β are scalar coefficients. This generalized formulation enables broad reuse across a wide range of linear algebra operations. Rather than replacing C, the result is added to it in a weighted manner, making the operation efficient for iterative computations and memory reuse.

In the context of deep learning, GEMM serves as the computational foundation for several key operations. For instance, fully connected (dense) layers in neural networks can be directly expressed as matrix multiplications, where inputs are multiplied by weight matrices to produce output activations. This is clearly illustrated in Figure 2.17 [8], which shows a two-layer multilayer perceptron (MLP). The graph representation in

Figure 2.17a emphasizes the connectivity between neurons, while Figure 2.17b highlights the matrix structure of the same network—where each layer of weighted sums corresponds to a matrix multiplication. In this view, the core computation of forward propagation becomes a series of GEMM operations across layers.

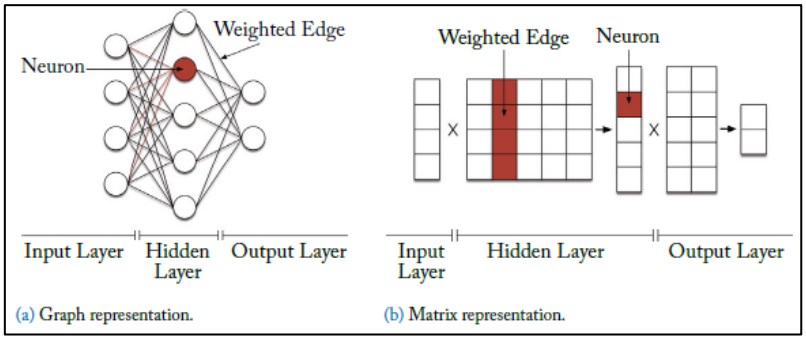


Figure 2.17: Two Ways of Visualizing the Same Multilayer Perceptron. [8]

Due to its regular structure and inherent parallelism, GEMM is highly amenable to hardware acceleration. In addition to traditional 2D matrix operations, modern AI applications—such as convolutional neural networks (CNNs)—extend the concept of GEMM into higher-dimensional (3D-like) computations, as illustrated in Figure 2.18 [10]. Many AI accelerators, including GPUs, TPUs, and custom ASICs, are specifically optimized for GEMM through dedicated compute units, tiling strategies, and support for low-precision formats such as INT8 and FP16. Maximizing GEMM efficiency is often essential for achieving high performance during both inference and training phases of deep neural network workloads.

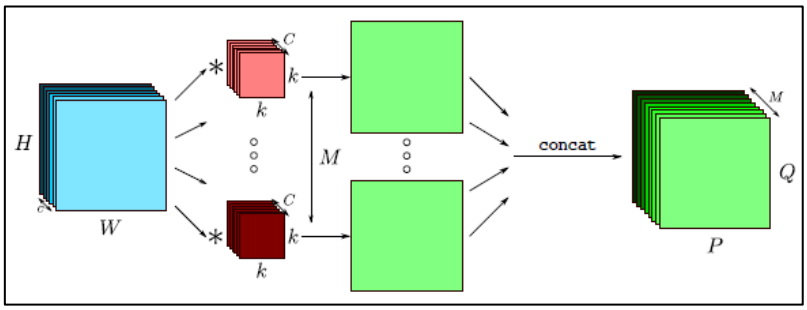


Figure 2.18: Multiple Channel Multiple Kernel (MCMK) Convolution. [10]

In this project, the Matrix Multiplication Unit within the Tiled Matrix Multiplication Unit is designed based on GEMM principles. Building upon this foundation, the architecture aims to improve matrix operation efficiency in resource-constrained edge computing environments, while basic matrix multiplication continues to play a central role in the overall system design.

2.4 Tensor Processing Unit

After understanding how matrix multiplication functions within deep neural network (DNN) applications, and how it can be implemented using systolic array architectures, the next step is to design the supplementary components that enable smooth data flow into the systolic matrix multiplication unit and ensure correct retrieval of computational results.

When Google introduced the Tensor Processing Unit (TPU) in 2017, it provided new insights into how hardware architecture can overcome the von Neumann bottleneck by leveraging a highly parallel array of processing elements (PEs) to achieve significantly higher throughput. Figure 2.19 illustrates the block diagram of Google's first-generation TPU [12], which has since become a widely referenced model for AI accelerator design.

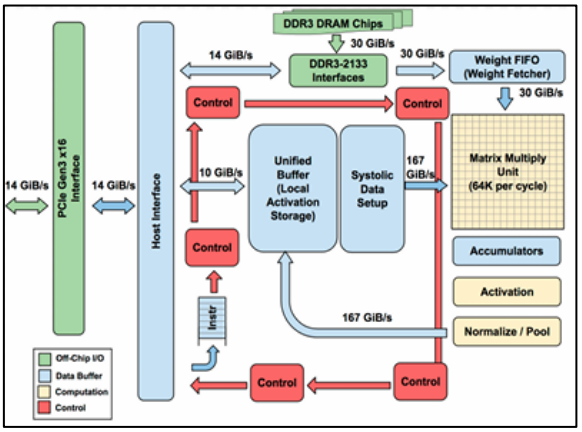


Figure 2.19: TPU Block Diagram. [12]

Similar to the hardware structure and data flow discussed in Section 2.2.3, the TPU's data flow follows a pattern that moves diagonally from the top-left to the bottom-right, as shown in Figure 2.20 [12]. In this design, activation values (Data) flow from left to right, while partial results (Partial Sums) propagate from top to bottom.

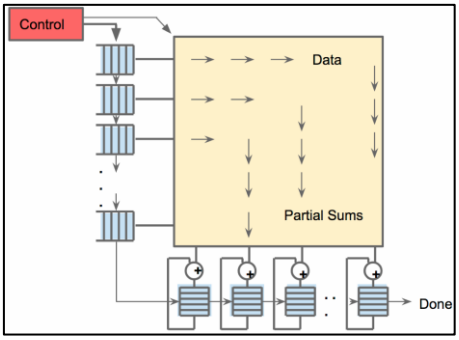


Figure 2.20: Systolic Data Flow of the Matrix Multiply Unit. [12]

Rather than employing a bi-directional systolic design, which would require more complex I/O port configurations, the TPU architecture simplifies the layout by placing input ports along the top and left edges of the matrix multiplication unit, with output ports positioned at the bottom. This configuration offers advantages in both modularity and implementation simplicity.

Figure 2.21 [20] provides a simplified block diagram of the matrix multiplication unit, illustrating its internal structure and data flow. Figure 2.22 [20] highlights several supplementary components that support data movement within the TPU, enabling complete and efficient computation. Together, these figures offer a clearer understanding of the architecture and functionality of the first-generation TPU.

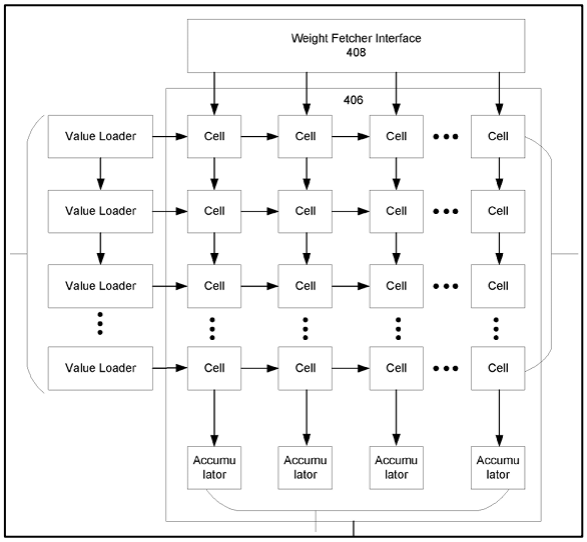


Figure 2.21: Simplified Block Diagram of the Matrix Multiplication Unit. [20]

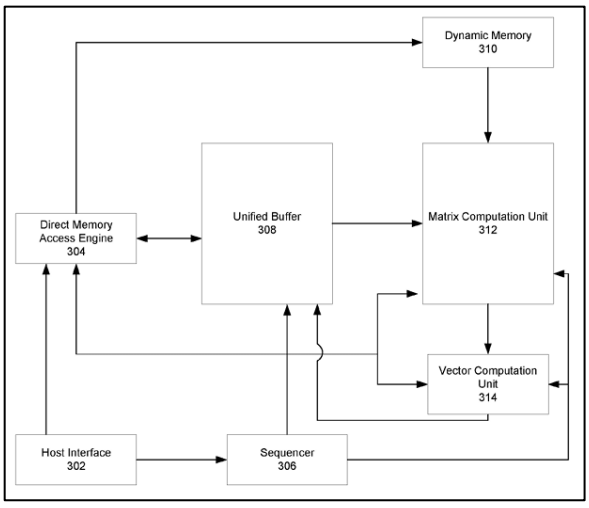


Figure 2.22: Partial TPU Block Diagram Illustrating Data Movement Directions. [20]

Within the TMMU developed in this project, an internal TPU-based structure has been implemented to support general matrix multiplication. Based on this architectural design, tiled matrix multiplication is achieved by feeding data in a specific sequence, along with the addition of extra registers in certain components to store temporary results during computation. Further implementation details will be discussed in the following section.

2.5 Summary

Chapter 2 provided an overview of the foundational concepts and related technologies that support the hardware design in this project. Beginning with the von Neumann architecture, the chapter outlined its historical significance and core components as a widely adopted computing model, while also introducing its limitations in handling the parallelism and memory demands of modern AI applications.

To address these challenges, the chapter presented systolic architectures, emphasizing their structured data flow and modularity for efficient parallel computation. Several systolic array designs were introduced, with a particular focus on their application in matrix multiplication, demonstrating how two-dimensional arrays can efficiently perform key operations in neural networks.

The chapter also discussed General Matrix Multiplication (GEMM) as a fundamental computational kernel in deep neural networks (DNNs), highlighting its use in fully connected layers and extended convolution tasks. Due to its regular structure and support for low-precision formats, GEMM remains a central target for hardware optimization in AI systems.

Finally, the chapter reviewed the architecture of Google's Tensor Processing Unit (TPU), which integrates systolic computation, implements GEMM in hardware, and streamlines data flow to address von Neumann bottlenecks. This architecture serves as an important reference for the Tiled Matrix Multiplication Unit (TMMU) developed in this project, which further aims to improve efficiency in edge-based AI workloads.

Together, these discussions provide the conceptual background for the hardware implementation described in the following chapters.

CHAPTER 3

TMMU: Overview and Control

As technology continues to evolve, its focus has shifted from powering large-scale systems like supercomputers for data centers or military equipment to becoming embedded in devices we use every day—such as smartphones, wearables, and automotive chips. These everyday technologies prioritize convenience and compactness, often operating under strict hardware resource constraints. As a result, new strategies are needed to maintain performance within limited hardware environments.

This chapter begins by introducing tiled matrix multiplication, a widely adopted strategy for enabling matrix operations on hardware-limited systems. By breaking down large computations into manageable pieces, this method improves hardware utilization and supports scalable performance without requiring extensive computational resources. Within the context of this project, tiled matrix multiplication is central to the architecture of the Tiled Matrix Multiplication Unit (TMMU), serving as a critical mechanism for matrix operations

Following the introduction of tiled matrix multiplication, the chapter then presents the CISC (Complex Instruction Set Computer) instruction format designed specifically for this project. This custom instruction set plays a critical role in managing the execution of matrix operations by encoding the necessary control signals for both the TMMU and the embedded TPU module. Its design balances instruction flexibility with encoding simplicity, ensuring that the control unit can interpret and dispatch operations efficiently within the constraints of the system.

The subsequent section explores the hierarchical controller structure implemented in this project, which consists of two levels of control. The Level 2 controller, positioned at the TMMU top level, orchestrates the data flow, instruction sequencing, and mode selection. The Level 1 controller, embedded within the TPU, manages the internal execution of matrix multiplication operations in a fine-grained manner. Each controller is discussed in detail, including its I/O interface, finite state machine design, and interaction with other components in the architecture.

Together, these elements form the basis of the TMMU's operation and control logic. This chapter lays the architectural groundwork for the Verilog-based hardware implementation presented in the following chapter, where each functional block and control mechanism is mapped into RTL design.

3.1 Tiled Matrix Multiplication

Tiled matrix multiplication is a widely adopted strategy for optimizing large matrix computations in hardware-constrained environments. Instead of computing a large matrix operation all at once, the data is divided into smaller blocks that can be processed sequentially or in parallel, depending on the architecture. This approach improves data reuse, reduces memory access bottlenecks, and enables efficient integration with compact compute units.

In this project, tiled matrix multiplication is used to perform 8×8 matrix operations using a 4×4 Matrix Multiplication Unit (MMU). This section introduces the computational flow of the design, discusses how memory data is stored and interpreted, and explains how input data is rearranged to support tiled execution in the systolic MMU structure.

3.1.1 Tiled Computation Flow for 8×8 Matrices

In tiled matrix multiplication, there are several possible computation orders for completing a full 8×8 matrix using four smaller 4×4 matrix blocks. Figure 3.1 illustrates the computation flow adopted in this project, where the green matrix represents the weights, the blue matrix represents the activations, and the red matrix represents the output (result) matrix. The result matrix is computed in the following order: top-left block (Block 1), bottom-left (Block 2), top-right (Block 3), and finally bottom-right (Block 4).

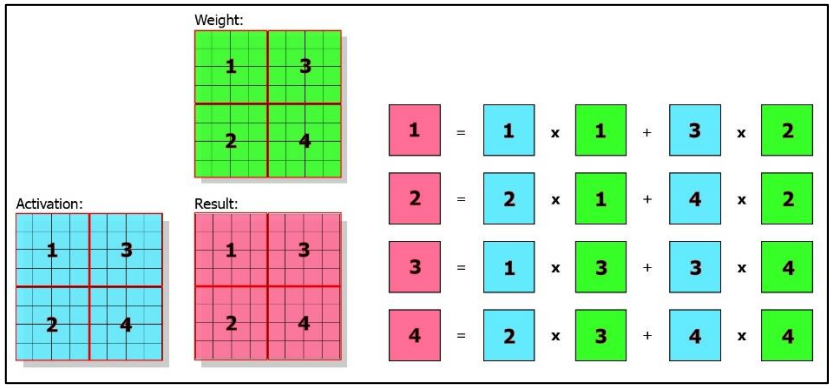


Figure 3.1: Tiled Matrix Multiplication Block Computation Order.

As shown in Figure 3.1, each output matrix block (OB) is computed by combining specific activation blocks (AB) and weight blocks (WB). For example, OB1 is calculated using AB1 × WB1 and AB3 × WB2, while OB2 uses AB2 × WB1 and AB4 × WB2. Similarly, OB3 is derived from AB1 × WB3 and AB3 × WB4, and OB4 from AB2 × WB3 and AB4 × WB4. This block-wise accumulation completes the full 8×8 result matrix. Although this computation order requires a few additional Read Host Memory instructions to load the correct weight blocks at the right time, it has proven to be efficient in terms of both instruction count and overall computation time in this project.

Alternative computation orders for tiled matrix multiplication do exist. However, each presents its own drawbacks that ultimately reduce the overall efficiency of the TMMU design in this project.

1. *Structural Modularity Issues:*

The first example, shown in Figure 3.2, presents challenges related to modularity. If the computation blocks for the weight and activation matrices are loaded in an inconsistent order, it leads to one of two problems: either each memory must implement a separate read mechanism, or the Tiled Systolic Data Setup Unit (to be discussed in Chapter 4) must support two distinct reordering mechanisms. Both options complicate the design, increase debugging effort, and raise implementation costs—without offering significant gains in performance.

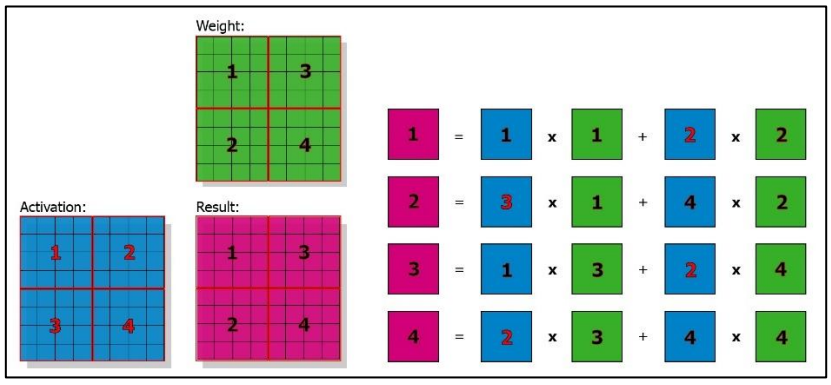


Figure 3.2: Tiled Computation Order Causing Modularity Constraints.

2. *Additional Instruction Overhead:*

The second example, illustrated in Figure 3.3, suffers from excessive weight updates. This approach requires frequent Weight Load instructions before each computation step. While the current method used in this project also involves some redundant instructions, it avoids repeated weight loading. Since weight loading involves considerable data movement, the approach in Figure 3.3 leads to increased power consumption, making

it less efficient than the method adopted here.

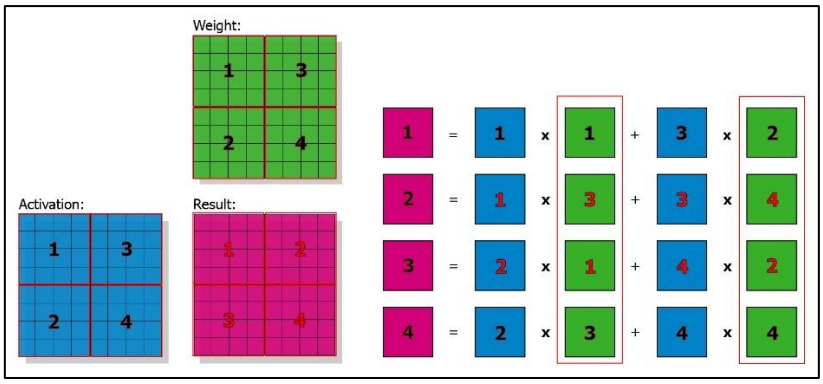


Figure 3.3: Tiled Computation Order Requiring Excessive Weight Updates.

3. *Inefficient Computation Flow:*

A third alternative, presented in Figure 3.4, also results in continuous weight loading during the computation process. As mentioned in the first issue, maintaining modularity requires the weight and activation matrices to follow the same computation order. However, even when this matching order is applied as shown in the figure, the design still suffers from repeated weight loading. This leads to greater power consumption due to unnecessary memory transactions.

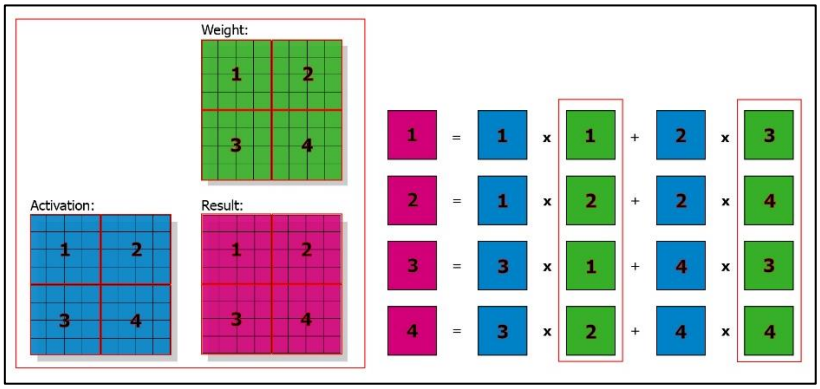


Figure 3.4: Tiled Computation Order with Inefficient Weight Reuse.

4. *Poor Weight Loading Order:*

The final example, shown in Figure 3.5, demonstrates the issue of disordered weight loading. As discussed earlier, maintaining consistent computation orders for the weight and activation matrices ensures structural modularity. However, in this case, the result matrix is computed in a different order, which forces the weight blocks to be loaded in a suboptimal sequence (1, 3, 2, 4). This disordered loading pattern leads to an increase in Read Host Memory instructions and additional effort in managing the weights in external DDR3 memory before sending them to the MMU.

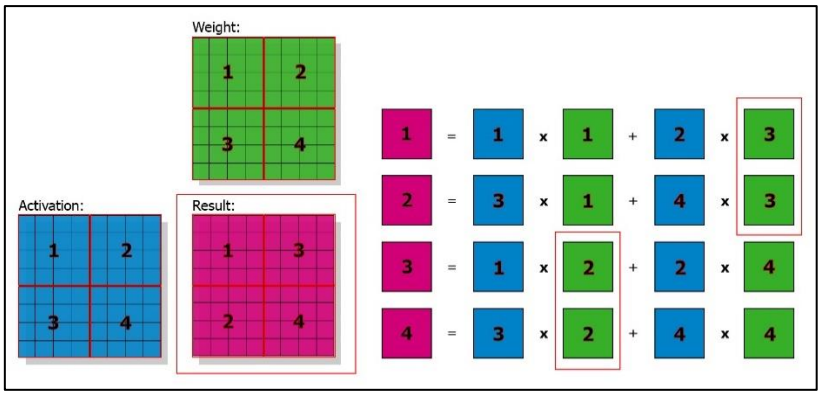


Figure 3.5: Tiled Computation Order with Suboptimal Weight Loading Sequence.

Given the performance, modularity, and power considerations across these alternative approaches, this project adopts the block computation order shown in Figure 3.1, which strikes the best balance between efficiency and simplicity.

3.1.2 Memory Layout: From Row-Major Format to Matrix View

To realize the TMM mechanism in this project, data order rearrangement plays a key role in enabling its implementation. Before diving into the mechanism, it is important to note that all memory structures in this design store data in row-major format. Figure 3.6 illustrates how an 8×8 matrix is stored using row-major order and how it maps from linear memory layout to matrix form.

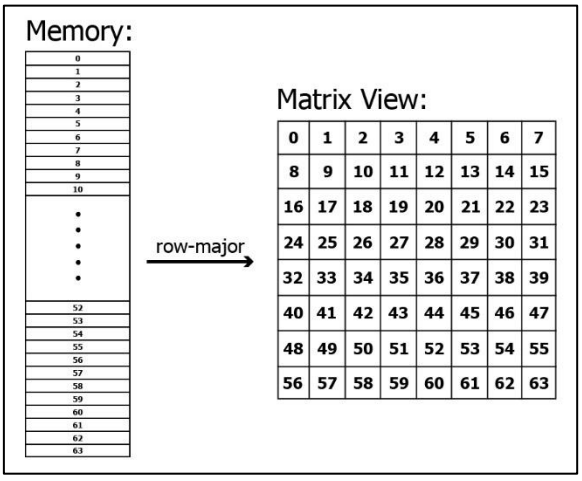


Figure 3.6: Row-major Data Storage in Memory and its Corresponding Matrix Representation.

3.1.3 Tiling Adaptation for Variable Matrix Sizes

The primary motivation for implementing tiled matrix multiplication in this project

stems from hardware resource limitations, which prevent computing the entire matrix at once. Specifically, an 8×8 matrix must be computed using only a 4×4 Matrix Multiplication Unit (MMU). As a result, data order rearrangement becomes a crucial part of the design.

Since the block computation follows the sequence outlined in Section 3.1.1, Figure 3.7 illustrates the full 8×8 matrix rearrangement process. On the left is the original matrix, with values stored in row-major order. To match the block computation sequence shown in the middle of the figure, the data must be reorganized into the format shown on the right. The square brackets in the bottom-left matrix represent the memory positions of the 4×4 submatrices, rather than the actual data values.

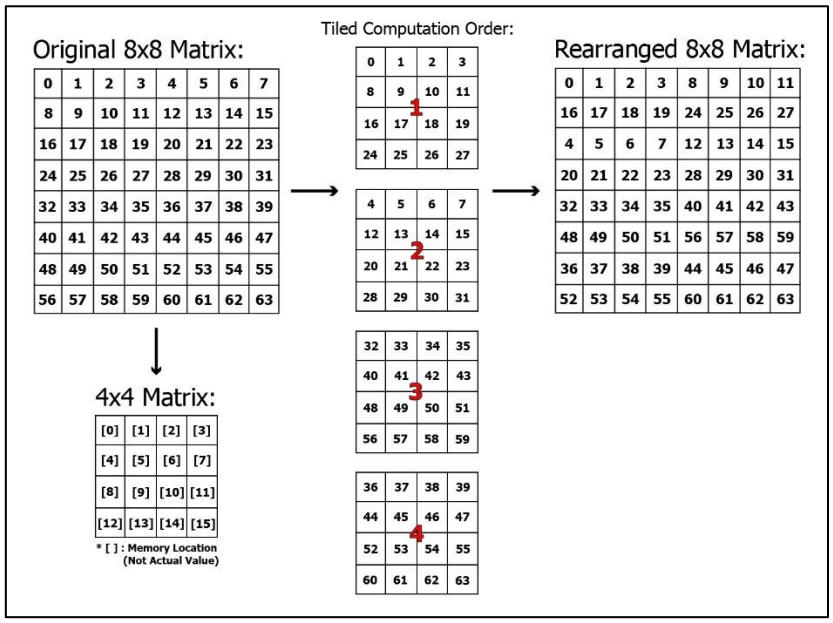


Figure 3.7: 8×8 Matrix Rearrangement Process.

Therefore, the core function of the Tiled Systolic Data Setup Unit is to rearrange matrix data before sending it to the TPU for computation. Since the TMMU supports matrix multiplication sizes ranging from 5×5 to 8×8, and the memory size is fixed at 64 locations, it is important to consider how the data is organized from a memory-mapping perspective. When viewed this way, the full 8×8 matrix—composed of four 4×4 submatrices—can be visualized as shown in Figure 3.8. In the figure, values enclosed in square brackets represent memory locations, not actual data. The red numbers in the center of each 4×4 block indicate the block computation sequence.

Since a 5×5 matrix requires only 25 out of 64 memory locations, 6×6 uses 36, and 7×7 uses 49, some memory locations will remain unused—these are referred to as sparse elements. A straightforward way to handle this sparsity is to fill the unused locations

with zeros, which do not affect the outcome of matrix multiplication. Figures 3.9 through 3.12 illustrate the data storage layout for 5×5, 6×6, 7×7, and 8×8 matrices. In each figure: (a) shows the memory-mapped format, (b) represents the matrix form, and (c) illustrates the 4×4 block computation sequence that is sent into the TPU for matrix multiplication.

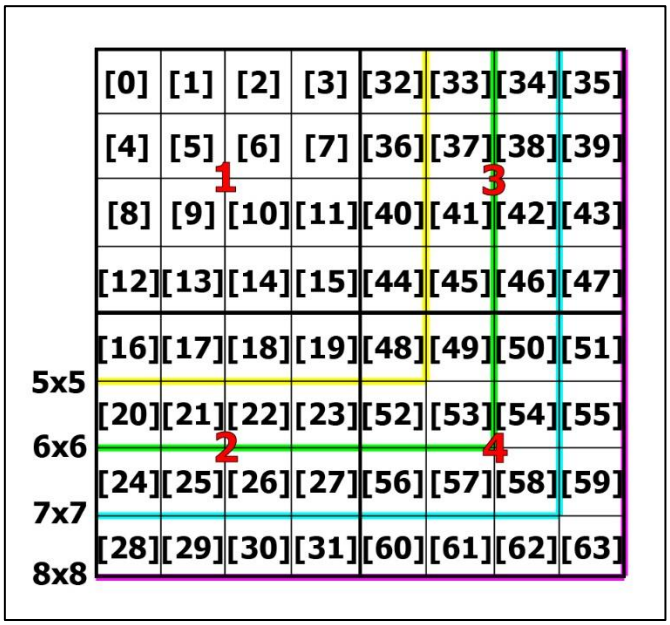


Figure 3.8: Memory-Based Rearrangement for Tiled Computation of Matrix Sizes from 5×5 to 8×8.

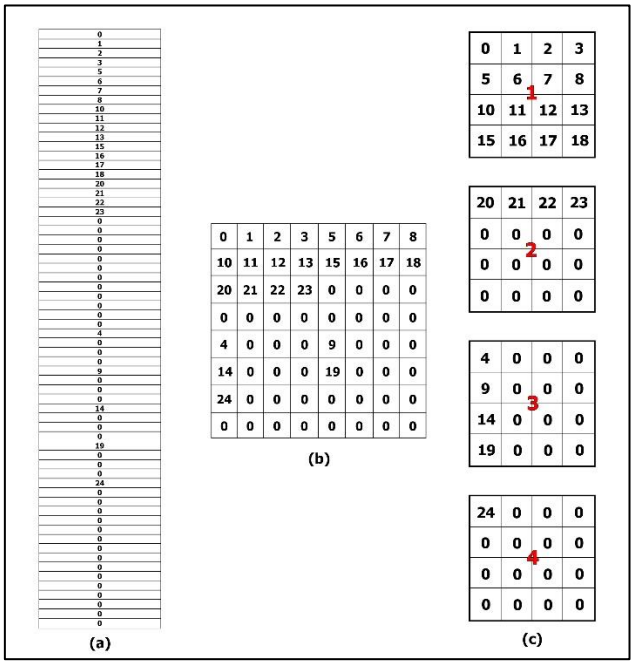


Figure 3.9: 5×5 Data Storage Layout. (a) Memory-mapped Format, (b) Matrix Form, and (c) Block Computation Sequence.

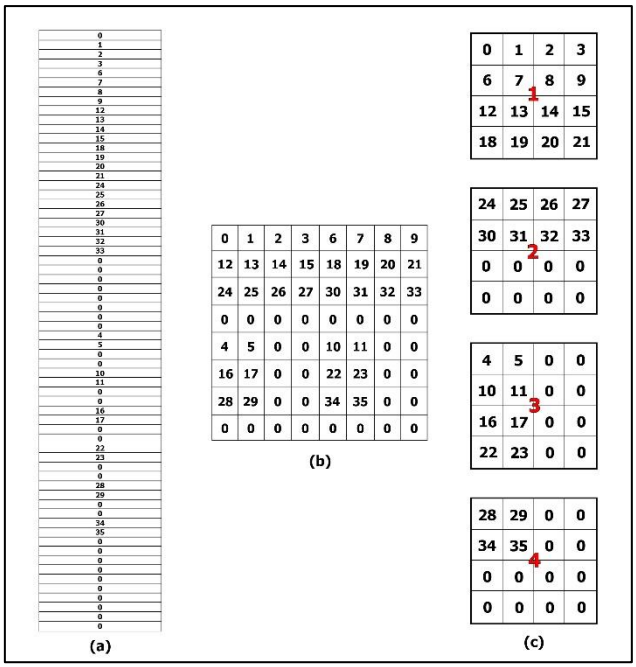


Figure 3.10: 6×6 Data Storage Layout. (a) Memory-mapped Format, (b) Matrix Form, and (c) Block Computation Sequence.

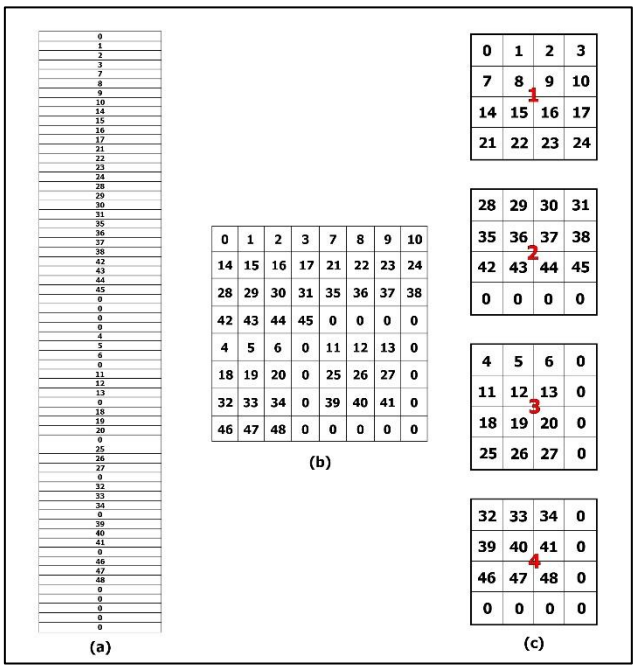


Figure 3.11: 7×7 Data Storage Layout. (a) Memory-mapped Format, (b) Matrix Form, and (c) Block Computation Sequence.

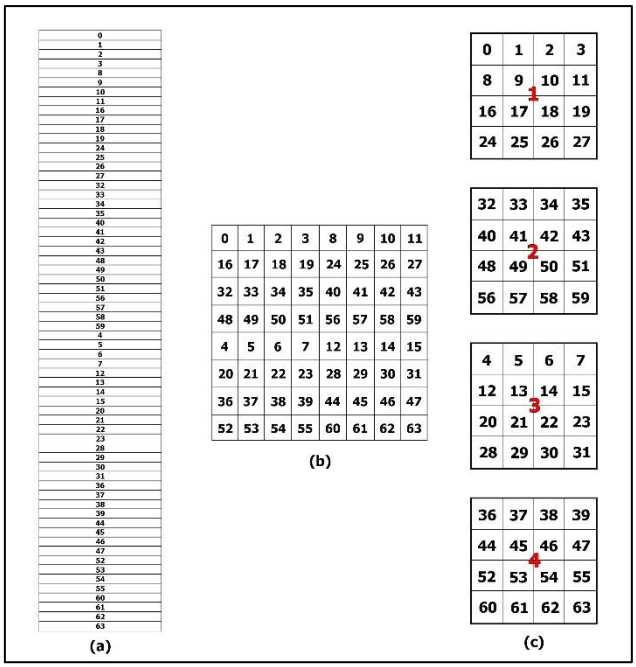


Figure 3.12: 8×8 Data Storage Layout. (a) Memory-mapped Format, (b) Matrix Form, and (c) Block Computation Sequence.

3.2 CISC Instruction Set

In order for both controllers to properly decode and send the correct signals that trigger each component within the TMMU, two separate CISC instruction sets have been developed to support system functionality. This customized CISC design avoids the need to carry actual computation data within the instruction stream, as would be required in RISC-based implementations, thus reducing overhead and improving control efficiency. Instead, only the necessary control information is passed for decoding.

Both instruction sets are 16 bits wide, with several bits reserved for future expansion. Upon receiving an instruction, the controller decodes the four most significant bits (MSBs), labeled IR[15:12], to determine the operation type. Table 3.1 shows the ISA for the Level 2 TMMU Controller. One specific instruction worth highlighting is Read L2 Host Memory, which includes additional bits for matrix operation configuration:

1. *IR[6] – Tiled Matrix Multiplication Definition Bit:* If set to 0, the controller executes basic matrix multiplication. If set to 1, it performs tiled matrix multiplication.
2. *IR[5:0] – Matrix Size Bits:* These 6 bits specify the matrix size in binary. This project supports matrix sizes up to 8 (i.e., 1000 in binary). Larger matrix sizes are reserved for future development.

Top Module (TMMU) Control Unit						
	ISA	**IR[15:12]**	**IR[11:7]**	**IR[6]**	**IR[5:4]**	**IR[3:0]**
1	*Load ISA*	0000	00000	0	00	0000
2	*Read L2 Host Memory*	0001	00000	TiledMM Defined Bit	6-bit for Computed Matrix Size	
3	*TPU work*	0010	00000	0	00	0000
4	*Write L2 Host Memory*	0011	00000	0	00	0000

Table 3.1: TMMU Controller CISC Instruction Set Overview.

Table 3.2 outlines the CISC ISA used for the TPU Controller. Like the TMMU ISA, the first four MSBs (IR[15:12]) specify the operation. Additional bits provide further control in two key functional areas:

Computation:

1. *IR[11:0] – Iteration Count:* These 12 bits define how many iterations (or computation cycles) should be performed on the current set of data stored in the Unified Buffer. This mechanism allows repeated computation on the same data without fetching from higher-level memory, improving efficiency.

Activation:

1. *IR[11:9] – Activation Function Select:* A 3-bit field to choose between activation functions such as ReLU, Sigmoid, and SoftMax. Currently, only ReLU is implemented in this project; the others are reserved for future work.
2. *IR[8] – Max Value Mode Select:* This bit determines the method of maximum value scaling post-activation. If set to 0, vector-wise scaling is used (each row divided by its own max value). If set to 1, matrix-wise scaling is applied (entire matrix divided by the global max value).
3. *IR[7:0] – User-Defined Scaling Factor:* These 8 bits define a user-specified scaling boundary. This serves to prevent result overflow and introduces controlled rounding or bias into the output.

TPU4x4 Controller						
	ISA	**IR[15:12]**	**IR[11:9]**	**IR[8]**	**IR[7:4]**	**IR[3:0]**
1	*Read Host Memory*	0000	000	0	0000	0000
2	*Read Weight*	0001	000	0	0000	0000
3	*Computation*	0010	12-bit for the Number of Iterations			
4	*Activation*	0011	Activation Function Select	V/M MaxVal	8-bit for User-defined Factor	
5	*Write Host Memory*	0100	000	0	0000	0000

Table 3.2: TPU Controller CISC Instruction Set Overview.

A potential area for future enhancement is to unify these two ISAs into a single instruction format. However, this would require additional bits and hardware—such as a comparator—to differentiate between TMMU and TPU instructions and ensure proper decoding. To maintain implementation simplicity and reduce hardware complexity, the project currently keeps the instruction sets separate.

3.3 Multi-Level Control Architecture

The TMMU includes two controllers: the Level 2 Controller, which manages components outside the TPU and initiates TPU operations, and the Level 1 Controller, also referred to as the TPU Controller, which handles internal control within the TPU. A detailed block diagram of the structure is provided in Chapter 4 for better understanding.

Figure 3.13 provides a simplified overview of the multi-level control architecture. As shown in the figure, only the Level 2 Controller is directly reset. The Level 1 Controller is initialized when the *L1_Controller_start_sig signal* is asserted by the Level 2 Controller. Once the TPU completes all computation tasks, the Level 1 Controller sends back a *TPU_Computation_finished_signal* to notify the Level 2 Controller to proceed with the next operations. Further details will be discussed in the following sections.

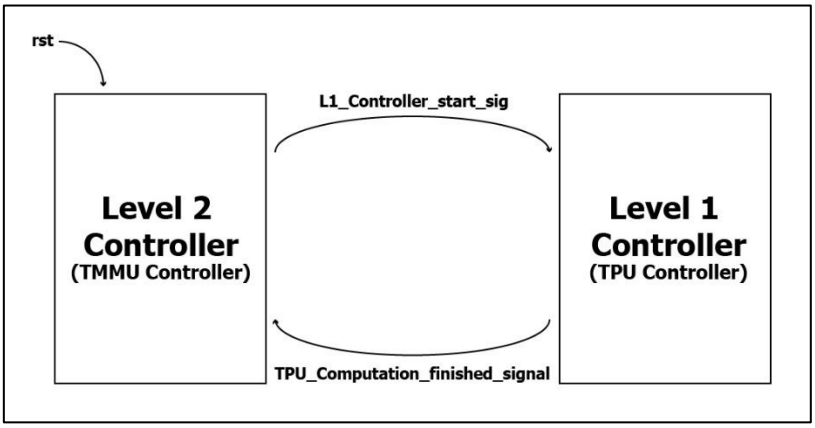


Figure 3.13: Simplified Multi-Level Control Architecture between TMMU and TPU.

3.3.1 Level 2 Controller: Top-Level TMMU Management

This section provides a detailed overview of the Level 2 Controller (TMMU Controller), including its control state diagram and the corresponding I/O signals used in each state.

Figure 3.14 presents the overall state diagram of the Level 2 Controller. As shown in the figure, the controller operates through 16 distinct states. Table 3.3 lists the full names and descriptions of each abbreviated state for clarity and reference.

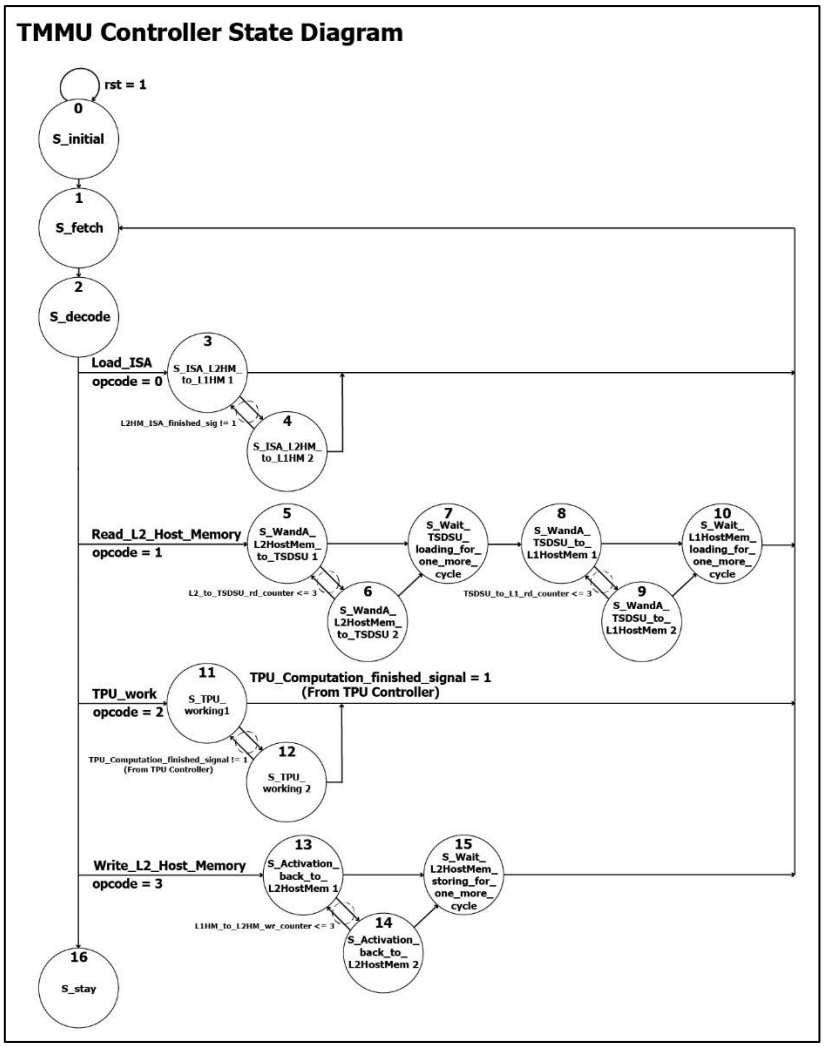


Figure 3.14: Level 2 Controller State Diagram.

0	*S_initial*	State_Initial
The initial state, which sends reset signals to the corresponding components.		
1	*S_fetch*	State_Fetch
The state that fetches an instruction from the Level 2 IR Memory.		
2	*S_decode*	State_Decode
The state that decodes the instruction and determines the next state based on the result.		
3	*S_ISA_L2HM_to_L1HM1*	State_ISA_Level 2 Host Memory_to_ L1 Host Memory_state1
The first state that loads the instruction from the ISA Level 2 Host Memory to the Level 1 IR Memory, allowing the TPU Controller to decode it.		
4	*S_ISA_L2HM_to_L1HM2*	State_ISA_Level 2 Host Memory_to_ L1 Host Memory_state2
The second state that loads the instruction from the ISA Level 2 Host Memory to the Level 1 IR Memory, allowing the TPU Controller to decode it.		

5	*S_WandA_L2HostMem_to_TSDSU1*	State_Weights and Activations_L2 Host Memory_to_ Tiled Systolic Data Setup Unit_state1
The first state that loads weights and activations from the corresponding Level 2 Host Memory into their respective Tiled Systolic Data Setup Units.		
6	*S_WandA_L2HostMem_to_TSDSU2*	State_Weights and Activations_L2 Host Memory_to_ Tiled Systolic Data Setup Unit_state2
The second state that loads weights and activations from the corresponding Level 2 Host Memory into their respective Tiled Systolic Data Setup Units.		
7	*S_Wait_TSDSU_loading_for_one_more_cycle*	State_Wait_Tiled Systolic Data Setup Unit_loading_for_one_more_cycle
The state that waits one additional cycle to ensure the Tiled Systolic Data Setup Units correctly store the weights and activations.		
8	*S_WandA_TSDSU_to_L1HostMem1*	State_Weights and Activations_Tiled Systolic Data Setup Unit_to_ L1 Host Memory_state1
The first state that loads weights and activations from the respective Tiled Systolic Data Setup Units into the corresponding Level 1 Host Memory.		
9	*S_WandA_TSDSU_to_L1HostMem2*	State_Weights and Activations_Tiled Systolic Data Setup Unit_to_ L1 Host Memory_state2
The second state that loads weights and activations from the respective Tiled Systolic Data Setup Units into the corresponding Level 1 Host Memory.		
10	*S_Wait_L1HostMem_loading_for_one_more_cycle*	State_Wait_L1 Host Memory_ loading_for_one_more_cycle
The state that waits one additional cycle to ensure the Level 1 Host Memory correctly store the weights and activations.		
11	*S_TPU_working1*	State_TPU_working_state1
The first state that waits while the TPU is performing computation.		
12	*S_TPU_working2*	State_TPU_working_state2
The second state that waits while the TPU is performing computation.		
13	*S_Activation_back_to_L2HostMem1*	State_Activations_back_to_ L2 Host Memory_state1
The first state that transfers computed activations from the Level 1 Host Memory back to the corresponding Level 2 Host Memory.		
14	*S_Activation_back_to_L2HostMem2*	State_Activations_back_to_ L2 Host Memory_state2
The second state that transfers computed activations from the Level 1 Host Memory back to the corresponding Level 2 Host Memory.		
15	*S_Wait_L2HostMem_storing_for_one_more_cycle*	State_Wait_L2 Host Memory_ storing_for_one_more_cycle

The state that waits one additional cycle to ensure the Level 2 Host Memory correctly store the computed activations.		
16	*S_stay*	State_stay
The state that remains idle while awaiting the next instruction.		

Table 3.3: State Name Descriptions for Level 2 Controller FSM.

The remainder of this section introduces each state in detail, along with the corresponding I/O signals asserted by the TMMU Controller. This breakdown helps clarify which signals are active in each state, offering insight into the dataflow and revealing opportunities for potential optimization.

Figure 3.15 illustrates the I/O signals used in states S0 through S4. In the initial state S0, all reset signals for the relevant components are asserted (set to 1) to ensure proper initialization. In state S1 (Fetch), the signal L2IR_rd is asserted to enable the Level 2 Instruction Register to read an instruction for decoding. In state S2 (Decode), if the opcode (IR[15:12]) equals 0000, the controller initiates the 'Load ISA' operation to transfer instructions to the TPU Controller. During states S3 and S4, the controller checks the L2HM_ISA_finished_sig signal, which is asserted by the ISA_L2_HostMem module. If this signal remains low, the controller continues toggling between S3 and S4 to load instructions. Once the signal is asserted, instruction loading halts, and the L2IR_counter is incremented to fetch the next instruction for decoding.

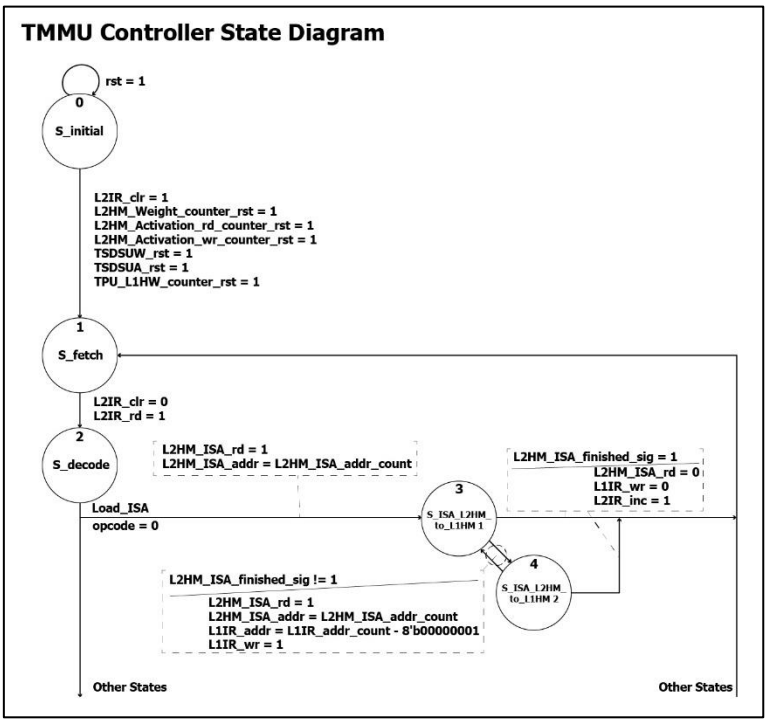


Figure 3.15: I/O Signals for Instruction Fetch and Decode States (S0–S4).

Figure 3.16 shows the I/O signals used in states S5 through S10. If the opcode (IR[15:12]) equals 0001, the controller initiates the ‘Read L2 Host Memory’ operation to transfer both weights and activations from the Level 2 Host Memory into the Tiled Systolic Data Setup Units (TSDSUs) and subsequently into the Level 1 Host Memory.

Before entering S5, the signal zerofill_sig is asserted (= 1) to initialize the TSDSUs' memory elements with zeros. During states S5 and S6, the internal counter L2_to_TSDSU_rd_counter is incremented on each cycle. In these states, both the Weight and Activation L2 Host Memories perform write operations, while the TSDSUs perform read operations—allowing them to overwrite the zero-filled memory with incoming data and rearrange it into the appropriate memory locations for matrix reordering.

Once L2_to_TSDSU_rd_counter > 3 (indicating that 64 data elements have been transferred), the controller transitions to S7, which provides a one-cycle delay to ensure all data is correctly stored in the TSDSUs.

In states S8 and S9, the controller asserts signals to allow the TSDSUs to perform read operations while the Level 1 Host Memories perform write operations, transferring the rearranged data into their respective memory structures. Once the counter TSDSU_to_L1_rd_counter exceeds 3, the controller moves to S10, which provides another one-cycle delay to ensure data is securely written to the L1 Host Memory. After that, the L2IR_counter is incremented to fetch the next instruction for decoding.

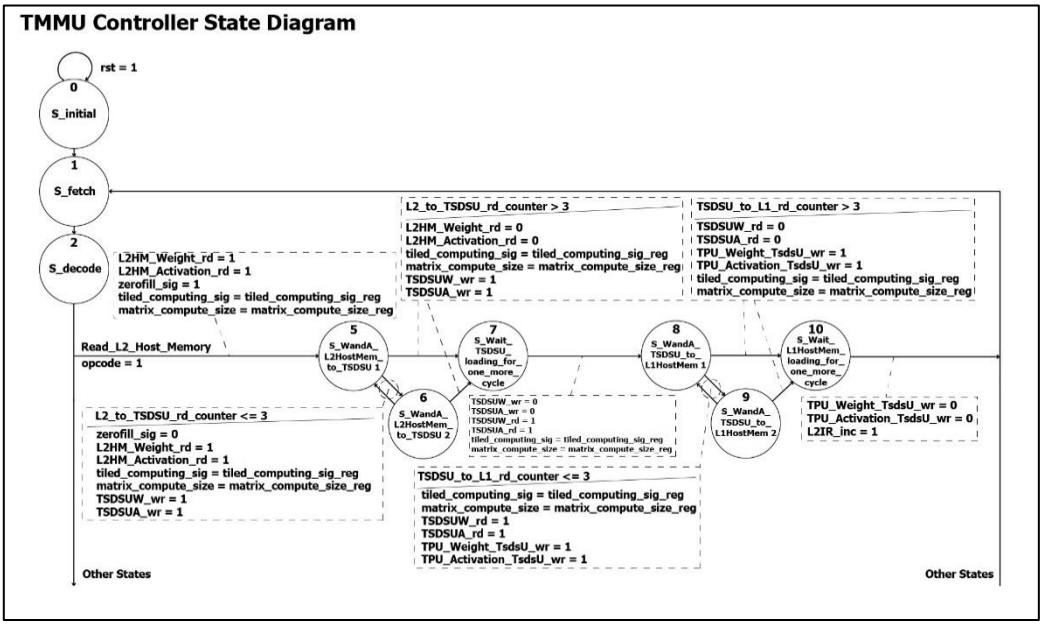


Figure 3.16: I/O Signals for Data Transfer States (S5–S10).

Figure 3.17 shows the I/O signals used in states S11 and S12. If the opcode (IR[15:12]) equals 0010, the controller initiates the “TPU Work” operation by asserting the L1_Controller_start_sig, which is sent to the Level 1 Controller (TPU Controller) to initialize it for computation.

During S11 and S12, if the TPU_Computation_finished_signal—generated by the TPU Controller—is not yet asserted (= 1), the controller continues to assert the tiled_computing_sig to indicate whether the operation should follow the tiled matrix multiplication path. Once the TPU finishes all matrix multiplication operations and the TPU_Computation_finished_signal is set to 1, the TMMU Controller increments the L2IR_counter to fetch the next instruction for decoding.

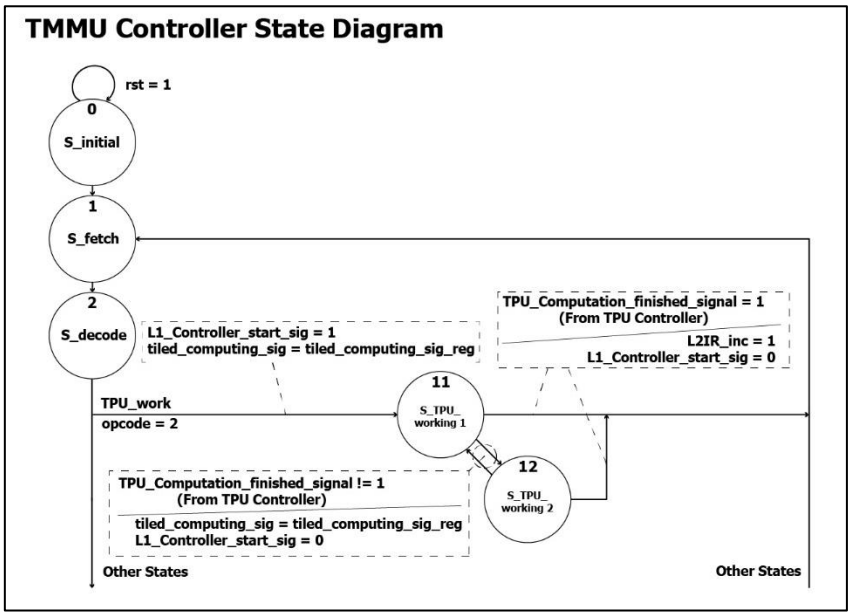


Figure 3.17: I/O Signals for TPU Computation Control States (S11–S12).

Figure 3.18 shows the I/O signals used in the remaining states (S13–S16). If the opcode (IR[15:12]) equals 0011, the controller initiates the ‘Write L2 Host Memory’ operation to transfer computed activations from the Level 1 Host Memory back to the Level 2 Host Memory. During this process, the Level 1 Host Memory performs read operations, while the Level 2 Host Memory executes write operations. Once the L1HM_to_L2HM_wr_counter exceeds 3 (indicating that all 64 data elements have been successfully transferred), the controller moves to state S15 to introduce a one-cycle delay, ensuring that all data is correctly written to memory. Afterward, the L2IR_counter is incremented to fetch the next instruction.

If the opcode is invalid, the TMMU Controller transitions to the stay state (S16), where it remains while waiting for future instructions to arrive.

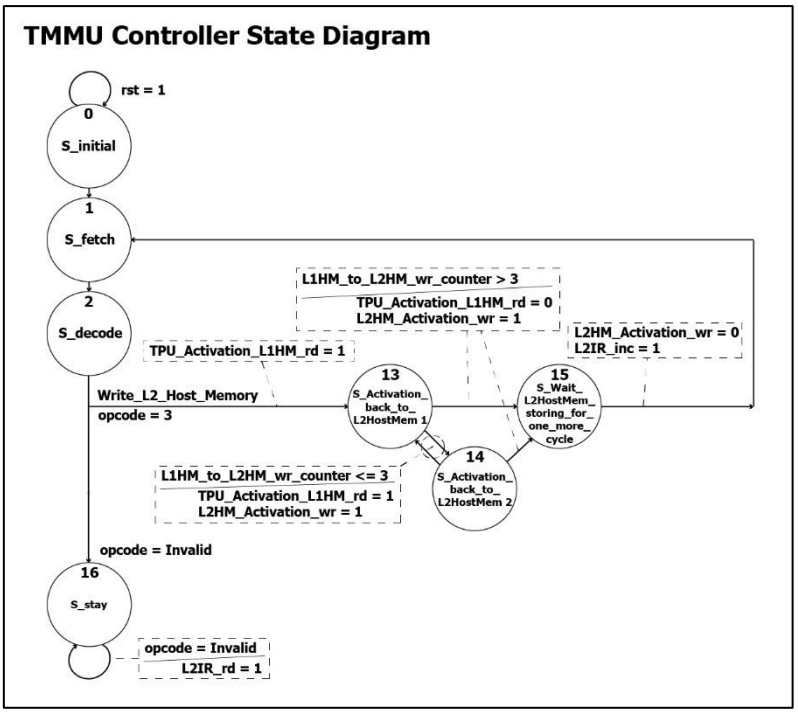


Figure 3.18: I/O Signals for Writeback and Wait States (S13–S16).

3.3.2 Level 1 Controller: TPU Control Logic

Once the TPU_Computation_finished_signal is asserted— as described in Figure 3.17— the TPU Controller is initialized and begins computing the matrix multiplication. When all results are computed and stored back into the Level 1 Host Memory, the TPU Controller sets TPU_Computation_finished_signal to 1, signaling the TMMU Controller to proceed with the remaining operations.

Figure 3.19 presents the overall state diagram of the Level 1 Controller. As shown in the figure, the controller operates through 35 distinct states. It is important to note that this diagram represents the control flow for non-tiled matrix multiplication; a detailed explanation of the tiled matrix multiplication state diagram will be presented in the following section. Table 3.4 provides the full names and descriptions of each abbreviated state for reference and clarity.

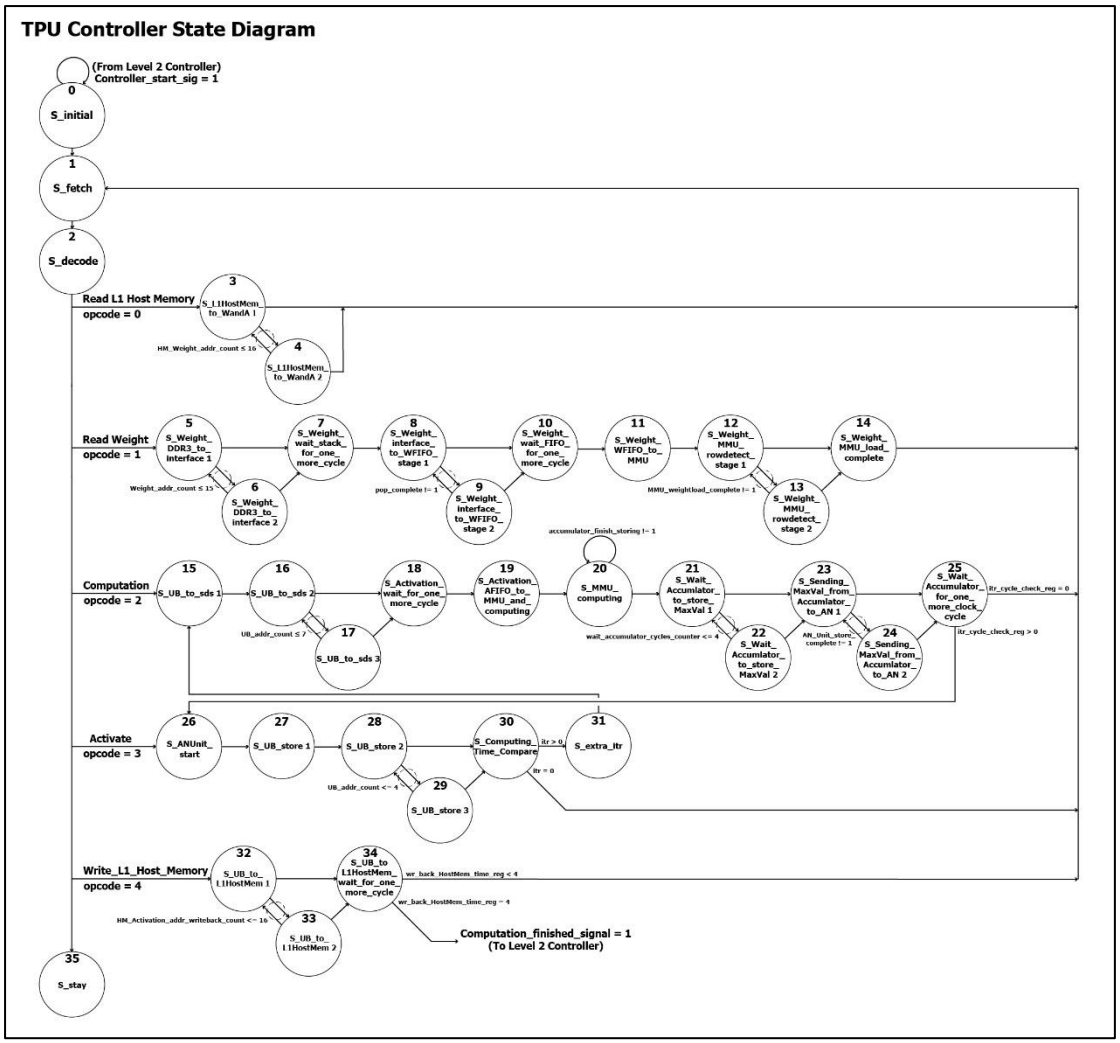


Figure 3.19: Level 1 Controller State Diagram.

0	*S_initial*	State_Initial
The initial state, which sends reset signals to the corresponding components.		
1	*S_fetch*	State_Fetch
The state that fetches an instruction from the IR Memory.		
2	*S_decode*	State_Decode
The state that decodes the instruction and determines the next state based on the result.		
3	*S_L1HostMem_to_WandA1*	State_Level 1 Host Memory_to_Weight DDR3 and Unified Buffer (Activation)_state1
The first state that loads weights and activations from the corresponding Level 1 Host Memory into the Weight DDR3 (for weights) and the Unified Buffer (for activations).		
4	*S_L1HostMem_to_WandA2*	State_Level 1 Host Memory_to_Weight DDR3 and Unified Buffer (Activation)_state2
The second state that loads weights and activations from the corresponding Level 1 Host Memory into the Weight DDR3 (for weights) and the Unified Buffer (for activations).		

5	*S_Weight_DDR3_to_interface1*	State_Weight DDR3_to_Weight Interface_state1
The first state that loads weights from the Weight DDR3 into the stack within the Weight Interface.		
6	*S_Weight_DDR3_to_interface2*	State_Weight DDR3_to_Weight Interface_state2
The second state that loads weights from the Weight DDR3 into the stack within the Weight Interface.		
7	*S_Weight_wait_stack_for_one_more_cycle*	State_Weight_wait_stack_for_one_more_cycle
The state that waits one additional cycle to ensure the stack within the Weight Interface correctly stores the loaded weights.		
8	*S_Weight_interface_to_WFIFO_stage1*	State_Weight Interface_to_Weight FIFOs_state1
The first state that transfers weights from the stack within the Weight Interface to the Weight FIFOs.		
9	*S_Weight_interface_to_WFIFO_stage2*	State_Weight Interface_to_Weight FIFOs_state2
The second state that transfers weights from the stack within the Weight Interface to the Weight FIFOs.		
10	*S_Weight_wait_FIFO_for_one_more_cycle*	State_Weight_wait_Weight FIFOs_for_one_more_cycle
The state that waits one additional cycle to ensure the Weight FIFOs correctly stores the loaded weights.		
11	*S_Weight_WFIFO_to_MMU*	State_Weight_Weight FIFOs_to_MMU
The state that loads weights from the Weight FIFOs into the registers of each systolic cell within the MMU.		
12	*S_Weight_MMU_rowdetect_stage1*	State_Weight_MMU_rowdetect_stage 1
The first state that enables each systolic cell to detect its row position and determine whether the incoming weight should be stored locally or forwarded to the next cell.		
13	*S_Weight_MMU_rowdetect_stage2*	State_Weight_MMU_rowdetect_stage 2
The second state that enables each systolic cell to detect its row position and determine whether the incoming weight should be stored locally or forwarded to the next cell.		
14	*S_Weight_MMU_load_complete*	State_Weight_MMU_load_complete
The state that signals to the controller that all weights have been successfully loaded into the MMU.		
15	*S_UB_to_sds1*	State_Unified Buffer_to_Systolic Data Setup Unit_state 1
The first state that transfers activations from the Unified Buffer, through the Systolic Data Setup Unit, into the Activation FIFOs.		
16	*S_UB_to_sds2*	State_Unified Buffer_to_Systolic Data Setup Unit_state 2

The second state that transfers activations from the Unified Buffer, through the Systolic Data Setup Unit, into the Activation FIFOs.		
17	*S_UB_to_sds3*	State_Unified Buffer_to_Systolic Data Setup Unit_state 3
The third state that transfers activations from the Unified Buffer, through the Systolic Data Setup Unit, into the Activation FIFOs.		
18	*S_Activation_Wait_for_one_more_cycle*	State_Activation_wait_for_one_ more_cycle
The state that waits one additional cycle to ensure the Activation FIFOs correctly stores the loaded activations.		
19	*S_Activation_AFIFO_to_MMU_and_computing*	State_Activation_Activation FIFOs_to_MMU_and_computing
The state that triggers the MMU to begin matrix multiplication using the incoming activations and load the results into the Accumulator.		
20	*S_MMU_computing*	State_MMU_computing
The state that keeps the MMU running and loads the results into the Accumulator. Computation is completed when the Accumulator sends a finished signal.		
21	*S_Wait_Accumlator_to_store_MaxVal1*	State_wait_Accumulator_to_store_ Maximum Values_state 1
The first state in which the Accumulator computes and stores the maximum values, both vector-wise and matrix-wise, based on the stored results.		
22	*S_Wait_Accumlator_to_store_MaxVal2*	State_wait_Accumulator_to_store_ Maximum Values_state 2
The second state in which the Accumulator computes and stores the maximum values, both vector-wise and matrix-wise, based on the stored results.		
23	*S_Sending_MaxVal_from_Accumlator_to_AN1*	State_sending_Maximum Values from_Accumulator_to_Activation Normalization Unit_state1
The first state that transfers the computed results and their corresponding maximum values from the Accumulator to the Activation Normalization Unit.		
24	*S_Sending_MaxVal_from_Accumlator_to_AN2*	State_sending_Maximum Values from_Accumulator_to_Activation Normalization Unit_state2
The second state that transfers the computed results and their corresponding maximum values from the Accumulator to the Activation Normalization Unit.		
25	*S_Wait_Accumulator_for_one_more_clock_cycle*	State_wait_Accumulator_for_one_ more_clock_cycle
The state that waits for one additional clock cycle to reset the related components.		

26	*S_ANUnit_start*	State_Activation Normalization Unit_start
The state that triggers the Activation Normalization Unit to start processing, including performing activation functions and rounding the results.		
27	*S_UB_store1*	State_Unified Buffer_store_state 1
The first state loads the rounded results from the Activation Normalization Unit into the Unified Buffer.		
28	*S_UB_store2*	State_Unified Buffer_store_state 2
The second state loads the rounded results from the Activation Normalization Unit into the Unified Buffer.		
29	*S_UB_store3*	State_Unified Buffer_store_state 3
The third state loads the rounded results from the Activation Normalization Unit into the Unified Buffer.		
30	*S_Computing_Time_Compare*	State_Computing_Time_Compare
The state that verifies if multiple iterations of the computation are required.		
31	*S_extra_itr*	State_extra_iteration
The state that decrements the iteration count and transitions to S_UB_to_SDS1 to perform the next computation.		
32	*S_UB_to_L1HostMem1*	State_Unified Buffer_to_Level 1 Host Memory_state 1
The first state that transfers the final results from the Unified Buffer to the Level 1 Host Memory.		
33	*S_UB_to_L1HostMem2*	State_Unified Buffer_to_Level 1 Host Memory_state 2
The second state that transfers the final results from the Unified Buffer to the Level 1 Host Memory.		
34	*S_UB_to_L1HostMem_wait_for_one_more_cycle*	State_Unified Buffer_to_Level 1 Host Memory_wait_for_one_more_cycle
The state that waits one additional cycle to ensure the Level 1 Host Memory correctly stores the final results, and then asserts the Computation_finished_signal to inform the TMMU Controller that all computations are complete.		
35	*S_stay*	State_stay
The state that remains idle while awaiting the next instruction.		

Table 3.4: State Name Descriptions for Level 1 Controller FSM.

The remainder of this section details each state of the TPU Controller, along with the corresponding I/O signals asserted during operation. Since the TPU in this project supports both basic matrix multiplication and tiled matrix multiplication, the controller's state diagrams are divided into two parts to clearly illustrate how each

signal behaves in different computation modes. The section begins by introducing the FSM for basic matrix multiplication, followed by the FSM for tiled matrix multiplication. This breakdown provides a clearer view of signal activity across states, offering insight into the dataflow and identifying opportunities for potential optimization.

Basic Matrix Multiplication:

Figure 3.20 illustrates the I/O signals used in states S0 through S4. In the initial state S0, all reset signals for the relevant components are asserted (set to 1) to ensure proper initialization. In state S1 (Fetch), the signal IR_rd is asserted to enable the Instruction Register to read an instruction for decoding. In state S2 (Decode), if the opcode (IR[15:12]) equals 0000, the controller initiates the 'Read L1 Host Memory' operation to transfer weights and activations from the Level 1 Host Memories to the Weight DDR3 (for weights) and the Unified Buffer (for activations).

During states S3 and S4, the controller checks the counter HM_Weight_addr_count to determine whether the data has been fully loaded into the target units. If the counter is less than or equal to 16, the controller continues toggling between S3 and S4 to load data. In these states, the Level 1 Host Memories perform read operations, while the Weight DDR3 and Unified Buffer perform write operations. The destination addresses Weight_DDR_addr and UB_addr are calculated using two internal counters: HM_Weight_addr_count and HM_Activation_addr_count.

Once the counter exceeds 16—indicating that all 16 values for both weights and activations have been successfully loaded—all memory operations stop, and both counters are reset. The controller then increments the IR_counter to fetch the next instruction for decoding.

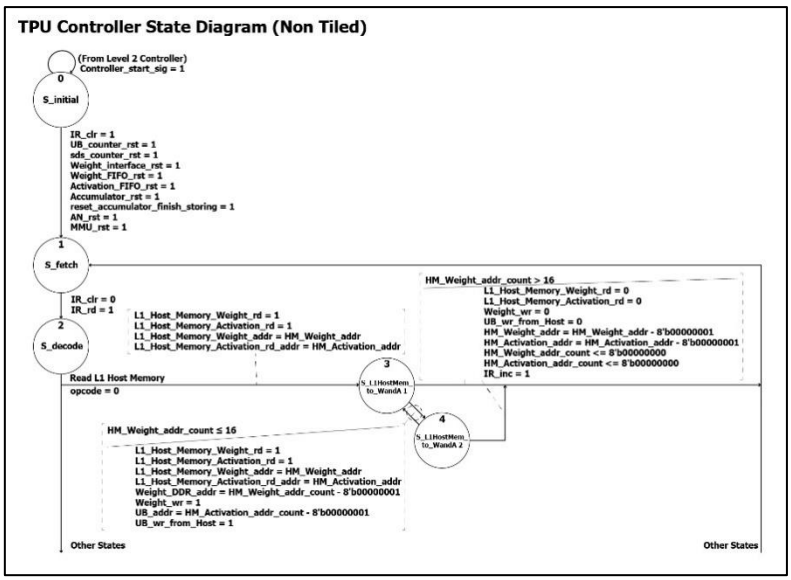


Figure 3.20: I/O Signals for Initialization and L1 Memory Read States (S0–S4).

Figure 3.21 shows the I/O signals used in states S5 through S14. When the opcode (IR[15:12]) equals 0001, the controller initiates the ‘Read Weight’ operation, which transfers weights from the Weight DDR3 to the Weight Interface, then to the Weight FIFOs, and finally into the MMU.

Upon decoding the 'Read Weight' instruction, the controller first asserts the Weight_interface_pushtime signal to inform the stack within the Weight Interface how many cycles it should push incoming weights. While the Weight_addr_count is less than or equal to 15, weights are continuously pushed from the Weight DDR3 into the stack, alternating between states S5 and S6. Once 16 weights are loaded, the controller transitions to state S7, which waits one additional clock cycle to ensure the weights are correctly stored in the stack.

Next, the controller asserts the Weight_interface_pop signal to begin popping weights from the stack into the corresponding Weight FIFOs. This pop operation also rearranges the data into the proper order. If the pop_complete signal from the Weight Interface remains low (0), the controller continues toggling between S8 and S9 until all weights are transferred. Once complete, the controller advances to state S10 for one extra cycle to confirm correct storage in the FIFOs.

From states S11 to S14, the controller enables read access on the Weight FIFOs and sends the MMU_w_pass signal to the MMU, activating weight-passing mode. The MMU uses four row detectors—one for each row in the 4×4 systolic cell array—to identify its row position and determine whether to forward the incoming weight to the

next cell or store it locally. If the MMU_weightload_complete signal remains low, the MMU continues loading weights from the FIFOs. Once the signal is asserted, indicating all weights have been loaded, the controller resets the FIFO pointer and increments the IR_counter to fetch the next instruction.

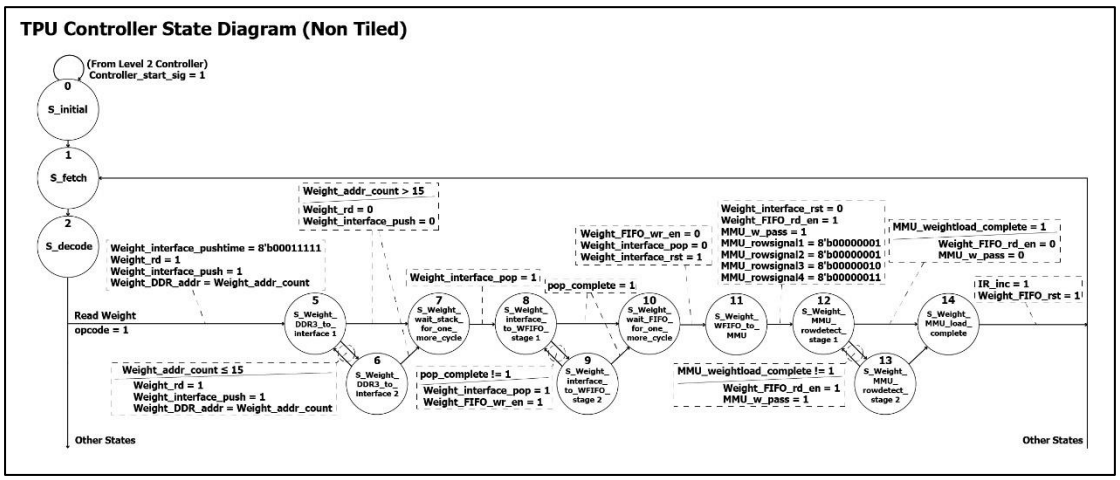


Figure 3.21: I/O Signals for Weight Transfer from DDR3 to MMU (S5–S14).

Figure 3.22 shows the I/O signals used in states S15 through S26. When the opcode IR[15:12] equals 0010, the controller initiates the 'Computation' operation, which transfers activations into the MMU and performs matrix multiplication using the weights previously stored in each systolic cell. The computed results are then stored in the Accumulator. After all values are computed, the Accumulator compares them to find the maximum values either in a vector-wise or matrix-wise fashion and sends this data to the Activation Normalization Unit.

From states S15 to S18, activations are transferred from the Unified Buffer (UB), through the Systolic Data Setup Unit (SDSU), and into the Activation FIFOs (AFIFOs) in preparation for computation. In S15, only the UB_rd signal is asserted, allowing the UB to perform a read operation—while the SDSU and AFIFOs remain inactive. Beginning from S16, the SDSU starts forwarding the incoming data by performing both read and write operations, and the AFIFOs begin writing the forwarded activations. As long as the internal counter UB_addr_count is less than or equal to 7, this process continues: the UB reads data, the SDSU forwards it, and the AFIFOs store it. Once all activation values are transferred, the controller moves to S18, allowing one additional clock cycle to ensure correct storage in the AFIFOs. During this state, the SDSU is also reset to prepare it for future use.

From S20, the MMU begins performing multiply-and-accumulate operations and sends the results to the Accumulator until the signal accumulator_finish_storing, generated

by the Accumulator, is asserted.

After the computation is complete, the Accumulator compares the results to determine maximum values across vectors or the entire matrix, depending on the operation. While the counter wait_accumulator_cycles_counter is less than or equal to 4, the Accumulator continues comparing and storing the maximum values, cycling through S21 and S22. Once this is complete, the controller transitions to S23 and S24, where it transfers both the computed results and the maximum values to the Activation Normalization Unit. This process continues until the signal AN_Unit_store_complete, generated by the AN_Unit, is asserted. At that point, all read and write enable signals are de-asserted, and the corresponding components are reset.

In S25, the controller checks the itr_cycle_check_reg register to determine whether additional iterations of the activation process are required. If the register value is equal to 1, the controller transitions to S26 to repeat the activation process; otherwise, the IR_counter is incremented to fetch the next instruction.

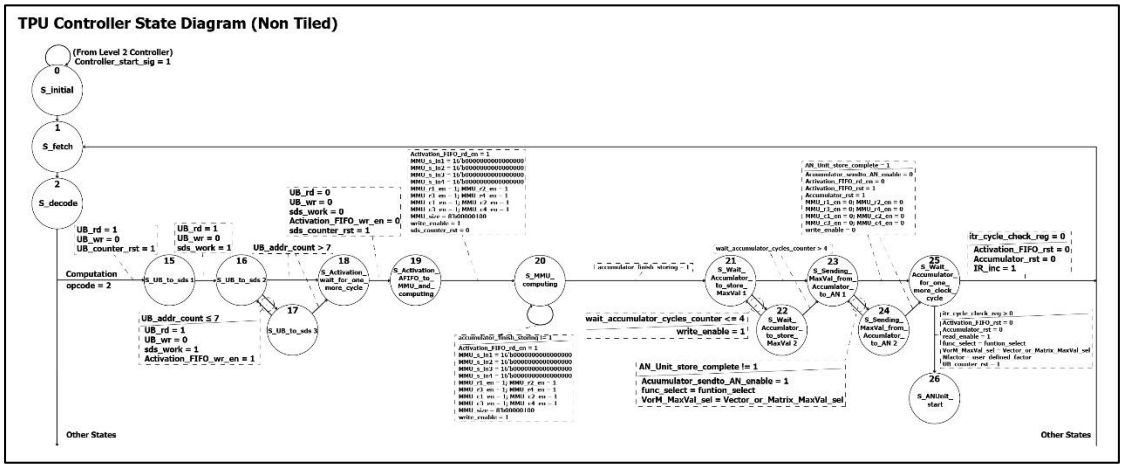


Figure 3.22: I/O Signals for Computation and Accumulation States (S15–S26).

Figure 3.23 shows the I/O signals used in states S26 through S31. When the opcode IR[15:12] equals 0011, the controller initiates the ‘Activation’ operation. During this process, the Activation Normalization Unit (AN_Unit) performs the selected activation function on the result values. These results are then scaled by multiplying with a user-defined factor and dividing by either the vector-wise or matrix-wise maximum value, depending on the V/M MaxVal bit (as described in Section 3.2). The final normalized results are written back to the Unified Buffer (UB) to update the activation values. If multiple iterations are required, the controller proceeds to the next round of computation directly—without reloading data from higher memory layers.

From S26 to S29, the AN_Unit and UB work in coordination to complete the activation process. In S26, only the AN_Unit is active, while UB write operations are disabled to prevent storing undefined (x) values. From S27 to S29, the AN_Unit continues processing, and the UB stores the resulting activation values into the appropriate memory locations. Once UB_addr_count exceeds 4, all operations are halted, and the controller transitions to S30 to check the iteration count. If the remaining iteration count is greater than zero, it is decremented, and the controller moves to S31, which allows the UB to read data for the next round before returning to S15. If the iteration count equals zero—indicating all computations are complete—the IR_counter is incremented to fetch the next instruction.

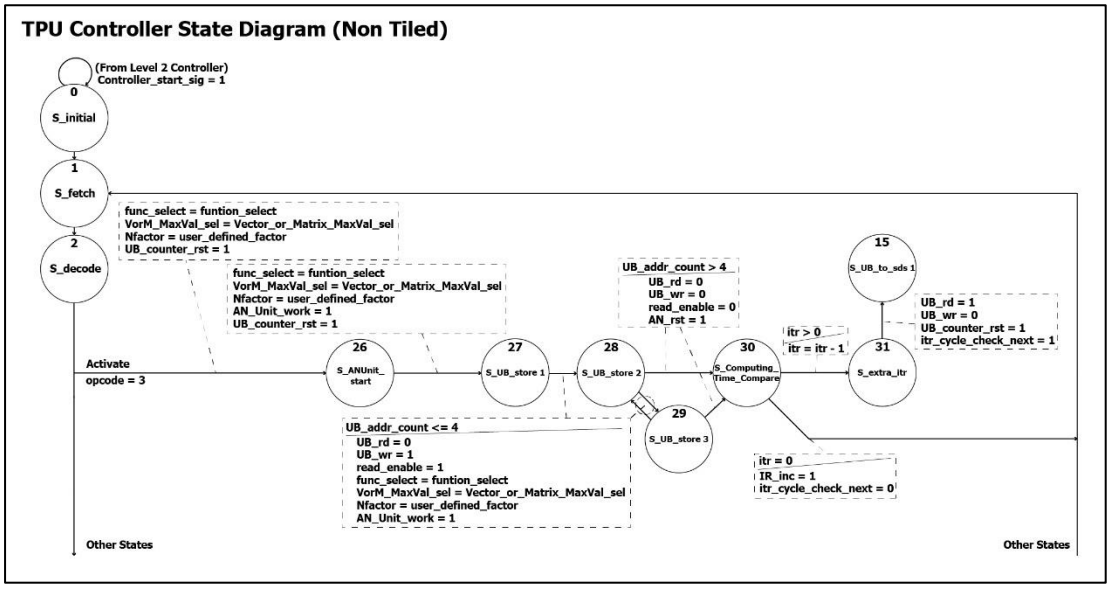


Figure 3.23: I/O Signals for Activation Processing and Iteration Control (S26–S31).

Figure 3.24 shows the I/O signals used in states S32 through S35. When the opcode IR[15:12] equals 0100, the controller initiates the 'Write L1 Host Memory' operation. Before entering S32, only the Unified Buffer (UB) is allowed to perform read operations, while the Level 1 Host Memory is prevented from writing to avoid storing undefined (x) values.

Between S32 and S33, if the counter HM_Activation_addr_writeback_count is less than or equal to 16, the Level 1 Host Memory continues writing the results, while the UB performs read operations. The write and read addresses are determined by L1_Host_Memory_Activation_wr_addr and UB_addr, respectively.

Once HM_Activation_addr_writeback_count exceeds 16, the controller transitions to S34 to wait for one additional clock cycle, ensuring that the results are properly stored

into the Level 1 Host Memory.

After S34, the controller evaluates the wr_back_HostMem_time_reg value. If this value is less than the configured threshold (e.g., 4 in this project), the IR_counter is incremented to fetch the next instruction. If wr_back_HostMem_time_reg equals 4, the output signal Computation_finished_signal is asserted and sent to the TMMU Controller to indicate that all computations have completed.

If the opcode is invalid, the TPU Controller enters the stay state (S35), where it idles while waiting for subsequent instructions.

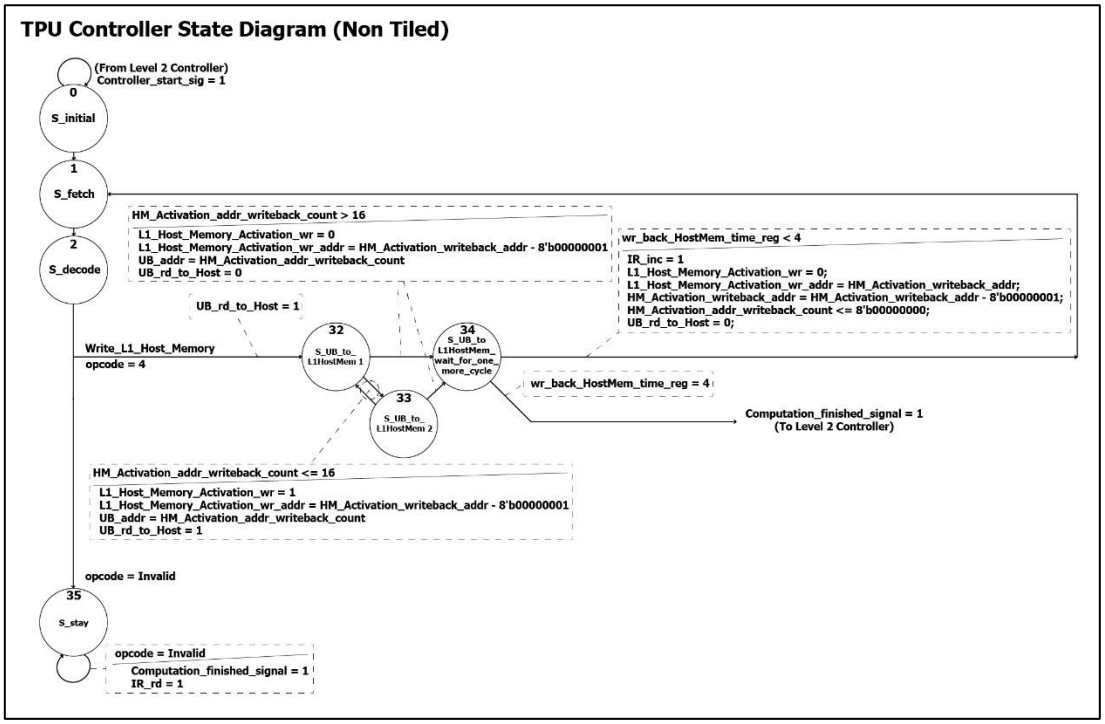


Figure 3.24: I/O Signals for Writeback and Finalization States (S32–S35).

Tiled Matrix Multiplication:

Since the TPU in this project also supports tiled matrix multiplication, the following section discusses the state diagram specific to this computation mode. While many operations remain the same as in basic matrix multiplication, states that involve operations unique to the tiled mode will be highlighted in blue. The other states, shown in black, are identical to those already described in the previous section.

Figure 3.25 shows the state diagram for states S0 to S4 under tiled matrix multiplication. As these states are responsible for loading values from the Level 1 Host Memories into Weight DDR3 (for weights) and the Unified Buffer (UB) (for activations), there is no

difference between tiled and non-tiled modes in this part of the process. Further details about these states have already been discussed in the basic matrix multiplication section.

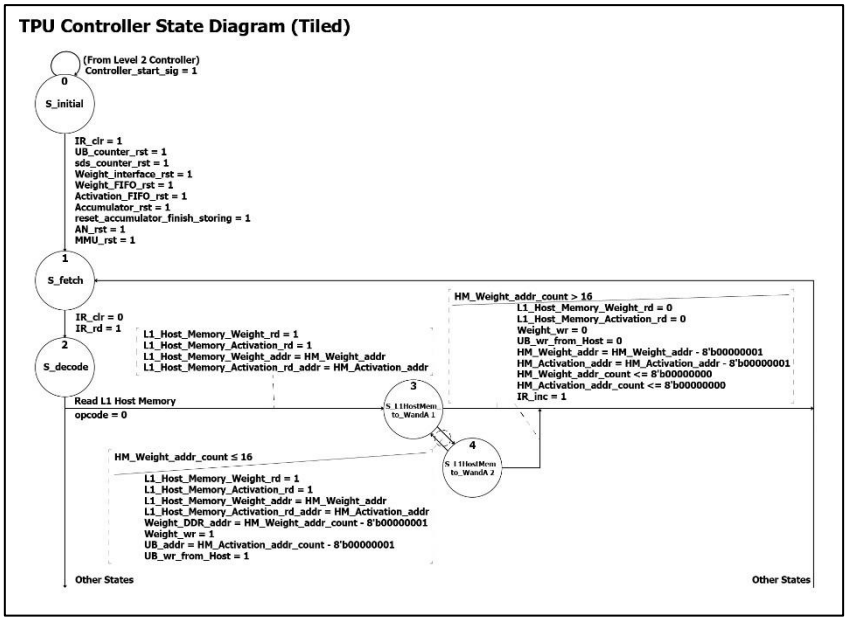


Figure 3.25: Initial States (S0–S4) in Tiled Matrix Multiplication Mode.

Figure 3.26 shows states S5 to S14 for the 'Read Weight' operation. In this case, there is no difference between the two modes—tiled and non-tiled—as the same weight values are loaded into the systolic cells of the MMU. Further details about these states have already been discussed in the basic matrix multiplication section.

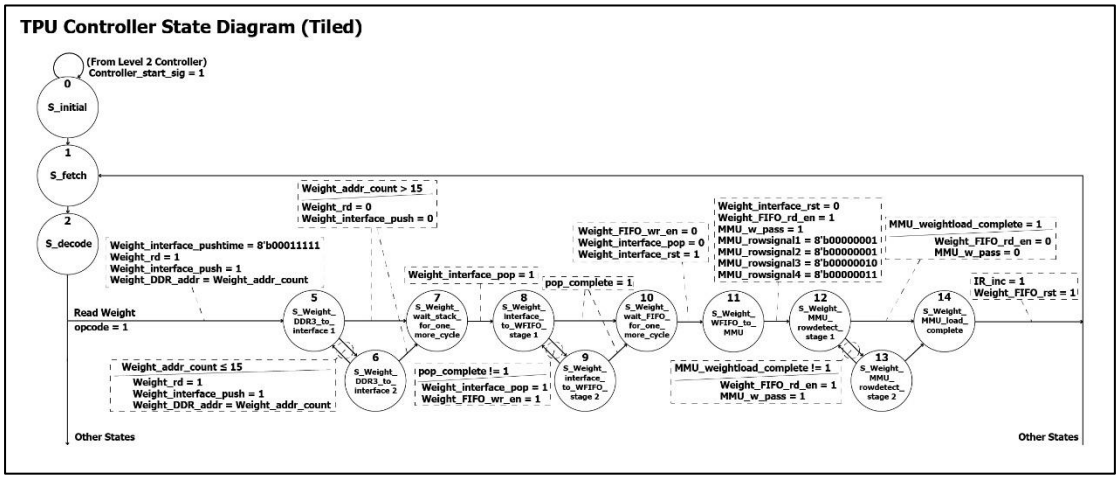


Figure 3.26: States S5–S14 for 'Read Weight' Operation in Tiled Matrix Multiplication Mode.

Figure 3.27 shows the 'Computation' operation for states S15 to S26. The major difference between basic and tiled matrix multiplication appears here. During state S20 (State_MMU_computing), tiled matrix multiplication requires the accumulation of

results from multiple matrix blocks (as discussed in Section 3.1). After completing the first computation, the controller cannot proceed directly to S21 to compare the maximum value. Instead, the intermediate results are stored in temporary memory within the Accumulator, which waits for additional block results to arrive and adds them together to form the final result.

Therefore, if the signal tiled_MM_storing_complete, generated by the Accumulator, has not yet been asserted, the controller will fetch the next instruction and perform the subsequent computation.

Once all computations are complete and the full 8×8 result matrix (64 values) is stored in the Accumulator, the controller transitions to S21 and S22 to perform a matrix-wise comparison to find the maximum value. In this case, the comparison counter must reach at least 16 to evaluate all data points. The remaining operations proceed the same way as in the basic matrix multiplication case, as described earlier.

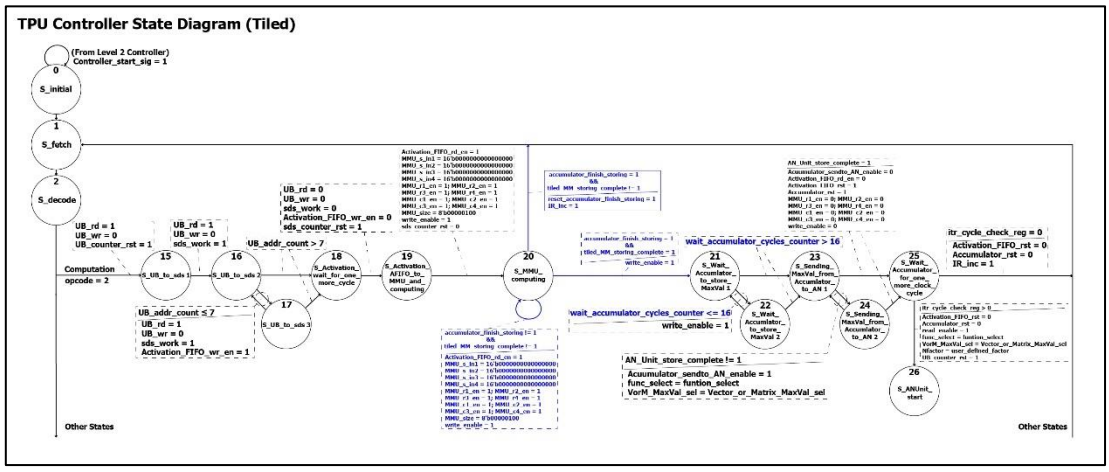


Figure 3.27: States S15–S26 for 'Computation' Operation in Tiled Matrix Multiplication Mode.

Figure 3.28 shows states S26 to S31 for the 'Activation' operation. In this case, there is no difference between the two modes—tiled and non-tiled—as the Activation Normalization Unit performs the activation function as well as the rounding operation. The Unified Buffer stores the resulting computed values. If additional iterations are required, the controller transitions back to S15 to repeat the computation; otherwise, it proceeds to fetch the next instruction. Further details about these states have already been discussed in the basic matrix multiplication section.

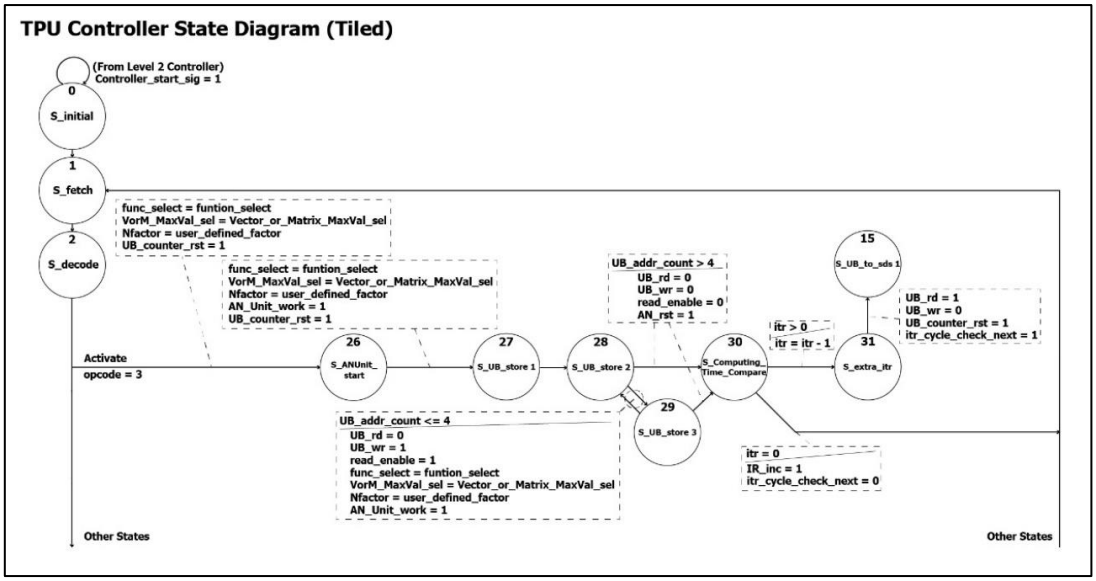


Figure 3.28: States S26–S31 for 'Activation' Operation in Tiled Matrix Multiplication Mode

Figure 3.29 shows states S32 to S35 for the 'Write L1 Host Memory' operation. In this case, there is no difference between the two modes—tiled and non-tiled—as the computed activations are transferred from the Unified Buffer to the Level 1 Host Memory. Since only 16 values can be transferred at a time, four iterations are required to store all 64 values. Therefore, if the wr_back_HostMem_time_reg is less than 4, the IR_counter is incremented to fetch the next instruction, typically another 'Write L1 Host Memory' operation. Once all 64 values are written back to the Level 1 Host Memory, the output signal Computation_finished_signal is asserted and sent to the TMMU Controller to indicate that all computations have been completed.

If the opcode is invalid, the TPU Controller transitions to the stay state (S35), where it idles while waiting for subsequent instructions.

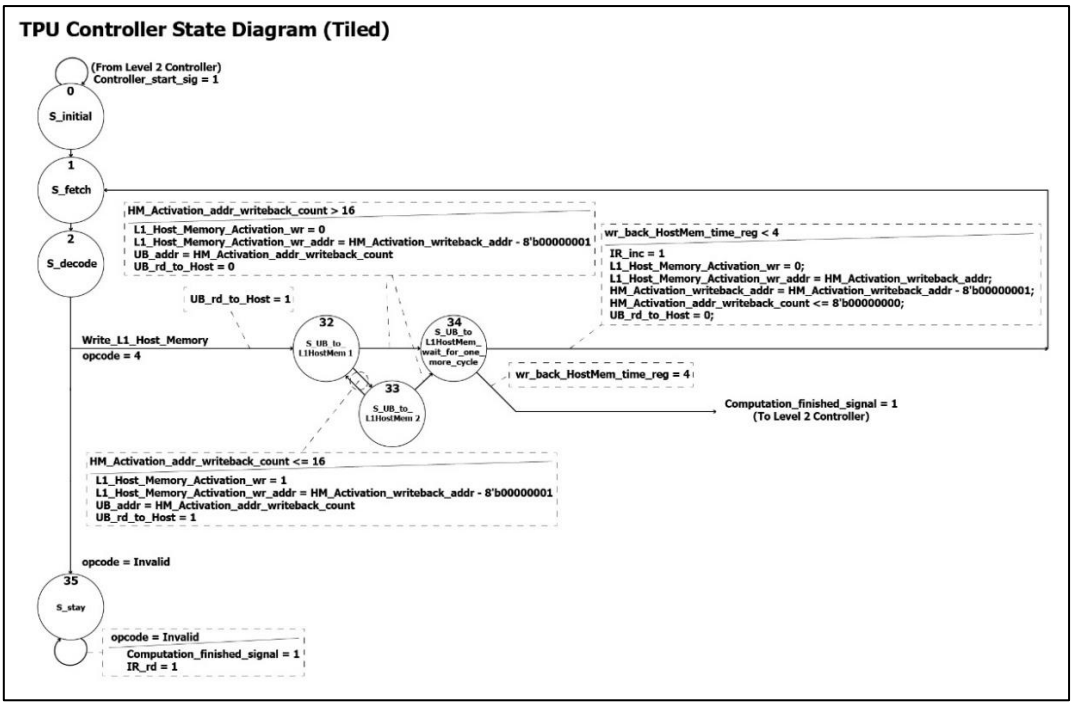


Figure 3.29: States S32–S35 for 'Write L1 Host Memory' Operation in Tiled Matrix Multiplication Mode

3.4 Summary

Chapter 3 detailed the control architecture and operational design of the Tiled Matrix Multiplication Unit (TMMU), a central component in this project's hardware implementation. The chapter began by introducing the tiled matrix multiplication strategy, a widely adopted method for dividing large matrix computations into smaller blocks. This approach enables efficient execution within constrained hardware environments by improving data locality and enabling scalable computation using a compact 4×4 Matrix Multiplication Unit.

To support this strategy, the chapter introduced a pair of custom CISC instruction sets—one for the TMMU controller and another for the embedded TPU controller. These instruction sets were designed to minimize instruction complexity while supporting flexible operation and system-level control. Each instruction is 16 bits wide, with key fields used to define matrix sizes, computation cycles, activation functions, and scaling behavior.

The chapter then explored the two-level controller hierarchy used in this project. The Level 2 Controller manages system-level instruction sequencing, memory coordination, and TPU activation, while the Level 1 Controller (TPU Controller) oversees fine-grained computation tasks such as data movement within the MMU and activation

processing. Finite state machines for both controllers were described in detail, including the signal behavior and dataflow patterns for both basic and tiled matrix multiplication modes.

Together, these components form the backbone of the TMMU architecture. The mechanisms introduced in this chapter lay the groundwork for the Verilog-based implementation discussed in the next chapter, where the RTL design and hardware mapping of the TMMU system are presented in full detail.

CHAPTER 4

TMMU: Structure and Implementation

Building upon the architectural concepts and control mechanisms discussed in the previous chapter, this chapter transitions into the structural and implementation aspects of the Tiled Matrix Multiplication Unit (TMMU). It begins with an overview of the complete TMMU system, presenting a high-level block diagram to illustrate the key modules and their interconnections. This provides a clear understanding of how the various components—such as memory units, controllers, data setup modules, and the TPU—are integrated into a cohesive architecture.

Following the structural overview, each major component within the TMMU is examined in detail. For every module, a functional block diagram is provided, and for some components, selected segments of the Verilog implementation are included to highlight important design decisions and internal logic. The complete Verilog code for all modules is made available in the supplementary section. This approach provides future learners and developers with a clear and practical foundation for designing similar hardware architectures and applying their own optimizations. By bridging high-level architectural concepts with real-world hardware implementation, this chapter connects system design theory with RTL development practice.

4.1 Overall TMMU Block Diagram

Figure 4.1 presents the overall block diagram of the TMMU on the left, with a detailed view of the internal TPU structure on the right. This system-level overview helps clarify the dataflow throughout the unit and illustrates how components are interconnected. Both controllers—Level 2 and Level 1—are linked to all relevant components within the architecture.

As shown in the figure, all values—including weights, activations, and instruction sets for both controllers—are received from higher levels of the memory hierarchy. In this

project, for simplicity of implementation, these values are pre-initialized into their respective Level 2 memory structures.

Weights and activations are first transferred from their respective Level 2 Host Memories through the Tiled Systolic Data Setup Units (TSDSUs), and then passed into their corresponding Level 1 Host Memories.

Weights are subsequently sent to the Weight DDR3 memory. When a 'Weight Load' operation is triggered, the data flows through the Weight Interface into the Weight FIFOs, and finally into the systolic cells of the Matrix Multiplication Unit (MMU). Activations, on the other hand, are sent to the Unified Buffer, then passed through the Systolic Data Setup Unit, through the Activation FIFOs, and into the MMU for computation. The resulting outputs are accumulated and stored in the Accumulator.

Once computation is complete, both the result matrix and the associated maximum values are forwarded to the Activation Normalization Unit (AN_Unit), where activation functions and rounding operations are performed.

After this processing step, the final activation values are returned to the Unified Buffer, then written back to the Level 1 Host Memory, and eventually to the Level 2 Host Memory for output. Since activations represent the computed outputs needed for further processing or inference, only they are sent back to the higher memory hierarchy. Weights and instruction data, which remain unchanged during computation, are not written back and are instead overwritten in subsequent operations.

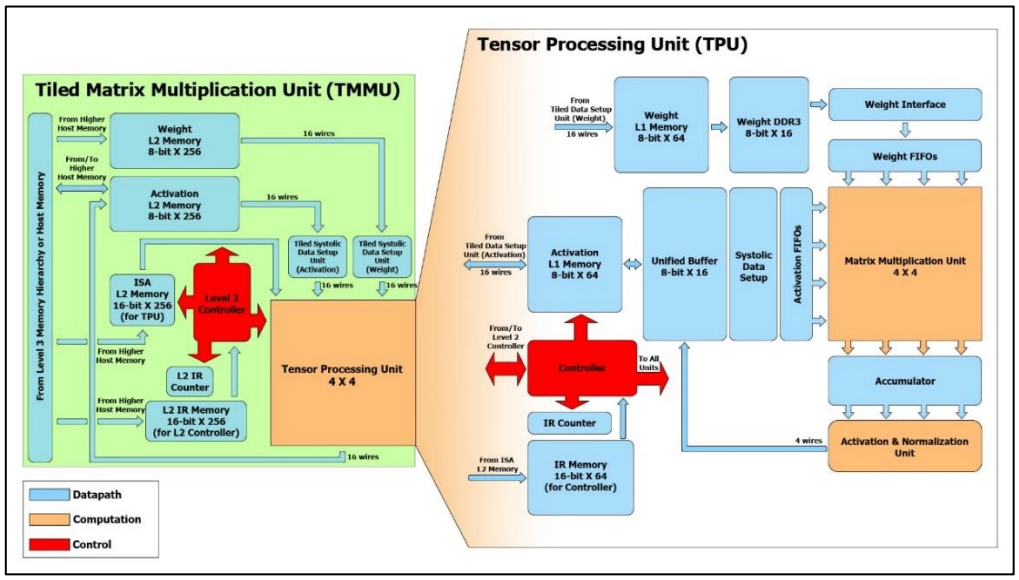


Figure 4.1: Overall Block Diagram of the TMMU and Internal TPU Structure.

4.2 Component Modules: Block Diagrams and Verilog Implementation

This section provides detailed explanations of each key hardware module within the TMMU architecture. For every major component, a functional block diagram is presented, accompanied by selected Verilog code segments that highlight important implementation details. The complete Verilog source code is provided in the supplementary section. This layered approach bridges the gap between architectural understanding and practical hardware design, offering both conceptual clarity and real-world insight.

The modules are grouped into two main categories: those that exist in the TMMU top-level (Level 2 layer), and those embedded within the TPU (Level 1 layer). Each module's functionality, interface signals, internal logic, and role within the overall dataflow are discussed. The provided Verilog code segments offer practical insights into how each component is structured, instantiated, and integrated into the broader system.

By examining the block diagrams alongside their implementation, this section serves as a valuable resource for understanding how high-level architectural concepts are translated into synthesizable RTL. This also provides guidance for future improvements, debugging, or further expansion of the TMMU design.

4.2.1 Modules in the TMMU Top-Level (Level 2 Layer)

This section will first introduce the components implemented at the Level 2 layer (the TMMU top-level), which manage instruction sequencing, data transfers, and communication between memory and the TPU. These modules play a critical role in coordinating high-level operations and preparing data for computation.

Figure 4.2 presents a partial Verilog code snippet highlighting key I/O signals, while Figure 4.3 illustrates the interconnection of selected components within the Level 2 layer (TMMU layer). The complete Verilog implementation of the TMMU can be found in Supplementary Figure 25.

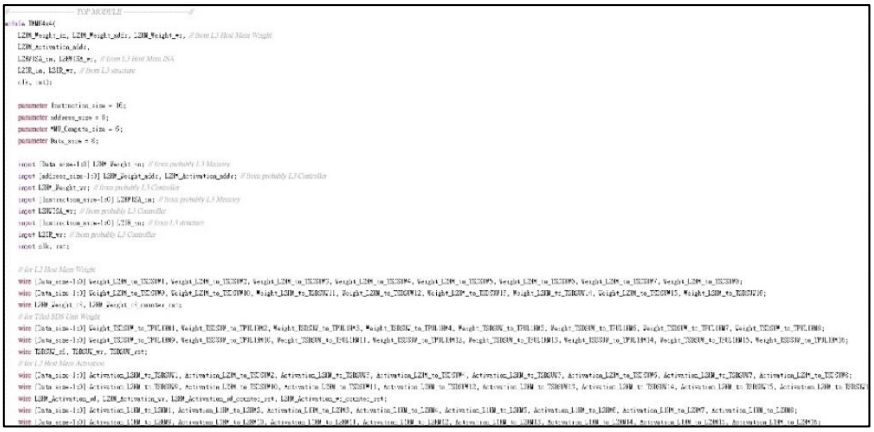


Figure 4.2: Partial Verilog Implementation of the TMMU Showing Key I/O Signal Definitions.

Figure 4.3: Interconnection Diagram of Core Components Within the Level 2 Layer of the TMMU.

L2 IR Memory:

The Level 2 IR Memory holds the instruction set that would typically be loaded from a higher-level host memory (e.g., Level 3) via the input signal W_data. However, for implementation simplicity, the instructions are pre-initialized directly into the memory. The TMMU Controller accesses instructions through the output signal R_data, with the addr input—driven by the L2 IR counter—specifying which instruction to retrieve. The IR_rd and IR_wr signals act as the read and write enables, respectively, allowing controlled access to the memory. The memory size is configured to 32 locations to align with floorplanning constraints.

Figure 4.4 shows the simplified block diagram of the L2_IR_Mem module. Figure 4.5 illustrates the Verilog code of the L2_IR_Mem module, including the pre-initialized instructions that are sent to the Level 2 Controller (TMMU Controller) for decoding. The complete code is available in Supplementary Figure 1.

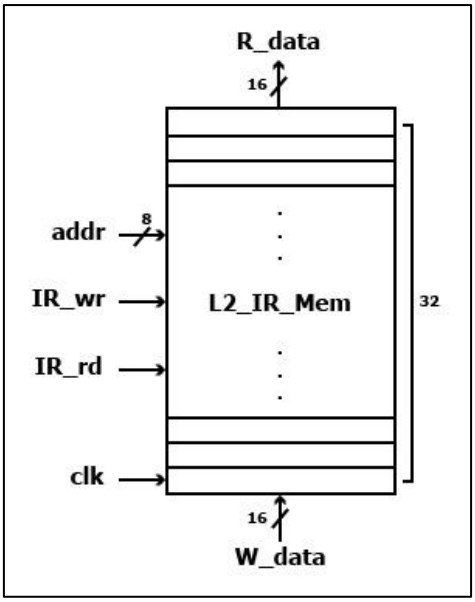


Figure 4.4: Block Diagram of L2_IR_Mem Module.

```
module L2_IR_Mem(R_data, W_data, addr, clk, IR_rd, IR_wr);
    parameter address_size = 8;
    parameter word_size = 16;
    parameter memory_size = 32;
    output [word_size-1:0] R_data;
    input [word_size-1:0] W_data;
    input [address_size-1:0] addr;
    input clk, IR_rd, IR_wr;
    reg [word_size-1:0] memory [memory_size-1:0];

    initial begin
        memory[0] = 16'b0000000000000000; // Load ISA
        memory[1] = 16'b0001000001001000; // Read L2 Host Memory
        memory[2] = 16'b0010000000000000; // TPU work
        memory[3] = 16'b0011000000000000; // Write L2 Host Memory
    end

    always @(posedge clk) begin
        if (IR_wr) begin
            memory[addr] <= W_data;
        end
    end

    assign R_data = memory[addr];

endmodule
```

Figure 4.5: Verilog Code of L2_IR_Mem Module with Pre-Initialized Instructions.

L2 IR Counter:

The L2 IR Counter determines which instruction is loaded into the TMMU Controller for decoding. When the IR_clr signal from the TMMU Controller is asserted (set to 1), the address signal IR_addr sent to the L2 IR Memory is reset to 0. Otherwise, the counter increments by 1 on each cycle, prompting the L2 IR Memory to load the next instruction in sequence. Figure 4.6 presents the simplified block diagram of the L2_IR_Counter module. The complete code is available in Supplementary Figure 2.

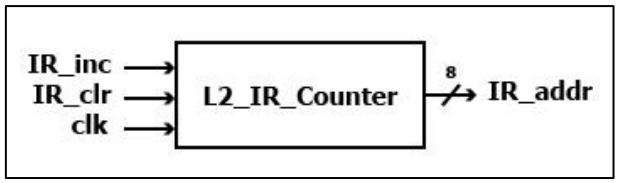


Figure 4.6: Block Diagram of the L2_IR_Counter Module.

L2 Host Memory Instruction Set:

The L2 Host Memory Instruction Set module stores instructions originating from the higher memory hierarchy, which are then forwarded to the IR_Mem within the TPU for decoding. These instructions determine the operations to be executed during various computation steps. The memory consists of 256 locations, each capable of storing a 16-bit instruction loaded via the W_data input. To address all memory locations, the addr signal is designed as 8 bits wide. When an instruction with the value 16'b1111111111111111 is encountered, the finished_sig output is asserted to notify the TMMU Controller that all instructions have been successfully loaded.

Figure 4.7 presents the simplified block diagram of the L2 Host Memory Instruction Set module. Figure 4.8 shows partial Verilog code for the module, including several pre-initialized instructions stored in memory. The complete code is available in Supplementary Figure 3, and the full pre-initialized instruction set for the 8×8 tiled matrix multiplication is shown in Figure 5.62.

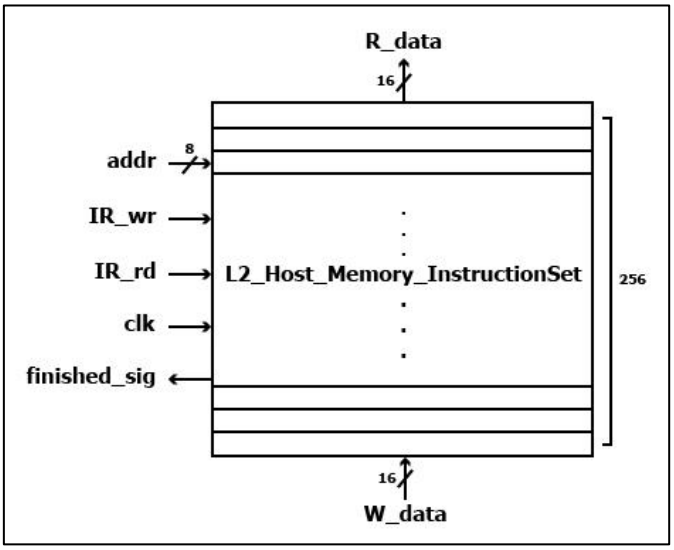
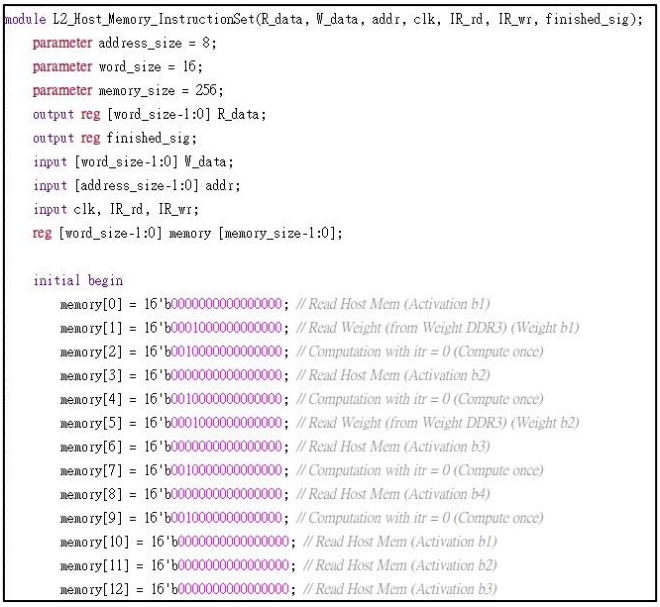


Figure 4.7: Block Diagram of the L2_Host_Memory_InstructionSet Module.



```
module L2_Host_Memory_InstructionSet(R_data, W_data, addr, clk, IR_rd, IR_wr, finished_sig);
    parameter address_size = 8;
    parameter word_size = 16;
    parameter memory_size = 256;
    output reg [word_size-1:0] R_data;
    output reg finished_sig;
    input [word_size-1:0] W_data;
    input [address_size-1:0] addr;
    input clk, IR_rd, IR_wr;
    reg [word_size-1:0] memory [memory_size-1:0];

    initial begin
        memory[0] = 16'b0000000000000000; // Read Host Mem (Activation b1)
        memory[1] = 16'b0001000000000000; // Read Weight (from Weight DDR3) (Weight b1)
        memory[2] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
        memory[3] = 16'b0000000000000000; // Read Host Mem (Activation b2)
        memory[4] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
        memory[5] = 16'b0001000000000000; // Read Weight (from Weight DDR3) (Weight b2)
        memory[6] = 16'b0000000000000000; // Read Host Mem (Activation b3)
        memory[7] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
        memory[8] = 16'b0000000000000000; // Read Host Mem (Activation b4)
        memory[9] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
        memory[10] = 16'b0000000000000000; // Read Host Mem (Activation b1)
        memory[11] = 16'b0000000000000000; // Read Host Mem (Activation b2)
        memory[12] = 16'b0000000000000000; // Read Host Mem (Activation b3)
```

Figure 4.8: Partial Verilog code of the L2 Host Memory Instruction Set.

L2 Host Memory Weight:

The L2 Host Memory Weight module stores weight values retrieved from the higher memory hierarchy, which are later transferred to the TPU for matrix multiplication. Since this project utilizes tiled matrix multiplication, the 8×8 matrix values stored in row-major order must be reordered by the Tiled Systolic Data Setup Unit (TSDSU). The 16 R_data outputs from this module are connected directly to the TSDSU to facilitate this reordering. The memory consists of 256 locations, with each value being 8 bits wide. To address all locations, the addr signal is also 8 bits wide.

Figure 4.9 presents the simplified block diagram of the L2 Host Memory Weight module. Figure 4.10 shows partial Verilog code for the module, including its I/O signals and a portion of the pre-initialized weight values. Figure 4.11 illustrates the read and write operations of the L2 Host Memory Weight module. The complete code is provided in Supplementary Figure 4.

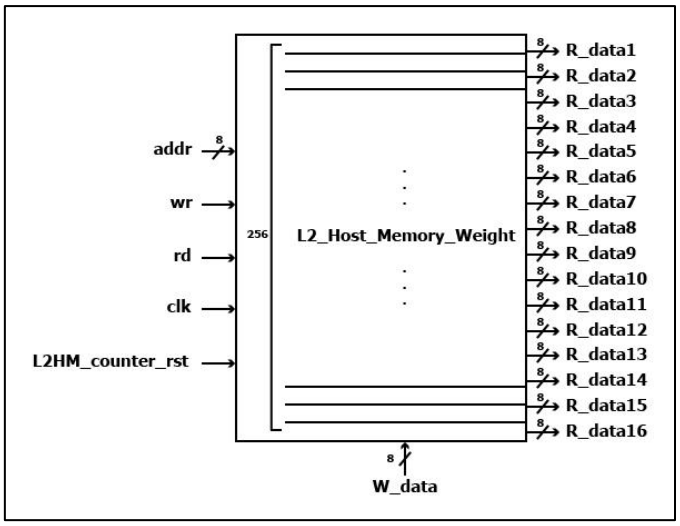


Figure 4.9: Block Diagram of the L2_Host_Memory_Weight Module.

```
module L2_Host_Memory_Weight(R_data1, R_data2, R_data3, R_data4, R_data5, R_data6, R_data7, R_data8,
    R_data9, R_data10, R_data11, R_data12, R_data13, R_data14, R_data15, R_data16,
    W_data, addr, clk, rd, wr,
    L2HM_counter_rst);
    parameter address_size = 8;
    parameter word_size = 8;
    parameter memory_size = 256;
    output reg [word_size-1:0] R_data1, R_data2, R_data3, R_data4, R_data5, R_data6, R_data7, R_data8; // read data to Tiled Systolic Data Setup Unit
    output reg [word_size-1:0] R_data9, R_data10, R_data11, R_data12, R_data13, R_data14, R_data15, R_data16; // read data to Tiled Systolic Data Setup Unit
    input [word_size-1:0] W_data; // write data from L2_Accumulator
    input [address_size-1:0] addr; // [7:0]
    input clk, rd, wr;
    input L2HM_counter_rst;
    reg [word_size-1:0] memory [memory_size-1:0];
    reg [word_size-1:0] rd_counter; // for propagating data purpose

    initial begin
        memory[0] = 8'b00000000; // 0
        memory[1] = 8'b00000001; // 1
        memory[2] = 8'b00000010; // 2
        memory[3] = 8'b00000011; // 3
        memory[4] = 8'b00000100; // 4
        memory[5] = 8'b00000101; // 5
        memory[6] = 8'b00000110; // 6
```

Figure 4.10: Partial Verilog Code of the L2 Host Memory Weight Module With I/O Definitions and Pre-Initialized Weights.

```
always @(posedge clk) begin
    if (wr) begin
        memory[addr] <= W_data; // Write data to memory at specified address
    end
end

always @(posedge clk) begin
    if (rd) begin
        case (rd_counter)
            0: begin
                R_data1  <= memory[0];
                R_data2  <= memory[1];
                R_data3  <= memory[2];
                R_data4  <= memory[3];
                R_data5  <= memory[4];
                R_data6  <= memory[5];
                R_data7  <= memory[6];
                R_data8  <= memory[7];
                R_data9  <= memory[8];
                R_data10 <= memory[9];
                R_data11 <= memory[10];
                R_data12 <= memory[11];
                R_data13 <= memory[12];
                R_data14 <= memory[13];
                R_data15 <= memory[14];
```

Figure 4.11: Read and Write Operations for the L2 Host Memory Weight Module.

L2 Host Memory Activation:

The L2 Host Memory Activation module stores activation values retrieved from the higher memory hierarchy, which are later transferred to the TPU for matrix multiplication. The memory contains 256 locations, each storing an 8-bit value. Accordingly, the addr signal is 8 bits wide to access the full memory range. Since this project employs tiled matrix multiplication, the 8×8 activation matrix—originally stored in row-major order—must be reordered by the Tiled Systolic Data Setup Unit (TSDSU). To support this, the module provides 16 R_data outputs directly connected to the TSDSU.

Additionally, the 16 W_data inputs come from the Level 1 Host Memory within the TPU to update the computed activation results. Two reset signals are used to reset the internal counters that support looping through memory locations. Figure 4.12 presents the simplified block diagram of the L2 Host Memory Activation module. Figure 4.13 shows the partial Verilog code for the L2 Host Memory Activation module, including the counters for read and write operations as well as a portion of the write logic. The complete code is provided in Supplementary Figure 5.

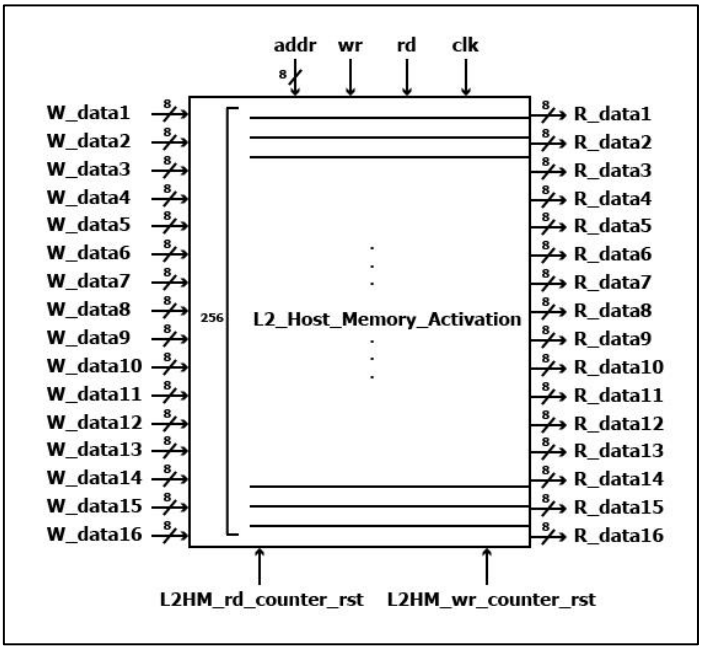


Figure 4.12: Block Diagram of the L2 Host Memory Activation Module.

```
always @(posedge clk or posedge L2HM_rd_counter_rst) begin
  if (L2HM_rd_counter_rst) begin
    rd_counter <= 0;
  end else if (rd) begin
    rd_counter <= rd_counter + 1;
    end
end

always @(posedge clk or posedge L2HM_wr_counter_rst) begin
  if (L2HM_wr_counter_rst) begin
    wr_counter <= 0;
  end else if (wr) begin
    wr_counter <= wr_counter + 1;
    end
end

always @(posedge clk) begin
    if (wr) begin
        case (wr_counter)
            0: begin
                memory[0]  <= W_data1;
                memory[1]  <= W_data2;
                memory[2]  <= W_data3;
                memory[3]  <= W_data4;
                memory[4]  <= W_data5;
                memory[5]  <= W_data6;
```

Figure 4.13: Partial Verilog Code for L2 Host Memory Activation Module with Read/Write Counters.

Tiled Systolic Data Setup Unit (Weight & Activation):

The Tiled Systolic Data Setup Unit (TSDSU) serves as a data-reordering module to enable tiled matrix multiplication. When the tiled_computing_sig is asserted, the 16 W_data input values are reordered based on the logic defined for tiled computation. If the signal is deasserted, the inputs are directly stored into the internal memory without reordering.

Prior to data input, the zerofill_sig is asserted to ensure that any sparse (unused) elements are filled with zeros, preventing undefined values (x) from propagating through the system. The 6-bit matrix_compute_size signal indicates the size of the matrix to be computed in the current round, guiding the reordering logic according to the patterns discussed in Section 3.1.3.

The 16 R_data outputs are then forwarded to both of the Level 1 Host Memory (Weight & Activation) to continue the dataflow for matrix multiplication. Figure 4.14 shows the simplified block diagram of the Tiled Systolic Data Setup Unit module.

Figure 4.15 shows the partial Verilog code for the Tiled Systolic Data Setup Unit module, including the zero-filling operation and part of the logic for tiled matrix multiplication when the computed matrix size equals 5. This illustrates the data storage approach. Figure 4.16 presents the partial code for the case when the matrix size equals

8. For basic matrix multiplication, the corresponding partial code is shown in Figure 4.17. The complete code is provided in Supplementary Figure 6.

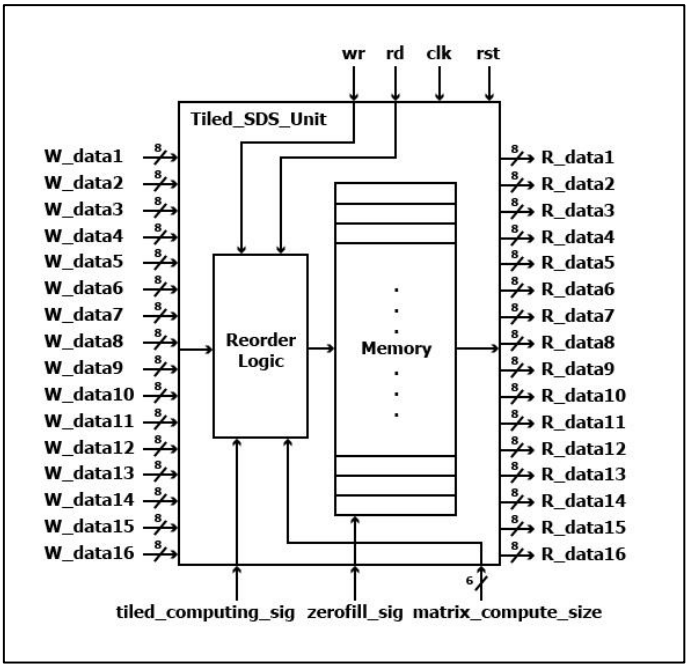


Figure 4.14: Block Diagram of the Tiled Systolic Data Setup Unit.

```
always @(posedge clk) begin
    if (zerofill_sig_reg) begin // Fill the memory with all 0's
        for (i = 0; i < 64; i = i + 1) begin
            memory[i] <= 8'b00000000;
        end
    end
    if (wr) begin
        if (tiled_computing_sig_reg) begin
            case (matrix_compute_size_reg) // The range should be 5x5 ~ 8x8
            5: begin
                    case (wr_counter)
                    0: begin
                        memory[0]  <= W_data1;
                        memory[1]  <= W_data2;
                        memory[2]  <= W_data3;
                        memory[3]  <= W_data4;
                        memory[32] <= W_data5;
                        memory[4]  <= W_data6;
                        memory[5]  <= W_data7;
                        memory[6]  <= W_data8;
                        memory[7]  <= W_data9;
                        memory[36] <= W_data10;
                        memory[8]  <= W_data11;
                        memory[9]  <= W_data12;
                        memory[10] <= W_data13;
                        memory[11] <= W_data14;
                        memory[40] <= W_data15;
                        memory[12] <= W_data16;
                    end
                    1: begin
                        memory[13] <= W_data1;
                        memory[14] <= W_data2;
                        memory[15] <= W_data3;
                        memory[44] <= W_data4;
                        memory[16] <= W_data5;
```

Figure 4.15: Partial Verilog Code for Tiled Systolic Data Setup Unit with Computed Matrix Size = 5.

```
8: begin
        case (wr_counter)
        0: begin
            memory[0]  <= W_data1;
            memory[1]  <= W_data2;
            memory[2]  <= W_data3;
            memory[3]  <= W_data4;
            memory[32]  <= W_data5;
            memory[33]  <= W_data6;
            memory[34]  <= W_data7;
            memory[35]  <= W_data8;
            memory[4]  <= W_data9;
            memory[5]  <= W_data10;
            memory[6]  <= W_data11;
            memory[7]  <= W_data12;
            memory[36]  <= W_data13;
            memory[37]  <= W_data14;
            memory[38]  <= W_data15;
            memory[39]  <= W_data16;
        end
        1: begin
            memory[8]  <= W_data1;
            memory[9]  <= W_data2;
            memory[10]  <= W_data3;
            memory[11]  <= W_data4;
            memory[40]  <= W_data5;
            memory[41]  <= W_data6;
            memory[42]  <= W_data7;
            memory[43]  <= W_data8;
            memory[12]  <= W_data9;
            memory[13]  <= W_data10;
            memory[14]  <= W_data11;
            memory[15]  <= W_data12;
            memory[44]  <= W_data13;
            memory[45]  <= W_data14;
            memory[46]  <= W_data15;
```

Figure 4.16: Partial Verilog Code for Tiled Systolic Data Setup Unit with Computed Matrix Size = 8.

```
end else begin // The matrix size that needs to be computed range from 1x1 ~ 4x4 (dont need tiled MM)
    case (matrix_compute_size_reg) // The range should be 1x1 ~ 4x4
        1: begin // 1x1 matrix multiplication
            case (wr_counter)
            0: begin
                memory[0]  <= W_data1;
            end
            1: begin
                memory[16]  <= W_data1;
            end
            2: begin
                memory[32]  <= W_data1;
            end
            3: begin
                memory[48]  <= W_data1;
            end
            default: ; // no operation
            endcase
        end
        2: begin // 2x2 matrix multiplication
            case (wr_counter)
            0: begin
                memory[0]  <= W_data1; // see the order send by FIFO
                memory[1]  <= W_data2;
                memory[4]  <= W_data3;
                memory[5]  <= W_data4;
            end
            1: begin
                memory[16]  <= W_data1;
                memory[17]  <= W_data2;
                memory[20]  <= W_data3;
                memory[21]  <= W_data4;
            end
            2: begin
                memory[32]  <= W_data1;
```

Figure 4.17: Partial Verilog Code for Basic Matrix Multiplication in TSDSU Module.

L2 Controller (TMMU Controller):

The Level 2 Controller, also referred to as the TMMU Controller, is responsible for handling all I/O signals associated with the components in the Level 2 layer of the architecture. The controller can be conceptually divided into two parts: a set of registers for internal counters and a main always block that governs state transitions within the control finite state machine (FSM).

The TMMU Controller fetches instructions from the L2 IR Memory, decodes them, and manages the operation of each component by asserting the appropriate control signals based on the current instruction and system state.

Figure 4.18 presents the simplified block diagram of the TMMU Controller module, where all relevant I/O signals are shown and mapped to the components they control. Figure 4.19 shows the partial Verilog code for the TMMU Controller, including the I/O signal definitions and the 16 states used for control transitions. The complete code is provided in Supplementary Figure 24.

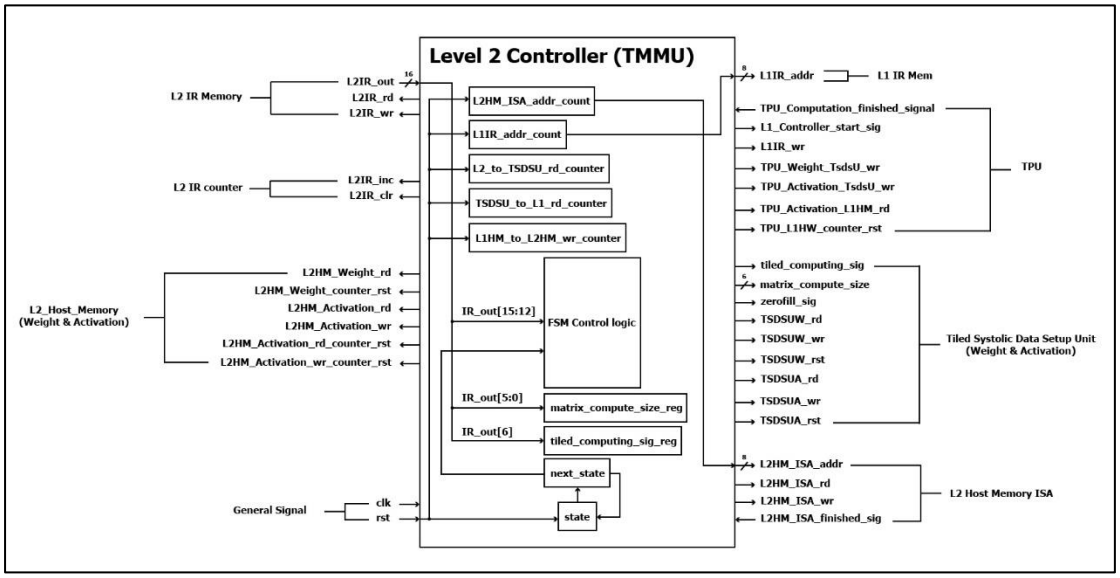
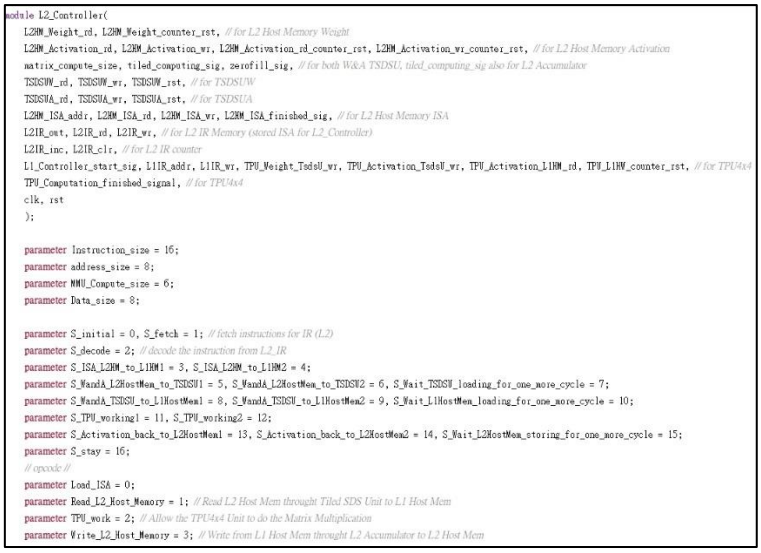


Figure 4.18: Block Diagram of the TMMU Controller Module



```
module L2_Controller(
    L2HM_Weight_rd, L2HM_Weight_counter_rst, // for L2 Host Memory Weight
    L2HM_Activation_rd, L2HM_Activation_wr, L2HM_Activation_rd_counter_rst, L2HM_Activation_wr_counter_rst, // for L2 Host Memory Activation
    matrix_compute_size, tiled_computing_sig, zerofill_sig, // for both W&A TSDSU, tiled_computing_sig also for L2 Accumulator
    TSDSUW_rd, TSDSUW_wr, TSDSUW_rst, // for TSDSUW
    TSDSUA_rd, TSDSUA_wr, TSDSUA_rst, // for TSDSUA
    L2HM_ISA_addr, L2HM_ISA_rd, L2HM_ISA_wr, L2HM_ISA_finished_sig, // for L2 Host Memory ISA
    L2IR_out, L2IR_rd, L2IR_wr, // for L2 IR Memory (stored ISA for L2_Controller)
    L2IR_inc, L2IR_clr, // for L2 IR counter
    L1_Controller_start_sig, L1IR_addr, L1IR_wr, TPU_Weight_TsdsU_wr, TPU_Activation_TsdsU_wr, TPU_Activation_L1HM_rd, TPU_L1HW_counter_rst, // for TPU4x4
    TPU_Computation_finished_signal, // for TPU4x4
    clk, rst
    );

    parameter Instruction_size = 16;
    parameter address_size = 8;
    parameter MMU_Compute_size = 6;
    parameter Data_size = 8;

    parameter S_initial = 0, S_fetch = 1; // fetch instructions for IR (L2)
    parameter S_decode = 2; // decode the instruction from L2_IR
    parameter S_ISA_L2HM_to_L1HM1 = 3, S_ISA_L2HM_to_L1HM2 = 4;
    parameter S_WandA_L2HostMem_to_TSDSU1 = 5, S_WandA_L2HostMem_to_TSDSU2 = 6, S_Wait_TSDSU_loading_for_one_more_cycle = 7;
    parameter S_WandA_TSDSU_to_L1HostMem1 = 8, S_WandA_TSDSU_to_L1HostMem2 = 9, S_Wait_L1HostMem_loading_for_one_more_cycle = 10;
    parameter S_TPU_working1 = 11, S_TPU_working2 = 12;
    parameter S_Activation_back_to_L2HostMem1 = 13, S_Activation_back_to_L2HostMem2 = 14, S_Wait_L2HostMem_storing_for_one_more_cycle = 15;
    parameter S_stay = 16;
    // opcode //
    parameter Load_ISA = 0;
    parameter Read_L2_Host_Memory = 1; // Read L2 Host Mem throught Tiled SDS Unit to L1 Host Mem
    parameter TPU_work = 2; // Allow the TPU4x4 Unit to do the Matrix Multiplication
    parameter Write_L2_Host_Memory = 3; // Write from L1 Host Mem throught L2 Accumulator to L2 Host Mem
```

Figure 4.19: Partial Verilog Code for TMMU Controller with I/O Signals and 16 Defined States.

4.2.2 Modules within the TPU (Level 1 Layer)

Next, after the weights and activations have been transferred into the Level 1 Host Memories within the TPU, the Level 2 Controller sends a start signal to reset and trigger the TPU Controller. This section introduces the components implemented at the Level 1 layer (TPU level), which handle instruction sequencing, internal data transfers, and communication between TPU modules. The core functionality of the TPU is to perform matrix multiplication on the provided data, making it the computational heart of the TMMU. Figure 4.20 shows the interconnection of selected components within the TPU layer. The complete Verilog implementation of the TPU can be found in Supplementary Figure 23.

```
//-------------------------IR related-------------------------//
IR_Mem IR_Mem1(.R_data(IR_out), .W_data(IR_in), .addr(IR_addr), .IR_wr_addr(IR_wr_addr), .clk(clk), .IR_rd(IR_rd), .IR_wr(IR_wr));
IR_counter IR_counter1(.IR_addr(IR_addr), .IR_inc(IR_inc), .IR_clr(IR_clr), .clk(clk));
//-------------------------Weight related-------------------------//
Weight_DDR3 WDDR3_1(.R_data(Weight_DDR3_to_Interface), .W_data(L1_Host_Memory_Weight_to_Weight_DDR3), .addr(Weight_DDR_addr), .clk(clk), .rd(Weight_rd), .wr(Weight_wr));
Weight_interface Winterface1(.clk(clk), .reset(Weight_interface_rst), .weight_in(Weight_DDR3_to_Interface), .pesh_time(Weight_interface_pesh_time), .pesh(Weight_interface_pesh), .pop(Weight_interface_pop), .pop_complete(pop_complete), .out1(Weight_In
Weight_FIFO WFIFO1(.clk(clk), .rst(Weight_FIFO_rst), .buf_in(Weight_Interface_to_FIFO1), .buf_out(Weight_FIFO_to_WMU1), .wr_en(Weight_FIFO_wr_en), .rd_en(Weight_FIFO_rd_en), .buf_empty(), .buf_full(), .fifo_counter());
Weight_FIFO WFIFO2(.clk(clk), .rst(Weight_FIFO_rst), .buf_in(Weight_Interface_to_FIFO2), .buf_out(Weight_FIFO_to_WMU2), .wr_en(Weight_FIFO_wr_en), .rd_en(Weight_FIFO_rd_en), .buf_empty(), .buf_full(), .fifo_counter());
Weight_FIFO WFIFO3(.clk(clk), .rst(Weight_FIFO_rst), .buf_in(Weight_Interface_to_FIFO3), .buf_out(Weight_FIFO_to_WMU3), .wr_en(Weight_FIFO_wr_en), .rd_en(Weight_FIFO_rd_en), .buf_empty(), .buf_full(), .fifo_counter());
Weight_FIFO WFIFO4(.clk(clk), .rst(Weight_FIFO_rst), .buf_in(Weight_Interface_to_FIFO4), .buf_out(Weight_FIFO_to_WMU4), .wr_en(Weight_FIFO_wr_en), .rd_en(Weight_FIFO_rd_en), .buf_empty(), .buf_full(), .fifo_counter());
//-------------------------MMU related-------------------------//
WMU4x4 WMU1(.a1(Activation_FIFO_to_WMU1), .a2(Activation_FIFO_to_WMU2), .a3(Activation_FIFO_to_WMU3), .a4(Activation_FIFO_to_WMU4), .w1(Weight_FIFO_to_WMU1), .w2(Weight_FIFO_to_WMU2), .w3(Weight_FIFO_to_WMU3), .w4(Weight_FIFO_to_WMU4),
.s_in1(WMU_s_in1), .s_in2(WMU_s_in2), .s_in3(WMU_s_in3), .s_in4(WMU_s_in4), .o1(WMU_to_Accumulator_out1), .o2(WMU_to_Accumulator_out2), .o3(WMU_to_Accumulator_out3), .o4(WMU_to_Accumulator_out4), .rst(WMU_rst), .clk(clk),
.w_pass(WMU_w_pass), .weightload_complete(WMU_weightload_complete), .r1_en(WMU_r1_en), .r2_en(WMU_r2_en), .r3_en(WMU_r3_en), .r4_en(WMU_r4_en), .c1_en(WMU_c1_en), .c2_en(WMU_c2_en), .c3_en(WMU_c3_en), .c4_en(WMU_c4_en),
.rowsignal1(WMU_rowsignal1), .rowsignal2(WMU_rowsignal2), .rowsignal3(WMU_rowsignal3), .rowsignal4(WMU_rowsignal4));
//-------------------------Other Computation related-------------------------//
//-------------------------before MMU-------------------------//
unified_buffer UB1(.addr(UB_addr), .clk(clk), .rd(UB_rd), .wr(UB_wr), .counter_rst(UB_counter_rst),
.Host_rd(UB_rd_to_Host), .Host_wr(UB_wr_from_Host), .Host_datain(L1_Host_Memory_Activation_to_Unified_Buffer), .Host_dataout(Unified_Buffer_to_L1_Host_Memory_Activation),
.data_in1(AM_to_UB1), .data_in2(AM_to_UB2), .data_in3(AM_to_UB3), .data_in4(AM_to_UB4),
.data_out1(UB_dataout1), .data_out2(UB_dataout2), .data_out3(UB_dataout3), .data_out4(UB_dataout4), .data_out5(UB_dataout5), .data_out6(UB_dataout6), .data_out7(UB_dataout7), .data_out8(UB_dataout8),
.data_out9(UB_dataout9), .data_out10(UB_dataout10), .data_out11(UB_dataout11), .data_out12(UB_dataout12), .data_out13(UB_dataout13), .data_out14(UB_dataout14), .data_out15(UB_dataout15), .data_out16(UB_dataout16));
sds4x4 sds1(.clk(clk), .counter_rst(sds_counter_rst), .sds_work(sds_work), .a0(UB_dataout1), .a1(UB_dataout2), .a2(UB_dataout3), .a3(UB_dataout4), .b0(UB_dataout5), .b1(UB_dataout6), .b2(UB_dataout7), .b3(UB_dataout8),
.c0(UB_dataout9), .c1(UB_dataout10), .c2(UB_dataout11), .c3(UB_dataout12), .d0(UB_dataout13), .d1(UB_dataout14), .d2(UB_dataout15), .d3(UB_dataout16), .output1(Activation_sds_to_AFIFO1), .output2(Activation_sds_to_AFIFO2), .output3(Activation_sds_to
Activation_FIFO AFIFO1(.clk(clk), .rst(Activation_FIFO_rst), .buf_in(Activation_sds_to_AFIFO1), .buf_out(Activation_FIFO_to_WMU1), .wr_en(Activation_FIFO_wr_en), .rd_en(Activation_FIFO_rd_en), .buf_empty(), .buf_full(), .fifo_counter());
Activation_FIFO AFIFO2(.clk(clk), .rst(Activation_FIFO_rst), .buf_in(Activation_sds_to_AFIFO2), .buf_out(Activation_FIFO_to_WMU2), .wr_en(Activation_FIFO_wr_en), .rd_en(Activation_FIFO_rd_en), .buf_empty(), .buf_full(), .fifo_counter());
Activation_FIFO AFIFO3(.clk(clk), .rst(Activation_FIFO_rst), .buf_in(Activation_sds_to_AFIFO3), .buf_out(Activation_FIFO_to_WMU3), .wr_en(Activation_FIFO_wr_en), .rd_en(Activation_FIFO_rd_en), .buf_empty(), .buf_full(), .fifo_counter());
Activation_FIFO AFIFO4(.clk(clk), .rst(Activation_FIFO_rst), .buf_in(Activation_sds_to_AFIFO4), .buf_out(Activation_FIFO_to_WMU4), .wr_en(Activation_FIFO_wr_en), .rd_en(Activation_FIFO_rd_en), .buf_empty(), .buf_full(), .fifo_counter());
//-------------------------After MMU-------------------------//
Accumulator4x4 Accumulator1(.MMU_size(WMU_size), .write_enable(write_enable), .read_enable(read_enable), .Accumulator_sendto_AM_enable(Accumulator_sendto_AM_enable),
.Vecwise_out1(Accumulator_to_AM_VecMaxVal1), .Vecwise_out2(Accumulator_to_AM_VecMaxVal2), .Vecwise_out3(Accumulator_to_AM_VecMaxVal3), .Vecwise_out4(Accumulator_to_AM_VecMaxVal4), .Matwise_out(Accumulator_to_AM_MatMaxVal),
.in1(WMU_to_Accumulator_out1), .in2(WMU_to_Accumulator_out2), .in3(WMU_to_Accumulator_out3), .in4(WMU_to_Accumulator_out4),
.out1(Accumulator_to_AM_out1), .out2(Accumulator_to_AM_out2), .out3(Accumulator_to_AM_out3), .out4(Accumulator_to_AM_out4),
.accumulator_finish_storing(accumulator_finish_storing), .reset_accumulator_finish_storing(reset_accumulator_finish_storing),
.tiled_computing_sig(tiled_computing_sig), .tiled_MM_storing_complete(tiled_MM_storing_complete),
```

Figure 4.20: Interconnection of Selected Components in the TPU Layer.

L1 Host Memory Weight:

The Level 1 Host Memory Weight module functions as a Level 1 memory component within the overall memory hierarchy. It receives 16 W_data input values from the Tiled Systolic Data Setup Unit (TSDSU), representing the weight values required for matrix multiplication. When the W_wr_from_TsdsU signal is asserted, these values are written into the internal memory.

The output signal R_data connects to the Weight DDR3 module and is used to read stored weight values when the rd signal is asserted. As the memory contains 64 locations, the addr signal is 6 bits wide to cover the full address space. An additional control signal, L1HW_counter_rst, serves as a reset for the internal counter, allowing address looping and reuse during repeated operations.

Since weight values do not require updates after computation, the Level 1 Host Memory Weight module includes fewer ports compared to the Level 1 Host Memory Activation module, which will be described in the following section.

Figure 4.21 presents the simplified block diagram of the Level 1 Host Memory Weight module. The complete Verilog implementation of the Level 1 Host Memory Weight can be found in Supplementary Figure 7.

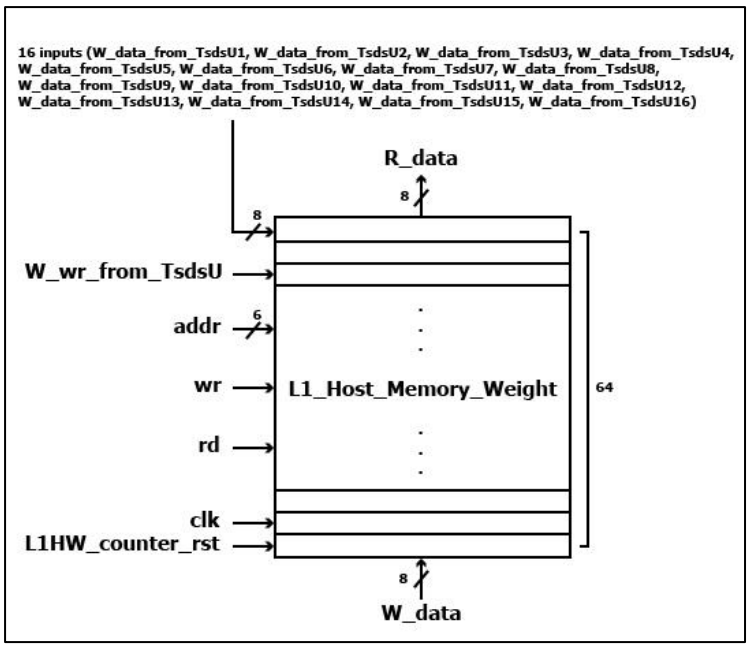


Figure 4.21: Block Diagram of the L1_Host_Memory_Weight Module.

L1 Host Memory Activation:

The Level 1 Host Memory Activation module functions as a Level 1 memory component within the overall memory hierarchy. It receives 16 W_data input values from the Tiled Systolic Data Setup Unit (TSDSU), which correspond to the activation values required for matrix multiplication. When the W_wr_from_TsdsU signal is asserted, these values are written into internal memory. Since the memory contains 64 locations, both rd_addr and wr_addr signals are 6 bits wide to fully address the memory space during read and write operations. The L1HW_counter_rst signal resets the internal address counter, enabling memory looping and efficient reuse.

The output signal R_data connects to the Unified Buffer module and is used to transfer stored activation values during the read phase when the rd signal is asserted. Once matrix multiplication is complete, the computed activation results are written back from the Unified Buffer to the L1 Host Memory Activation module when the wr signal is asserted. After all values are successfully stored, the A_rd_backto_L2HM signal is asserted, allowing the results to be read out through the 16 A_data output ports and sent back to the Level 2 Host Memory.

The tiled_computing_sig ensures that results from the Unified Buffer are written into the correct memory positions when tiled matrix multiplication is used. For basic matrix multiplication, results are written directly to memory without any reordering.

Figure 4.22 presents the simplified block diagram of the Level 1 Host Memory Activation module. Figure 4.23 shows the partial Verilog code for the module, including its I/O signals. The complete Verilog implementation can be found in Supplementary Figure 8.

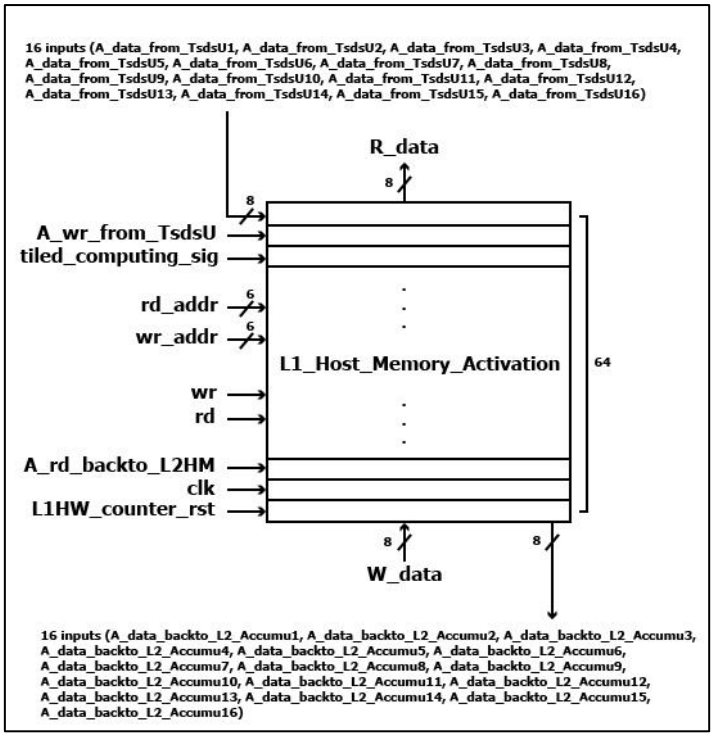


Figure 4.22: Block Diagram of the L1_Host_Memory_Activation Module.

```
module L1_Host_Memory_Activation(tiled_computing_sig, R_data, W_data, rd_addr, wr_addr, clk, rd, wr,
    A_data_from_TsdsU1, A_data_from_TsdsU2, A_data_from_TsdsU3, A_data_from_TsdsU4, A_data_from_TsdsU5, A_data_from_TsdsU6, A_data_from_TsdsU7, A_data_from_TsdsU8,
    A_data_from_TsdsU9, A_data_from_TsdsU10, A_data_from_TsdsU11, A_data_from_TsdsU12, A_data_from_TsdsU13, A_data_from_TsdsU14, A_data_from_TsdsU15, A_data_from_TsdsU16,
    A_wr_from_TsdsU, A_data_backto_L2_Accumu1, A_data_backto_L2_Accumu2, A_data_backto_L2_Accumu3, A_data_backto_L2_Accumu4, A_data_backto_L2_Accumu5, A_data_backto_L2_Accumu6, A_data_backto_L2_Accumu7, A_data_backto_L2_Accumu8,
    A_data_backto_L2_Accumu9, A_data_backto_L2_Accumu10, A_data_backto_L2_Accumu11, A_data_backto_L2_Accumu12, A_data_backto_L2_Accumu13, A_data_backto_L2_Accumu14, A_data_backto_L2_Accumu15, A_data_backto_L2_Accumu16,
    A_rd_backto_L2HM, L1HW_counter_rst
    );
    parameter address_size = 6;
    parameter word_size = 8;
    parameter memory_size = 64;
    input tiled_computing_sig;
    output reg [word_size-1:0] R_data; // read data to UB
    input [word_size-1:0] W_data; // write data from UB
    input [word_size-1:0] A_data_from_TsdsU1, A_data_from_TsdsU2, A_data_from_TsdsU3, A_data_from_TsdsU4, A_data_from_TsdsU5, A_data_from_TsdsU6, A_data_from_TsdsU7, A_data_from_TsdsU8; // write ACTIVATION data from Tiled SDS Unit to L1 Host Memory
    input [word_size-1:0] A_data_from_TsdsU9, A_data_from_TsdsU10, A_data_from_TsdsU11, A_data_from_TsdsU12, A_data_from_TsdsU13, A_data_from_TsdsU14, A_data_from_TsdsU15, A_data_from_TsdsU16; // write ACTIVATION data from Tiled SDS Unit to L1 Host Memory
    output reg [word_size-1:0] A_data_backto_L2_Accumu1, A_data_backto_L2_Accumu2, A_data_backto_L2_Accumu3, A_data_backto_L2_Accumu4, A_data_backto_L2_Accumu5, A_data_backto_L2_Accumu6, A_data_backto_L2_Accumu7, A_data_backto_L2_Accumu8; // read ACTIVA
    output reg [word_size-1:0] A_data_backto_L2_Accumu9, A_data_backto_L2_Accumu10, A_data_backto_L2_Accumu11, A_data_backto_L2_Accumu12, A_data_backto_L2_Accumu13, A_data_backto_L2_Accumu14, A_data_backto_L2_Accumu15, A_data_backto_L2_Accumu16; // read d
    input A_rd_backto_L2HM; // read enable to allow data read back to L2 Host Memory
    input A_wr_from_TsdsU; // write enable to allow data write from L2 Host Memory to L1 Host Memory
    input [address_size-1:0] rd_addr, wr_addr; // [7:0]
    input clk, rd, wr, L1HW_counter_rst;
    reg [word_size-1:0] memory [memory_size-1:0];
    reg [word_size-1:0] wr_counter, rd_counter; // for storing data purpose
```

Figure 4.23: Partial Verilog Code for Level 1 Host Memory Activation Module.

IR Memory:

The IR Memory stores the instruction set that is loaded from the L2 Host Memory Instruction Set via the W_data input when the IR_wr signal is asserted. The TPU Controller accesses instructions through the R_data output, with the addr input—driven by the IR counter—specifying which instruction to retrieve when the IR_rd signal is asserted. These control signals enable precise and controlled access to the memory.

The memory is configured with 64 locations, as tiled matrix multiplication requires more instructions than basic operations (which typically fit within 32 instructions). The address ports are designed with additional bits to support potential expansion, should

more instructions or memory locations be needed in future implementations.

Figure 4.24 shows the simplified block diagram of the IR_Mem module. The complete Verilog implementation can be found in Supplementary Figure 9.

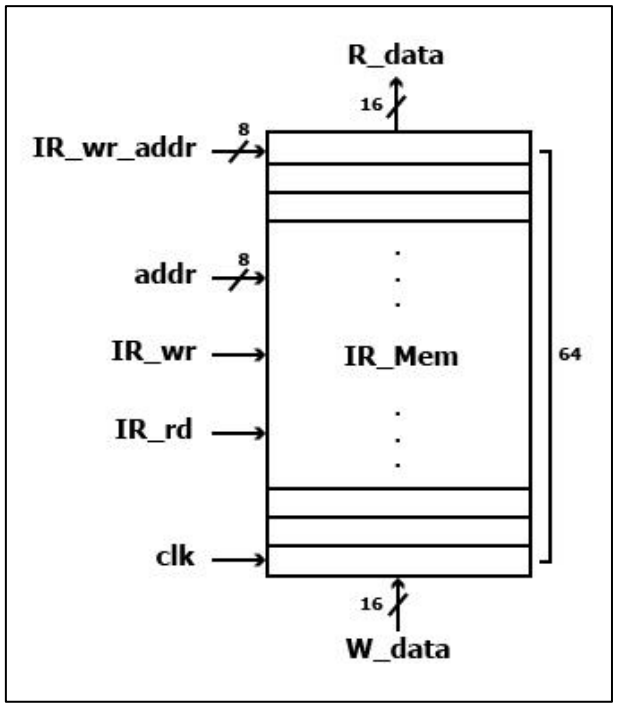


Figure 4.24: Block Diagram of the IR_Mem Module.

IR Counter:

The IR Counter determines which instruction is loaded into the TPU Controller for decoding. When the IR_clr signal from the TPU Controller is asserted (set to 1), the IR_addr signal sent to the IR Memory is reset to 0. Otherwise, the counter increments by 1 on each clock cycle, prompting the IR Memory to load the next instruction in sequence. Figure 4.25 presents the simplified block diagram of the IR_Counter module. The complete Verilog implementation can be found in Supplementary Figure 10.

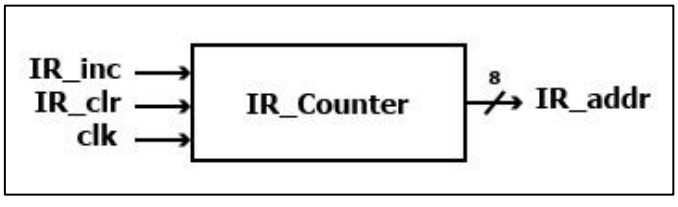


Figure 4.25: Block Diagram of the IR_Counter Module.

Weight DDR3:

The Weight DDR3 stores weight values transferred from the Level 1 Host Memory Weight via the W_data input signal, where they are held in preparation for matrix multiplication. The R_data output connects to the Weight Interface, which then forwards the weights to the Weight FIFOs.

Since each computation in the MMU processes a 4×4 matrix, the Weight DDR3 is configured with 16 memory locations. Accordingly, the 4-bit addr input—controlled by the TPU Controller—specifies which memory location to access. The rd and wr signals serve as the read and write enables, respectively.

Figure 4.26 presents the simplified block diagram of the Weight DDR3 module. The complete Verilog implementation can be found in Supplementary Figure 11.

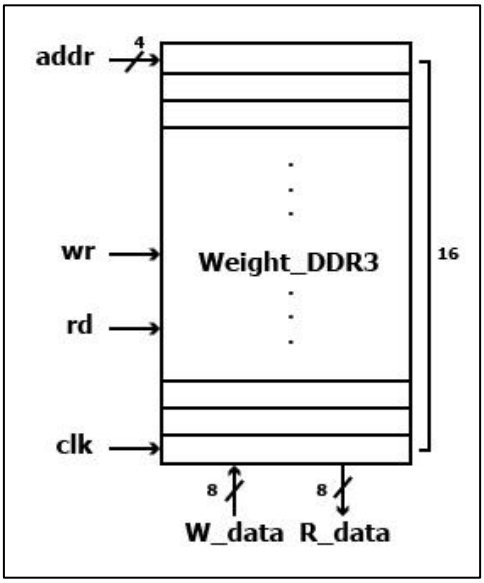


Figure 4.26: Block Diagram of the Weight_DDR3 Module.

Weight Interface:

The Weight Interface serves as a bridge between the Weight DDR3 and the Weight FIFOs, playing a key role in reordering 16 weight values from row-major format into four specific groupings for efficient processing in the MMU. When the input signal push is asserted, weight values are received from the Weight DDR3 via the 8-bit weight_in port and temporarily stored in an internal stack within the Weight Interface. The push_time signal specifies how many weights should be pushed into the stack, while the count signal determines the target location for each value within the stack.

Once all weights are loaded, the controller asserts the pop signal to sequentially output the values from the stack to four dedicated output ports, each connected to a corresponding Weight FIFO. After the full set of weights has been propagated, the pop_complete signal is asserted and sent to the control unit to indicate completion.

Figure 4.27 presents the simplified block diagram of the Weight Interface module. Figure 4.28 shows the partial Verilog code for the module, including the I/O ports and the pust_time updating logic. The complete Verilog implementation can be found in Supplementary Figure 12.

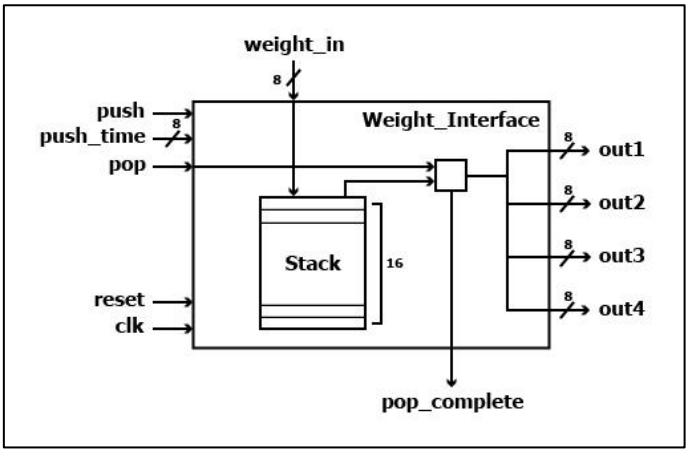


Figure 4.27: Block Diagram of the Weight_Interface Module.

```
module Weight_interface(clk, reset, weight_in, push_time, push,
      pop, pop_complete, out1, out2, out3, out4);

   parameter Data_size = 8;
   parameter STACK_SIZE = 16;
   reg [Data_size-1:0] stack [STACK_SIZE-1:0]; // Stack to store weights
   reg [Data_size-1:0] count; // Counter to track inputs and stack index
   input clk;
   input reset;
   input push, pop;
   input [Data_size-1:0] weight_in;
   input [Data_size-1:0] push_time; // Signal to push weights into the stack
   output reg pop_complete;
   output reg [Data_size-1:0] out1;
   output reg [Data_size-1:0] out2;
   output reg [Data_size-1:0] out3;
   output reg [Data_size-1:0] out4;

   reg [Data_size-1:0] pushtime;
   // Pop sequence control
   reg [Data_size-1:0] pop_count; // Counter to control the pop sequence

   // Register pushtime to capture push_time input
   always @(posedge clk or posedge reset) begin
       if (reset) begin
           pushtime <= 5'b0;
       end else begin
           pushtime <= push_time;
       end
   end

   // Stack push phase
   always @(posedge clk or posedge reset) begin
       if (reset) begin
           count <= 0; // Reset the counter
       end else if (push) begin
```

Figure 4.28: Partial Verilog Code for Weight Interface Module.

Weight FIFO:

The Weight FIFO module operates as a first-in-first-out (FIFO) buffer that transfers weight values from the Weight Interface to the Matrix Multiplication Unit (MMU). When the wr_en (write enable) signal is asserted, the input port buf_in receives weight values and stores them in the FIFO memory. When the rd_en (read enable) signal is asserted, the stored weights are sequentially read out through buf_out and forwarded to the MMU.

To manage data flow and prevent overflow or underflow, the module uses several internal tracking signals: fifo_counter monitors the number of stored elements, while buf_empty and buf_full indicate the availability status of the FIFO. Since the MMU in this project contains 4 columns, four independent Weight FIFOs are implemented—one for each column—to properly forward weights into the systolic array.

Figure 4.29 presents the simplified block diagram of the Weight FIFO module. The complete Verilog implementation can be found in Supplementary Figure 13.

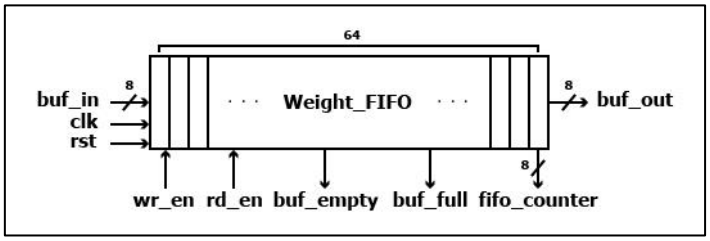


Figure 4.29: Block Diagram of the Weight_FIFO Module.

Unified Buffer:

The Unified Buffer stores activation values in preparation for matrix multiplication. It receives activations from two sources: the Level 1 Host Memory and the Activation Normalization Unit (AN Unit).

Initially, activation values are transferred from the Level 1 Host Memory through the 8-bit input port Host_datain when the Host_wr signal is asserted. Once all 16 activation values are successfully stored in internal memory, the buffer enters a read phase. When rd = 1 and wr = 0, the stored data is distributed across 16 output ports and sent to the Systolic Data Setup Unit (SDSU) for reordering.

After computation, the TPU Controller sets rd = 0 and wr = 1, enabling new activation results from the AN Unit to be written back into the Unified Buffer at specific memory locations. Once these 16 updated values are stored, the signal Host_rd is asserted,

allowing the results to be forwarded to the Level 1 Host Memory via the 8-bit Host_dataout port. This process ensures the computed results are preserved in the higher memory hierarchy, while freeing the Unified Buffer to receive the next block of results without overwriting previously stored data.

Figure 4.30 presents the simplified block diagram of the Unified Buffer module. Figure 4.31 shows the partial Verilog code for the Unified Buffer, including both internal read/write operations and the read/write interface with the L1 Host Memory. The complete Verilog implementation can be found in Supplementary Figure 17.

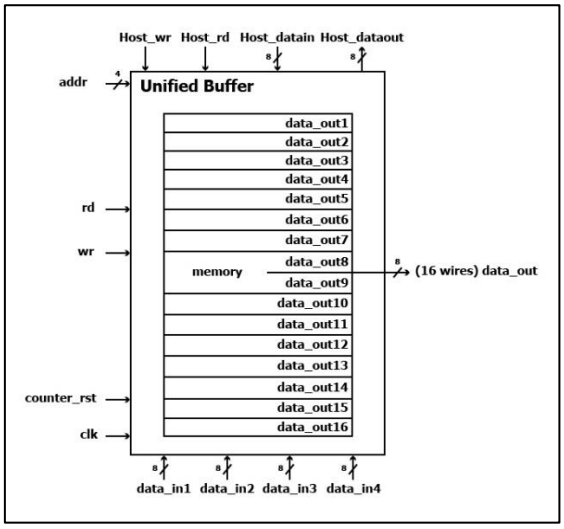


Figure 4.30: Block Diagram of the Unified Buffer Module.

```
            3'd3: begin
                memory[12] <= data_in1;
                memory[13] <= data_in2;
                memory[14] <= data_in3;
                memory[15] <= data_in4; // Final expected write to memory[15]
                end
            default: ; // No action, ensuring no unintended overwrites
        endcase
    end
    else if (rd && !wr) begin
        // Assign outputs based on memory values
        data_out1 <= memory[0];
        data_out2 <= memory[1];
        data_out3 <= memory[2];
        data_out4 <= memory[3];
        data_out5 <= memory[4];
        data_out6 <= memory[5];
        data_out7 <= memory[6];
        data_out8 <= memory[7];
        data_out9 <= memory[8];
        data_out10 <= memory[9];
        data_out11 <= memory[10];
        data_out12 <= memory[11];
        data_out13 <= memory[12];
        data_out14 <= memory[13];
        data_out15 <= memory[14];
        data_out16 <= memory[15];
    end else if (Host_wr) begin
        memory[addr] <= Host_datain;
    end else if (Host_rd) begin
        Host_dataout <= memory[addr];
    end
end
```

Figure 4.31: Partial Verilog Code for Unified Buffer.

Systolic Data Setup Unit:

The Systolic Data Setup Unit (SDSU) receives activation values from the Unified Buffer through 16 input wires and reorders them into four output ports following a specific sequence. This reordering ensures that the activations arrive at the Matrix Multiplication Unit (MMU) with the correct timing to perform accurate multiplication with the corresponding weight values. The specific ordering pattern is demonstrated in the simulation results presented later in this chapter. Figure 4.32 presents the simplified block diagram of the Systolic Data Setup Unit module. Figure 4.33 shows the partial Verilog code for this module, illustrating the data rearrangement logic used to achieve the parallelogram-shaped ordering before forwarding data to the MMU. The complete Verilog implementation can be found in Supplementary Figure 18.

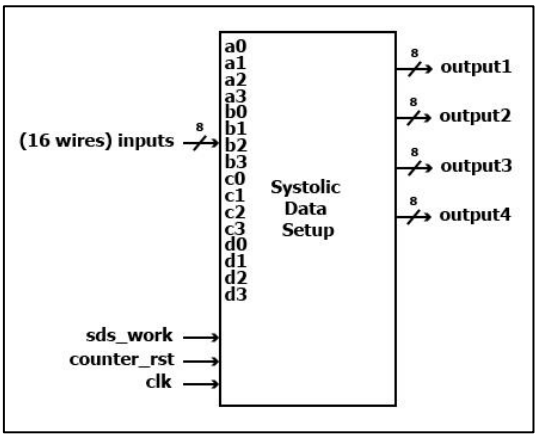


Figure 4.32: Block Diagram of the Systolic Data Setup Unit Module.

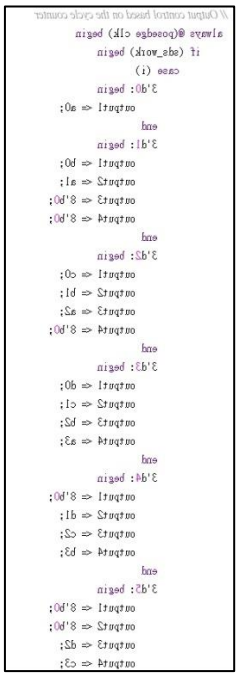


Figure 4.33: Partial Verilog Code for Systolic Data Setup Unit.

Activation FIFO:

The Activation FIFO module functions as a first-in-first-out (FIFO) buffer that transfers activation values from the Unified Buffer, via the Systolic Data Setup Unit (SDSU), to the Matrix Multiplication Unit (MMU). When the wr_en (write enable) signal is asserted, the buf_in port receives activation values, which are then stored in the FIFO's internal memory. When the rd_en (read enable) signal is asserted, the stored activations are sequentially output through buf_out and sent to the MMU.

To regulate data flow and prevent overflow or underflow, the module employs several internal tracking signals: fifo_counter monitors the number of stored elements, while buf_empty and buf_full indicate the FIFO's readiness for reading or writing. Since the MMU architecture in this project includes four rows, four independent Activation FIFO modules are instantiated—one per row—to correctly forward activation data into the systolic array.

Figure 4.34 presents the simplified block diagram of the Activation FIFO module. The complete Verilog implementation can be found in Supplementary Figure 19.

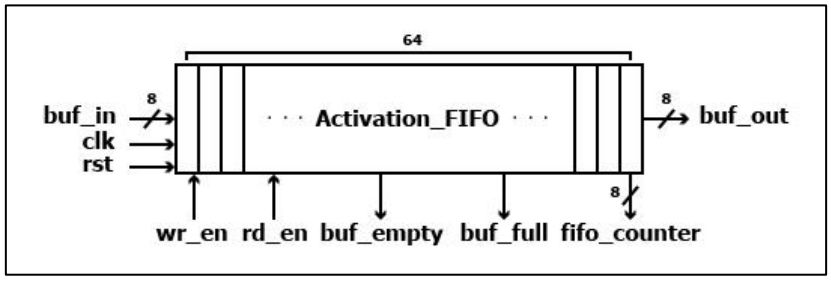


Figure 4.34: Block Diagram of the Activation_FIFO Module.

Row Detector:

The Row Detector module in the Matrix Multiplication Unit (MMU) assists each systolic cell in determining its respective row position. This functionality is essential during the weight loading phase, as it ensures that each weight is either stored in the appropriate weight register or forwarded to the systolic cell in the next row, depending on its location.

The module receives an input signal, rowsignal, from the TPU Controller. This signal is internally processed to generate the corresponding weight_passtime signals, which are then sent to the systolic cells within each row. These output signals guide the cells in deciding whether to retain the current weight or pass it further down the row.

Figure 4.35 shows the complete Verilog code for the Row Detector module, including the logic for computing weight_passtime before sending the output to each row of the Systolic Cell.

```
module row_detector(rowsignal, weight_passtime);
    parameter BIT_WIDTH = 8;
    input [BIT_WIDTH-1:0] rowsignal; // for this PE to check what row it is located, 8-bit for expansion to 256x256 MMU
    output reg [BIT_WIDTH-1:0] weight_passtime;

    always @(*) begin
        weight_passtime = 8'b00000011 - rowsignal; // Adjust this to match the MMU row logic
    end

endmodule
```

Figure 4.35: Complete Verilog Code for Row Detector.

Systolic Cell:

The Systolic Cell in the MMU serves as a fundamental processing element for matrix multiplication. When the input signal weight_pass is asserted (set to 1), the cell enters weight loading mode, accepting weight values through its top input ports and storing them in an internal weight register at the appropriate clock cycles. During this phase, the output signal weightloading is asserted to indicate active weight loading. Once all weights are stored, weightloading is de-asserted (set to 0), signaling that the cell is ready to perform computation.

During the computation phase, the control signals row_en and col_en must both be asserted to activate the cell's processing logic. If either signal remains low, the cell bypasses computation and simply forwards the accumulated sum (psum_input) to the next cell. This behavior is reserved for potential future optimization. The idea is that when both row_en and col_en are de-asserted, the cell can skip the multiplication and accumulation steps—reducing unnecessary operations and conserving power by directly passing the partial sum to the next row.

As illustrated in Figure 4.36, the core operation performed by each systolic cell follows the equation:

$$output = (activation \times weight) + previous\_sum$$

This design relies on precise data arrival and synchronized timing, making correct data forwarding to each cell at the right clock cycle essential. To ensure this, the structure in this project includes extensive reordering modules and a comprehensive TPU Controller to manage the timing and control flow throughout the MMU.

However, during implementation, the weight_passtime signal sent from the Row Detector module caused a multi-driven port issue. This will be discussed in more detail in the next chapter. For now, it is important to note that this module requires further optimization to resolve such hardware conflicts and improve overall design reliability.

Figure 4.36 presents the simplified block diagram of the Systolic Cell module. Figure 4.37 shows a portion of the Verilog code that demonstrates the improper coding style responsible for the multiple-driven nets issue in the Systolic Cell. Specifically, the signal weight_passtime_reg is modified in two separate always blocks, which violates standard HDL design constraints. The complete Verilog implementation can be found in Supplementary Figure 15.

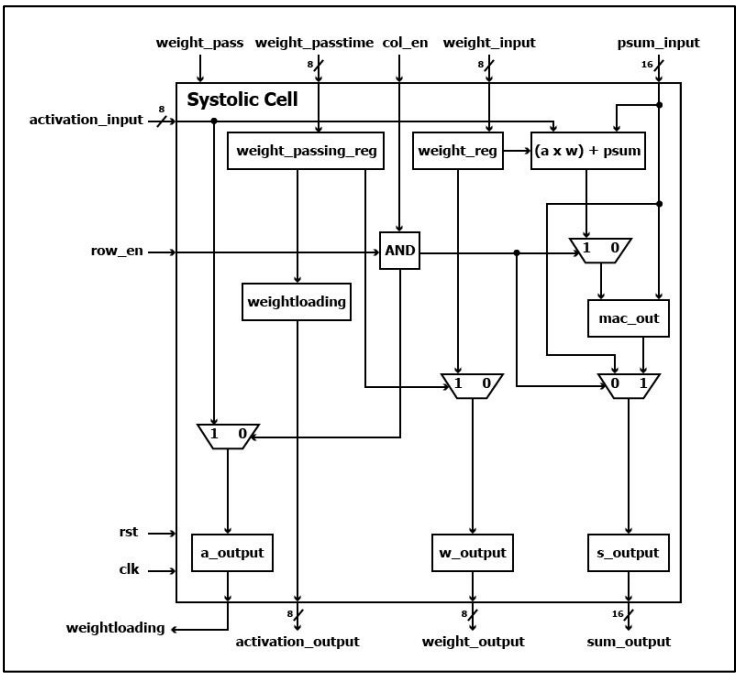
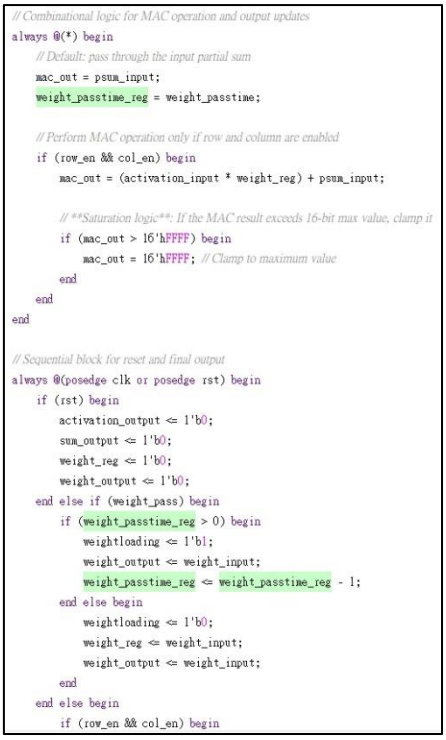


Figure 4.36: Block Diagram of the Systolic Cell Module.



```
// Combinational logic for MAC operation and output updates
always @(*) begin
    // Default: pass through the input partial sum
    mac_out = psum_input;
    weight_passtime_reg = weight_passtime;

    // Perform MAC operation only if row and column are enabled
    if (row_en && col_en) begin
        mac_out = (activation_input * weight_reg) + psum_input;

        // **Saturation logic**: If the MAC result exceeds 16-bit max value, clamp it
        if (mac_out > 16'hFFFF) begin
            mac_out = 16'hFFFF; // Clamp to maximum value
        end
    end
end

// Sequential block for reset and final output
always @(posedge clk or posedge rst) begin
    if (rst) begin
        activation_output <= 1'b0;
        sum_output <= 1'b0;
        weight_reg <= 1'b0;
        weight_output <= 1'b0;
    end else if (weight_pass) begin
        if (weight_passtime_reg > 0) begin
            weightloading <= 1'b1;
            weight_output <= weight_input;
            weight_passtime_reg <= weight_passtime_reg - 1;
        end else begin
            weightloading <= 1'b0;
            weight_reg <= weight_input;
            weight_output <= weight_input;
        end
    end else begin
        if (row_en && col_en) begin
```

Figure 4.37: Verilog Code Snippet from the Systolic Cell Leading to Multiple-Driven Nets Issue.

Matrix Multiplication Unit:

The Matrix Multiplication Unit (MMU) is the core computational block within the TPU, responsible for performing matrix multiplication using a systolic array architecture. This architecture relies on precise data forwarding in a specific sequence to ensure correct and efficient computation. The MMU is composed of 16 interconnected Systolic Cells arranged in a 4×4 array, with 4 accompanying Row Detectors that help manage the flow of weight data across rows while preserving modularity in the design.

Once all weights have been successfully loaded into the MMU, the weightload_complete signal is asserted to inform the control unit that it can proceed to the next state in the computation process. The MMU produces four output values, each of which is sent to the Accumulator module for result collection and post-processing.

Figure 4.38 presents the simplified block diagram of the MMU module. Figure 4.39 shows a portion of the Verilog code that illustrates how the Row Detectors and Systolic Cells are interconnected within the MMU. The complete Verilog implementation is provided in Supplementary Figure 16.

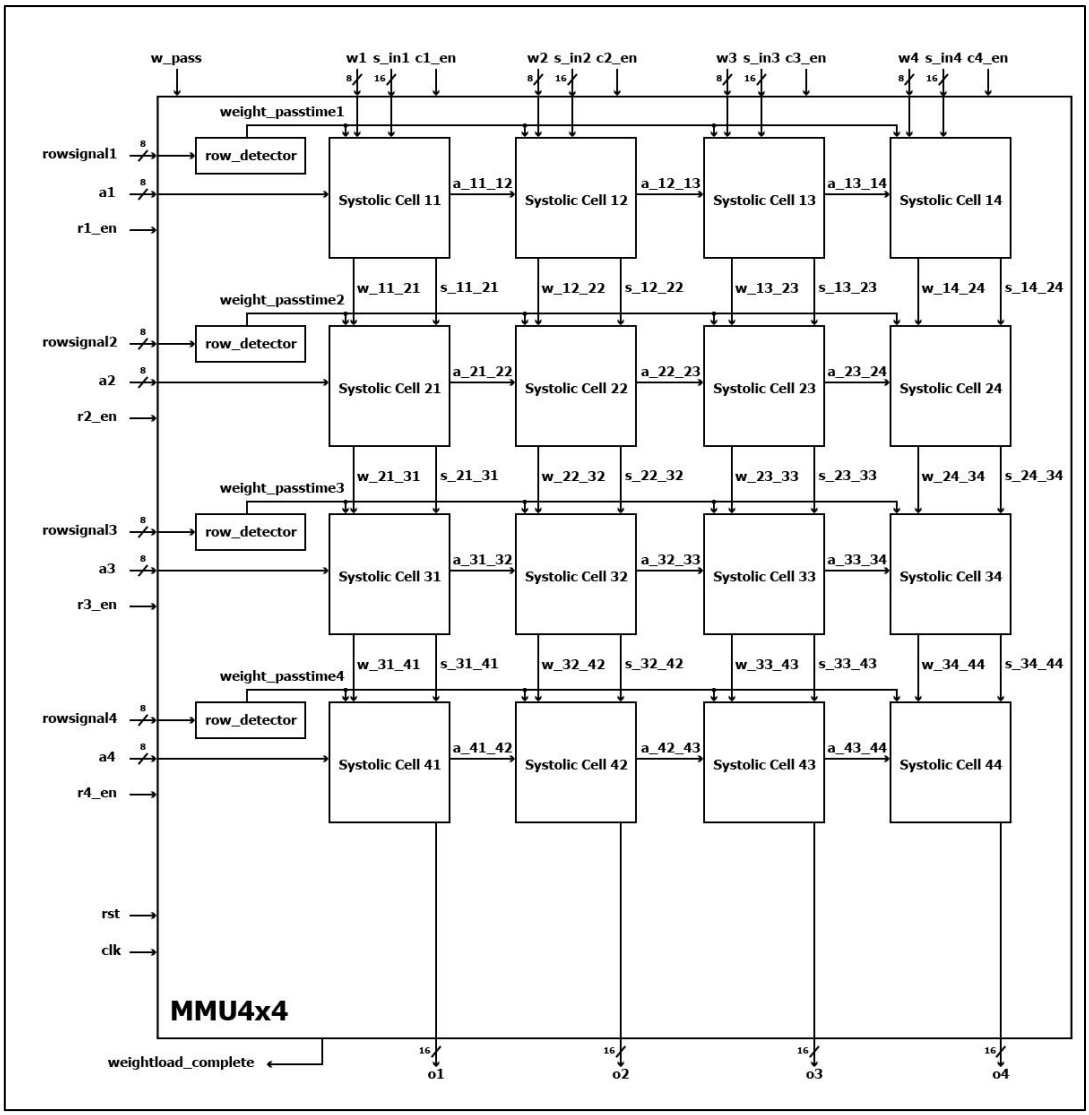


Figure 4.38: Block Diagram of the Matrix Multiplication Unit Module.

```
//--------------------------------ROW2--------------------------------//
row_detector u2(
.rowsignal(rowsignal2), .weight_passtime(weight_passtime2)
);

//--------------------------------ROW3--------------------------------//
row_detector u3(
.rowsignal(rowsignal3), .weight_passtime(weight_passtime3)
);

//--------------------------------ROW4--------------------------------//
row_detector u4(
.rowsignal(rowsignal4), .weight_passtime(weight_passtime4)
);

//--------------------------------COL 1--------------------------------//
systolic_cell pe11(
.clk(clk), .rst(rst), .row_en(r1_en), .col_en(c1_en), .weight_pass(w_pass),
.weight_passtime(weight_passtime1), .weightloading(weightloading1),
.activation_input(a1), .weight_input(w1), .psum_input(s_in1),
.activation_output(a_11_12), .sum_output(s_11_21), .weight_output(w_11_21)
);

systolic_cell pe21(
.clk(clk), .rst(rst), .row_en(r2_en), .col_en(c1_en), .weight_pass(w_pass),
.weight_passtime(weight_passtime2), .weightloading(weightloading2),
.activation_input(a2), .weight_input(w_11_21), .psum_input(s_11_21),
.activation_output(a_21_22), .sum_output(s_21_31), .weight_output(w_21_31)
);

systolic_cell pe31(
.clk(clk), .rst(rst), .row_en(r3_en), .col_en(c1_en), .weight_pass(w_pass),
.weight_passtime(weight_passtime3), .weightloading(weightloading3),
.activation_input(a3), .weight_input(w_21_31), .psum_input(s_21_31),
.activation_output(a_31_32), .sum_output(s_31_41), .weight_output(w_31_41)
```

Figure 4.39: Partial Verilog Code of MMU Showing Interconnection of Row Detectors and Systolic Cells.

Accumulator:

The Accumulator stores the results computed by the MMU and performs maximum value comparisons—both vector-wise and matrix-wise—for use in the rounding process handled by the Activation Normalization Unit (AN_Unit).

The Accumulator currently contains two internal memory structures. If the tiled_computing_sig is asserted, the results from the four input ports are stored in the Temp registers and TMM_memories for intermediate accumulation. In contrast, during basic matrix multiplication, the inputs are stored directly into the Memories. Once all results have been successfully written, the Accumulator_finish_storing signal is asserted and sent to the TPU Controller to indicate readiness for the next operation.

As discussed in Section 3.3.2, for basic matrix multiplication, the TPU Controller will proceed to enable the Accumulator to compute the maximum values. However, in tiled

matrix multiplication mode, the Accumulator must first receive and store the complete 8×8 result matrix. Only after this condition is met will the tiled_MM_storing_complete signal be asserted, allowing maximum value comparison to begin—specifically in matrix-wise fashion.

The read_index and cycle_counter signals are used to control when new values are stored, ensuring that only valid data is retained while outdated or redundant data is discarded.

In future work, the Accumulator will be extended to support the accumulation of exponential sum values required for activation functions such as Sigmoid and SoftMax.

Figure 4.40 presents the simplified block diagram of the Accumulator module. Figures 4.41 and 4.42 show partial Verilog code for the Accumulator. Figure 4.41 illustrates how inputs are stored into the Temp memories and then written into the TMM_memories based on specific cases and counters. Figure 4.42 demonstrates the logic used for vector-wise and matrix-wise maximum value comparisons. The complete Verilog implementation can be found in Supplementary Figure 20.

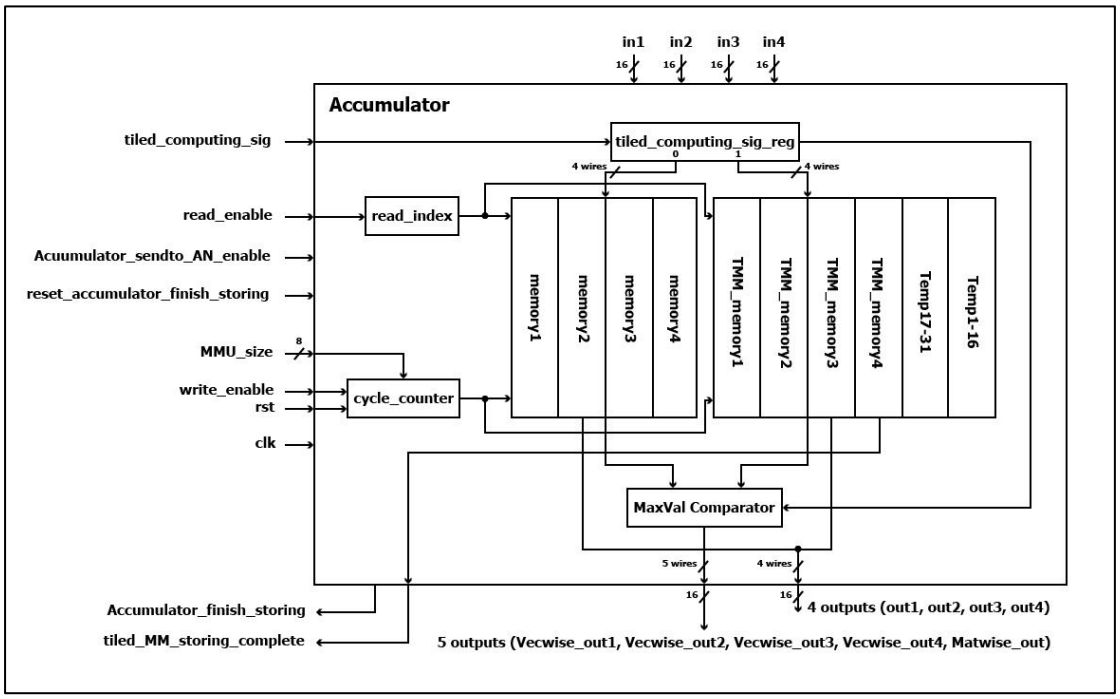
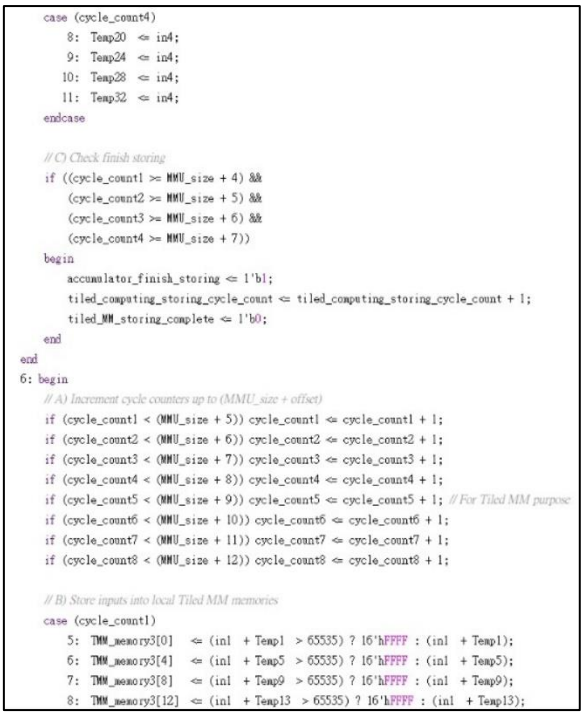


Figure 4.40: Block Diagram of the Accumulator Module.



```
    case (cycle_count4)
        8:  Temp20  <= in4;
        9:  Temp24  <= in4;
        10:  Temp28  <= in4;
        11:  Temp32  <= in4;
    endcase

    // C) Check finish storing
    if ((cycle_count1 >= MMU_size + 4) &&
        (cycle_count2 >= MMU_size + 5) &&
        (cycle_count3 >= MMU_size + 6) &&
        (cycle_count4 >= MMU_size + 7))
    begin
        accumulator_finish_storing <= 1'b1;
        tiled_computing_storing_cycle_count <= tiled_computing_storing_cycle_count + 1;
        tiled_MM_storing_complete <= 1'b0;
    end
end
6: begin
    // A) Increment cycle counters up to (MMU_size + offset)
    if (cycle_count1 < (MMU_size + 5)) cycle_count1 <= cycle_count1 + 1;
    if (cycle_count2 < (MMU_size + 6)) cycle_count2 <= cycle_count2 + 1;
    if (cycle_count3 < (MMU_size + 7)) cycle_count3 <= cycle_count3 + 1;
    if (cycle_count4 < (MMU_size + 8)) cycle_count4 <= cycle_count4 + 1;
    if (cycle_count5 < (MMU_size + 9)) cycle_count5 <= cycle_count5 + 1; // For Tiled MM purpose
    if (cycle_count6 < (MMU_size + 10)) cycle_count6 <= cycle_count6 + 1;
    if (cycle_count7 < (MMU_size + 11)) cycle_count7 <= cycle_count7 + 1;
    if (cycle_count8 < (MMU_size + 12)) cycle_count8 <= cycle_count8 + 1;

    // B) Store inputs into local Tiled MM memories
    case (cycle_count1)
        5:  TMM_memory3[0]   <= (in1  + Temp1  > 65535) ? 16'hFFFF : (in1  + Temp1);
        6:  TMM_memory3[4]   <= (in1  + Temp5  > 65535) ? 16'hFFFF : (in1  + Temp5);
        7:  TMM_memory3[8]   <= (in1  + Temp9  > 65535) ? 16'hFFFF : (in1  + Temp9);
        8:  TMM_memory3[12]  <= (in1  + Temp13  > 65535) ? 16'hFFFF : (in1  + Temp13);
```

Figure 4.41: Partial Verilog Code of the Accumulator Showing Temp and TMM Memory Storage Logic.

```
always @(posedge clk) begin
    if (accumulator_finish_storing && !tiled_computing_sig_reg) begin

        // A) Row-By-Row Compare
        if (row_compare_index < memory_size) begin
            if (cycle_count4 >= (8 + row_compare_index)) begin
                // read memory for row_compare_index
                compareVal1 = memory1[row_compare_index];
                compareVal2 = memory2[row_compare_index];
                compareVal3 = memory3[row_compare_index];
                compareVal4 = memory4[row_compare_index];

                max_1or2  = (compareVal1 > compareVal2)? compareVal1 : compareVal2;
                max_3or4  = (compareVal3 > compareVal4)? compareVal3 : compareVal4;
                maxAll    = (max_1or2 > max_3or4)? max_1or2 : max_3or4;

                case(row_compare_index)
                  2'd0: Vectorwise_MaxVal_r1 <= maxAll;
                  2'd1: Vectorwise_MaxVal_r2 <= maxAll;
                  2'd2: Vectorwise_MaxVal_r3 <= maxAll;
                  2'd3: Vectorwise_MaxVal_r4 <= maxAll;
                endcase

                row_compare_index <= row_compare_index + 1;
            end
        end
        // B) Once all rows are done, do final matrixwise compare
        else if (row_compare_index == memory_size) begin
            max_row12 = (Vectorwise_MaxVal_r1 > Vectorwise_MaxVal_r2)
                        ? Vectorwise_MaxVal_r1 : Vectorwise_MaxVal_r2;
            max_row34 = (Vectorwise_MaxVal_r3 > Vectorwise_MaxVal_r4)
                        ? Vectorwise_MaxVal_r3 : Vectorwise_MaxVal_r4;
            Matrixwise_MaxVal <= (max_row12 > max_row34)? max_row12 : max_row34;
            end
     end
end
```

Figure 4.42: Partial Verilog Code of the Accumulator for Vector-Wise and Matrix-Wise Max Value Comparison.

Activation Normalization Unit:

The Activation Normalization Unit (AN_Unit) is responsible for applying activation functions—such as ReLU, Sigmoid, and SoftMax—to the results produced by the MMU before sending them back to the Unified Buffer. In this project, only the ReLU function is implemented; Sigmoid and SoftMax are reserved for future work.

To maintain proper data flow and timing, the AN_Unit includes its own TMM_memory structure to temporarily store results received from the Accumulator. Once all values are processed through the activation function and stored, the TPU Controller asserts the AN_Unit_work signal to trigger the rounding operation.

During rounding, the 8-bit input Nfactor_in—a user-defined scaling factor—is used to set the rounding boundary. This ensures that all results are scaled to a specific range, maintaining bit-width constraints and reducing precision overflow. After multiplication with Nfactor_in, the intermediate results become 24-bit wide. These are then divided by a 16-bit maximum value (either vector-wise or matrix-wise), and finally rounded back down to 8-bit values. The final 8-bit results are sent to the Unified Buffer through the four output ports.

Figure 4.43 presents the simplified block diagram of the AN_Unit module. Figures 4.44 and 4.45 show partial Verilog code for the AN_Unit. Figure 4.44 illustrates how the ReLU activation function is applied to each input and stored in the corresponding TMM_memories within the AN_Unit based on the store_cycle_count counter. Figure 4.45 demonstrates the logic used for normalization and rounding operations—the upper section shows partial logic for tiled matrix multiplication, while the lower section handles basic matrix multiplication. The complete Verilog implementation can be found in Supplementary Figure 21.

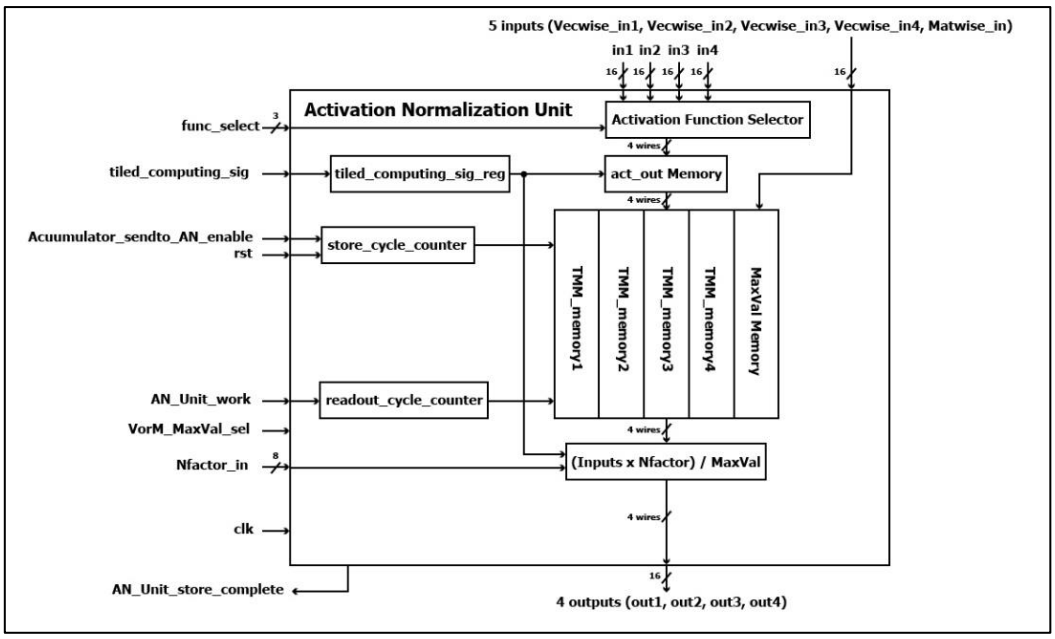


Figure 4.43: Block Diagram of the Activation Normalization Unit Module.

```
        default: begin // ReLU logic
            act_out1 = (in1[15] == 0) ? in1 : 16'b0; // Check MSB for sign
            act_out2 = (in2[15] == 0) ? in2 : 16'b0;
            act_out3 = (in3[15] == 0) ? in3 : 16'b0;
            act_out4 = (in4[15] == 0) ? in4 : 16'b0;
        end
    endcase
end

always @(posedge clk) begin
    if (rst) begin
        AN_Unit_store_complete = 0;
    end else if (AN_Unit_work) begin
        readout_cycle_count = readout_cycle_count + 1;
    end else if(Acuumulator_sendto_AN_enable && !tiled_computing_sig_reg) begin
        Vectorwise_MaxVal_r1 = Vecwise_in1;
        Vectorwise_MaxVal_r2 = Vecwise_in2;
        Vectorwise_MaxVal_r3 = Vecwise_in3;
        Vectorwise_MaxVal_r4 = Vecwise_in4;
        Matrixwise_MaxVal = Matwise_in;
    case (store_cycle_count)
        0: begin
           TMM_memory1[0] = act_out1;
           TMM_memory2[0] = act_out2;
           TMM_memory3[0] = act_out3;
           TMM_memory4[0] = act_out4;
           AN_Unit_store_complete = 0;
        end
        1: begin
           TMM_memory1[0] = act_out1;
           TMM_memory2[0] = act_out2;
           TMM_memory3[0] = act_out3;
           TMM_memory4[0] = act_out4;
           AN_Unit_store_complete = 0;
        end
        2: begin
```

Figure 4.44: Partial Verilog Code of the AN_Unit Showing ReLU Activation and Storage Logic.

```
        8'd21: begin
            temp24b_out1 = (TMM_memory1[14] * Nfactor_reg);
            temp24b_out2 = (TMM_memory2[14] * Nfactor_reg);
            temp24b_out3 = (TMM_memory3[14] * Nfactor_reg);
            temp24b_out4 = (TMM_memory4[14] * Nfactor_reg);
        end
        8'd22: begin
            temp24b_out1 = (TMM_memory1[15] * Nfactor_reg);
            temp24b_out2 = (TMM_memory2[15] * Nfactor_reg);
            temp24b_out3 = (TMM_memory3[15] * Nfactor_reg);
            temp24b_out4 = (TMM_memory4[15] * Nfactor_reg);
        end
    endcase

    // Normalize the output
    out1 = temp24b_out1 / Matrixwise_MaxVal;
    out2 = temp24b_out2 / Matrixwise_MaxVal;
    out3 = temp24b_out3 / Matrixwise_MaxVal;
    out4 = temp24b_out4 / Matrixwise_MaxVal;

end else if (AN_Unit_work && !tiled_computing_sig_reg && !VorM_MaxVal_sel_reg) begin
            case(readout_cycle_count)
            8'd0: begin
                temp24b_out1 = (TMM_memory1[0] * Nfactor_reg);
                temp24b_out2 = (TMM_memory2[0] * Nfactor_reg);
                temp24b_out3 = (TMM_memory3[0] * Nfactor_reg);
                temp24b_out4 = (TMM_memory4[0] * Nfactor_reg);
                out1 = temp24b_out1 / Vectorwise_MaxVal_r1;
                out2 = temp24b_out2 / Vectorwise_MaxVal_r1;
                out3 = temp24b_out3 / Vectorwise_MaxVal_r1;
                out4 = temp24b_out4 / Vectorwise_MaxVal_r1;
                end
```

Figure 4.45: Partial Verilog Code of the AN_Unit Demonstrating Normalization and Rounding for Tiled and Basic Multiplication.

L1 Controller (TPU Controller):

The Level 1 Controller, also known as the TPU Controller, is responsible for managing all I/O signals associated with the components within the TPU layer of the architecture. Functionally, the controller can be divided into two main parts: a collection of registers for internal counters, and a main always block that governs state transitions within the control finite state machine (FSM).

The TPU Controller fetches instructions from the IR Memory, decodes them, and orchestrates the operation of each component by asserting the appropriate control signals based on the current instruction and system state.

Figure 4.46 presents the simplified block diagram of the TPU Controller module, with all relevant I/O signals clearly shown and mapped to their respective controlled components.

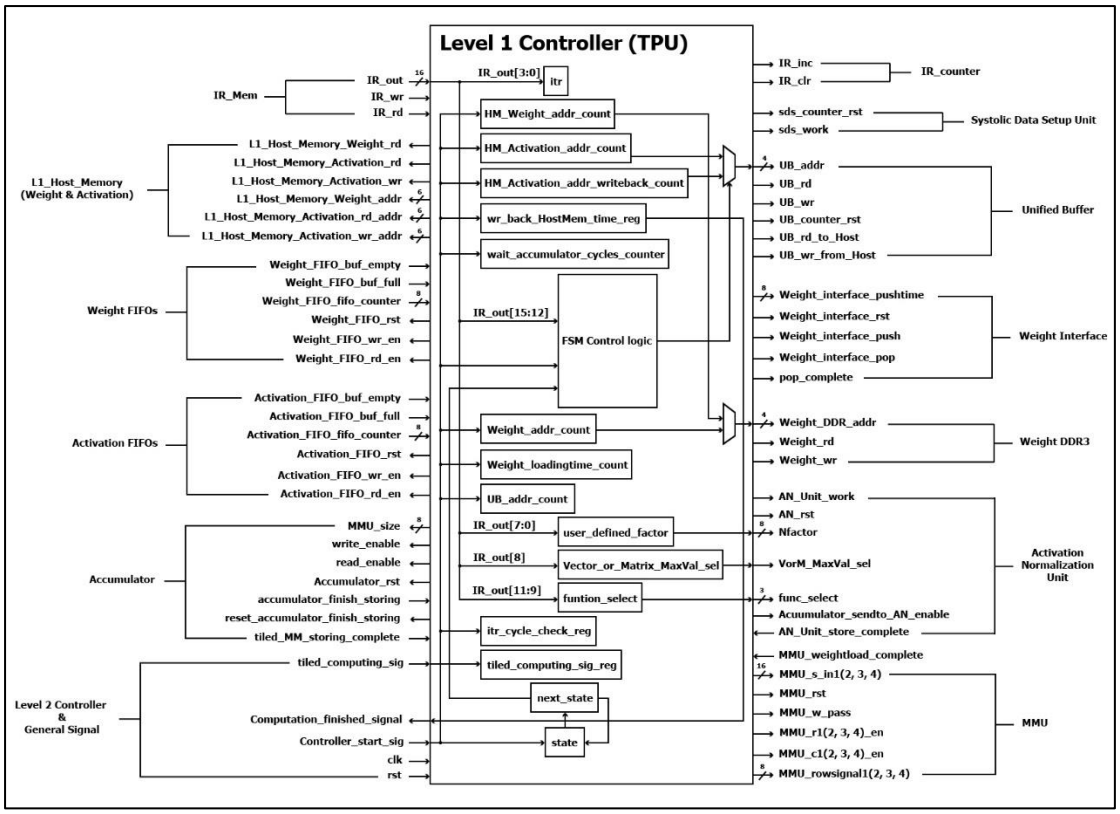


Figure 4.46: Block Diagram of the TPU Controller Module.

Figures 4.47 to 4.50 present partial Verilog code for the TPU Controller. Figure 4.47 displays most of the I/O signals and the 35 states used for state transitions within the controller. Figure 4.48 shows several address counters responsible for controlling the memory locations where values are stored. However, due to the current coding style of some of these counters, the TPU Controller also suffers from a multiple-driven nets issue, which should be addressed in future work. Figure 4.49 illustrates the beginning of the main always block responsible for state transitions, where many control signals are reset to 0 at the start to avoid repetitive assignments in each state. Finally, Figure 4.50 provides an example of a state transition to demonstrate how the Verilog code is structured for control logic. The complete Verilog implementation can be found in Supplementary Figure 22.

```
UB_addr, UB_rd, UB_wr, UB_counter_rst, UB_rd_to_Host, UB_wr_from_Host, // for Unified Buffer
sds_counter_rst, sds_work, // for systolic data setup
Weight_DDR_addr, Weight_rd, Weight_wr, // for Weight DDR3 memory
Weight_interface_pushtime, Weight_interface_rst, Weight_interface_push, Weight_interface_pop, pop_complete, // for Weight interface
Weight_FIFO_rst, Weight_FIFO_wr_en, Weight_FIFO_rd_en,
Weight_FIFO_buf_empty, Weight_FIFO_buf_full, Weight_FIFO_fifo_counter, // for Weight FIFO
Activation_FIFO_rst, Activation_FIFO_wr_en, Activation_FIFO_rd_en,
Activation_FIFO_buf_empty, Activation_FIFO_buf_full, Activation_FIFO_fifo_counter, // for Activation FIFO
MMU_size, write_enable, read_enable, Accumulator_rst, accumulator_finish_storing, reset_accumulator_finish_storing, tiled_MM_storing_complete, // for Accumulator
func_select, VorM_MaxVal_sel, Xfactor, AM_Unit_work, AM_rst, // for Activation_Normalization_Unit
Accumulator_sendto_AM_enable, AM_Unit_store_complete, // Enable signal to allow MaxVal be sent from Accumulator to Activation_Normalization_Unit
MMU_rst, MMU_w_pass, MMU_weightload_complete, MMU_s_in1, MMU_s_in2, MMU_s_in3, MMU_s_in4, // MMU_weightload_complete is an 'input' to detect whether the weights are finishing loading into MMU
MMU_r1_en, MMU_r2_en, MMU_r3_en, MMU_r4_en, MMU_c1_en, MMU_c2_en, MMU_c3_en, MMU_c4_en,
MMU_rowsignal1, MMU_rowsignal2, MMU_rowsignal3, MMU_rowsignal4, // for MMU
Computation_finished_signal, clk, rst);

parameter Instruction_size = 16;
parameter HostMemory_addr_size = 6;
parameter address_size = 4;
parameter MMUsize = 8;
parameter Data_size = 8;
parameter S_initial = 0, S_fetch = 1; // fetch instructions for IR (in TPU)
parameter S_decode = 2; // decode the instruction from IR
parameter S_L1HostMem_to_WandA1 = 3, S_L1HostMem_to_WandA2 = 4;
// Weight load //
parameter S_Weight_DDR3_to_interface1 = 5, S_Weight_DDR3_to_interface2 = 6, S_Weight_wait_stack_for_one_more_cycle = 7;
parameter S_Weight_interface_to_WFIFO_stage1 = 8, S_Weight_interface_to_WFIFO_stage2 = 9, S_Weight_wait_FIFO_for_one_more_cycle = 10;
parameter S_Weight_WFIFO_to_MMU = 11, S_Weight_MMU_rowdetect_stage1 = 12, S_Weight_MMU_rowdetect_stage2 = 13, S_Weight_MMU_load_complete = 14;
parameter S_UB_to_sds1 = 15, S_UB_to_sds2 = 16, S_UB_to_sds3 = 17, S_Activation_Wait_for_one_more_cycle = 18, S_Activation_AFIFO_to_MMU_and_computing = 19;
parameter S_MMU_computing = 20, S_Wait_Accumulator_to_store_MaxVal1 = 21, S_Wait_Accumulator_to_store_MaxVal2 = 22;
parameter S_Sending_MaxVal_from_Accumulator_to_AM1 = 23, S_Sending_MaxVal_from_Accumulator_to_AM2 = 24, S_Wait_Accumulator_for_one_more_clock_cycle = 25;
parameter S_AMUnit_start = 26, S_UB_store1 = 27, S_UB_store2 = 28, S_UB_store3 = 29;
parameter S_Computing_Time_Compare = 30, S_extra_itr = 31;
parameter S_UB_to_L1HostMem1 = 32, S_UB_to_L1HostMem2 = 33, S_UB_to_L1HostMem_wait_for_one_more_cycle = 34;
parameter S_stay = 35;
```

Figure 4.47: I/O Signals and State Definitions for the TPU Controller.

```
always @(posedge clk) begin
    if (Controller_start_sig) begin
        HM_Activation_addr_count <= 6'b000000;
        HM_Activation_addr <= 6'b000000;
    end else if (inner_controller_rst) begin
        HM_Activation_addr_count <= 6'b000000;
    end else if ((next_state == S_L1HostMem_to_WandA1 || next_state == S_L1HostMem_to_WandA2) && HM_Activation_addr_count < 17) begin
        HM_Activation_addr <= HM_Activation_addr + 6'b000001;
        HM_Activation_addr_count <= HM_Activation_addr_count + 6'b000001;
    end
end

always @(posedge clk) begin
    if (Controller_start_sig) begin
        HM_Activation_addr_writeback_count <= 5'b00000;
        HM_Activation_writeback_addr <= 6'b000000;
    end else if (inner_controller_rst) begin
        HM_Activation_addr_writeback_count <= 6'b000000;
    end else if ((next_state == S_UB_to_L1HostMem1 || next_state == S_UB_to_L1HostMem2) && HM_Activation_addr_writeback_count < 17) begin
        HM_Activation_writeback_addr <= HM_Activation_writeback_addr + 6'b000001;
        HM_Activation_addr_writeback_count <= HM_Activation_addr_writeback_count + 6'b000001;
    end
end

// Sequential logic to manage Weight_addr_count for each clock cycle in S_Weight_load_loop
always @(posedge clk) begin
    if (Controller_start_sig) begin
        Weight_addr_count <= 8'b00000000;
    end else if (next_state == S_Weight_wait_stack_for_one_more_cycle) begin
        Weight_addr_count <= 8'b00000000;
    end else if ((next_state == S_Weight_DDR3_to_interface1 || next_state == S_Weight_DDR3_to_interface2) && Weight_addr_count < 15) begin
        Weight_addr_count <= Weight_addr_count + 8'b00000001;
    end
end
```

Figure 4.48: Partial Verilog Code for Address Counters in the TPU Controller.

Figure 4.49: Beginning of the Main always Block for TPU State Transitions.

```
S_Weight_wait_stack_for_one_more_cycle: begin
                $display("State: Weight completely loaded into WInterface ! ! !");
                Weight_interface_pop = 1; // have to presend this pop signal one state earlier, or, all FIFOs[0] will load output = 0.
                next_state = S_Weight_interface_to_WFIFO_stage1;
                end
S_Weight_interface_to_WFIFO_stage1: begin
                if(pop_complete != 1) begin
                Weight_interface_pop = 1;
                Weight_FIFO_wr_en = 1;
                $display("State: Weight_load_to_WFIFO. . .");
                next_state = S_Weight_interface_to_WFIFO_stage2;
                end
                else begin
                next_state = S_Weight_wait_FIFO_for_one_more_cycle;
                end
                end
 S_Weight_interface_to_WFIFO_stage2: begin
                if(pop_complete != 1) begin
                Weight_interface_pop = 1;
                Weight_FIFO_wr_en = 1;
                $display("State: Weight_load_to_WFIFO. . .");
                next_state = S_Weight_interface_to_WFIFO_stage1;
                end
                else begin
                next_state = S_Weight_wait_FIFO_for_one_more_cycle;
                end
                end
S_Weight_wait_FIFO_for_one_more_cycle: begin
                $display("State: Weight completely loaded into WFIFO");
                Weight_FIFO_wr_en = 0;
                Weight_interface_pop = 0;
                Weight_interface_rst = 1;
                next_state = S_Weight_WFIFO_to_MMU;
                end
S_Weight_WFIFO_to_MMU: begin
```

Figure 4.50: Verilog Code Snippet Illustrating State Transition Logic in the TPU Controller.

4.3 Summary

Chapter 4 presented the structural composition and Verilog-based implementation of the Tiled Matrix Multiplication Unit (TMMU), transitioning from architectural concepts to practical hardware design. The chapter began with a high-level block diagram illustrating the full TMMU system and the internal TPU structure. This visual overview clarified how key modules—such as memory units, controllers, data setup units, and the MMU—interact through a tightly coordinated dataflow architecture.

To support the tiled matrix multiplication flow described in the previous chapter, Chapter 4 provided detailed coverage of every core hardware component within the TMMU. These were organized into two major layers: the Level 2 layer (TMMU top-level) and the Level 1 layer (TPU). Each module was introduced with a functional block diagram, accompanied by key portions of its Verilog implementation to highlight important design decisions and internal logic. Full module code listings are provided separately in the supplementary section for reference and completeness.

At the Level 2 layer, modules such as the L2 IR Memory, Host Memory blocks (for weights, activations, and instructions), the Tiled Systolic Data Setup Unit, and the TMMU Controller were described. These components are responsible for instruction management, memory coordination, and data preparation prior to computation. The chapter explained how these units ensure that weight and activation data are properly formatted and delivered to the TPU.

At the Level 1 layer, the focus shifted to the internal workings of the TPU. Modules such as the Unified Buffer, IR Memory, MMU (composed of Systolic Cells and Row Detectors), Accumulator, and Activation Normalization Unit were explored in depth. These elements collectively perform matrix multiplication, activation functions, and result normalization, operating in tightly timed synchronization under the direction of the TPU Controller. Special attention was given to the MMU's systolic array structure and the role of reordering components in ensuring proper data alignment.

The content in this chapter forms the RTL foundation of the TMMU and translates the architectural goals outlined in Chapter 3 into a functioning, synthesizable hardware model. This implementation-level perspective not only validates the proposed design but also lays the groundwork for hardware testing, system integration, and future optimization in the chapters that follow.

CHAPTER 5

TMMU: Performance and Future Work

With a clear understanding of the TMMU's architecture, control state diagrams, and component block diagrams established in the previous chapter, this chapter focuses on evaluating the performance and outlining future directions for improvement. The discussion is divided into four main parts: simulation results, power estimation, Innovus floorplanning, and future improvements.

The simulation results, generated using Xilinx Vivado, showcase the complete dataflow of the TMMU over time. Simulations for eight different matrix sizes are presented. With a clock period of 20 ns, the TMMU successfully completes a 4×4 matrix multiplication in 3,000 ns (150 clock cycles) and an 8×8 multiplication in approximately 17,000 ns (850 clock cycles), demonstrating its scalability and functional correctness.

The power estimation, also generated in Vivado, provides a detailed breakdown of power consumption across each hardware component. The analysis includes on-chip power, junction temperature, and related thermal/power parameters. This section also highlights an implementation issue—related to multi-driven nets—that affected the accuracy of power reporting and will require attention in future revisions.

The Innovus floorplanning section demonstrates how the Verilog-based TMMU design can be transitioned into a physical layout suitable for fabrication. By converting the RTL design into a floorplanned architecture, this project validates the physical feasibility of the TMMU and reveals early fabrication potential. The process also exposes design rule violations and connectivity issues that will need to be resolved for tape-out readiness.

Finally, the future improvements section revisits all known issues encountered in this project—including multi-driven port conflicts, timing and DRC violations, and tool licensing limitations—and discusses possible solutions. This section also outlines potential extensions of the TMMU architecture that could be explored in future work,

such as additional activation functions or deeper matrix sizes.

5.1 Simulation Results

The simulation was conducted using Xilinx Vivado, with a 20 ns clock period configured in the testbench environment. All test inputs—weights, activations, and instruction sequences—were manually pre-initialized into the corresponding memory modules to simplify the setup. The waveforms provide a cycle-accurate view of the TMMU's operation under both BMM and TMM modes, allowing verification of data correctness and module synchronization.

Since the TMMU developed in this project supports both Basic Matrix Multiplication (BMM) mode and Tiled Matrix Multiplication (TMM) mode, the simulation results are organized into two sections based on computation mode. The MMU has a fixed size of 4×4, so any matrix larger than 4×4 must be computed using the tiled approach (TMM), while smaller or equal matrix sizes can be processed using the basic approach (BMM).

Through these simulation results, the full dataflow of the TMMU system can be clearly observed. All data—including weights, activations, and instruction sets—are pre-initialized into the Level 2 Host Memories and the Level 2 IR Memory. From there, data is passed through the Tiled Systolic Data Setup Units into the TPU for computation. The internal movement within the TPU is also illustrated in the waveform outputs, confirming that intermediate results are computed and stored correctly—both in terms of values and memory locations.

The main simulation examples showcase an 8×8 matrix in TMM mode, as most steps throughout the simulation process are repeated across other matrix sizes. For matrix sizes that result in different control behaviors—such as those requiring padding or block alignment—additional figures are provided to highlight and explain these variations.

5.1.1 Tiled Matrix Multiplication (5×5 to 8×8)

The simulation results form the core focus of this project's evaluation. As described in the previous section, all input data—including weights, activations, and instruction sets—are pre-initialized into the Level 2 Host Memory to simplify simulation setup. Figure 5.1 illustrates the initial activation contents of the Level 2 Host Memory Activation.

For ease of result verification, the memory is populated with sequential values from 0 to 255 across the 256 available memory locations. In the first 8×8 tiled matrix multiplication operation, the system retrieves the first 64 values (from 0 to 63) and forwards them to the TPU for computation. This setup allows the simulation to clearly demonstrate how data is loaded, processed, and propagated through the TMMU pipeline.

```
# run 20000ns
State: S_L2_initial
State: S_L2_initial
Activations at L2 HostMem (16x16 Matrix Form):
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
 32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
 48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
 64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
 96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
```

Figure 5.1: Pre-Initialized Level 2 Host Memory Activation Contents at Time = 0 ns

As shown in Figures 5.2 to 5.4, the memory structures in the lower hierarchy do not contain any values during the initial state and therefore display undefined values (represented as 'x').

```
# run 20000ns
State: S_L2_initial
State: S_L2_initial
Value at L1 HostMem Weight[0]:  x
Value at L1 HostMem Weight[1]:  x
Value at L1 HostMem Weight[2]:  x
Value at L1 HostMem Weight[3]:  x
Value at L1 HostMem Weight[4]:  x
Value at L1 HostMem Weight[5]:  x
Value at L1 HostMem Weight[6]:  x
Value at L1 HostMem Weight[7]:  x
Value at L1 HostMem Weight[8]:  x
Value at L1 HostMem Weight[9]:  x
Value at L1 HostMem Weight[10]:  x
Value at L1 HostMem Weight[11]:  x
Value at L1 HostMem Weight[12]:  x
Value at L1 HostMem Weight[13]:  x
Value at L1 HostMem Weight[14]:  x
Value at L1 HostMem Weight[15]:  x
Value at L1 HostMem Weight[16]:  x
```

Figure 5.2: Contents of Level 1 Host Memory (Weight) at Time = 0 ns

```
Value at L1 HostMem Activation[0]:   x
Value at L1 HostMem Activation[1]:   x
Value at L1 HostMem Activation[2]:   x
Value at L1 HostMem Activation[3]:   x
Value at L1 HostMem Activation[4]:   x
Value at L1 HostMem Activation[5]:   x
Value at L1 HostMem Activation[6]:   x
Value at L1 HostMem Activation[7]:   x
Value at L1 HostMem Activation[8]:   x
Value at L1 HostMem Activation[9]:   x
Value at L1 HostMem Activation[10]:   x
Value at L1 HostMem Activation[11]:   x
Value at L1 HostMem Activation[12]:   x
Value at L1 HostMem Activation[13]:   x
Value at L1 HostMem Activation[14]:   x
Value at L1 HostMem Activation[15]:   x
Value at L1 HostMem Activation[16]:   x
```

Figure 5.3: Contents of Level 1 Host Memory (Activation) at Time = 0 ns

```
Value at L1 IR Mem[0]: xxxx
Value at L1 IR Mem[1]: xxxx
Value at L1 IR Mem[2]: xxxx
Value at L1 IR Mem[3]: xxxx
Value at L1 IR Mem[4]: xxxx
Value at L1 IR Mem[5]: xxxx
Value at L1 IR Mem[6]: xxxx
Value at L1 IR Mem[7]: xxxx
Value at L1 IR Mem[8]: xxxx
Value at L1 IR Mem[9]: xxxx
Value at L1 IR Mem[10]: xxxx
Value at L1 IR Mem[11]: xxxx
Value at L1 IR Mem[12]: xxxx
Value at L1 IR Mem[13]: xxxx
Value at L1 IR Mem[14]: xxxx
Value at L1 IR Mem[15]: xxxx
Value at L1 IR Mem[16]: xxxx
```

Figure 5.4: Contents of Level 1 IR Memory at Time = 0 ns.

As the TMMU Controller begins fetching instructions from the Level 2 IR Memory, it first initiates the process of forwarding these instructions into the TPU's Level 1 IR Memory. It is important to reiterate that the instruction sets used by the Level 2 and Level 1 controllers are different and function independently; there is no interaction or overlap between the two. Figure 5.5 shows the instruction loading process taking place at time = 20 ns.

```
time = 20, clk = 0, IR_addr = 00, IR_out: 0000
State: S_L2_fetch
time = 30, clk = 1, IR_addr = 00, IR_out: 0000
time = 40, clk = 0, IR_addr = 00, IR_out: 0000
opcode: 0000
State: Load_ISA
time = 50, clk = 1, IR_addr = 00, IR_out: 0000
time = 60, clk = 0, IR_addr = 00, IR_out: 0000
State: ISA loading to L1 Host Mem1...
time = 70, clk = 1, IR_addr = 00, IR_out: 0000
time = 80, clk = 0, IR_addr = 00, IR_out: 0000
State: ISA loading to L1 Host Mem2...
time = 90, clk = 1, IR_addr = 00, IR_out: 0000
time = 100, clk = 0, IR_addr = 00, IR_out: 0000
State: ISA loading to L1 Host Mem1...
time = 110, clk = 1, IR_addr = 00, IR_out: 0000
time = 120, clk = 0, IR_addr = 00, IR_out: 0000
State: ISA loading to L1 Host Mem2...
time = 130, clk = 1, IR_addr = 00, IR_out: 0000
time = 140, clk = 0, IR_addr = 00, IR_out: 0000
```

Figure 5.5: Instruction Loading from ISA L2 Memory to Level 1 IR Memory (in TPU) at 20 ns.

By time = 800 ns, all instructions required by the TPU Controller have been successfully loaded into the Level 1 IR Memory. For an 8×8 tiled matrix multiplication, approximately 30 instructions are needed. Some of these instructions serve purely to reload weight and activation values to ensure that the computed results follow the correct sequence and alignment. Figure 5.6 shows the contents of the Level 1 IR Memory when the Load ISA operation completes at time = 790 ns.

```
time = 770, clk = 1, IR_addr = 00, IR_out: 0000
time = 780, clk = 0, IR_addr = 00, IR_out: 0000
State: ISA loading to L1 Host Mem1...
State: ISA finished loading into L1 Host Mem!!!
time = 790, clk = 1, IR_addr = 00, IR_out: 0000
time = 800, clk = 0, IR_addr = 00, IR_out: 0000
Value at L1 IR Mem[0]: 0000
Value at L1 IR Mem[1]: 1000
Value at L1 IR Mem[2]: 2000
Value at L1 IR Mem[3]: 0000
Value at L1 IR Mem[4]: 2000
Value at L1 IR Mem[5]: 1000
Value at L1 IR Mem[6]: 0000
Value at L1 IR Mem[7]: 2000
Value at L1 IR Mem[8]: 0000
Value at L1 IR Mem[9]: 2000
Value at L1 IR Mem[10]: 0000
Value at L1 IR Mem[11]: 0000
Value at L1 IR Mem[12]: 0000
Value at L1 IR Mem[13]: 1000
Value at L1 IR Mem[14]: 0000
Value at L1 IR Mem[15]: 0000
Value at L1 IR Mem[16]: 2000
Value at L1 IR Mem[17]: 0000
Value at L1 IR Mem[18]: 2000
Value at L1 IR Mem[19]: 0000
Value at L1 IR Mem[20]: 0000
Value at L1 IR Mem[21]: 1000
Value at L1 IR Mem[22]: 0000
Value at L1 IR Mem[23]: 0000
Value at L1 IR Mem[24]: 0000
Value at L1 IR Mem[25]: 2000
Value at L1 IR Mem[26]: 0000
Value at L1 IR Mem[27]: 2000
Value at L1 IR Mem[28]: 3164
```

Figure 5.6: Level 1 IR Memory Contents After Load ISA Completion at 790 ns.

After the instructions for the TPU are fully loaded into the memory structure, the TMMU Controller begins transferring both weight and activation values from the Level 2 Host Memories through the Tiled Systolic Data Setup Units (TSDSUs) into their corresponding Level 1 Host Memories. This step is essential to prepare the TPU for matrix computation.

Figure 5.7 illustrates this value-forwarding process, which completes at time = 940 ns. For an 8×8 matrix, 64 values must be forwarded for both the weights and the activations. Figure 5.8 presents a partial view of the data stored inside the two TSDSUs—one for weights and one for activations—at time = 940 ns. The complete memory contents are available in the supplementary materials.

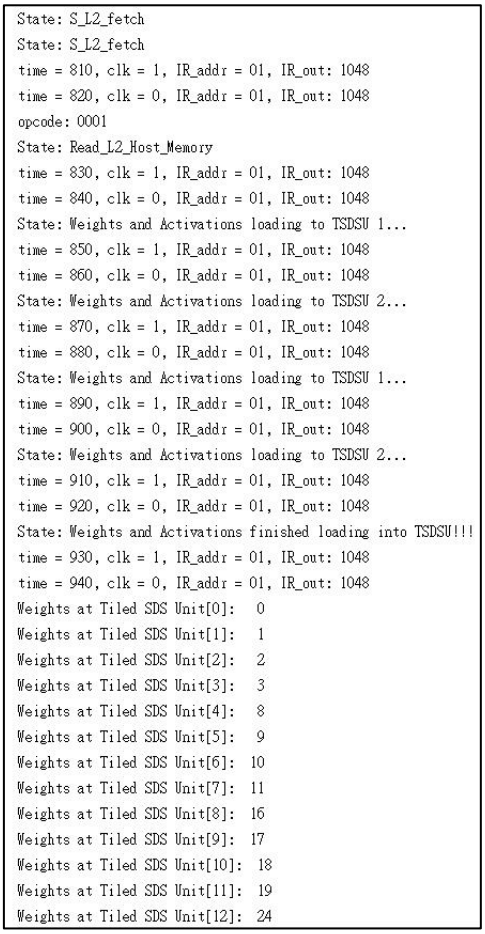


Figure 5.7: Weight and Activation Forwarding to Level 1 Host Memories (Completed at 940 ns).

```
Weights at Tiled SDS Unit[46]:  30
Weights at Tiled SDS Unit[47]:  31
Weights at Tiled SDS Unit[48]:  36
Weights at Tiled SDS Unit[49]:  37
Weights at Tiled SDS Unit[50]:  38
Weights at Tiled SDS Unit[51]:  39
Weights at Tiled SDS Unit[52]:  44
Weights at Tiled SDS Unit[53]:  45
Weights at Tiled SDS Unit[54]:  46
Weights at Tiled SDS Unit[55]:  47
Weights at Tiled SDS Unit[56]:  52
Weights at Tiled SDS Unit[57]:  53
Weights at Tiled SDS Unit[58]:  54
Weights at Tiled SDS Unit[59]:  55
Weights at Tiled SDS Unit[60]:  60
Weights at Tiled SDS Unit[61]:  61
Weights at Tiled SDS Unit[62]:  62
Weights at Tiled SDS Unit[63]:  63
Activations at Tiled SDS Unit[0]:   0
Activations at Tiled SDS Unit[1]:   1
Activations at Tiled SDS Unit[2]:   2
Activations at Tiled SDS Unit[3]:   3
Activations at Tiled SDS Unit[4]:   8
Activations at Tiled SDS Unit[5]:   9
Activations at Tiled SDS Unit[6]:  10
Activations at Tiled SDS Unit[7]:  11
Activations at Tiled SDS Unit[8]:  16
Activations at Tiled SDS Unit[9]:  17
Activations at Tiled SDS Unit[10]:  18
Activations at Tiled SDS Unit[11]:  19
Activations at Tiled SDS Unit[12]:  24
Activations at Tiled SDS Unit[13]:  25
Activations at Tiled SDS Unit[14]:  26
Activations at Tiled SDS Unit[15]:  27
Activations at Tiled SDS Unit[16]:  32
Activations at Tiled SDS Unit[17]:  33
```

Figure 5.8: Partial Memory Contents of Tiled Systolic Data Setup Units During Forwarding.

Next, between time = 940 ns and time = 1040 ns, both weight and activation values are transferred from the TSDSUs to their corresponding Level 1 Host Memories. Figure 5.9 illustrates the data-forwarding process, which completes at time = 1040 ns. Figure 5.10 presents a partial view of the data stored in the two Level 1 Host Memories—one for weights and one for activations—at that moment. The complete memory contents are provided in the supplementary materials.

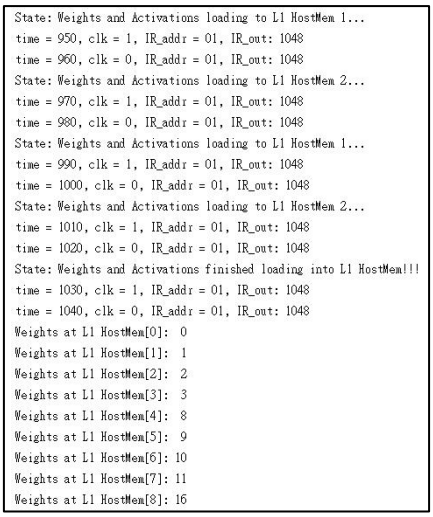


Figure 5.9: Data Forwarding from TSDSUs to Level 1 Host Memories (Completed at 1040 ns).

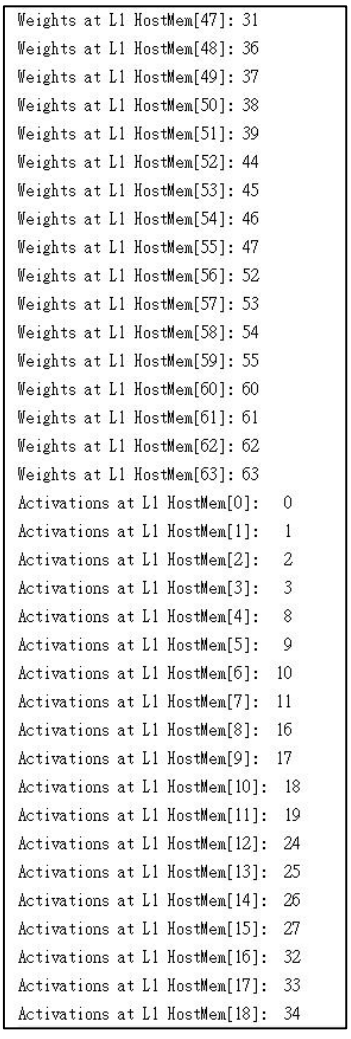


Figure 5.10: Partial Contents of Level 1 Host Memory (Weight & Activation) at Time = 1040 ns.

Since the activation values are stored in memory using row-major format, it is difficult to visualize the entire 8×8 matrix in a single figure. Therefore, Figures 5.11 through 5.14 present the activation storage format for each supported matrix size in this project—from 5×5 to 8×8. These figures follow the same rearranged matrix ordering scheme discussed in Section 3.1.3, ensuring that the values forwarded for computation are properly aligned and sent in the correct sequence.

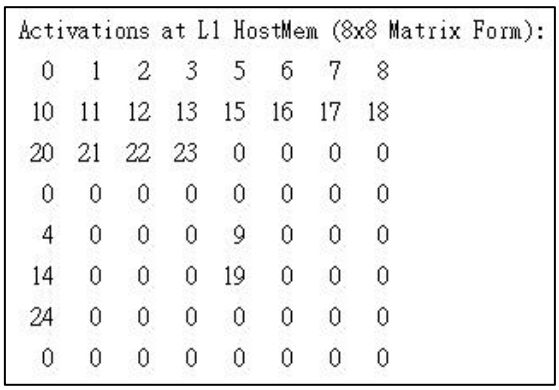


Figure 5.11: Activation Storage Format for 5×5 Matrix in Level 1 Host Memory.

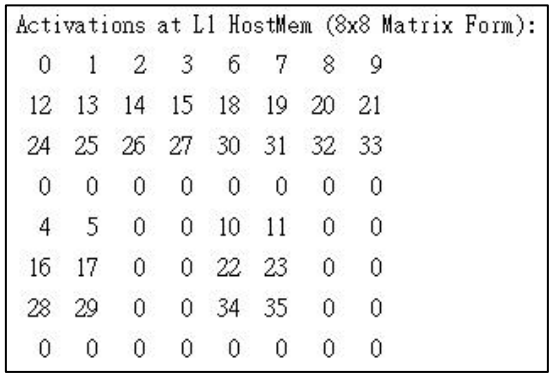


Figure 5.12: Activation Storage Format for 6×6 Matrix in Level 1 Host Memory.

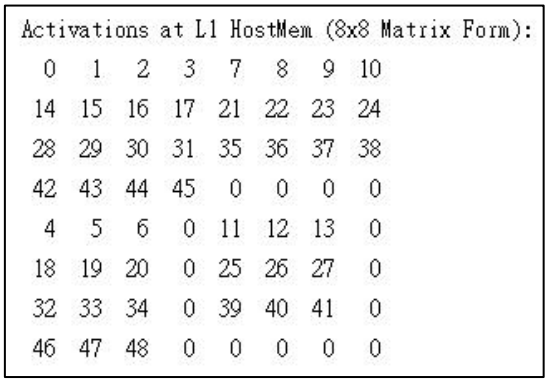


Figure 5.13: Activation Storage Format for 7×7 Matrix in Level 1 Host Memory.

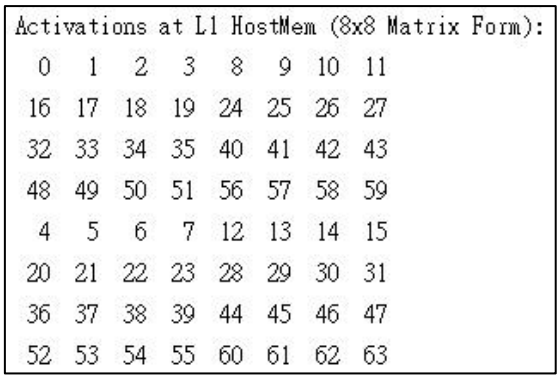


Figure 5.14: Activation Storage Format for 8×8 Matrix in Level 1 Host Memory.

After all instructions and values have been successfully forwarded into the TPU, the TMMU Controller asserts the start signal to trigger the TPU Controller for further operations. Figure 5.15 illustrates the handoff process, where control transitions from the Level 2 Controller (TMMU Controller) to the TPU Controller at time = 1050 ns. The first operation executed by the TPU is to forward the weight and activation values from the Level 1 Host Memories into their corresponding memory structures: the Weight DDR3 for weights, and the Unified Buffer for activations.

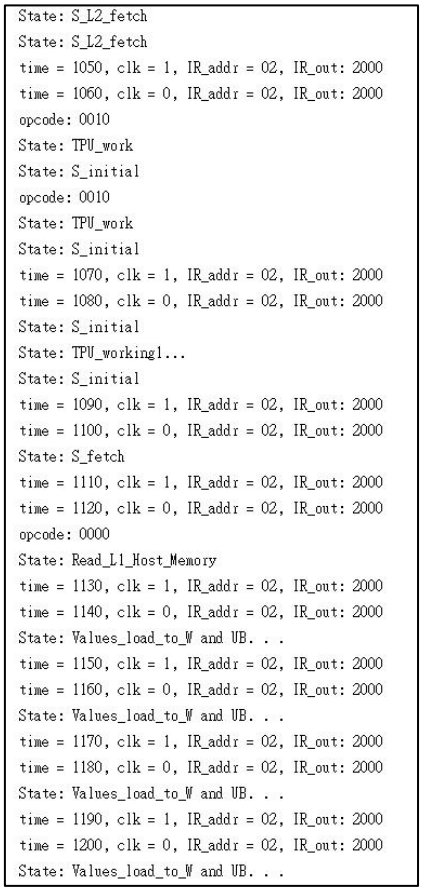


Figure 5.15: Controller Handoff and Data Transfer Initiation at time = 1050 ns.

Both the weights and activations are fully stored in their corresponding memory structures by time = 1470 ns. Since each structure contains only 16 memory locations (sized for a 4×4 block), only one computation block can be held at a time. Figure 5.16 shows the first block of data stored in the Weight DDR3 and Unified Buffer, which will be used for the first round of computation in tiled matrix multiplication.

```
State: Values_completely_load_to_W and UB. . .
time = 1470, clk = 1, IR_addr = 02, IR_out: 2000
time = 1480, clk = 0, IR_addr = 02, IR_out: 2000
Value at Weight DDR3[0]:   0
Value at Weight DDR3[1]:   1
Value at Weight DDR3[2]:   2
Value at Weight DDR3[3]:   3
Value at Weight DDR3[4]:   8
Value at Weight DDR3[5]:   9
Value at Weight DDR3[6]:  10
Value at Weight DDR3[7]:  11
Value at Weight DDR3[8]:  16
Value at Weight DDR3[9]:  17
Value at Weight DDR3[10]:  18
Value at Weight DDR3[11]:  19
Value at Weight DDR3[12]:  24
Value at Weight DDR3[13]:  25
Value at Weight DDR3[14]:  26
Value at Weight DDR3[15]:  27
Value at UB Mem[0]:   0
Value at UB Mem[1]:   1
Value at UB Mem[2]:   2
Value at UB Mem[3]:   3
Value at UB Mem[4]:   8
Value at UB Mem[5]:   9
Value at UB Mem[6]:  10
Value at UB Mem[7]:  11
Value at UB Mem[8]:  16
Value at UB Mem[9]:  17
Value at UB Mem[10]:  18
Value at UB Mem[11]:  19
Value at UB Mem[12]:  24
Value at UB Mem[13]:  25
Value at UB Mem[14]:  26
Value at UB Mem[15]:  27
```

Figure 5.16: First 4×4 block of weights and activations stored in Weight DDR3 and Unified Buffer at time = 1470 ns.

Next, once the weight values are stored in the Weight DDR3, the subsequent instruction is the ‘Read Weight’ operation, which initiates the process of loading these weights into the MMU in preparation for computation. The data transfer path begins at the Weight DDR3, passes through the Weight Interface, flows into the Weight FIFOs, and finally reaches the Systolic Cells within the MMU. Figure 5.17 illustrates the ‘Read Weight’ operation and the corresponding state transition to 'Weight Load to WInterface', starting at time = 1490 ns.

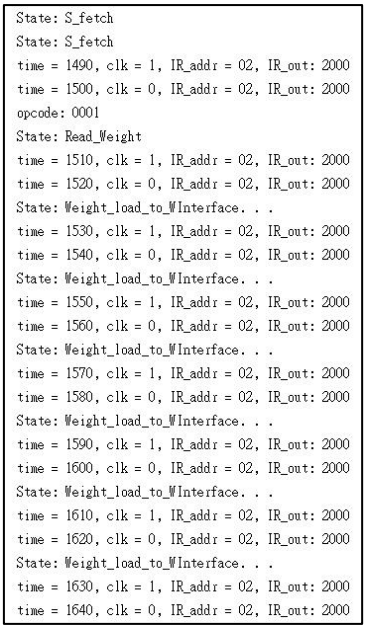


Figure 5.17: 'Read Weight' Operation and Data Transfer from Weight DDR3 to Weight Interface Beginning at Time = 1490 ns

Figure 5.18 shows the operation after the weights are stored in the Weight Interface. Starting at time = 1850 ns, the weights are forwarded to the four Weight FIFOs and then sent to the MMU. The 'row detector' state determines whether each loaded weight should be stored in the register or propagated to the next systolic cell.

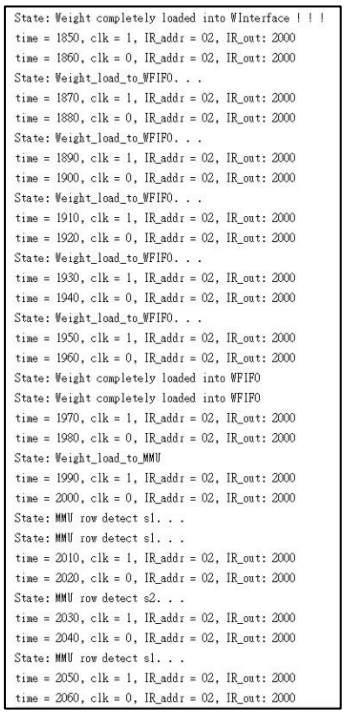


Figure 5.18: Weight Forwarding from Weight Interface to Weight FIFOs and MMU with Row Detection Logic Starting at Time = 1850 ns.

Figure 5.19 shows that all weight values have been successfully loaded into the 4×4 systolic cell array of the MMU at time = 2110 ns. The values are arranged in row-major order.

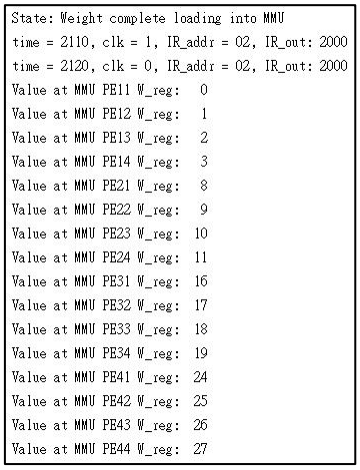


Figure 5.19: Weight Values Loaded into the 4×4 Systolic Cell Array in Row-Major Order at Time = 2110 ns.

After all weight values are successfully stored in the systolic array within the MMU, the TPU Controller proceeds by asserting the necessary control signals to initiate the activation data flow. Activation values are forwarded from the Unified Buffer, passed through the Systolic Data Setup Unit (SDSU), and then loaded into the four Activation FIFOs in preparation for computation. Figure 5.20 shows the execution of the "Computation" instruction beginning at time = 2150 ns, along with the associated data transfer operations.

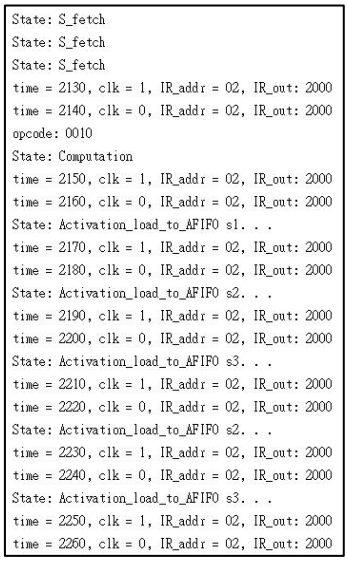


Figure 5.20: Execution of 'Computation' Instruction and Activation Data Transfer from Unified Buffer to Activation FIFOs at Time = 2150 ns.

Figure 5.21 shows the activation data stored within the four Activation FIFOs at time = 2350 ns. To ensure correct results during computation and avoid timing conflicts, the values inside these FIFOs are arranged in a parallelogram pattern rather than a square. This arrangement ensures that each row of the MMU receives activation values at the correct clock cycle. If the activations were aligned in a square format, rows would not be synchronized—for example, the second row might receive its data too early—resulting in incorrect matrix multiplication outputs.

As shown in the second part of the figure, the values stored in the Activation FIFOs are subsequently forwarded into the MMU for matrix computation.

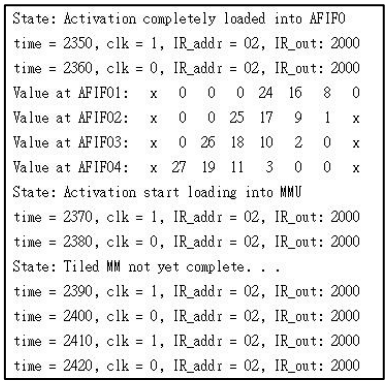


Figure 5.21: Activation Data Stored in the Four Activation FIFOs at Time = 2350 ns.

At time = 2610 ns, the MMU completes all matrix computations, and the resulting values are transferred to the Accumulator. Since this simulation example demonstrates tiled matrix multiplication, the results are first stored in the Temp memory arrays within the Accumulator. These intermediate values will remain in storage until subsequent computation blocks are processed and accumulated, ultimately producing the final output for Block 1, as discussed in Section 3.1.1. Figure 5.22 shows the intermediate results stored in the Temp memory array at time = 2610 ns.

As shown in Figure 5.23, the current results are stored in the Temp memories, while the TMM_memories still hold undefined values (x) since no data has been accumulated into them yet.

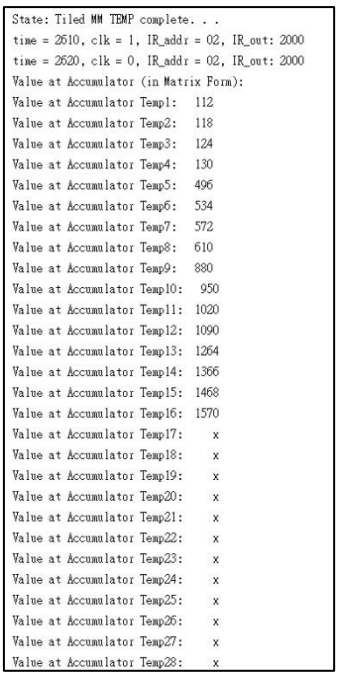


Figure 5.22: Intermediate Results Stored in Accumulator Temp Memory at Time = 2610 ns.

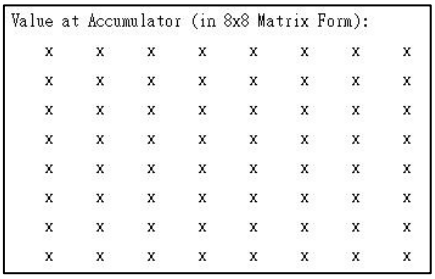


Figure 5.23: TMM_memories Remain Uninitialized (x) Awaiting Future Accumulation at Time = 2610 ns.

Next, the TPU Controller fetches the next block of values—from memory[17] to memory[31]—from the Level 1 Host Memory Activation and transfers them to the Unified Buffer, overwriting the previously stored data. Figure 5.24 shows the new "Read L1 Host Memory" operation initiated at time = 2630 ns. The updated activation values stored in the Unified Buffer are shown in Figure 5.25. Compared to Figure 5.16, it is evident that the earlier values have been successfully overwritten.

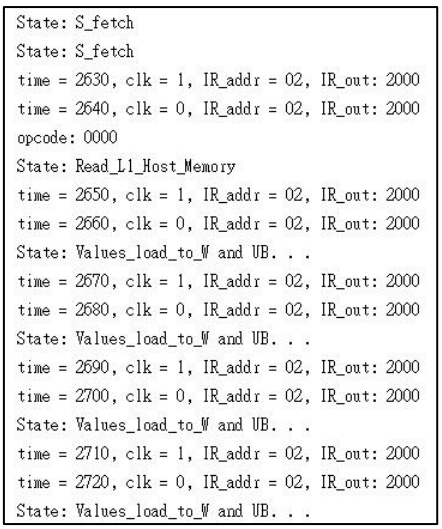


Figure 5.24: Read Operation from L1 Host Memory for Second Block at Time = 2630 ns.

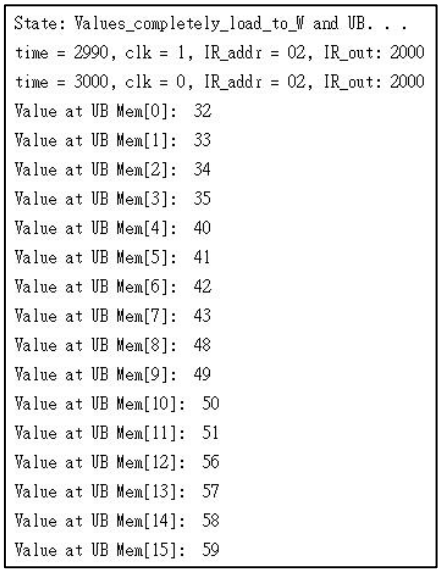


Figure 5.25: Updated Unified Buffer Contents After Overwriting with New Activations.

Following the same steps described earlier, the next set of activation values is forwarded from the Unified Buffer, through the Systolic Data Setup Unit, into the Activation FIFOs, and then loaded into the MMU for computation. The resulting values are stored in the Accumulator's Temp memories at time = 3490 ns, as shown in Figure 5.26. These values are not yet added to the first block's results, as they correspond to a different region of the matrix. The actual accumulation occurs when the matching block—aligned with the first—is computed, as discussed in Section 3.1.1.

```
time = 3490, clk = 1, IR_addr = 02, IR_out: 2000
time = 3500, clk = 0, IR_addr = 02, IR_out: 2000
Value at Accumulator (in Matrix Form):
Value at Accumulator Temp1:   112
Value at Accumulator Temp2:   118
Value at Accumulator Temp3:   124
Value at Accumulator Temp4:   130
Value at Accumulator Temp5:   496
Value at Accumulator Temp6:   534
Value at Accumulator Temp7:   572
Value at Accumulator Temp8:   610
Value at Accumulator Temp9:   880
Value at Accumulator Temp10:   950
Value at Accumulator Temp11:  1020
Value at Accumulator Temp12:  1090
Value at Accumulator Temp13:  1264
Value at Accumulator Temp14:  1366
Value at Accumulator Temp15:  1468
Value at Accumulator Temp16:  1570
Value at Accumulator Temp17:  1648
Value at Accumulator Temp18:  1782
Value at Accumulator Temp19:  1916
Value at Accumulator Temp20:  2050
Value at Accumulator Temp21:  2032
Value at Accumulator Temp22:  2198
Value at Accumulator Temp23:  2364
Value at Accumulator Temp24:  2530
Value at Accumulator Temp25:  2416
Value at Accumulator Temp26:  2614
Value at Accumulator Temp27:  2812
Value at Accumulator Temp28:  3010
Value at Accumulator Temp29:  2800
Value at Accumulator Temp30:  3030
Value at Accumulator Temp31:  3260
Value at Accumulator Temp32:  3490
Value at Accumulator TMM_memory1[0]:    x
```

Figure 5.26: Second Set of Computed Results Stored in Temp Memories at Time = 3490 ns, Awaiting Accumulation with the Corresponding Result Block.

Next, the weight values in the MMU's systolic array are updated with a new set (second block). Figures 5.27 and 5.28 show the 'Read Weight' operation beginning at time = 3530 ns, and the updated weights fully loaded into the MMU by time = 4130 ns

```
State: Read_Weight
time = 3530, clk = 1, IR_addr = 02, IR_out: 2000
time = 3540, clk = 0, IR_addr = 02, IR_out: 2000
State: Weight_load_to_WInterface. . .
time = 3550, clk = 1, IR_addr = 02, IR_out: 2000
time = 3560, clk = 0, IR_addr = 02, IR_out: 2000
State: Weight_load_to_WInterface. . .
time = 3570, clk = 1, IR_addr = 02, IR_out: 2000
time = 3580, clk = 0, IR_addr = 02, IR_out: 2000
State: Weight_load_to_WInterface. . .
time = 3590, clk = 1, IR_addr = 02, IR_out: 2000
time = 3600, clk = 0, IR_addr = 02, IR_out: 2000
State: Weight_load_to_WInterface. . .
time = 3610, clk = 1, IR_addr = 02, IR_out: 2000
time = 3620, clk = 0, IR_addr = 02, IR_out: 2000
State: Weight_load_to_WInterface. . .
time = 3630, clk = 1, IR_addr = 02, IR_out: 2000
time = 3640, clk = 0, IR_addr = 02, IR_out: 2000
State: Weight_load_to_WInterface. . .
```

Figure 5.27: 'Read Weight' Operation Begins to Load the Second Weight Block into the MMU at Time = 3530 ns.

```
State: Weight complete loading into MMU
time = 4130, clk = 1, IR_addr = 02, IR_out: 2000
time = 4140, clk = 0, IR_addr = 02, IR_out: 2000
Value at MMU PE11 W_reg:  32
Value at MMU PE12 W_reg:  33
Value at MMU PE13 W_reg:  34
Value at MMU PE14 W_reg:  35
Value at MMU PE21 W_reg:  40
Value at MMU PE22 W_reg:  41
Value at MMU PE23 W_reg:  42
Value at MMU PE24 W_reg:  43
Value at MMU PE31 W_reg:  48
Value at MMU PE32 W_reg:  49
Value at MMU PE33 W_reg:  50
Value at MMU PE34 W_reg:  51
Value at MMU PE41 W_reg:  56
Value at MMU PE42 W_reg:  57
Value at MMU PE43 W_reg:  58
Value at MMU PE44 W_reg:  59
```

Figure 5.28: Updated Weights in the MMU's Systolic Array.

As the third block of activation values is required, new activations are loaded from the Level 1 Host Memory into the Activation FIFOs for computation. Figure 5.29 shows the start of this new computation round at time = 4770 ns.

```
State: Activation start loading into MMU
time = 4770, clk = 1, IR_addr = 02, IR_out: 2000
time = 4780, clk = 0, IR_addr = 02, IR_out: 2000
State: Tiled MM not yet complete. . .
time = 4790, clk = 1, IR_addr = 02, IR_out: 2000
time = 4800, clk = 0, IR_addr = 02, IR_out: 2000
time = 4810, clk = 1, IR_addr = 02, IR_out: 2000
time = 4820, clk = 0, IR_addr = 02, IR_out: 2000
time = 4830, clk = 1, IR_addr = 02, IR_out: 2000
time = 4840, clk = 0, IR_addr = 02, IR_out: 2000
time = 4850, clk = 1, IR_addr = 02, IR_out: 2000
time = 4860, clk = 0, IR_addr = 02, IR_out: 2000
time = 4870, clk = 1, IR_addr = 02, IR_out: 2000
time = 4880, clk = 0, IR_addr = 02, IR_out: 2000
time = 4890, clk = 1, IR_addr = 02, IR_out: 2000
time = 4900, clk = 0, IR_addr = 02, IR_out: 2000
time = 4910, clk = 1, IR_addr = 02, IR_out: 2000
time = 4920, clk = 0, IR_addr = 02, IR_out: 2000
time = 4930, clk = 1, IR_addr = 02, IR_out: 2000
time = 4940, clk = 0, IR_addr = 02, IR_out: 2000
time = 4950, clk = 1, IR_addr = 02, IR_out: 2000
time = 4960, clk = 0, IR_addr = 02, IR_out: 2000
time = 4970, clk = 1, IR_addr = 02, IR_out: 2000
time = 4980, clk = 0, IR_addr = 02, IR_out: 2000
time = 4990, clk = 1, IR_addr = 02, IR_out: 2000
time = 5000, clk = 0, IR_addr = 02, IR_out: 2000
State: Tiled MM TEMP complete. . .
time = 5010, clk = 1, IR_addr = 02, IR_out: 2000
time = 5020, clk = 0, IR_addr = 02, IR_out: 2000
```

Figure 5.29: Third Block of Activation Being Computed at Time = 4770 ns.

The computed results are stored in the Accumulator. This time, the new block is added to the first set of results previously stored in Temp[0] to Temp[15], producing the first portion of the 8×8 result matrix. These values are then stored in TMM_memory1, as shown in Figure 5.30 at time = 5020 ns.

Value at Accumulator (in 8x8 Matrix Form):

1120	1148	1176	1204	x	x	x	x
2912	3004	3096	3188	x	x	x	x
4704	4860	5016	5172	x	x	x	x
6496	6716	6936	7156	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x

Figure 5.30: Accumulated Results Forming the First Part of the 8×8 Output at Time = 5020 ns.

Following the same steps described above, and with the correct instructions loaded into the TPU Controller, the second block of the result matrix is stored into TMM_memory at time = 5900 ns. The stored results are presented in Figure 5.31.

Value at Accumulator (in 8x8 Matrix Form):

1120	1148	1176	1204	x	x	x	x
2912	3004	3096	3188	x	x	x	x
4704	4860	5016	5172	x	x	x	x
6496	6716	6936	7156	x	x	x	x
8288	8572	8856	9140	x	x	x	x
10080	10428	10776	11124	x	x	x	x
11872	12284	12696	13108	x	x	x	x
13664	14140	14616	15092	x	x	x	x

Figure 5.31: Second Block of the 8×8 Result Matrix Stored in TMM_memory at Time = 5900 ns.

Similarly, as instructions continue to be fetched and decoded by the TPU Controller for specific operations, the third block of the result matrix is stored in TMM_memory at time = 12,860 ns, as shown in Figure 5.32.

Value at Accumulator (in 8x8 Matrix Form):

1120	1148	1176	1204	1232	1260	1288	1316
2912	3004	3096	3188	3280	3372	3464	3556
4704	4860	5016	5172	5328	5484	5640	5796
6496	6716	6936	7156	7376	7596	7816	8036
8288	8572	8856	9140	x	x	x	x
10080	10428	10776	11124	x	x	x	x
11872	12284	12696	13108	x	x	x	x
13664	14140	14616	15092	x	x	x	x

Figure 5.32: Third Block of the 8×8 Result Matrix Stored in TMM_memory at Time = 12,860 ns.

At time = 13,740 ns, all four blocks of the 8×8 result matrix are fully stored in the TMM_memory within the Accumulator. Figure 5.33 shows the final stored results, while Figure 5.34 presents the expected output calculated using an online matrix multiplication tool. The comparison verifies that the TMMU's computed results are correct. The input 8×8 matrices consist of values ranging from 0 to 63.

Value at Accumulator (in 8x8 Matrix Form):

1120	1148	1176	1204	1232	1260	1288	1316
2912	3004	3096	3188	3280	3372	3464	3556
4704	4860	5016	5172	5328	5484	5640	5796
6496	6716	6936	7156	7376	7596	7816	8036
8288	8572	8856	9140	9424	9708	9992	10276
10080	10428	10776	11124	11472	11820	12168	12516
11872	12284	12696	13108	13520	13932	14344	14756
13664	14140	14616	15092	15568	16044	16520	16996

Figure 5.33: Final 8×8 Matrix Multiplication Result Stored in TMM_memory at Time = 13,740 ns.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
1	1120	1148	1176	1204	1232	1260	1288	1316
2	2912	3004	3096	3188	3280	3372	3464	3556
3	4704	4860	5016	5172	5328	5484	5640	5796
4	6496	6716	6936	7156	7376	7596	7816	8036
5	8288	8572	8856	9140	9424	9708	9992	10276
6	10080	10428	10776	11124	11472	11820	12168	12516
7	11872	12284	12696	13108	13520	13932	14344	14756
8	13664	14140	14616	15092	15568	16044	16520	16996

Figure 5.34: 8×8 Matrix Result from Online Calculator (Input: Values 0 to 63).

In addition to computing full 8×8 matrices, the tiled matrix multiplication mode

implemented in this project also supports smaller matrix sizes such as 5×5, 6×6, and 7×7. The computed results for these sizes at time = 13,740 ns are shown in Figures 5.35, 5.37, and 5.39, each displayed in an 8×8 matrix layout. It can be observed that zero-padding via the Tiled Systolic Data Setup Unit functions correctly—rows and columns beyond the actual matrix size yield results of zero, as expected. For comparison, Figures 5.36, 5.38, and 5.40 show the expected results generated using an online matrix multiplication tool. The input matrices consist of values ranging from 0 to 24 for 5×5, 0 to 35 for 6×6, and 0 to 48 for 7×7.

Value at Accumulator (in 8x8 Matrix Form):

150	160	170	180	190	0	0	0
400	435	470	505	540	0	0	0
650	710	770	830	890	0	0	0
900	985	1070	1155	1240	0	0	0
1150	1260	1370	1480	1590	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 5.35: 5×5 Matrix Result Stored in TMM_memory at Time = 13,740 ns.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
1	150	160	170	180	190	0	0	0
2	400	435	470	505	540	0	0	0
3	650	710	770	830	890	0	0	0
4	900	985	1070	1155	1240	0	0	0
5	1150	1260	1370	1480	1590	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

Figure 5.36: 5×5 Matrix Result from Online Calculator (Input: Values 0 to 24).

Value at Accumulator (in 8x8 Matrix Form):

330	345	360	375	390	405	0	0
870	921	972	1023	1074	1125	0	0
1410	1497	1584	1671	1758	1845	0	0
1950	2073	2196	2319	2442	2565	0	0
2490	2649	2808	2967	3126	3285	0	0
3030	3225	3420	3615	3810	4005	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 5.37: 6×6 Matrix Result Stored in TMM_memory at Time = 13,740 ns.

	C1	C2	C3	C4	C5	C6	C7	C8
1	330	345	360	375	390	405	0	0
2	870	921	972	1023	1074	1125	0	0
3	1410	1497	1584	1671	1758	1845	0	0
4	1950	2073	2196	2319	2442	2565	0	0
5	2490	2649	2808	2967	3126	3285	0	0
6	3030	3225	3420	3615	3810	4005	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

Figure 5.38: 6×6 Matrix Result from Online Calculator (Input: Values 0 to 35).

Value at Accumulator (in 8x8 Matrix Form):

637	658	679	700	721	742	763	0
1666	1736	1806	1876	1946	2016	2086	0
2695	2814	2933	3052	3171	3290	3409	0
3724	3892	4060	4228	4396	4564	4732	0
4753	4970	5187	5404	5621	5838	6055	0
5782	6048	6314	6580	6846	7112	7378	0
6811	7126	7441	7756	8071	8386	8701	0
0	0	0	0	0	0	0	0

Figure 5.39: 7×7 Matrix Result Stored in TMM_memory at Time = 13,740 ns.

	C1	C2	C3	C4	C5	C6	C7	C8
1	637	658	679	700	721	742	763	0
2	1666	1736	1806	1876	1946	2016	2086	0
3	2695	2814	2933	3052	3171	3290	3409	0
4	3724	3892	4060	4228	4396	4564	4732	0
5	4753	4970	5187	5404	5621	5838	6055	0
6	5782	6048	6314	6580	6846	7112	7378	0
7	6811	7126	7441	7756	8071	8386	8701	0
8	0	0	0	0	0	0	0	0

Figure 5.40: 7×7 Matrix Result from Online Calculator (Input: Values 0 to 48).

After 64 results have been stored in the TMM_memory, the Accumulator transitions to the next state, where it performs a matrix-wise comparison to determine the overall maximum value across the result matrix. This process begins at time = 13,750 ns, as shown in Figure 5.41. Once the maximum value is computed, the Accumulator forwards both the result values and the maximum value to the Activation Normalization Unit (AN_Unit), as illustrated in Figure 5.42.

Although this implementation requires large memory structures in both the Accumulator and the AN_Unit, it ensures correct timing alignment and prevents undefined values (x) from appearing in subsequent processing stages. Figure 5.43 shows that all values are successfully loaded into the AN_Unit and arranged in the correct 8×8 matrix order to preserve computation accuracy.

```
State: Storing MaxVal1...
time = 13750, clk = 1, IR_addr = 02, IR_out: 2000
time = 13760, clk = 0, IR_addr = 02, IR_out: 2000
State: Storing MaxVal2...
time = 13770, clk = 1, IR_addr = 02, IR_out: 2000
time = 13780, clk = 0, IR_addr = 02, IR_out: 2000
State: Storing MaxVal1...
time = 13790, clk = 1, IR_addr = 02, IR_out: 2000
time = 13800, clk = 0, IR_addr = 02, IR_out: 2000
State: Storing MaxVal2...
time = 13810, clk = 1, IR_addr = 02, IR_out: 2000
time = 13820, clk = 0, IR_addr = 02, IR_out: 2000
```

Figure 5.41: Accumulator Begins Matrix-Wise Maximum Value Comparison at Time = 13,750 ns.

```
State: Finished storing MaxVal and sending to AN_Unit
time = 14070, clk = 1, IR_addr = 02, IR_out: 2000
time = 14080, clk = 0, IR_addr = 02, IR_out: 2000
Check the Max Val at Accumulator(r1, r2, r3, r4, All):  7156 15092  8036 16996 16996
State: Values storing from Accumulator to AN_Unit1...
State: Values storing from Accumulator to AN_Unit1...
time = 14090, clk = 1, IR_addr = 02, IR_out: 2000
time = 14100, clk = 0, IR_addr = 02, IR_out: 2000
State: Values storing from Accumulator to AN_Unit2...
time = 14110, clk = 1, IR_addr = 02, IR_out: 2000
time = 14120, clk = 0, IR_addr = 02, IR_out: 2000
State: Values storing from Accumulator to AN_Unit1...
time = 14130, clk = 1, IR_addr = 02, IR_out: 2000
time = 14140, clk = 0, IR_addr = 02, IR_out: 2000
State: Values storing from Accumulator to AN_Unit2...
```

Figure 5.42: Results and Maximum Value Transferred from Accumulator to Activation Normalization Unit.

```
State: Values stored in AN_Unit!
State: Values stored in AN_Unit!
time = 14430, clk = 1, IR_addr = 02, IR_out: 2000
time = 14440, clk = 0, IR_addr = 02, IR_out: 2000
time = 14450, clk = 1, IR_addr = 02, IR_out: 2000
time = 14460, clk = 0, IR_addr = 02, IR_out: 2000
Check the Max Val at AN_Unit(r1, r2, r3, r4, All):  7156 15092  8036 16996 16996
Value at AN_Unit (in 8x8 Matrix Form):
 1120  1148  1176  1204  1232  1260  1288  1316
 2912  3004  3096  3188  3280  3372  3464  3556
 4704  4860  5016  5172  5328  5484  5640  5796
 6496  6716  6936  7156  7376  7596  7816  8036
 8288  8572  8856  9140  9424  9708  9992 10276
10080 10428 10776 11124 11472 11820 12168 12516
11872 12284 12696 13108 13520 13932 14344 14756
13664 14140 14616 15092 15568 16044 16520 16996
```

Figure 5.43: All Values Loaded into AN_Unit and Arranged in Correct 8×8 Matrix Format.

Since the 64 results from the 8×8 matrix multiplication has been computed and stored in the AN_Unit, the TPU Controller initiates the next operation, ‘Activate’, to apply the activation function and perform normalization (rounding). This process is essential for reducing precision and aligning with neural network computation requirements. Figure 5.44 shows the ‘Activate’ operation beginning at time = 14,470 ns.

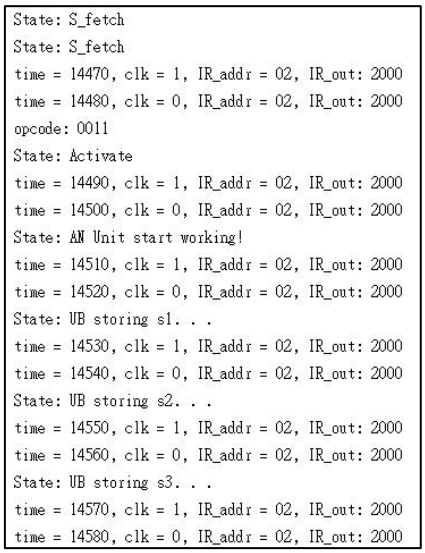


Figure 5.44: 'Activate' Operation Triggered by the TPU Controller at Time = 14,470 ns.

Because the 8×8 result matrix contains 64 values and the Unified Buffer supports only 16 memory locations, the processed data must be transferred back to the Level 1 Host Memory Activation module block by block. This avoids overwriting data and ensures that all values are preserved correctly.

Figure 5.45 shows the first block of processed results being transferred back to the Unified Buffer at time = 14,630 ns. To avoid overwriting, these values must be sent to the Level 1 Host Memory Activation module before the next block is loaded into the buffer. This is shown in Figure 5.46, where the ‘Write L1 Host Memory’ operation begins at time = 14,690 ns. By time = 15,050 ns, the first 16 values are fully stored in the Level 1 Host Memory Activation, as shown in the partial results in Figure 5.47. These new values directly overwrite their corresponding old values, while other memory locations remain unchanged.

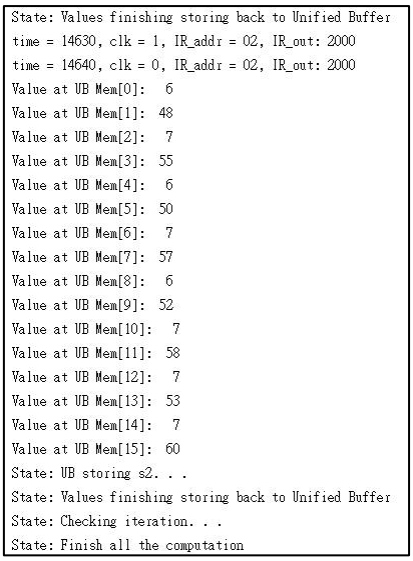


Figure 5.45: First Block of Activated and Normalized Results Transferred to the Unified Buffer at Time = 14,630 ns.

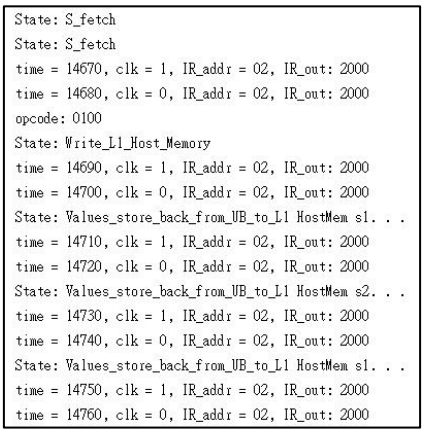


Figure 5.46: 'Write L1 Host Memory' Operation Initiated at Time = 14,690 ns.

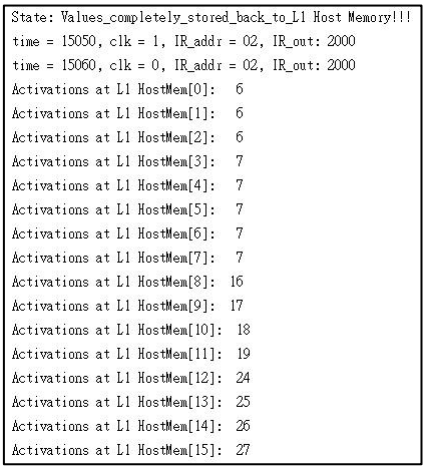


Figure 5.47: First 16 Processed Values Stored in Level 1 Host Memory Activation at Time = 15,050 ns.

By repeating the same operations—'Activate' and 'Write L1 Host Memory'—the second block of results is sent to the Unified Buffer at time = 15,230 ns, as shown in Figure 5.48. These values are then forwarded to the Level 1 Host Memory Activation at time = 15,650 ns, as shown in Figure 5.49.

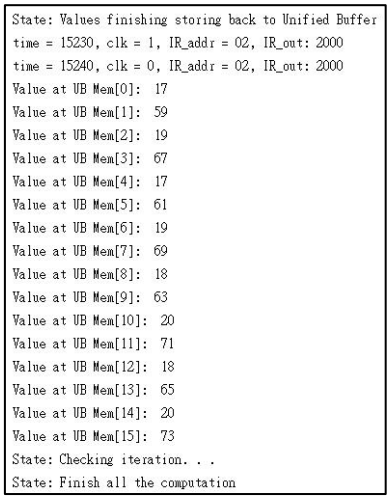


Figure 5.48: Second Block of Activated Results Sent to the Unified Buffer at Time = 15,230 ns.

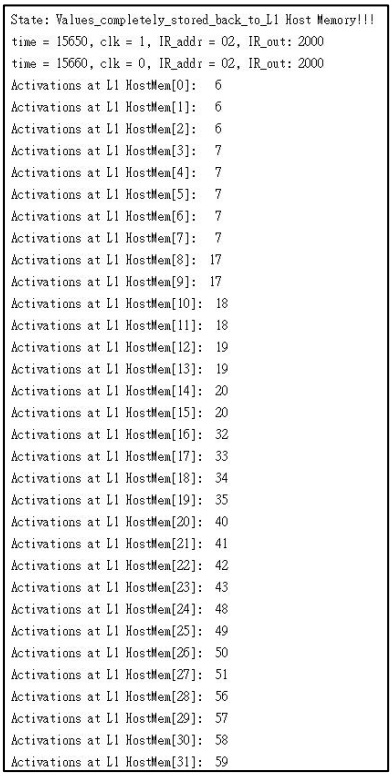


Figure 5.49: Second Block of Results Written to Level 1 Host Memory Activation at Time = 15,650 ns.

The third block of results is stored in the Unified Buffer at time = 15,830 ns, as shown in Figure 5.50, and is then forwarded to the Level 1 Host Memory Activation at time = 16,250 ns, as shown in Figure 5.51. As seen in the figures, more memory locations in the Level 1 Host Memory Activation have been overwritten with the updated results.

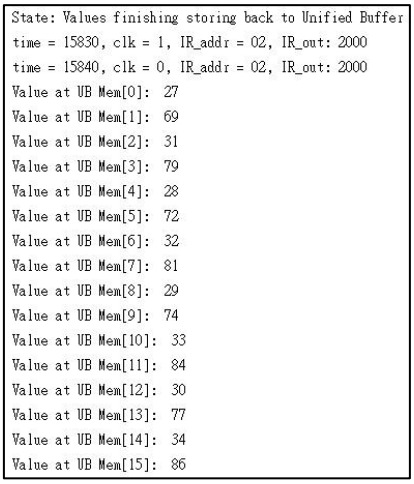


Figure 5.50: Third Block of Activated Results Stored in the Unified Buffer at Time = 15,830 ns.

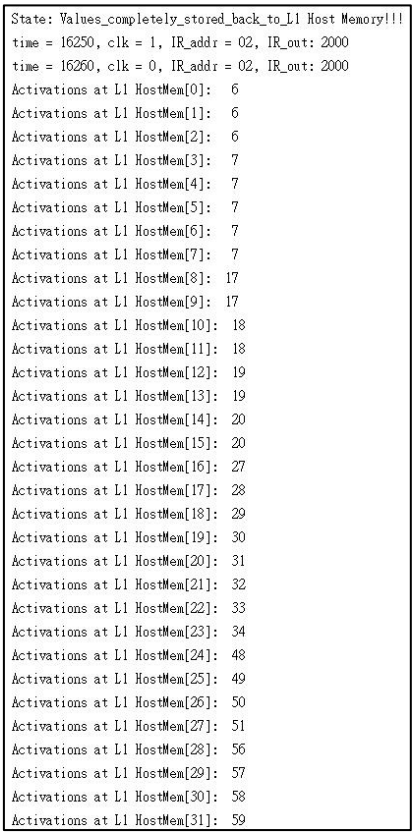


Figure 5.51: Third Block of Results Written to Level 1 Host Memory Activation at Time = 16,250 ns.

Figure 5.52 shows the final block of results being stored into the Unified Buffer at time = 16,430 ns. In this simulation example, the maximum value is located at TMM_memory[63] in AN_Unit. Since the user-defined normalization factor is set to 100, the corresponding activation and normalization step ensures that the highest result value is scaled to 100.

Figure 5.53 shows the final block of results being written back to the Level 1 Host Memory Activation at time = 16,850 ns. The partial memory layout confirms that all previous values have been updated with the newly computed and normalized results.

```
State: Values finishing storing back to Unified Buffer
time = 16430, clk = 1, IR_addr = 02, IR_out: 2000
time = 16440, clk = 0, IR_addr = 02, IR_out: 2000
Value at UB Mem[0]:  38
Value at UB Mem[1]:  80
Value at UB Mem[2]:  43
Value at UB Mem[3]:  91
Value at UB Mem[4]:  39
Value at UB Mem[5]:  83
Value at UB Mem[6]:  44
Value at UB Mem[7]:  94
Value at UB Mem[8]:  40
Value at UB Mem[9]:  85
Value at UB Mem[10]:  45
Value at UB Mem[11]:  97
Value at UB Mem[12]:  42
Value at UB Mem[13]:  88
Value at UB Mem[14]:  47
Value at UB Mem[15]: 100
```

Figure 5.52: Final Block of Results Stored in the Unified Buffer at Time = 16,430 ns.

```
State: Values_completely_stored_back_to_L1 Host Memory!!!
State: (Message from L1_Controller) TPU work completed!!!
State: (Message from L2_Controller) TPU_finished computing!!!
time = 16850, clk = 1, IR_addr = 02, IR_out: 2000
time = 16860, clk = 0, IR_addr = 02, IR_out: 2000
Activations at L1 HostMem[0]:   6
Activations at L1 HostMem[1]:   6
Activations at L1 HostMem[2]:   6
Activations at L1 HostMem[3]:   7
Activations at L1 HostMem[4]:   7
Activations at L1 HostMem[5]:   7
Activations at L1 HostMem[6]:   7
Activations at L1 HostMem[7]:   7
Activations at L1 HostMem[8]:  17
Activations at L1 HostMem[9]:  17
Activations at L1 HostMem[10]:  18
Activations at L1 HostMem[11]:  18
Activations at L1 HostMem[12]:  19
Activations at L1 HostMem[13]:  19
Activations at L1 HostMem[14]:  20
Activations at L1 HostMem[15]:  20
Activations at L1 HostMem[16]:  27
Activations at L1 HostMem[17]:  28
Activations at L1 HostMem[18]:  29
Activations at L1 HostMem[19]:  30
Activations at L1 HostMem[20]:  31
Activations at L1 HostMem[21]:  32
Activations at L1 HostMem[22]:  33
Activations at L1 HostMem[23]:  34
Activations at L1 HostMem[24]:  38
Activations at L1 HostMem[25]:  39
Activations at L1 HostMem[26]:  40
Activations at L1 HostMem[27]:  42
Activations at L1 HostMem[28]:  43
```

Figure 5.53: Final Block of Results Written to Level 1 Host Memory Activation at Time = 16,850 ns.

Since the values in the raw memory layout are difficult to interpret comprehensively, Figure 5.54 presents the stored results in an 8×8 matrix format from the Level 1 Host Memory Activation. As shown, the values have been significantly reduced through activation and normalization, compared to the original computed results shown in Figure 5.33. These reduced values can be interpreted as the updated activations in a neural network training process—where activation functions and quantization help prepare data for further propagation or learning, while keeping precision within hardware constraints.

```
Activations at L1 HostMem (8x8 Matrix Form):
  6   6   6   7   7   7   7   7
 17  17  18  18  19  19  20  20
 27  28  29  30  31  32  33  34
 38  39  40  42  43  44  45  47
 48  50  52  53  55  57  58  60
 59  61  63  65  67  69  71  73
 69  72  74  77  79  81  84  86
 80  83  85  88  91  94  97 100
```

Figure 5.54: Final 8×8 Matrix Format of Normalized Results Stored in Level 1 Host Memory Activation.

After all values are stored in the Level 1 Host Memory Activation, they need to be forwarded back to the Level 2 Host Memory Activation to update the top-level memory structure. The TPU Controller signals completion by sending a finish signal to the TMMU Controller, which then initiates the next operation. Figure 5.55 shows the 'Write L2 Host Memory' operation, performed by the TMMU Controller at time = 16,870 ns.

```
State: S_L2_fetch
State: S_L2_fetch
time = 16870, clk = 1, IR_addr = 03, IR_out: 3000
time = 16880, clk = 0, IR_addr = 03, IR_out: 3000
opcode: 0011
State: Write_L2_Host_Memory
time = 16890, clk = 1, IR_addr = 03, IR_out: 3000
time = 16900, clk = 0, IR_addr = 03, IR_out: 3000
State: Activations loading to L2 HostMem 1...
time = 16910, clk = 1, IR_addr = 03, IR_out: 3000
time = 16920, clk = 0, IR_addr = 03, IR_out: 3000
State: Activations loading to L2 HostMem 2...
time = 16930, clk = 1, IR_addr = 03, IR_out: 3000
time = 16940, clk = 0, IR_addr = 03, IR_out: 3000
State: Activations loading to L2 HostMem 1...
time = 16950, clk = 1, IR_addr = 03, IR_out: 3000
time = 16960, clk = 0, IR_addr = 03, IR_out: 3000
time = 16970, clk = 1, IR_addr = 03, IR_out: 3000
time = 16980, clk = 0, IR_addr = 03, IR_out: 3000
State: Activations Completely loaded back to L2 HostMem!!!
time = 16990, clk = 1, IR_addr = 03, IR_out: 3000
time = 17000, clk = 0, IR_addr = 03, IR_out: 3000
Activations at L2 HostMem[0]:  6
Activations at L2 HostMem[1]:  6
Activations at L2 HostMem[2]:  6
Activations at L2 HostMem[3]:  7
Activations at L2 HostMem[4]:  7
Activations at L2 HostMem[5]:  7
Activations at L2 HostMem[6]:  7
Activations at L2 HostMem[7]:  7
Activations at L2 HostMem[8]:  17
```

Figure 5.55: 'Write L2 Host Memory' Operation Initiated by the TMMU Controller at Time = 16,870 ns.

Figures 5.56 and 5.57 show the results stored in the Level 2 Host Memory Activation at time = 17,000 ns, displayed in memory layout format.

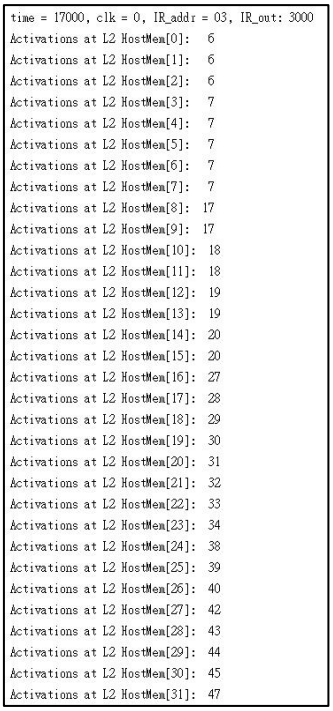


Figure 5.56: First Part ([0] to [31]) of Results in Level 2 Host Memory Activation at Time = 17,000 ns.

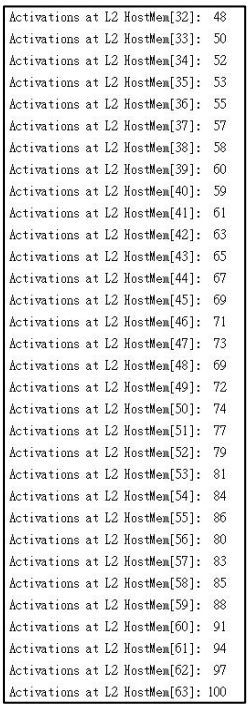


Figure 5.57: Second Part ([32] to [63]) of Results in Level 2 Host Memory Activation at Time = 17,000 ns.

Figure 5.58 illustrates that the remaining memory addresses (starting from HostMem[64]) still retain their original values, as the 64 updated activations did not overwrite these locations. If further computations are to be performed, the next values to be loaded will begin at L2 Host HostMem[64].

```
Activations at L2 HostMem[64]:  64
Activations at L2 HostMem[65]:  65
Activations at L2 HostMem[66]:  66
Activations at L2 HostMem[67]:  67
Activations at L2 HostMem[68]:  68
Activations at L2 HostMem[69]:  69
Activations at L2 HostMem[70]:  70
Activations at L2 HostMem[71]:  71
Activations at L2 HostMem[72]:  72
Activations at L2 HostMem[73]:  73
Activations at L2 HostMem[74]:  74
Activations at L2 HostMem[75]:  75
Activations at L2 HostMem[76]:  76
Activations at L2 HostMem[77]:  77
Activations at L2 HostMem[78]:  78
Activations at L2 HostMem[79]:  79
Activations at L2 HostMem[80]:  80
Activations at L2 HostMem[81]:  81
Activations at L2 HostMem[82]:  82
Activations at L2 HostMem[83]:  83
Activations at L2 HostMem[84]:  84
Activations at L2 HostMem[85]:  85
Activations at L2 HostMem[86]:  86
Activations at L2 HostMem[87]:  87
Activations at L2 HostMem[88]:  88
Activations at L2 HostMem[89]:  89
Activations at L2 HostMem[90]:  90
```

Figure 5.58: Remaining Values Starting from HostMem[64] at Time = 17,000 ns, Indicating Future Data to Be Loaded.

Figure 5.59 shows the 256 memory locations of the Level 2 Host Memory Activation displayed in a 16×16 matrix format at time = 17000 ns, illustrating both the updated values and the original pre-initialized values still present in memory.

```
Activations at L2 HostMem (16x16 Matrix Form):
  6   6   6   7   7   7   7   7  17  17  18  18  19  19  20  20
 27  28  29  30  31  32  33  34  38  39  40  42  43  44  45  47
 48  50  52  53  55  57  58  60  59  61  63  65  67  69  71  73
 69  72  74  77  79  81  84  86  80  83  85  88  91  94  97 100
 64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
 96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
```

Figure 5.59: Full 256 Memory Locations of Level 2 Host Memory Activation in 16×16 Matrix Format Showing Updated and Initial Values.

Figure 5.60 shows that when no valid instructions remain in the L2 IR Memory, the TMMU Controller transitions to the S_L2_stay state and halts, waiting for further instructions to arrive.

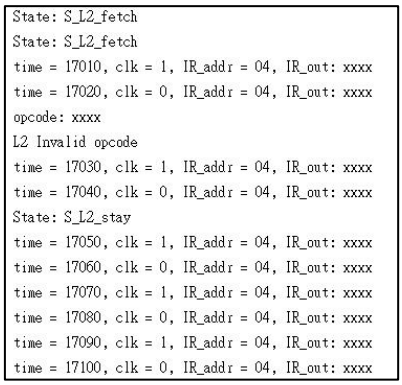


Figure 5.60: TMMU Controller Entering S_L2_stay State When No Valid Instructions Are Available in L2 IR Memory.

Up to this point, the entire section has presented the full simulation process of an 8×8 tiled matrix multiplication using a 4×4 Matrix Multiplication Unit (MMU). By walking through the simulation step by step, the dataflow across various states—as well as the reordering and storage patterns—can be understood comprehensively. For reference, Figure 5.61 shows the instruction set sent to the TMMU Controller, while Figure 5.62 shows the instructions sent to the TPU Controller.

```
initial begin
    memory[0] = 16'b0000000000000000; // Load ISA
    memory[1] = 16'b0001000001001000; // Read L2 Host Memory
    memory[2] = 16'b0010000000000000; // TPU work
    memory[3] = 16'b0011000000000000; // Write L2 Host Memory

end
```

Figure 5.61: Instruction Set Sent to the TMMU Controller During 8×8 Tiled Matrix Multiplication.

```
initial begin
    memory[0] = 16'b0000000000000000; // Read Host Mem (Activation b1)
    memory[1] = 16'b0001000000000000; // Read Weight (from Weight DDR3) (Weight b1)
    memory[2] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
    memory[3] = 16'b0000000000000000; // Read Host Mem (Activation b2)
    memory[4] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
    memory[5] = 16'b0001000000000000; // Read Weight (from Weight DDR3) (Weight b2)
    memory[6] = 16'b0000000000000000; // Read Host Mem (Activation b3)
    memory[7] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
    memory[8] = 16'b0000000000000000; // Read Host Mem (Activation b4)
    memory[9] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
    memory[10] = 16'b0000000000000000; // Read Host Mem (Activation b1)
    memory[11] = 16'b0000000000000000; // Read Host Mem (Activation b2)
    memory[12] = 16'b0000000000000000; // Read Host Mem (Activation b3)
    memory[13] = 16'b0001000000000000; // Read Weight (from Weight DDR3) (Weight b3)
    memory[14] = 16'b0000000000000000; // Read Host Mem (Activation b4)
    memory[15] = 16'b0000000000000000; // Read Host Mem (Activation b1)
    memory[16] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
    memory[17] = 16'b0000000000000000; // Read Host Mem (Activation b2)
    memory[18] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
    memory[19] = 16'b0000000000000000; // Read Host Mem (Activation b3)
    memory[20] = 16'b0000000000000000; // Read Host Mem (Activation b4)
    memory[21] = 16'b0001000000000000; // Read Weight (from Weight DDR3) (Weight b4)
    memory[22] = 16'b0000000000000000; // Read Host Mem (Activation b1)
    memory[23] = 16'b0000000000000000; // Read Host Mem (Activation b2)
    memory[24] = 16'b0000000000000000; // Read Host Mem (Activation b3)
    memory[25] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
    memory[26] = 16'b0000000000000000; // Read Host Mem (Activation b4)
    memory[27] = 16'b0010000000000000; // Computation with itr = 0 (Compute once)
    memory[28] = 16'b0011000101100100; // Activation (MaxVal Normalization)
    memory[29] = 16'b0100000000000000; // write back Host Mem (1/4)
    memory[30] = 16'b0011000101100100; // Activation (MaxVal Normalization)
    memory[31] = 16'b0100000000000000; // write back Host Mem (2/4)
    memory[32] = 16'b0011000101100100; // Activation (MaxVal Normalization)
    memory[33] = 16'b0100000000000000; // write back Host Mem (3/4)
    memory[34] = 16'b0011000101100100; // Activation (MaxVal Normalization)
    memory[35] = 16'b0100000000000000; // write back Host Mem (4/4)
    memory[36] = 16'b1111111111111111; // END instruction
end
```

Figure 5.62: Instruction Set Sent to the TPU Controller During 8×8 Tiled Matrix Multiplication.

5.1.2 Basic Matrix Multiplication (1×1 to 4×4)

Since the TMMU developed in this project also supports basic matrix multiplication (BMM) when the target matrix size does not exceed the 4×4 MMU dimensions, the overall state transitions and dataflow closely resemble those described in the tiled matrix multiplication section. To avoid redundant explanations and figures, this section focuses on key result snapshots.

The following figures present the computed results stored in the Activation Normalization Unit (AN_Unit) at time = 2230 ns, alongside outputs from an online matrix calculator to verify correctness. Additional figures show how these results are written back to the Level 2 Host Memory by time = 3030 ns. These visualizations not only confirm functional accuracy but also help identify opportunities for future optimization to make the TMMU more efficient and comprehensive.

Figure 5.63 shows the computed 4×4 results stored in the AN_Unit in matrix format at time = 2230 ns. Figure 5.64 presents the corresponding 4×4 result calculated using an online matrix calculator, where the input matrices consist of values ranging from 0 to 15.

```
time = 2230, clk = 1, IR_addr = 02, IR_out: 2000
time = 2240, clk = 0, IR_addr = 02, IR_out: 2000
Check the Max Val at AN_Unit(r1, r2, r3, r4, All):    74   218   362   506   506
Value at AN_Unit (in 4x4 Matrix Form):
   56    62    68    74
  152   174   196   218
  248   286   324   362
  344   398   452   506
```

Figure 5.63: Computed 4×4 Matrix Result Stored in AN_Unit at Time = 2230 ns.

	C_1	C_2	C_3	C_4
1	56	62	68	74
2	152	174	196	218
3	248	286	324	362
4	344	398	452	506

Figure 5.64: 4×4 Matrix Result from Online Calculator (Input: Values 0 to 15).

Following the same steps, the computed results are passed through the activation function and normalized based on the user-defined boundary. They are then forwarded sequentially through the Unified Buffer, stored in the Level 1 Host Memory Activation, and ultimately written back into the Level 2 Host Memory Activation. Figure 5.65 shows the final results stored in the Level 2 Host Memory Activation at time = 3030 ns.

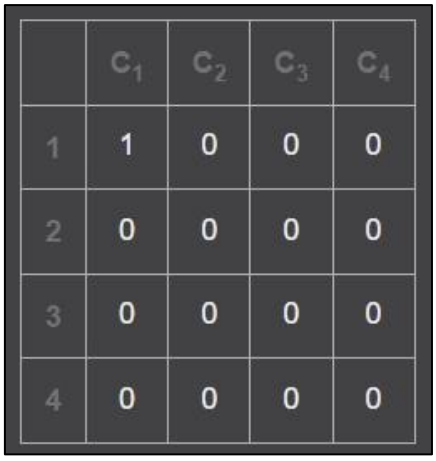
```
State: Activations Completely loaded back to L2 HostMem!!!
time = 3030, clk = 1, IR_addr = 03, IR_out: 3000
time = 3040, clk = 0, IR_addr = 03, IR_out: 3000
Activations at L2 HostMem (16x16 Matrix Form):
 11  12  13  14  30  34  38  43  49  56  64  71  67  78  89 100
 16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
 32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
 48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
 64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
 96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
```

Figure 5.65: Final Results Stored in Level 2 Host Memory Activation with Updated 16 Values at Time = 3030 ns.

Figure 5.66 to 5.68 present the simulation of a 1×1 matrix multiplication. Figure 5.66 shows the computed result stored in the AN_Unit at time = 2230 ns in matrix format. Figure 5.67 displays the corresponding 1×1 result calculated using an online matrix calculator. To avoid zero multiplication errors, both input matrices consist of the value 1. Finally, Figure 5.68 shows the updated result stored in the Level 2 Host Memory Activation at time = 3030 ns.

```
time = 2230, clk = 1, IR_addr = 02, IR_out: 2000
time = 2240, clk = 0, IR_addr = 02, IR_out: 2000
Check the Max Val at AN_Unit(r1, r2, r3, r4, All):    1    0    0    0    1
Value at AN_Unit (in 4x4 Matrix Form):
    1    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
```

Figure 5.66: Computed 1×1 Matrix Result Stored in AN_Unit at Time = 2230 ns.



	C_1	C_2	C_3	C_4
1	1	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

Figure 5.67: 1×1 Matrix Result from Online Calculator (Input: Value 1).

```
State: Activations Completely loaded back to L2 HostMem!!!
time = 3030, clk = 1, IR_addr = 03, IR_out: 3000
time = 3040, clk = 0, IR_addr = 03, IR_out: 3000
Activations at L2 HostMem (16x16 Matrix Form):
100   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 32   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 48   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
 96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
```

Figure 5.68: Final results stored in Level 2 Host Memory Activation with updated values for 1×1 matrix multiplication at time = 3030 ns.

Figures 5.69 to 5.71 present the simulation results of a 2×2 matrix multiplication. Figure 5.69 shows the computed results stored in the AN_Unit at time = 2230 ns in matrix format. Figure 5.70 displays the corresponding 2×2 result calculated using an online matrix calculator. Figure 5.71 shows the updated results stored in the Level 2 Host Memory Activation at time = 3030 ns. The input matrices consist of values ranging from 0 to 3.

```
time = 2230, clk = 1, IR_addr = 02, IR_out: 2000
time = 2240, clk = 0, IR_addr = 02, IR_out: 2000
Check the Max Val at AN_Unit(r1, r2, r3, r4, All):     3    11     0     0    11
Value at AN_Unit (in 4x4 Matrix Form):
    2     3     0     0
    6    11     0     0
    0     0     0     0
    0     0     0     0
```

Figure 5.69: Computed 2×2 Matrix Result Stored in AN_Unit at Time = 2230 ns.

	C_1	C_2	C_3	C_4
1	2	3	0	0
2	6	11	0	0
3	0	0	0	0
4	0	0	0	0

Figure 5.70: 2×2 Matrix Result from Online Calculator (Input: Values 0 to 3).

```
State: Activations Completely loaded back to L2 HostMem!!!
time = 3030, clk = 1, IR_addr = 03, IR_out: 3000
time = 3040, clk = 0, IR_addr = 03, IR_out: 3000
Activations at L2 HostMem (16x16 Matrix Form):
 18  27   0   0  54 100   0   0   0   0   0   0   0   0   0   0
 16  17   0   0  18  19   0   0   0   0   0   0   0   0   0   0
 32  33   0   0  34  35   0   0   0   0   0   0   0   0   0   0
 48  49   0   0  50  51   0   0   0   0   0   0   0   0   0   0
 64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
 96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
```

Figure 5.71: Final results stored in Level 2 Host Memory Activation with updated values for 2×2 matrix multiplication at time = 3030 ns.

Figures 5.72 to 5.74 present the simulation results of a 3×3 matrix multiplication. Figure 5.72 shows the computed results stored in the AN_Unit at time = 2230 ns in matrix

format. Figure 5.73 displays the corresponding 3×3 result calculated using an online matrix calculator. Figure 5.74 shows the updated results stored in the Level 2 Host Memory Activation at time = 3030 ns. The input matrices consist of values ranging from 0 to 8.

```
time = 2230, clk = 1, IR_addr = 02, IR_out: 2000
time = 2240, clk = 0, IR_addr = 02, IR_out: 2000
Check the Max Val at AN_Unit(r1, r2, r3, r4, All):    21    66   111     0   111
Value at AN_Unit (in 4x4 Matrix Form):
   15    18    21     0
   42    54    66     0
   69    90   111     0
    0     0     0     0
```

Figure 5.72: Computed 3×3 Matrix Result Stored in AN_Unit at Time = 2230 ns.

	C_1	C_2	C_3	C_4
1	15	18	21	0
2	42	54	66	0
3	69	90	111	0
4	0	0	0	0

Figure 5.73: 3×3 Matrix Result from Online Calculator (Input: Values 0 to 8).

```
State: Activations Completely loaded back to L2 HostMem!!!
time = 3030, clk = 1, IR_addr = 03, IR_out: 3000
time = 3040, clk = 0, IR_addr = 03, IR_out: 3000
Activations at L2 HostMem (16x16 Matrix Form):
 13  16  18   0  37  48  59   0  62  81 100   0   0   0   0   0
 16  17  18   0  19  20  21   0  22  23  24   0   0   0   0   0
 32  33  34   0  35  36  37   0  38  39  40   0   0   0   0   0
 48  49  50   0  51  52  53   0  54  55  56   0   0   0   0   0
 64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
 96  97  98  99 100 101 102 103 104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
```

Figure 5.74: Final results stored in Level 2 Host Memory Activation with updated values for 3×3 matrix multiplication at time = 3030 ns.

A future optimization that must be addressed is the presence of zero-filled values in unexpected memory locations beyond HostMem[15], as observed in the 1×1, 2×2, and 3×3 matrix multiplication results stored in the Level 2 Host Memory Activation. There are several possible causes for this issue:

1. The testbench used for simulation is tailored for 4×4 matrix operations.
2. The instruction set loaded into the TPU is also configured for 4×4 matrix sizes.
3. The Tiled Systolic Data Setup Unit automatically fills unused memory locations with zeros, which are then propagated and stored in the Level 2 Host Memory Activation.

To address this, future work should include developing instruction sets and logic specifically optimized for smaller matrices (e.g., 1×1 to 3×3). This will ensure that components operate more efficiently and only load the values required for computation. For instance, a 1×1 matrix multiplication should not require loading and zero-filling 16 values. By sending only the necessary values and eliminating redundant zeros, the design will be more efficient and more closely aligned with real-world applications.

5.2 Power Analysis

The power analysis summary for each component, generated by Xilinx Vivado, is provided in this section to evaluate the power performance of the TMMU. These results help illustrate the relationship between component complexity and corresponding power consumption. To determine whether the reported power values are efficient or not, further comparison with similar hardware designs is necessary. For now, the presented values serve primarily as reference points.

During the power analysis process, an important issue was discovered that must be addressed in future work. For accurate power analysis, the Verilog code must successfully pass synthesis and implementation in Xilinx Vivado. However, certain modules in this project were written in a way that caused multiple driver net conflicts—an issue that does not impact simulation but prevents successful synthesis. This must be resolved before the design can move forward toward physical implementation.

Further discussion of this issue is provided in Section 5.4. Figure 5.75 shows the environmental setup used for power reporting, including parameters such as ambient temperature, airflow, and other relevant factors. The subsequent figures present power analysis summaries for each individual component in the TMMU, along with the total on-chip power consumption of the entire system.

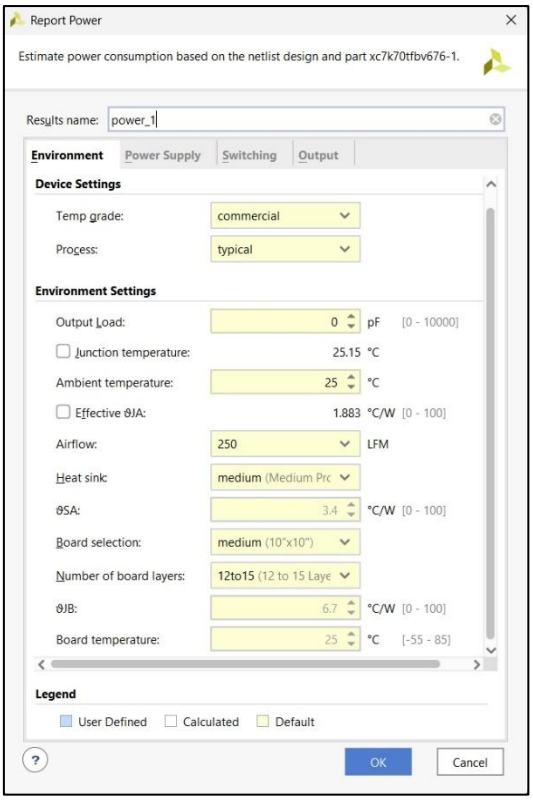


Figure 5.75: Power Analysis Environment Settings.

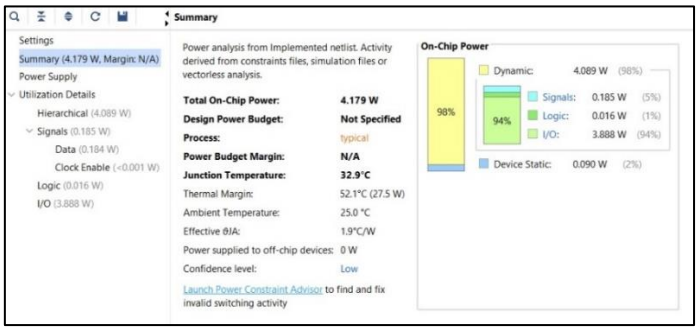


Figure 5.76: Power Analysis Summary for Level 2 IR Memory.

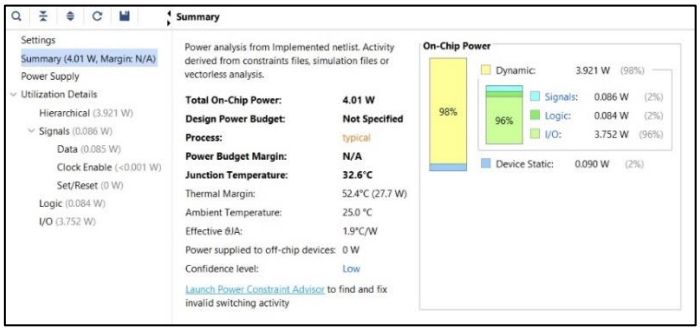


Figure 5.77: Power Analysis Summary for Level 2 IR Counter.

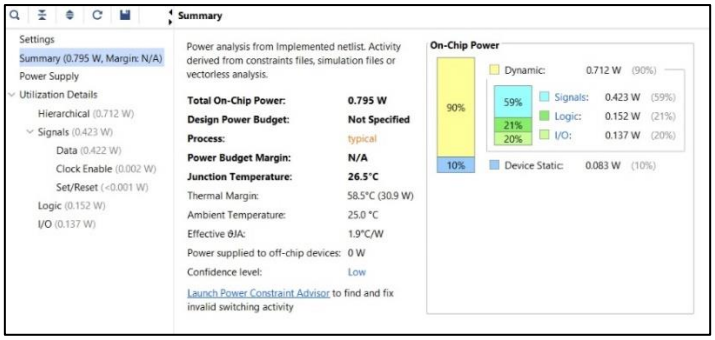


Figure 5.78: Power Analysis Summary for Level 2 Host Memory Instruction Set.

As shown in Figures 5.79 and 5.80, the total on-chip power for both Level 2 Host Memories (Weight and Activation) is relatively high, around 21–22 W. This may be due to the pre-initialized values stored within these memory structures. In contrast, as shown in Figures 5.83 and 5.84, the Level 1 Host Memories consume noticeably less power.

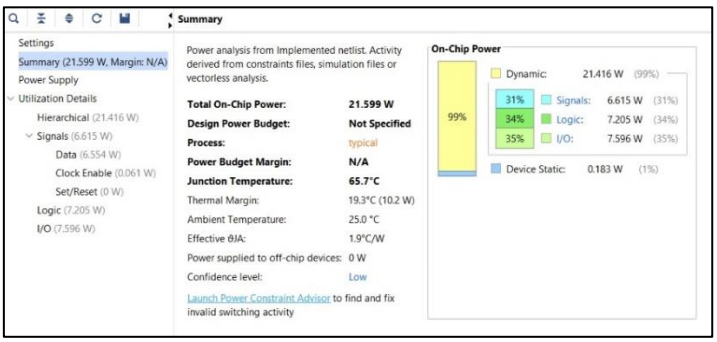


Figure 5.79: Power Analysis Summary for Level 2 Host Memory Weight.

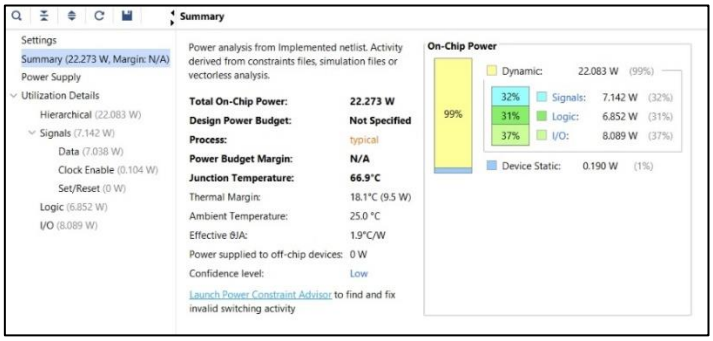


Figure 5.80: Power Analysis Summary for Level 2 Host Memory Activation.

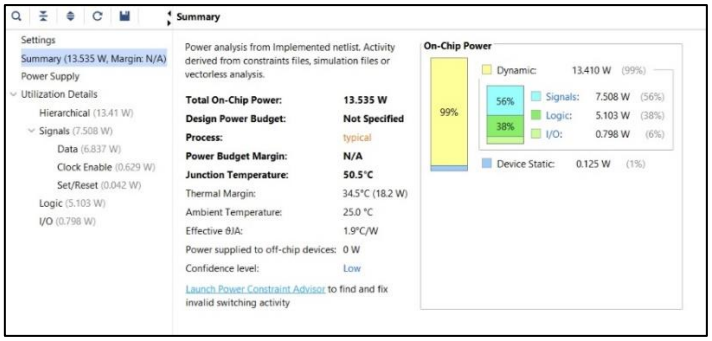


Figure 5.81: Power Analysis Summary for Tiled Systolic Data Setup Unit.

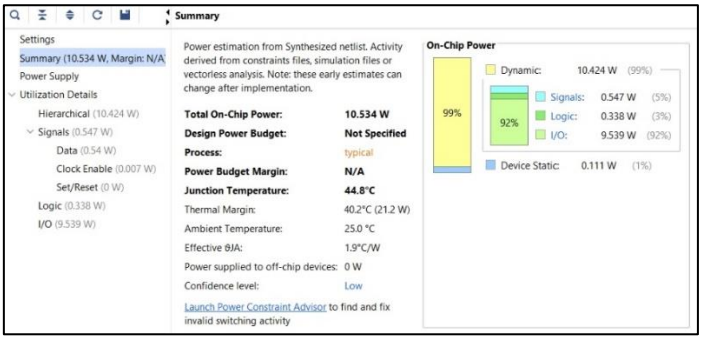


Figure 5.82: Power Analysis Summary for Level 2 Controller (TMMU Controller).

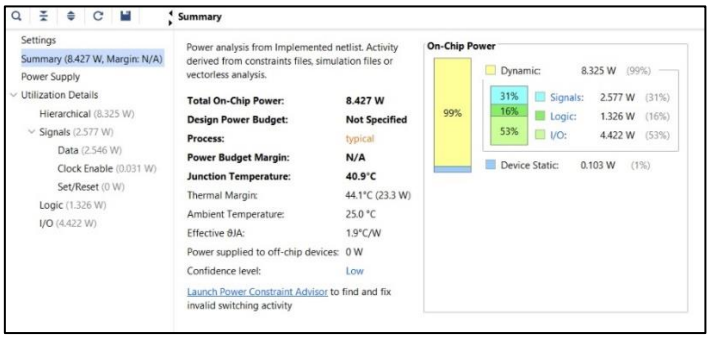


Figure 5.83: Power Analysis Summary for Level 1 Host Memory Weight.

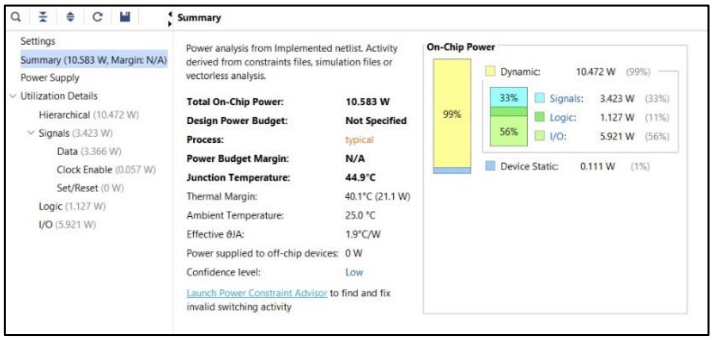


Figure 5.84: Power Analysis Summary for Level 1 Host Memory Activation.

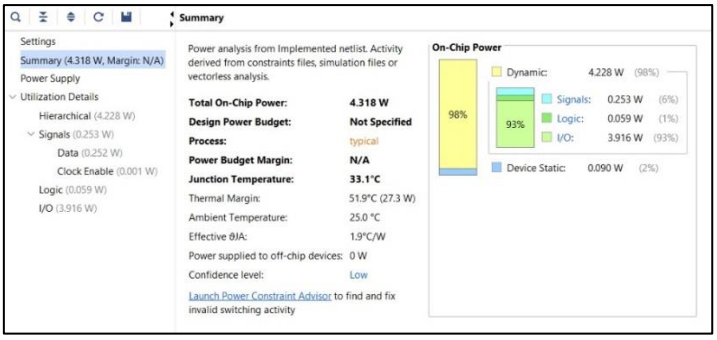


Figure 5.85: Power Analysis Summary for Level 1 IR Memory.

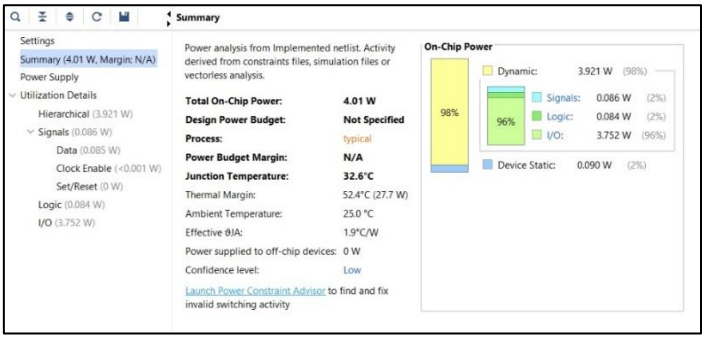


Figure 5.86: Power Analysis Summary for Level 1 IR Counter.

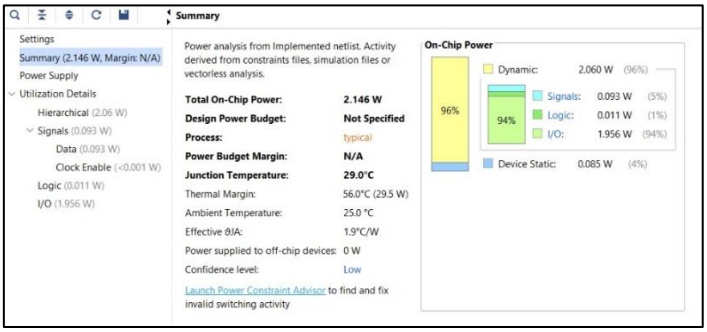


Figure 5.87: Power Analysis Summary for Weight DDR3.

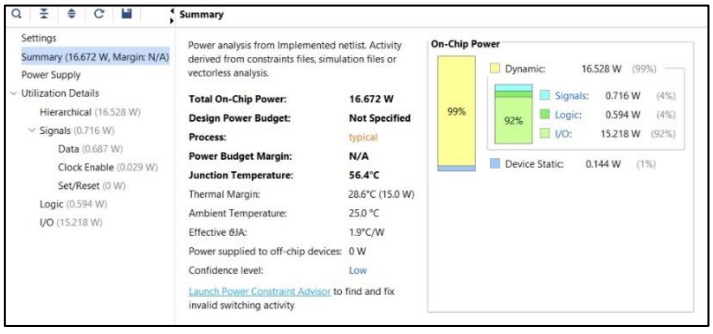


Figure 5.88: Power Analysis Summary for Weight Interface.

Both the Weight FIFO and Activation FIFO, as shown in Figures 5.89 and 5.92, have identical on-chip power consumption. This demonstrates that identical designs yield consistent power analysis results.

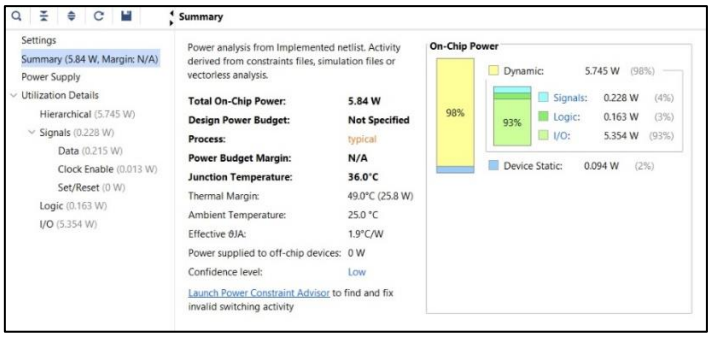


Figure 5.89: Power Analysis Summary for Weight FIFO.

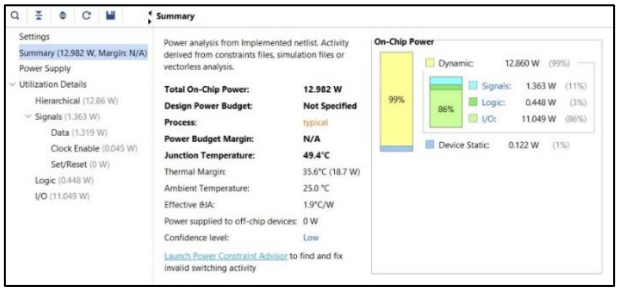


Figure 5.90: Power Analysis Summary for Unified Buffer.

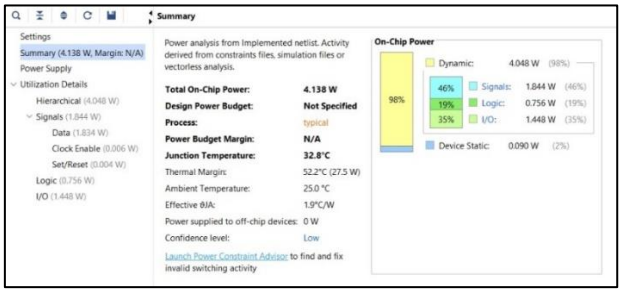


Figure 5.91: Power Analysis Summary for Systolic Data Setup Unit.

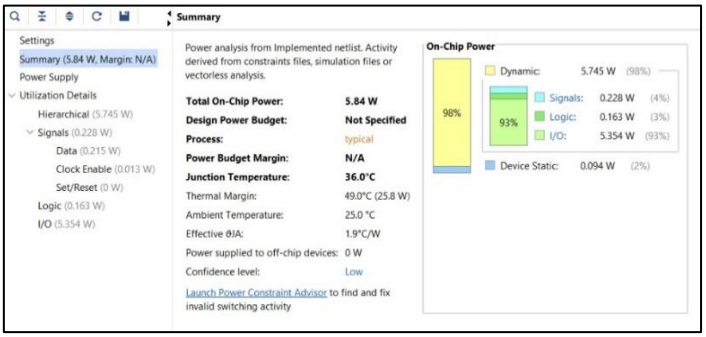


Figure 5.92: Power Analysis Summary for Activation FIFO.

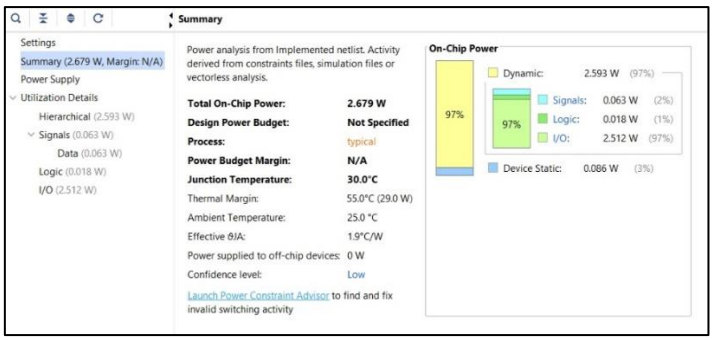


Figure 5.93: Power Analysis Summary for Row Detector.

As mentioned earlier in Section 4.2.2, the Systolic Cell implementation encountered a multiple-driven nets issue, primarily due to a lack of familiarity with HDL constraints prior to performing power analysis. The error messages can be viewed in Figure 5.95. This issue prevents correct implementation of the design, leading to inaccurate power estimation results. To ensure valid synthesis and power analysis, future work must address this problem by rewriting the affected modules using a coding style that adheres to proper HDL design rules and avoids such conflicts.

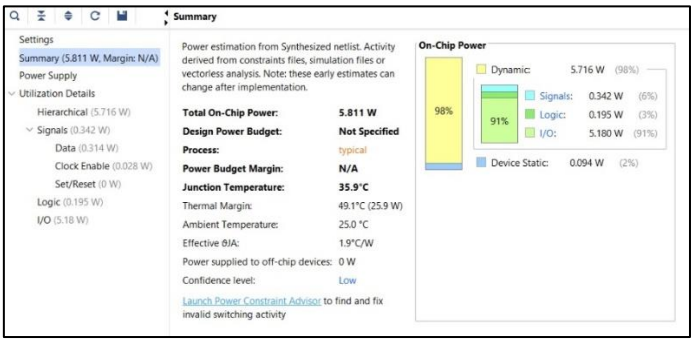


Figure 5.94: Power Analysis Summary for Systolic Cell.

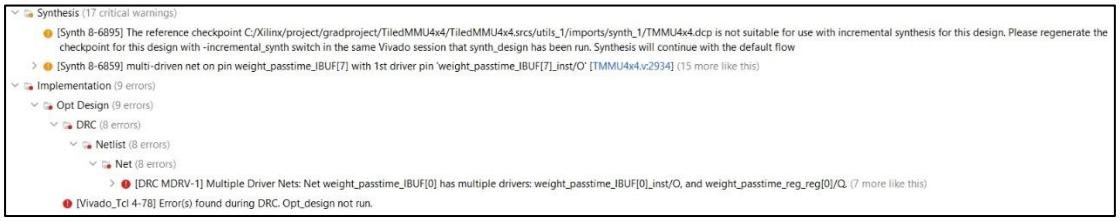


Figure 5.95: Implementation Error Messages for Systolic Cell.

Since the Matrix Multiplication Unit (MMU) is constructed as an array of 16 Systolic Cells, the multiple-driven net issue present in the Systolic Cell design also affects the entire MMU, as showing in Figure 5.96 and 5.97. Therefore, resolving this issue in the Systolic Cell should be prioritized in future work. Once the Systolic Cell is corrected, the MMU may function properly as well—provided no additional multi-driven conflicts exist in other interconnected nets.

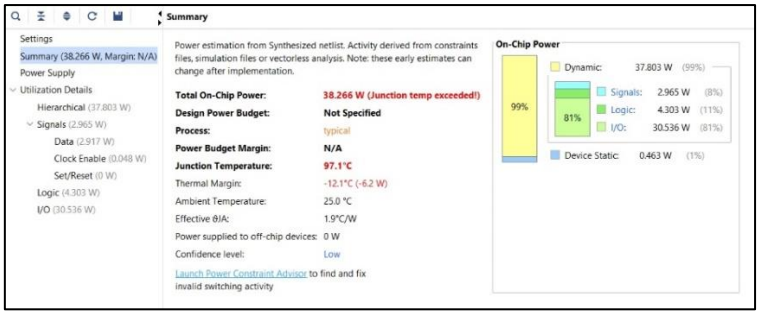


Figure 5.96: Power Analysis Summary for Matrix Multiplication Unit.

Figure 5.97: Implementation Error Messages for Matrix Multiplication Unit.

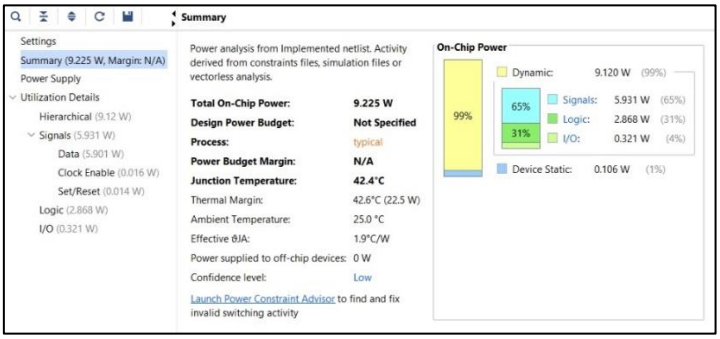


Figure 5.98: Power Analysis Summary for Accumulator.

Although the implementation of the AN_Unit does not encounter multiple-driven net issues, its complexity poses another challenge. This module contains numerous internal instances and performs heavy computations—including multiplication and division—within each case statement. As a result, Xilinx Vivado is unable to generate an accurate power analysis, and a junction temperature exceeded warning is triggered during estimation. Future work should focus on reducing the computational workload within the AN_Unit or introducing supporting modules to distribute the processing load more effectively.

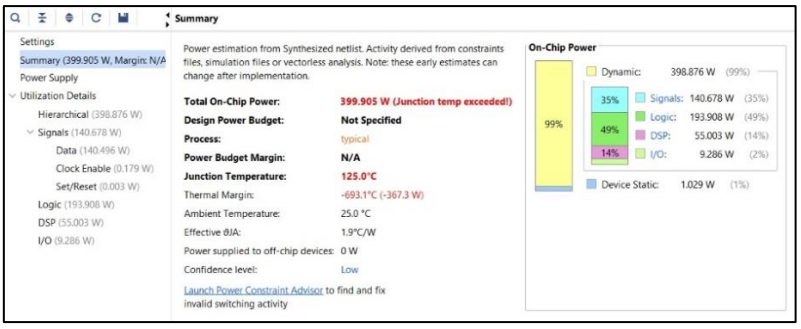


Figure 5.99: Power Analysis Summary for Activation Normalization Unit.

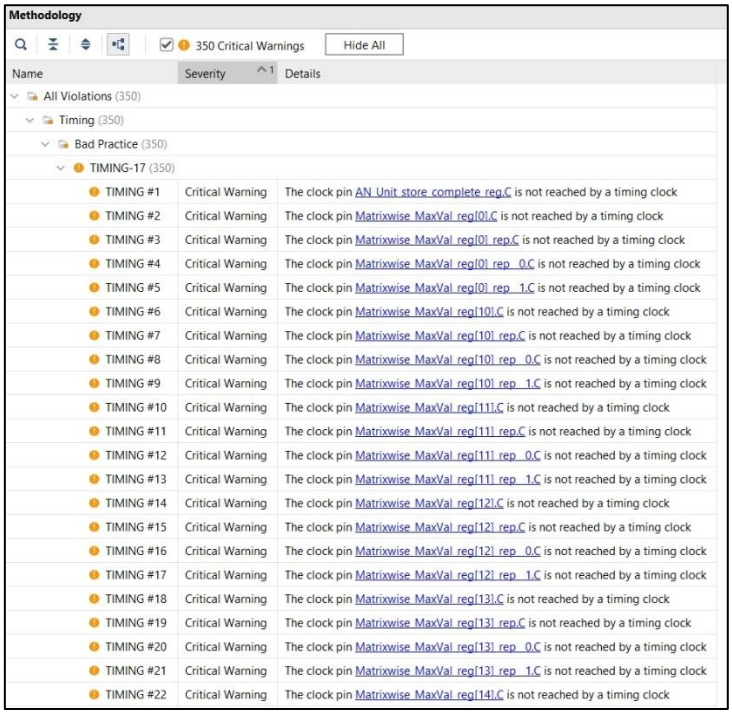


Figure 5.100: Critical Warnings for Activation Normalization Unit.

Although the TMMU Controller did not encounter multiple-driven net issues—thanks to its relatively simple design with fewer counters and states—the TPU Controller did experience such problems. This was primarily due to certain counters being updated across multiple always blocks, which resulted in conflicting assignments. Future work should focus on reorganizing the TPU Controller to ensure that each counter is managed within a single always block. Alternatively, exploring different Verilog coding styles and design patterns could help address these issues more effectively.

One potential solution is to delegate some counters—such as those used for address tracking—to their respective components. In this approach, the TPU Controller would simply assert a trigger signal, while the counter logic would reside within the target module. This would reduce complexity within the controller itself and minimize the

risk of multi-driven net conflicts.

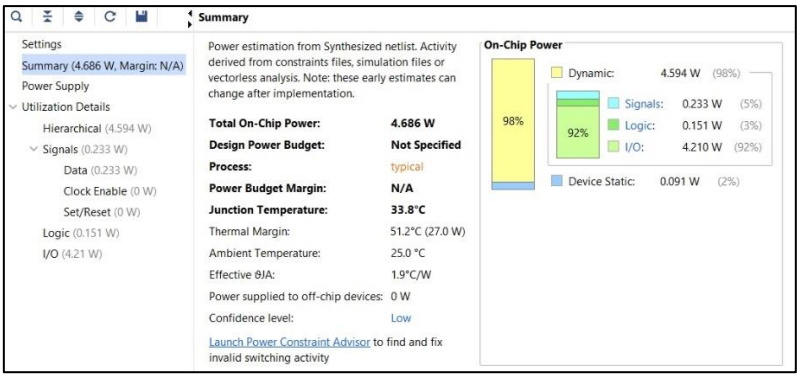


Figure 5.101: Power Analysis Summary for Level 1 Controller.

Figure 5.102: Implementation Error Messages for Level 1 Controller.

Since key components within the TPU—specifically the MMU and the TPU Controller—encounter multiple-driven net issues, the overall on-chip power analysis for the TPU is also affected. These conflicts prevent accurate synthesis and result in an unreliable power estimation. Figure 5.103 shows the incorrect power analysis report generated for the TPU, while Figure 5.104 presents the corresponding error messages encountered during the implementation process.

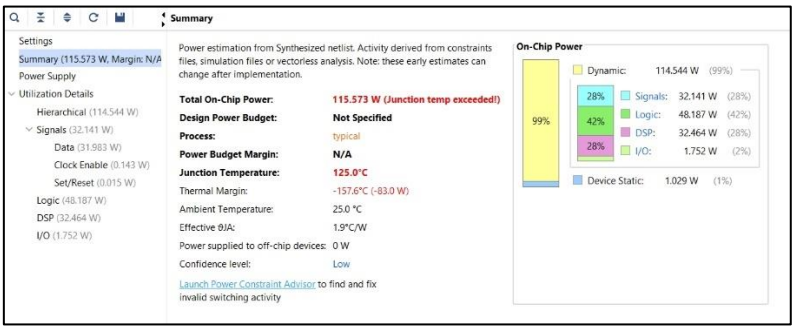


Figure 5.103: Power Analysis Summary for TPU.

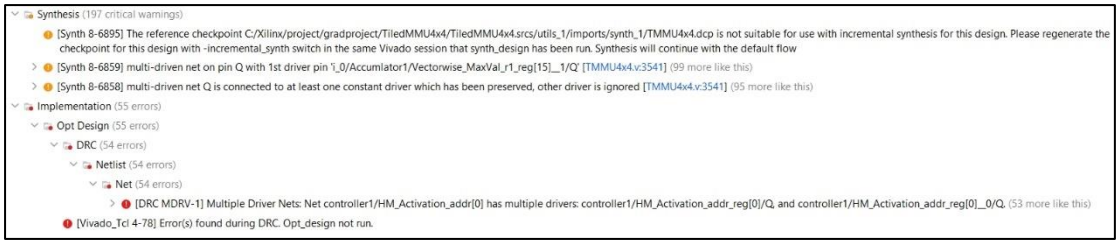


Figure 5.104: Implementation Error Messages for TPU.

The TMMU on-chip power analysis revealed an unusual issue: rather than reporting multiple-driven net errors, Xilinx Vivado generated an “empty design” error. Despite all I/O ports and component connections being thoroughly verified, the root cause of this issue may still stem from unresolved multiple-driven net conflicts within the TPU (Level 1 layer). These unresolved hardware conflicts may prevent Vivado from properly synthesizing the design, ultimately leading it to treat the entire module as empty during power analysis. Future work should prioritize resolving these issues within the TPU to determine whether correcting the lower-level conflicts eliminates the empty design error.

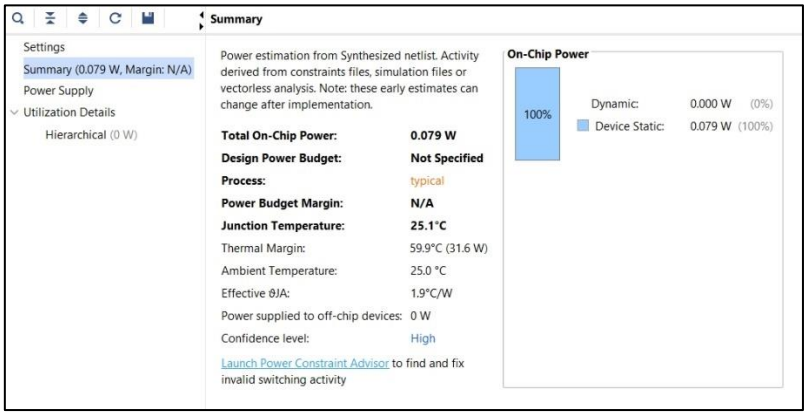


Figure 5.105: Power Analysis Summary for TMMU.

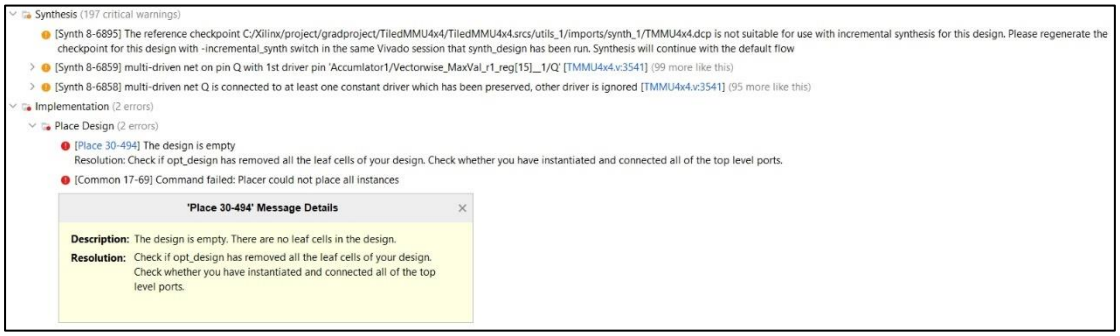


Figure 5.106: Implementation Error Messages for TMMU.

As demonstrated in the previous section, several critical coding issues—such as multiple-driven nets—have led to incorrect power analysis and prevented successful

design implementation. It is important to highlight that, once these issues were identified, significant time and effort were devoted to resolving them in pursuit of a complete and accurate power analysis. Unfortunately, due to time constraints, these issues remain unresolved and are left as future work opportunities for further improvement.

5.3 Floorplanning and Physical Design

To bridge the gap between RTL-level coding and practical hardware realization—and to explore potential opportunities for fabrication—this project includes floorplanning, timing analysis, and connectivity and geometry verification. These steps were completed using Cadence Innovus. This section presents the floorplanning diagrams for each major component within the TMMU design. Additional figures, such as detailed violation reports and routing information, are provided in the supplementary section for reference.

Through the floorplanning process, it became evident that the more complex a design is, the more likely it is to encounter violations during the routing stage. These violations highlight critical areas that require further investigation in order to resolve and optimize physical layout constraints effectively. Figure 5.107 shows the floorplanning configuration window, where key parameters—such as Core Utilization and Core Margin—can be adjusted. These settings directly impact chip area compactness and can influence the number and severity of layout violations.

In this project, the Core Utilization is uniformly set to 0.5, and the Core Margin is set to 10 units on each boundary to maintain consistency across all component floorplans. Any modules that require different parameter settings for layout optimization will be explicitly discussed in their respective sections.

Additionally, a more significant challenge emerged during the netlist generation phase. This issue illustrates a broader barrier faced by students and entry-level engineers in this field: access to professional EDA tools is often limited, and understanding how to use them effectively frequently requires industry-specific knowledge. Without access to proper training or guidance, learners may struggle to gain meaningful hands-on experience with real-world digital implementation workflows—unless they are part of an institution or organization that provides the necessary support.

Figure 5.107: Floorplanning Configuration Window in Cadence Innovus.

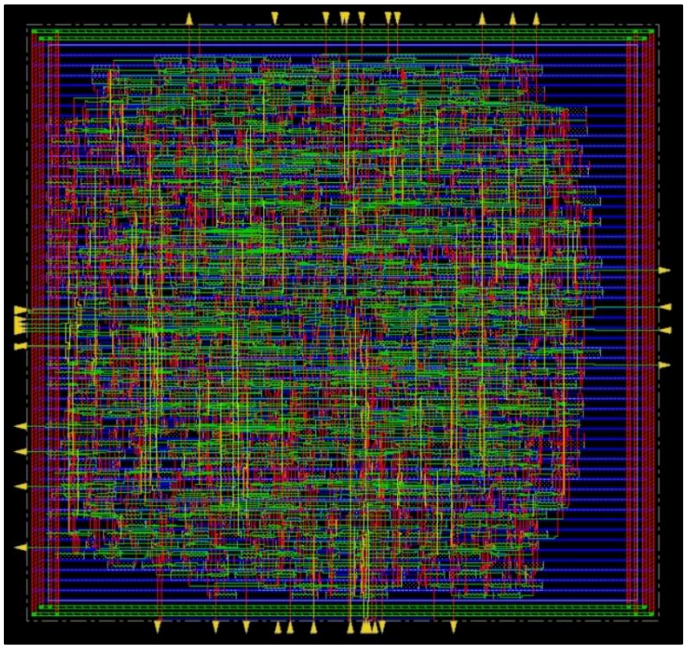


Figure 5.108: Floorplanning Result for Level 2 IR Memory.

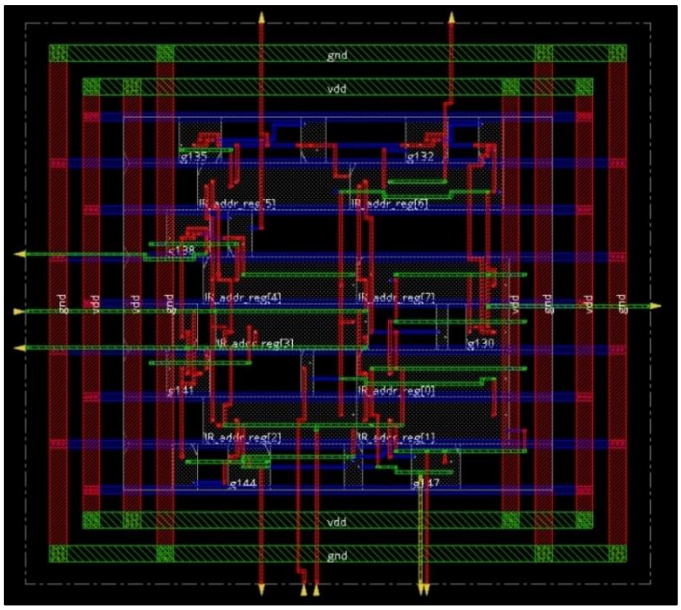


Figure 5.109: Floorplanning Result for Level 2 IR Counter.

As shown in Figures 5.110 and 5.111, the floorplanning result and violation messages for the Level 2 Host Memory Instruction Set reveal challenges due to its large memory structure and 16-bit instruction width. The compact layout and suboptimal I/O port placement—automatically generated by Innovus—lead to antenna violations, indicated by the white crosses in Figure 5.111. Future work should focus on optimizing the core utilization parameters and learning techniques for redistributing I/O ports across multiple boundaries to reduce congestion on a single side.

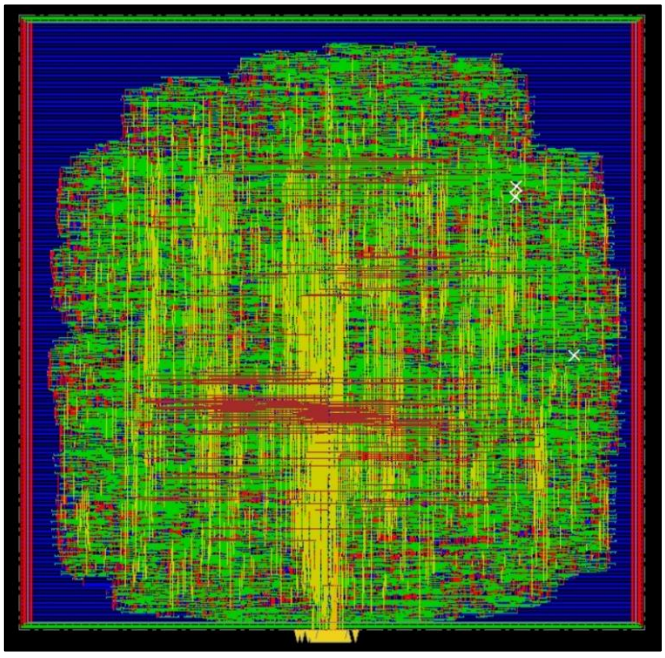


Figure 5.110: Floorplanning Result for Level 2 Host Memory Instruction Set.

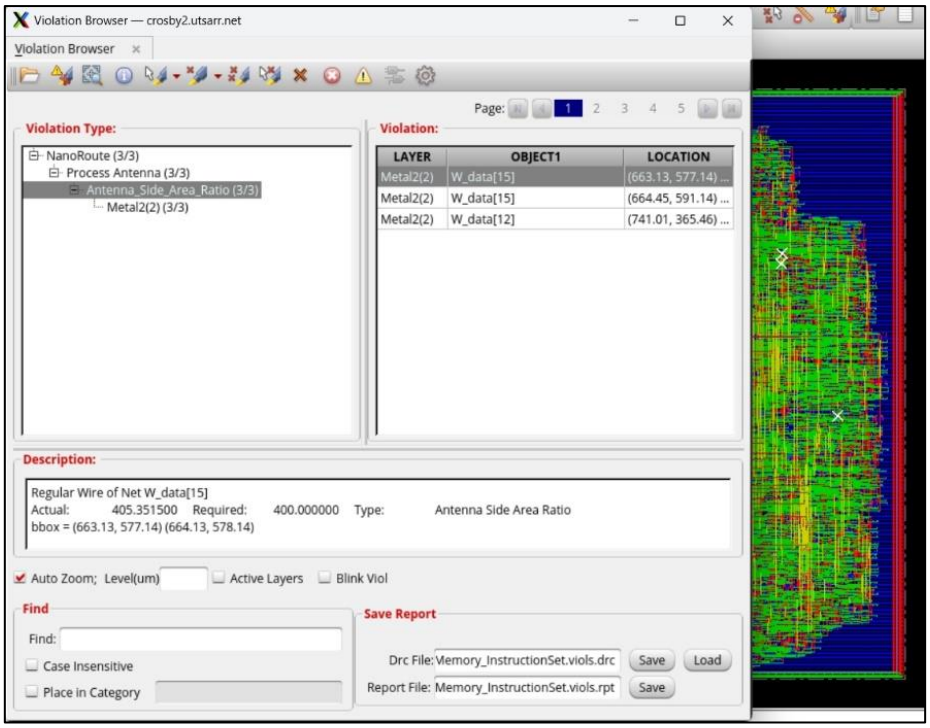


Figure 5.111: Violation Browser for Level 2 Host Memory Instruction Set.

Figures 5.112 and 5.113 show that the same antenna violation occurs in the floorplanning result of the Level 2 Host Memory Weight. A potential reason for this could be the large memory structure and the read operation, which relies heavily on case statements and contributes to routing complexity.

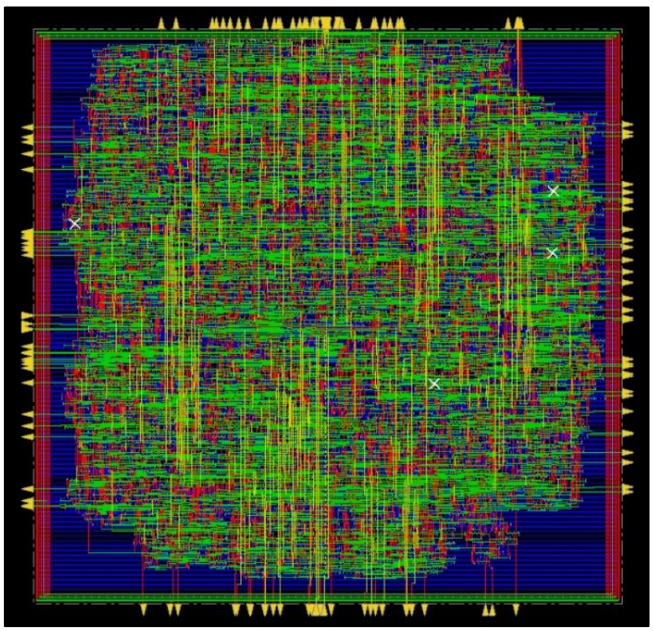


Figure 5.112: Floorplanning Result for Level 2 Host Memory Weight.

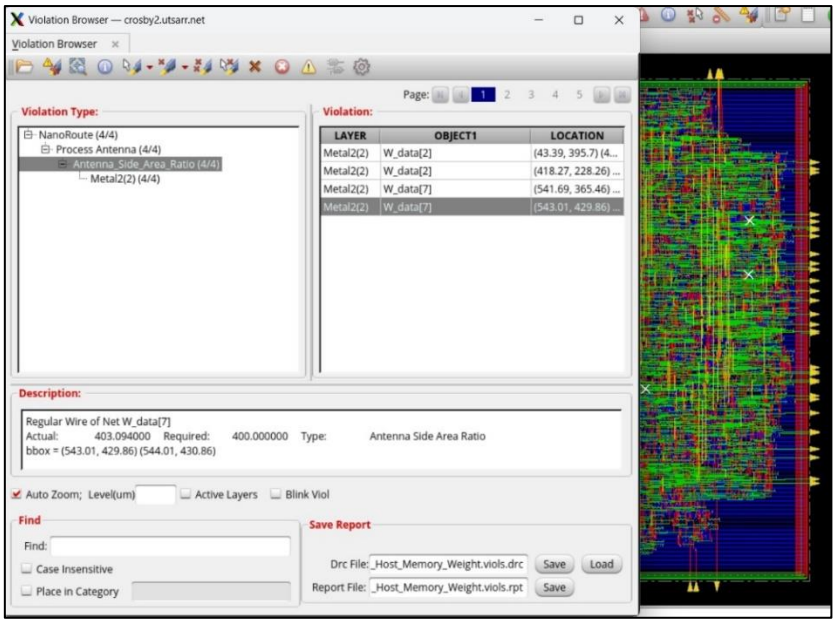


Figure 5.113: Violation Browser for Level 2 Host Memory Weight.

However, as shown in Figure 5.114, the floorplanning result for the Level 2 Host Memory Activation does not exhibit the same violation. This is an interesting outcome, given that the Level 2 Host Memory Activation design includes both read and write operations (to update activation results), making it even more complex than the Level 2 Host Memory Weight. The reason for the absence of violations in this case remains unclear and warrants further study. Since routing is automatically handled by Innovus, its internal mechanisms are not fully accessible or transparent—especially for students or individuals without access to in-depth tool documentation.

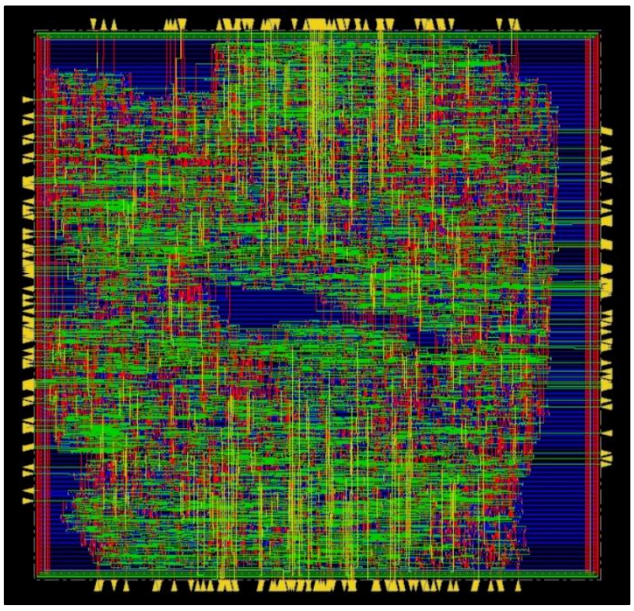


Figure 5.114: Floorplanning Result for Level 2 Host Memory Activation.

Figure 5.115 shows the floorplanning result for the Tiled Systolic Data Setup Unit. Despite having several case statements to handle data reordering for different matrix sizes, the placement result appears relatively balanced and evenly distributed across the allocated area. However, one antenna violation still occurs in the design, as highlighted in Figure 5.116.

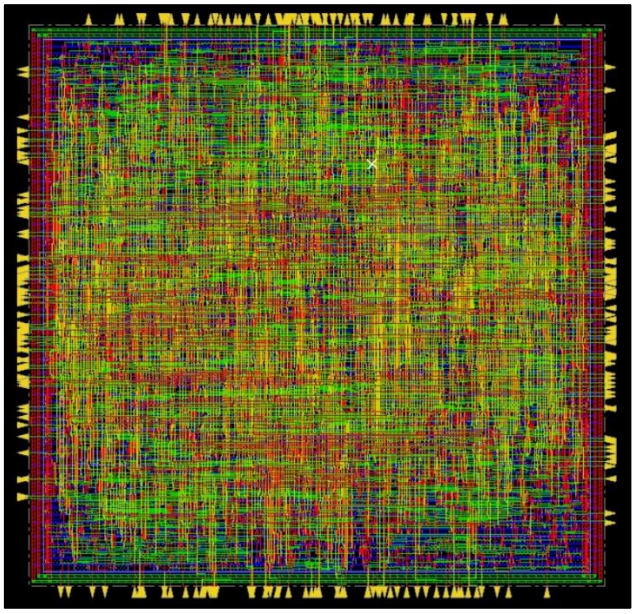


Figure 5.115: Floorplanning Result for Tiled Systolic Data Setup Unit.

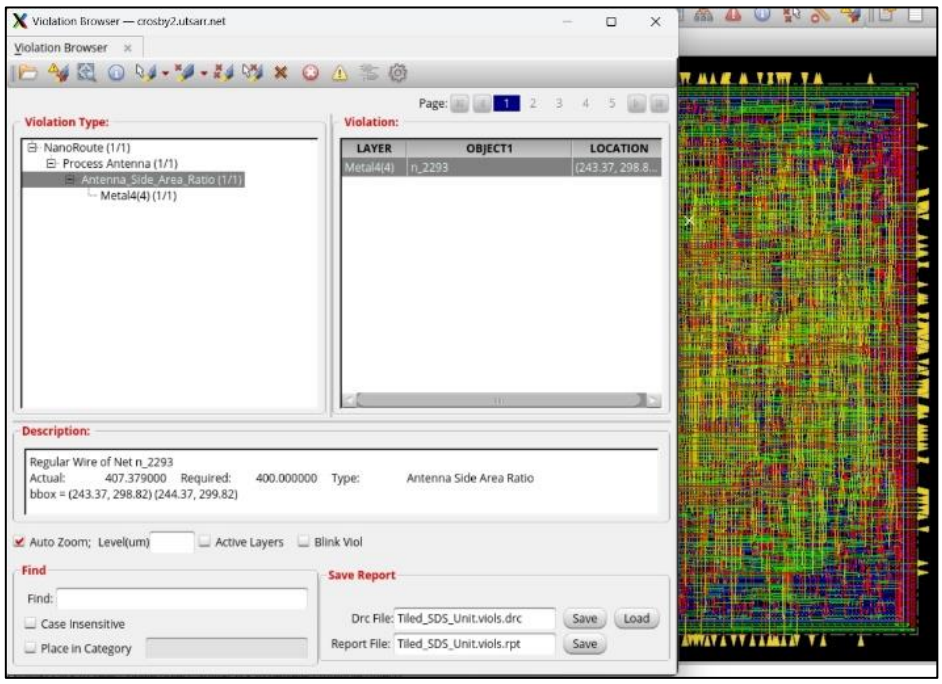


Figure 5.116: Violation Browser for Tiled Systolic Data Setup Unit.

Figure 5.117 shows the floorplanning result for the Level 2 Controller (TMMU Controller). Although the controller handles many state transitions, the floorplanning result appears concise and simple. A likely reason for this is the minimal use of memory structures within the controller—only a few registers are used for counters—resulting in a relatively lightweight layout.

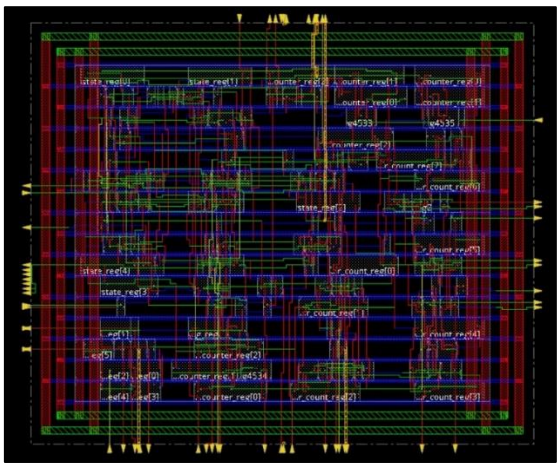


Figure 5.117: Floorplanning Result for TMMU Controller.

Figure 5.118 shows the floorplanning result for the Level 1 Host Memory Weight. Unlike the Level 2 Host Memory Weight, no violations are observed in this floorplan. A potential reason for this is the reduced memory size—from 256 locations in the Level 2 memory to just 64 in the Level 1 memory. This reduction decreases the number of memory structures, or instances, required in the design, thereby lowering the likelihood of violations during placement and routing.

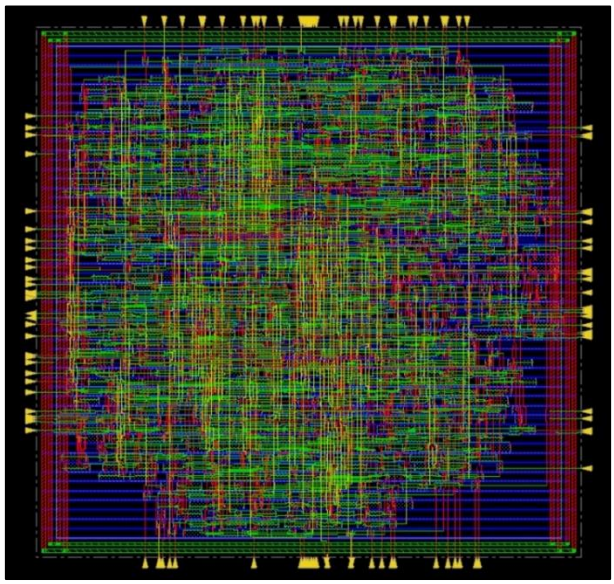


Figure 5.118: Floorplanning Result for Level 1 Host Memory Weight.

Unfortunately, as shown in Figure 5.119, the Level 1 Host Memory Activation exhibits one antenna violation in the floorplanning results. Figure 5.120 presents the violation browser for the Level 1 Host Memory Activation. This violation is likely caused by the functionality of the module, which handles two sources for both input and output: one from and to the Level 2 Host Memory Activation, and another from and to the Unified Buffer. The heavy data forwarding and interconnections managed by the Level 1 Host Memory Activation—which serves as an interface between these components—may have contributed to the routing violation.

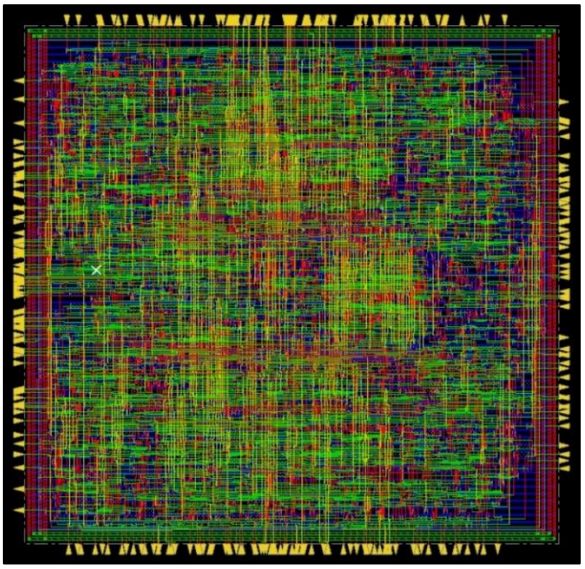


Figure 5.119: Floorplanning Result for Level 1 Host Memory Activation.

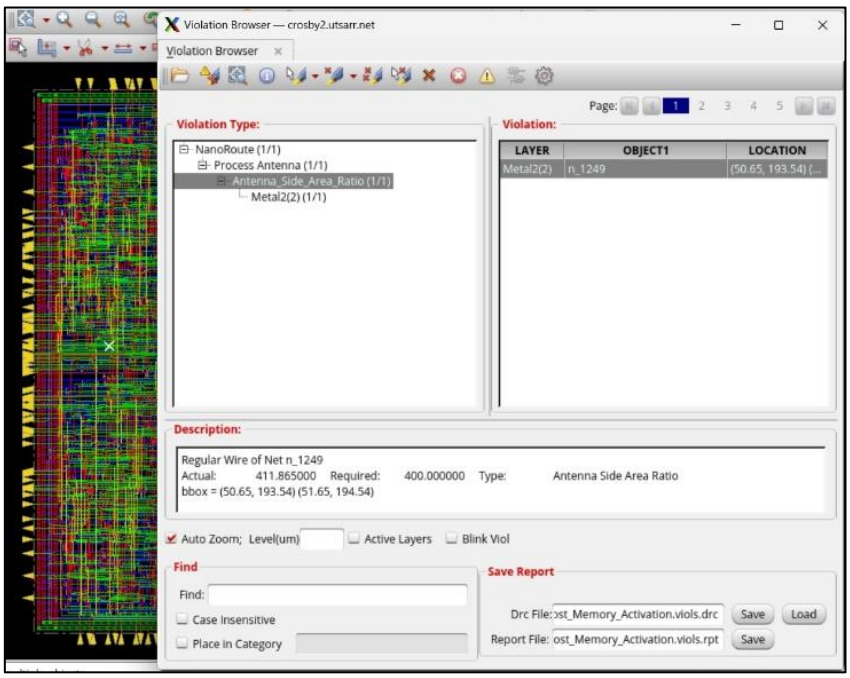


Figure 5.120: Violation Browser for Level 1 Host Memory Activation.

Figure 5.121 and Figure 5.122 show the floorplanning results for the Level 1 IR Memory and the Level 1 IR Counter, respectively. The designs of these components are relatively simple, and as a result, no violations are observed in either of the floorplanning outcomes.

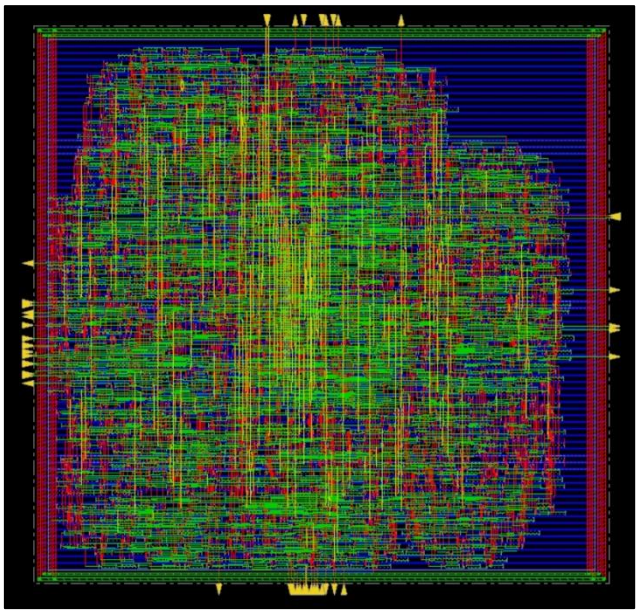


Figure 5.121: Floorplanning Result for Level 1 IR Memory.

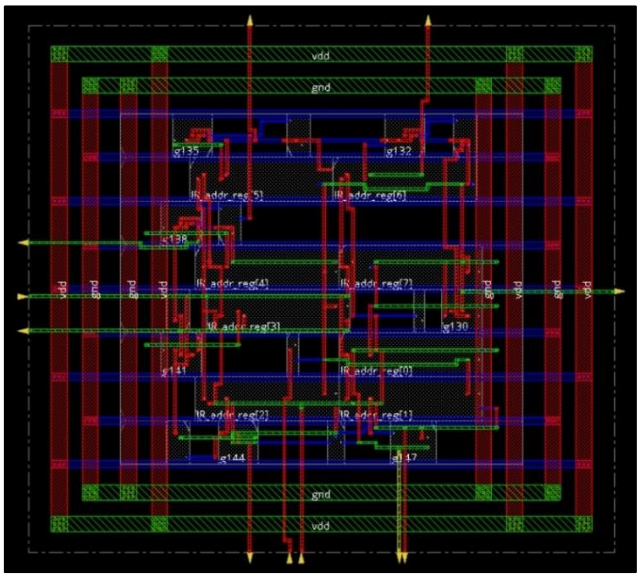


Figure 5.122: Floorplanning Result for Level 1 IR Counter.

Figures 5.123, 5.124, and 5.125 show the floorplanning results for the Weight DDR3, Weight Interface, and Weight FIFO, respectively. No violations are observed in these floorplanning outcomes, likely because the memory sizes at this level (TPU level) are further reduced from 64 to just 16 locations.

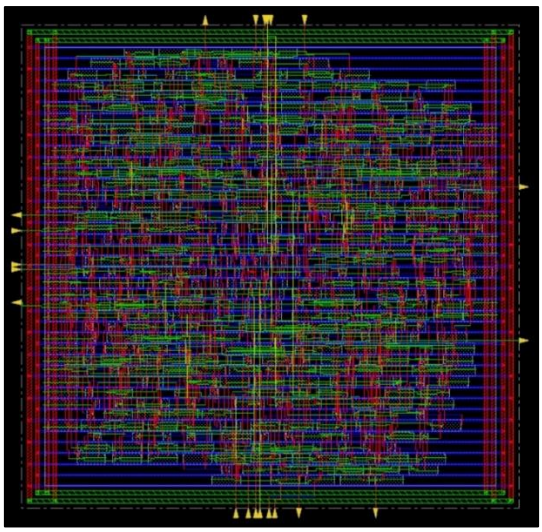


Figure 5.123: Floorplanning Result for Weight DDR3.

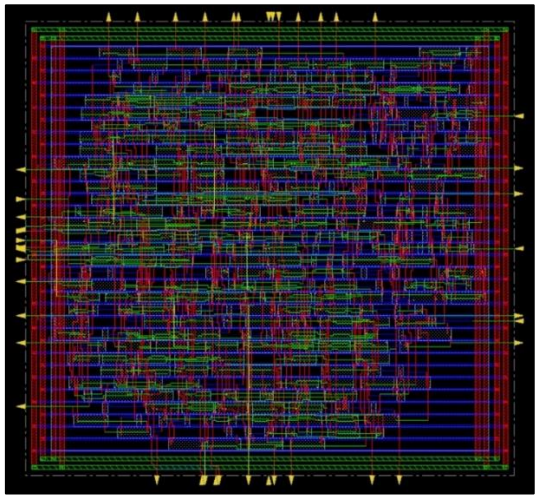


Figure 5.124: Floorplanning Result for Weight Interface.

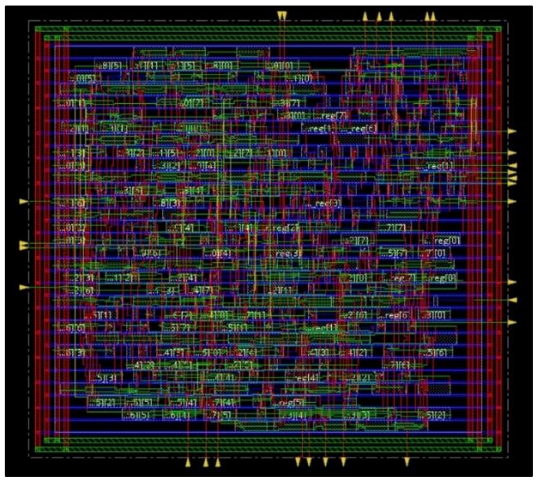


Figure 5.125: Floorplanning Result for Weight FIFO.

As shown in Figure 5.126, the floorplanning result for the Unified Buffer does not exhibit any violations. Despite having two input sources (one from the Level 1 Host Memory Activation and one from the AN_Unit) and two output paths (one back to the Level 1 Host Memory Activation and another to the Systolic Data Setup Unit with 16 output wires), the reduced memory size at the TPU level—from 64 to 16 locations—likely contributes to the lower likelihood of routing violations in this design.

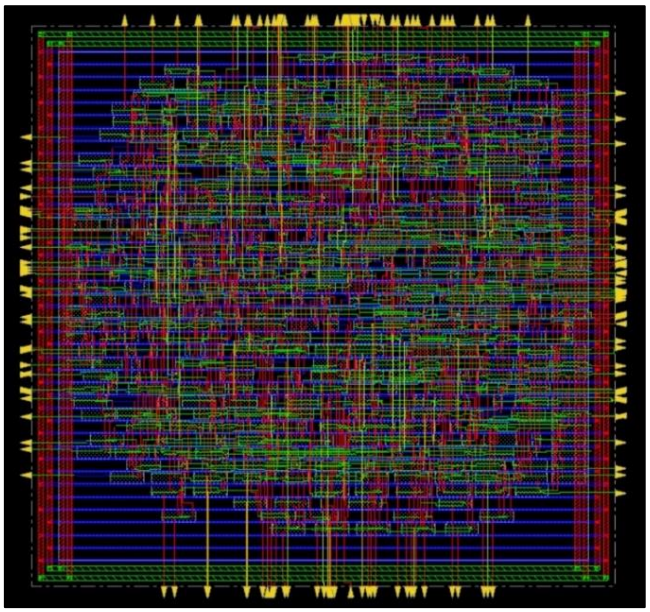


Figure 5.126: Floorplanning Result for Unified Buffer.

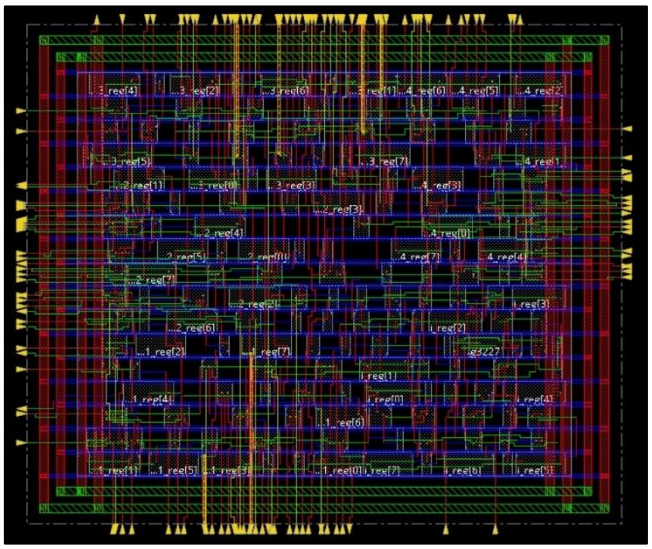


Figure 5.127: Floorplanning Result for Systolic Data Setup Unit.

Figure 5.128 shows the floorplanning result for the Activation FIFO. The result is identical to that of the Weight FIFO, as both components are designed in exactly the same way. This consistency verifies the uniformity and determinism of the outcomes generated by Cadence Innovus for modules with matching structural design.

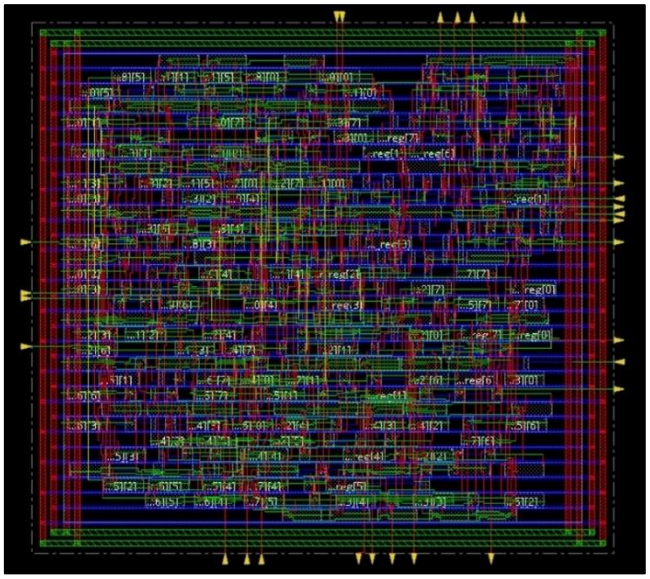


Figure 5.128: Floorplanning Result for Activation FIFO.

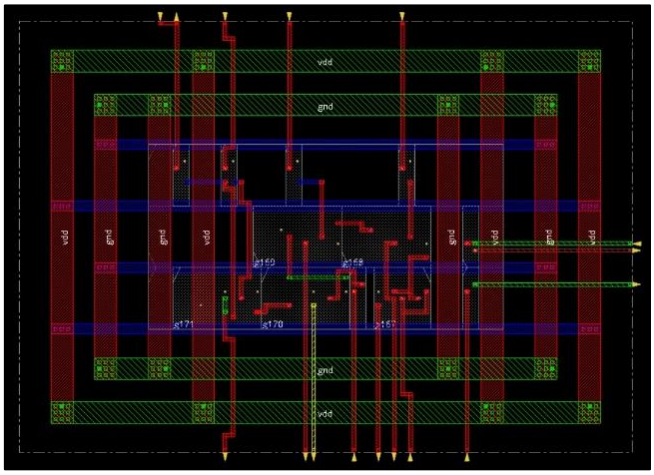


Figure 5.129: Floorplanning Result for Row Detector.

Although the implementation of the Systolic Cell contains multiple-driven net issues, as discussed in Section 5.2, the improper coding style did not prevent the generation of the floorplanning result. Notably, Figure 5.130 shows that the outcome contains zero violations, suggesting that these logic-level issues may not directly propagate to the physical floorplanning process—or may simply be ignored by Innovus. This could potentially lead to more serious issues during actual fabrication. The result highlights the importance of performing comprehensive design checks at both the logical and physical levels before fabrication to ensure functional and manufacturable silicon.

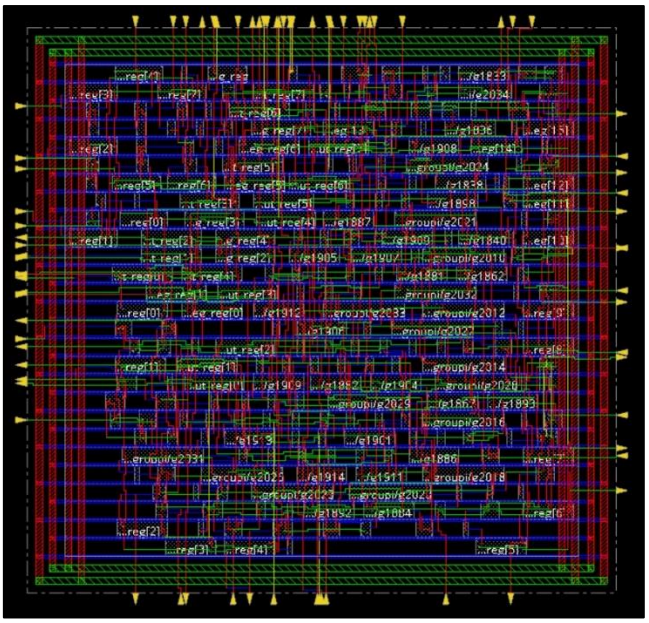


Figure 5.130: Floorplanning Result for Systolic Cell.

However, since the Systolic Cell has been expanded into a 16-cell array to form the Matrix Multiplication Unit (MMU), the floorplanning results—shown in Figure 5.131—exhibit not only antenna violations but also parallel violations. These issues are likely caused by the highly compact and dense nature of the MMU design, which may have made it difficult for Innovus to find suitable routing paths. As a result, overlapping or closely packed routes led to parallel violations. This outcome suggests that further optimization—either by adjusting placement constraints or refining the module layout—is necessary to improve routability and reduce violations in future iterations.

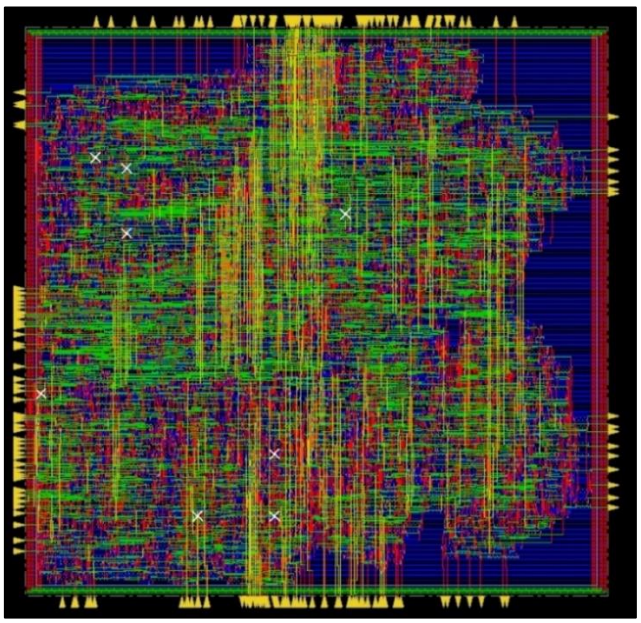


Figure 5.131: Floorplanning Result for Matrix Multiplication Unit.

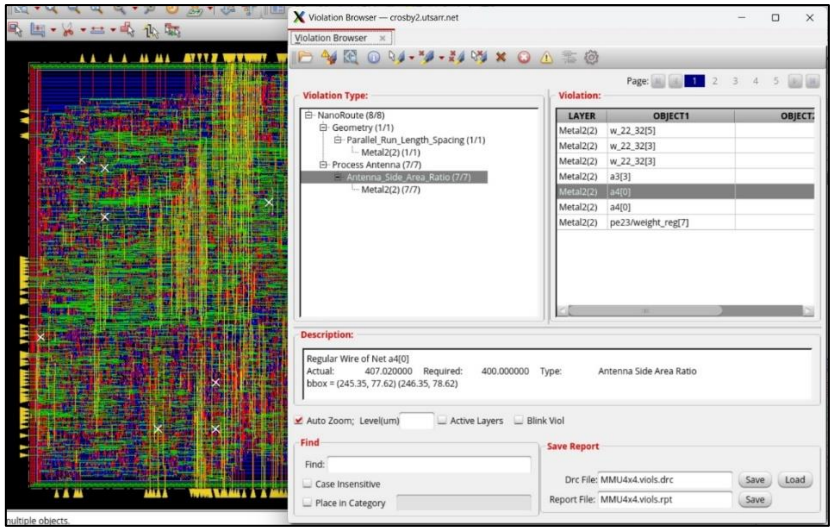


Figure 5.132: Violation Browser for Matrix Multiplication Unit.

Figure 5.133 shows the floorplanning result for the Accumulator. Due to the large number of internal memory structures within this module, the routing appears highly compact across the chip area. As a result, some violations are observed in the outcome, as shown in Figure 5.134.

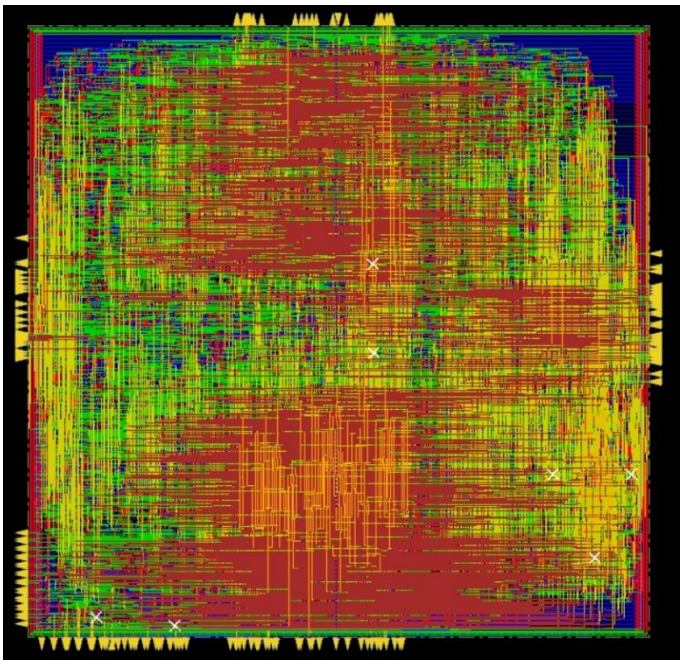


Figure 5.133: Floorplanning Result for Accumulator.

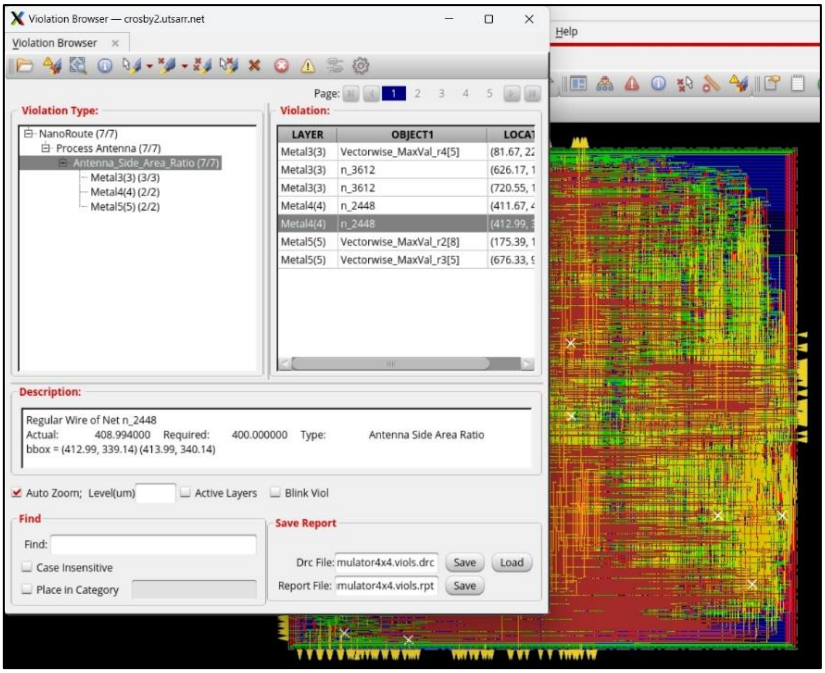


Figure 5.133: Violation Browser for Accumulator.

The next component that requires specific discussion is the Activation Normalization Unit (AN_Unit). Like the Accumulator, the AN_Unit contains numerous memory structures to store both the computed results and the maximum values. In addition to performing activation functions, it handles normalization and rounding operations, which involve a significant number of registers—some of which are 16-bit or even 24-bit wide—to temporarily store intermediate results. The multiplication and division operations used in normalization also demand substantial hardware resources.

As shown in Figure 5.134, the implementation failed due to a critical limitation: the number of instances used in the AN_Unit exceeded the maximum allowed by the current license configuration. MobaXterm terminated abnormally as a result. This highlights a broader challenge faced by students—restricted access to high-end licensing and compute resources—which limits their ability to fully explore and implement more complex hardware designs. Notably, the scale of this design is modest when compared to commercial ASIC projects, yet it still encounters such licensing barriers, underscoring the gap between academic tools and industry-level support.

```
  Message Summary for Library /home/hdj697/Innovus_TPU4x4/slow.lib:
  *****************************************************************
  An unsupported construct was detected in this library. [LBR-40]: 3
  Created nominal operating condition. [LBR-412]: 1
  *****************************************************************

  Setting attribute of root '/': 'library' = /home/hdj697/Innovus_TPU4x4/slow.lib
Info    : Found unusable library cells. [LBR-415]
        : Library: '/home/hdj697/Innovus_TPU4x4/slow.lib', Total cells: 462, Unusable cells: 12.
        List of unusable cells: 'HOLDX1 RF1R1WX2 RF2R1WX2 RFRDX1 RFRDX2 RFRDX4 TIEHI TIELO TTLATX1 TTLATX2 ... and others.'
        : For more information, refer to 'Cells Identified as Unusable' in the RC User Guide. The number of unusable cells that is listed depends on the setti
ng of the 'information_level' root attribute. If set to a value less than 6, the list is limited to 10 unusable cells. If set to a value equal to or higher th
an 6, all unusable cells are listed.
Info    : Elaborating Design. [ELAB-1]
        : Elaborating top-level block 'Activation_Normalization_Unit4x4' from file '/home/hdj697/Innovus_TPU4x4/Activation_Normalization_Unit4x4.v'.
Warning : Using default parameter value for module elaboration. [CDFG-818]
        : Elaborating block 'Activation_Normalization_Unit4x4' with default parameters value.
Info    : Done Elaborating Design. [ELAB-3]
        : Done elaborating 'Activation_Normalization_Unit4x4'.

The maximum allowed instance count has been exceeded.
Unmapped instance count = 274557, max = 200000.
The number of instances in the design has exceeded the maximum allowed for the
current license.  Licenses must be reserved (stacked) at start up, consider
specifying additional licenses using the '-N' option.
For more information see: cdnshelp -> RTL Compiler -> Command Reference ->
General -> rc -> Options and Arguments

Abnormal exit.
```

Figure 5.134: License Limitation Error During AN_Unit Floorplanning Due to Excessive Instance Count.

Therefore, to obtain at least a partial floorplanning result for the AN_Unit, Figure 5.135 shows the layout after removing the code related to basic matrix multiplication from the AN_Unit. Even with this compromise, the floorplanning result for the tiled matrix multiplication still exhibited several antenna and parallel violations, as shown in Figure 5.136. For this result, the core utilization was set to 0.5 and the core margin to 10. Another floorplanning attempt was made with modified parameters—core utilization increased to 0.6 while keeping the core margin at 10. This adjustment led to a slight reduction in violations, as shown in Figure 5.137. However, deeper knowledge of Innovus is needed to fully resolve all violations in the future.

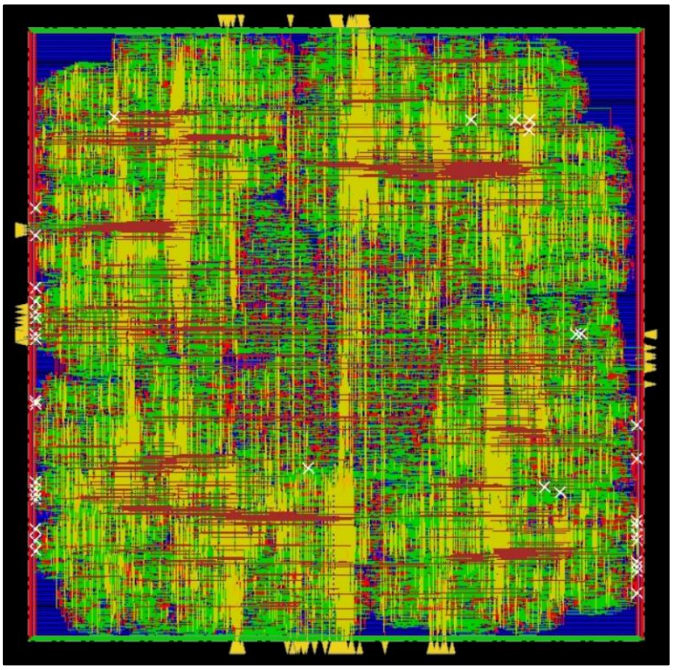


Figure 5.135: Partial Floorplanning Result for AN_Unit with Basic Matrix Multiplication Code Removed.

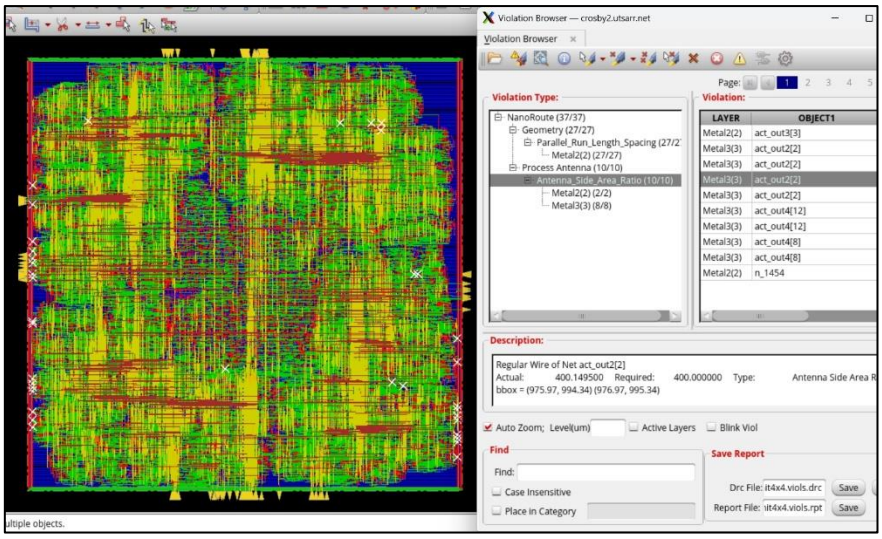


Figure 5.136: Floorplanning Result for AN_Unit (Tiled Mode Only) with Core Utilization = 0.5 and Core Margin = 10, Showing Antenna and Parallel Violations.

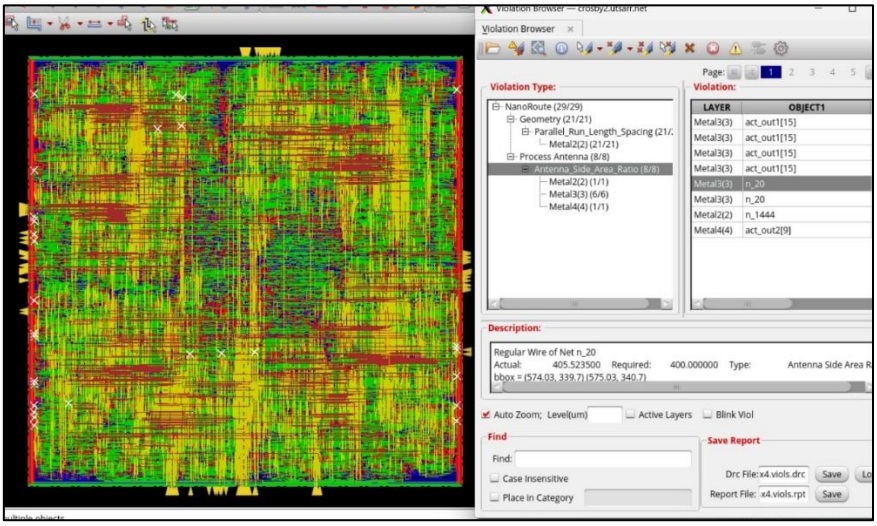


Figure 5.137: Floorplanning Result for AN_Unit with Core Utilization Increased to 0.6, Showing Slight Reduction in Violations.

Figure 5.138 shows the floorplanning result for the Level 1 Controller (TPU Controller). Since the TPU Controller contains more counters and states (35 states) compared to the TMMU Controller (16 states), its placement and routing results are also more complex.

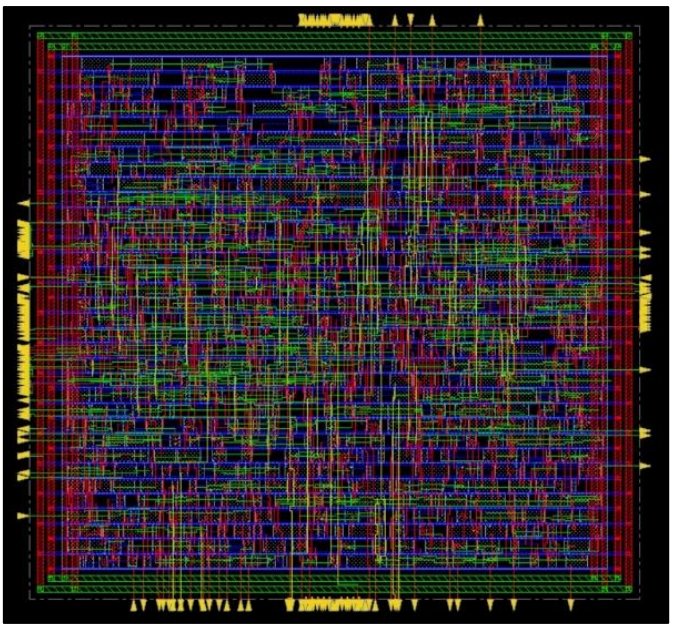


Figure 5.138: Floorplanning Result for TPU Controller.

Because the AN_Unit within the TPU exceeded the instance limit of the current license, the TPU—as a larger module composed of multiple components—also surpasses this limit. Figure 5.139 displays the MobaXterm output showing the instance count and the corresponding abnormal exit error message.

```
The maximum allowed instance count has been exceeded.
Unmapped instance count = 292993, max = 200000.
The number of instances in the design has exceeded the maximum allowed for the
current license.  Licenses must be reserved (stacked) at start up, consider
specifying additional licenses using the '-N' option.
For more information see: cdnshelp -> RTL Compiler -> Command Reference ->
General -> rc -> Options and Arguments

Abnormal exit.
```

Figure 5.139: License Limitation Error During TPU Floorplanning Due to Excessive Instance Count.

Similarly, to obtain at least a partial floorplanning result of the TPU, components such as the Level 1 Host Memories (Weight and Activation) were removed, and the AN_Unit implementation was also simplified. Figure 5.140 shows the resulting partial floorplan of the TPU. By using the schematic tool in Innovus, the approximate locations of the remaining components within the floorplan were identified and illustrated.

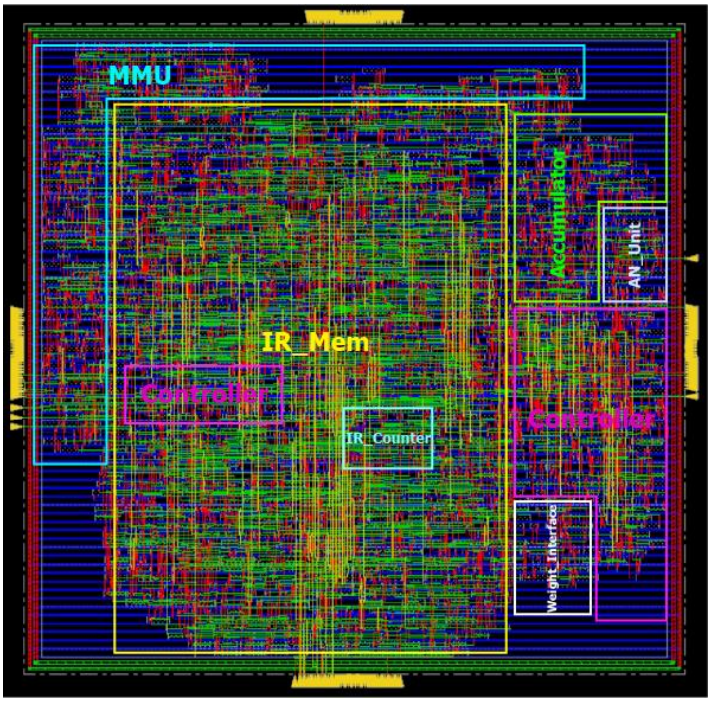


Figure 5.140: Partial Floorplanning Result for TPU.

Obviously, including the TPU in the implementation causes the TMMU to exceed the instance limitation imposed by the current license. Figure 5.141 displays the MobaXterm output showing the instance count and the corresponding abnormal exit error message. However, the memory structures cannot simply be removed, as they play essential roles in realizing the full functionality of tiled matrix multiplication. As a result, the floorplanning for the complete TMMU remains infeasible unless the instance limit issue is resolved—either by upgrading the license or exploring alternative open-source floorplanning tools that support a greater number of instances.

```
The maximum allowed instance count has been exceeded.
Unmapped instance count = 300937, max = 200000.
The number of instances in the design has exceeded the maximum allowed for the
current license.  Licenses must be reserved (stacked) at start up, consider
specifying additional licenses using the '-N' option.
For more information see: cdnshelp -> RTL Compiler -> Command Reference ->
General -> rc -> Options and Arguments

Abnormal exit.
```

Figure 5.141: License Limitation Error During TMMU Floorplanning Due to Excessive Instance Count.

5.4 Future Improvements

As all the results and performance metrics have been presented in the sections above, this section focuses on the future work and improvements for the TMMU project.

It begins by identifying and analyzing the current issues present in the design, along with potential solutions to address these challenges. Once those issues are resolved, the second part of this section explores possible optimization strategies and functional extensions that could further enhance the TMMU architecture.

This chapter helps complete the project by outlining a forward-looking roadmap—guiding future developers and researchers in improving the system both structurally and functionally.

5.4.1 Current Issues and Challenges in TMMU Design

The current TMMU design, while functional in simulation and offering valuable contributions toward implementing a TPU and extending it into the TMMU architecture, still reveals several critical issues and limitations. These must be addressed to achieve a more complete, optimized, and fabrication-ready system.

1. Multiple-Driven Nets Issue:

Some components—most notably the Systolic Cell and TPU Controller—suffer from multiple-driven net conflicts, which prevent proper synthesis and lead to inaccurate power analysis. These issues stem largely from the challenge of writing such a large-scale hardware implementation independently, without guidance or prior experience. Future work must focus on restructuring the logic and adhering to proper HDL design practices to resolve these conflicts.

One potential solution is to redistribute the design workload. For example, instead of using Row Detectors to drive the weight_passtime signals, each Systolic Cell could independently compute its own position and behavior based on global inputs. While this may increase design complexity and reduce modular clarity, it offers a way to eliminate the multi-driver issue.

Similarly, for the TPU Controller, rather than managing all counters internally, a more modular approach would delegate each counter to the corresponding component it serves. The controller would then only need to assert a control signal to trigger the counting process. This restructuring would simplify the state machine logic and help maintain a clean, single-always-block architecture within the TPU Controller.

2. Finite State Machine (FSM) Optimization:

Several controllers—particularly the TPU Controller—rely on overly complex state diagrams, which can hinder both readability and hardware efficiency. Future work

should focus on reducing redundant states and consolidating logic, which would lead to cleaner control flow and better performance.

In the current design, state transitions are determined by internal counters, which track cycles to decide when to exit a state. However, this differs from the approach used in traditional RISC processor controllers, where external input signals from components (e.g., "done" or "ready" flags) drive the transitions. Adopting this method would not only simplify the controller logic but also clarify the interaction between modules, making the system more modular and easier to debug and extend.

3. Innovus-Related Constraints:

The use of Cadence Innovus exposed two major limitations: a lack of familiarity with the tool's workflow and a license-imposed instance limit. These issues prevented full-chip floorplanning and highlighted the need for deeper, tool-specific expertise or access to a more robust license configuration.

Specifically, the current license restricts netlist generation to designs with fewer than 200,000 instances. Given that TMMU is a comprehensive architecture aimed at handling complex computations, it naturally involves a large number of hardware resources—making it easy to exceed this threshold. When this limit is breached, the tool exits abnormally, preventing netlist generation and ultimately blocking access to any floorplanning results. Without floorplanning, it's impossible to identify or fix routing violations, optimize core utilization, or assess real fabrication viability.

Even in cases where floorplanning is successful, the lack of systematic, beginner-friendly documentation or structured tutorials for Innovus makes learning the tool challenging. Many available resources assume prior knowledge, skip essential setup steps, or begin with overly complex examples. To unlock the full optimization potential and resolve routing issues, more comprehensive learning and hands-on experience with Innovus workflows will be essential for future development.

4. Use of Real-World Datasets:

The current implementation relies on artificial, sequential inputs (e.g., 0–255) for testing and validation purposes. While effective for functional verification and dataflow analysis, this does not reflect realistic usage scenarios. To better evaluate the TMMU's performance and power consumption, future work should integrate real-world datasets, such as MNIST or other INT8-based inference datasets.

Notably, during presentations or discussions about this project, a recurring question emerged: “Have you tested it with real data?” The frequency of this inquiry reinforces the importance of incorporating real activation and weight inputs to validate the architecture’s effectiveness in practical machine learning applications. Doing so would not only enhance credibility but also provide a clearer picture of how the TMMU performs under authentic workloads.

5. Benchmarking Against Similar Architectures:

To thoroughly evaluate the efficiency of the TMMU, it is essential to benchmark it against similar AI accelerators or systolic-array-based matrix multipliers, whether from academic research or commercial products. As mentioned in the introduction, many tech companies have developed their own custom hardware architectures for tiled matrix multiplication.

A future comparison between the TMMU implemented in this project and these existing designs would provide valuable insight into its relative performance, scalability, and efficiency. Such benchmarking would not only help quantify how close the current design is to industry standards, but also identify specific areas for improvement in terms of hardware utilization, throughput, and power consumption.

6. Toolchain Expansion - Synopsys Design Compiler:

To support ASIC-oriented synthesis and optimization, incorporating Synopsys Design Compiler as a complementary tool would enhance the design and evaluation process by aligning it with industry-standard methodologies. While this project utilized Xilinx Vivado to estimate power consumption for each component, many academic and commercial studies report power metrics using Synopsys Design Compiler, making it a more suitable choice for benchmarking and comparison.

In contrast to Vivado, which primarily provides on-chip power analysis, Synopsys Design Compiler offers more comprehensive performance metrics, including energy per area, dynamic/static power, and timing closure under ASIC constraints. Integrating this tool in future work would make the TMMU project more robust and comparable, while also extending its relevance to a broader range of ASIC design workflows.

7. Dual CISC ISA Integration:

The use of two separate CISC instruction sets—one for the TMMU Controller and one for the TPU Controller—adds complexity to the overall architecture. Future revisions

could explore the possibility of merging or simplifying the ISA structure to improve maintainability, decoding efficiency, and system-level control.

Although merging the two instruction sets would reduce redundancy and streamline ISA design, it introduces new challenges. For instance, additional logic would be required to decode and route instructions to the correct controller, increasing complexity in the instruction-handling pipeline. An alternative approach might involve designing a centralized controller that manages the entire TMMU operation. However, such a solution could concentrate too much workload on a single controller, potentially impacting modularity, scalability, and parallelism—which are critical for efficient hardware design.

8. Functional Reorganization of Accumulator and AN_Unit:

Both the Accumulator and the Activation Normalization Unit (AN_Unit) currently operate using separate, memory-intensive workflows. Refactoring or reorganizing their responsibilities could help streamline dataflow, reduce redundant processing, and simplify activation-related operations.

As shown in the supplementary code, along with the power estimation and floorplanning results discussed in previous sections, both components suffer from compact layout and heavy memory structures, making them more susceptible to placement and routing violations. Additionally, the current implementation centralizes too much workload in these two units. A more scalable approach would be to distribute specific tasks to dedicated submodules, which can alleviate bottlenecks and improve maintainability and hardware efficiency.

9. Support for Sigmoid and SoftMax:

Currently, only the ReLU activation function is supported. Extending the Activation Normalization Unit (AN_Unit) to include Sigmoid and SoftMax would allow the TMMU to handle a broader range of neural network operations and improve its general applicability.

Both Sigmoid and SoftMax are widely used in modern deep learning models. While there were initial attempts during development to incorporate these functions, significant implementation challenges were encountered—especially with SoftMax. This function requires additional exponential and division operations, as well as an accumulation step over the full output range. These computations would place a heavy

burden on the Accumulator and AN_Unit, two components already constrained by intensive workloads.

To support these functions without overloading the current system, future development should include a carefully planned architectural extension. This might involve introducing dedicated processing submodules or auxiliary units designed specifically for complex activation functions, ensuring functionality is added without compromising system efficiency.

10. Improved Logic for Small-Sized Matrix Multiplication (1×1 to 3×3):

For small matrix sizes, the current design loads unnecessary padding and performs operations inefficiently. Implementing a dedicated logic path optimized for 1×1 to 3×3 Basic Matrix Multiplication (BMM) would significantly improve both execution speed and hardware resource usage.

As shown in the simulation results in Section 5.1.2, the current implementation for computing small matrices (1×1 to 3×3) lacks proper handling in terms of instruction sequencing and data arrangement. While the final results are correct, some unrelated memory locations are overwritten with zero due to excessive padding and lack of range-specific control. Moreover, the use of a full 4×4 MMU for such small matrices leads to inefficient computation.

Future work should focus on redesigning the logic path for this specific range. This may involve writing specialized instructions, optimizing dataflow to avoid redundant storage, and bypassing certain memory structures to streamline the process. Doing so would enhance the completeness and adaptability of the TMMU architecture, especially for lightweight neural network tasks at the edge.

5.4.2 Future Extensions and Opportunities for TMMU

With the core TMMU architecture successfully demonstrated and partially verified through simulation and floorplanning, several promising directions emerge for expanding the scope and impact of this design—once the current issues outlined in the previous section are resolved. These future extensions aim to scale the system for more demanding computational workloads and transition the RTL design into a fabricated hardware prototype suitable for real-world deployment.

1. Multi-TMMU Integration for Larger Matrix Sizes:

The current TMMU is designed for 8×8 matrix multiplication using a 4×4 MMU core.

To scale for larger matrices such as 16×16 or beyond, future work can explore chaining or parallelizing multiple TMMU blocks. This would require the development of a higher-level coordination unit—potentially a global scheduler or controller—that can manage data distribution, synchronization, and result aggregation across multiple TMMUs operating in parallel. This unit must also handle load balancing, ensuring that each TMMU block receives the correct matrix sub-blocks and that the output results are properly merged.

Additionally, extending to larger matrices will introduce more complexity in instruction scheduling, memory interfacing, and timing coordination. Careful design will be needed to maintain pipeline efficiency and avoid bottlenecks between computation and data transfer. Memory bandwidth requirements will also increase, necessitating optimized memory hierarchies or burst-transfer techniques to support larger datasets. Incorporating such a multi-TMMU framework would not only enhance scalability but also bring the architecture closer to real-world AI workloads that rely on high-dimensional matrix operations.

2. ASIC Fabrication and Real-World Validation:

To transition from simulation to physical implementation, future work can focus on fabricating the TMMU chip. Fabrication would allow the architecture to be tested beyond the theoretical and simulated environment, enabling key hardware metrics—such as power consumption, area utilization, latency, and throughput—to be validated directly on silicon. These real-world measurements would provide a much more accurate understanding of how the TMMU performs under different workloads and operating conditions, including thermal behavior and voltage stability.

Moving toward fabrication also represents a critical step in bridging academic prototyping with industry-grade development. It would require the RTL design to undergo full design-for-manufacturing (DFM) preparation, including synthesis, place and route, static timing analysis, and thorough verification for functionality and corner cases. Collaborating with a fabrication foundry or leveraging multi-project wafer (MPW) services such as those offered by TSMC or Google’s open-source shuttle programs could make this step accessible for research purposes.

Successfully taping out and testing the chip would open doors for deployment in edge devices and embedded AI systems where lightweight, power-efficient computation is critical. Moreover, it would establish a robust, real-world benchmark for the TMMU architecture, strengthening its viability for commercial or academic expansion. This

milestone could also support broader research into domain-specific accelerators and lead to iterative improvements based on measured silicon performance.

5.5 Summary

Chapter 5 delivered a comprehensive evaluation of the Tiled Matrix Multiplication Unit (TMMU), moving from simulation-based verification to hardware performance assessment and physical design readiness. This chapter synthesized the behavioral correctness, power consumption, floorplanning feasibility, and architectural scalability of the system while highlighting design challenges and outlining avenues for enhancement.

The chapter began with simulation results generated using Xilinx Vivado, demonstrating correct data propagation, instruction sequencing, and computation across both Basic and Tiled Matrix Multiplication modes. Multiple matrix sizes were tested, with waveforms confirming accurate timing, memory coordination, and result accumulation. These simulations validated the core functionality of the TMMU, particularly the synchronization between controllers, memory units, and the MMU.

Power analysis followed, offering per-component energy insights under typical operational conditions. However, the analysis also revealed that several modules—particularly the Systolic Cell and TPU Controller—suffered from multiple-driven nets, resulting in inaccurate power estimates and failed implementations. These findings emphasized the importance of code quality and synthesis compatibility in preparing designs for physical realization.

To further evaluate fabrication readiness, floorplanning using Cadence Innovus was performed for each major module. The layout results confirmed the physical plausibility of many components, but also exposed common placement and routing violations—particularly in memory-heavy or densely interconnected modules like the Accumulator and AN_Unit. Moreover, the instance limit imposed by the Innovus license prevented full-chip floorplanning for the TPU and TMMU, underscoring the resource challenges in academic environments.

Finally, the chapter concluded with a forward-looking roadmap. A two-part future work section detailed both the issues requiring resolution—such as FSM optimization, toolchain expansion, and real-world benchmarking—and broader opportunities for architectural extension, including support for larger matrices through multi-TMMU integration and eventual chip fabrication. Together, the results and insights in Chapter

5 not only verify the functional viability of the TMMU but also chart a clear path for advancing the system toward industrial relevance and real-world deployment.

CHAPTER 6

Conclusion

In an era where computational performance and energy efficiency are equally critical, the design of specialized hardware accelerators—such as the Tiled Matrix Multiplication Unit (TMMU)—represents a pivotal step toward scalable, application-specific computing. This project set out to explore the full design and implementation flow of a tiled matrix multiplier based on a systolic array architecture, addressing the demands of modern AI workloads through modularity, efficiency, and design clarity.

The TMMU architecture was developed from the ground up, starting with a solid understanding of matrix computation principles and evolving into a fully realized Verilog-based hardware system. Its layered design—separating the responsibilities of the Level 2 (TMMU) and Level 1 (TPU) controllers—demonstrates how hierarchical control and memory coordination can enable efficient data movement and computation. Through careful architectural planning, the TMMU supports both basic and tiled matrix multiplication operations, offering flexibility for various matrix sizes and paving the way for scalable parallel processing. Specifically, the TMMU Controller consists of 16 states, while the TPU Controller comprises 35 states.

Simulations validated the correctness and synchronization of each module, with all computed inputs represented in 8-bit format to support INT8 computation. The results revealed the nuanced interplay between instruction handling, memory flow, and systolic data propagation. Cycle-accurate waveforms confirmed the system’s functional integrity, particularly its ability to manage large matrix operations in tiled mode using a fixed-size MMU. These findings provided valuable insights into the system's temporal behavior and highlighted the effectiveness of data reordering and activation normalization mechanisms. Operating at a 20 ns clock cycle, the TMMU completes a 4×4 matrix multiplication in 3,000 ns (150 clock cycles) and an 8×8 multiplication in approximately 17,000 ns (850 clock cycles).

In parallel, the project’s power analysis and floorplanning efforts underscored the challenges and realities of transitioning from RTL simulation to physical implementation. Power estimation revealed not only the consumption profiles of individual components but also pointed to design issues such as multi-driven nets that

require further optimization. Floorplanning in Cadence Innovus translated the abstract Verilog modules into tangible chip layouts, exposing routing violations and tool limitations that often go unnoticed in simulation. These findings reinforced the importance of synthesis-friendly design practices and emphasized the role of physical design in achieving fabrication-ready systems.

Despite its successful verification at the functional level, the TMMU project also revealed several key areas for improvement. Issues such as controller complexity, inefficient handling of small matrices, limited activation function support, and licensing barriers in design tools will require attention in future iterations. Nevertheless, these limitations offer valuable learning opportunities and serve as stepping stones toward a more robust, optimized architecture.

The TMMU's modular and hierarchical structure opens the door to a variety of future enhancements. Scaling the design to support larger matrix sizes through multi-TMMU integration, extending the instruction set architecture, and transitioning toward ASIC fabrication are all viable paths forward. Integrating real-world datasets and benchmarking the TMMU against commercial accelerators would further validate its performance and applicability in practical AI environments.

Ultimately, this project demonstrates that a thoughtful blend of architectural insight, RTL-level implementation, and system-level evaluation can lead to meaningful contributions in the field of hardware acceleration. By tackling real-world challenges across simulation, power, and layout domains, the TMMU design lays a solid foundation for continued exploration into energy-efficient, high-performance matrix computation. As the demand for specialized hardware accelerators continues to rise, this work serves as both a reference and a springboard for future innovation in domain-specific architectures.

Reference

[1] J. von Neumann, "First draft of a report on the EDVAC," in IEEE Annals of the History of Computing, vol. 15, no. 4, pp. 27-75, 1993, doi: 10.1109/85.238389.

[2] Eigenmann, R., & Lilja, D.J. (1999). Von Neumann Computers.

[3] O'Regan, G. (2018). Von Neumann Architecture. In: The Innovation in Computing Companion. Springer, Cham. https://doi.org/10.1007/978-3-030-02619-6_54

[4] Intel. (2025). What is clock speed? https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html

[5] Ivanov D, Chezhegov A, Kiselev M, Grunin A, Larionov D. Neuromorphic artificial intelligence systems. Front Neurosci. 2022 Sep 14; 16:959626. doi: 10.3389/fnins.2022.959626.

[6] Why systolic architectures. (1982). IEEE Computer, 15(1), 300–309. https://doi.org/10.1109/MC.1982.1653825

[7] M. Vucha and A. Rajawat, "Design and FPGA Implementation of Systolic Array Architecture for Matrix Multiplication," in International Journal of Computer Applications, vol. 26, 2011, doi: 10.5120/3084-4222.

[8] Reagen, B., Adolf, R., Whatmough, P., Wei, GY., Brooks, D. (2017). Foundations of Deep Learning. In: Deep Learning for Computer Architects. Synthesis Lectures on Computer Architecture. Springer, Cham. https://doi.org/10.1007/978-3-031-01756-8_2

[9] Ramírez, C., Castelló, A., Martínez, H. et al. Parallel GEMM-based convolution for deep learning on multicore RISC-V processors. J Supercomput 80, 12623–12643 (2024). https://doi.org/10.1007/s11227-024-05927-y

[10] A. Vasudevan, A. Anderson and D. Gregg, "Parallel Multi Channel convolution using General Matrix Multiplication," in 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Seattle, WA, USA, 2017, pp. 19-24, doi: 10.1109/ASAP.2017.7995254.

[11] Anderson, A., Vasudevan, A., Keane, C., & Gregg, D. (2017). Low-memory GEMM-based convolution algorithms for deep neural networks. ArXiv, abs/1709.03395.

[12] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 2017, pp. 1-12, doi: 10.1145/3079856.3080246.
[13] Google Cloud. (2025). Cloud TPU v3. https://cloud.google.com/tpu/docs/v3.
[14] Google Cloud. (2025). Cloud TPU v4. https://cloud.google.com/tpu/docs/v4.
[15] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Zheng, X. (2016). {TensorFlow}: a system for {Large-Scale} machine learning. In 12th USENIX symposium on operating systems design and implementation (OSDI 16) (pp. 265-283).
[16] L. Bottou et al., "Comparison of classifier methods: a case study in handwritten digit recognition," Proceedings of the 12th IAPR International Conference on Pattern Recognition, Vol. 3 - Conference C: Signal Processing (Cat. No.94CH3440-5), Jerusalem, Israel, 1994, pp. 77-82 vol.2, doi: 10.1109/ICPR.1994.576879.
[17] Ross, J., Jouppi, N., Phelps, A., Young, C., Norrie, T., Thorson, G.,Luu, D., 2015. Neural Network Processor, Patent Application No. 62/164,931.
[18] Ross, J., Phelps, A., 2015. Computing Convolutions Using a Neural Network Processor, Patent Application No. 62/164,902.
[19] Ross, J., 2015. Prefetching Weights for a Neural Network Processor, Patent Application No. 62/164,981.
[20] Ross, J., Thorson, G., 2015. Rotating Data for Neural Network Computations, Patent Application No. 62/164,908.
[21] Thorson, G., Clark, C., Luu, D., 2015. Vector Computation Unit in a Neural Network Processor, Patent Application No. 62/165,022.
[22] Young, C., 2015. Batch Processing in a Neural Network Processor, Patent Application No. 62/165,020

Supplementary

Verilog Code for each Components:

*The Verilog code is provided for a general understanding of each component and to give an idea of the amount of work involved. Please contact the author if more details are needed.

```
module L2_IR_Mem(R_data, W_data, addr, clk, IR_rd, IR_wr);
   parameter address_size = 8;
      parameter word_size = 16;
      parameter memory_size = 32;
      output [word_size-1:0] R_data;
      input [word_size-1:0] W_data;
      input [address_size-1:0] addr;
      input clk, IR_rd, IR_wr;
      reg [word_size-1:0] memory [memory_size-1:0];

      always @(posedge clk) begin
            if (IR_wr) begin
                  memory[addr] <= W_data;
            end
      end

      assign R_data = memory[addr];

endmodule
```

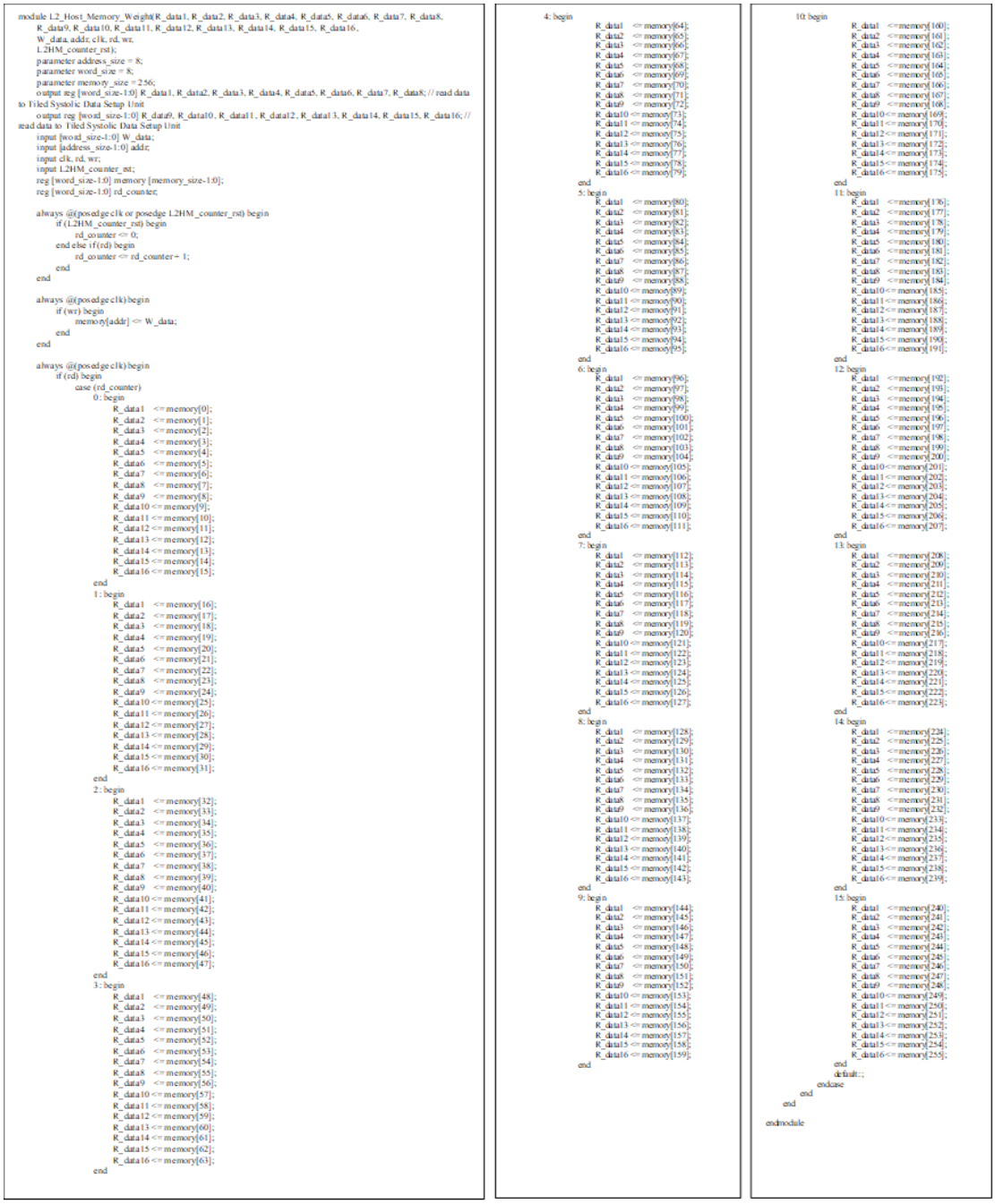
Supplementary Figure 1: Level 2 IR Memory.

```
module L2_IR_counter(IR_addr, IR_inc, IR_clr, clk);
   parameter word_size = 8;
      output reg [word_size-1:0] IR_addr;
      input   IR_inc, IR_clr, clk;

      always@(posedge clk or posedge IR_clr) begin
            if(IR_clr==1) IR_addr <= 8'b0;
                  else if (IR_inc == 1) IR_addr <= IR_addr + 1;
      end

endmodule
```

Supplementary Figure 2: Level 2 IR Counter.

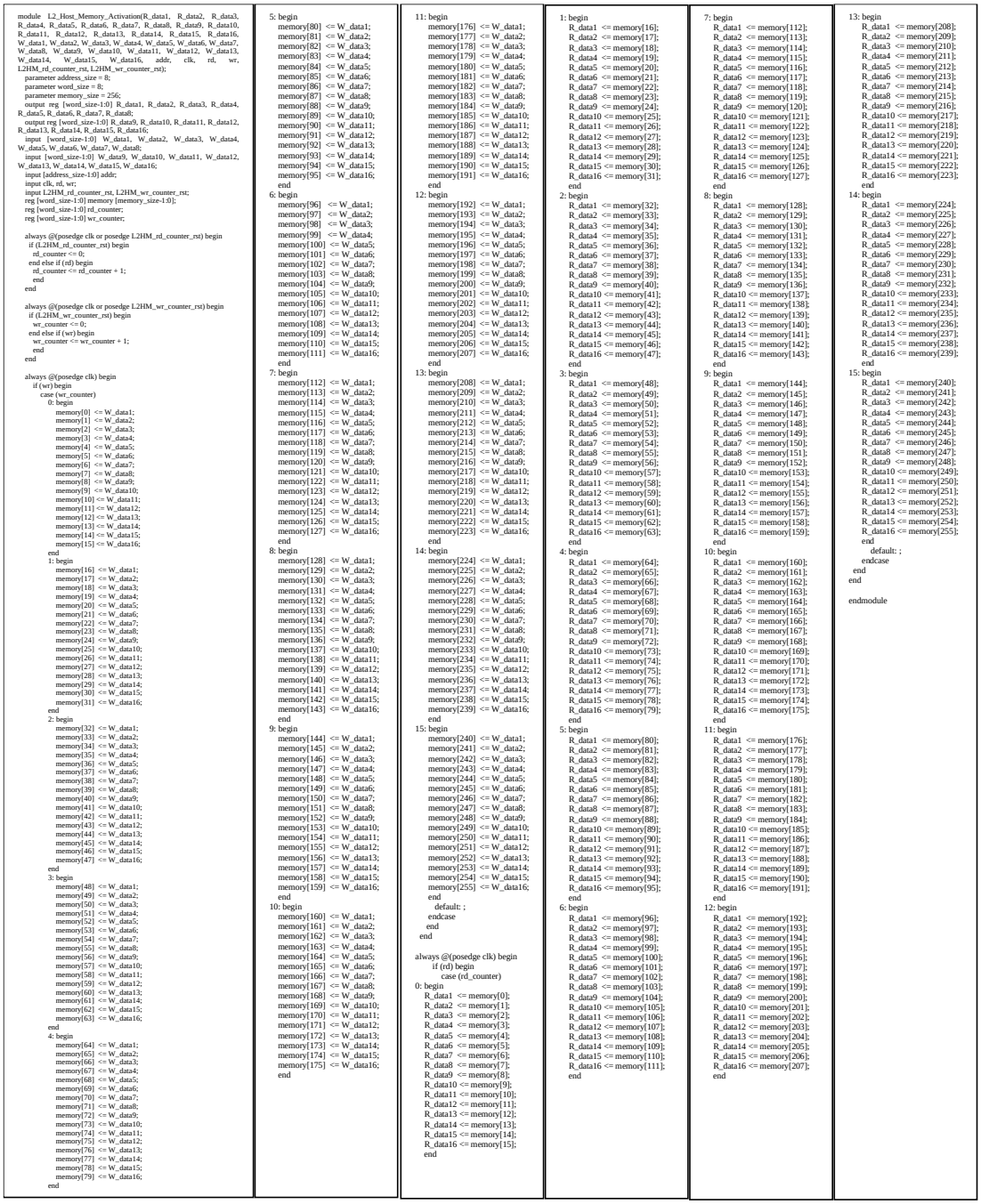
```
module L2_Host_Memory_InstructionSet(R_data, W_data, addr, clk, IR_rd, IR_wr,
finished_sig);
      parameter address_size = 8;
      parameter word_size = 16;
      parameter memory_size = 256;
      output reg [word_size-1:0] R_data;
      output reg finished_sig;
      input [word_size-1:0] W_data;
      input [address_size-1:0] addr;
      input clk, IR_rd, IR_wr;
      reg [word_size-1:0] memory [memory_size-1:0];

      always @(posedge clk) begin
            if (IR_wr) begin
                  memory[addr] <= W_data;
            end else if (IR_rd) begin
                  if (memory[addr] != 16'b1111111111111111) begin
                        R_data <= memory[addr];
                        finished_sig <= 0;
                  end else begin
                        finished_sig <= 1;
                  end
            end
      end

endmodule
```

Supplementary Figure 3: Level 2 Memory for Instruction Set.



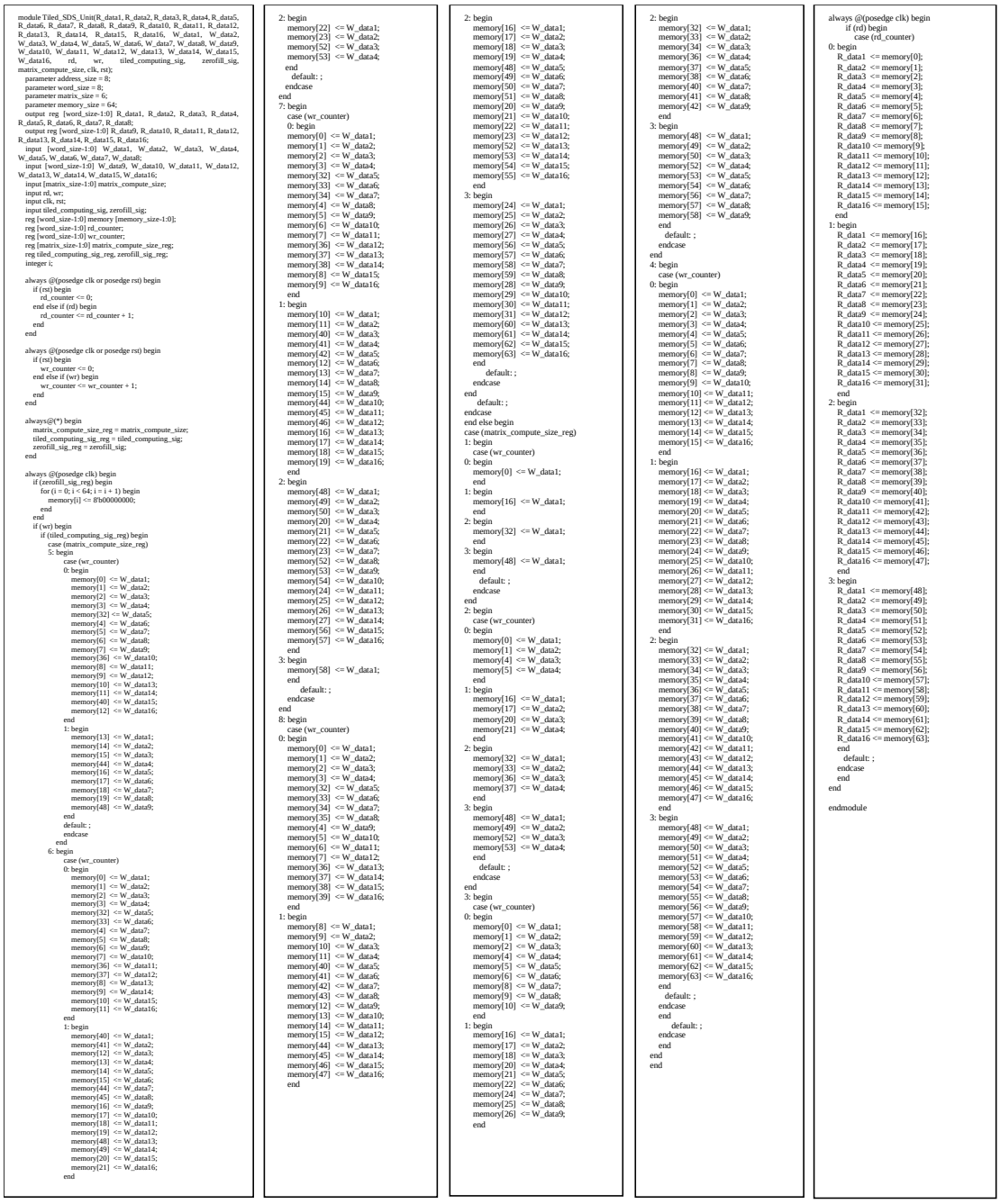
```
module L2_Host_Memory_Weight(R_data1, R_data2, R_data3, R_data4, R_data5, R_data6, R_data7, R_data8,
    R_data9, R_data10, R_data11, R_data12, R_data13, R_data14, R_data15, R_data16,
    W_data, addr, clk, rd, wr,
    L2HM_counter_rst);
    parameter address_size = 8;
    parameter word_size = 8;
    parameter memory_size = 256;
    output reg [word_size-1:0] R_data1, R_data2, R_data3, R_data4, R_data5, R_data6, R_data7, R_data8; // read data
to Tiled Systolic Data Setup Unit
    output reg [word_size-1:0] R_data9, R_data10, R_data11, R_data12, R_data13, R_data14, R_data15, R_data16; //
read data to Tiled Systolic Data Setup Unit
    input [word_size-1:0] W_data;
    input [address_size-1:0] addr;
    input clk, rd, wr;
    input L2HM_counter_rst;
    reg [word_size-1:0] memory [memory_size-1:0];
    reg [word_size-1:0] rd_counter;

    always @(posedge clk or posedge L2HM_counter_rst) begin
        if (L2HM_counter_rst) begin
            rd_counter <= 0;
        end else if (rd) begin
            rd_counter <= rd_counter + 1;
        end
    end

    always @(posedge clk) begin
        if (wr) begin
            memory[addr] <= W_data;
        end
    end

    always @(posedge clk) begin
        if (rd) begin
            case (rd_counter)
                0: begin
                    R_data1  <= memory[0];
                    R_data2  <= memory[1];
                    R_data3  <= memory[2];
                    R_data4  <= memory[3];
                    R_data5  <= memory[4];
                    R_data6  <= memory[5];
                    R_data7  <= memory[6];
                    R_data8  <= memory[7];
                    R_data9  <= memory[8];
                    R_data10 <= memory[9];
                    R_data11 <= memory[10];
                    R_data12 <= memory[11];
                    R_data13 <= memory[12];
                    R_data14 <= memory[13];
                    R_data15 <= memory[14];
                    R_data16 <= memory[15];
                end
                1: begin
                    R_data1  <= memory[16];
                    R_data2  <= memory[17];
                    R_data3  <= memory[18];
                    R_data4  <= memory[19];
                    R_data5  <= memory[20];
                    R_data6  <= memory[21];
                    R_data7  <= memory[22];
                    R_data8  <= memory[23];
                    R_data9  <= memory[24];
                    R_data10 <= memory[25];
                    R_data11 <= memory[26];
                    R_data12 <= memory[27];
                    R_data13 <= memory[28];
                    R_data14 <= memory[29];
                    R_data15 <= memory[30];
                    R_data16 <= memory[31];
                end
                2: begin
                    R_data1  <= memory[32];
                    R_data2  <= memory[33];
                    R_data3  <= memory[34];
                    R_data4  <= memory[35];
                    R_data5  <= memory[36];
                    R_data6  <= memory[37];
                    R_data7  <= memory[38];
                    R_data8  <= memory[39];
                    R_data9  <= memory[40];
                    R_data10 <= memory[41];
                    R_data11 <= memory[42];
                    R_data12 <= memory[43];
                    R_data13 <= memory[44];
                    R_data14 <= memory[45];
                    R_data15 <= memory[46];
                    R_data16 <= memory[47];
                end
                3: begin
                    R_data1  <= memory[48];
                    R_data2  <= memory[49];
                    R_data3  <= memory[50];
                    R_data4  <= memory[51];
                    R_data5  <= memory[52];
                    R_data6  <= memory[53];
                    R_data7  <= memory[54];
                    R_data8  <= memory[55];
                    R_data9  <= memory[56];
                    R_data10 <= memory[57];
                    R_data11 <= memory[58];
                    R_data12 <= memory[59];
                    R_data13 <= memory[60];
                    R_data14 <= memory[61];
                    R_data15 <= memory[62];
                    R_data16 <= memory[63];
                end
                4: begin
                    R_data1  <= memory[64];
                    R_data2  <= memory[65];
                    R_data3  <= memory[66];
                    R_data4  <= memory[67];
                    R_data5  <= memory[68];
                    R_data6  <= memory[69];
                    R_data7  <= memory[70];
                    R_data8  <= memory[71];
                    R_data9  <= memory[72];
                    R_data10 <= memory[73];
                    R_data11 <= memory[74];
                    R_data12 <= memory[75];
                    R_data13 <= memory[76];
                    R_data14 <= memory[77];
                    R_data15 <= memory[78];
                    R_data16 <= memory[79];
                end
                5: begin
                    R_data1  <= memory[80];
                    R_data2  <= memory[81];
                    R_data3  <= memory[82];
                    R_data4  <= memory[83];
                    R_data5  <= memory[84];
                    R_data6  <= memory[85];
                    R_data7  <= memory[86];
                    R_data8  <= memory[87];
                    R_data9  <= memory[88];
                    R_data10 <= memory[89];
                    R_data11 <= memory[90];
                    R_data12 <= memory[91];
                    R_data13 <= memory[92];
                    R_data14 <= memory[93];
                    R_data15 <= memory[94];
                    R_data16 <= memory[95];
                end
                6: begin
                    R_data1  <= memory[96];
                    R_data2  <= memory[97];
                    R_data3  <= memory[98];
                    R_data4  <= memory[99];
                    R_data5  <= memory[100];
                    R_data6  <= memory[101];
                    R_data7  <= memory[102];
                    R_data8  <= memory[103];
                    R_data9  <= memory[104];
                    R_data10 <= memory[105];
                    R_data11 <= memory[106];
                    R_data12 <= memory[107];
                    R_data13 <= memory[108];
                    R_data14 <= memory[109];
                    R_data15 <= memory[110];
                    R_data16 <= memory[111];
                end
                7: begin
                    R_data1  <= memory[112];
                    R_data2  <= memory[113];
                    R_data3  <= memory[114];
                    R_data4  <= memory[115];
                    R_data5  <= memory[116];
                    R_data6  <= memory[117];
                    R_data7  <= memory[118];
                    R_data8  <= memory[119];
                    R_data9  <= memory[120];
                    R_data10 <= memory[121];
                    R_data11 <= memory[122];
                    R_data12 <= memory[123];
                    R_data13 <= memory[124];
                    R_data14 <= memory[125];
                    R_data15 <= memory[126];
                    R_data16 <= memory[127];
                end
                8: begin
                    R_data1  <= memory[128];
                    R_data2  <= memory[129];
                    R_data3  <= memory[130];
                    R_data4  <= memory[131];
                    R_data5  <= memory[132];
                    R_data6  <= memory[133];
                    R_data7  <= memory[134];
                    R_data8  <= memory[135];
                    R_data9  <= memory[136];
                    R_data10 <= memory[137];
                    R_data11 <= memory[138];
                    R_data12 <= memory[139];
                    R_data13 <= memory[140];
                    R_data14 <= memory[141];
                    R_data15 <= memory[142];
                    R_data16 <= memory[143];
                end
                9: begin
                    R_data1  <= memory[144];
                    R_data2  <= memory[145];
                    R_data3  <= memory[146];
                    R_data4  <= memory[147];
                    R_data5  <= memory[148];
                    R_data6  <= memory[149];
                    R_data7  <= memory[150];
                    R_data8  <= memory[151];
                    R_data9  <= memory[152];
                    R_data10 <= memory[153];
                    R_data11 <= memory[154];
                    R_data12 <= memory[155];
                    R_data13 <= memory[156];
                    R_data14 <= memory[157];
                    R_data15 <= memory[158];
                    R_data16 <= memory[159];
                end
                10: begin
                    R_data1  <= memory[160];
                    R_data2  <= memory[161];
                    R_data3  <= memory[162];
                    R_data4  <= memory[163];
                    R_data5  <= memory[164];
                    R_data6  <= memory[165];
                    R_data7  <= memory[166];
                    R_data8  <= memory[167];
                    R_data9  <= memory[168];
                    R_data10 <= memory[169];
                    R_data11 <= memory[170];
                    R_data12 <= memory[171];
                    R_data13 <= memory[172];
                    R_data14 <= memory[173];
                    R_data15 <= memory[174];
                    R_data16 <= memory[175];
                end
                11: begin
                    R_data1  <= memory[176];
                    R_data2  <= memory[177];
                    R_data3  <= memory[178];
                    R_data4  <= memory[179];
                    R_data5  <= memory[180];
                    R_data6  <= memory[181];
                    R_data7  <= memory[182];
                    R_data8  <= memory[183];
                    R_data9  <= memory[184];
                    R_data10 <= memory[185];
                    R_data11 <= memory[186];
                    R_data12 <= memory[187];
                    R_data13 <= memory[188];
                    R_data14 <= memory[189];
                    R_data15 <= memory[190];
                    R_data16 <= memory[191];
                end
                12: begin
                    R_data1  <= memory[192];
                    R_data2  <= memory[193];
                    R_data3  <= memory[194];
                    R_data4  <= memory[195];
                    R_data5  <= memory[196];
                    R_data6  <= memory[197];
                    R_data7  <= memory[198];
                    R_data8  <= memory[199];
                    R_data9  <= memory[200];
                    R_data10 <= memory[201];
                    R_data11 <= memory[202];
                    R_data12 <= memory[203];
                    R_data13 <= memory[204];
                    R_data14 <= memory[205];
                    R_data15 <= memory[206];
                    R_data16 <= memory[207];
                end
                13: begin
                    R_data1  <= memory[208];
                    R_data2  <= memory[209];
                    R_data3  <= memory[210];
                    R_data4  <= memory[211];
                    R_data5  <= memory[212];
                    R_data6  <= memory[213];
                    R_data7  <= memory[214];
                    R_data8  <= memory[215];
                    R_data9  <= memory[216];
                    R_data10 <= memory[217];
                    R_data11 <= memory[218];
                    R_data12 <= memory[219];
                    R_data13 <= memory[220];
                    R_data14 <= memory[221];
                    R_data15 <= memory[222];
                    R_data16 <= memory[223];
                end
                14: begin
                    R_data1  <= memory[224];
                    R_data2  <= memory[225];
                    R_data3  <= memory[226];
                    R_data4  <= memory[227];
                    R_data5  <= memory[228];
                    R_data6  <= memory[229];
                    R_data7  <= memory[230];
                    R_data8  <= memory[231];
                    R_data9  <= memory[232];
                    R_data10 <= memory[233];
                    R_data11 <= memory[234];
                    R_data12 <= memory[235];
                    R_data13 <= memory[236];
                    R_data14 <= memory[237];
                    R_data15 <= memory[238];
                    R_data16 <= memory[239];
                end
                15: begin
                    R_data1  <= memory[240];
                    R_data2  <= memory[241];
                    R_data3  <= memory[242];
                    R_data4  <= memory[243];
                    R_data5  <= memory[244];
                    R_data6  <= memory[245];
                    R_data7  <= memory[246];
                    R_data8  <= memory[247];
                    R_data9  <= memory[248];
                    R_data10 <= memory[249];
                    R_data11 <= memory[250];
                    R_data12 <= memory[251];
                    R_data13 <= memory[252];
                    R_data14 <= memory[253];
                    R_data15 <= memory[254];
                    R_data16 <= memory[255];
                end
                default:;
            endcase
        end
    end

endmodule
```

Supplementary Figure 4: Level 2 Memory for Weight.



```
module L2_Host_Memory_Activation(R_data1, R_data2, R_data3, R_data4, R_data5, R_data6, R_data7, R_data8, R_data9, R_data10, R_data11, R_data12, R_data13, R_data14, R_data15, R_data16, W_data1, W_data2, W_data3, W_data4, W_data5, W_data6, W_data7, W_data8, W_data9, W_data10, W_data11, W_data12, W_data13, W_data14, W_data15, W_data16, addr, clk, rd, wr, L2HM_rd_counter_rst, L2HM_wr_counter_rst);
  parameter address_size = 8;
  parameter word_size = 8;
  parameter memory_size = 256;
  output reg [word_size-1:0] R_data1, R_data2, R_data3, R_data4, R_data5, R_data6, R_data7, R_data8;
  output reg [word_size-1:0] R_data9, R_data10, R_data11, R_data12, R_data13, R_data14, R_data15, R_data16;
  input [word_size-1:0] W_data1, W_data2, W_data3, W_data4, W_data5, W_data6, W_data7, W_data8;
  input [word_size-1:0] W_data9, W_data10, W_data11, W_data12, W_data13, W_data14, W_data15, W_data16;
  input [address_size-1:0] addr;
  input clk, rd, wr;
  input L2HM_rd_counter_rst, L2HM_wr_counter_rst;
  reg [word_size-1:0] memory [memory_size-1:0];
  reg [word_size-1:0] rd_counter;
  reg [word_size-1:0] wr_counter;

  always @(posedge clk or posedge L2HM_rd_counter_rst) begin
    if (L2HM_rd_counter_rst) begin
      rd_counter <= 0;
    end else if (rd) begin
      rd_counter <= rd_counter + 1;
    end
  end

  always @(posedge clk or posedge L2HM_wr_counter_rst) begin
    if (L2HM_wr_counter_rst) begin
      wr_counter <= 0;
    end else if (wr) begin
      wr_counter <= wr_counter + 1;
    end
  end

  always @(posedge clk) begin
    if (wr) begin
      case (wr_counter)
        0: begin
          memory[0] <= W_data1;
          memory[1] <= W_data2;
          memory[2] <= W_data3;
          memory[3] <= W_data4;
          memory[4] <= W_data5;
          memory[5] <= W_data6;
          memory[6] <= W_data7;
          memory[7] <= W_data8;
          memory[8] <= W_data9;
          memory[9] <= W_data10;
          memory[10] <= W_data11;
          memory[11] <= W_data12;
          memory[12] <= W_data13;
          memory[13] <= W_data14;
          memory[14] <= W_data15;
          memory[15] <= W_data16;
        end
        1: begin
          memory[16] <= W_data1;
          memory[17] <= W_data2;
          memory[18] <= W_data3;
          memory[19] <= W_data4;
          memory[20] <= W_data5;
          memory[21] <= W_data6;
          memory[22] <= W_data7;
          memory[23] <= W_data8;
          memory[24] <= W_data9;
          memory[25] <= W_data10;
          memory[26] <= W_data11;
          memory[27] <= W_data12;
          memory[28] <= W_data13;
          memory[29] <= W_data14;
          memory[30] <= W_data15;
          memory[31] <= W_data16;
        end
        2: begin
          memory[32] <= W_data1;
          memory[33] <= W_data2;
          memory[34] <= W_data3;
          memory[35] <= W_data4;
          memory[36] <= W_data5;
          memory[37] <= W_data6;
          memory[38] <= W_data7;
          memory[39] <= W_data8;
          memory[40] <= W_data9;
          memory[41] <= W_data10;
          memory[42] <= W_data11;
          memory[43] <= W_data12;
          memory[44] <= W_data13;
          memory[45] <= W_data14;
          memory[46] <= W_data15;
          memory[47] <= W_data16;
        end
        3: begin
          memory[48] <= W_data1;
          memory[49] <= W_data2;
          memory[50] <= W_data3;
          memory[51] <= W_data4;
          memory[52] <= W_data5;
          memory[53] <= W_data6;
          memory[54] <= W_data7;
          memory[55] <= W_data8;
          memory[56] <= W_data9;
          memory[57] <= W_data10;
          memory[58] <= W_data11;
          memory[59] <= W_data12;
          memory[60] <= W_data13;
          memory[61] <= W_data14;
          memory[62] <= W_data15;
          memory[63] <= W_data16;
        end
        4: begin
          memory[64] <= W_data1;
          memory[65] <= W_data2;
          memory[66] <= W_data3;
          memory[67] <= W_data4;
          memory[68] <= W_data5;
          memory[69] <= W_data6;
          memory[70] <= W_data7;
          memory[71] <= W_data8;
          memory[72] <= W_data9;
          memory[73] <= W_data10;
          memory[74] <= W_data11;
          memory[75] <= W_data12;
          memory[76] <= W_data13;
          memory[77] <= W_data14;
          memory[78] <= W_data15;
          memory[79] <= W_data16;
        end
        5: begin
          memory[80] <= W_data1;
          memory[81] <= W_data2;
          memory[82] <= W_data3;
          memory[83] <= W_data4;
          memory[84] <= W_data5;
          memory[85] <= W_data6;
          memory[86] <= W_data7;
          memory[87] <= W_data8;
          memory[88] <= W_data9;
          memory[89] <= W_data10;
          memory[90] <= W_data11;
          memory[91] <= W_data12;
          memory[92] <= W_data13;
          memory[93] <= W_data14;
          memory[94] <= W_data15;
          memory[95] <= W_data16;
        end
        6: begin
          memory[96] <= W_data1;
          memory[97] <= W_data2;
          memory[98] <= W_data3;
          memory[99] <= W_data4;
          memory[100] <= W_data5;
          memory[101] <= W_data6;
          memory[102] <= W_data7;
          memory[103] <= W_data8;
          memory[104] <= W_data9;
          memory[105] <= W_data10;
          memory[106] <= W_data11;
          memory[107] <= W_data12;
          memory[108] <= W_data13;
          memory[109] <= W_data14;
          memory[110] <= W_data15;
          memory[111] <= W_data16;
        end
        7: begin
          memory[112] <= W_data1;
          memory[113] <= W_data2;
          memory[114] <= W_data3;
          memory[115] <= W_data4;
          memory[116] <= W_data5;
          memory[117] <= W_data6;
          memory[118] <= W_data7;
          memory[119] <= W_data8;
          memory[120] <= W_data9;
          memory[121] <= W_data10;
          memory[122] <= W_data11;
          memory[123] <= W_data12;
          memory[124] <= W_data13;
          memory[125] <= W_data14;
          memory[126] <= W_data15;
          memory[127] <= W_data16;
        end
        8: begin
          memory[128] <= W_data1;
          memory[129] <= W_data2;
          memory[130] <= W_data3;
          memory[131] <= W_data4;
          memory[132] <= W_data5;
          memory[133] <= W_data6;
          memory[134] <= W_data7;
          memory[135] <= W_data8;
          memory[136] <= W_data9;
          memory[137] <= W_data10;
          memory[138] <= W_data11;
          memory[139] <= W_data12;
          memory[140] <= W_data13;
          memory[141] <= W_data14;
          memory[142] <= W_data15;
          memory[143] <= W_data16;
        end
        9: begin
          memory[144] <= W_data1;
          memory[145] <= W_data2;
          memory[146] <= W_data3;
          memory[147] <= W_data4;
          memory[148] <= W_data5;
          memory[149] <= W_data6;
          memory[150] <= W_data7;
          memory[151] <= W_data8;
          memory[152] <= W_data9;
          memory[153] <= W_data10;
          memory[154] <= W_data11;
          memory[155] <= W_data12;
          memory[156] <= W_data13;
          memory[157] <= W_data14;
          memory[158] <= W_data15;
          memory[159] <= W_data16;
        end
        10: begin
          memory[160] <= W_data1;
          memory[161] <= W_data2;
          memory[162] <= W_data3;
          memory[163] <= W_data4;
          memory[164] <= W_data5;
          memory[165] <= W_data6;
          memory[166] <= W_data7;
          memory[167] <= W_data8;
          memory[168] <= W_data9;
          memory[169] <= W_data10;
          memory[170] <= W_data11;
          memory[171] <= W_data12;
          memory[172] <= W_data13;
          memory[173] <= W_data14;
          memory[174] <= W_data15;
          memory[175] <= W_data16;
        end
        11: begin
          memory[176] <= W_data1;
          memory[177] <= W_data2;
          memory[178] <= W_data3;
          memory[179] <= W_data4;
          memory[180] <= W_data5;
          memory[181] <= W_data6;
          memory[182] <= W_data7;
          memory[183] <= W_data8;
          memory[184] <= W_data9;
          memory[185] <= W_data10;
          memory[186] <= W_data11;
          memory[187] <= W_data12;
          memory[188] <= W_data13;
          memory[189] <= W_data14;
          memory[190] <= W_data15;
          memory[191] <= W_data16;
        end
        12: begin
          memory[192] <= W_data1;
          memory[193] <= W_data2;
          memory[194] <= W_data3;
          memory[195] <= W_data4;
          memory[196] <= W_data5;
          memory[197] <= W_data6;
          memory[198] <= W_data7;
          memory[199] <= W_data8;
          memory[200] <= W_data9;
          memory[201] <= W_data10;
          memory[202] <= W_data11;
          memory[203] <= W_data12;
          memory[204] <= W_data13;
          memory[205] <= W_data14;
          memory[206] <= W_data15;
          memory[207] <= W_data16;
        end
        13: begin
          memory[208] <= W_data1;
          memory[209] <= W_data2;
          memory[210] <= W_data3;
          memory[211] <= W_data4;
          memory[212] <= W_data5;
          memory[213] <= W_data6;
          memory[214] <= W_data7;
          memory[215] <= W_data8;
          memory[216] <= W_data9;
          memory[217] <= W_data10;
          memory[218] <= W_data11;
          memory[219] <= W_data12;
          memory[220] <= W_data13;
          memory[221] <= W_data14;
          memory[222] <= W_data15;
          memory[223] <= W_data16;
        end
        14: begin
          memory[224] <= W_data1;
          memory[225] <= W_data2;
          memory[226] <= W_data3;
          memory[227] <= W_data4;
          memory[228] <= W_data5;
          memory[229] <= W_data6;
          memory[230] <= W_data7;
          memory[231] <= W_data8;
          memory[232] <= W_data9;
          memory[233] <= W_data10;
          memory[234] <= W_data11;
          memory[235] <= W_data12;
          memory[236] <= W_data13;
          memory[237] <= W_data14;
          memory[238] <= W_data15;
          memory[239] <= W_data16;
        end
        15: begin
          memory[240] <= W_data1;
          memory[241] <= W_data2;
          memory[242] <= W_data3;
          memory[243] <= W_data4;
          memory[244] <= W_data5;
          memory[245] <= W_data6;
          memory[246] <= W_data7;
          memory[247] <= W_data8;
          memory[248] <= W_data9;
          memory[249] <= W_data10;
          memory[250] <= W_data11;
          memory[251] <= W_data12;
          memory[252] <= W_data13;
          memory[253] <= W_data14;
          memory[254] <= W_data15;
          memory[255] <= W_data16;
        end
        default: ;
      endcase
    end
  end

  always @(posedge clk) begin
    if (rd) begin
      case (rd_counter)
        0: begin
          R_data1 <= memory[0];
          R_data2 <= memory[1];
          R_data3 <= memory[2];
          R_data4 <= memory[3];
          R_data5 <= memory[4];
          R_data6 <= memory[5];
          R_data7 <= memory[6];
          R_data8 <= memory[7];
          R_data9 <= memory[8];
          R_data10 <= memory[9];
          R_data11 <= memory[10];
          R_data12 <= memory[11];
          R_data13 <= memory[12];
          R_data14 <= memory[13];
          R_data15 <= memory[14];
          R_data16 <= memory[15];
        end
        1: begin
          R_data1 <= memory[16];
          R_data2 <= memory[17];
          R_data3 <= memory[18];
          R_data4 <= memory[19];
          R_data5 <= memory[20];
          R_data6 <= memory[21];
          R_data7 <= memory[22];
          R_data8 <= memory[23];
          R_data9 <= memory[24];
          R_data10 <= memory[25];
          R_data11 <= memory[26];
          R_data12 <= memory[27];
          R_data13 <= memory[28];
          R_data14 <= memory[29];
          R_data15 <= memory[30];
          R_data16 <= memory[31];
        end
        2: begin
          R_data1 <= memory[32];
          R_data2 <= memory[33];
          R_data3 <= memory[34];
          R_data4 <= memory[35];
          R_data5 <= memory[36];
          R_data6 <= memory[37];
          R_data7 <= memory[38];
          R_data8 <= memory[39];
          R_data9 <= memory[40];
          R_data10 <= memory[41];
          R_data11 <= memory[42];
          R_data12 <= memory[43];
          R_data13 <= memory[44];
          R_data14 <= memory[45];
          R_data15 <= memory[46];
          R_data16 <= memory[47];
        end
        3: begin
          R_data1 <= memory[48];
          R_data2 <= memory[49];
          R_data3 <= memory[50];
          R_data4 <= memory[51];
          R_data5 <= memory[52];
          R_data6 <= memory[53];
          R_data7 <= memory[54];
          R_data8 <= memory[55];
          R_data9 <= memory[56];
          R_data10 <= memory[57];
          R_data11 <= memory[58];
          R_data12 <= memory[59];
          R_data13 <= memory[60];
          R_data14 <= memory[61];
          R_data15 <= memory[62];
          R_data16 <= memory[63];
        end
        4: begin
          R_data1 <= memory[64];
          R_data2 <= memory[65];
          R_data3 <= memory[66];
          R_data4 <= memory[67];
          R_data5 <= memory[68];
          R_data6 <= memory[69];
          R_data7 <= memory[70];
          R_data8 <= memory[71];
          R_data9 <= memory[72];
          R_data10 <= memory[73];
          R_data11 <= memory[74];
          R_data12 <= memory[75];
          R_data13 <= memory[76];
          R_data14 <= memory[77];
          R_data15 <= memory[78];
          R_data16 <= memory[79];
        end
        5: begin
          R_data1 <= memory[80];
          R_data2 <= memory[81];
          R_data3 <= memory[82];
          R_data4 <= memory[83];
          R_data5 <= memory[84];
          R_data6 <= memory[85];
          R_data7 <= memory[86];
          R_data8 <= memory[87];
          R_data9 <= memory[88];
          R_data10 <= memory[89];
          R_data11 <= memory[90];
          R_data12 <= memory[91];
          R_data13 <= memory[92];
          R_data14 <= memory[93];
          R_data15 <= memory[94];
          R_data16 <= memory[95];
        end
        6: begin
          R_data1 <= memory[96];
          R_data2 <= memory[97];
          R_data3 <= memory[98];
          R_data4 <= memory[99];
          R_data5 <= memory[100];
          R_data6 <= memory[101];
          R_data7 <= memory[102];
          R_data8 <= memory[103];
          R_data9 <= memory[104];
          R_data10 <= memory[105];
          R_data11 <= memory[106];
          R_data12 <= memory[107];
          R_data13 <= memory[108];
          R_data14 <= memory[109];
          R_data15 <= memory[110];
          R_data16 <= memory[111];
        end
        7: begin
          R_data1 <= memory[112];
          R_data2 <= memory[113];
          R_data3 <= memory[114];
          R_data4 <= memory[115];
          R_data5 <= memory[116];
          R_data6 <= memory[117];
          R_data7 <= memory[118];
          R_data8 <= memory[119];
          R_data9 <= memory[120];
          R_data10 <= memory[121];
          R_data11 <= memory[122];
          R_data12 <= memory[123];
          R_data13 <= memory[124];
          R_data14 <= memory[125];
          R_data15 <= memory[126];
          R_data16 <= memory[127];
        end
        8: begin
          R_data1 <= memory[128];
          R_data2 <= memory[129];
          R_data3 <= memory[130];
          R_data4 <= memory[131];
          R_data5 <= memory[132];
          R_data6 <= memory[133];
          R_data7 <= memory[134];
          R_data8 <= memory[135];
          R_data9 <= memory[136];
          R_data10 <= memory[137];
          R_data11 <= memory[138];
          R_data12 <= memory[139];
          R_data13 <= memory[140];
          R_data14 <= memory[141];
          R_data15 <= memory[142];
          R_data16 <= memory[143];
        end
        9: begin
          R_data1 <= memory[144];
          R_data2 <= memory[145];
          R_data3 <= memory[146];
          R_data4 <= memory[147];
          R_data5 <= memory[148];
          R_data6 <= memory[149];
          R_data7 <= memory[150];
          R_data8 <= memory[151];
          R_data9 <= memory[152];
          R_data10 <= memory[153];
          R_data11 <= memory[154];
          R_data12 <= memory[155];
          R_data13 <= memory[156];
          R_data14 <= memory[157];
          R_data15 <= memory[158];
          R_data16 <= memory[159];
        end
        10: begin
          R_data1 <= memory[160];
          R_data2 <= memory[161];
          R_data3 <= memory[162];
          R_data4 <= memory[163];
          R_data5 <= memory[164];
          R_data6 <= memory[165];
          R_data7 <= memory[166];
          R_data8 <= memory[167];
          R_data9 <= memory[168];
          R_data10 <= memory[169];
          R_data11 <= memory[170];
          R_data12 <= memory[171];
          R_data13 <= memory[172];
          R_data14 <= memory[173];
          R_data15 <= memory[174];
          R_data16 <= memory[175];
        end
        11: begin
          R_data1 <= memory[176];
          R_data2 <= memory[177];
          R_data3 <= memory[178];
          R_data4 <= memory[179];
          R_data5 <= memory[180];
          R_data6 <= memory[181];
          R_data7 <= memory[182];
          R_data8 <= memory[183];
          R_data9 <= memory[184];
          R_data10 <= memory[185];
          R_data11 <= memory[186];
          R_data12 <= memory[187];
          R_data13 <= memory[188];
          R_data14 <= memory[189];
          R_data15 <= memory[190];
          R_data16 <= memory[191];
        end
        12: begin
          R_data1 <= memory[192];
          R_data2 <= memory[193];
          R_data3 <= memory[194];
          R_data4 <= memory[195];
          R_data5 <= memory[196];
          R_data6 <= memory[197];
          R_data7 <= memory[198];
          R_data8 <= memory[199];
          R_data9 <= memory[200];
          R_data10 <= memory[201];
          R_data11 <= memory[202];
          R_data12 <= memory[203];
          R_data13 <= memory[204];
          R_data14 <= memory[205];
          R_data15 <= memory[206];
          R_data16 <= memory[207];
        end
        13: begin
          R_data1 <= memory[208];
          R_data2 <= memory[209];
          R_data3 <= memory[210];
          R_data4 <= memory[211];
          R_data5 <= memory[212];
          R_data6 <= memory[213];
          R_data7 <= memory[214];
          R_data8 <= memory[215];
          R_data9 <= memory[216];
          R_data10 <= memory[217];
          R_data11 <= memory[218];
          R_data12 <= memory[219];
          R_data13 <= memory[220];
          R_data14 <= memory[221];
          R_data15 <= memory[222];
          R_data16 <= memory[223];
        end
        14: begin
          R_data1 <= memory[224];
          R_data2 <= memory[225];
          R_data3 <= memory[226];
          R_data4 <= memory[227];
          R_data5 <= memory[228];
          R_data6 <= memory[229];
          R_data7 <= memory[230];
          R_data8 <= memory[231];
          R_data9 <= memory[232];
          R_data10 <= memory[233];
          R_data11 <= memory[234];
          R_data12 <= memory[235];
          R_data13 <= memory[236];
          R_data14 <= memory[237];
          R_data15 <= memory[238];
          R_data16 <= memory[239];
        end
        15: begin
          R_data1 <= memory[240];
          R_data2 <= memory[241];
          R_data3 <= memory[242];
          R_data4 <= memory[243];
          R_data5 <= memory[244];
          R_data6 <= memory[245];
          R_data7 <= memory[246];
          R_data8 <= memory[247];
          R_data9 <= memory[248];
          R_data10 <= memory[249];
          R_data11 <= memory[250];
          R_data12 <= memory[251];
          R_data13 <= memory[252];
          R_data14 <= memory[253];
          R_data15 <= memory[254];
          R_data16 <= memory[255];
        end
        default: ;
      endcase
    end
  end

endmodule
```

Supplementary Figure 5: Level 2 Memory for Activation.



```
module Tiled_SDS_Unit(R_data1, R_data2, R_data3, R_data4, R_data5,
R_data6, R_data7, R_data8, R_data9, R_data10, R_data11, R_data12,
R_data13, R_data14, R_data15, R_data16, W_data1, W_data2,
W_data3, W_data4, W_data5, W_data6, W_data7, W_data8, W_data9,
W_data10, W_data11, W_data12, W_data13, W_data14, W_data15,
W_data16, rd, wr, tiled_computing_sig, zerofill_sig,
matrix_compute_size, clk, rst);
  parameter address_size = 8;
  parameter word_size = 8;
  parameter matrix_size = 6;
  parameter memory_size = 64;
  output reg [word_size-1:0] R_data1, R_data2, R_data3, R_data4,
R_data5, R_data6, R_data7, R_data8;
  output reg [word_size-1:0] R_data9, R_data10, R_data11, R_data12,
R_data13, R_data14, R_data15, R_data16;
  input [word_size-1:0] W_data1, W_data2, W_data3, W_data4,
W_data5, W_data6, W_data7, W_data8;
  input [word_size-1:0] W_data9, W_data10, W_data11, W_data12,
W_data13, W_data14, W_data15, W_data16;
  input [matrix_size-1:0] matrix_compute_size;
  input rd, wr;
  input clk, rst;
  input tiled_computing_sig, zerofill_sig;
  reg [word_size-1:0] memory [memory_size-1:0];
  reg [word_size-1:0] rd_counter;
  reg [word_size-1:0] wr_counter;
  reg [matrix_size-1:0] matrix_compute_size_reg;
  reg tiled_computing_sig_reg, zerofill_sig_reg;
  integer i;

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      rd_counter <= 0;
    end else if (rd) begin
      rd_counter <= rd_counter + 1;
    end
  end

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      wr_counter <= 0;
    end else if (wr) begin
      wr_counter <= wr_counter + 1;
    end
  end

  always@(*) begin
    matrix_compute_size_reg = matrix_compute_size;
    tiled_computing_sig_reg = tiled_computing_sig;
    zerofill_sig_reg = zerofill_sig;
  end

  always @(posedge clk) begin
    if (zerofill_sig_reg) begin
      for (i = 0; i < 64; i = i + 1) begin
        memory[i] <= 8'b00000000;
      end
    end
    if (wr) begin
      if (tiled_computing_sig_reg) begin
        case (matrix_compute_size_reg)
          5: begin
              case (wr_counter)
                0: begin
                  memory[0]  <= W_data1;
                  memory[1]  <= W_data2;
                  memory[2]  <= W_data3;
                  memory[3]  <= W_data4;
                  memory[32] <= W_data5;
                  memory[4]  <= W_data6;
                  memory[5]  <= W_data7;
                  memory[6]  <= W_data8;
                  memory[7]  <= W_data9;
                  memory[36] <= W_data10;
                  memory[8]  <= W_data11;
                  memory[9]  <= W_data12;
                  memory[10] <= W_data13;
                  memory[11] <= W_data14;
                  memory[40] <= W_data15;
                  memory[12] <= W_data16;
                end
                1: begin
                  memory[13] <= W_data1;
                  memory[14] <= W_data2;
                  memory[15] <= W_data3;
                  memory[44] <= W_data4;
                  memory[16] <= W_data5;
                  memory[17] <= W_data6;
                  memory[18] <= W_data7;
                  memory[19] <= W_data8;
                  memory[48] <= W_data9;
                end
                default: ;
                endcase
              end
          6: begin
              case (wr_counter)
                0: begin
                  memory[0]  <= W_data1;
                  memory[1]  <= W_data2;
                  memory[2]  <= W_data3;
                  memory[3]  <= W_data4;
                  memory[32] <= W_data5;
                  memory[33] <= W_data6;
                  memory[4]  <= W_data7;
                  memory[5]  <= W_data8;
                  memory[6]  <= W_data9;
                  memory[7]  <= W_data10;
                  memory[36] <= W_data11;
                  memory[37] <= W_data12;
                  memory[8]  <= W_data13;
                  memory[9]  <= W_data14;
                  memory[10] <= W_data15;
                  memory[11] <= W_data16;
                end
                1: begin
                  memory[40] <= W_data1;
                  memory[41] <= W_data2;
                  memory[12] <= W_data3;
                  memory[13] <= W_data4;
                  memory[14] <= W_data5;
                  memory[15] <= W_data6;
                  memory[44] <= W_data7;
                  memory[45] <= W_data8;
                  memory[16] <= W_data9;
                  memory[17] <= W_data10;
                  memory[18] <= W_data11;
                  memory[19] <= W_data12;
                  memory[48] <= W_data13;
                  memory[49] <= W_data14;
                  memory[20] <= W_data15;
                  memory[21] <= W_data16;
                end
```

```
  2: begin
      memory[22]  <= W_data1;
      memory[23]  <= W_data2;
      memory[52]  <= W_data3;
      memory[53]  <= W_data4;
    end
      default: ;
    endcase
  end
  7: begin
    case (wr_counter)
    0: begin
      memory[0]  <= W_data1;
      memory[1]  <= W_data2;
      memory[2]  <= W_data3;
      memory[3]  <= W_data4;
      memory[32] <= W_data5;
      memory[33] <= W_data6;
      memory[34] <= W_data7;
      memory[4]  <= W_data8;
      memory[5]  <= W_data9;
      memory[6]  <= W_data10;
      memory[7]  <= W_data11;
      memory[36] <= W_data12;
      memory[37] <= W_data13;
      memory[38] <= W_data14;
      memory[8]  <= W_data15;
      memory[9]  <= W_data16;
    end
  1: begin
      memory[10]  <= W_data1;
      memory[11]  <= W_data2;
      memory[40]  <= W_data3;
      memory[41]  <= W_data4;
      memory[42]  <= W_data5;
      memory[12]  <= W_data6;
      memory[13]  <= W_data7;
      memory[14]  <= W_data8;
      memory[15]  <= W_data9;
      memory[44]  <= W_data10;
      memory[45]  <= W_data11;
      memory[46]  <= W_data12;
      memory[16]  <= W_data13;
      memory[17]  <= W_data14;
      memory[18]  <= W_data15;
      memory[19]  <= W_data16;
    end
  2: begin
      memory[48]  <= W_data1;
      memory[49]  <= W_data2;
      memory[50]  <= W_data3;
      memory[20]  <= W_data4;
      memory[21]  <= W_data5;
      memory[22]  <= W_data6;
      memory[23]  <= W_data7;
      memory[52]  <= W_data8;
      memory[53]  <= W_data9;
      memory[54]  <= W_data10;
      memory[24]  <= W_data11;
      memory[25]  <= W_data12;
      memory[26]  <= W_data13;
      memory[27]  <= W_data14;
      memory[56]  <= W_data15;
      memory[57]  <= W_data16;
    end
  3: begin
      memory[58]  <= W_data1;
    end
        default: ;
    endcase
  end
  8: begin
    case (wr_counter)
    0: begin
      memory[0]  <= W_data1;
      memory[1]  <= W_data2;
      memory[2]  <= W_data3;
      memory[3]  <= W_data4;
      memory[32]  <= W_data5;
      memory[33]  <= W_data6;
      memory[34]  <= W_data7;
      memory[35]  <= W_data8;
      memory[4]  <= W_data9;
      memory[5]  <= W_data10;
      memory[6]  <= W_data11;
      memory[7]  <= W_data12;
      memory[36]  <= W_data13;
      memory[37]  <= W_data14;
      memory[38]  <= W_data15;
      memory[39]  <= W_data16;
    end
    1: begin
      memory[8]  <= W_data1;
      memory[9]  <= W_data2;
      memory[10]  <= W_data3;
      memory[11]  <= W_data4;
      memory[40]  <= W_data5;
      memory[41]  <= W_data6;
      memory[42]  <= W_data7;
      memory[43]  <= W_data8;
      memory[12]  <= W_data9;
      memory[13]  <= W_data10;
      memory[14]  <= W_data11;
      memory[15]  <= W_data12;
      memory[44]  <= W_data13;
      memory[45]  <= W_data14;
      memory[46]  <= W_data15;
      memory[47]  <= W_data16;
    end
```

```
  2: begin
      memory[16]  <= W_data1;
      memory[17]  <= W_data2;
      memory[18]  <= W_data3;
      memory[19]  <= W_data4;
      memory[48]  <= W_data5;
      memory[49]  <= W_data6;
      memory[50]  <= W_data7;
      memory[51]  <= W_data8;
      memory[20]  <= W_data9;
      memory[21]  <= W_data10;
      memory[22]  <= W_data11;
      memory[23]  <= W_data12;
      memory[52]  <= W_data13;
      memory[53]  <= W_data14;
      memory[54]  <= W_data15;
      memory[55]  <= W_data16;
    end
  3: begin
      memory[24]  <= W_data1;
      memory[25]  <= W_data2;
      memory[26]  <= W_data3;
      memory[27]  <= W_data4;
      memory[56]  <= W_data5;
      memory[57]  <= W_data6;
      memory[58]  <= W_data7;
      memory[59]  <= W_data8;
      memory[28]  <= W_data9;
      memory[29]  <= W_data10;
      memory[30]  <= W_data11;
      memory[31]  <= W_data12;
      memory[60]  <= W_data13;
      memory[61]  <= W_data14;
      memory[62]  <= W_data15;
      memory[63]  <= W_data16;
    end
        default: ;
    endcase
  end
      default: ;
  endcase
  end else begin
  case (matrix_compute_size_reg)
    1: begin
      case (wr_counter)
      0: begin
        memory[0]  <= W_data1;
      end
      1: begin
        memory[16]  <= W_data1;
      end
      2: begin
        memory[32]  <= W_data1;
      end
      3: begin
        memory[48]  <= W_data1;
      end
        default: ;
      endcase
    end
    2: begin
      case (wr_counter)
      0: begin
        memory[0]  <= W_data1;
        memory[1]  <= W_data2;
        memory[4]  <= W_data3;
        memory[5]  <= W_data4;
      end
      1: begin
        memory[16]  <= W_data1;
        memory[17]  <= W_data2;
        memory[20]  <= W_data3;
        memory[21]  <= W_data4;
      end
      2: begin
        memory[32]  <= W_data1;
        memory[33]  <= W_data2;
        memory[36]  <= W_data3;
        memory[37]  <= W_data4;
      end
      3: begin
        memory[48]  <= W_data1;
        memory[49]  <= W_data2;
        memory[52]  <= W_data3;
        memory[53]  <= W_data4;
      end
        default: ;
      endcase
    end
    3: begin
      case (wr_counter)
      0: begin
        memory[0]  <= W_data1;
        memory[1]  <= W_data2;
        memory[2]  <= W_data3;
        memory[4]  <= W_data4;
        memory[5]  <= W_data5;
        memory[6]  <= W_data6;
        memory[8]  <= W_data7;
        memory[9]  <= W_data8;
        memory[10]  <= W_data9;
      end
      1: begin
        memory[16]  <= W_data1;
        memory[17]  <= W_data2;
        memory[18]  <= W_data3;
        memory[20]  <= W_data4;
        memory[21]  <= W_data5;
        memory[22]  <= W_data6;
        memory[24]  <= W_data7;
        memory[25]  <= W_data8;
        memory[26]  <= W_data9;
      end
```

```
  2: begin
      memory[32]  <= W_data1;
      memory[33]  <= W_data2;
      memory[34]  <= W_data3;
      memory[36]  <= W_data4;
      memory[37]  <= W_data5;
      memory[38]  <= W_data6;
      memory[40]  <= W_data7;
      memory[41]  <= W_data8;
      memory[42]  <= W_data9;
    end
  3: begin
      memory[48]  <= W_data1;
      memory[49]  <= W_data2;
      memory[50]  <= W_data3;
      memory[52]  <= W_data4;
      memory[53]  <= W_data5;
      memory[54]  <= W_data6;
      memory[56]  <= W_data7;
      memory[57]  <= W_data8;
      memory[58]  <= W_data9;
    end
      default: ;
    endcase
  end
  4: begin
    case (wr_counter)
    0: begin
      memory[0]  <= W_data1;
      memory[1]  <= W_data2;
      memory[2]  <= W_data3;
      memory[3]  <= W_data4;
      memory[4]  <= W_data5;
      memory[5]  <= W_data6;
      memory[6]  <= W_data7;
      memory[7]  <= W_data8;
      memory[8]  <= W_data9;
      memory[9]  <= W_data10;
      memory[10]  <= W_data11;
      memory[11]  <= W_data12;
      memory[12]  <= W_data13;
      memory[13]  <= W_data14;
      memory[14]  <= W_data15;
      memory[15]  <= W_data16;
    end
    1: begin
      memory[16]  <= W_data1;
      memory[17]  <= W_data2;
      memory[18]  <= W_data3;
      memory[19]  <= W_data4;
      memory[20]  <= W_data5;
      memory[21]  <= W_data6;
      memory[22]  <= W_data7;
      memory[23]  <= W_data8;
      memory[24]  <= W_data9;
      memory[25]  <= W_data10;
      memory[26]  <= W_data11;
      memory[27]  <= W_data12;
      memory[28]  <= W_data13;
      memory[29]  <= W_data14;
      memory[30]  <= W_data15;
      memory[31]  <= W_data16;
    end
    2: begin
      memory[32]  <= W_data1;
      memory[33]  <= W_data2;
      memory[34]  <= W_data3;
      memory[35]  <= W_data4;
      memory[36]  <= W_data5;
      memory[37]  <= W_data6;
      memory[38]  <= W_data7;
      memory[39]  <= W_data8;
      memory[40]  <= W_data9;
      memory[41]  <= W_data10;
      memory[42]  <= W_data11;
      memory[43]  <= W_data12;
      memory[44]  <= W_data13;
      memory[45]  <= W_data14;
      memory[46]  <= W_data15;
      memory[47]  <= W_data16;
    end
    3: begin
      memory[48]  <= W_data1;
      memory[49]  <= W_data2;
      memory[50]  <= W_data3;
      memory[51]  <= W_data4;
      memory[52]  <= W_data5;
      memory[53]  <= W_data6;
      memory[54]  <= W_data7;
      memory[55]  <= W_data8;
      memory[56]  <= W_data9;
      memory[57]  <= W_data10;
      memory[58]  <= W_data11;
      memory[59]  <= W_data12;
      memory[60]  <= W_data13;
      memory[61]  <= W_data14;
      memory[62]  <= W_data15;
      memory[63]  <= W_data16;
    end
      default: ;
    endcase
    end
  end
  end
```

```
  always @(posedge clk) begin
      if (rd) begin
        case (rd_counter)
    0: begin
      R_data1  <= memory[0];
      R_data2  <= memory[1];
      R_data3  <= memory[2];
      R_data4  <= memory[3];
      R_data5  <= memory[4];
      R_data6  <= memory[5];
      R_data7  <= memory[6];
      R_data8  <= memory[7];
      R_data9  <= memory[8];
      R_data10 <= memory[9];
      R_data11 <= memory[10];
      R_data12 <= memory[11];
      R_data13 <= memory[12];
      R_data14 <= memory[13];
      R_data15 <= memory[14];
      R_data16 <= memory[15];
    end
    1: begin
      R_data1  <= memory[16];
      R_data2  <= memory[17];
      R_data3  <= memory[18];
      R_data4  <= memory[19];
      R_data5  <= memory[20];
      R_data6  <= memory[21];
      R_data7  <= memory[22];
      R_data8  <= memory[23];
      R_data9  <= memory[24];
      R_data10 <= memory[25];
      R_data11 <= memory[26];
      R_data12 <= memory[27];
      R_data13 <= memory[28];
      R_data14 <= memory[29];
      R_data15 <= memory[30];
      R_data16 <= memory[31];
    end
    2: begin
      R_data1  <= memory[32];
      R_data2  <= memory[33];
      R_data3  <= memory[34];
      R_data4  <= memory[35];
      R_data5  <= memory[36];
      R_data6  <= memory[37];
      R_data7  <= memory[38];
      R_data8  <= memory[39];
      R_data9  <= memory[40];
      R_data10 <= memory[41];
      R_data11 <= memory[42];
      R_data12 <= memory[43];
      R_data13 <= memory[44];
      R_data14 <= memory[45];
      R_data15 <= memory[46];
      R_data16 <= memory[47];
    end
    3: begin
      R_data1  <= memory[48];
      R_data2  <= memory[49];
      R_data3  <= memory[50];
      R_data4  <= memory[51];
      R_data5  <= memory[52];
      R_data6  <= memory[53];
      R_data7  <= memory[54];
      R_data8  <= memory[55];
      R_data9  <= memory[56];
      R_data10 <= memory[57];
      R_data11 <= memory[58];
      R_data12 <= memory[59];
      R_data13 <= memory[60];
      R_data14 <= memory[61];
      R_data15 <= memory[62];
      R_data16 <= memory[63];
    end
      default: ;
    endcase
    end
  end

endmodule
```

Supplementary Figure 6: Tiled Systolic Data Setup Unit.

```
module L1_Host_Memory_Weight(R_data, W_data, addr, clk, rd, wr,
W_data_from_TsdsU1, W_data_from_TsdsU2, W_data_from_TsdsU3,
W_data_from_TsdsU4, W_data_from_TsdsU5, W_data_from_TsdsU6,
W_data_from_TsdsU7, W_data_from_TsdsU8, W_data_from_TsdsU9,
W_data_from_TsdsU10, W_data_from_TsdsU11, W_data_from_TsdsU12,
W_data_from_TsdsU13, W_data_from_TsdsU14, W_data_from_TsdsU15,
W_data_from_TsdsU16, W_wr_from_TsdsU, L1HW_counter_rst);
    parameter address_size = 6;
    parameter word_size = 8;
    parameter memory_size = 64;
    output reg [word_size-1:0] R_data;
    input [word_size-1:0] W_data;
    input [word_size-1:0] W_data_from_TsdsU1, W_data_from_TsdsU2,
W_data_from_TsdsU3, W_data_from_TsdsU4, W_data_from_TsdsU5,
W_data_from_TsdsU6, W_data_from_TsdsU7, W_data_from_TsdsU8;
    input [word_size-1:0] W_data_from_TsdsU9, W_data_from_TsdsU10,
W_data_from_TsdsU11, W_data_from_TsdsU12, W_data_from_TsdsU13,
W_data_from_TsdsU14, W_data_from_TsdsU15, W_data_from_TsdsU16;
    input W_wr_from_TsdsU;
    input [address_size-1:0] addr;
    input clk, rd, wr, L1HW_counter_rst;
    reg [address_size-1:0] memory [memory_size-1:0];
    reg [word_size-1:0] counter;

    always @(posedge clk or posedge L1HW_counter_rst) begin
        if (L1HW_counter_rst) begin
            counter <= 0;
        end else if (W_wr_from_TsdsU) begin
            counter <= counter + 1;
        end
    end

    always @(posedge clk) begin
        if (wr) begin
            memory[addr] <= W_data;
        end else if (W_wr_from_TsdsU) begin
            case (counter)
                0: begin
                    memory[0]  <= W_data_from_TsdsU1;
                    memory[1]  <= W_data_from_TsdsU2;
                    memory[2]  <= W_data_from_TsdsU3;
                    memory[3]  <= W_data_from_TsdsU4;
                    memory[4]  <= W_data_from_TsdsU5;
                    memory[5]  <= W_data_from_TsdsU6;
                    memory[6]  <= W_data_from_TsdsU7;
                    memory[7]  <= W_data_from_TsdsU8;
                    memory[8]  <= W_data_from_TsdsU9;
                    memory[9]  <= W_data_from_TsdsU10;
                    memory[10] <= W_data_from_TsdsU11;
                    memory[11] <= W_data_from_TsdsU12;
                    memory[12] <= W_data_from_TsdsU13;
                    memory[13] <= W_data_from_TsdsU14;
                    memory[14] <= W_data_from_TsdsU15;
                    memory[15] <= W_data_from_TsdsU16;
                end
                1: begin
                    memory[16]  <= W_data_from_TsdsU1;
                    memory[17]  <= W_data_from_TsdsU2;
                    memory[18]  <= W_data_from_TsdsU3;
                    memory[19]  <= W_data_from_TsdsU4;
                    memory[20]  <= W_data_from_TsdsU5;
                    memory[21]  <= W_data_from_TsdsU6;
                    memory[22]  <= W_data_from_TsdsU7;
                    memory[23]  <= W_data_from_TsdsU8;
                    memory[24]  <= W_data_from_TsdsU9;
                    memory[25] <= W_data_from_TsdsU10;
                    memory[26] <= W_data_from_TsdsU11;
                    memory[27] <= W_data_from_TsdsU12;
                    memory[28] <= W_data_from_TsdsU13;
                    memory[29] <= W_data_from_TsdsU14;
                    memory[30] <= W_data_from_TsdsU15;
                    memory[31] <= W_data_from_TsdsU16;
                end
                2: begin
                    memory[32]  <= W_data_from_TsdsU1;
                    memory[33]  <= W_data_from_TsdsU2;
                    memory[34]  <= W_data_from_TsdsU3;
                    memory[35]  <= W_data_from_TsdsU4;
                    memory[36]  <= W_data_from_TsdsU5;
                    memory[37]  <= W_data_from_TsdsU6;
                    memory[38]  <= W_data_from_TsdsU7;
                    memory[39]  <= W_data_from_TsdsU8;
                    memory[40]  <= W_data_from_TsdsU9;
                    memory[41] <= W_data_from_TsdsU10;
                    memory[42] <= W_data_from_TsdsU11;
                    memory[43] <= W_data_from_TsdsU12;
                    memory[44] <= W_data_from_TsdsU13;
                    memory[45] <= W_data_from_TsdsU14;
                    memory[46] <= W_data_from_TsdsU15;
                    memory[47] <= W_data_from_TsdsU16;
                end
                3: begin
                    memory[48]  <= W_data_from_TsdsU1;
                    memory[49]  <= W_data_from_TsdsU2;
                    memory[50]  <= W_data_from_TsdsU3;
                    memory[51]  <= W_data_from_TsdsU4;
                    memory[52]  <= W_data_from_TsdsU5;
                    memory[53]  <= W_data_from_TsdsU6;
                    memory[54]  <= W_data_from_TsdsU7;
                    memory[55]  <= W_data_from_TsdsU8;
                    memory[56]  <= W_data_from_TsdsU9;
                    memory[57] <= W_data_from_TsdsU10;
                    memory[58] <= W_data_from_TsdsU11;
                    memory[59] <= W_data_from_TsdsU12;
                    memory[60] <= W_data_from_TsdsU13;
                    memory[61] <= W_data_from_TsdsU14;
                    memory[62] <= W_data_from_TsdsU15;
                    memory[63] <= W_data_from_TsdsU16;
                end
                default: ;
            endcase
        end
    end

    always @(posedge clk) begin
        if (rd) begin
            R_data <= memory[addr];
        end
    end

endmodule
```

Supplementary Figure 7: Level 1 Host Memory for Weight.

```
module L1_Host_Memory_Activation(tiled_computing_sig, R_data, W_data, rd_addr, wr_addr, clk, rd, wr,
A_data_from_TsdsU1, A_data_from_TsdsU2, A_data_from_TsdsU3, A_data_from_TsdsU4, A_data_from_TsdsU5,
A_data_from_TsdsU6, A_data_from_TsdsU7, A_data_from_TsdsU8, A_data_from_TsdsU9, A_data_from_TsdsU10,
A_data_from_TsdsU11, A_data_from_TsdsU12, A_data_from_TsdsU13, A_data_from_TsdsU14,
A_data_from_TsdsU15, A_data_from_TsdsU16, A_wr_from_TsdsU, A_data_backto_L2_Accumu1,
A_data_backto_L2_Accumu2, A_data_backto_L2_Accumu3, A_data_backto_L2_Accumu4,
A_data_backto_L2_Accumu5, A_data_backto_L2_Accumu6, A_data_backto_L2_Accumu7,
A_data_backto_L2_Accumu8, A_data_backto_L2_Accumu9, A_data_backto_L2_Accumu10,
A_data_backto_L2_Accumu11, A_data_backto_L2_Accumu12, A_data_backto_L2_Accumu13,
A_data_backto_L2_Accumu14, A_data_backto_L2_Accumu15, A_data_backto_L2_Accumu16,
A_rd_backto_L2HM, L1HW_counter_rst);
    parameter address_size = 6;
    parameter word_size = 8;
    parameter memory_size = 64;
    input tiled_computing_sig;
    output reg [word_size-1:0] R_data;
    input [word_size-1:0] W_data;
    input [word_size-1:0] A_data_from_TsdsU1, A_data_from_TsdsU2, A_data_from_TsdsU3, A_data_from_TsdsU4,
A_data_from_TsdsU5, A_data_from_TsdsU6, A_data_from_TsdsU7, A_data_from_TsdsU8;
    input [word_size-1:0] A_data_from_TsdsU9, A_data_from_TsdsU10, A_data_from_TsdsU11,
A_data_from_TsdsU12, A_data_from_TsdsU13, A_data_from_TsdsU14, A_data_from_TsdsU15,
A_data_from_TsdsU16;
    output reg [word_size-1:0] A_data_backto_L2_Accumu1, A_data_backto_L2_Accumu2,
A_data_backto_L2_Accumu3, A_data_backto_L2_Accumu4, A_data_backto_L2_Accumu5,
A_data_backto_L2_Accumu6, A_data_backto_L2_Accumu7, A_data_backto_L2_Accumu8;
    output reg [word_size-1:0] A_data_backto_L2_Accumu9, A_data_backto_L2_Accumu10,
A_data_backto_L2_Accumu11, A_data_backto_L2_Accumu12, A_data_backto_L2_Accumu13,
A_data_backto_L2_Accumu14, A_data_backto_L2_Accumu15, A_data_backto_L2_Accumu16;
    input A_rd_backto_L2HM;
    input A_wr_from_TsdsU;
    input [address_size-1:0] rd_addr, wr_addr;
    input clk, rd, wr, L1HW_counter_rst;
    reg [word_size-1:0] memory [memory_size-1:0];
    reg [word_size-1:0] wr_counter, rd_counter;

    always @(posedge clk or posedge L1HW_counter_rst) begin
        if (L1HW_counter_rst) begin
            wr_counter <= 0;
            rd_counter <= 0;
        end else if (A_wr_from_TsdsU) begin
            wr_counter <= wr_counter + 1;
        end else if (A_rd_backto_L2HM) begin
            rd_counter <= rd_counter + 1;
        end
    end

    always @(posedge clk) begin
        if (!tiled_computing_sig && wr) begin
            memory[wr_addr] <= W_data;
        end else if (tiled_computing_sig && wr) begin
            case (wr_addr)
                0:  memory[0]  <= W_data;
                1:  memory[32] <= W_data;
                2:  memory[4]  <= W_data;
                3:  memory[36] <= W_data;
                4:  memory[1]  <= W_data;
                5:  memory[33] <= W_data;
                6:  memory[5]  <= W_data;
                7:  memory[37] <= W_data;
                8:  memory[2]  <= W_data;
                9:  memory[34] <= W_data;
                10: memory[6]  <= W_data;
                11: memory[38] <= W_data;
                12: memory[3]  <= W_data;
                13: memory[35] <= W_data;
                14: memory[7]  <= W_data;
                15: memory[39] <= W_data;
                16: memory[8]  <= W_data;
                17: memory[40] <= W_data;
                18: memory[12] <= W_data;
                19: memory[44] <= W_data;
                20: memory[9]  <= W_data;
                21: memory[41] <= W_data;
                22: memory[13] <= W_data;
                23: memory[45] <= W_data;
                24: memory[10] <= W_data;
                25: memory[42] <= W_data;
                26: memory[14] <= W_data;
                27: memory[46] <= W_data;
                28: memory[11] <= W_data;
                29: memory[43] <= W_data;
                30: memory[15] <= W_data;
                31: memory[47] <= W_data;
                32: memory[16] <= W_data;
                33: memory[48] <= W_data;
                34: memory[20] <= W_data;
                35: memory[52] <= W_data;
                36: memory[17] <= W_data;
                37: memory[49] <= W_data;
                38: memory[21] <= W_data;
                39: memory[53] <= W_data;
                40: memory[18] <= W_data;
                41: memory[50] <= W_data;
                42: memory[22] <= W_data;
                43: memory[54] <= W_data;
                44: memory[19] <= W_data;
                45: memory[51] <= W_data;
                46: memory[23] <= W_data;
                47: memory[55] <= W_data;
                48: memory[24] <= W_data;
                49: memory[56] <= W_data;
                50: memory[28] <= W_data;
                51: memory[60] <= W_data;
                52: memory[25] <= W_data;
                53: memory[57] <= W_data;
                54: memory[29] <= W_data;
                55: memory[61] <= W_data;
                56: memory[26] <= W_data;
                57: memory[58] <= W_data;
                58: memory[30] <= W_data;
                59: memory[62] <= W_data;
                60: memory[27] <= W_data;
                61: memory[59] <= W_data;
                62: memory[31] <= W_data;
                63: memory[63] <= W_data;
            endcase
        end else if (A_wr_from_TsdsU) begin
            case (wr_counter)
                0: begin
                    memory[0]  <= A_data_from_TsdsU1;
                    memory[1]  <= A_data_from_TsdsU2;
                    memory[2]  <= A_data_from_TsdsU3;
                    memory[3]  <= A_data_from_TsdsU4;
                    memory[4]  <= A_data_from_TsdsU5;
                    memory[5]  <= A_data_from_TsdsU6;
                    memory[6]  <= A_data_from_TsdsU7;
                    memory[7]  <= A_data_from_TsdsU8;
                    memory[8]  <= A_data_from_TsdsU9;
                    memory[9]  <= A_data_from_TsdsU10;
                    memory[10] <= A_data_from_TsdsU11;
                    memory[11] <= A_data_from_TsdsU12;
                    memory[12] <= A_data_from_TsdsU13;
                    memory[13] <= A_data_from_TsdsU14;
                    memory[14] <= A_data_from_TsdsU15;
                    memory[15] <= A_data_from_TsdsU16;
                end
                1: begin
                    memory[16]  <= A_data_from_TsdsU1;
                    memory[17]  <= A_data_from_TsdsU2;
                    memory[18]  <= A_data_from_TsdsU3;
                    memory[19]  <= A_data_from_TsdsU4;
                    memory[20]  <= A_data_from_TsdsU5;
                    memory[21]  <= A_data_from_TsdsU6;
                    memory[22]  <= A_data_from_TsdsU7;
                    memory[23]  <= A_data_from_TsdsU8;
                    memory[24]  <= A_data_from_TsdsU9;
                    memory[25] <= A_data_from_TsdsU10;
                    memory[26] <= A_data_from_TsdsU11;
                    memory[27] <= A_data_from_TsdsU12;
                    memory[28] <= A_data_from_TsdsU13;
                    memory[29] <= A_data_from_TsdsU14;
                    memory[30] <= A_data_from_TsdsU15;
                    memory[31] <= A_data_from_TsdsU16;
                end
                2: begin
                    memory[32]  <= A_data_from_TsdsU1;
                    memory[33]  <= A_data_from_TsdsU2;
                    memory[34]  <= A_data_from_TsdsU3;
                    memory[35]  <= A_data_from_TsdsU4;
                    memory[36]  <= A_data_from_TsdsU5;
                    memory[37]  <= A_data_from_TsdsU6;
                    memory[38]  <= A_data_from_TsdsU7;
                    memory[39]  <= A_data_from_TsdsU8;
                    memory[40]  <= A_data_from_TsdsU9;
                    memory[41] <= A_data_from_TsdsU10;
                    memory[42] <= A_data_from_TsdsU11;
                    memory[43] <= A_data_from_TsdsU12;
                    memory[44] <= A_data_from_TsdsU13;
                    memory[45] <= A_data_from_TsdsU14;
                    memory[46] <= A_data_from_TsdsU15;
                    memory[47] <= A_data_from_TsdsU16;
                end
                3: begin
                    memory[48]  <= A_data_from_TsdsU1;
                    memory[49]  <= A_data_from_TsdsU2;
                    memory[50]  <= A_data_from_TsdsU3;
                    memory[51]  <= A_data_from_TsdsU4;
                    memory[52]  <= A_data_from_TsdsU5;
                    memory[53]  <= A_data_from_TsdsU6;
                    memory[54]  <= A_data_from_TsdsU7;
                    memory[55]  <= A_data_from_TsdsU8;
                    memory[56]  <= A_data_from_TsdsU9;
                    memory[57] <= A_data_from_TsdsU10;
                    memory[58] <= A_data_from_TsdsU11;
                    memory[59] <= A_data_from_TsdsU12;
                    memory[60] <= A_data_from_TsdsU13;
                    memory[61] <= A_data_from_TsdsU14;
                    memory[62] <= A_data_from_TsdsU15;
                    memory[63] <= A_data_from_TsdsU16;
                end
                default: ;
            endcase
        end else if (A_rd_backto_L2HM) begin
            case (rd_counter)
                0: begin
                    A_data_backto_L2_Accumu1   <= memory[0];
                    A_data_backto_L2_Accumu2   <= memory[1];
                    A_data_backto_L2_Accumu3   <= memory[2];
                    A_data_backto_L2_Accumu4   <= memory[3];
                    A_data_backto_L2_Accumu5   <= memory[4];
                    A_data_backto_L2_Accumu6   <= memory[5];
                    A_data_backto_L2_Accumu7   <= memory[6];
                    A_data_backto_L2_Accumu8   <= memory[7];
                    A_data_backto_L2_Accumu9   <= memory[8];
                    A_data_backto_L2_Accumu10  <= memory[9];
                    A_data_backto_L2_Accumu11  <= memory[10];
                    A_data_backto_L2_Accumu12  <= memory[11];
                    A_data_backto_L2_Accumu13  <= memory[12];
                    A_data_backto_L2_Accumu14  <= memory[13];
                    A_data_backto_L2_Accumu15  <= memory[14];
                    A_data_backto_L2_Accumu16  <= memory[15];
                end
                1: begin
                    A_data_backto_L2_Accumu1   <= memory[16];
                    A_data_backto_L2_Accumu2   <= memory[17];
                    A_data_backto_L2_Accumu3   <= memory[18];
                    A_data_backto_L2_Accumu4   <= memory[19];
                    A_data_backto_L2_Accumu5   <= memory[20];
                    A_data_backto_L2_Accumu6   <= memory[21];
                    A_data_backto_L2_Accumu7   <= memory[22];
                    A_data_backto_L2_Accumu8   <= memory[23];
                    A_data_backto_L2_Accumu9   <= memory[24];
                    A_data_backto_L2_Accumu10  <= memory[25];
                    A_data_backto_L2_Accumu11  <= memory[26];
                    A_data_backto_L2_Accumu12  <= memory[27];
                    A_data_backto_L2_Accumu13  <= memory[28];
                    A_data_backto_L2_Accumu14  <= memory[29];
                    A_data_backto_L2_Accumu15  <= memory[30];
                    A_data_backto_L2_Accumu16  <= memory[31];
                end
                2: begin
                    A_data_backto_L2_Accumu1   <= memory[32];
                    A_data_backto_L2_Accumu2   <= memory[33];
                    A_data_backto_L2_Accumu3   <= memory[34];
                    A_data_backto_L2_Accumu4   <= memory[35];
                    A_data_backto_L2_Accumu5   <= memory[36];
                    A_data_backto_L2_Accumu6   <= memory[37];
                    A_data_backto_L2_Accumu7   <= memory[38];
                    A_data_backto_L2_Accumu8   <= memory[39];
                    A_data_backto_L2_Accumu9   <= memory[40];
                    A_data_backto_L2_Accumu10  <= memory[41];
                    A_data_backto_L2_Accumu11  <= memory[42];
                    A_data_backto_L2_Accumu12  <= memory[43];
                    A_data_backto_L2_Accumu13  <= memory[44];
                    A_data_backto_L2_Accumu14  <= memory[45];
                    A_data_backto_L2_Accumu15  <= memory[46];
                    A_data_backto_L2_Accumu16  <= memory[47];
                end
                3: begin
                    A_data_backto_L2_Accumu1   <= memory[48];
                    A_data_backto_L2_Accumu2   <= memory[49];
                    A_data_backto_L2_Accumu3   <= memory[50];
                    A_data_backto_L2_Accumu4   <= memory[51];
                    A_data_backto_L2_Accumu5   <= memory[52];
                    A_data_backto_L2_Accumu6   <= memory[53];
                    A_data_backto_L2_Accumu7   <= memory[54];
                    A_data_backto_L2_Accumu8   <= memory[55];
                    A_data_backto_L2_Accumu9   <= memory[56];
                    A_data_backto_L2_Accumu10  <= memory[57];
                    A_data_backto_L2_Accumu11  <= memory[58];
                    A_data_backto_L2_Accumu12  <= memory[59];
                    A_data_backto_L2_Accumu13  <= memory[60];
                    A_data_backto_L2_Accumu14  <= memory[61];
                    A_data_backto_L2_Accumu15  <= memory[62];
                    A_data_backto_L2_Accumu16  <= memory[63];
                end
                default: ;
            endcase
        end
    end

    always @(posedge clk) begin
        if (rd) begin
            R_data <= memory[rd_addr];
        end
    end

endmodule
```

Supplementary Figure 8: Level 1 Host Memory for Activation.

```
module IR_Mem(R_data, W_data, addr, IR_wr_addr, clk, IR_rd, IR_wr);
    parameter address_size = 8;
    parameter word_size = 16;
    parameter memory_size = 64;
    output [word_size-1:0] R_data;
    input [word_size-1:0] W_data;
    input [address_size-1:0] addr, IR_wr_addr;
    input clk, IR_rd, IR_wr;
    reg [word_size-1:0] memory [memory_size-1:0];

    always @(posedge clk) begin
        if (IR_wr) begin
            memory[IR_wr_addr] <= W_data;
        end
    end
    assign R_data = memory[addr];

endmodule
```

Supplementary Figure 9: IR Memory.

```
module IR_counter(IR_addr, IR_inc, IR_clr, clk);
    parameter word_size = 8;
    output reg [word_size-1:0] IR_addr;
    input   IR_inc, IR_clr, clk;

    always@(posedge clk or posedge IR_clr) begin
        if(IR_clr==1) IR_addr <= 8'b0;
        else if (IR_inc == 1) IR_addr <= IR_addr + 1;
    end

endmodule
```

Supplementary Figure 10: IR Counter.

```
module Weight_DDR3(R_data, W_data, addr, clk, rd, wr);
    parameter address_size = 4;
    parameter word_size = 8;
    parameter memory_size = 64;
    output [word_size-1:0] R_data;
    input [word_size-1:0] W_data;
    input [address_size:0] addr;
    input clk, rd, wr;
    reg [word_size-1:0] memory [memory_size-1:0];

    always @(posedge clk) begin
        if (wr) begin
            memory[addr] <= W_data;
        end
    end

    assign R_data = memory[addr];

endmodule
```

Supplementary Figure 11: Weight DDR3.

```
module Weight_interface(clk, reset, weight_in, push_time, push, pop, pop_complete, out1, out2, out3,
out4);

    parameter Data_size = 8;
    parameter STACK_SIZE = 16;
    reg [Data_size-1:0] stack [STACK_SIZE-1:0];
    reg [Data_size-1:0] count;
    input clk;
    input reset;
    input push, pop;
    input [Data_size-1:0] weight_in;
    input [Data_size-1:0] push_time;
    output reg pop_complete;
    output reg [Data_size-1:0] out1;
    output reg [Data_size-1:0] out2;
    output reg [Data_size-1:0] out3;
    output reg [Data_size-1:0] out4;

    reg [Data_size-1:0] pushtime;
    reg [Data_size-1:0] pop_count;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pushtime <= 5'b0;
        end else begin
            pushtime <= push_time;
        end
    end

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            count <= 0;
        end else if (push) begin
            stack[count] <= weight_in;
            count <= count + 1;
        end else if (pop_complete) begin
            count <= 0;
        end
    end

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pop_count <= 0;
            out1 <= 8'b0;
            out2 <= 8'b0;
            out3 <= 8'b0;
            out4 <= 8'b0;
            pop_complete <= 1'b0;
        end else if (pop) begin
            case (pop_count)
                0: begin
                    out4 <= stack[15];
                    out3 <= stack[14];
                    out2 <= stack[13];
                    out1 <= stack[12];
                    pop_complete <= 1'b0;
                end
                1: begin
                    out4 <= stack[11];
                    out3 <= stack[10];
                    out2 <= stack[9];
                    out1 <= stack[8];
                    pop_complete <= 1'b0;
                end
                2: begin
                    out4 <= stack[7];
                    out3 <= stack[6];
                    out2 <= stack[5];
                    out1 <= stack[4];
                    pop_complete <= 1'b0;
                end
                3: begin
                    out4 <= stack[3];
                    out3 <= stack[2];
                    out2 <= stack[1];
                    out1 <= stack[0];
                    pop_complete <= 1'b0;
                end
                4: begin
                    pop_complete <= 1'b1;
                end
                default: begin
                    out1 <= 8'bx;
                    out2 <= 8'bx;
                    out3 <= 8'bx;
                    out4 <= 8'bx;
                end
            endcase
            pop_count <= pop_count + 1;
        end
    end

endmodule
```

Supplementary Figure 12: Weight Interface.

```
module Weight_FIFO(clk, rst, buf_in, buf_out, wr_en, rd_en, buf_empty, buf_full,
fifo_counter);

    parameter Data_size = 8;
    input clk, rst, wr_en, rd_en;
    input [7:0] buf_in;
    output [7:0] buf_out;
    output buf_empty, buf_full;
    output [7:0] fifo_counter;

    reg [Data_size-1:0] buf_out;
    reg buf_empty, buf_full;
    reg [Data_size-1:0] fifo_counter;
    reg [3:0] rd_ptr, wr_ptr;
    reg [Data_size-1:0] buf_mem[63:0];

    always @(fifo_counter) begin
        buf_empty= (fifo_counter==0);
        buf_full = (fifo_counter==64);
    end

    always @(posedge clk or posedge rst) begin
        if(rst)
            fifo_counter <= 0;
        else if( (!buf_full && wr_en) && (!buf_empty && rd_en) )
            fifo_counter <= fifo_counter;
        else if( !buf_full && wr_en )
            fifo_counter <= fifo_counter + 1;
        else if( !buf_empty && rd_en )
            fifo_counter <= fifo_counter - 1;
        else
            fifo_counter <= fifo_counter;
    end

    always @(posedge clk or posedge rst) begin
        if(rst)
            buf_out <= 0;
        else begin
            if(rd_en && !buf_empty)
                buf_out <= buf_mem[rd_ptr];
            else
                buf_out <= buf_out;
        end
    end

    always @(posedge clk) begin
        if(wr_en && !buf_full)
            buf_mem[wr_ptr] <= buf_in;
        else
            buf_mem[wr_ptr] <= buf_mem[wr_ptr];
    end

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            wr_ptr <= 0;
            rd_ptr <= 0;
        end
        else begin
            if(!buf_full && wr_en)
                wr_ptr <= wr_ptr + 1;
            else
                wr_ptr <= wr_ptr;
            if(!buf_empty && rd_en)
                rd_ptr <= rd_ptr + 1;
            else
                rd_ptr <= rd_ptr;
        end
    end

endmodule
```

Supplementary Figure 13: Weight FIFO.

```
module row_detector(rowsignal, weight_passtime);
    parameter BIT_WIDTH = 8;
    input [BIT_WIDTH-1:0] rowsignal;
    output reg [BIT_WIDTH-1:0] weight_passtime;

    always @(*) begin
        weight_passtime = 8'b00000011 - rowsignal;
    end

endmodule
```

Supplementary Figure 14: Row Detector.

```
module systolic_cell(
    clk, rst, row_en, col_en, weight_pass, weight_passtime,
    weightloading,
    activation_input, weight_input, psum_input,
    activation_output, sum_output, weight_output
);
    parameter BIT_WIDTH = 8;
    input [2*BIT_WIDTH-1:0] psum_input;
    input [BIT_WIDTH-1:0] activation_input;
    input [BIT_WIDTH-1:0] weight_input;
    input [BIT_WIDTH-1:0] weight_passtime;
    input rst;
    input clk;
    input row_en;
    input col_en;
    input weight_pass;
    output reg weightloading;
    output reg [BIT_WIDTH-1:0] activation_output;
    output reg [2*BIT_WIDTH-1:0] sum_output;
    output reg [BIT_WIDTH-1:0] weight_output;

    reg [BIT_WIDTH-1:0] weight_passtime_reg;
    reg [BIT_WIDTH-1:0] weight_reg;
    reg [2*BIT_WIDTH-1:0] mac_out;

    always @(*) begin
        mac_out = psum_input;
        weight_passtime_reg = weight_passtime;
        if (row_en && col_en) begin
            mac_out = (activation_input * weight_reg) + psum_input;
            if (mac_out > 16'hFFFF) begin
                mac_out = 16'hFFFF;
            end
        end
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            activation_output <= 1'b0;
            sum_output <= 1'b0;
            weight_reg <= 1'b0;
            weight_output <= 1'b0;
        end else if (weight_pass) begin
            if (weight_passtime_reg > 0) begin
                weightloading <= 1'b1;
                weight_output <= weight_input;
                weight_passtime_reg <= weight_passtime_reg - 1;
            end else begin
                weightloading <= 1'b0;
                weight_reg <= weight_input;
                weight_output <= weight_input;
            end
        end else begin
            if (row_en && col_en) begin
                activation_output <= activation_input;
                sum_output <= mac_out;
            end else begin
                sum_output <= psum_input;
            end
        end
    end

endmodule
```

Supplementary Figure 15: Systolic Cell.

```
module MMU4x4(a1, a2, a3, a4, w1, w2, w3, w4, s_in1, s_in2, s_in3, s_in4, o1, o2, o3, o4, rst, clk,
w_pass, weightload_complete, r1_en, r2_en, r3_en, r4_en, c1_en, c2_en, c3_en, c4_en, rowsignal1,
rowsignal2, rowsignal3, rowsignal4);
    parameter BIT_WIDTH = 8;
    input [BIT_WIDTH-1:0] a1, a2, a3, a4;
    input [BIT_WIDTH-1:0] w1, w2, w3, w4;
    input [2*BIT_WIDTH-1:0] s_in1, s_in2, s_in3, s_in4;
    input r1_en, r2_en, r3_en, r4_en, c1_en, c2_en, c3_en, c4_en;
    input [BIT_WIDTH-1:0] rowsignal1, rowsignal2, rowsignal3, rowsignal4;
    input clk, rst, w_pass;
    output wire [2*BIT_WIDTH-1:0] o1, o2, o3, o4;
    output reg weightload_complete;

    wire [2*BIT_WIDTH-1:0] s_11_21, s_21_31, s_31_41, s_12_22, s_22_32, s_32_42, s_13_23,
s_23_33, s_33_43, s_14_24, s_24_34, s_34_44;
    wire [BIT_WIDTH-1:0] w_11_21, w_21_31, w_31_41, w_12_22, w_22_32, w_32_42,
w_13_23, w_23_33, w_33_43, w_14_24, w_24_34, w_34_44;
    wire [BIT_WIDTH-1:0] a_11_12, a_12_13, a_13_14, a_21_22, a_22_23, a_23_24, a_31_32,
a_32_33, a_33_34, a_41_42, a_42_43, a_43_44;
    wire weightloading1, weightloading2, weightloading3, weightloading4;
    wire [BIT_WIDTH-1:0] weight_passtime1, weight_passtime2, weight_passtime3,
weight_passtime4;

    always @(*) begin
        if (weightloading1 || weightloading2 || weightloading3 || weightloading4) begin
            weightload_complete <= 1'b0;
        end else begin
            weightload_complete <= 1'b1;
            end
        end

    //---------------------ROW1----------------------//
    row_detector u1(
    .rowsignal(rowsignal1), .weight_passtime(weight_passtime1)
    );

    //---------------------ROW2----------------------//
    row_detector u2(
    .rowsignal(rowsignal2), .weight_passtime(weight_passtime2)
    );

    //---------------------ROW3----------------------//
    row_detector u3(
    .rowsignal(rowsignal3), .weight_passtime(weight_passtime3)
    );

    //---------------------ROW4----------------------//
    row_detector u4(
    .rowsignal(rowsignal4), .weight_passtime(weight_passtime4)
    );

    //---------------------COL 1----------------------//
    systolic_cell pe11(
    .clk(clk), .rst(rst), .row_en(r1_en), .col_en(c1_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime1), .weightloading(weightloading1),
    .activation_input(a1), .weight_input(w1), .psum_input(s_in1),
    .activation_output(a_11_12), .sum_output(s_11_21), .weight_output(w_11_21)
    );

    systolic_cell pe21(
    .clk(clk), .rst(rst), .row_en(r2_en), .col_en(c1_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime2), .weightloading(weightloading2),
    .activation_input(a2), .weight_input(w_11_21), .psum_input(s_11_21),
    .activation_output(a_21_22), .sum_output(s_21_31), .weight_output(w_21_31)
    );

    systolic_cell pe31(
    .clk(clk), .rst(rst), .row_en(r3_en), .col_en(c1_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime3), .weightloading(weightloading3),
    .activation_input(a3), .weight_input(w_21_31), .psum_input(s_21_31),
    .activation_output(a_31_32), .sum_output(s_31_41), .weight_output(w_31_41)
    );

    systolic_cell pe41(
    .clk(clk), .rst(rst), .row_en(r4_en), .col_en(c1_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime4), .weightloading(weightloading4),
    .activation_input(a4), .weight_input(w_31_41), .psum_input(s_31_41),
    .activation_output(a_41_42), .sum_output(o1), .weight_output()
    );
```

```
//---------------------COL 2----------------------//
    systolic_cell pe12(
    .clk(clk), .rst(rst), .row_en(r1_en), .col_en(c2_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime1), .weightloading(),
    .activation_input(a_11_12), .weight_input(w2), .psum_input(s_in2),
    .activation_output(a_12_13), .sum_output(s_12_22), .weight_output(w_12_22)
    );

    systolic_cell pe22(
    .clk(clk), .rst(rst), .row_en(r2_en), .col_en(c2_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime2), .weightloading(),
    .activation_input(a_21_22), .weight_input(w_12_22), .psum_input(s_12_22),
    .activation_output(a_22_23), .sum_output(s_22_32), .weight_output(w_22_32)
    );

    systolic_cell pe32(
    .clk(clk), .rst(rst), .row_en(r3_en), .col_en(c2_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime3), .weightloading(),
    .activation_input(a_31_32), .weight_input(w_22_32), .psum_input(s_22_32),
    .activation_output(a_32_33), .sum_output(s_32_42), .weight_output(w_32_42)
    );

    systolic_cell pe42(
    .clk(clk), .rst(rst), .row_en(r4_en), .col_en(c2_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime4), .weightloading(),
    .activation_input(a_41_42), .weight_input(w_32_42), .psum_input(s_32_42),
    .activation_output(a_42_43), .sum_output(o2), .weight_output()
    );

    //---------------------COL 3----------------------//
    systolic_cell pe13(
    .clk(clk), .rst(rst), .row_en(r1_en), .col_en(c3_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime1), .weightloading(),
    .activation_input(a_12_13), .weight_input(w3), .psum_input(s_in3),
    .activation_output(a_13_14), .sum_output(s_13_23), .weight_output(w_13_23)
    );

    systolic_cell pe23(
    .clk(clk), .rst(rst), .row_en(r2_en), .col_en(c3_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime2), .weightloading(),
    .activation_input(a_22_23), .weight_input(w_13_23), .psum_input(s_13_23),
    .activation_output(a_23_24), .sum_output(s_23_33), .weight_output(w_23_33)
    );

    systolic_cell pe33(
    .clk(clk), .rst(rst), .row_en(r3_en), .col_en(c3_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime3), .weightloading(),
    .activation_input(a_32_33), .weight_input(w_23_33), .psum_input(s_23_33),
    .activation_output(a_33_34), .sum_output(s_33_43), .weight_output(w_33_43)
    );

    systolic_cell pe43(
    .clk(clk), .rst(rst), .row_en(r4_en), .col_en(c3_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime4), .weightloading(),
    .activation_input(a_42_43), .weight_input(w_33_43), .psum_input(s_33_43),
    .activation_output(a_43_44), .sum_output(o3), .weight_output()
    );

    //---------------------COL 4----------------------//
    systolic_cell pe14(
    .clk(clk), .rst(rst), .row_en(r1_en), .col_en(c4_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime1), .weightloading(),
    .activation_input(a_13_14), .weight_input(w4), .psum_input(s_in4),
    .activation_output(), .sum_output(s_14_24), .weight_output(w_14_24)
    );

    systolic_cell pe24(
    .clk(clk), .rst(rst), .row_en(r2_en), .col_en(c4_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime2), .weightloading(),
    .activation_input(a_23_24), .weight_input(w_14_24), .psum_input(s_14_24),
    .activation_output(), .sum_output(s_24_34), .weight_output(w_24_34)
    );

    systolic_cell pe34(
    .clk(clk), .rst(rst), .row_en(r3_en), .col_en(c4_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime3), .weightloading(),
    .activation_input(a_33_34), .weight_input(w_24_34), .psum_input(s_24_34),
    .activation_output(), .sum_output(s_34_44), .weight_output(w_34_44)
    );

    systolic_cell pe44(
    .clk(clk), .rst(rst), .row_en(r4_en), .col_en(c4_en), .weight_pass(w_pass),
    .weight_passtime(weight_passtime4), .weightloading(),
    .activation_input(a_43_44), .weight_input(w_34_44), .psum_input(s_34_44),
    .activation_output(), .sum_output(o4), .weight_output()
    );

endmodule
```

Supplementary Figure 16: Matrix Multiplication Unit.

```
module unified_buffer(addr, clk, rd, wr, Host_rd, Host_wr, Host_datain, Host_dataout, counter_rst,
data_in1, data_in2, data_in3, data_in4, data_out1, data_out2, data_out3, data_out4, data_out5,
data_out6, data_out7, data_out8, data_out9, data_out10, data_out11, data_out12, data_out13,
data_out14, data_out15, data_out16);
    parameter BIT_WIDTH = 8;
    parameter address_size = 4;
    parameter word_size = 8;
    parameter memory_size = 16;
    input [word_size-1:0] data_in1, data_in2, data_in3, data_in4;
    output reg [word_size-1:0] data_out1, data_out2, data_out3, data_out4, data_out5, data_out6,
data_out7, data_out8, data_out9, data_out10, data_out11, data_out12, data_out13, data_out14,
data_out15, data_out16;
    input [address_size-1:0] addr;
    input counter_rst;
    input clk;
    input rd, wr;
    input Host_rd, Host_wr;
    input [word_size-1:0] Host_datain;
    output reg [word_size-1:0] Host_dataout;
    reg [word_size-1:0] memory [memory_size-1:0];
    reg [BIT_WIDTH-1:0] i;

    always @(posedge clk) begin
        if (counter_rst) begin
            i <= 0;
        end
        else begin
            i <= i + 1;
        end
    end

    always @(posedge clk) begin
        if (wr && !rd) begin
            case (i)
                3'd0: begin
                        memory[0] <= data_in1;
                        memory[1] <= data_in2;
                        memory[2] <= data_in3;
                        memory[3] <= data_in4;
                        end
                3'd1: begin
                        memory[4] <= data_in1;
                        memory[5] <= data_in2;
                        memory[6] <= data_in3;
                        memory[7] <= data_in4;
                        end
                3'd2: begin
                        memory[8] <= data_in1;
                        memory[9] <= data_in2;
                        memory[10] <= data_in3;
                        memory[11] <= data_in4;
                        end
                3'd3: begin
                        memory[12] <= data_in1;
                        memory[13] <= data_in2;
                        memory[14] <= data_in3;
                        memory[15] <= data_in4;
                        end
                default ;
            endcase
        end
        else if (rd && !wr) begin
            data_out1 <= memory[0];
            data_out2 <= memory[1];
            data_out3 <= memory[2];
            data_out4 <= memory[3];
            data_out5 <= memory[4];
            data_out6 <= memory[5];
            data_out7 <= memory[6];
            data_out8 <= memory[7];
            data_out9 <= memory[8];
            data_out10 <= memory[9];
            data_out11 <= memory[10];
            data_out12 <= memory[11];
            data_out13 <= memory[12];
            data_out14 <= memory[13];
            data_out15 <= memory[14];
            data_out16 <= memory[15];
        end else if (Host_wr) begin
            memory[addr] <= Host_datain;
        end else if (Host_rd) begin
            Host_dataout <= memory[addr];
        end
    end

endmodule
```

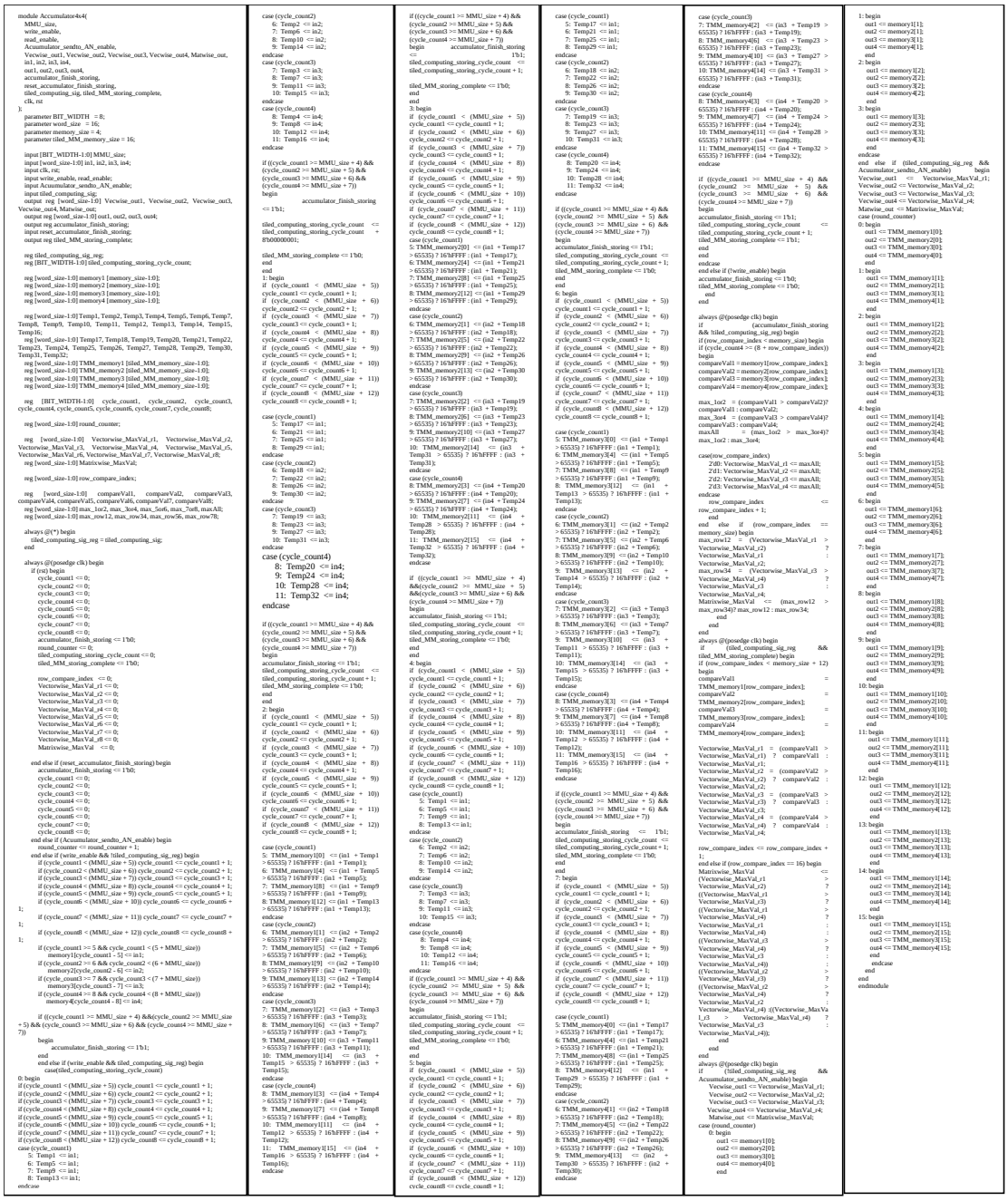
Supplementary Figure 17: Unified Buffer.

```
module sds4x4(clk, counter_rst, sds_work, a0, a1, a2, a3, b0, b1, b2, b3, c0, c1, c2, c3, d0,
d1, d2, d3, output1, output2, output3, output4);
    parameter BIT_WIDTH = 8;
    input clk;
    input counter_rst;
    input sds_work;
    input [BIT_WIDTH-1:0] a0, a1, a2, a3, b0, b1, b2, b3, c0, c1, c2, c3, d0, d1, d2, d3;
    output reg [BIT_WIDTH-1:0] output1;
    output reg [BIT_WIDTH-1:0] output2;
    output reg [BIT_WIDTH-1:0] output3;
    output reg [BIT_WIDTH-1:0] output4;
    reg [BIT_WIDTH-1:0] i;

    always @(posedge clk) begin
        if (counter_rst) begin
            i <= 0;
        end
        else if (sds_work) begin
            i <= i + 1;
        end
    end

    always @(posedge clk) begin
        if (sds_work) begin
            case (i)
                3'd0: begin
                    output1 <= a0;
                end
                3'd1: begin
                    output1 <= b0;
                    output2 <= a1;
                    output3 <= 8'b0;
                    output4 <= 8'b0;
                end
                3'd2: begin
                    output1 <= c0;
                    output2 <= b1;
                    output3 <= a2;
                    output4 <= 8'b0;
                end
                3'd3: begin
                    output1 <= d0;
                    output2 <= c1;
                    output3 <= b2;
                    output4 <= a3;
                end
                3'd4: begin
                    output1 <= 8'b0;
                    output2 <= d1;
                    output3 <= c2;
                    output4 <= b3;
                end
                3'd5: begin
                    output1 <= 8'b0;
                    output2 <= 8'b0;
                    output3 <= d2;
                    output4 <= c3;
                end
                3'd6: begin
                    output1 <= 8'b0;
                    output2 <= 8'b0;
                    output3 <= 8'b0;
                    output4 <= d3;
                end
                default: begin
                    output1 <= 8'b0;
                    output2 <= 8'b0;
                    output3 <= 8'b0;
                    output4 <= 8'b0;
                end
            endcase
        end
    end
endmodule
```

Supplementary Figure 18: Systolic Data Setup Unit.

```
module Activation_FIFO(clk, rst, buf_in, buf_out, wr_en, rd_en, buf_empty,
buf_full, fifo_counter);

    parameter Data_size = 8;
    input clk, rst, wr_en, rd_en;
    input [7:0] buf_in;
    output [7:0] buf_out;
    output buf_empty, buf_full;
    output [7:0] fifo_counter;

    reg [Data_size-1:0] buf_out;
    reg buf_empty, buf_full;
    reg [Data_size-1:0] fifo_counter;
    reg [3:0] rd_ptr, wr_ptr;
    reg [Data_size-1:0] buf_mem[63:0];

    always @(fifo_counter) begin
        buf_empty= (fifo_counter==0);
        buf_full = (fifo_counter==64);
    end

    always @(posedge clk or posedge rst) begin
        if(rst)
            fifo_counter <= 0;
        else if( (!buf_full && wr_en) && (!buf_empty && rd_en) )
            fifo_counter <= fifo_counter;
        else if( !buf_full && wr_en )
            fifo_counter <= fifo_counter + 1;
        else if( !buf_empty && rd_en )
            fifo_counter <= fifo_counter - 1;
        else
            fifo_counter <= fifo_counter;
    end

    always @(posedge clk or posedge rst) begin
        if(rst)
            buf_out <= 0;
        else begin
            if(rd_en && !buf_empty)
                buf_out <= buf_mem[rd_ptr];
            else
                buf_out <= buf_out;
        end
    end

    always @(posedge clk) begin
        if(wr_en && !buf_full)
            buf_mem[wr_ptr] <= buf_in;
        else
            buf_mem[wr_ptr] <= buf_mem[wr_ptr];
    end

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            wr_ptr <= 0;
            rd_ptr <= 0;
        end
        else begin
            if(!buf_full && wr_en) begin
                wr_ptr <= wr_ptr + 1;
            end
            else
                wr_ptr <= wr_ptr;
            if(!buf_empty && rd_en) begin
                rd_ptr <= rd_ptr + 1;
            end
            else
                rd_ptr <= rd_ptr;
        end
    end

endmodule
```

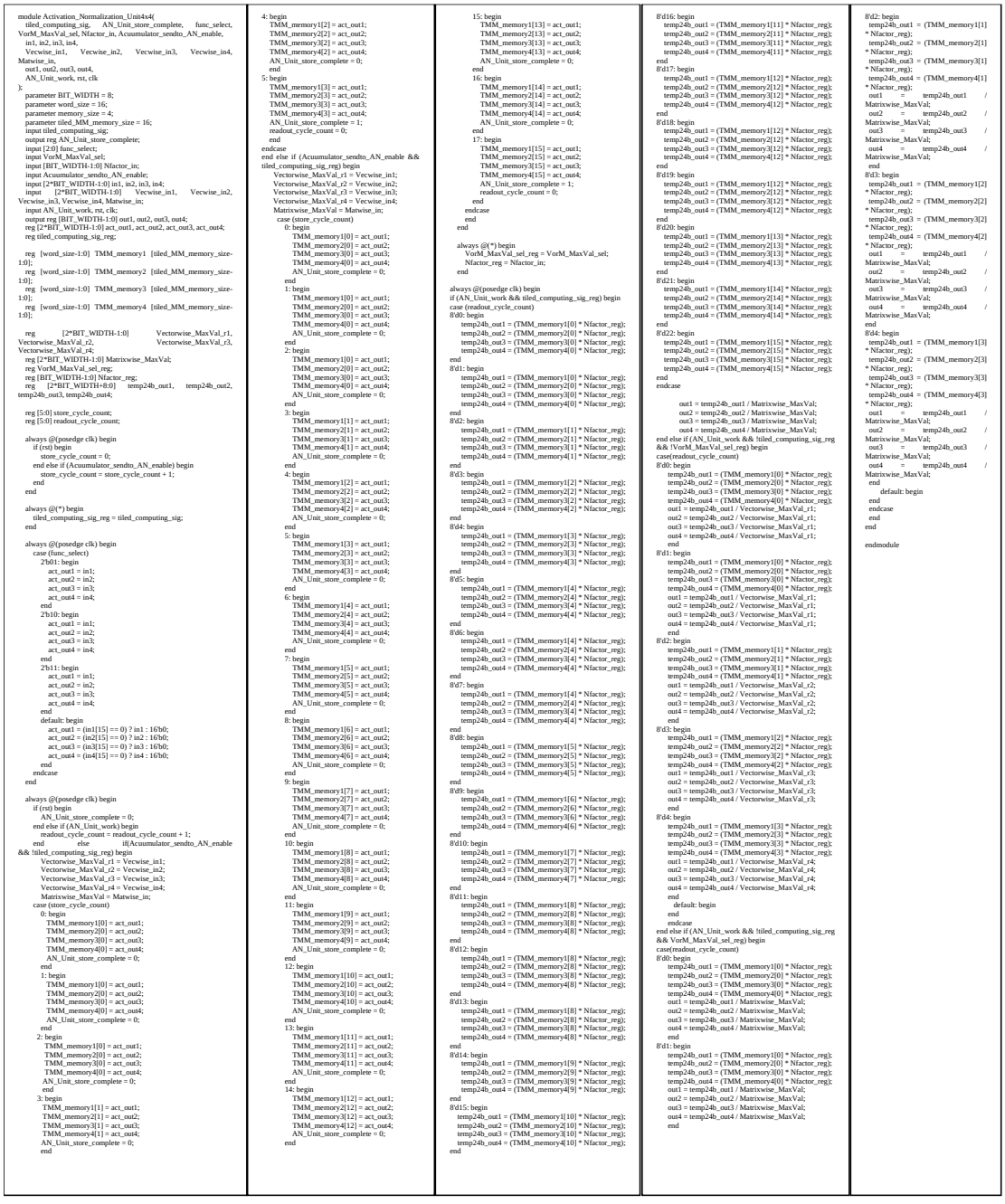
Supplementary Figure 19: Activation FIFO.



```
module Accumulator4x4(
    MMU_size,
    write_enable,
    read_enable,
    Accumulator_sendto_AN_enable,
    Vecwise_out1, Vecwise_out2, Vecwise_out3, Vecwise_out4, Matwise_out,
    in1, in2, in3, in4,
    out1, out2, out3, out4,
    accumulator_finish_storing,
    reset_accumulator_finish_storing,
    tiled_computing_sig, tiled_MM_storing_complete,
    clk, rst
);
    parameter BIT_WIDTH  = 8;
    parameter word_size  = 16;
    parameter memory_size = 4;
    parameter tiled_MM_memory_size = 16;

    input [BIT_WIDTH-1:0] MMU_size;
    input [word_size-1:0] in1, in2, in3, in4;
    input clk, rst;
    input write_enable, read_enable;
    input Accumulator_sendto_AN_enable;
    input tiled_computing_sig;
    output reg [word_size-1:0]  Vecwise_out1,  Vecwise_out2,  Vecwise_out3,
Vecwise_out4, Matwise_out;
    output reg [word_size-1:0] out1, out2, out3, out4;
    output reg accumulator_finish_storing;
    input reset_accumulator_finish_storing;
    output reg tiled_MM_storing_complete;

    reg tiled_computing_sig_reg;
    reg [BIT_WIDTH-1:0] tiled_computing_storing_cycle_count;

    reg [word_size-1:0] memory1 [memory_size-1:0];
    reg [word_size-1:0] memory2 [memory_size-1:0];
    reg [word_size-1:0] memory3 [memory_size-1:0];
    reg [word_size-1:0] memory4 [memory_size-1:0];

    reg [word_size-1:0] Temp1, Temp2, Temp3, Temp4, Temp5, Temp6, Temp7,
Temp8, Temp9, Temp10, Temp11, Temp12, Temp13, Temp14, Temp15,
Temp16;
    reg [word_size-1:0] Temp17, Temp18, Temp19, Temp20, Temp21, Temp22,
Temp23, Temp24, Temp25, Temp26, Temp27, Temp28, Temp29, Temp30,
Temp31, Temp32;
    reg [word_size-1:0] TMM_memory1 [tiled_MM_memory_size-1:0];
    reg [word_size-1:0] TMM_memory2 [tiled_MM_memory_size-1:0];
    reg [word_size-1:0] TMM_memory3 [tiled_MM_memory_size-1:0];
    reg [word_size-1:0] TMM_memory4 [tiled_MM_memory_size-1:0];

    reg  [BIT_WIDTH-1:0]  cycle_count1,  cycle_count2,  cycle_count3,
cycle_count4, cycle_count5, cycle_count6, cycle_count7, cycle_count8;

    reg [word_size-1:0] round_counter;

    reg  [word_size-1:0]  VectorwiseMaxVal_r1,  VectorwiseMaxVal_r2,
VectorwiseMaxVal_r3,  VectorwiseMaxVal_r4,  VectorwiseMaxVal_r5,
VectorwiseMaxVal_r6, VectorwiseMaxVal_r7, VectorwiseMaxVal_r8;
    reg [word_size-1:0] MatrixwiseMaxVal;

    reg [word_size-1:0] row_compare_index;

    reg  [word_size-1:0]  compareVal1,  compareVal2,  compareVal3,
compareVal4, compareVal5, compareVal6, compareVal7, compareVal8;
    reg [word_size-1:0] max_1or2, max_3or4, max_5or6, max_7or8, maxAll;
    reg [word_size-1:0] max_row12, max_row34, max_row56, max_row78;

    always @(*) begin
        tiled_computing_sig_reg = tiled_computing_sig;
    end

    always @(posedge clk) begin
        if (rst) begin
            cycle_count1 <= 0;
            cycle_count2 <= 0;
            cycle_count3 <= 0;
            cycle_count4 <= 0;
            cycle_count5 <= 0;
            cycle_count6 <= 0;
            cycle_count7 <= 0;
            cycle_count8 <= 0;
            accumulator_finish_storing <= 1'b0;
            round_counter <= 0;
            tiled_computing_storing_cycle_count <= 0;
            tiled_MM_storing_complete <= 1'b0;

            row_compare_index  <= 0;
            VectorwiseMaxVal_r1 <= 0;
            VectorwiseMaxVal_r2 <= 0;
            VectorwiseMaxVal_r3 <= 0;
            VectorwiseMaxVal_r4 <= 0;
            VectorwiseMaxVal_r5 <= 0;
            VectorwiseMaxVal_r6 <= 0;
            VectorwiseMaxVal_r7 <= 0;
            VectorwiseMaxVal_r8 <= 0;
            MatrixwiseMaxVal   <= 0;

        end else if (reset_accumulator_finish_storing) begin
            accumulator_finish_storing <= 1'b0;
            cycle_count1 <= 0;
            cycle_count2 <= 0;
            cycle_count3 <= 0;
            cycle_count4 <= 0;
            cycle_count5 <= 0;
            cycle_count6 <= 0;
            cycle_count7 <= 0;
            cycle_count8 <= 0;
        end else if (Accumulator_sendto_AN_enable) begin
            round_counter <= round_counter + 1;
        end else if (write_enable && tiled_computing_sig_reg) begin
            if (cycle_count1 < (MMU_size + 5)) cycle_count1 <= cycle_count1 + 1;
            if (cycle_count2 < (MMU_size + 6)) cycle_count2 <= cycle_count2 + 1;
            if (cycle_count3 < (MMU_size + 7)) cycle_count3 <= cycle_count3 + 1;
            if (cycle_count4 < (MMU_size + 8)) cycle_count4 <= cycle_count4 + 1;
            if (cycle_count5 < (MMU_size + 9)) cycle_count5 <= cycle_count5 + 1;
            if (cycle_count6 < (MMU_size + 10)) cycle_count6 <= cycle_count6 +
1;

            if (cycle_count7 < (MMU_size + 11)) cycle_count7 <= cycle_count7 +
1;

            if (cycle_count8 < (MMU_size + 12)) cycle_count8 <= cycle_count8 +
1;

            if (cycle_count1 >= 5 && cycle_count1 < (5 + MMU_size))
                memory1[cycle_count1 - 5] <= in1;
            if (cycle_count2 >= 6 && cycle_count2 < (6 + MMU_size))
                memory2[cycle_count2 - 6] <= in2;
            if (cycle_count3 >= 7 && cycle_count3 < (7 + MMU_size))
                memory3[cycle_count3 - 7] <= in3;
            if (cycle_count4 >= 8 && cycle_count4 < (8 + MMU_size))
                memory4[cycle_count4 - 8] <= in4;

            if ((cycle_count1 >= MMU_size + 4) &&(cycle_count2 >= MMU_size
+ 5) && (cycle_count3 >= MMU_size + 6) && (cycle_count4 >= MMU_size +
7))
                begin
                    accumulator_finish_storing <= 1'b1;
                end
            end else if (write_enable && tiled_computing_sig_reg) begin
                case(tiled_computing_storing_cycle_count)

0: begin
if (cycle_count1 < (MMU_size + 5)) cycle_count1 <= cycle_count1 + 1;
if (cycle_count2 < (MMU_size + 6)) cycle_count2 <= cycle_count2 + 1;
if (cycle_count3 < (MMU_size + 7)) cycle_count3 <= cycle_count3 + 1;
if (cycle_count4 < (MMU_size + 8)) cycle_count4 <= cycle_count4 + 1;
if (cycle_count5 < (MMU_size + 9)) cycle_count5 <= cycle_count5 + 1;
if (cycle_count6 < (MMU_size + 10)) cycle_count6 <= cycle_count6 + 1;
if (cycle_count7 < (MMU_size + 11)) cycle_count7 <= cycle_count7 + 1;
if (cycle_count8 < (MMU_size + 12)) cycle_count8 <= cycle_count8 + 1;
case (cycle_count1)
    5: Temp1 <= in1;
    6: Temp5 <= in1;
    7: Temp9 <= in1;
    8: Temp13 <= in1;
endcase
case (cycle_count2)
    6: Temp2 <= in2;
    7: Temp6 <= in2;
    8: Temp10 <= in2;
    9: Temp14 <= in2;
endcase
case (cycle_count3)
    7: Temp3 <= in3;
    8: Temp7 <= in3;
    9: Temp11 <= in3;
    10: Temp15 <= in3;
endcase
case (cycle_count4)
    8: Temp4 <= in4;
    9: Temp8 <= in4;
    10: Temp12 <= in4;
    11: Temp16 <= in4;
endcase

if ((cycle_count1 >= MMU_size + 4) &&
(cycle_count2 >= MMU_size + 5) &&
(cycle_count3 >= MMU_size + 6) &&
(cycle_count4 >= MMU_size + 7))
begin
                    accumulator_finish_storing
<= 1'b1;

tiled_computing_storing_cycle_count  <=
tiled_computing_storing_cycle_count  +
8'b00000001;

tiled_MM_storing_complete <= 1'b0;
end
end
1: begin
if (cycle_count1 < (MMU_size + 5))
cycle_count1 <= cycle_count1 + 1;
if (cycle_count2 < (MMU_size + 6))
cycle_count2 <= cycle_count2 + 1;
if (cycle_count3 < (MMU_size + 7))
cycle_count3 <= cycle_count3 + 1;
if (cycle_count4 < (MMU_size + 8))
cycle_count4 <= cycle_count4 + 1;
if (cycle_count5 < (MMU_size + 9))
cycle_count5 <= cycle_count5 + 1;
if (cycle_count6 < (MMU_size + 10))
cycle_count6 <= cycle_count6 + 1;
if (cycle_count7 < (MMU_size + 11))
cycle_count7 <= cycle_count7 + 1;
if (cycle_count8 < (MMU_size + 12))
cycle_count8 <= cycle_count8 + 1;

case (cycle_count1)
    5: Temp17 <= in1;
    6: Temp21 <= in1;
    7: Temp25 <= in1;
    8: Temp29 <= in1;
endcase
case (cycle_count2)
    6: Temp18 <= in2;
    7: Temp22 <= in2;
    8: Temp26 <= in2;
    9: Temp30 <= in2;
endcase
case (cycle_count3)
    7: Temp19 <= in3;
    8: Temp23 <= in3;
    9: Temp27 <= in3;
    10: Temp31 <= in3;
endcase
case (cycle_count4)
    8: Temp20 <= in4;
    9: Temp24 <= in4;
    10: Temp28 <= in4;
    11: Temp32 <= in4;
endcase

if ((cycle_count1 >= MMU_size + 4) &&
(cycle_count2 >= MMU_size + 5) &&
(cycle_count3 >= MMU_size + 6) &&
(cycle_count4 >= MMU_size + 7))
begin
accumulator_finish_storing <= 1'b1;
tiled_computing_storing_cycle_count  <=
tiled_computing_storing_cycle_count + 1;
tiled_MM_storing_complete <= 1'b0;
end
2: begin
if (cycle_count1 < (MMU_size + 5))
cycle_count1 <= cycle_count1 + 1;
if (cycle_count2 < (MMU_size + 6))
cycle_count2 <= cycle_count2 + 1;
if (cycle_count3 < (MMU_size + 7))
cycle_count3 <= cycle_count3 + 1;
if (cycle_count4 < (MMU_size + 8))
cycle_count4 <= cycle_count4 + 1;
if (cycle_count5 < (MMU_size + 9))
cycle_count5 <= cycle_count5 + 1;
if (cycle_count6 < (MMU_size + 10))
cycle_count6 <= cycle_count6 + 1;
if (cycle_count7 < (MMU_size + 11))
cycle_count7 <= cycle_count7 + 1;
if (cycle_count8 < (MMU_size + 12))
cycle_count8 <= cycle_count8 + 1;

case (cycle_count1)
5: TMM_memory1[0]  <= (in1  + Temp1
> 65535) ? 16'hFFFF : (in1  + Temp1);
6: TMM_memory1[4]  <= (in1  + Temp5
> 65535) ? 16'hFFFF : (in1  + Temp5);
7: TMM_memory1[8]  <= (in1  + Temp9
> 65535) ? 16'hFFFF : (in1  + Temp9);
8: TMM_memory1[12] <= (in1  + Temp13
> 65535) ? 16'hFFFF : (in1  + Temp13);
endcase
case (cycle_count2)
6: TMM_memory1[1]  <= (in2  + Temp2
> 65535) ? 16'hFFFF : (in2  + Temp2);
7: TMM_memory1[5]  <= (in2  + Temp6
> 65535) ? 16'hFFFF : (in2  + Temp6);
8: TMM_memory1[9]  <= (in2  + Temp10
> 65535) ? 16'hFFFF : (in2  + Temp10);
9: TMM_memory1[13] <= (in2  + Temp14
> 65535) ? 16'hFFFF : (in2  + Temp14);
endcase
case (cycle_count3)
7: TMM_memory1[2]  <= (in3  + Temp3
> 65535) ? 16'hFFFF : (in3  + Temp3);
8: TMM_memory1[6]  <= (in3  + Temp7
> 65535) ? 16'hFFFF : (in3  + Temp7);
9: TMM_memory1[10] <= (in3  + Temp11
> 65535) ? 16'hFFFF : (in3  + Temp11);
10: TMM_memory1[14]  <= (in3  +
Temp15  > 65535) ? 16'hFFFF : (in3  +
Temp15);
endcase
case (cycle_count4)
8: TMM_memory1[3]  <= (in4  + Temp4
> 65535) ? 16'hFFFF : (in4  + Temp4);
9: TMM_memory1[7]  <= (in4  + Temp8
> 65535) ? 16'hFFFF : (in4  + Temp8);
10: TMM_memory1[11]  <= (in4  +
Temp12  > 65535) ? 16'hFFFF : (in4  +
Temp12);
11: TMM_memory1[15]  <= (in4  +
Temp16  > 65535) ? 16'hFFFF : (in4  +
Temp16);
endcase

if ((cycle_count1 >= MMU_size + 4) &&
(cycle_count2 >= MMU_size + 5) &&
(cycle_count3 >= MMU_size + 6) &&
(cycle_count4 >= MMU_size + 7))
begin
                    accumulator_finish_storing
<= 1'b1;
tiled_computing_storing_cycle_count  <=
tiled_computing_storing_cycle_count + 1;

tiled_MM_storing_complete <= 1'b0;
end
end
3: begin
if (cycle_count1 < (MMU_size + 5))
cycle_count1 <= cycle_count1 + 1;
if (cycle_count2 < (MMU_size + 6))
cycle_count2 <= cycle_count2 + 1;
if (cycle_count3 < (MMU_size + 7))
cycle_count3 <= cycle_count3 + 1;
if (cycle_count4 < (MMU_size + 8))
cycle_count4 <= cycle_count4 + 1;
if (cycle_count5 < (MMU_size + 9))
cycle_count5 <= cycle_count5 + 1;
if (cycle_count6 < (MMU_size + 10))
cycle_count6 <= cycle_count6 + 1;
if (cycle_count7 < (MMU_size + 11))
cycle_count7 <= cycle_count7 + 1;
if (cycle_count8 < (MMU_size + 12))
cycle_count8 <= cycle_count8 + 1;

case (cycle_count1)
5: TMM_memory1[0]  <= (in1  + Temp17
> 65535) ? 16'hFFFF : (in1  + Temp17);
6: TMM_memory1[4]  <= (in1  + Temp21
> 65535) ? 16'hFFFF : (in1  + Temp21);
7: TMM_memory1[8]  <= (in1  + Temp25
> 65535) ? 16'hFFFF : (in1  + Temp25);
8: TMM_memory1[12] <= (in1  + Temp29
> 65535) ? 16'hFFFF : (in1  + Temp29);
endcase
case (cycle_count2)
6: TMM_memory1[1]  <= (in2  + Temp18
> 65535) ? 16'hFFFF : (in2  + Temp18);
7: TMM_memory1[5]  <= (in2  + Temp22
> 65535) ? 16'hFFFF : (in2  + Temp22);
8: TMM_memory1[9]  <= (in2  + Temp26
> 65535) ? 16'hFFFF : (in2  + Temp26);
9: TMM_memory1[13] <= (in2  + Temp30
> 65535) ? 16'hFFFF : (in2  + Temp30);
endcase
case (cycle_count3)
7: TMM_memory1[2]  <= (in3  + Temp19
> 65535) ? 16'hFFFF : (in3  + Temp19);
8: TMM_memory1[6]  <= (in3  + Temp23
> 65535) ? 16'hFFFF : (in3  + Temp23);
9: TMM_memory1[10] <= (in3  + Temp27
> 65535) ? 16'hFFFF : (in3  + Temp27);
10: TMM_memory1[14]  <= (in3  +
Temp31  > 65535) ? 16'hFFFF : (in3  +
Temp31);
endcase
case (cycle_count4)
8: TMM_memory1[3]  <= (in4  + Temp20
> 65535) ? 16'hFFFF : (in4  + Temp20);
9: TMM_memory1[7]  <= (in4  + Temp24
> 65535) ? 16'hFFFF : (in4  + Temp24);
10: TMM_memory1[11]  <= (in4  +
Temp28  > 65535) ? 16'hFFFF : (in4  +
Temp28);
11: TMM_memory1[15]  <= (in4  +
Temp32  > 65535) ? 16'hFFFF : (in4  +
Temp32);
endcase

if ((cycle_count1 >= MMU_size + 4)
&&(cycle_count2 >= MMU_size + 5)
&&(cycle_count3 >= MMU_size + 6) &&
(cycle_count4 >= MMU_size + 7))
begin
accumulator_finish_storing <= 1'b1;
tiled_computing_storing_cycle_count  <=
tiled_computing_storing_cycle_count + 1;
tiled_MM_storing_complete <= 1'b0;
end
4: begin
if (cycle_count1 < (MMU_size + 5))
cycle_count1 <= cycle_count1 + 1;
if (cycle_count2 < (MMU_size + 6))
cycle_count2 <= cycle_count2 + 1;
if (cycle_count3 < (MMU_size + 7))
cycle_count3 <= cycle_count3 + 1;
if (cycle_count4 < (MMU_size + 8))
cycle_count4 <= cycle_count4 + 1;
if (cycle_count5 < (MMU_size + 9))
cycle_count5 <= cycle_count5 + 1;
if (cycle_count6 < (MMU_size + 10))
cycle_count6 <= cycle_count6 + 1;
if (cycle_count7 < (MMU_size + 11))
cycle_count7 <= cycle_count7 + 1;
if (cycle_count8 < (MMU_size + 12))
cycle_count8 <= cycle_count8 + 1;

case (cycle_count1)
    5: Temp1  <= in1;
    6: Temp5  <= in1;
    7: Temp9  <= in1;
    8: Temp13 <= in1;
endcase
case (cycle_count2)
    6: Temp2  <= in2;
    7: Temp6  <= in2;
    8: Temp10 <= in2;
    9: Temp14 <= in2;
endcase
case (cycle_count3)
    7: Temp3  <= in3;
    8: Temp7  <= in3;
    9: Temp11 <= in3;
    10: Temp15 <= in3;
endcase
case (cycle_count4)
    8: Temp4  <= in4;
    9: Temp8  <= in4;
    10: Temp12 <= in4;
    11: Temp16 <= in4;
endcase
if ((cycle_count1 >= MMU_size + 4) &&
(cycle_count2 >= MMU_size + 5) &&
(cycle_count3 >= MMU_size + 6) &&
(cycle_count4 >= MMU_size + 7))
begin
accumulator_finish_storing <= 1'b1;
tiled_computing_storing_cycle_count  <=
tiled_computing_storing_cycle_count + 1;
tiled_MM_storing_complete <= 1'b0;
end
end
5: begin
if (cycle_count1 < (MMU_size + 5))
cycle_count1 <= cycle_count1 + 1;
if (cycle_count2 < (MMU_size + 6))
cycle_count2 <= cycle_count2 + 1;
if (cycle_count3 < (MMU_size + 7))
cycle_count3 <= cycle_count3 + 1;
if (cycle_count4 < (MMU_size + 8))
cycle_count4 <= cycle_count4 + 1;
if (cycle_count5 < (MMU_size + 9))
cycle_count5 <= cycle_count5 + 1;
if (cycle_count6 < (MMU_size + 10))
cycle_count6 <= cycle_count6 + 1;
if (cycle_count7 < (MMU_size + 11))
cycle_count7 <= cycle_count7 + 1;
if (cycle_count8 < (MMU_size + 12))
cycle_count8 <= cycle_count8 + 1;

case (cycle_count1)
    5: Temp17 <= in1;
    6: Temp21 <= in1;
    7: Temp25 <= in1;
    8: Temp29 <= in1;
endcase
case (cycle_count2)
    6: Temp18 <= in2;
    7: Temp22 <= in2;
    8: Temp26 <= in2;
    9: Temp30 <= in2;
endcase
case (cycle_count3)
    7: Temp19 <= in3;
    8: Temp23 <= in3;
    9: Temp27 <= in3;
    10: Temp31 <= in3;
endcase
case (cycle_count4)
    8: Temp20 <= in4;
    9: Temp24 <= in4;
    10: Temp28 <= in4;
    11: Temp32 <= in4;
endcase

if ((cycle_count1 >= MMU_size + 4) &&
(cycle_count2 >= MMU_size + 5) &&
(cycle_count3 >= MMU_size + 6) &&
(cycle_count4 >= MMU_size + 7))
begin
accumulator_finish_storing <= 1'b1;
tiled_computing_storing_cycle_count  <=
tiled_computing_storing_cycle_count + 1;
tiled_MM_storing_complete <= 1'b0;
end
6: begin
if (cycle_count1 < (MMU_size + 5))
cycle_count1 <= cycle_count1 + 1;
if (cycle_count2 < (MMU_size + 6))
cycle_count2 <= cycle_count2 + 1;
if (cycle_count3 < (MMU_size + 7))
cycle_count3 <= cycle_count3 + 1;
if (cycle_count4 < (MMU_size + 8))
cycle_count4 <= cycle_count4 + 1;
if (cycle_count5 < (MMU_size + 9))
cycle_count5 <= cycle_count5 + 1;
if (cycle_count6 < (MMU_size + 10))
cycle_count6 <= cycle_count6 + 1;
if (cycle_count7 < (MMU_size + 11))
cycle_count7 <= cycle_count7 + 1;
if (cycle_count8 < (MMU_size + 12))
cycle_count8 <= cycle_count8 + 1;

case (cycle_count1)
5: TMM_memory1[0]  <= (in1  + Temp1
> 65535) ? 16'hFFFF : (in1  + Temp1);
6: TMM_memory1[4]  <= (in1  + Temp5
> 65535) ? 16'hFFFF : (in1  + Temp5);
7: TMM_memory1[8]  <= (in1  + Temp9
> 65535) ? 16'hFFFF : (in1  + Temp9);
8: TMM_memory1[12] <= (in1  +
Temp13  > 65535) ? 16'hFFFF : (in1  +
Temp13);
endcase
case (cycle_count2)
6: TMM_memory1[1]  <= (in2  + Temp2
> 65535) ? 16'hFFFF : (in2  + Temp2);
7: TMM_memory1[5]  <= (in2  + Temp6
> 65535) ? 16'hFFFF : (in2  + Temp6);
8: TMM_memory1[9]  <= (in2  + Temp10
> 65535) ? 16'hFFFF : (in2  + Temp10);
9: TMM_memory1[13] <= (in2  +
Temp14  > 65535) ? 16'hFFFF : (in2  +
Temp14);
endcase
case (cycle_count3)
7: TMM_memory1[2]  <= (in3  + Temp3
> 65535) ? 16'hFFFF : (in3  + Temp3);
8: TMM_memory1[6]  <= (in3  + Temp7
> 65535) ? 16'hFFFF : (in3  + Temp7);
9: TMM_memory1[10] <= (in3  +
Temp11  > 65535) ? 16'hFFFF : (in3  +
Temp11);
10: TMM_memory1[14]  <= (in3  +
Temp15  > 65535) ? 16'hFFFF : (in3  +
Temp15);
endcase
case (cycle_count4)
8: TMM_memory1[3]  <= (in4  + Temp4
> 65535) ? 16'hFFFF : (in4  + Temp4);
9: TMM_memory1[7]  <= (in4  + Temp8
> 65535) ? 16'hFFFF : (in4  + Temp8);
10: TMM_memory1[11]  <= (in4  +
Temp12  > 65535) ? 16'hFFFF : (in4  +
Temp12);
11: TMM_memory1[15]  <= (in4  +
Temp16  > 65535) ? 16'hFFFF : (in4  +
Temp16);
endcase

case (cycle_count3)
7: TMM_memory4[2]  <= (in3  + Temp19 >
65535) ? 16'hFFFF : (in3  + Temp19);
8: TMM_memory4[6]  <= (in3  + Temp23 >
65535) ? 16'hFFFF : (in3  + Temp23);
9: TMM_memory4[10] <= (in3  + Temp27 >
65535) ? 16'hFFFF : (in3  + Temp27);
10: TMM_memory4[14]  <= (in3  + Temp31 >
65535) ? 16'hFFFF : (in3  + Temp31);
endcase
case (cycle_count4)
8: TMM_memory4[3]  <= (in4  + Temp20 >
65535) ? 16'hFFFF : (in4  + Temp20);
9: TMM_memory4[7]  <= (in4  + Temp24 >
65535) ? 16'hFFFF : (in4  + Temp24);
10: TMM_memory4[11]  <= (in4  + Temp28 >
65535) ? 16'hFFFF : (in4  + Temp28);
11: TMM_memory4[15]  <= (in4  + Temp32 >
65535) ? 16'hFFFF : (in4  + Temp32);
endcase

if ((cycle_count1 >= MMU_size + 4) &&
(cycle_count2 >= MMU_size + 5) &&
(cycle_count3 >= MMU_size + 6) &&
(cycle_count4 >= MMU_size + 7))
begin
accumulator_finish_storing <= 1'b1;
tiled_computing_storing_cycle_count <=
tiled_computing_storing_cycle_count + 1;
tiled_MM_storing_complete <= 1'b1;
end
else if (!write_enable)
accumulator_finish_storing <= 1'b0;
tiled_MM_storing_complete <= 1'b0;
    end
end

always @(posedge clk) begin
if (accumulator_finish_storing
&& !tiled_computing_sig_reg) begin
if (row_compare_index < memory_size)
if (cycle_count1 >= (8 + row_compare_index))
begin
compareVal1 = memory1[row_compare_index];
compareVal2 = memory2[row_compare_index];
compareVal3 = memory3[row_compare_index];
compareVal4 = memory4[row_compare_index];

max_1or2 = (compareVal1 > compareVal2)?
compareVal1 : compareVal2;
max_3or4 = (compareVal3 > compareVal4)?
compareVal3 : compareVal4;
maxAll      = (max_1or2 > max_3or4)?
max_1or2 : max_3or4;

case(row_compare_index)
    2'd0: VectorwiseMaxVal_r1 <= maxAll;
    2'd1: VectorwiseMaxVal_r2 <= maxAll;
    2'd2: VectorwiseMaxVal_r3 <= maxAll;
    2'd3: VectorwiseMaxVal_r4 <= maxAll;
endcase
    row_compare_index           <=
row_compare_index + 1;
    end
    end else if (row_compare_index  ==
memory_size) begin
max_row12  = (VectorwiseMaxVal_r1 >
VectorwiseMaxVal_r2)                ?
VectorwiseMaxVal_r1                 :
VectorwiseMaxVal_r2;
max_row34  = (VectorwiseMaxVal_r3 >
VectorwiseMaxVal_r4)                ?
VectorwiseMaxVal_r3                 :
VectorwiseMaxVal_r4;
MatrixwiseMaxVal   <=  (max_row12  >
max_row34)? max_row12 : max_row34;
        end
    end
end
always @(posedge clk) begin
    if    (tiled_computing_sig_reg
&& !tiled_MM_storing_complete) begin
if (row_compare_index < memory_size + 12)
begin
compareVal1                         =
TMM_memory1[row_compare_index];
compareVal2                         =
TMM_memory2[row_compare_index];
compareVal3                         =
TMM_memory3[row_compare_index];
compareVal4                         =
TMM_memory4[row_compare_index];

VectorwiseMaxVal_r1  =  (compareVal1  >
VectorwiseMaxVal_r1)  ?  compareVal1  :
VectorwiseMaxVal_r1;
VectorwiseMaxVal_r2  =  (compareVal2  >
VectorwiseMaxVal_r2)  ?  compareVal2  :
VectorwiseMaxVal_r2;
VectorwiseMaxVal_r3  =  (compareVal3  >
VectorwiseMaxVal_r3)  ?  compareVal3  :
VectorwiseMaxVal_r3;
VectorwiseMaxVal_r4  =  (compareVal4  >
VectorwiseMaxVal_r4)  ?  compareVal4  :
VectorwiseMaxVal_r4;

row_compare_index <= row_compare_index +
1;
end else if (row_compare_index == 16) begin
MatrixwiseMaxVal                    <=
((VectorwiseMaxVal_r1               >
VectorwiseMaxVal_r2)                ?
((VectorwiseMaxVal_r1               >
VectorwiseMaxVal_r3)                ?
((VectorwiseMaxVal_r1               >
VectorwiseMaxVal_r4)                ?
VectorwiseMaxVal_r1                 :
VectorwiseMaxVal_r4)                :
((VectorwiseMaxVal_r3               >
VectorwiseMaxVal_r4)                ?
VectorwiseMaxVal_r3                 :
VectorwiseMaxVal_r4))               :
((VectorwiseMaxVal_r2               >
VectorwiseMaxVal_r3)                ?
((VectorwiseMaxVal_r2               >
VectorwiseMaxVal_r4)                ?
VectorwiseMaxVal_r2                 :
VectorwiseMaxVal_r4)                :
((VectorwiseMaxVal_r3               >
VectorwiseMaxVal_r4)                ?
VectorwiseMaxVal_r3                 :
VectorwiseMaxVal_r4));
        end
    end
end
endmodule
```

```
1: begin
    out1 <= memory1[1];
    out2 <= memory2[1];
    out3 <= memory3[1];
    out4 <= memory4[1];
  end
2: begin
    out1 <= memory1[2];
    out2 <= memory2[2];
    out3 <= memory3[2];
    out4 <= memory4[2];
  end
3: begin
    out1 <= memory1[3];
    out2 <= memory2[3];
    out3 <= memory3[3];
    out4 <= memory4[3];
  end
endcase
end else if (tiled_computing_sig_reg &&
Accumulator_sendto_AN_enable)  begin
Vecwise_out1    <=  Vectorwise_MaxVal_r1;
Vecwise_out2 <= Vectorwise_MaxVal_r2;
Vecwise_out3 <= Vectorwise_MaxVal_r3;
Vecwise_out4 <= Vectorwise_MaxVal_r4;
Matwise_out  <= Matrixwise_MaxVal;
case (round_counter)
0: begin
    out1 <= TMM_memory1[0];
    out2 <= TMM_memory2[0];
    out3 <= TMM_memory3[0];
    out4 <= TMM_memory4[0];
  end
1: begin
    out1 <= TMM_memory1[1];
    out2 <= TMM_memory2[1];
    out3 <= TMM_memory3[1];
    out4 <= TMM_memory4[1];
  end
2: begin
    out1 <= TMM_memory1[2];
    out2 <= TMM_memory2[2];
    out3 <= TMM_memory3[2];
    out4 <= TMM_memory4[2];
  end
3: begin
    out1 <= TMM_memory1[3];
    out2 <= TMM_memory2[3];
    out3 <= TMM_memory3[3];
    out4 <= TMM_memory4[3];
  end
4: begin
    out1 <= TMM_memory1[4];
    out2 <= TMM_memory2[4];
    out3 <= TMM_memory3[4];
    out4 <= TMM_memory4[4];
  end
5: begin
    out1 <= TMM_memory1[5];
    out2 <= TMM_memory2[5];
    out3 <= TMM_memory3[5];
    out4 <= TMM_memory4[5];
  end
6: begin
    out1 <= TMM_memory1[6];
    out2 <= TMM_memory2[6];
    out3 <= TMM_memory3[6];
    out4 <= TMM_memory4[6];
  end
7: begin
    out1 <= TMM_memory1[7];
    out2 <= TMM_memory2[7];
    out3 <= TMM_memory3[7];
    out4 <= TMM_memory4[7];
  end
8: begin
    out1 <= TMM_memory1[8];
    out2 <= TMM_memory2[8];
    out3 <= TMM_memory3[8];
    out4 <= TMM_memory4[8];
  end
9: begin
    out1 <= TMM_memory1[9];
    out2 <= TMM_memory2[9];
    out3 <= TMM_memory3[9];
    out4 <= TMM_memory4[9];
  end
10: begin
    out1 <= TMM_memory1[10];
    out2 <= TMM_memory2[10];
    out3 <= TMM_memory3[10];
    out4 <= TMM_memory4[10];
  end
11: begin
    out1 <= TMM_memory1[11];
    out2 <= TMM_memory2[11];
    out3 <= TMM_memory3[11];
    out4 <= TMM_memory4[11];
  end
12: begin
    out1 <= TMM_memory1[12];
    out2 <= TMM_memory2[12];
    out3 <= TMM_memory3[12];
    out4 <= TMM_memory4[12];
  end
13: begin
    out1 <= TMM_memory1[13];
    out2 <= TMM_memory2[13];
    out3 <= TMM_memory3[13];
    out4 <= TMM_memory4[13];
  end
14: begin
    out1 <= TMM_memory1[14];
    out2 <= TMM_memory2[14];
    out3 <= TMM_memory3[14];
    out4 <= TMM_memory4[14];
  end
15: begin
    out1 <= TMM_memory1[15];
    out2 <= TMM_memory2[15];
    out3 <= TMM_memory3[15];
    out4 <= TMM_memory4[15];
  end
    endcase
  end
end
endmodule
```

Supplementary Figure 20: Accumulator.



```
module Activation_Normalization_Unit4x4(
    tiled_computing_sig,      AN_Unit_store_complete,      func_select,
VorM_MaxVal_sel, Nfactor_in, Acuumulator_sendto_AN_enable,
    in1, in2, in3, in4,
    Vecwise_in1,      Vecwise_in2,      Vecwise_in3,      Vecwise_in4,
Matwise_in,
    out1, out2, out3, out4,
    AN_Unit_work, rst, clk
);
    parameter BIT_WIDTH = 8;
    parameter word_size = 16;
    parameter memory_size = 4;
    parameter tiled_MM_memory_size = 16;
    input tiled_computing_sig;
    output reg AN_Unit_store_complete;
    input [2:0] func_select;
    input VorM_MaxVal_sel;
    input [BIT_WIDTH-1:0] Nfactor_in;
    input Acuumulator_sendto_AN_enable;
    input [2*BIT_WIDTH-1:0] in1, in2, in3, in4;
    input      [2*BIT_WIDTH-1:0]      Vecwise_in1,      Vecwise_in2,
Vecwise_in3, Vecwise_in4, Matwise_in;
    input AN_Unit_work, rst, clk;
    output reg [BIT_WIDTH-1:0] out1, out2, out3, out4;
    reg [2*BIT_WIDTH-1:0] act_out1, act_out2, act_out3, act_out4;
    reg tiled_computing_sig_reg;

    reg   [word_size-1:0]  TMM_memory1  [tiled_MM_memory_size-
1:0];
    reg   [word_size-1:0]  TMM_memory2  [tiled_MM_memory_size-
1:0];
    reg   [word_size-1:0]  TMM_memory3  [tiled_MM_memory_size-
1:0];
    reg   [word_size-1:0]  TMM_memory4  [tiled_MM_memory_size-
1:0];

    reg          [2*BIT_WIDTH-1:0]          Vectorwise_MaxVal_r1,
Vectorwise_MaxVal_r2,                    Vectorwise_MaxVal_r3,
Vectorwise_MaxVal_r4;
    reg [2*BIT_WIDTH-1:0] Matrixwise_MaxVal;
    reg VorM_MaxVal_sel_reg;
    reg [BIT_WIDTH-1:0] Nfactor_reg;
    reg     [2*BIT_WIDTH+8:0]      temp24b_out1,      temp24b_out2,
temp24b_out3, temp24b_out4;

    reg [5:0] store_cycle_count;
    reg [5:0] readout_cycle_count;

    always @(posedge clk) begin
        if (rst) begin
            store_cycle_count = 0;
        end else if (Acuumulator_sendto_AN_enable) begin
            store_cycle_count = store_cycle_count + 1;
        end
    end

    always @(*) begin
        tiled_computing_sig_reg = tiled_computing_sig;
    end

    always @(posedge clk) begin
        case (func_select)
            2'b01: begin
                act_out1 = in1;
                act_out2 = in2;
                act_out3 = in3;
                act_out4 = in4;
            end
            2'b10: begin
                act_out1 = in1;
                act_out2 = in2;
                act_out3 = in3;
                act_out4 = in4;
            end
            2'b11: begin
                act_out1 = in1;
                act_out2 = in2;
                act_out3 = in3;
                act_out4 = in4;
            end
            default: begin
                act_out1 = (in1[15] == 0) ? in1 : 16'b0;
                act_out2 = (in2[15] == 0) ? in2 : 16'b0;
                act_out3 = (in3[15] == 0) ? in3 : 16'b0;
                act_out4 = (in4[15] == 0) ? in4 : 16'b0;
            end
        endcase
    end

    always @(posedge clk) begin
        if (rst) begin
            AN_Unit_store_complete = 0;
        end else if (AN_Unit_work) begin
            readout_cycle_count = readout_cycle_count + 1;
        end                else                if(Acuumulator_sendto_AN_enable
&& !tiled_computing_sig_reg) begin
            Vectorwise_MaxVal_r1 = Vecwise_in1;
            Vectorwise_MaxVal_r2 = Vecwise_in2;
            Vectorwise_MaxVal_r3 = Vecwise_in3;
            Vectorwise_MaxVal_r4 = Vecwise_in4;
            Matrixwise_MaxVal = Matwise_in;
        case (store_cycle_count)
            0: begin
                TMM_memory1[0] = act_out1;
                TMM_memory2[0] = act_out2;
                TMM_memory3[0] = act_out3;
                TMM_memory4[0] = act_out4;
                AN_Unit_store_complete = 0;
            end
            1: begin
                TMM_memory1[0] = act_out1;
                TMM_memory2[0] = act_out2;
                TMM_memory3[0] = act_out3;
                TMM_memory4[0] = act_out4;
                AN_Unit_store_complete = 0;
            end
            2: begin
                TMM_memory1[0] = act_out1;
                TMM_memory2[0] = act_out2;
                TMM_memory3[0] = act_out3;
                TMM_memory4[0] = act_out4;
                AN_Unit_store_complete = 0;
            end
            3: begin
                TMM_memory1[1] = act_out1;
                TMM_memory2[1] = act_out2;
                TMM_memory3[1] = act_out3;
                TMM_memory4[1] = act_out4;
                AN_Unit_store_complete = 0;
            end
            4: begin
                TMM_memory1[2] = act_out1;
                TMM_memory2[2] = act_out2;
                TMM_memory3[2] = act_out3;
                TMM_memory4[2] = act_out4;
                AN_Unit_store_complete = 0;
            end
            5: begin
                TMM_memory1[3] = act_out1;
                TMM_memory2[3] = act_out2;
                TMM_memory3[3] = act_out3;
                TMM_memory4[3] = act_out4;
                AN_Unit_store_complete = 0;
            end
            6: begin
                TMM_memory1[4] = act_out1;
                TMM_memory2[4] = act_out2;
                TMM_memory3[4] = act_out3;
                TMM_memory4[4] = act_out4;
                AN_Unit_store_complete = 0;
            end
            7: begin
                TMM_memory1[5] = act_out1;
                TMM_memory2[5] = act_out2;
                TMM_memory3[5] = act_out3;
                TMM_memory4[5] = act_out4;
                AN_Unit_store_complete = 0;
            end
            8: begin
                TMM_memory1[6] = act_out1;
                TMM_memory2[6] = act_out2;
                TMM_memory3[6] = act_out3;
                TMM_memory4[6] = act_out4;
                AN_Unit_store_complete = 0;
            end
            9: begin
                TMM_memory1[7] = act_out1;
                TMM_memory2[7] = act_out2;
                TMM_memory3[7] = act_out3;
                TMM_memory4[7] = act_out4;
                AN_Unit_store_complete = 0;
            end
            10: begin
                TMM_memory1[8] = act_out1;
                TMM_memory2[8] = act_out2;
                TMM_memory3[8] = act_out3;
                TMM_memory4[8] = act_out4;
                AN_Unit_store_complete = 0;
            end
            11: begin
                TMM_memory1[9] = act_out1;
                TMM_memory2[9] = act_out2;
                TMM_memory3[9] = act_out3;
                TMM_memory4[9] = act_out4;
                AN_Unit_store_complete = 0;
            end
            12: begin
                TMM_memory1[10] = act_out1;
                TMM_memory2[10] = act_out2;
                TMM_memory3[10] = act_out3;
                TMM_memory4[10] = act_out4;
                AN_Unit_store_complete = 0;
            end
            13: begin
                TMM_memory1[11] = act_out1;
                TMM_memory2[11] = act_out2;
                TMM_memory3[11] = act_out3;
                TMM_memory4[11] = act_out4;
                AN_Unit_store_complete = 0;
            end
            14: begin
                TMM_memory1[12] = act_out1;
                TMM_memory2[12] = act_out2;
                TMM_memory3[12] = act_out3;
                TMM_memory4[12] = act_out4;
                AN_Unit_store_complete = 0;
            end
            15: begin
                TMM_memory1[13] = act_out1;
                TMM_memory2[13] = act_out2;
                TMM_memory3[13] = act_out3;
                TMM_memory4[13] = act_out4;
                AN_Unit_store_complete = 0;
            end
            16: begin
                TMM_memory1[14] = act_out1;
                TMM_memory2[14] = act_out2;
                TMM_memory3[14] = act_out3;
                TMM_memory4[14] = act_out4;
                AN_Unit_store_complete = 0;
            end
            17: begin
                TMM_memory1[15] = act_out1;
                TMM_memory2[15] = act_out2;
                TMM_memory3[15] = act_out3;
                TMM_memory4[15] = act_out4;
                AN_Unit_store_complete = 1;
                readout_cycle_count = 0;
            end
        endcase
        end
    end

    always @(*) begin
        VorM_MaxVal_sel_reg = VorM_MaxVal_sel;
        Nfactor_reg = Nfactor_in;
    end

always @(posedge clk) begin
if (AN_Unit_work && tiled_computing_sig_reg) begin
case (readout_cycle_count)
8'd0: begin
    temp24b_out1 = (TMM_memory1[0] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[0] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[0] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[0] * Nfactor_reg);
end
8'd1: begin
    temp24b_out1 = (TMM_memory1[0] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[0] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[0] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[0] * Nfactor_reg);
end
8'd2: begin
    temp24b_out1 = (TMM_memory1[1] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[1] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[1] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[1] * Nfactor_reg);
end
8'd3: begin
    temp24b_out1 = (TMM_memory1[2] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[2] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[2] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[2] * Nfactor_reg);
end
8'd4: begin
    temp24b_out1 = (TMM_memory1[3] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[3] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[3] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[3] * Nfactor_reg);
end
8'd5: begin
    temp24b_out1 = (TMM_memory1[4] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[4] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[4] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[4] * Nfactor_reg);
end
8'd6: begin
    temp24b_out1 = (TMM_memory1[4] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[4] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[4] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[4] * Nfactor_reg);
end
8'd7: begin
    temp24b_out1 = (TMM_memory1[4] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[4] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[4] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[4] * Nfactor_reg);
end
8'd8: begin
    temp24b_out1 = (TMM_memory1[5] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[5] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[5] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[5] * Nfactor_reg);
end
8'd9: begin
    temp24b_out1 = (TMM_memory1[6] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[6] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[6] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[6] * Nfactor_reg);
end
8'd10: begin
    temp24b_out1 = (TMM_memory1[7] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[7] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[7] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[7] * Nfactor_reg);
end
8'd11: begin
    temp24b_out1 = (TMM_memory1[8] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[8] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[8] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[8] * Nfactor_reg);
end
8'd12: begin
    temp24b_out1 = (TMM_memory1[8] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[8] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[8] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[8] * Nfactor_reg);
end
8'd13: begin
    temp24b_out1 = (TMM_memory1[8] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[8] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[8] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[8] * Nfactor_reg);
end
8'd14: begin
    temp24b_out1 = (TMM_memory1[9] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[9] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[9] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[9] * Nfactor_reg);
end
8'd15: begin
   temp24b_out1 = (TMM_memory1[10] * Nfactor_reg);
   temp24b_out2 = (TMM_memory2[10] * Nfactor_reg);
   temp24b_out3 = (TMM_memory3[10] * Nfactor_reg);
   temp24b_out4 = (TMM_memory4[10] * Nfactor_reg);
end
8'd16: begin
    temp24b_out1 = (TMM_memory1[11] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[11] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[11] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[11] * Nfactor_reg);
end
8'd17: begin
    temp24b_out1 = (TMM_memory1[12] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[12] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[12] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[12] * Nfactor_reg);
end
8'd18: begin
    temp24b_out1 = (TMM_memory1[12] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[12] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[12] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[12] * Nfactor_reg);
end
8'd19: begin
    temp24b_out1 = (TMM_memory1[12] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[12] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[12] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[12] * Nfactor_reg);
end
8'd20: begin
    temp24b_out1 = (TMM_memory1[13] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[13] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[13] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[13] * Nfactor_reg);
end
8'd21: begin
    temp24b_out1 = (TMM_memory1[14] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[14] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[14] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[14] * Nfactor_reg);
end
8'd22: begin
    temp24b_out1 = (TMM_memory1[15] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[15] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[15] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[15] * Nfactor_reg);
end
endcase

        out1 = temp24b_out1 / Matrixwise_MaxVal;
        out2 = temp24b_out2 / Matrixwise_MaxVal;
        out3 = temp24b_out3 / Matrixwise_MaxVal;
        out4 = temp24b_out4 / Matrixwise_MaxVal;
end else if (AN_Unit_work && !tiled_computing_sig_reg
&& !VorM_MaxVal_sel_reg) begin
case(readout_cycle_count)
8'd0: begin
    temp24b_out1 = (TMM_memory1[0] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[0] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[0] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[0] * Nfactor_reg);
    out1 = temp24b_out1 / Vectorwise_MaxVal_r1;
    out2 = temp24b_out2 / Vectorwise_MaxVal_r1;
    out3 = temp24b_out3 / Vectorwise_MaxVal_r1;
    out4 = temp24b_out4 / Vectorwise_MaxVal_r1;
end
8'd1: begin
    temp24b_out1 = (TMM_memory1[0] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[0] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[0] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[0] * Nfactor_reg);
    out1 = temp24b_out1 / Vectorwise_MaxVal_r1;
    out2 = temp24b_out2 / Vectorwise_MaxVal_r1;
    out3 = temp24b_out3 / Vectorwise_MaxVal_r1;
    out4 = temp24b_out4 / Vectorwise_MaxVal_r1;
end
8'd2: begin
    temp24b_out1 = (TMM_memory1[1] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[1] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[1] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[1] * Nfactor_reg);
    out1 = temp24b_out1 / Vectorwise_MaxVal_r2;
    out2 = temp24b_out2 / Vectorwise_MaxVal_r2;
    out3 = temp24b_out3 / Vectorwise_MaxVal_r2;
    out4 = temp24b_out4 / Vectorwise_MaxVal_r2;
end
8'd3: begin
    temp24b_out1 = (TMM_memory1[2] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[2] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[2] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[2] * Nfactor_reg);
    out1 = temp24b_out1 / Vectorwise_MaxVal_r3;
    out2 = temp24b_out2 / Vectorwise_MaxVal_r3;
    out3 = temp24b_out3 / Vectorwise_MaxVal_r3;
    out4 = temp24b_out4 / Vectorwise_MaxVal_r3;
end
8'd4: begin
    temp24b_out1 = (TMM_memory1[3] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[3] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[3] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[3] * Nfactor_reg);
    out1 = temp24b_out1 / Vectorwise_MaxVal_r4;
    out2 = temp24b_out2 / Vectorwise_MaxVal_r4;
    out3 = temp24b_out3 / Vectorwise_MaxVal_r4;
    out4 = temp24b_out4 / Vectorwise_MaxVal_r4;
end
    default: begin
end
endcase
end else if (AN_Unit_work && !tiled_computing_sig_reg
&& VorM_MaxVal_sel_reg) begin
case(readout_cycle_count)
8'd0: begin
    temp24b_out1 = (TMM_memory1[0] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[0] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[0] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[0] * Nfactor_reg);
    out1 = temp24b_out1 / Matrixwise_MaxVal;
    out2 = temp24b_out2 / Matrixwise_MaxVal;
    out3 = temp24b_out3 / Matrixwise_MaxVal;
    out4 = temp24b_out4 / Matrixwise_MaxVal;
end
8'd1: begin
    temp24b_out1 = (TMM_memory1[0] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[0] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[0] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[0] * Nfactor_reg);
    out1 = temp24b_out1 / Matrixwise_MaxVal;
    out2 = temp24b_out2 / Matrixwise_MaxVal;
    out3 = temp24b_out3 / Matrixwise_MaxVal;
    out4 = temp24b_out4 / Matrixwise_MaxVal;
end
8'd2: begin
    temp24b_out1 = (TMM_memory1[1] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[1] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[1] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[1] * Nfactor_reg);
    out1 = temp24b_out1 / Matrixwise_MaxVal;
    out2 = temp24b_out2 / Matrixwise_MaxVal;
    out3 = temp24b_out3 / Matrixwise_MaxVal;
    out4 = temp24b_out4 / Matrixwise_MaxVal;
end
8'd3: begin
    temp24b_out1 = (TMM_memory1[2] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[2] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[2] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[2] * Nfactor_reg);
    out1 = temp24b_out1 / Matrixwise_MaxVal;
    out2 = temp24b_out2 / Matrixwise_MaxVal;
    out3 = temp24b_out3 / Matrixwise_MaxVal;
    out4 = temp24b_out4 / Matrixwise_MaxVal;
end
8'd4: begin
    temp24b_out1 = (TMM_memory1[3] * Nfactor_reg);
    temp24b_out2 = (TMM_memory2[3] * Nfactor_reg);
    temp24b_out3 = (TMM_memory3[3] * Nfactor_reg);
    temp24b_out4 = (TMM_memory4[3] * Nfactor_reg);
    out1 = temp24b_out1 / Matrixwise_MaxVal;
    out2 = temp24b_out2 / Matrixwise_MaxVal;
    out3 = temp24b_out3 / Matrixwise_MaxVal;
    out4 = temp24b_out4 / Matrixwise_MaxVal;
end
endmodule
```

Supplementary Figure 21: Activation & Normalization Unit.

Supplementary Figure 22: TPU Controller.

Supplementary Figure 23: Tensor Processing Unit (TPU) Top Module.

```
module L2_Controller(
  L2HM_Weight_rd, L2HM_Weight_counter_rst, // for L2 Host Memory Weight
  L2HM_Activation_rd, L2HM_Activation_wr, L2HM_Activation_rd_counter_rst, L2HM_Activation_wr_counter_rst, // for L2 Host Memory Activation
  matrix_compute_size, tiled_computing_sig, zerofill_sig, // for both W&A TSDSU, tiled_computing_sig also for L2 Accumulator
  TSDSUW_rd, TSDSUW_wr, TSDSUW_rst, // for TSDSUW
  TSDSUA_rd, TSDSUA_wr, TSDSUA_rst, // for TSDSUA
  L2HM_ISA_addr, L2HM_ISA_rd, L2HM_ISA_wr, L2HM_ISA_finished_sig, // for L2 Host Memory ISA
  L2IR_out, L2IR_rd, L2IR_wr, // for L2 IR Memory (stored ISA for L2_Controller)
  L2IR_inc, L2IR_clr, // for L2 IR counter
  L1_Controller_start_sig, L1IR_addr, L1IR_wr, TPU_Weight_TsdsU_wr, TPU_Activation_TsdsU_wr, TPU_Activation_L1HM_rd, TPU_L1HW_counter_rst, // for TPU4x4
  TPU_Computation_finished_signal, // for TPU4x4
  clk, rst
  );

  parameter Instruction_size = 16;
  parameter address_size = 8;
  parameter MMU_Compute_size = 6;
  parameter Data_size = 8;

  parameter S_initial = 0, S_fetch = 1;
  parameter S_decode = 2;
  parameter S_ISA_L2HM_to_L1HM1 = 3, S_ISA_L2HM_to_L1HM2 = 4;
  parameter S_WandA_L2HostMem_to_TSDSU1 = 5, S_WandA_L2HostMem_to_TSDSU2 = 6, S_Wait_TSDSU_loading_for_one_more_cycle = 7;
  parameter S_WandA_TSDSU_to_L1HostMem1 = 8, S_WandA_TSDSU_to_L1HostMem2 = 9, S_Wait_L1HostMem_loading_for_one_more_cycle = 10;
  parameter S_TPU_working1 = 11, S_TPU_working2 = 12;
  parameter S_Activation_back_to_L2HostMem1 = 13, S_Activation_back_to_L2HostMem2 = 14, S_Wait_L2HostMem_storing_for_one_more_cycle = 15;
  parameter S_stay = 16;
  // opcode //
  parameter Load_ISA = 0;
  parameter Read_L2_Host_Memory = 1;
  parameter TPU_work = 2;
  parameter Write_L2_Host_Memory = 3;

  output reg L2HM_Weight_rd, L2HM_Weight_counter_rst; // for L2 Host Memory Weight
  output reg L2HM_Activation_rd, L2HM_Activation_wr, L2HM_Activation_rd_counter_rst, L2HM_Activation_wr_counter_rst; // for L2 Host Memory Activation
  output reg [MMU_Compute_size-1:0] matrix_compute_size; // for both W&A TSDSU, tiled_computing_sig also for L2 Accumulator
  output reg tiled_computing_sig, zerofill_sig;
  output reg TSDSUW_rd, TSDSUW_wr, TSDSUW_rst; // for TSDSUW
  output reg TSDSUA_rd, TSDSUA_wr, TSDSUA_rst; // for TSDSUA
  output reg [addresss_size-1:0] L2HM_ISA_addr; // for L2 Host Memory ISA
  output reg L2HM_ISA_rd, L2HM_ISA_wr;
  input L2HM_ISA_finished_sig;
  input [Instruction_size-1:0] L2IR_out; // for L2 IR Memory (stored ISA for L2_Controller)
  output reg L2IR_rd, L2IR_wr;
  output reg L2IR_inc, L2IR_clr; // for L2 IR counter
  output reg L1_Controller_start_sig, L1IR_wr, TPU_Weight_TsdsU_wr, TPU_Activation_TsdsU_wr, TPU_Activation_L1HM_rd, TPU_L1HW_counter_rst; // for TPU4x4
  output reg [addresss_size-1:0] L1IR_addr; // for L1 IR Mem
  input TPU_Computation_finished_signal;
  input clk, rst;

  reg [4:0] state, next_state;
  wire [Instruction_size-1:0] L2IR_out;
  wire [3:0] opcode = L2IR_out[15:12];
  reg [address_size-1:0] L2HM_ISA_addr_count;
  reg [address_size-1:0] L1IR_addr_count;
  reg [address_size-1:0] L2_to_TSDSU_rd_counter;
  reg [address_size-1:0] TSDSU_to_L1_rd_counter;
  reg tiled_computing_sig_reg;
  reg [MMU_Compute_size-1:0] matrix_compute_size_reg;
  reg [address_size-1:0] L1HM_to_L2HM_wr_counter;

  always@(posedge clk or posedge rst)begin: State_transitions
    if(rst == 1) state <= S_initial; else state <= next_state;
  end

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      L2HM_ISA_addr_count <= 8'b00000000;
    end else if (next_state == S_ISA_L2HM_to_L1HM1 || next_state == S_ISA_L2HM_to_L1HM2) begin
      L2HM_ISA_addr_count <= L2HM_ISA_addr_count + 8'b00000001;
    end
  end

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      L1IR_addr_count <= 8'b00000000;
    end else if (next_state == S_ISA_L2HM_to_L1HM1 || next_state == S_ISA_L2HM_to_L1HM2) begin
      L1IR_addr_count <= L1IR_addr_count + 8'b00000001;
    end
  end

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      L2_to_TSDSU_rd_counter <= 8'b00000000;
    end else if ((next_state == S_WandA_L2HostMem_to_TSDSU1 || next_state == S_WandA_L2HostMem_to_TSDSU2) && L2_to_TSDSU_rd_counter < 4) begin
      L2_to_TSDSU_rd_counter <= L2_to_TSDSU_rd_counter + 8'b00000001;
    end
  end

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      TSDSU_to_L1_rd_counter <= 8'b00000000;
    end else if ((next_state == S_WandA_TSDSU_to_L1HostMem1 || next_state == S_WandA_TSDSU_to_L1HostMem2) && TSDSU_to_L1_rd_counter < 4) begin
      TSDSU_to_L1_rd_counter <= TSDSU_to_L1_rd_counter + 8'b00000001;
    end
  end

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      L1HM_to_L2HM_wr_counter <= 8'b00000000;
    end else if ((next_state == S_Activation_back_to_L2HostMem1 || next_state == S_Activation_back_to_L2HostMem2) && L1HM_to_L2HM_wr_counter < 4) begin
      L1HM_to_L2HM_wr_counter <= L1HM_to_L2HM_wr_counter + 8'b00000001;
    end
  end

  always@(L2IR_out) begin
    if (L2IR_out[15:12] == 1) begin
      tiled_computing_sig_reg <= L2IR_out[6];
    end else begin
      tiled_computing_sig_reg <= tiled_computing_sig_reg;
    end
  end

  always@(L2IR_out) begin
    if (L2IR_out[15:12] == 1) begin
      matrix_compute_size_reg = L2IR_out[5:0];
    end
  end
  always@(state or opcode or L2HM_ISA_finished_sig or TPU_Computation_finished_signal)begin
    L2HM_Weight_rd = 0; L2HM_Weight_counter_rst = 0; L2HM_Activation_rd = 0; L2HM_Activation_wr = 0; L2HM_Activation_rd_counter_rst = 0; L2HM_Activation_wr_counter_rst = 0;
    matrix_compute_size = 6'b000000; tiled_computing_sig = 0; zerofill_sig = 0;
    TSDSUW_rd = 0; TSDSUW_wr = 0; TSDSUW_rst = 0; TSDSUA_rd = 0; TSDSUA_wr = 0; TSDSUA_rst = 0;
    L2HM_ISA_addr = 8'b00000000; L2HM_ISA_rd = 0; L2HM_ISA_wr = 0; L2IR_rd = 0; L2IR_wr = 0; L2IR_inc = 0; L2IR_clr = 0;
    L1_Controller_start_sig = 0; L1IR_addr = 8'b00000000; L1IR_wr = 0; TPU_Weight_TsdsU_wr = 0; TPU_Activation_TsdsU_wr = 0; TPU_Activation_L1HM_rd = 0; TPU_L1HW_counter_rst = 0;
    next_state = state;

    case (state)
      S_initial   :begin
                   next_state = S_fetch;
                   L2IR_clr = 1;
                   L2HM_Weight_counter_rst = 1;
                   L2HM_Activation_rd_counter_rst = 1;
                   L2HM_Activation_wr_counter_rst = 1;
                   TSDSUW_rst = 1;
                   TSDSUA_rst = 1;
                   TPU_L1HW_counter_rst = 1;
                   $display("State: S_I2_initial");
                   end
      S_fetch     :begin
                   next_state = S_decode;
                   L2IR_clr = 0;
                   L2IR_rd = 1;
                   $display("State: S_I2_fetch");
                   end
      S_decode    :begin
                   $display("opcode: %b", opcode);
                   case(opcode)
                   // opcode = 0
                   Load_ISA: begin
                   L2HM_ISA_rd = 1;
                   L2HM_ISA_addr = L2HM_ISA_addr_count;
                   next_state = S_ISA_L2HM_to_L1HM1;
                   $display("State: Load_ISA");
                   end

                   // opcode = 1
                   Read_L2_Host_Memory: begin
                   L2HM_Weight_rd = 1;
                   L2HM_Activation_rd = 1;
                   zerofill_sig = 1;
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   next_state = S_WandA_L2HostMem_to_TSDSU1;
                   $display("State: Read_L2_Host_Memory");
                   end

                   // opcode = 2
                   TPU_work: begin
                   L1_Controller_start_sig = 1;
                   tiled_computing_sig = tiled_computing_sig_reg;
                   next_state = S_TPU_working1;
                   $display("State: TPU_work");
                   end

                   //opcode = 3
                   Write_L2_Host_Memory: begin
                   TPU_Activation_L1HM_rd = 1;
                   next_state = S_Activation_back_to_L2HostMem1;
                   $display("State: Write_L2_Host_Memory");
                   end

                   default:
                   begin next_state = S_stay;
                   $display("L2 Invalid opcode");
                   end
                   endcase
                   end
      S_ISA_L2HM_to_L1HM1: begin
                   if(!L2HM_ISA_finished_sig)begin
                   $display("State: ISA loading to L1 Host Mem1... ");
                   L2HM_ISA_rd = 1;
                   L2HM_ISA_addr = L2HM_ISA_addr_count;
                   L1IR_addr = L1IR_addr_count - 8'b00000001;
                   L1IR_wr = 1;
                   next_state = S_ISA_L2HM_to_L1HM2;
                   end else if(L2HM_ISA_finished_sig) begin
                   $display("State: ISA finished loading into L1 Host Mem!!!");
                   L2HM_ISA_rd = 0;
                   L1IR_wr = 0;
                   next_state = S_fetch;
                   L2IR_inc = 1;
                   end
                   end
      S_ISA_L2HM_to_L1HM2: begin
                   if(!L2HM_ISA_finished_sig)begin
                   $display("State: ISA loading to L1 Host Mem2... ");
                   L2HM_ISA_rd = 1;
                   L2HM_ISA_addr = L2HM_ISA_addr_count;
                   L1IR_addr = L1IR_addr_count - 8'b00000001;
                   L1IR_wr = 1;
                   next_state = S_ISA_L2HM_to_L1HM1;
                   end else if(L2HM_ISA_finished_sig) begin
                   $display("State: ISA finished loading into L1 Host Mem!!!");
                   L2HM_ISA_rd = 0;
                   L1IR_wr = 0;
                   next_state = S_fetch;
                   L2IR_inc = 1;
                   end
                   end
      S_WandA_L2HostMem_to_TSDSU1: begin
                   if(L2_to_TSDSU_rd_counter <= 3)begin
                   $display("State: Weights and Activations loading to TSDSU 1... ");
                   zerofill_sig = 0;
                   L2HM_Weight_rd = 1;
                   L2HM_Activation_rd = 1;
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   TSDSUW_wr = 1;
                   TSDSUA_wr = 1;
                   next_state = S_WandA_L2HostMem_to_TSDSU2;
                   end else begin
                   $display("State: Weights and Activations loading to TSDSU 1... ");
                   L2HM_Weight_rd = 0;
                   L2HM_Activation_rd = 0;
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   TSDSUW_wr = 1;
                   TSDSUA_wr = 1;
                   next_state = S_Wait_TSDSU_loading_for_one_more_cycle;
                   end
                   end
      S_WandA_L2HostMem_to_TSDSU2: begin
                   if(L2_to_TSDSU_rd_counter <= 3)begin
                   $display("State: Weights and Activations loading to TSDSU 2... ");
                   zerofill_sig = 0;
                   L2HM_Weight_rd = 1;
                   L2HM_Activation_rd = 1;
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   TSDSUW_wr = 1;
                   TSDSUA_wr = 1;
                   next_state = S_WandA_L2HostMem_to_TSDSU1;
                   end else begin
                   $display("State: Weights and Activations loading to TSDSU 2... ");
                   L2HM_Weight_rd = 0;
                   L2HM_Activation_rd = 0;
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   TSDSUW_wr = 1;
                   TSDSUA_wr = 1;
                   next_state = S_Wait_TSDSU_loading_for_one_more_cycle;
                   end
                   end
      S_Wait_TSDSU_loading_for_one_more_cycle: begin
                   $display("State: Weights and Activations finished loading into TSDSU!!!");
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   TSDSUW_wr = 0;
                   TSDSUA_wr = 0;
                   TSDSUW_rd = 1;
                   TSDSUA_rd = 1;
                   next_state = S_WandA_TSDSU_to_L1HostMem1;
                   end
      S_WandA_TSDSU_to_L1HostMem1: begin
                   if(TSDSU_to_L1_rd_counter <= 3)begin
                   $display("State: Weights and Activations loading to L1 HostMem 1... ");
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   TSDSUW_rd = 1;
                   TSDSUA_rd = 1;
                   TPU_Weight_TsdsU_wr = 1;
                   TPU_Activation_TsdsU_wr = 1;
                   next_state = S_WandA_TSDSU_to_L1HostMem2;
                   end else begin
                   $display("State: Weights and Activations loading to L1 HostMem 1... ");
                   TSDSUW_rd = 0;
                   TSDSUA_rd = 0;
                   TPU_Weight_TsdsU_wr = 1;
                   TPU_Activation_TsdsU_wr = 1;
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   next_state = S_Wait_L1HostMem_loading_for_one_more_cycle;
                   end
                   end
      S_WandA_TSDSU_to_L1HostMem2: begin
                   if(TSDSU_to_L1_rd_counter <= 3)begin
                   $display("State: Weights and Activations loading to L1 HostMem 2... ");
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   TSDSUW_rd = 1;
                   TSDSUA_rd = 1;
                   TPU_Weight_TsdsU_wr = 1;
                   TPU_Activation_TsdsU_wr = 1;
                   next_state = S_WandA_TSDSU_to_L1HostMem1;
                   end else begin
                   $display("State: Weights and Activations loading to L1 HostMem 2... ");
                   TSDSUW_rd = 0;
                   TSDSUA_rd = 0;
                   TPU_Weight_TsdsU_wr = 1;
                   TPU_Activation_TsdsU_wr = 1;
                   tiled_computing_sig = tiled_computing_sig_reg;
                   matrix_compute_size = matrix_compute_size_reg;
                   next_state = S_Wait_L1HostMem_loading_for_one_more_cycle;
                   end
                   end
      S_Wait_L1HostMem_loading_for_one_more_cycle: begin
                   $display("State: Weights and Activations finished loading into L1 HostMem!!!");
                   TPU_Weight_TsdsU_wr = 0;
                   TPU_Activation_TsdsU_wr = 0;
                   next_state = S_fetch;
                   L2IR_inc = 1;
                   end
      S_TPU_working1:     begin
                   L1_Controller_start_sig = 0;
                   if (!TPU_Computation_finished_signal) begin
                   tiled_computing_sig = tiled_computing_sig_reg;
                   $display("State: TPU_working1...");
                   end else begin
                   $display("State: (Message from L2_Controller) TPU_finished computing!!!");
                   next_state = S_fetch;
                   L2IR_inc = 1;
                   end
                   end
      S_TPU_working2:     begin
                   L1_Controller_start_sig = 0;
                   if (!TPU_Computation_finished_signal) begin
                   tiled_computing_sig = tiled_computing_sig_reg;
                   $display("State: TPU_working2...");
                   end else begin
                   $display("State: (Message from L2_Controller) TPU_finished computing!!!");
                   next_state = S_fetch;
                   L2IR_inc = 1;
                   end
                   end
      S_Activation_back_to_L2HostMem1: begin
                   if (L1HM_to_L2HM_wr_counter <= 3) begin
                   $display("State: Activations loading to L2 HostMem 1... ");
                   TPU_Activation_L1HM_rd = 1;
                   L2HM_Activation_wr = 1;
                   next_state = S_Activation_back_to_L2HostMem2;
                   end else begin
                   TPU_Activation_L1HM_rd = 0;
                   L2HM_Activation_wr = 1;
                   next_state = S_Wait_L2HostMem_storing_for_one_more_cycle;
                   end
                   end
      S_Activation_back_to_L2HostMem2: begin
                   if (L1HM_to_L2HM_wr_counter <= 3) begin
                   $display("State: Activations loading to L2 HostMem 2... ");
                   TPU_Activation_L1HM_rd = 1;
                   L2HM_Activation_wr = 1;
                   next_state = S_Activation_back_to_L2HostMem1;
                   end else begin
                   TPU_Activation_L1HM_rd = 0;
                   L2HM_Activation_wr = 1;
                   next_state = S_Wait_L2HostMem_storing_for_one_more_cycle;
                   end
                   end
      S_Wait_L2HostMem_storing_for_one_more_cycle: begin
                   $display("State: Activations Completely loaded back to L2 HostMem!!! ");
                   L2HM_Activation_wr = 0;
                   next_state = S_fetch;
                   L2IR_inc = 1;
      S_stay:     begin
                   L2IR_rd = 1;
                   $display("State: S_I2_stay");
      end
                   default: begin next_state = S_initial;
                   $display("State: Default, resetting to S_I2_initial");
                   end
                   endcase
      end

      endmodule
```

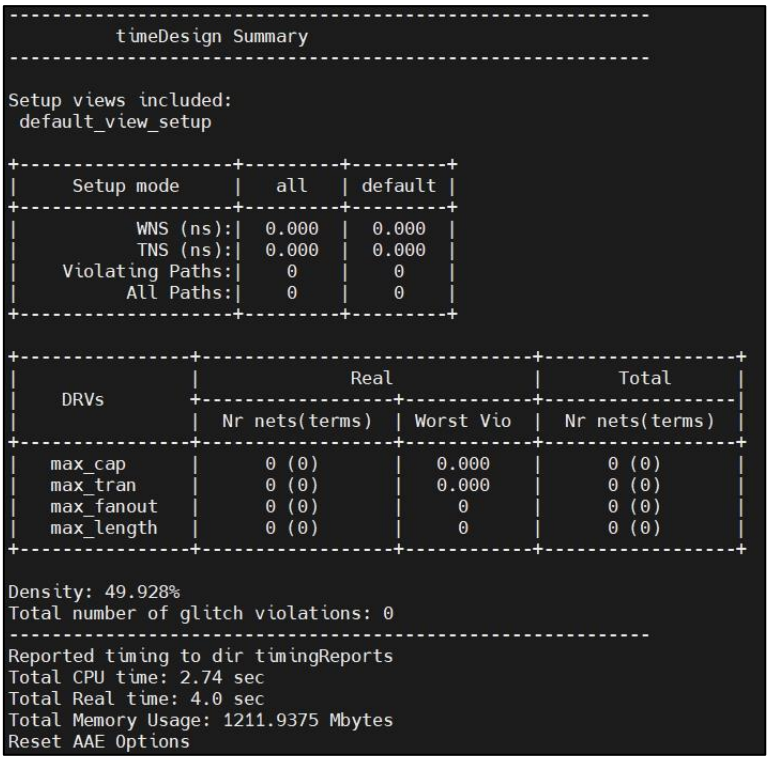
Supplementary Figure 24: Level 2 Controller.

```
module TMMU4x4(
    L2HM_Weight_in, L2HM_Weight_addr, L2HM_Weight_wr, // from L3 Host Mem Weight
    L2HM_Activation_addr,
    L2HWISA_in, L2HWISA_wr, // from L3 Host Mem ISA
    L2IR_in, L2IR_wr, // from L3 structure
    clk, rst);

    parameter Instruction_size = 16;
    parameter address_size = 8;
    parameter MMU_Compute_size = 6;
    parameter Data_size = 8;

    input [Data_size-1:0] L2HM_Weight_in; // from probably L3 Memory
    input [address_size-1:0] L2HM_Weight_addr, L2HM_Activation_addr; // from probably L3 Controller
    input L2HM_Weight_wr; // from probably L3 Controller
    input [Instruction_size-1:0] L2HWISA_in; // from probably L3 Memory
    input L2HWISA_wr; // from probably L3 Controller
    input [Instruction_size-1:0] L2IR_in; // from L3 structure
    input L2IR_wr; // from probably L3 Controller
    input clk, rst;

    // for L2 Host Mem Weight
    wire [Data_size-1:0] Weight_L2HM_to_TSDSUW1, Weight_L2HM_to_TSDSUW2, Weight_L2HM_to_TSDSUW3,
Weight_L2HM_to_TSDSUW4, Weight_L2HM_to_TSDSUW5, Weight_L2HM_to_TSDSUW6, Weight_L2HM_to_TSDSUW7,
Weight_L2HM_to_TSDSUW8;
    wire [Data_size-1:0] Weight_L2HM_to_TSDSUW9, Weight_L2HM_to_TSDSUW10, Weight_L2HM_to_TSDSUW11,
Weight_L2HM_to_TSDSUW12, Weight_L2HM_to_TSDSUW13, Weight_L2HM_to_TSDSUW14, Weight_L2HM_to_TSDSUW15,
Weight_L2HM_to_TSDSUW16;
    wire L2HM_Weight_rd, L2HM_Weight_rd_counter_rst;
    // for Tiled SDS Unit Weight
    wire [Data_size-1:0] Weight_TSDSUW_to_TPULIHM1, Weight_TSDSUW_to_TPULIHM2,
Weight_TSDSUW_to_TPULIHM3, Weight_TSDSUW_to_TPULIHM4, Weight_TSDSUW_to_TPULIHM5,
Weight_TSDSUW_to_TPULIHM6, Weight_TSDSUW_to_TPULIHM7, Weight_TSDSUW_to_TPULIHM8;
    wire [Data_size-1:0] Weight_TSDSUW_to_TPULIHM9, Weight_TSDSUW_to_TPULIHM10,
Weight_TSDSUW_to_TPULIHM11, Weight_TSDSUW_to_TPULIHM12, Weight_TSDSUW_to_TPULIHM13,
Weight_TSDSUW_to_TPULIHM14, Weight_TSDSUW_to_TPULIHM15, Weight_TSDSUW_to_TPULIHM16;
    wire TSDSUW_rd, TSDSUW_wr, TSDSUW_rst;
    // for L2 Host Mem Activation
    wire [Data_size-1:0] Activation_L2HM_to_TSDSUW1, Activation_L2HM_to_TSDSUW2, Activation_L2HM_to_TSDSUW3,
Activation_L2HM_to_TSDSUW4, Activation_L2HM_to_TSDSUW5, Activation_L2HM_to_TSDSUW6,
Activation_L2HM_to_TSDSUW7, Activation_L2HM_to_TSDSUW8;
    wire [Data_size-1:0] Activation_L2HM_to_TSDSUW9, Activation_L2HM_to_TSDSUW10, Activation_L2HM_to_TSDSUW11,
Activation_L2HM_to_TSDSUW12, Activation_L2HM_to_TSDSUW13, Activation_L2HM_to_TSDSUW14,
Activation_L2HM_to_TSDSUW15, Activation_L2HM_to_TSDSUW16;
    wire L2HM_Activation_rd, L2HM_Activation_wr, L2HM_Activation_rd_counter_rst, L2HM_Activation_wr_counter_rst;
    wire [Data_size-1:0] Activation_L1HM_to_L2HM1, Activation_L1HM_to_L2HM2, Activation_L1HM_to_L2HM3,
Activation_L1HM_to_L2HM4, Activation_L1HM_to_L2HM5, Activation_L1HM_to_L2HM6, Activation_L1HM_to_L2HM7,
Activation_L1HM_to_L2HM8;
    wire [Data_size-1:0] Activation_L1HM_to_L2HM9, Activation_L1HM_to_L2HM10, Activation_L1HM_to_L2HM11,
Activation_L1HM_to_L2HM12, Activation_L1HM_to_L2HM13, Activation_L1HM_to_L2HM14, Activation_L1HM_to_L2HM15,
Activation_L1HM_to_L2HM16;
    // for Tiled SDS Unit Activation
    wire [Data_size-1:0] Activation_TSDSUW_to_TPULIHM1, Activation_TSDSUW_to_TPULIHM2,
Activation_TSDSUW_to_TPULIHM3, Activation_TSDSUW_to_TPULIHM4, Activation_TSDSUW_to_TPULIHM5,
Activation_TSDSUW_to_TPULIHM6, Activation_TSDSUW_to_TPULIHM7, Activation_TSDSUW_to_TPULIHM8;
    wire [Data_size-1:0] Activation_TSDSUW_to_TPULIHM9, Activation_TSDSUW_to_TPULIHM10,
Activation_TSDSUW_to_TPULIHM11, Activation_TSDSUW_to_TPULIHM12, Activation_TSDSUW_to_TPULIHM13,
Activation_TSDSUW_to_TPULIHM14, Activation_TSDSUW_to_TPULIHM15, Activation_TSDSUW_to_TPULIHM16;
    wire TSDSUA_rd, TSDSUA_wr, TSDSUA_rst;
    // for tiled computation
    wire [MMU_Compute_size-1:0] matrix_compute_size;
    wire tiled_computing_sig, zerofill_sig;
    // for L2 Host Mem ISA
    wire [Instruction_size-1:0] L2HWISA_out;
    wire L2HWISA_rd;
    wire [address_size-1:0] L2HM_ISA_addr;
    wire L2HM_ISA_finished_sig;
    // for L2 IR Mem (send to L2 Controller)
    wire [Instruction_size-1:0] L2IR_out;
    wire L2IR_rd;
    wire [address_size-1:0] L2IR_addr;
    wire L2IR_inc, L2IR_clr;
    // for TPU4x4 Unit
    wire L1_Controller_start_sig, L1IR_wr, TPU_Weight_Tsdsl1_wr, TPU_Activation_Tsdsl1_wr, TPU_Activation_L1HM_rd,
TPU_L1HW_counter_rst, TPU_Computation_finished_signal;
    wire [address_size-1:0] L1IR_addr;


    L2_Host_Memory_Weight
L2HMW1(.R_data1(Weight_L2HM_to_TSDSUW1), .R_data2(Weight_L2HM_to_TSDSUW2), .R_data3(Weight_L2HM_to_TSDSU
W3), .R_data4(Weight_L2HM_to_TSDSUW4), .R_data5(Weight_L2HM_to_TSDSUW5), .R_data6(Weight_L2HM_to_TSDSUW6),
.R_data7(Weight_L2HM_to_TSDSUW7), .R_data8(Weight_L2HM_to_TSDSUW8),
    .R_data9(Weight_L2HM_to_TSDSUW9), .R_data10(Weight_L2HM_to_TSDSUW10), .R_data11(Weight_L2HM_to_TSDSUW
11), .R_data12(Weight_L2HM_to_TSDSUW12), .R_data13(Weight_L2HM_to_TSDSUW13), .R_data14(Weight_L2HM_to_TSDSU
W14), .R_data15(Weight_L2HM_to_TSDSUW15), .R_data16(Weight_L2HM_to_TSDSUW16),
    .W_data1(L2HM_Weight_in), .addr(L2HM_Weight_addr), .clk(clk), .rst(L2HM_Weight_rd), .wr(L2HM_Weight_wr),
    .L2HM_counter_rst(L2HM_Weight_rd_counter_rst));

    Tiled_SDS_Unit TSDSUW( // for weight
    .R_data1(Weight_TSDSUW_to_TPULIHM1), .R_data2(Weight_TSDSUW_to_TPULIHM2), .R_data3(Weight_TSDSUW_to_T
PULIHM3), .R_data4(Weight_TSDSUW_to_TPULIHM4), .R_data5(Weight_TSDSUW_to_TPULIHM5), .R_data6(Weight_TSDS
UW_to_TPULIHM6), .R_data7(Weight_TSDSUW_to_TPULIHM7), .R_data8(Weight_TSDSUW_to_TPULIHM8),
    .R_data9(Weight_TSDSUW_to_TPULIHM9), .R_data10(Weight_TSDSUW_to_TPULIHM10), .R_data11(Weight_TSDSUW_t
o_TPULIHM11), .R_data12(Weight_TSDSUW_to_TPULIHM12), .R_data13(Weight_TSDSUW_to_TPULIHM13), .R_data14(Wei
ght_TSDSUW_to_TPULIHM14), .R_data15(Weight_TSDSUW_to_TPULIHM15), .R_data16(Weight_TSDSUW_to_TPULIHM16),
    .W_data1(Weight_L2HM_to_TSDSUW1), .W_data2(Weight_L2HM_to_TSDSUW2), .W_data3(Weight_L2HM_to_TSDSUW3)
, .W_data4(Weight_L2HM_to_TSDSUW4), .W_data5(Weight_L2HM_to_TSDSUW5), .W_data6(Weight_L2HM_to_TSDSUW6), .
W_data7(Weight_L2HM_to_TSDSUW7), .W_data8(Weight_L2HM_to_TSDSUW8),
    .W_data9(Weight_L2HM_to_TSDSUW9), .W_data10(Weight_L2HM_to_TSDSUW10), .W_data11(Weight_L2HM_to_TSDSU
W11), .W_data12(Weight_L2HM_to_TSDSUW12), .W_data13(Weight_L2HM_to_TSDSUW13), .W_data14(Weight_L2HM_to_TS
DSUW14), .W_data15(Weight_L2HM_to_TSDSUW15), .W_data16(Weight_L2HM_to_TSDSUW16),
    .rd(TSDSUW_rd), .wr(TSDSUW_wr), .tiled_computing_sig(tiled_computing_sig), .zerofill_sig(zerofill_sig), .matrix_compute_si
ze(matrix_compute_size), .clk(clk), .rst(TSDSUW_rst)
    );

    L2_Host_Memory_Activation
L2HMA1(.R_data1(Activation_L2HM_to_TSDSUW1), .R_data2(Activation_L2HM_to_TSDSUW2), .R_data3(Activation_L2HM_to
_TSDSUW3), .R_data4(Activation_L2HM_to_TSDSUW4), .R_data5(Activation_L2HM_to_TSDSUW5), .R_data6(Activation_L2H
M_to_TSDSUW6), .R_data7(Activation_L2HM_to_TSDSUW7), .R_data8(Activation_L2HM_to_TSDSUW8),
    .R_data9(Activation_L2HM_to_TSDSUW9), .R_data10(Activation_L2HM_to_TSDSUW10), .R_data11(Activation_L2HM_to_
TSDSUW11), .R_data12(Activation_L2HM_to_TSDSUW12), .R_data13(Activation_L2HM_to_TSDSUW13), .R_data14(Activation_
L2HM_to_TSDSUW14), .R_data15(Activation_L2HM_to_TSDSUW15), .R_data16(Activation_L2HM_to_TSDSUW16),
    .W_data1(Activation_L1HM_to_L2HM1), .W_data2(Activation_L1HM_to_L2HM2), .W_data3(Activation_L1HM_to_L2HM3),
.W_data4(Activation_L1HM_to_L2HM4), .W_data5(Activation_L1HM_to_L2HM5), .W_data6(Activation_L1HM_to_L2HM6), .W_
data7(Activation_L1HM_to_L2HM7), .W_data8(Activation_L1HM_to_L2HM8),
    .W_data9(Activation_L1HM_to_L2HM9), .W_data10(Activation_L1HM_to_L2HM10), .W_data11(Activation_L1HM_to_L2H
M11), .W_data12(Activation_L1HM_to_L2HM12), .W_data13(Activation_L1HM_to_L2HM13), .W_data14(Activation_L1HM_to_L
2HM14), .W_data15(Activation_L1HM_to_L2HM15), .W_data16(Activation_L1HM_to_L2HM16),
    .addr(L2HM_Activation_addr), .clk(clk), .rd(L2HM_Activation_rd), .wr(L2HM_Activation_wr),
    .L2HM_rd_counter_rst(L2HM_Activation_rd_counter_rst), .L2HM_wr_counter_rst(L2HM_Activation_wr_counter_rst));

    Tiled_SDS_Unit TSDSUA( // for activation
    .R_data1(Activation_TSDSUW_to_TPULIHM1), .R_data2(Activation_TSDSUW_to_TPULIHM2), .R_data3(Activation_TSDS
UW_to_TPULIHM3), .R_data4(Activation_TSDSUW_to_TPULIHM4), .R_data5(Activation_TSDSUW_to_TPULIHM5), .R_data6(
Activation_TSDSUW_to_TPULIHM6), .R_data7(Activation_TSDSUW_to_TPULIHM7), .R_data8(Activation_TSDSUW_to_TPULI
HM8),
    .R_data9(Activation_TSDSUW_to_TPULIHM9), .R_data10(Activation_TSDSUW_to_TPULIHM10), .R_data11(Activation_IS
DSUW_to_TPULIHM11), .R_data12(Activation_TSDSUW_to_TPULIHM12), .R_data13(Activation_TSDSUW_to_TPULIHM13),
.R_data14(Activation_TSDSUW_to_TPULIHM14), .R_data15(Activation_TSDSUW_to_TPULIHM15), .R_data16(Activation_TSD
SUW_to_TPULIHM16),
    .W_data1(Activation_L2HM_to_TSDSUW1), .W_data2(Activation_L2HM_to_TSDSUW2), .W_data3(Activation_L2HM_to_T
SDSUW3), .W_data4(Activation_L2HM_to_TSDSUW4), .W_data5(Activation_L2HM_to_TSDSUW5), .W_data6(Activation_L2HM
_to_TSDSUW6), .W_data7(Activation_L2HM_to_TSDSUW7), .W_data8(Activation_L2HM_to_TSDSUW8),
    .W_data9(Activation_L2HM_to_TSDSUW9), .W_data10(Activation_L2HM_to_TSDSUW10), .W_data11(Activation_L2HM_to
_TSDSUW11), .W_data12(Activation_L2HM_to_TSDSUW12), .W_data13(Activation_L2HM_to_TSDSUW13), .W_data14(Activati
on_L2HM_to_TSDSUW14), .W_data15(Activation_L2HM_to_TSDSUW15), .W_data16(Activation_L2HM_to_TSDSUW16),
    .rd(TSDSUA_rd), .wr(TSDSUA_wr), .tiled_computing_sig(tiled_computing_sig), .zerofill_sig(zerofill_sig), .matrix_compute_siz
e(matrix_compute_size), .clk(clk), .rst(TSDSUA_rst)
    );

    L2_Host_Memory_InstructionSet
L2HMISA1(.R_data(L2HWISA_out), .W_data(L2HWISA_in), .addr(L2HM_ISA_addr), .clk(clk), .IR_rd(L2HWISA_rd), .IR_wr(L2
HWISA_wr), .finished_sig(L2HM_ISA_finished_sig));

    L2_IR_Mem
L2IRMEM1(.R_data(L2IR_out), .W_data(L2IR_in), .addr(L2IR_addr), .clk(clk), .IR_rd(L2IR_rd), .IR_wr(L2IR_wr));

    L2_IR_counter L2IRC1(.IR_addr(L2IR_addr), .IR_ins(L2IR_inc), .IR_clr(L2IR_clr), .clk(clk));

    TPU4x4
TPU1( .Controller_start_sig(L1_Controller_start_sig), .IR_ins(L2HWISA_out), .IR_wr_addr(L1IR_addr), .IR_wr(L1IR_wr),
    .Weight_Tsdsl1_wr( TPU_Weight_Tsdsl1_wr), .Activation_Tsdsl1_wr(TPU_Activation_Tsdsl1_wr), .Activation_L1HM_rd( TPU
_Activation_L1HM_rd),
    .Weight_Tsdsl1_in(Weight_TSDSUW_to_TPULIHM1), .Weight_Tsdsl12_in(Weight_TSDSUW_to_TPULIHM2), .Weight_Ts
dsl13_in(Weight_TSDSUW_to_TPULIHM3), .Weight_Tsdsl14_in(Weight_TSDSUW_to_TPULIHM4), .Weight_Tsdsl15_in(Weight
_TSDSUW_to_TPULIHM5), .Weight_Tsdsl16_in(Weight_TSDSUW_to_TPULIHM6), .Weight_Tsdsl17_in(Weight_TSDSUW_to_
TPULIHM7), .Weight_Tsdsl18_in(Weight_TSDSUW_to_TPULIHM8),
    .Weight_Tsdsl19_in(Weight_TSDSUW_to_TPULIHM9), .Weight_Tsdsl110_in(Weight_TSDSUW_to_TPULIHM10), .Weight_
Tsdsl111_in(Weight_TSDSUW_to_TPULIHM11), .Weight_Tsdsl112_in(Weight_TSDSUW_to_TPULIHM12), .Weight_Tsdsl113_i
n(Weight_TSDSUW_to_TPULIHM13), .Weight_Tsdsl114_in(Weight_TSDSUW_to_TPULIHM14), .Weight_Tsdsl115_in(Weight_
TSDSUW_to_TPULIHM15), .Weight_Tsdsl116_in(Weight_TSDSUW_to_TPULIHM16),
    .Activation_Tsdsl1_in(Activation_TSDSUW_to_TPULIHM1), .Activation_Tsdsl12_in(Activation_TSDSUW_to_TPULIHM2)
, .Activation_Tsdsl13_in(Activation_TSDSUW_to_TPULIHM3), .Activation_Tsdsl14_in(Activation_TSDSUW_to_TPULIHM4), .A
ctivation_Tsdsl15_in(Activation_TSDSUW_to_TPULIHM5), .Activation_Tsdsl16_in(Activation_TSDSUW_to_TPULIHM6), .Activ
ation_Tsdsl17_in(Activation_TSDSUW_to_TPULIHM7), .Activation_Tsdsl18_in(Activation_TSDSUW_to_TPULIHM8),
    .Activation_Tsdsl19_in(Activation_TSDSUW_to_TPULIHM9), .Activation_Tsdsl110_in(Activation_TSDSUW_to_TPULIHM1
0), .Activation_Tsdsl111_in(Activation_TSDSUW_to_TPULIHM11), .Activation_Tsdsl112_in(Activation_TSDSUW_to_TPULIHM
12), .Activation_Tsdsl113_in(Activation_TSDSUW_to_TPULIHM13), .Activation_Tsdsl114_in(Activation_TSDSUW_to_TPULIH
M14), .Activation_Tsdsl115_in(Activation_TSDSUW_to_TPULIHM15), .Activation_Tsdsl116_in(Activation_TSDSUW_to_TPULI
HM16),
    .Activation_L1HM_out1(Activation_L1HM_to_L2HM1), .Activation_L1HM_out2(Activation_L1HM_to_L2HM2), .Activation_
L1HM_out3(Activation_L1HM_to_L2HM3), .Activation_L1HM_out4(Activation_L1HM_to_L2HM4), .Activation_L1HM_out5(Acti
vation_L1HM_to_L2HM5), .Activation_L1HM_out6(Activation_L1HM_to_L2HM6), .Activation_L1HM_out7(Activation_L1HM_to
_L2HM7), .Activation_L1HM_out8(Activation_L1HM_to_L2HM8),
    .Activation_L1HM_out9(Activation_L1HM_to_L2HM9), .Activation_L1HM_out10(Activation_L1HM_to_L2HM10), .Activatio
n_L1HM_out11(Activation_L1HM_to_L2HM11), .Activation_L1HM_out12(Activation_L1HM_to_L2HM12), .Activation_L1HM_o
ut13(Activation_L1HM_to_L2HM13), .Activation_L1HM_out14(Activation_L1HM_to_L2HM14), .Activation_L1HM_out15(Activat
ion_L1HM_to_L2HM15), .Activation_L1HM_out16(Activation_L1HM_to_L2HM16),
    .L1HW_counter_rst( TPU_L1HW_counter_rst), .tiled_computing_sig(tiled_computing_sig),
    .Computation_finished_signal(TPU_Computation_finished_signal), .clk(clk), .rst(rst));

    L2_Controller L2Contoroller1(
    .L2HM_Weight_rd(L2HM_Weight_rd), .L2HM_Weight_counter_rst(L2HM_Weight_rd_counter_rst), // for L2 Host Memory
Weight
    .L2HM_Activation_rd(L2HM_Activation_rd), .L2HM_Activation_wr(L2HM_Activation_wr), .L2HM_Activation_rd_counter_rs
t(L2HM_Activation_rd_counter_rst), .L2HM_Activation_wr_counter_rst(L2HM_Activation_wr_counter_rst), // for L2 Host Memory
Activation
    .matrix_compute_size(matrix_compute_size), .tiled_computing_sig(tiled_computing_sig), .zerofill_sig(zerofill_sig), // for both
W&A TSDSU, tiled_computing_sig also for L2 Accumulator
    .TSDSUW_rd(TSDSUW_rd), .TSDSUW_wr(TSDSUW_wr), .TSDSUW_rst(TSDSUW_rst), // for TSDSUW
    .TSDSUA_rd(TSDSUA_rd), .TSDSUA_wr(TSDSUA_wr), .TSDSUA_rst(TSDSUA_rst), // for TSDSUA
    .L2HM_ISA_add(L2HM_ISA_addr), .L2HM_ISA_rd(L2HWISA_rd), .L2HM_ISA_wr(), .L2HM_ISA_finished_sig(L2HM_IS
A_finished_sig), // for L2 Host Memory ISA
    .L2IR_out(L2IR_out), .L2IR_rd(L2IR_rd), .L2IR_wr(), // for L2 IR Memory (stored ISA for L2_Controller)
    .L2IR_inc(L2IR_inc), .L2IR_clr(L2IR_clr), // for L2 IR counter
    .L1_Controller_start_sig(L1_Controller_start_sig), .L1IR_addr(L1IR_addr), .L1IR_wr(L1IR_wr), .TPU_Weight_Tsdsl1_wr(TPU
_Weight_Tsdsl1_wr), .TPU_Activation_Tsdsl1_wr(TPU_Activation_Tsdsl1_wr), .TPU_Activation_L1HM_rd( TPU_Activation_L1H
M_rd), .TPU_L1HW_counter_rst( TPU_L1HW_counter_rst), // for TPU4x4
    .TPU_Computation_finished_signal(TPU_Computation_finished_signal), // for TPU4x4
    .clk(clk), .rst(rst)
    );

endmodule
```

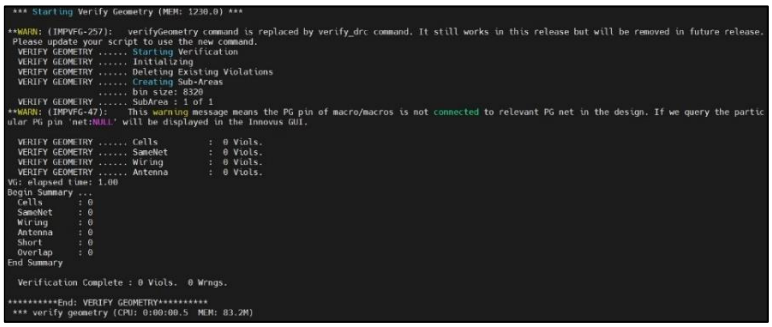
Supplementary Figure 25: Tiled Matrix Multiplication Unit Top Module.

Timing Report and Geometry/Connectivity Verification Results from Cadence Innovus for Each Component:

*The results are provided for reference only. Detailed understanding of the timing reports and violation analysis will require further learning and exploration in future work.



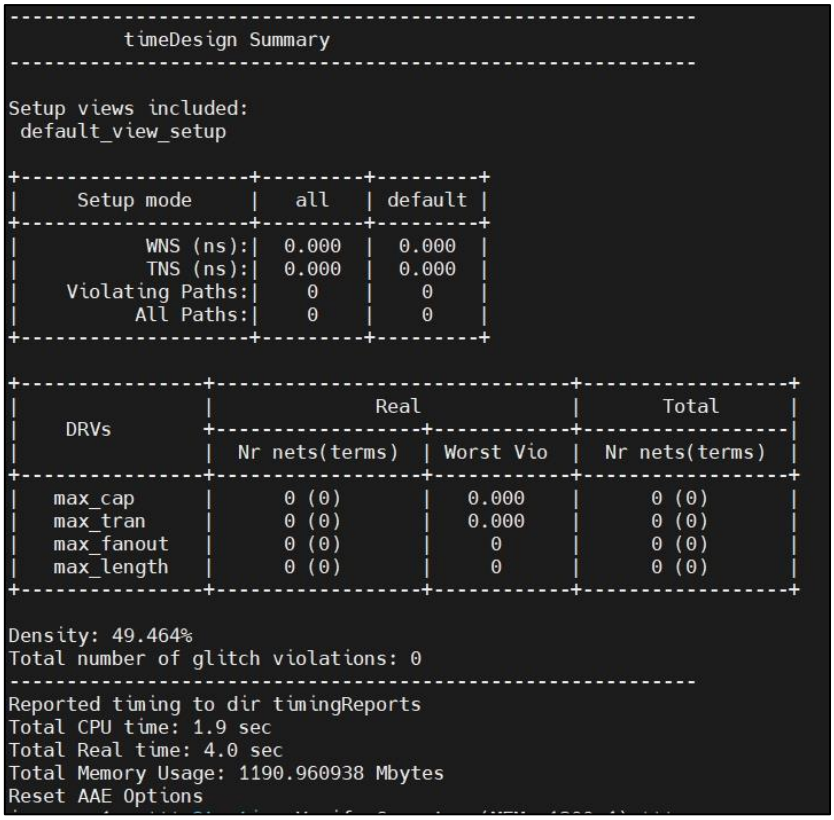
```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.928%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 2.74 sec
Total Real time: 4.0 sec
Total Memory Usage: 1211.9375 Mbytes
Reset AAE Options
```



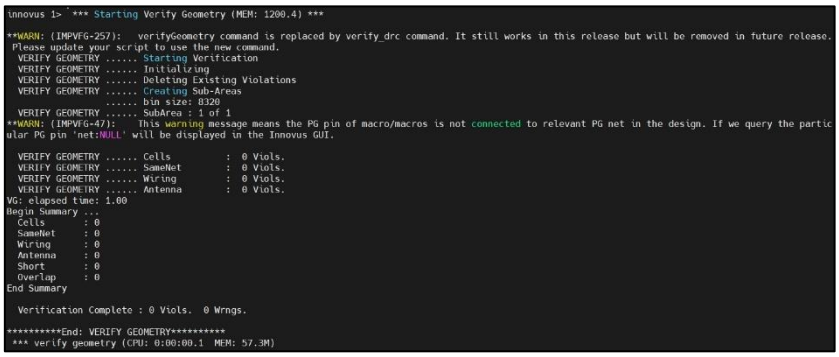
```
*** Starting Verify Geometry (MEM: 1230.0) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                        bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells         : 0 Viols.
  VERIFY GEOMETRY ..... SameNet       : 0 Viols.
  VERIFY GEOMETRY ..... Wiring        : 0 Viols.
  VERIFY GEOMETRY ..... Antenna       : 0 Viols.
VG: elapsed time: 1.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

*********End: VERIFY GEOMETRY*********
 *** verify geometry (CPU: 0:00:00.5  MEM: 83.2M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 17:12:52 2025

Design Name: L2_IR_Mem
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (314.8200, 297.3600)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 17:12:52 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.1  MEM: 0.000M)
```

Supplementary Figure 26: Level 2 IR Memory.



```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.464%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 1.9 sec
Total Real time: 4.0 sec
Total Memory Usage: 1190.960938 Mbytes
Reset AAE Options
```



```
innovus 1> *** Starting Verify Geometry (MEM: 1200.4) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 1.00
Begin Summary ...
  Cells      : 0
  SameNet    : 0
  Wiring     : 0
  Antenna    : 0
  Short      : 0
  Overlap    : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.1  MEM: 57.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 17:35:44 2025

Design Name: L2_IR_counter
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (67.3200, 60.4800)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 17:35:44 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 27: Level 2 IR Counter.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.982%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 9.87 sec
Total Real time: 11.0 sec
Total Memory Usage: 1293.9375 Mbytes
Reset AAE Options
```

```
innovus 1> *** Starting Verify Geometry (MEM: 1303.4) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          : 0 Viols.
  VERIFY GEOMETRY ..... SameNet        : 0 Viols.
  VERIFY GEOMETRY ..... Wiring         : 0 Viols.
  VERIFY GEOMETRY ..... Antenna        : 0 Viols.
VG: elapsed time: 4.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:03.7  MEM: 313.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 18:12:03 2025

Design Name: L2_Host_Memory_InstructionSet
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (834.9000, 821.5200)
Error Limit = 1000; Warning Limit = 50
Check all nets
**** 18:12:03 **** Processed 5000 nets.

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 18:12:04 2025
Time Elapsed: 0:00:01.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.4  MEM: 0.000M)
```

Supplementary Figure 28: Level 2 Memory for Instruction Set.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.996%
Total number of glitch violations: 0
------------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 5.67 sec
Total Real time: 7.0 sec
Total Memory Usage: 1248.328125 Mbytes
Reset AAE Options
```

```
innovus 1> *** Starting Verify Geometry (MEM: 1257.8) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 2.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:01.9  MEM: 185.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 18:34:21 2025

Design Name: L2_Host_Memory_Weight
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (614.4600, 599.7600)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 18:34:21 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.2  MEM: 0.000M)
```

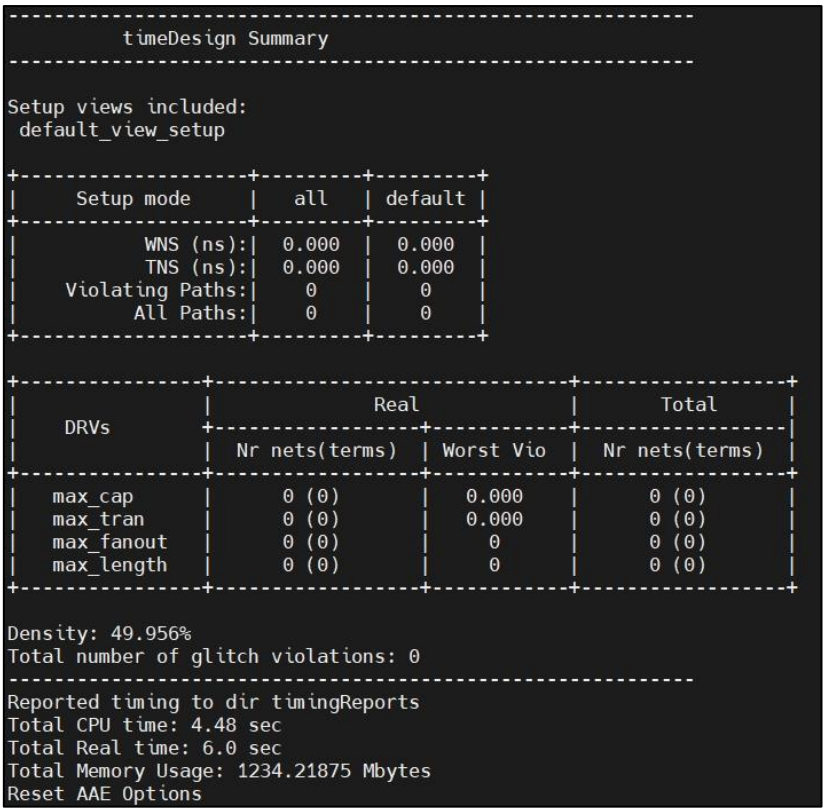
Supplementary Figure 29: Level 2 Memory for Weight.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.986%
Total number of glitch violations: 0
------------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 5.18 sec
Total Real time: 7.0 sec
Total Memory Usage: 1250.859375 Mbytes
Reset AAE Options
```

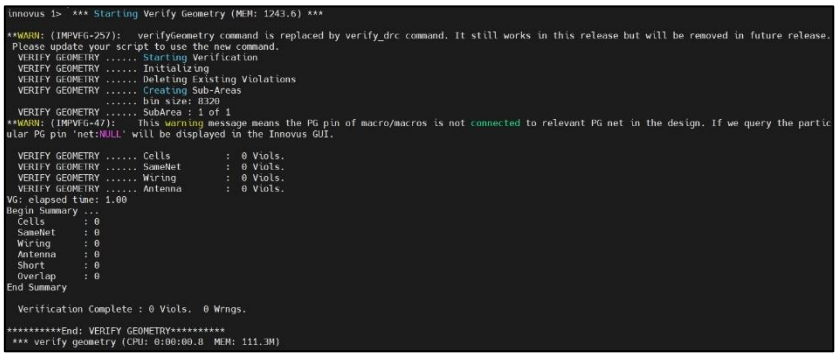
```
innovus 1>  *** Starting Verify Geometry (MEM: 1260.3) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                        ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 2.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:01.9  MEM: 181.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 18:48:54 2025

Design Name: L2_Host_Memory_Activation
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (613.1400, 594.7200)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 18:48:54 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.2  MEM: 0.000M)
```

Supplementary Figure 30: Level 2 Memory for Activation.



```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.956%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 4.48 sec
Total Real time: 6.0 sec
Total Memory Usage: 1234.21875 Mbytes
Reset AAE Options
```



```
innovus 1> *** Starting Verify Geometry (MEM: 1243.6) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                        bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):   This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 1.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.8  MEM: 111.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 19:00:19 2025

Design Name: Tiled_SDS_Unit
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (408.5400, 398.1600)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 19:00:19 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.1  MEM: 0.000M)
```

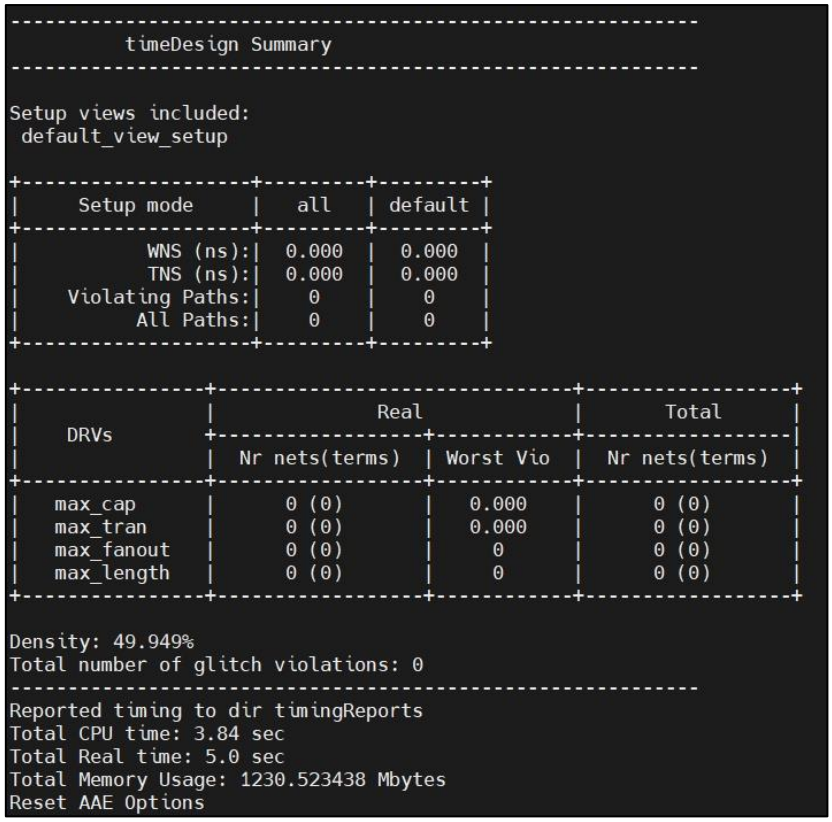
Supplementary Figure 31: Tiled Systolic Data Setup Unit.

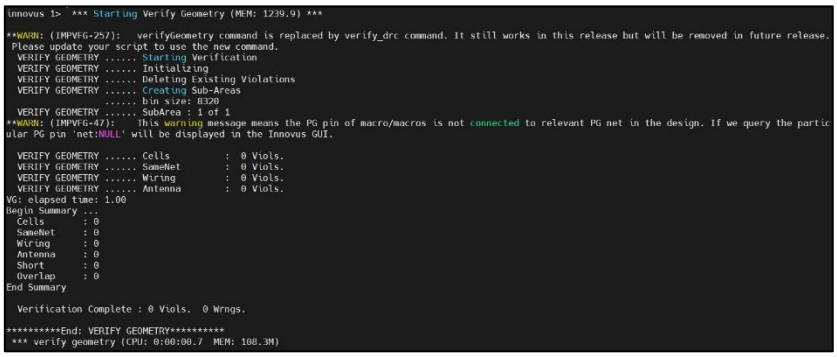
```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.953%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 2.63 sec
Total Real time: 4.0 sec
Total Memory Usage: 1215.257812 Mbytes
Reset AAE Options
```

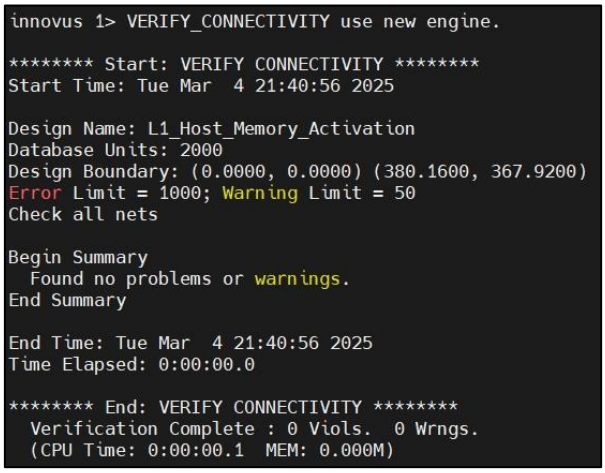
```
innovus 1> *** Starting Verify Geometry (MEM: 1224.7) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.4  MEM: 82.9M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 21:17:28 2025

Design Name: L1_Host_Memory_Weight
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (289.7400, 282.2400)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 21:17:28 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.1  MEM: 0.000M)
```

Supplementary Figure 32: Level 1 Host Memory for Weight.







Supplementary Figure 33: Level 1 Host Memory for Activation.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.977%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 3.19 sec
Total Real time: 4.0 sec
Total Memory Usage: 1223.257812 Mbytes
Reset AAE Options
```

```
innovus 1>  *** Starting Verify Geometry (MEM: 1232.7) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                        ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells         :  0 Viols.
  VERIFY GEOMETRY ..... SameNet       :  0 Viols.
  VERIFY GEOMETRY ..... Wiring        :  0 Viols.
  VERIFY GEOMETRY ..... Antenna       :  0 Viols.
VG: elapsed time: 1.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.9  MEM: 119.9M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 21:59:43 2025

Design Name: IR_Mem
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (430.9800, 418.3200)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 21:59:44 2025
Time Elapsed: 0:00:01.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.1  MEM: 0.000M)
```

Supplementary Figure 34: IR Memory.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.464%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 1.8 sec
Total Real time: 3.0 sec
Total Memory Usage: 1198.84375 Mbytes
Reset AAE Options
```

```
innovus 1> *** Starting Verify Geometry (MEM: 1209.3) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                        ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.1  MEM: 57.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 22:13:11 2025

Design Name: IR_counter
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (67.3200, 60.4800)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 22:13:11 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 35: IR Counter.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.952%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 2.18 sec
Total Real time: 3.0 sec
Total Memory Usage: 1204.507812 Mbytes
Reset AAE Options
```

```
innovus 1>  *** Starting Verify Geometry (MEM: 1213.9) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells      : 0
  SameNet    : 0
  Wiring     : 0
  Antenna    : 0
  Short      : 0
  Overlap    : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.2  MEM: 69.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 22:24:45 2025

Design Name: Weight_DDR3
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (227.7000, 221.7600)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 22:24:45 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

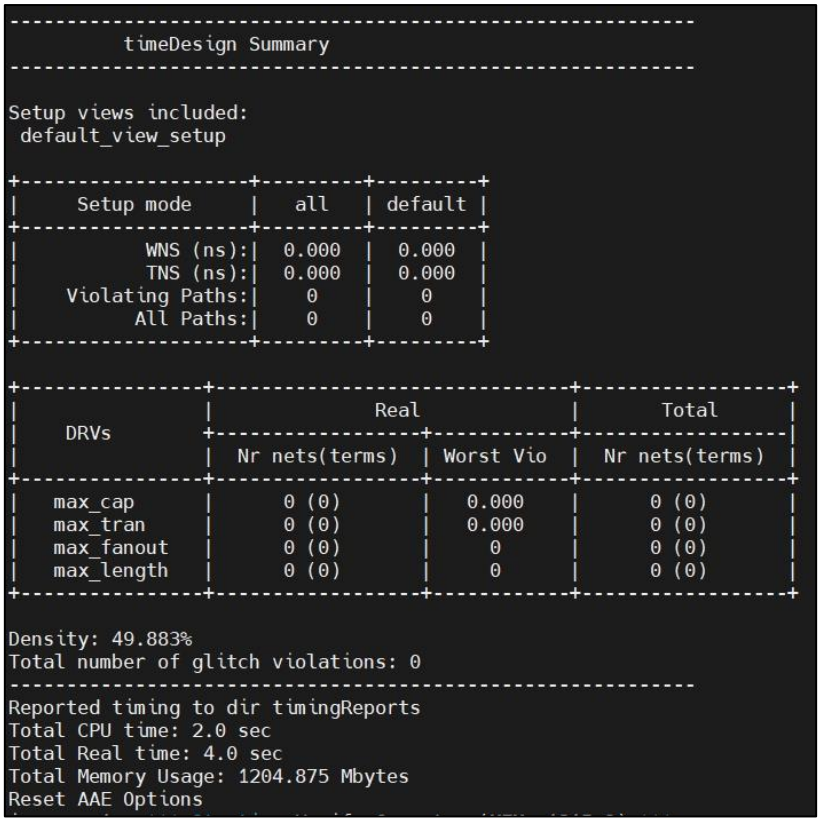
Supplementary Figure 36: Weight DDR3.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.866%
Total number of glitch violations: 0
------------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 2.0 sec
Total Real time: 3.0 sec
Total Memory Usage: 1204.164062 Mbytes
Reset AAE Options
```

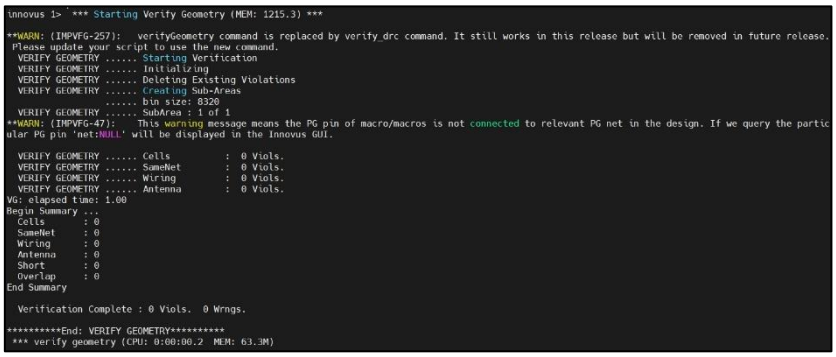
```
innovus 1> *** Starting Verify Geometry (MEM: 1213.6) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          : 0 Viols.
  VERIFY GEOMETRY ..... SameNet        : 0 Viols.
  VERIFY GEOMETRY ..... Wiring         : 0 Viols.
  VERIFY GEOMETRY ..... Antenna        : 0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells      : 0
  SameNet    : 0
  Wiring     : 0
  Antenna    : 0
  Short      : 0
  Overlap    : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.2  MEM: 64.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 22:49:26 2025

Design Name: Weight_interface
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (200.6400, 186.4800)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 22:49:26 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 37: Weight Interface.



```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|         All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.883%
Total number of glitch violations: 0
------------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 2.0 sec
Total Real time: 4.0 sec
Total Memory Usage: 1204.875 Mbytes
Reset AAE Options
```



```
innovus 1> *** Starting Verify Geometry (MEM: 1215.3) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):   This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells         :  0 Viols.
  VERIFY GEOMETRY ..... SameNet       :  0 Viols.
  VERIFY GEOMETRY ..... Wiring        :  0 Viols.
  VERIFY GEOMETRY ..... Antenna       :  0 Viols.
VG: elapsed time: 1.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.2  MEM: 63.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 22:59:42 2025

Design Name: Weight_FIFO
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (193.3800, 176.4000)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 22:59:42 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 38: Weight FIFO.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.242%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 1.79 sec
Total Real time: 3.0 sec
Total Memory Usage: 1193.703125 Mbytes
Reset AAE Options
```

```
innovus 1> *** Starting Verify Geometry (MEM: 1204.1) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):   This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells         :  0 Viols.
  VERIFY GEOMETRY ..... SameNet       :  0 Viols.
  VERIFY GEOMETRY ..... Wiring        :  0 Viols.
  VERIFY GEOMETRY ..... Antenna       :  0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.0  MEM: 58.9M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 23:15:16 2025

Design Name: row_detector
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (50.1600, 35.2800)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 23:15:16 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 39: Row Detector.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.888%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 1.98 sec
Total Real time: 3.0 sec
Total Memory Usage: 1200.9375 Mbytes
Reset AAE Options
```

```
innovus 1> *** Starting Verify Geometry (MEM: 1211.4) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):   This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.1  MEM: 60.9M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 23:25:10 2025

Design Name: systolic_cell
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (163.0200, 156.2400)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 23:25:10 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 40: Systolic Cell.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.952%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 4.83 sec
Total Real time: 5.0 sec
Total Memory Usage: 1245.304688 Mbytes
Reset AAE Options
```

```
innovus 1> *** Starting Verify Geometry (MEM: 1254.7) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  1 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 1.00
Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 1
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

  Verification Complete : 1 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:01.5  MEM: 162.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 23:35:12 2025

Design Name: MMU4x4
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (574.2000, 559.4400)
Error Limit = 1000; Warning Limit = 50
Check all nets
**** 23:35:12 **** Processed 5000 nets.

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 23:35:12 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.2  MEM: 0.000M)
```

Supplementary Figure 41: Matrix Multiplication Unit.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.890%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 2.25 sec
Total Real time: 4.0 sec
Total Memory Usage: 1213.039062 Mbytes
Reset AAE Options
```

```
innovus 1>  *** Starting Verify Geometry (MEM: 1222.5) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells         :  0 Viols.
  VERIFY GEOMETRY ..... SameNet       :  0 Viols.
  VERIFY GEOMETRY ..... Wiring        :  0 Viols.
  VERIFY GEOMETRY ..... Antenna       :  0 Viols.
VG: elapsed time: 1.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.3  MEM: 70.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Mar  4 23:50:06 2025

Design Name: unified_buffer
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (231.0000, 221.7600)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Mar  4 23:50:06 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 42: Unified Buffer.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.871%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 1.96 sec
Total Real time: 3.0 sec
Total Memory Usage: 1200.203125 Mbytes
Reset AAE Options
```

```
innovus 1> *** Starting Verify Geometry (MEM: 1210.8) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):   This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells         :  0 Viols.
  VERIFY GEOMETRY ..... SameNet       :  0 Viols.
  VERIFY GEOMETRY ..... Wiring        :  0 Viols.
  VERIFY GEOMETRY ..... Antenna       :  0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.1  MEM: 57.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Wed Mar  5 00:01:52 2025

Design Name: sds4x4
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (126.0600, 105.8400)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Wed Mar  5 00:01:52 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 43: Systolic Data Setup Unit.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +-----------------+-------------+------------------|
|                |  Nr nets(terms) | Worst Vio   |  Nr nets(terms)  |
+----------------+-----------------+-------------+------------------+
|   max_cap      |      0 (0)      |   0.000     |      0 (0)       |
|   max_tran     |      0 (0)      |   0.000     |      0 (0)       |
|   max_fanout   |      0 (0)      |     0       |      0 (0)       |
|   max_length   |      0 (0)      |     0       |      0 (0)       |
+----------------+-----------------+-------------+------------------+

Density: 49.883%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 2.05 sec
Total Real time: 4.0 sec
Total Memory Usage: 1204.867188 Mbytes
Reset AAE Options
```

```
innovus 1>  *** Starting Verify Geometry (MEM: 1214.3) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):   This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells         : 0 Viols.
  VERIFY GEOMETRY ..... SameNet       : 0 Viols.
  VERIFY GEOMETRY ..... Wiring        : 0 Viols.
  VERIFY GEOMETRY ..... Antenna       : 0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.2  MEM: 64.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Wed Mar  5 00:13:59 2025

Design Name: Activation_FIFO
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (193.3800, 176.4000)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Wed Mar  5 00:13:59 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 44: Activation FIFO.

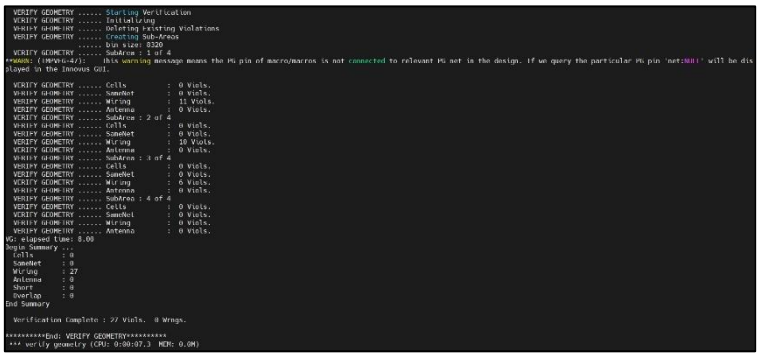
```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.995%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 9.7 sec
Total Real time: 10.0 sec
Total Memory Usage: 1292.09375 Mbytes
Reset AAE Options
```

```
innovus 1>  *** Starting Verify Geometry (MEM: 1301.5) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
 Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                        ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the partic
ular PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells         : 0 Viols.
  VERIFY GEOMETRY ..... SameNet       : 0 Viols.
  VERIFY GEOMETRY ..... Wiring        : 0 Viols.
  VERIFY GEOMETRY ..... Antenna       : 0 Viols.
VG: elapsed time: 3.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY*********
 *** verify geometry (CPU: 0:00:03.0  MEM: 268.8M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Wed Mar  5 00:41:45 2025

Design Name: Accumulator4x4
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (741.8400, 730.8000)
Error Limit = 1000; Warning Limit = 50
Check all nets
**** 00:41:46 **** Processed 5000 nets.

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Wed Mar  5 00:41:46 2025
Time Elapsed: 0:00:01.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.4  MEM: 0.000M)
```

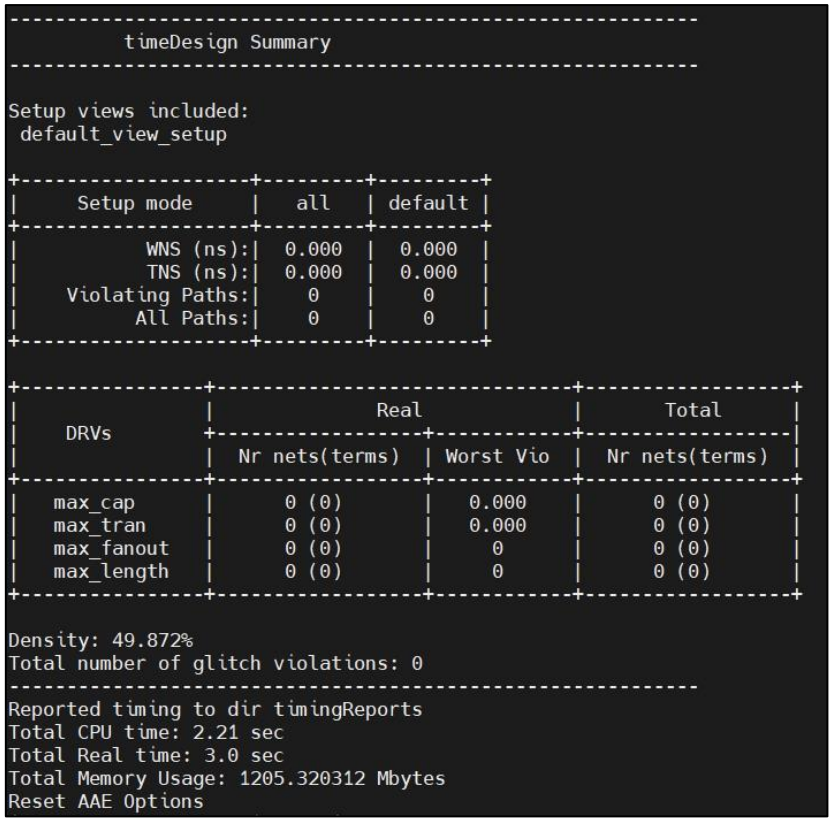
Supplementary Figure 45: Accumulator.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.987%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 23.96 sec
Total Real time: 24.0 sec
Total Memory Usage: 1433.429688 Mbytes
Reset AAE Options
```

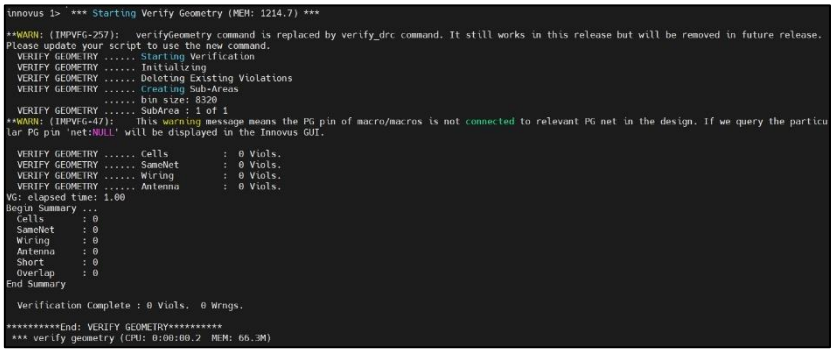



```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Wed Mar  5 14:25:50 2025

Design Name: Activation_Normalization_Unit4x4
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (1201.2000, 1194.4800)
Error Limit = 1000; Warning Limit = 50
Check all nets
**** 14:25:50 **** Processed 5000 nets.
**** 14:25:50 **** Processed 10000 nets.
**** 14:25:50 **** Processed 15000 nets.
**** 14:25:51 **** Processed 20000 nets.
**** 14:25:51 **** Processed 25000 nets.

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Wed Mar  5 14:25:51 2025
Time Elapsed: 0:00:01.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:01.2  MEM: 0.000M)
```

Supplementary Figure 46: Activation & Normalization Unit.



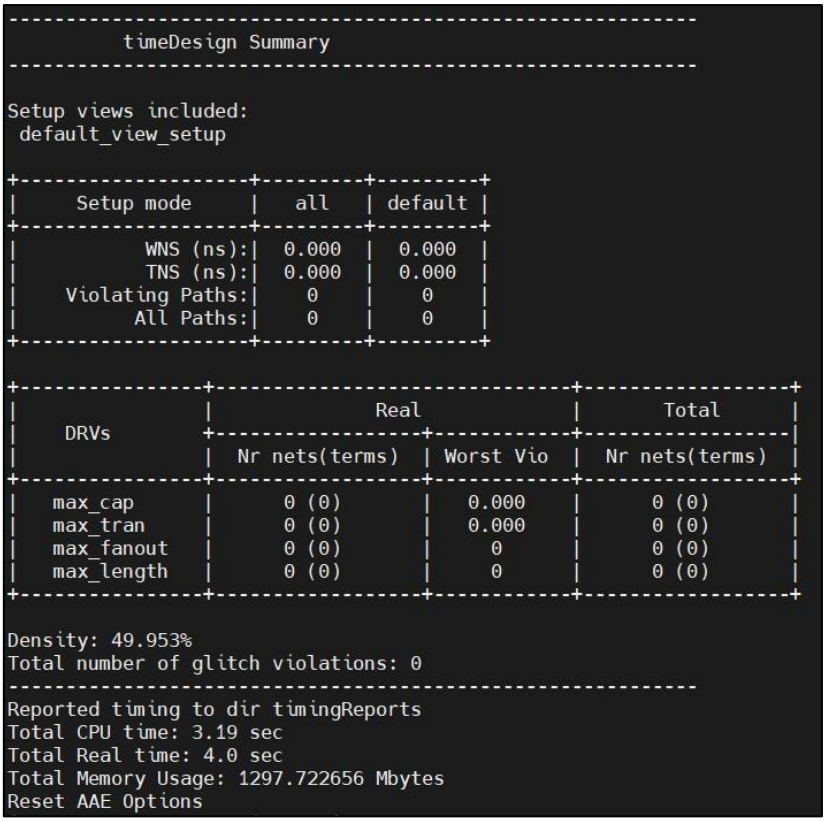
```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 49.872%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 2.21 sec
Total Real time: 3.0 sec
Total Memory Usage: 1205.320312 Mbytes
Reset AAE Options
```

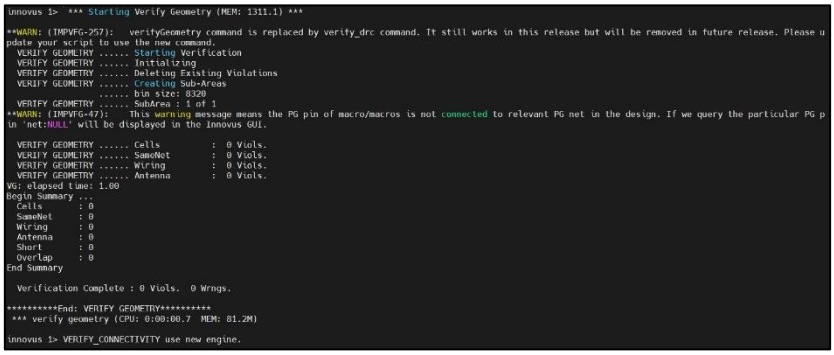


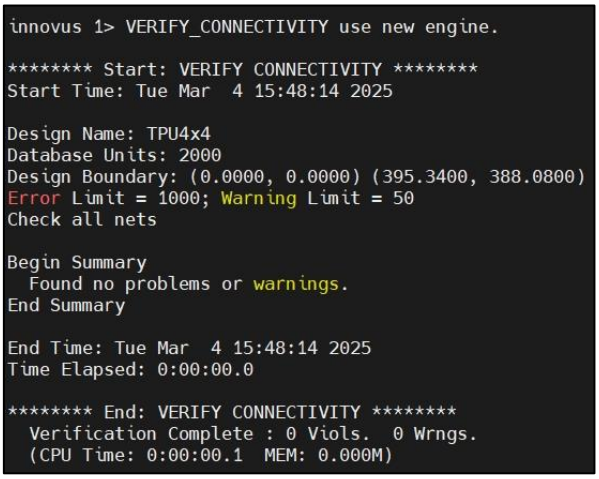
```
innovus 1> *** Starting Verify Geometry (MEM: 1214.7) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                        ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the particu
lar PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          : 0 Viols.
  VERIFY GEOMETRY ..... SameNet        : 0 Viols.
  VERIFY GEOMETRY ..... Wiring         : 0 Viols.
  VERIFY GEOMETRY ..... Antenna        : 0 Viols.
VG: elapsed time: 1.00
Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 0
  Antenna   : 0
  Short     : 0
  Overlap   : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.2  MEM: 66.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Wed Mar  5 14:58:44 2025

Design Name: Controller
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (203.2800, 191.5200)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Wed Mar  5 14:58:44 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 47: TPU Controller.







Supplementary Figure 48: Tensor Processing Unit (TPU) Top Module — Excluding Level 1 Host Memories and Featuring a Reduced AN_Unit.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 default_view_setup

+--------------------+---------+---------+
|     Setup mode     |   all   | default |
+--------------------+---------+---------+
|           WNS (ns):|  0.000  |  0.000  |
|           TNS (ns):|  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |
|          All Paths:|    0    |    0    |
+--------------------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 50.000%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports
Total CPU time: 1.85 sec
Total Real time: 3.0 sec
Total Memory Usage: 1200.375 Mbytes
Reset AAE Options
```

```
innovus 1> *** Starting Verify Geometry (MEM: 1210.8) ***

**WARN: (IMPVFG-257):   verifyGeometry command is replaced by verify_drc command. It still works in this release but will be removed in future release.
Please update your script to use the new command.
  VERIFY GEOMETRY ..... Starting Verification
  VERIFY GEOMETRY ..... Initializing
  VERIFY GEOMETRY ..... Deleting Existing Violations
  VERIFY GEOMETRY ..... Creating Sub-Areas
                  ..... bin size: 8320
  VERIFY GEOMETRY ..... SubArea : 1 of 1
**WARN: (IMPVFG-47):    This warning message means the PG pin of macro/macros is not connected to relevant PG net in the design. If we query the particu
lar PG pin 'net:NULL' will be displayed in the Innovus GUI.

  VERIFY GEOMETRY ..... Cells          :  0 Viols.
  VERIFY GEOMETRY ..... SameNet        :  0 Viols.
  VERIFY GEOMETRY ..... Wiring         :  0 Viols.
  VERIFY GEOMETRY ..... Antenna        :  0 Viols.
VG: elapsed time: 0.00
Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

  Verification Complete : 0 Viols.  0 Wrngs.

**********End: VERIFY GEOMETRY**********
 *** verify geometry (CPU: 0:00:00.1  MEM: 57.3M)
```

```
innovus 1> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Wed Mar  5 15:16:36 2025

Design Name: L2_Controller
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (120.1200, 100.8000)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Wed Mar  5 15:16:36 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.0  MEM: 0.000M)
```

Supplementary Figure 49: Level 2 Controller.