8-2023

Programmable Processing-in-Memory Core and Cluster Design and Verification

Namita Bhosle
nb1453@rit.edu

PROGRAMMABLE PROCESSING-IN-MEMORY CORE AND CLUSTER DESIGN AND VERIFICATION

INCLUDING AUTOMATED TOOLSETS FOR GENERATION OF

USER-DEFINED pPIM CORE AND CLUSTER DESIGN DATABASES

NAMITA BHOSLE

PROGRAMMABLE PROCESSING-IN-MEMORY CORE AND CLUSTER DESIGN AND VERIFICATION

INCLUDING AUTOMATED TOOLSETS FOR GENERATION OF

USER-DEFINED pPIM CORE AND CLUSTER DESIGN DATABASES

by

NAMITA BHOSLE

GRADUATE THESIS

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
AUGUST, 2023

PROGRAMMABLE PROCESSING-IN-MEMORY CORE AND CLUSTER DESIGN AND VERIFICATION

INCLUDING AUTOMATED TOOLSETS FOR GENERATION OF

USER-DEFINED PPIM CORE AND CLUSTER DESIGN DATABASES

NAMITA BHOSLE

Committee Approval:
We, the undersigned committee members, certify that Namita Bhosle has completed the requirements for the Master of Science degree in Electrical Engineering.

Mr. Mark A. Indovina, *Graduate Research Advisor* Date: 14 August 2023
Senior Lecturer, Department of Electrical and Microelectronic Engineering

---

Dr. Amlan Ganguly, *Department Head* Date: 14 August 2023
Professor, Department of Computer Engineering

Dr. Dorin Patru Date: 14 August 2023
Associate Professor, Department of Electrical and Microelectronic Engineering

---

Dr. Daniel B. Phillips Date: 14 August 2023
Associate Professor, Department of Electrical and Microelectronic Engineering

---

Dr. Ferat Sahin, *Department Head* Date: 14 August 2023
Professor, Department of Electrical and Microelectronic Engineering

Dedication

To my loving family, dear friends and mentors.

Your encouragement, kindness, and guidance have been the driving forces throughout my journey. This thesis is a testament to your unwavering belief in my potential. Thank you for everything.

Namita Bhosle

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Namita Bhosle

August, 2023

Acknowledgements

I wish to thank several individuals who have contributed immensely throughout my term at RIT.

Firstly, I wish to convey my sincere and heartfelt gratitude to my advisor Mr. Mark Indovina. Thank you for believing, supporting and guiding me through my graduate studies. I am glad to have worked with you. Your emphasis on developing skills such as good-quality presentation and communication, persistence and perfection have left an indelible mark. These skills would definitely benefit me as I transition through various roles throughout my career.

I wish to thank Dr. Ganguly for his timely feedback and encouragement throughout the course of my thesis. I am grateful for his support and guidance. I express my gratitude to Dr. Patru and Dr. Philip for their suggestions. A special thanks goes to Purab Sutradhar for sharing valuable insights.

I would take a moment to thank my friends that I met here at RIT– Prajakta, Rutvi, and Tushar for the fond memories, and being there for me. I would also like to extend my thanks to my fellow Teaching Assistants, ICE lab mates, and my DS1 students for fun times at RIT that I will cherish forever. Finally, I deeply appreciate the unwavering support of my family, my brother Yuvraj and my closest friend Nikhil, who have consistently stood by my side.

My sincerest gratitude to everyone for always looking out for me and wishing the best for me!

Namita Bhosle

Abstract

The surge in demand for semiconductors and AI-driven applications has led to an amplified requirement for swift and efficient semiconductor design production cycles. These shortened cycles, however, pose a challenge as they reduce the time available for complex chip design and verification, consequently increasing the likelihood of producing error-prone chips.

Current research concentrates on utilizing a Lookup Table (LUT) based pPIM (Programmable Processor in Memory) technology to deliver efficient, low-latency, and low-power computation specifically tailored to 4-bit data requirements, ideal for repetitive and data-expensive AI algorithms. This thesis presents the user-defined Generation 2 pPIM, a redesigned and enhanced version of the existing static Generation 1 architecture, offering a scalable, configurable, and fully automated framework to expedite the design, verification, and implementation process.

The user-defined Gen 2 pPIM Cluster, consisting of nine interconnected user-defined Gen 2 pPIM Cores and an included Accumulator, extends the capability to execute complex operations like Multiply-and-Accumulate (MAC). Both the Gen 2 pPIM Core and Gen 2 pPIM Cluster undergo extensive verification and testing. A Python-based suite has been developed to enable user-specific Gen 2 pPIM Core and Gen 2 pPIM Cluster design and verification efficiently, providing an end-to-end automated toolkit catering to the user's specific needs.

All variants of core and cluster design are benchmarked with 28nm, 65nm and 180nm technology libraries and are compared for area, power and timing.

Contents

List of Figures

List of Tables

Glossary

Acronyms

AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
CLI	Command Line Interface
CNN	Convolutional Neural Networks
CRAM	Computational Random Access Memory
DNN	Deep Neural Networks
DPI	Direct Programming Interface
DRAM	Dynamic Random Access Memory
DUT	Design Under Test
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language

IC	Integrated Circuits
IRAM	Intelligent Random Access Memory
LLM	Large Language Model
LUT	Look-up Table
MAC	Multiply-and-Accumulate
ML	Machine Learning
MUX	Multiplexer
NLP	Natural Language Processing
PIM	Processor in Memory
pPIM	Programmable Processor in Memory
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SVA	SystemVerilog Assertions
TLM	Transaction Level Modeling
UVM	Universal Verification Methodology

Chapter 1

Introduction

In the era of the "Silicon Age", semiconductor chips have become indispensable in our daily lives with every increasing demands in computing performance and capacity. However, the outbreak of the COVID-19 pandemic brought a paradigm shift, affecting nearly every aspect of the world. The semiconductor industry was not spared, as the repercussions were evident by the global chip shortage in the latter half of 2020. Although COVID-19 exacerbated the chip shortage, it stemmed from factors including supply-demand imbalance as the demand for electronic devices escalated, the growing intricacies of chip design, and disruption in semiconductor production and distribution. Despite this, due to the surge in remote work and online education, the demand for electronic devices remained strong, even increased. This amplified the scarcity of consumer electronics. The ripple effect of the chip shortage reverberated globally, impacting various industries. This prompted countries to bolster investments to boost chip manufacturing capacity. For instance, the United States passed the CHIPS Act [1] as a proactive measure. It is crucial to acknowledge that increasing investments alone cannot fully resolve the issue. The entire chip production

[1] The CHIPS and Science Act, signed into law on August 9, 2022, by President Joe Biden, is a U.S. federal statute enacted by the 117th United States Congress. This legislation allocates approximately $280 billion in funding to enhance domestic semiconductor research and manufacturing capabilities within the United States.

process, from design to manufacturing and testing, takes months or even years, depending on the complexity of the integrated circuits (ICs).

Design and verification represent pivotal stages in the chip development cycle depicted in Figure 1.1, presenting their own unique challenges. The design methodology serves as the foundation for producing a reliable semiconductors, while effective verification is critical in identifying and rectifying design flaws before production. The shrinking production life cycle leaves less time for verification, increasing the risk of error-prone designs. Modern verification techniques, such as the Universal Verification Methodology (UVM), are designed to handle the complexities of verification effectively. These methods facilitate the use of layered testbench architectures, promote re-usability and maintainability, and support parallel development processes.

The rapid advancements in artificial intelligence (AI) in robotics, autonomous vehicles, and smart home electronics have sparked an escalating demand for state-of-the-art processors that offer exceptional computational power. These processors, commonly referred to as AI chips, play a critical role in accelerating the repetitive, predictable, and independent calculations required by AI algorithms. A promising approach is a Lookup Table (LUT) based Programmable Processor in Memory (pPIM). By employing LUTs as processing units, this architecture enables rapid computations essential for Convolutional Neural Networks (CNN) or Deep Neural Networks (DNN), all while minimizing latency and power consumption. The distinguishing feature of the LUT-based pPIM is its inherent flexibility to adapt its functionality to any arbitrary operation.

This thesis introduces the design and implementation of enhanced pPIM architectures. A primary aim of this thesis is to develop an advanced toolset that automates the generation of user-defined, synthesizable, pPIM cores and pPIM clusters at the Register Transfer Level (RTL), accompanied by a comprehensive UVM testbench with functional and assertion coverage. Moreover, the tools are engineered to facilitate seamless scalability of the pPIM core and its corresponding multilayered testbench, accommodating operand sizes ranging from 2 bits to 8 bits,

thereby allowing for flexible and customized configurations tailored to meet specific application requirements. The integration of automation, optimization techniques, and advanced verification methodologies can significantly reduce design iterations, accelerate time-to-market, and improve overall productivity in chip design process.

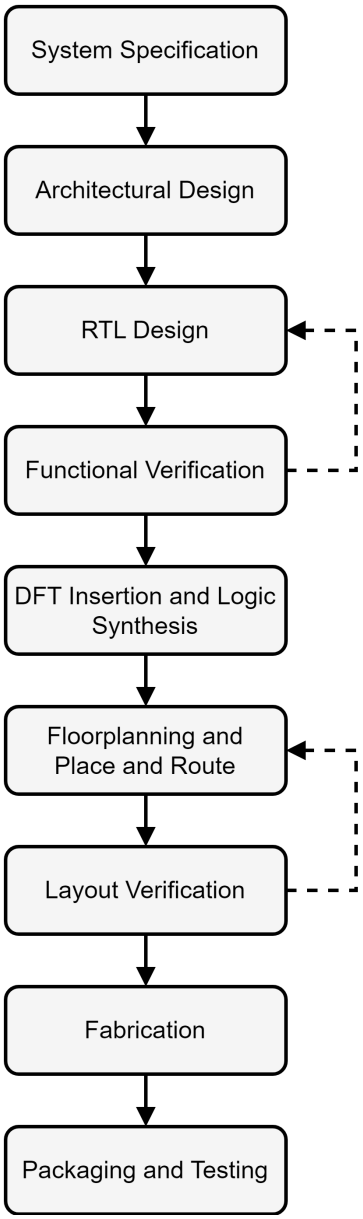

Figure 1.1: Semiconductor Chip Development Process

1.1 Terminology

The key terms used within this thesis work are outlined below:

1. Generation 1 pPIM (Gen 1): Previous pPIM Core and pPIM Cluster architecture.
2. Generation 2 pPIM (Gen 2): Redesigned pPIM Core and pPIM Cluster architecture.
3. Input Data Word for Core: Represented by variables A and B, these are the data inputs specific to individual cores.
4. Input Data Word for Cluster: Referenced as A_CL and B_CL, these are the input data words designated for the cluster.
5. Width of Data Words: The size of operands for the core, denoted as W.
6. LUT Function Words: $2W_{words} \times 2^{2W}_{bitsperword}$. pPIM is programmed using function words. Size of one function word is 2^{2W} and $2W$ such function words are needed to program one LUT core.

1.2 Research Goals

The primary objectives of this research project is the design of the Generation 2 pPIM Core and pPIM Cluster architecture, and to investigate and develop automation tools utilizing Python for the generation of user-defined, synthesizable Gen 2 pPIM Core and Gen 2 pPIM Cluster designs, accompanied by the corresponding UVM testbench to verify their functionality. To accomplish these objectives, the following goals are pursued in the course of this thesis:

1. To acquire a comprehensive understanding of the LUT-based pPIM architecture.

2. To analyze the existing codebase of Gen 1 pPIM Core and pPIM Cluster to identify areas for redesign, enhancement, and optimization.

3. To architect the Gen 2 pPIM Core and develop synthesizable Verilog RTL code for the Gen 2 pPIM Core, adhering to industry standards and best design practices.

4. To architect the Gen 2 pPIM Cluster and develop synthesizable Verilog RTL code for the Gen 2 pPIM Cluster, adhering to industry standards and best design practices.

5. To research and develop a user-friendly Command Line Interface (CLI) that enables users to effortlessly generate Gen 2 pPIM Core and Gen 2 pPIM Cluster designs based on user input. The CLI offers flexible customization options, such as the ability to select operand widths ranging from 2 to 8 bits, choose a target technology library for synthesis, and specify a preferred module name for the generated designs.

6. To research and develop a comprehensive UVM testbench incorporating constrained-random stimulus generation to effectively validate the functionality of the Gen 2 pPIM Core, specifically for arithmetic operations such as addition, subtraction, multiplication, and division.

7. To research and develop a comprehensive UVM testbench to validate the functionality of the Gen 2 pPIM Cluster, with a special focus on the Multiply-and-Accumulate (MAC) operation.

8. To enhance the command line interface program by automating the generation of testbench code for both Gen 2 pPIM Core and Gen 2 pPIM Cluster based on the user-defined design specifications.

9. To run test scenarios across all design variants of Gen 2 pPIM Core and Gen 2 pPIM

Cluster and collect code coverage, functional and assertion-based coverage data to assess the effectiveness and correctness of the tool-generated designs.

10. To perform RTL and Netlist DFT synthesis for all Gen 2 pPIM variants, with the user-selected technology libraries (such as 28nm, 65nm, and 180nm). Collect and compare DFT full scan test coverage data, as well as area, timing, and power consumption metrics.

1.3 Thesis Contributions

The thesis contributes to the research and development of the LUT-based pPIM processor in the following ways:

1. Architecture and detail design of Gen 2 pPIM Core: The Gen 1 pPIM core has been redesigned to achieve a simpler, more scalable, and synthesizable RTL design. As part of these improvements, the generate statements have been replaced with module instantiation, ensuring enhanced portability across various synthesis tools. Additionally, as part of design improvement, the two feedback multiplexers in the pPIM architecture [3–5] have been removed from the design.

2. Architecture and detail design of Gen 2 pPIM Cluster: The Gen 1 pPIM Cluster has been redesigned resulting in a simplified structure. The updated design enables individual programming of each core within the cluster, and brings the accumulator, and an output register inside the cluster thereby enhancing the cluster's flexibility for performing a wide range of operations. The router is redesigned to allow non-blocking, all-to-all communication between the cores, accumulator and output register.

3. Architecture and detail design of enhanced Testbenches: The UVM testbench includes random testing for addition, subtraction, division and multiplication and is supported with

functional coverage and SystemVerilog Assertions (SVA) to ensure thorough verification of the Gen 2 pPIM core. The UVM testbench for the Gen 2 pPIM Cluster design includes thorough testing, with specific enhancements for MAC operations.

4. User-defined operand widths: The new designs allows for flexible, scalable operand widths, eliminating the previous limitation of only supporting 4-bit inputs.

5. Research and development of Python-Based Design Database Generator Tool Suite: A Python-based command-line interface has been developed to automate the generation of user-defined Gen 2 pPIM pPIM Core and pPIM Cluster designs, along with their verification testbenches. This tool suite reduces manual effort, speeds up the design and verification process, and helps explore different design options more effectively. This tool has proven to enhance design verification accuracy and efficiency by reducing human involvement and errors.

These contributions advance the LUT-based pPIM processor research by enhancing code portability, flexibility, verification capabilities and input scalability.

1.4 Organization

The thesis is organized as follows:

- Chapter 2 explores the previous works related to automatic code generation techniques, Processing-in-Memory architectures, and verification methodologies.

- Chapter 3 presents the new micro-architectures of Gen 2 pPIM Core and Gen 2 pPIM Cluster.

- Chapter 4 outlines the test-plan designed for the verification of Gen 2 pPIM Core.

- Chapter 5 outlines the test-plan designed for the verification of Gen 2 pPIM Cluster.
- Chapter 6 introduces the code generation toolkits along with its user guide for Gen 2 pPIM Core and Gen 2 pPIM Cluster.
- Chapter 7 discusses the verification and synthesis results.
- Chapter 8 provides a concluding summary of the thesis and identifies potential areas for future improvement.
- Chapter I presents schematics and trial layouts of Gen 2 pPIM Core and Gen 2 pPIM Cluster.

Chapter 2

Literature Review

This chapter includes a literature review focusing on topics that are relevant to this thesis such as code generation, processing-in-memory architectures and verification methodologies.

2.1 Automatic Code Generation

Automatic code generation has gained significant attention from researchers and engineers, as it offers a higher level of abstraction by generating code from high-level specifications. Efforts have been made to automate the code-writing process using Natural Language Processing (NLP) and Large Language Models (LLM) for generating Hardware Description Language (HDL) code [6]. However, this research presents challenges in terms of ensuring the generation of syntactically correct and semantically accurate RTL code. Misinterpretations or ambiguous language can lead to erroneous or nonsensical outputs. Prompt engineering on pre-trained LLMs is commonly employed in natural language processing techniques for generating code. However, accurately capturing complex design requirements and constraints in a prompt can be cumbersome. Additionally, extensive fine-tuning of LLMs is necessary to ensure their comprehension of the

hardware design domain and to produce sensible code.

A more reliable and conventional approach to automatic hardware code generation involves the application of scripting languages like Perl, Python [7], or XML formats [8]. These code generators allow users to define their design specifications as input and obtain corresponding RTL code as output. Once the framework is developed, it becomes reusable for generating code for various design specifications. This re-usability enables benchmarking of the same design with different specifications by simply adjusting the input. The code generator tools efficiently leverage repetitive design patterns commonly encountered in Verilog code. For instance, a register module or a multiplexer can be instantiated and reused multiple times in a design, but they only need to be coded once in the tools. Unlike code generated by natural language models, the manually developed and tested code generators provide greater reliability and control over the output.

In essence, code generation is the process of writing code that writes code. The automatic approach to code generation saves development time, albeit requiring an initial investment in developing the code generation framework. Code generators not only reduce coding errors but also provide a systematic and predefined approach, eliminating the need for redundant coding. However, a challenge lies in striking a balance between the flexibility given to the user and the constraints imposed on the generated code to ensure correctness while maintaining design intent.

2.2 Processing-in-memory

Artificial intelligence (AI) and machine learning (ML) (AI/ML) technologies are playing an increasingly vital role in numerous everyday applications. Voice and image recognition, virtual assistants, e-commerce platforms, autonomous vehicles, social media, and health monitoring are just a few examples of these applications that heavily rely on AI/ML for sophisticated data analysis. Growing dependence on mining and processing huge data-sets is placing a substantial

burden on the data storage and resource movement capabilities of today's computers [9]. The transfer of large data chunks between memory subsystems and the processor creates bottlenecks and leads to significant delays. The resulting long latencies negatively impact power consumption, performance efficiency, and system reliability. Traditional computing paradigms were not initially designed to effectively handle such extensive data volumes, prompting the need for a paradigm shift. Processing-in-Memory has garnered significant attention from both academia and industries as it aims to bring computation to the memory, mitigating data movement costs [10]. Processor-in-memory conducts computations and processing directly within the memory array, eliminates the need for extensive data transfers, and offers a promising solution to the existing challenges.

2.2.1 Memory Wall

The foundation of modern computers is largely based on von Neumann's stored-program computer concept, which incorporates separate memory and processing units. In this architecture as shown in Figure 2.2, program and instruction data are stored in the main memory, and the CPU retrieves instructions and data via a common bus and executes them sequentially. However, with the increasing demand for memory-intensive computing due to large datasets with poor data locality, the conventional von Neumann structure faces challenges. The movement of data to and from the main memory introduces significant latency, and memory access times have not kept pace with the rising clock frequencies of modern processors. As a result, there is a growing gap between CPU and memory speeds, commonly referred to as the "memory wall" or "von-Neumann bottleneck," suggesting that execution time is heavily dependent on the speed at which data can be transferred from DRAM to the CPU.

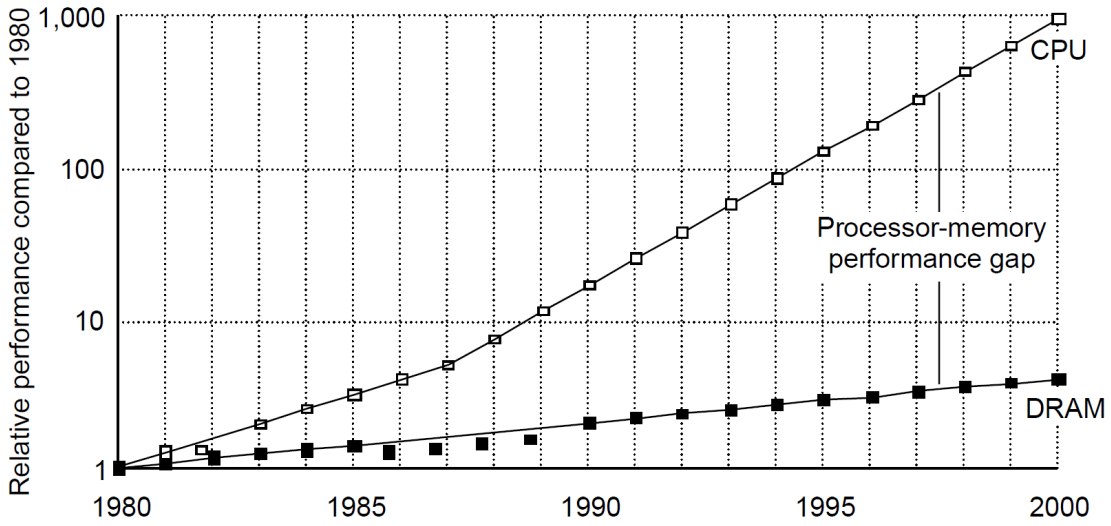

Figure 2.1: Temporal Evolution of Performance Gap: Processor-Memory Request Time (for a single core processor) vs. DRAM Access Latency [1][2]

To address these challenges, efforts have been made to reduce latency by introducing hierarchical cache systems. Caches prefetch frequently accessed instructions and data, aiming to minimize cache misses. However, the effectiveness of caches relies on the existence of temporal or spatial locality in the data. For applications that heavily rely on large datasets, this approach may prove insufficient. Firstly, such datasets require large caches, which can pose practical limitations. Secondly, even if a dataset is cached, and a cache miss occurs, retrieving another dataset from the main memory can result in significant latency, leading to sub-optimal performance. Additionally, the bandwidth constraints of the memory hierarchy limit the efficiency of data movement between different memory levels [11].

Non von-Neumann machines were developed as an alternative to the traditional von Neumann architecture with the goal of addressing performance bottlenecks, by introducing parallelism and multi-threading. Multi-threading focuses on concurrent execution within a single processor core,

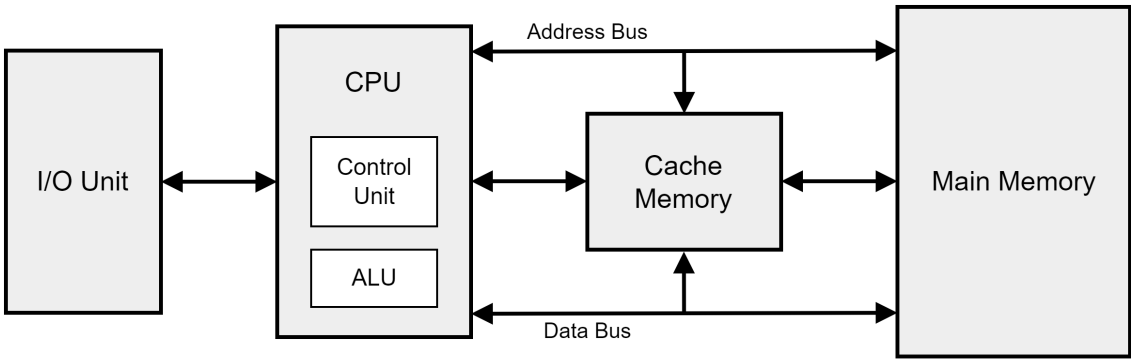

Figure 2.2: Structure of a Standard Cache-organized von-Neumann Machine

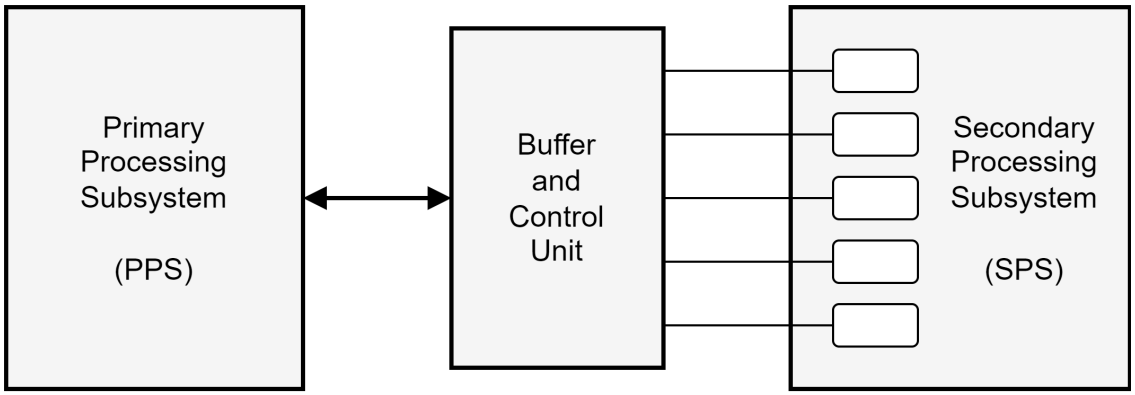

Figure 2.3: Structure of a non von-Neumann Machine

while parallelism involves simultaneous execution across multiple cores or processors. A non von-Neumann architecture consists of a Primary Processing Subsystem (PPS) for internal evaluation of operations and a Secondary Processing Subsystem (SPS) with intelligent mass-storage devices [12]. Refer the Figure 2.3 below. This architecture enables efficient and parallel processing. This combination of subsystems facilitates efficient and parallel processing.

Non von-Neumann architecture pushed Processing-in-Memory further by challenging the conventional separation of processing and memory units, leading to the exploration of new

architectural paradigms. The concept of processor in memory (PIM) emerged as a highly discussed and promising approach in memory-intensive computer architecture. PIM, also known as logic-in-memory or smart-RAM, addresses the limitations by integrating one or more processors directly into high-bandwidth DRAM banks [13]. This integration brings the memory and processing units closer together, significantly reducing the need for frequent data transfers between them. By executing computations within the memory itself, PIM effectively eliminates unnecessary data movements and offers the potential for significant performance improvements in memory-intensive applications.

Section 2.2.2 provides a comprehensive overview of different Processing-in-Memory architectures that have surfaced to date.

2.2.2 Evolution and Advancements in PIM Architectures

The concept of processor in memory (PIM) has a rich history in computer architecture research, with numerous approaches and advancements proposed over the years. One of the earliest ideas can be traced back to the 1960s in paper [14] when researchers postulated the concept of a cache-organized "logic-in-memory" computer in which each sector of cache is dedicated to perform independent logic. This early work laid the foundation for further exploration into PIM architectures.

The case study in 1997 [2, 15] explores the integration of a microprocessor and DRAM on a single chip, called Intelligent RAM (IRAM). It highlights that IRAM offers reduced latency (5-10x), increased bandwidth (50-100x), improved energy efficiency (2-4x), and potential cost savings by eliminating excess memory and minimizing board area. The paper also proposes that IRAM has the potential to improve the performance of vector processors. Vector instructions to access memory in blocks thus relying on larger but slower main memory instead of smaller and faster caches. In this scenario, IRAM can leverage its low latency and high bandwidth to deliver

impressive performance gains.

The late 1990s saw the introduction of Computational RAM (CRAM) [16], a distinctive approach that integrates computation and memory. CRAM can function as a conventional memory chip or a Single-Instruction Multiple-Data (SIMD) stream computer. By aligning processing elements with memory columns and utilizing a common memory row address for each row, the CRAM architecture enables SIMD operations. This unique amalgamation of processing power and memory holds promising potential for enhancing performance and efficiency across diverse domains.

The Active Pages model [17] involves shifting data-intensive computations to the memory system. By offloading data manipulations to the logic embedded within the memory, this approach allows the processor to maintain high speeds while enabling parallel execution.

The Smart Memories chip [18], as proposed by Stanford researchers in the early 2000s, is a modular computer consisting of multiple processing tiles arranged in a quad configuration. Each tile contains its own local memory, a processor core, and an interconnect. The quads are interconnected to facilitate communication among different sets of quads. With its dynamic routing capability, the chip offers flexibility to adapt to a wide range of applications.

Throughout the 1990s and early 2000s, researchers sought to exploit the parallelism and proximity benefits of PIM architectures to overcome memory access limitations. The Programmable Processor in Memory (pPIM) is an innovative look-up table based [4] architecture that introduces further advancements to the area of PIM. It stands out with its ability to flexibly perform a wide range of computations through the reconfiguration of LUT using function words. By using precalculated results stored in LUT, the pPIM architecture accelerates data-intensive computations, resulting in notable performance improvements. The pPIM architecture is discussed in in-depth details in Chapter 3

2.3 Verification Methodologies

With the increasing demand for application-specific ICs (ASICs) and FPGAs (Field Programmable Gate Array), chip complexity is experiencing exponential growth. As a result, functional verification has become an essential aspect of the chip development cycle as shown in 1.1. The primary objective of hardware verification is to ensure that the device accurately reflects its design specification. Given the exorbitant costs of re-spins and additional development time associated with chip failures after tape-out, it is crucial to address logic or functional flaws effectively [19].

To tackle this, verification engineers are adopting advanced verification methodologies that provide higher levels of abstraction, reusability, and scalability. Hierarchical and modular testbench architectures enable the creation of reusable verification components and the efficient verification of larger designs. Advanced verification methodologies, coupled with the White-box and Grey-box testing approaches that mirror the design process, have proven highly effective. Verification engineers independently review the design specification, assess the design's intent, and formulate a verification plan [20]. This approach enhances visibility into the design implementation from the get-go, enabling early bug detection. Discrepancies may arise if certain edge cases are overlooked or if the RTL designer misinterprets the design specification.

By closely aligning with the design process, where designers study the architectural specification and translate it into RTL code, verification engineers can construct a testbench that ensures the implementation matches with their interpretation of the design specification. The iterative cycle of design-verification continues until the interpretations of both design and verification engineers converge, ideally achieving alignment simultaneously.

2.3.1 SystemVerilog for Verification

SystemVerilog is a hardware description and verification language widely used in the field of electronic design automation (EDA). Including the features of well-known Verilog HDL, SystemVerilog is a object-oriented programming language with additional features to support digital design-verification. It offers powerful verification features such as constrained randomization, assertions, and coverage constructs.

A couple of decades ago, when SystemVerilog was gaining popularity as a verification tool, user's choice of verification methodology was tied to specific tool vendor. To address this limitation and promote vendor independence, a committee was formed. Their objective was to develop an open standard methodology that could be used with major tool vendor's offerings. This collaborative effort resulted in the creation of the Universal Verification Methodology (UVM) by Accellera in 2010 [21].

2.3.2 Universal Verification Methodology (UVM)

The Universal Verification Methodology (UVM) is a standardized and comprehensive framework for advanced verification in the semiconductor industry. Built on top of SystemVerilog, UVM provides a set of guidelines, libraries, and methodologies that enhances the verification process. A crucial feature of UVM is its *factory* mechanism, which allows for dynamic object creation and configuration. This mechanism let users to replace or override testbench components, like sequences, with different types of sequences without modifying the testbench code [21].

Modern verification methodologies, such as SystemVerilog and UVM, advocate for a layered testbench structure. Although the layered architecture may appear complex, it actually helps in breaking down the code into smaller, manageable pieces ensuring clear separation of concerns and enabling a systematic verification approach. Each layer within the testbench is responsible

for specific functionalities, such as generating signals, driving them into the DUT, checking DUT responses, and measuring coverage [20]. By keeping the testbench separate from the test cases, the same stimuli can be reused across multiple projects.

In this thesis, the pPIM core and cluster testbenches are developed using UVM. The testbench architecture is thoroughly discussed in Chapters 4 and 5 which provides in-depth insights into the testbench design and implementation.

Chapter 3

pPIM Architecture

This chapter presents detailed explanation of Gen 2 pPIM Core and Gen 2 pPIM Cluster micro-architectures.

In the pPIM architecture proposed in [3–5], a programmable processor in memory is implemented based on Look-up tables. This architecture is specifically designed to efficiently handle memory-intensive calculations within Convolutional Neural Networks, eliminating the need for data movement. Figure 3.1 captures the hierarchy in pPIM components. The fundamental component of this architecture is the pPIM core, which is capable of executing various operations on two 4-bit inputs. A total of nine pPIM cores, are interconnected via a Router at the highest level to form a pPIM cluster. This cluster, residing within the DRAM memory banks, operates on two 8-bit inputs and can execute multiple operations simultaneously. Several such pPIM clusters are strategically positioned between the memory subarrays. The structure repeats along DRAM bank.

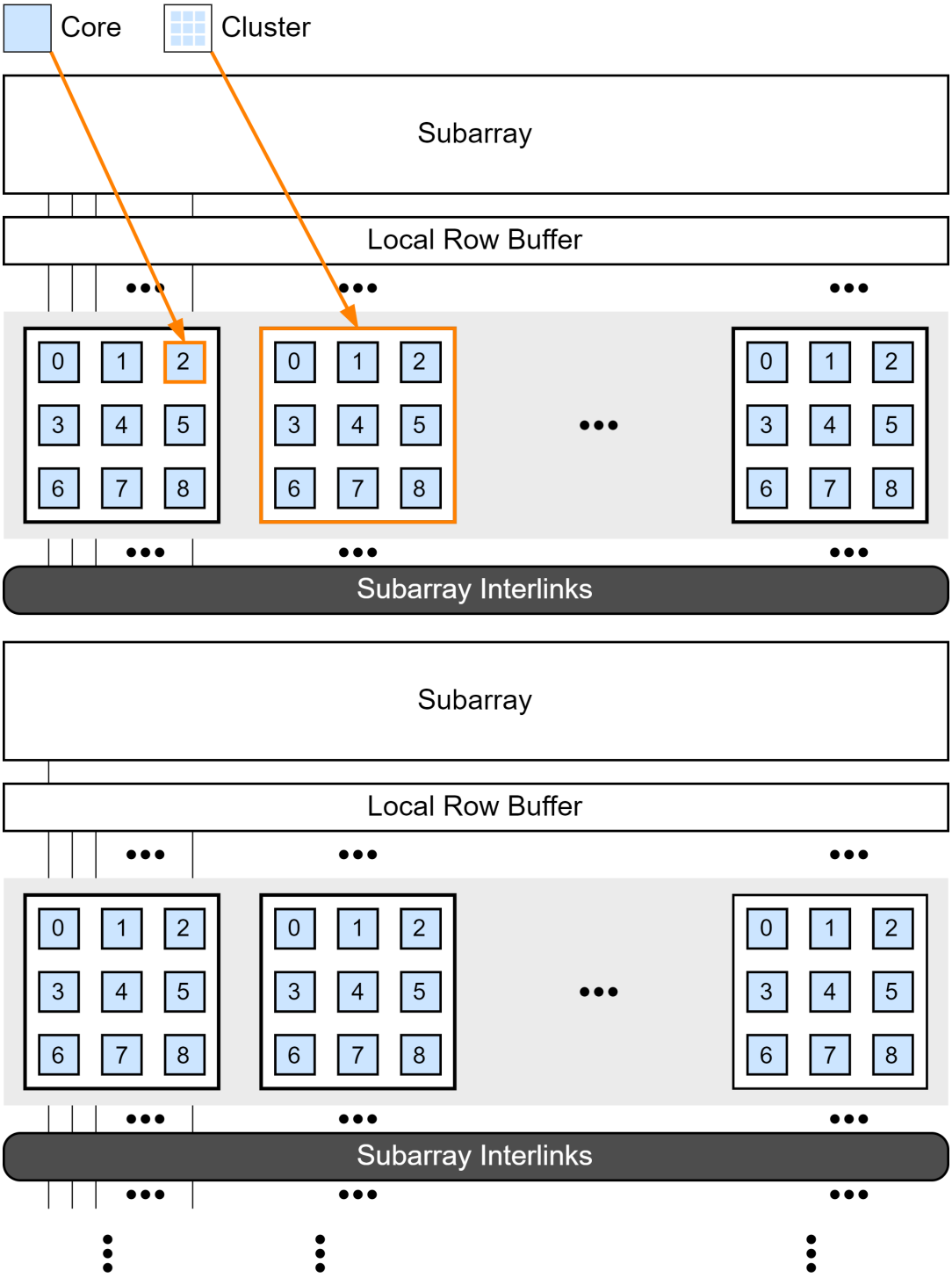

Figure 3.1: Top-level View of the pPIM Architecture [3]

3.1 Generation 2 pPIM Core

The Core serves as the central component of the pPIM architecture. It is designed as a re-programmable lookup table that stores all the potential outcomes for a desired operation in advance. To help understand its functioning, the lookup table can be visualized as a two-dimensional matrix, as depicted in Figure 3.2. With 4-bit data words A and B as inputs, each having 16 possible combinations, a 16x16 matrix is created by arranging A and B as rows and columns, respectively. This matrix comprises 256 entries, representing the 8-bit outputs for the corresponding inputs, denoted as Y. By utilizing the matrix preloaded with all possible outputs for two 4-bit inputs, the data words A and B act as pointers to specific locations within the matrix.

For example, if the intention is to perform a multiplication operation, the matrix would be populated with all possible results of multiplying A by B. If A has a value of 2 and B has a value of 15 (highlighted in Grey), the intersection of the third row and the sixteenth column (considering 0-based indexing) in the matrix would represent the outcome of multiplying $2_{10}(0010_2)$ by $15_{10}(1111_2)$, which is $30_{10}(0001\,1110_2)$ (highlighted in Green).

A \ B (X)	0000	0001	0010	0011	• • •	1111
0000	00000000	00000000	00000000	00000000	• • •	00000000
0001	00000000	00000001	00000010	00000011		00001111
0010	00000000	00000010	00000100	00000110		00011110
0011	00000000	00000011	00000110	00001001		00101101
⋮	⋮					⋮
1111	00000000	11110000	00011110	00101101	• • •	11111111

Figure 3.2: Conceptual Representation of LUT Mapping

The Gen 2 pPIM Core is composed of various components as shown in Figure 3.3 that work together to execute operations based on programmed function word. These components include two input registers holding *data words*, *A* and *B*, an address decoder, a register file containing *function word* registers, and output multiplexers. Function words are pre-programmed with desired operations like addition, subtraction, multiplication, or logical operations. For functions that require a single operand, the two input data words can be concatenated to create a unified input. The operation of the pPIM core is explained as follows:

- The two input registers store the data words used for the operation.
- The address decoder generates unique addresses for accessing function word registers in

the register file.

- The register file holds the function word registers, each containing a portion of the programmed function word.
- The programmed function word is broken down and loaded into the respective function word registers in the register file at specific function addresses.
- The function word registers are then fed as inputs to a series of multiplexers. Hence, switching between functionalities can be easily accomplished by reading a new function word from the register file and feeding it into the MUX inputs.
- To determine the output of the operation, the two input data words are concatenated together to form a select line for the multiplexers. The select line serves as an address to pick the appropriate entry from the look-up table. These multiplexers create the illusion of a look-up table by taking the function word as input and selecting specific bits based on the data words stored in the input registers.
- Each multiplexer outputs one bit of the result based on the selected entry, and all the bits together form the complete output data word.

For instance, consider a pPIM core with 4-bit operands, each operand can have 16 possible combinations, resulting in 256 possible outcomes, each 8-bit long. The size of the each function word would be 256 bits, eight of such function registers reside in register file. The address decoder generates eight unique addresses to access each of these function registers. The function word registers then serve as inputs to the eight multiplexers, each operating as a 256-to-1 multiplexer. The concatenation of two 4-bit data words forms an 8-bit select line for the multiplexers, and the resulting 8-bit output data word is generated by the output multiplexers.

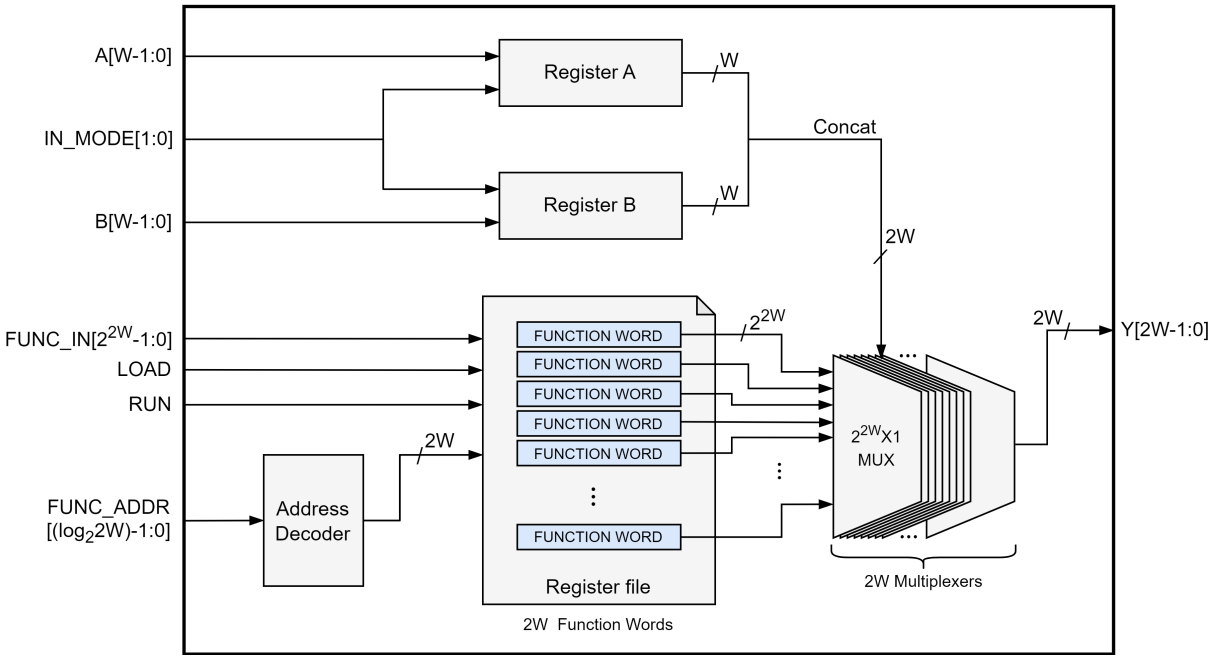

Figure 3.3: Generation 2 pPIM Core Architecture

In this thesis, the Gen 2 pPIM Core architecture is presented and implemented, showcasing its ability to handle operand widths spanning from 2 to 8 bits. The formulae depicted in Figure 3.3 have been derived to enable the scaling of the Gen 2 pPIM Core components. Table 3.1 provides an overview of how the sizes of the pPIM components scale in relation to the width of operands.

Table 3.1: pPIM Core Components Dimensions at Different Operand Widths

Data Words (A, B) (Bits)	**Function Word (Bits)**	**Number of Function Words**	**Function Address (Bits)**	**MUX**	**Number of MUXes**	**Output (Y) (Bits)**
2	16	4	2	16-to-1	4	4
3	64	6	3	64-to-1	6	6
4	256	8	3	256-to-1	8	8
5	1024	10	4	1024-to-1	10	10
6	4096	12	4	4096-to-1	12	12
7	16384	14	4	16384-to-1	14	14
8	65536	16	4	65536-to-1	16	16

Evidently, the size of the LUT exhibits exponential growth with increasing widths of inputs. To ensure the validity and functionality of the designs, they have been implemented as synthesizable units and subjected to functional testing using a corresponding UVM testbench. Chapter 7 delves into a comparative analysis of the area, power, and timing results for each core design.

3.2 Generation 2 pPIM Cluster

A pPIM cluster consists of nine programmable cores that work together to perform complex operations. While a single core can handle basic functions like addition and subtraction, it lacks the capability to perform more advanced tasks such as multiply-and-accumulate, matrix multiplication or polynomial calculations. By breaking down complex operations into simpler

tasks and distributing them among multiple pPIM cores within a cluster, more sophisticated operations can be performed. By combining the capabilities of multiple cores, the cluster can execute complex operations across one or multiple cycles. The cores are connected using a *Router*, which orchestrates the data flow between them. Accumulator inside the cluster stores the intermediate results an operation.

To enhance the performance of data-intensive operations, a pPIM cluster is formed by interconnecting nine pPIM cores. Each core within the cluster is programmable, allowing it to execute different functions independently. The connectivity pattern ensures that each core can communicate with every other core in the cluster, including itself, enabling all-to-all communication. The *Accumulator* inside the cluster is also interconnected in an all-to-all manner as illustrated in the Figure 3.4. This interconnection is achieved using a crossbar architecture, referred to as a router. Router architecture is discussed in section 3.2.1.

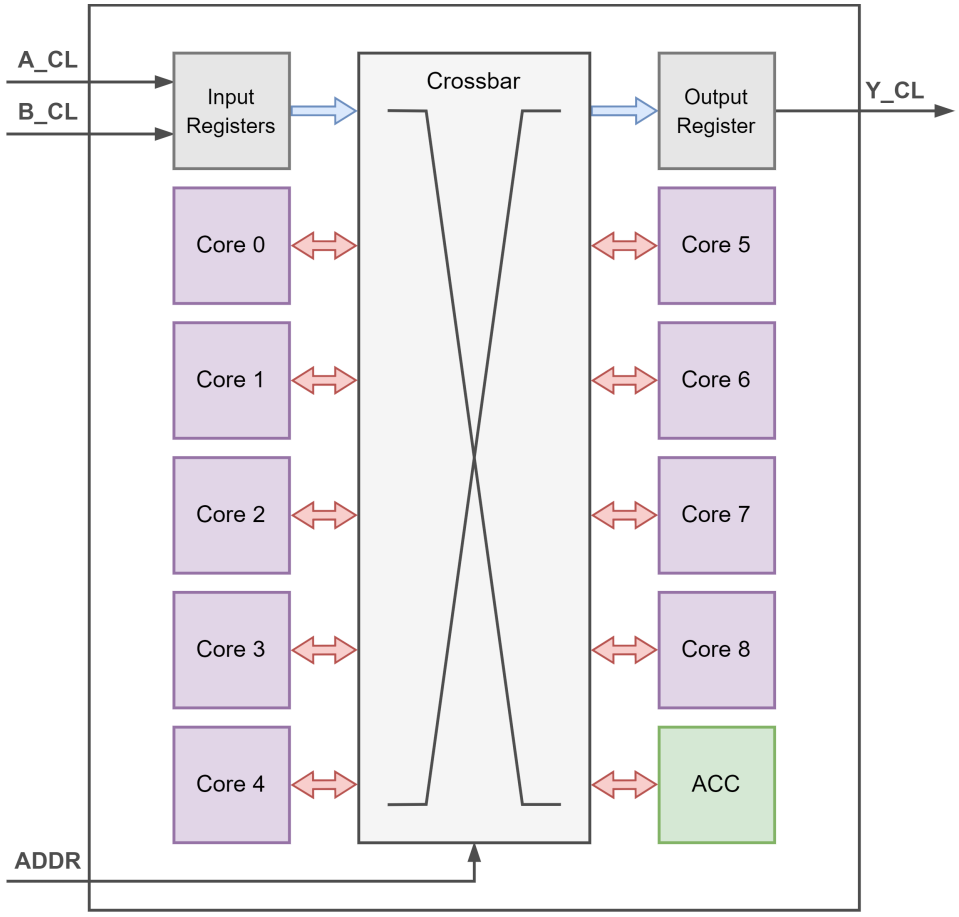

Figure 3.4: Top-level view of Gen 2 pPIM Cluster

At the cluster's top level, there are two input operands, A_CL and B_CL, which are specific to the cluster. The cluster's input resolution is double that of each core, resulting in the cluster output Y_CL being twice the size of the core outputs. To ensure operand width compatibility, the cluster data words are split into higher and lower nibbles, then routed to the cores for further processing.

The output from the each pPIM core ($Y0$, $Y1$, $Y2$, .., $Y8$) is also divided into two halves and routed to the inputs of all nine cores ($A0$, $B0$), ($A1$, $B1$), .., ($A8$, $B8$). The cluster contains an accumulator. The accumulator contents are also rerouted to the core inputs for further utilization. The accumulator is designed to align with the core design specifications. Further details can be found in section 3.2.2.

Finally, the cluster output is captured in an output register. The pPIM cluster's final output can be chosen from the core outputs ($Y0$, $Y1$, $Y2$, .., $Y8$) or from the four halves of the accumulator ($ACC0$, $ACC1$, $ACC2$, $ACC3$).

To illustrate the basic operation of the cluster, consider one of the cluster designs with 8-bit cluster operands. In this case, a cluster performing operations on 8-bit cluster operands will have nine cores, each with 4-bit core inputs. As each core can only accept two 4-bit inputs (A and B), the 8-bit cluster inputs (A_CL and B_CL) are divided into higher and lower nibbles and then routed to the cores for further processing. To ensure compatibility with the input port sizes, the 8-bit output data word (Y) from each core is divided into two 4-bit halves. These cores execute their programmed functions on the selected operands. To store necessary intermediate results, a 16-bit accumulator is available, which is also split into four 4-bit halves and routed to the core inputs. The cluster's 16-bit final output, Y_CL, is a combination of its four 4-bit counterparts, which can be chosen from either the core outputs or the accumulator outputs. This flexibility allows for efficient data processing and selection of the most relevant results for the desired computation.

This thesis presents a Gen 2 pPIM Cluster design accommodating Gen 2 pPIM Core operand widths ranging from 2 to 8 bits. The cluster's operand width is set to double that of the core's operand width. Additionally, the accumulator and output registers are adjusted accordingly to suit the design. The Table 3.2 below outlines how the cluster components are scaled for various design configurations.

Table 3.2: Generation 2 pPIM Cluster Components Dimensions at Different Core Operand Widths

Cluster Data Words (A_CL, B_CL) (Bits)	**Core Data Words (A, B) (Bits)**	**Accumulator (ACC0, ACC1, ACC2, ACC3) (Bits)**	**Output Register (Y0_CL, Y1_CL, Y2_CL, Y3_CL) (Bits)**	**Core Output (Y) (Bits)**	**Cluster Output (Y_CL) (Bits)**
4	2	$2 \times 4 = 8$	$2 \times 4 = 8$	4	8
6	3	$3 \times 4 = 12$	$3 \times 4 = 12$	6	12
8	4	$4 \times 4 = 16$	$4 \times 4 = 16$	8	16
10	5	$5 \times 4 = 20$	$5 \times 4 = 20$	10	20
12	6	$6 \times 4 = 24$	$6 \times 4 = 24$	12	24
14	7	$7 \times 4 = 28$	$7 \times 4 = 28$	14	28
16	8	$8 \times 4 = 32$	$8 \times 4 = 32$	16	32

3.2.1 Router

The routing mechanism is specifically designed to enable all-to-all communication between the cores, accumulator, and the output register in a non-blocking, cross-bar architecture. As depicted in Figure 3.5, input multiplexers in front of each core, accumulator, and output register allow for the selection of individual inputs from a range of available options. This setup ensures efficient and flexible data exchange among the components within the cluster.

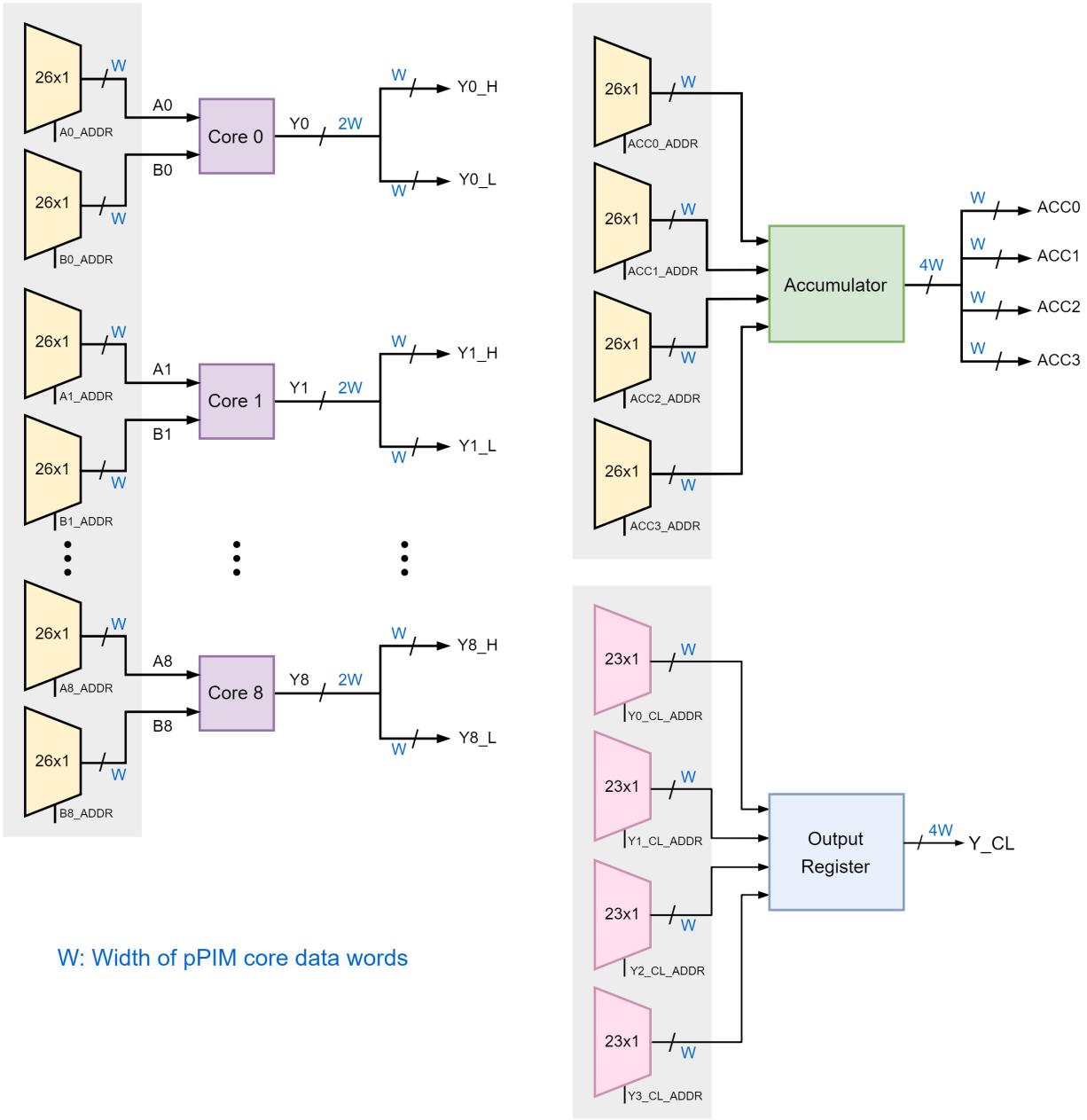

Figure 3.5: pPIM Gen 2 Cluster Exploded View Showing Gen 2 Router Architecture

3.2.1.1 Input Multiplexer

The Input MUX plays a crucial role in the router design, enabling both the core and the accumulator to perform the following functions:

1. Utilize outputs generated by all pPIM cores, including itself.
2. Access results stored in the accumulator.
3. Accept new sets of cluster inputs.

Each core employs a pair of these Input MUXes, one for each of its inputs, A and B. On the other hand, the accumulator requires four Input MUXes, as it is divided into four segments.

This particular Input MUX is implemented as a 26-to-1 MUX, meaning it has 26 inputs and one output. To select from the 26 inputs, a 5-bit wide select line is needed (as calculated by $ceil(\log_2(26)) = 5$). Refer to Figure 3.6. The user can program the address of these select lines at the top-level of the cluster, with the default address being set to 5'b11111.

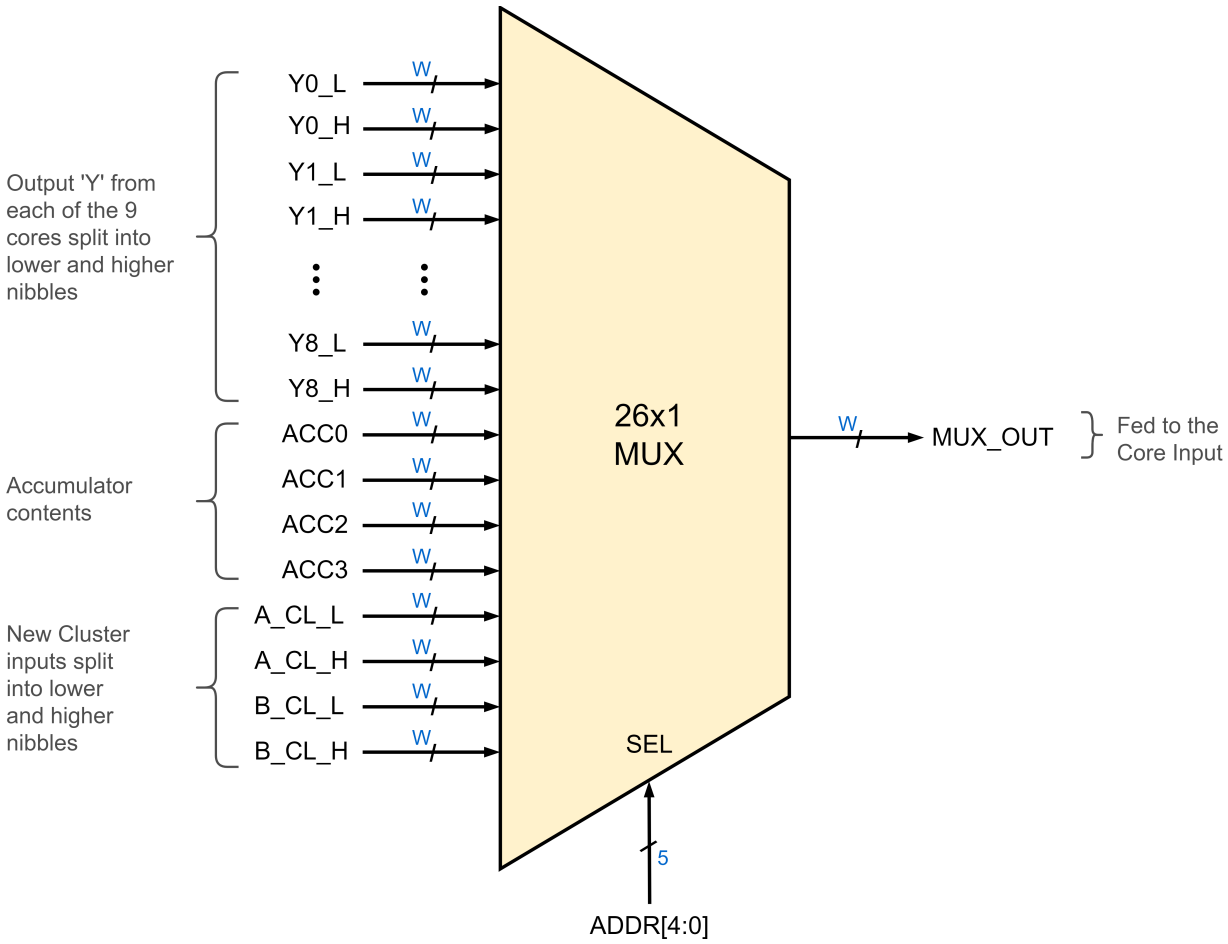

Figure 3.6: Input Multiplexer

3.2.1.2 Output Multiplexer

The Output MUX allows each nibble of the cluster output to choose its content from:

1. Outputs of the nine cores.

2. Contents of the accumulator.

The pPIM cluster requires four of these Output Multiplexers, with each one dedicated to each segment of the final output. The Output MUX is implemented as a 23-to-1 MUX, meaning it has 23 inputs and one output. To select from the 23 inputs, a 5-bit wide select line is needed (as

calculated by $ceil(\log_2(23)) = 5$). Refer to Figure 3.7. The user can pick the address of for select lines at the top-level of the cluster, with the default address being set to 5'b11111. Additionally, the final output *Y_CL* is looped back to one of the MUX inputs to preserve the output.

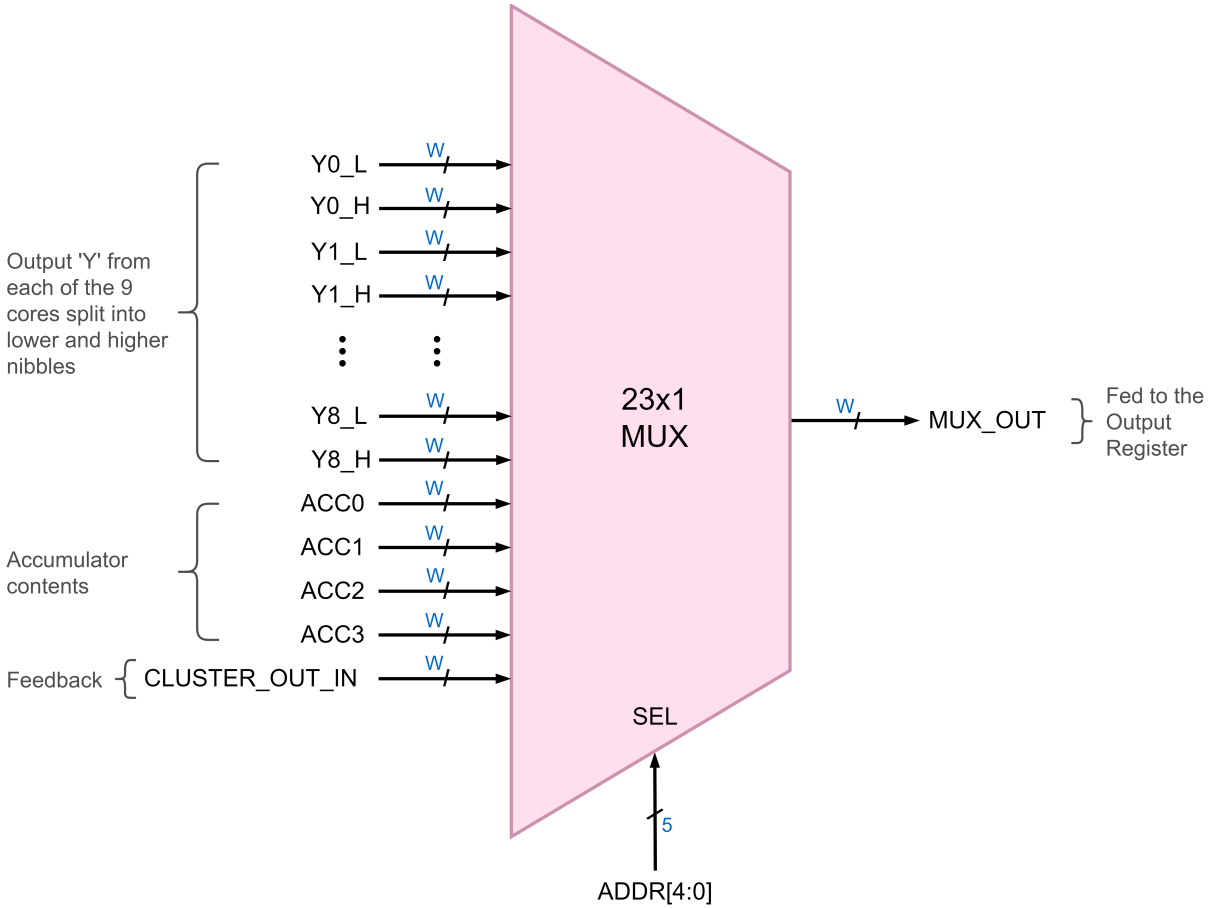

Figure 3.7: Output Multiplexer

3.2.2 Accumulator

The cluster incorporates an accumulator register, providing the capability to store intermediate or final results. The accumulator is divided into four smaller registers, each dedicated to storing one nibble. These four nibbles are concatenated together to form the complete accumulator, as depicted in the Figure 3.8 below. The contents of accumulator are selected using an input

multiplexer, one for each nibble.

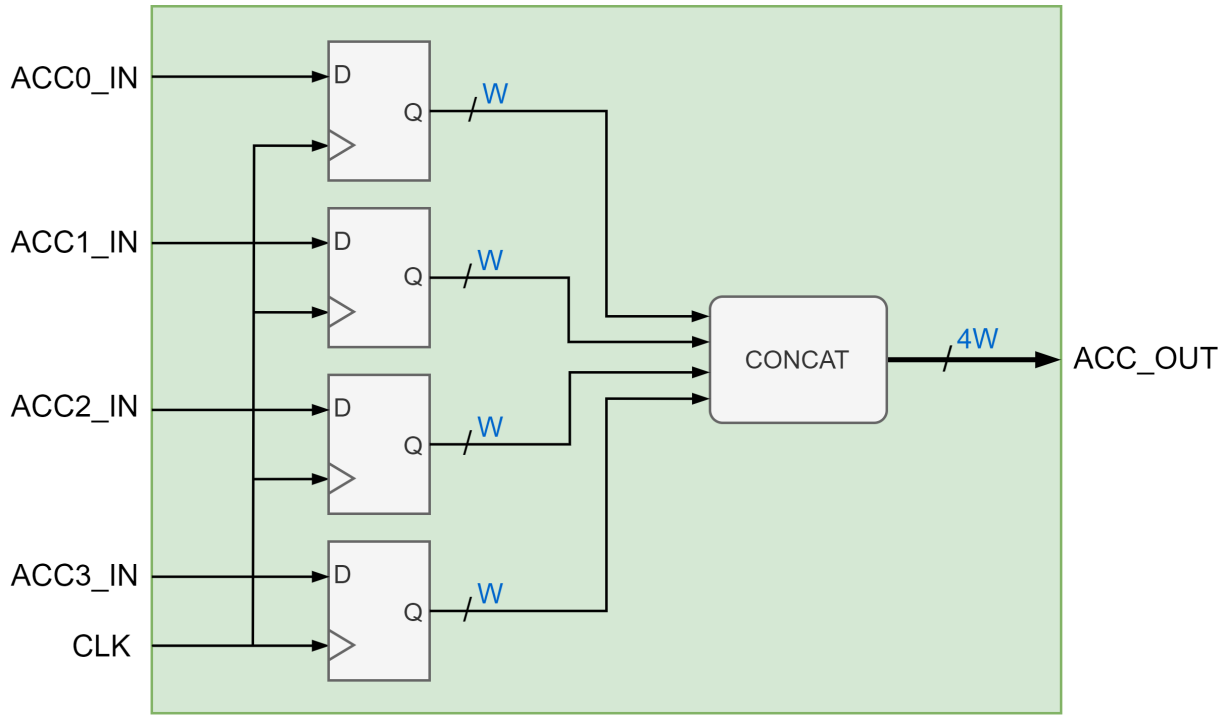

Figure 3.8: Accumulator

3.2.3 Output Register

The Gen 2 pPIM Cluster design, introduces an Output Register that holds the final result of the pPIM cluster (Y_CL). This output register consists of four registers, resulting in the concatenation of into four distinct nibbles, as illustrated in Figure 3.9. The contents of the output register are selected using an output multiplexer, one for each nibble.

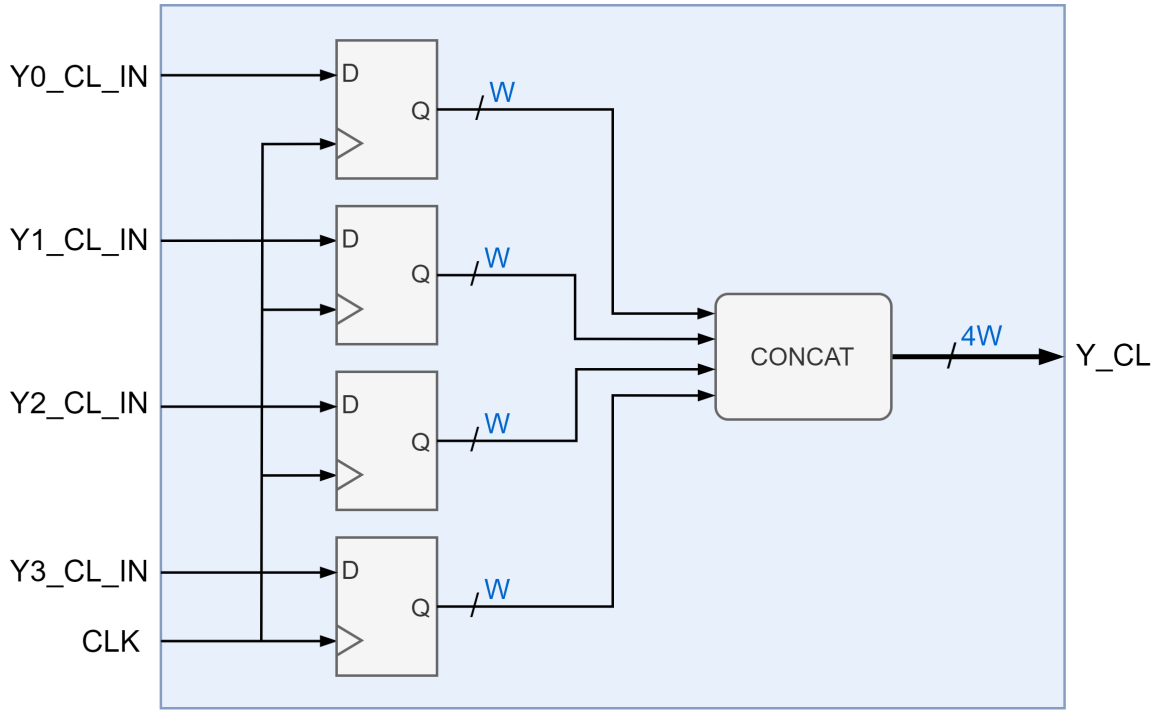

Figure 3.9: Output Register

The Generation 2 pPIM Core and pPIM Cluster designs were thoroughly discussed in this chapter.

Chapter 4

pPIM Core Verification

The Design Under Test (DUT) here is a Gen 2 pPIM Core and this chapter concentrates on creating a verification plan to test the Gen 2 pPIM Core design.

4.1 Testplan

The testplan for Gen 2 pPIM Core is as follows:

- **Stimulus generation**
  - Input Data Words A and B are randomized.
  - Enumerated type opcode {ADD, SUB, MULT, DIV} is randomized as well.
    - Constraint 1: To guarantee at least one occurrence of each opcode during randomization.
    - Constraint 2: To maintain equal opcode distribution with a weight of 25% for each opcode.

- **Checker**
  - The DUT-generated output Y, along with the random stimuli A, B, and opcode, are sampled and broadcast over the analysis ports.
  - Simple SystemVerilog models are created for each arithmetic operation (Addition, Subtraction, Multiplication, and Division) to compute the expected result based on the randomly generated stimuli.
  - The DUT result are compared against the expected result for each arithmetic operation to ensure correctness.
- **Functional Coverage**
  - Covergroup for Data Words and Opcode
    - Coverpoints for A, B, and opcode: To track each possible value for these variables, ensuring that every combination of A, B, and opcode is hit at least once.
    - Cross of A, B and Opcode: To ensure all opcode and data word combinations are exercised.
  - Covergroups for DUT vs. Expected Results
    - Separate *covergroups* are defined for all potential results of $A+B$, $A-B$, $A\times B$, $A\div B$. Python scripts are utilized to generate these *covergroups*.
    - Cross between *binsof* expected values with the *binsof* DUT values is performed for each result obtained during simulation.
- **Concurrent Assertions**
  - DUT output should always match with the Expected output.

- The signals “read” (RUN) and “write” (LOAD) should never be asserted simultaneously.
- A and B should retain their past values for at least 8 clock cycles after “LOAD” is initiated
  - Input data word A should remain constant while function words are being loaded.
  - Input data word B should remain constant while function words are being loaded.
- The input signals should have valid values at the clock edge.
  - Input A should not be in an unknown state (X or Z) at clock edge.
  - Input B should not be in an unknown state (X or Z) at the clock edge.

4.2 UVM Testbench Architecture

The Figure 4.1 illustrates the general UVM testbench architecture, which consists of a hierarchical arrangement of various testbench components. The *Testbench Top* serves as the top-level entity, instantiating the DUT along with interfaces. It also initiates the execution of tests. *Tests* define specific scenarios comprising environments, configurations, and stimulus generation. The *Environment* further organizes the testbench components by segregating agents, scoreboards, functional coverage, and checkers. The *Agent*, at the next level in the hierarchy, acts as a container for a driver, a monitor, and a sequencer. Depending on its purpose, an agent can be active or passive. An active agent injects stimuli into the DUT through its driver, while a passive agent focuses solely on monitoring and instantiates a monitor component without requiring a driver or sequencer. Passive agents are often employed for coverage or checker-related tasks. *Sequences* are reusable test scenario blocks, define specific patterns of stimuli applied to the DUT. A *Sequence item* represents a transaction, containing data information and optional randomization and constraints.

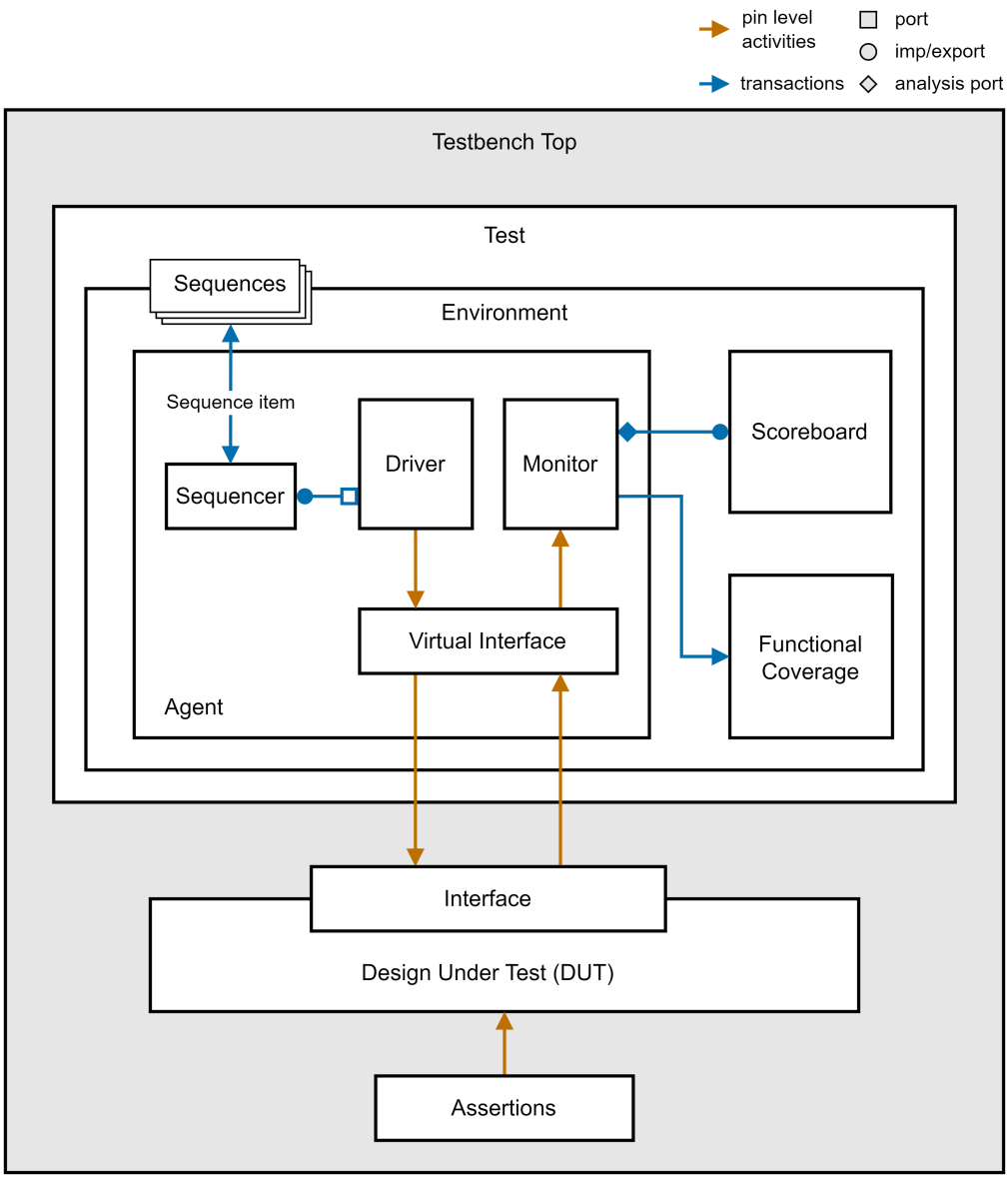

Figure 4.1: Hierarchical UVM Testbench Architecture

The sequencer serves as a mediator between sequences and drivers, facilitating communication through the Transaction Level Modeling (TLM) interface under the hood. The *Driver*, as the name suggests, drives the transactions or sequence items to the DUT using an interface. Simultaneously, the *Monitor* observes the DUT signals through a virtual interface, converting them into sequence-item packets that are broadcast to other testbench components, such as scoreboards and coverage monitors. The *Scoreboard* verifies the functional correctness of the DUT by comparing its output with the expected results. It subscribes to transactions broadcast by the monitor to gather DUT outputs. Additionally, the scoreboard may incorporate reference models using Direct Programming Interface (DPI), which interpret the DUT behavior algorithmically and produce expected results. These reference models can be implemented in various languages, including SystemVerilog, C/C++, or SystemC, and are considered as the "golden truth" against which the DUT results are compared.

Modern testbenches integrate *Assertions*-based verification, ensuring design properties always hold true. When an assertion fails during simulation, it signals potential issue in the design behavior, prompting engineers to investigate and rectify them early on. *Functional Coverage* is used as a metric to measure the completeness of testing with respect to functional aspects or features of a design. Monitoring coverage goals during verification helps identify untested and unreachable areas in the design. To achieve greater coverage, more penetrating testcases are designed.

This comprehensive approach of developing a testbench ensures a more reliable and effective verification process. The next section specifically discusses the testbench architecture for the pPIM Core.

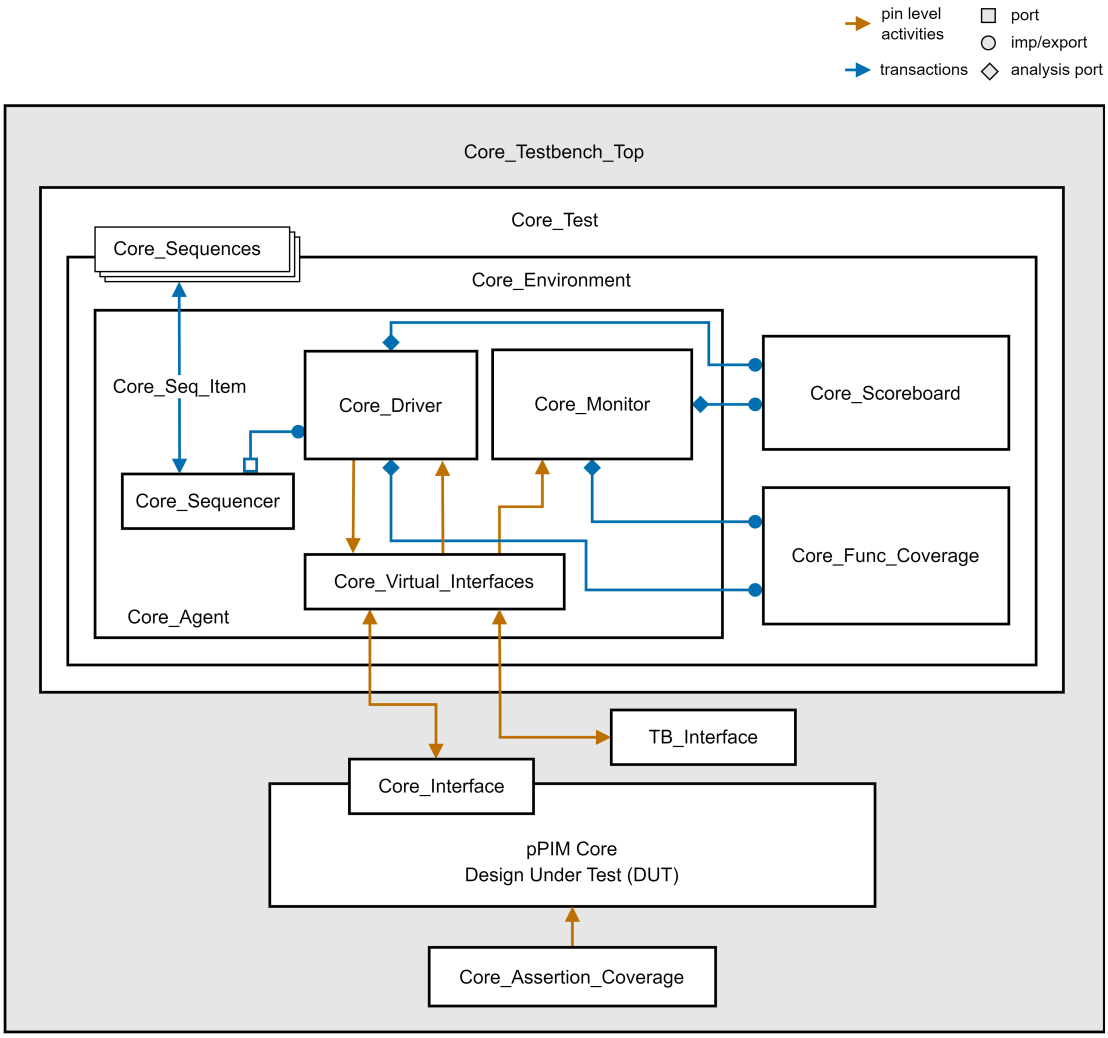

Figure 4.2: pPIM Core Testbench Architecture

4.2.1 Core Testbench Components

The pPIM Core testbench assesses the functionality of the Core by executing four arithmetic operations Add, Subtract, Multiply, and Divide at random, using randomly generated stimuli. It captures the response of the DUT and concurrently verifies it against the expected response. The testbench organization is shown in the Figure 4.2.

Each component in the testbench is registered in the UVM factory and plays a vital role, as described below:

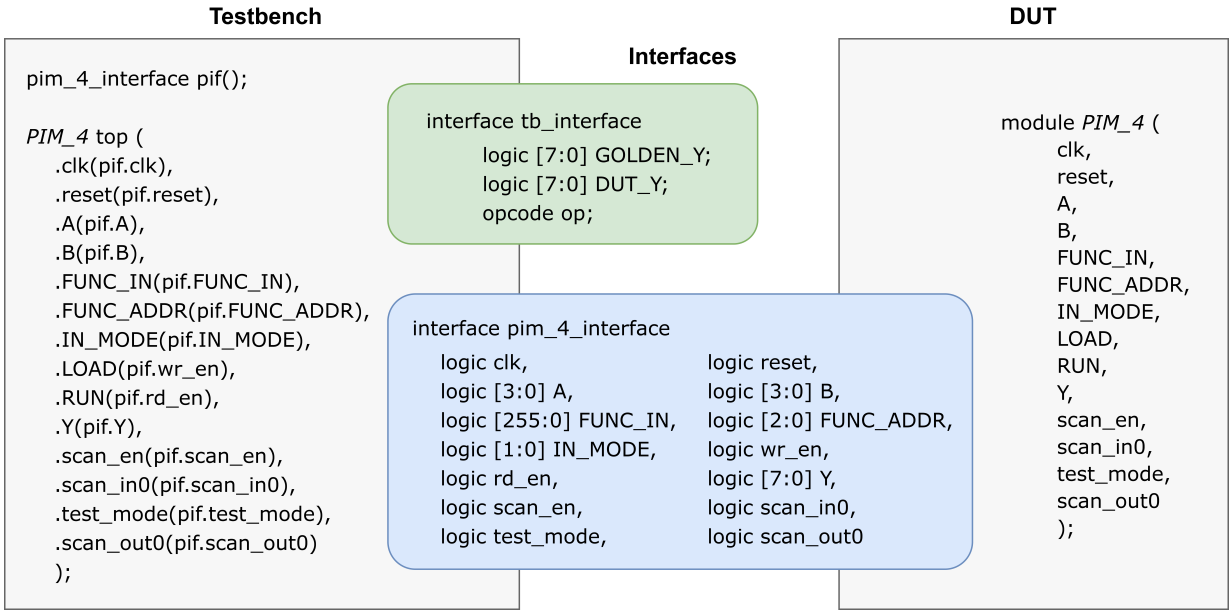

Figure 4.3: Interfaces connecting DUT (Core) to Testbench Environment

4.2.1.1 PIM Interfaces

The initial and essential step in the design verification process is establishing a connection between the testbench and the design. The testbench envelops the design, injecting stimulus into it and capturing its responses. Interface is created to communicate with the design ports. This interface acts as a smart bundle of wires, that allows to specify timing and signal direction [20].

In the Figure 4.3, two interfaces, namely *pim_interface* and *tb_interface*, are used in this testbench. The *pim_interface* is instantiated like a class within the testbench top module, with signals connected to the ports of the DUT using a dot (.) operator. The role of the other interface will be explained in a later section.

It's important to note that both the interface and the DUT are static components. On the other hand, the UVM testbench components are dynamic, represented as class objects that connect to the design during runtime. To bridge the gap between the static modules and dynamic objects, interface is declares as "virtual" and is used as a pointer or handle to the actual physical interface.

Both interfaces are registered with "uvm_resource_db" as virtual entities.

4.2.1.2 PIM Sequence-item

The *uvm_sequence_item* class is a derived class of the *uvm_transaction* class, enabling testbench developers to generate user-defined sequence items. Within the pPIM Core testbench, the *pim_seq_item* class extends from *uvm_sequence_item* and introduces two input variables A and B, declared with the “rand” keyword, indicating that they will be randomized during simulation. Additionally, a method is defined to randomize the opcode using the built-in *randomize()* method. The primary purpose of *uvm_sequence_item* is to transport data from *uvm_sequences* through the *uvm_sequencer* to a *uvm_driver* [22].

4.2.1.3 PIM Sequence

UVM sequence is a collection of sequence items. The class *pim_sequence* inherits from the *uvm_sequence* base class and is parameterized with *uvm_sequence_item*. This class contains a *body()* method, which defines the the desired behavior of a sequence.

Function Word Generation: The function words for all the four opcodes are programmed in the *body()* method of the class *pim_sequence* and are stored in separate variables of type reg. The function words are determined by evaluating the output for intended operation for every possible pairing of inputs. To calculate the outputs for each pairing, an iterative process is employed, spanning across $2^W * 2^W$ iterations, where W represents the width of the data words. During each iteration, the output bits are split and stored in separate temporary registers at the corresponding iteration index.

The *body()* method also contains the code to generate and send the randomized packet of *pim_sequence_item*. The *start_item()* and *finish_item()* methods provided by base class are used to control this randomization process, ensuring that the intended data is randomized before being

sent to the driver.

4.2.1.4 PIM Sequencer

The UVM sequencer plays a role of a mediator by transferring packets of sequence items from a sequence to a driver for further processing. In this context, the *pim_sequencer* class inherits from the base class *uvm_sequencer* and is parameterized to handle *pim_sequence_item* objects.

4.2.1.5 PIM Driver

The *pim_driver* class is extended from a base class *uvm_driver* and parameterized to accept *pim_sequence_item*. Its primary function is to take sequence item packets from the sequencer and drive them into the DUT using a virtual *pim_interface*. This process occurs during the *run_phase()* of the simulation.

In the *run_phase()* method, the driver repeatedly calls the *get_next_item()* method on the seq_item_port. This method is blocking, meaning it waits until an item is available in the sequencer's FIFO before proceeding. These sequence items contain randomly generated opcode, representing different arithmetic operations and random input data words.

The driver has four distinct tasks that handle passing the contents of temporary registers programmed in the *body()* method of the sequence into the function registers of the DUT. Depending on the opcode received, the driver calls the appropriate task, ensuring that the function words are correctly loaded with the outcomes of randomly generated instructions.

To avoid frequent reprogramming of the LUT, a logic is employed to keep the same opcode for a significant number of cycles before a new opcode is loaded into the DUT. This allows the random sequence items A and B to be applied to the pPIM Core with the same instruction for an extended period.

Before calling *item_done()* on the seq_item_port, the driver publishes the random opcode

packet to the scoreboard and functional coverage monitor via the analysis port. *item_done()* signals the sequencer that it can send the next sequence item for processing.

The *pim_driver* includes a *reset()* method, which sets the PIM to known values before the randomization process starts.

4.2.1.6 PIM Monitor

The *pim_monitor* is a derivative of the *uvm_monitor* base class. During its operation, the monitor repeatedly samples the input values (A, B) observed on the DUT pins at the positive edge of the clock via virtual *pim_interface*. It is responsible for observing the DUT output (Y) resulting from the operation. Only during the function word loading process, the monitor ignores the output. All three values of the pins are encapsulates into a packet of type *pim_seq_item* and writes it on an *uvm_analysis_port,* to be utilized by other testbench components.

4.2.1.7 PIM Agent

The *pim_agent* is an active component extended from *uvm_agent* that creates instances of *pim_sequencer*, *pim_driver*, and *pim_monitor* during the $build_phase()$. In the *connect_phase()*, the *pim_driver* and *pim_sequencer* are interconnected as follows:

$$pim\_driver.seq\_item\_port.connect(pim\_sequencer.seq\_item\_export) \tag{4.1}$$

4.2.1.8 PIM Scoreboard

The *pim_scoreboard* extends from the base class *uvm_scoreboard*. The scoreboard subscribes to the data published by driver and the monitor over *uvm_analysis_imp* ports. It imports the randomized opcode from the driver and the values of A, B and Y as seen by DUT from the monitor. To import the data from the analysis ports the *write()* methods are implemented.

In the *run_phase()*, the scoreboard performs calculations to determine the expected/golden results using simple models based on the opcode. Addition and multiplication require no adjustments in the expected result calculations. However, for subtraction, adjustments are made as the DUT performs subtraction in 2's complement. In the case of division, if the operand B (denominator) is 0, the expected result is set to the maximum possible value of the register to avoid division by zero.

The value of the final output (Y) published by the monitor is stored in a variable as the DUT Result in the scoreboard, while the expected result is calculated using the models. The two results are then compared to check for any discrepancies during the simulation. If there is a difference between the two results, an UVM_ERROR is asserted.

Additionally, the expected and DUT outputs are published over two separate analysis ports for functional coverage. These two values are also driven to the virtual *tb_interface* pins DUT_Y and GOLDEN_Y to support assertion coverage.

The *pim_scoreboard* also incorporates the *report_phase()* to provide a count of each opcode at the end of the simulation.

4.2.1.9 PIM Functional Coverage

The *pim_fcov* is a class that extends the *uvm_component*. It declares four *uvm_analysis_imp* ports, each with a separate *write()* method to receive packets from other testbench components. These packets include: 1) Values of A, B, and Y as observed by the DUT from the monitor, 2) Opcodes from the driver, 3) DUT output from the scoreboard. 4) Expected output also from the scoreboard.

The coverage plan is outlined in the 4.1. The first covergroup focuses on data words A and B and the opcode, defining coverpoints for each of them. The cross coverage for the three coverpoints is also measured in the same covergroup.

To implement covergroups for each DUT vs. Expected comparison, a python code proves

to be helpful. Given the limited number of input pairs combinations for a given width of the operand, there are only a limited number of values of outputs generated by the DUT. With the use of a python code, unique bins for output Y are calculated for each pair of A and B for all four operations. These sets are provided as python lists to another python script, which iterates over and prints the covergroups for all possible combinations of input pairs for all the operations. The crossing of DUT and expected values reconfirms that our calculations are consistent with the results obtained from the PIM.

4.2.1.10 PIM Environment

An environment acts as a container that houses various components including agents, scoreboards, functional coverage monitors etc. The user defined *pim_environment* class is derived from the base class *uvm_env*, instantiates *pim_agent*, *pim_scoreboard* and *pim_fcov*. At this level of the UVM testbench hierarchy, all TLM (Transaction-Level Modeling) port connections are established. During the *connect_phase()* of the pim_environment, six such TLM analysis port-import connections are set up. These connections are crucial for facilitating communication and data exchange between different testbench components. For a more detailed view of these connections, refer to the Figure 4.4 below.

4.2.1.11 PIM Test

The *pim_test* is a top-level component that inherits from *uvm_test* and holds all the details specific to the test. The *pim_test* instantiates *pim_environment* and *pim_sequence* handle. The sequence is initiated in the test by invoking the *start()* method on the sequencer, as shown in the following line of code:

$$pim\_sequence.start(pim\_sequencer); \tag{4.2}$$

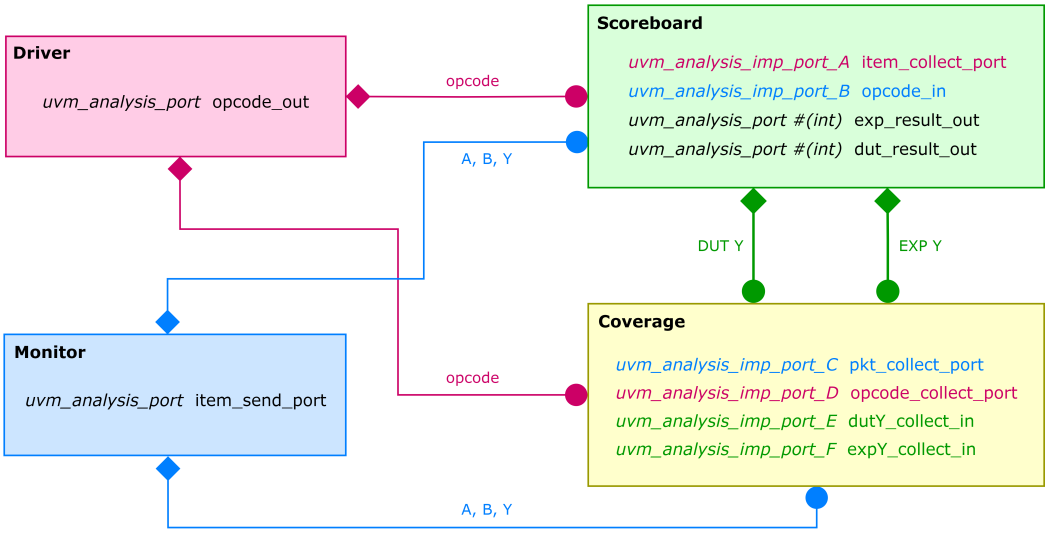

Figure 4.4: TLM Analysis Ports and Imports Interconnecting PIM Testbench Components

4.2.1.12 PIM Testbench Top

The *pim_top* serves as the testbench top and acts as a container for various components, including the instantiation of the *PIM* (DUT), *pim_interface*, *tb_interface*, and *pim_assertions* module. The test execution begins in the initial block of the testbench top by calling the *run_test()* task. This task accepts the test name as a string parameter. It then triggers the phasing mechanism, which orchestrates the sequential execution of UVM phases in their predefined order.

This chapter comprehensively covered the architecture and implementation details of each component within the Core testbench. The simulation results are summarized in Chapter 7. Moving forward, the Chapter 5 will focus on the Cluster testbench, discussing its structure and functionality.

Chapter 5

pPIM Cluster Verification

The Design Under Test (DUT) here is Gen 2 pPIM Cluster and this chapter introduces a verification plan and methodology to test the Gen 2 pPIM Cluster. The primary focus of this testbench is on the implementation of the Multiply-and-Accumulate (MAC) operation. MAC operations is demonstrated as one of the use-cases of a cluster as they are both computationally intensive and frequently used in CNNs [3]. Through this verification process, we aim to ensure the correctness and efficiency of the MAC operation along with validating the functionality of Gen 2 pPIM Cluster.

5.1 Testplan

The testplan for Gen 2 pPIM Cluster is as follows:

- **Stimulus generation**
  - Cluster input Data Words A_CL and B_CL are randomized.
  - Router addresses are performed to perform MAC algorithm.

- **Checker**
  - The DUT-generated output Y_CL, along with the random stimuli A_CL and B_CL are sampled and broadcast over the analysis ports.
  - SystemVerilog model is written for each the MAC operation to compute the expected result on the randomly generated stimuli.
  - The DUT result are compared against the expected result to ensure functional correctness of MAC algorithm.
- **Functional Coverage**
  - Covergroup for A_CL, B_CL and Y_CL
    - Coverpoints for A_CL, B_CL and Y_CL: To track each possible value for these variables, ensuring that every combination is hit at least once.
  - Covergroup for DUT vs. Expected Results
    - Cross between *binsof* expected values with the *binsof* DUT values is performed for each result obtained during simulation.

5.2 UVM Testbench Architecture

As observed in the preceding chapters, the pPIM cluster is comprised of nine pPIM cores, each subjected to rigorous testing to verify its functionality. The Testbench architecture for the cluster closely resembles that of the core testbench architecture. It adheres to the same general hierarchy of testbench components, as illustrated in the Figure 5.1.

The uniqueness of this testbench lies in its driver. In the driver, each of the nine cores is programmed separately, and a special algorithm is implemented to map the MAC operation by

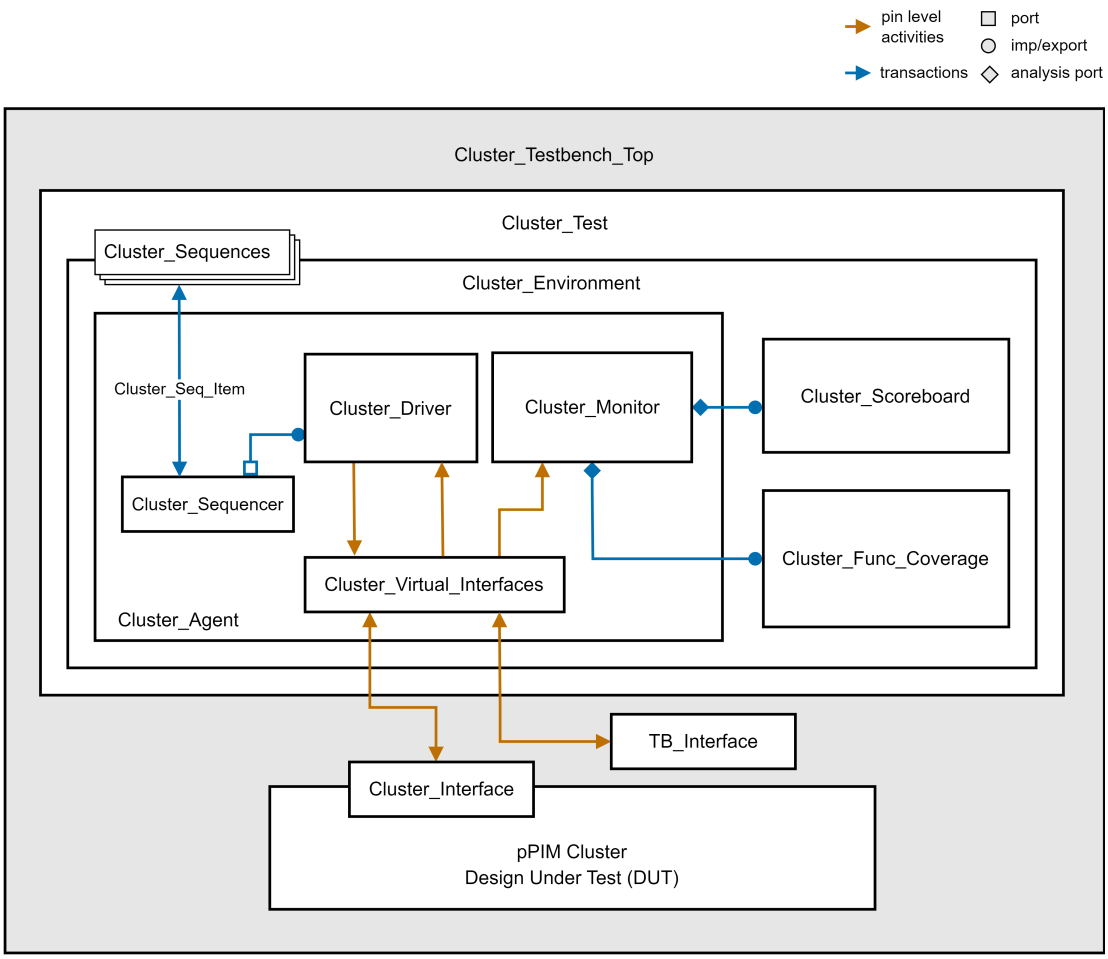

Figure 5.1: pPIM Cluster Testbench Architecture

effectively harnessing the capabilities of these nine cores and the router.

5.2.1 Cluster Testbench Components

The utility of different testbench components is discussed in this section.

5.2.1.1 PIM Cluster Interfaces

The UVM testbench for the pPIM cluster is designed with two interfaces: *pim_cluster_interface* and *pim_cluster_tb_interface*. The *pim_cluster_interface* encompasses all the wires connecting to the ports of the DUT, as well as separate scan ports for each core.

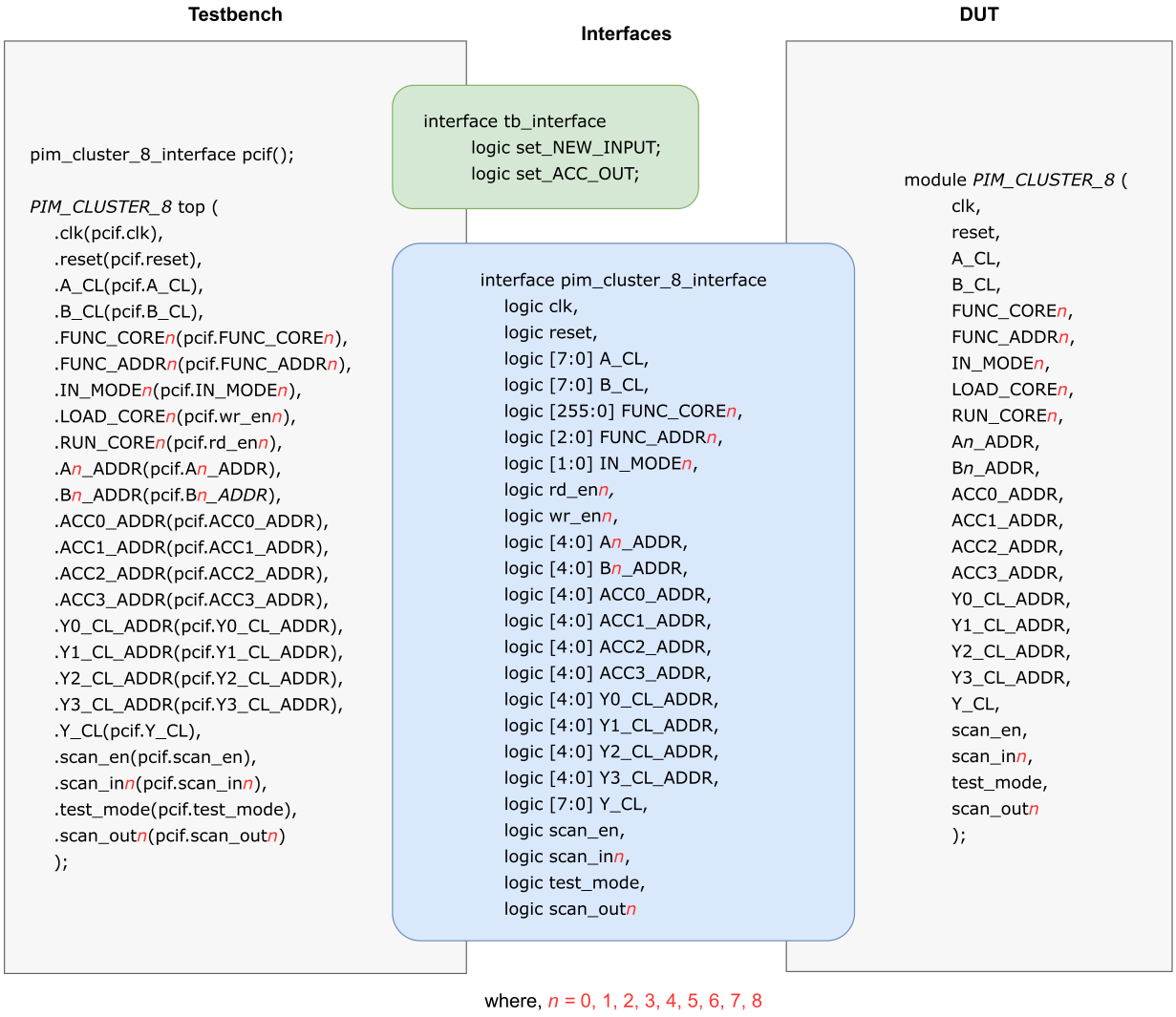

Figure 5.2: Interfaces connecting DUT (Cluster) to Testbench Environment

On the other hand, *the pim_cluster_tb_interface* declares a couple of logic-type variables: "set_ACC_OUT" and "set_NEW_INPUT." These variables serve as flags within the testbench, enabling control during the simulation process.

5.2.1.2 PIM Cluster Sequence

The *pim_cluster_seq_item* class declares A_CL and B_CL as random inputs for the DUT. The *body()* task within the *pim_cluster_sequence* is responsible for programming the function words and storing them into registers. Also, it initiates the randomization of cluster sequence items. Subsequently, the *pim_cluster_driver* receives the randomized packet of sequence items via the *pim_cluster_sequencer*.

5.2.1.3 PIM Cluster Driver

The pim_cluster_driver is responsible for implementing the MAC (Multiply-Accumulate) operation algorithm, as discussed in detail in 5.4. However, the driver can also be programmed to execute other complex operations on the cluster.

The driver consists of following tasks:

- Task *pim_cluster_reset()*: sets all the ports of all cores within the cluster as well as the cluster ports to known values.
- Task *load_PIM_cores_MAC()*: The purpose of this task is to load the four pPIM (Parallel Processing in Memory) cores with multiplication function words and the remaining five cores with addition functions. Additionally, the addresses of all the input MUXes are set to 5'b11111, which represents the default address.
- Tasks *step1()*, *step2()*, .., *step10()*: These tasks represent individual stages of the MAC algorithm. In each step, specific cores are programmed to perform designated tasks, facilitating the data flow between the cores within the cluster. Breaking down the data flow into separate tasks improves the organization of the code.
- Task *multiply_accumulate()*: In this task, the aforementioned individual sub-tasks are called

in a sequence to execute the complete MAC algorithm. Additionally, this task takes care of pipelining, ensuring efficient processing.

These tasks are invoked during the *run phase()* of the driver. The signal "set_NEW_INPUT" is asserted when a new input is detected and is deasserted in the following cycle. Similarly, the signal "set_ACC_OUT" is asserted when the final result is available in the accumulator and deasserted in the next cycle. These two signals are handled over the virtual interface *pim_cluster_tb_interface*, aiding in synchronizing the packets to be sent to the scoreboard for further processing.

5.2.1.4 PIM Cluster Monitor

The *pim_cluster_monitor* plays a crucial role in observing the signals of the DUT and converting them into packets of *pim_cluster_seq_item*, which are then broadcasted over the analysis ports.

The monitor sets up two separate analysis ports. The first port is dedicated to transmitting the cluster inputs, while the second for transmitting the cluster outputs to the scoreboard.

When the "*set_NEW_INPUT*" flag is asserted, indicating the arrival of a new input, the monitor captures the input data from the DUT at positive edge of the clock. This captured input data is converted into sequence item packets and then written onto the corresponding analysis port.

On the other hand, when the "*set_ACC_OUT*" flag is asserted, signaling the presence of the final result in the accumulator, the monitor captures this output data from the DUT's signals. The captured output data is then written onto the corresponding analysis port. This write operation takes place at the negative clock edge.

5.2.1.5 PIM Cluster Scoreboard

The *pim_cluster_scoreboard* subscribes to data packets from the monitor to obtain the cluster inputs A_CL and B_CL, as well as the DUT output, Y_CL. A simple model is implemented in the scoreboard to compute the expected MAC result.

The model calculates the product of two input values and then adds this product to the value of MAC from the previous cycle. This operation effectively performs the multiplication and accumulation process. After the multiplication and accumulation step, the model checks whether the resulting value in expected result exceeds or equals 65536. If it does, which means the accumulated result has overflowed beyond the range of 16 bits (2^{16}), and subtracts 65536 is subtracted from the expected result to keep it within the valid range.

For verification, the expected output is compared with the DUT output. If any discrepancies are found, an UVM_ERROR is asserted, indicating a potential issue in the design or implementation. The multiply-accumulate operation is thoroughly tested with $5000 * 256$ passes to ensure comprehensive verification of the PIM cluster's functionality.

Additionally, the scoreboard publishes the DUT output and the Expected output over the analysis ports. These results are used by the functional coverage to measure the coverage metric of the testbench.

5.2.1.6 PIM Cluster Environment

The *pim_cluster_environment* class inherits from the base class *uvm_env*, instantiates *pim_cluster_agent*, *pim_cluster_scoreboard* and *pim_cluster_fcov*. All the TLM (Transaction-Level Modeling) port connections are done in the environment during its *connect_phase()*. Six such TLM analysis port-import connections are set up. These connections enable communication and data exchange between various components within the testbench. For a detailed view of these connections, please refer to the connection diagram 5.3.

5.2.1.7 PIM Cluster Top

In the cluster testbench, the top-level module instantiates the Cluster design and establishes connections between its ports and the testbench using an interface. The testbench top then calls

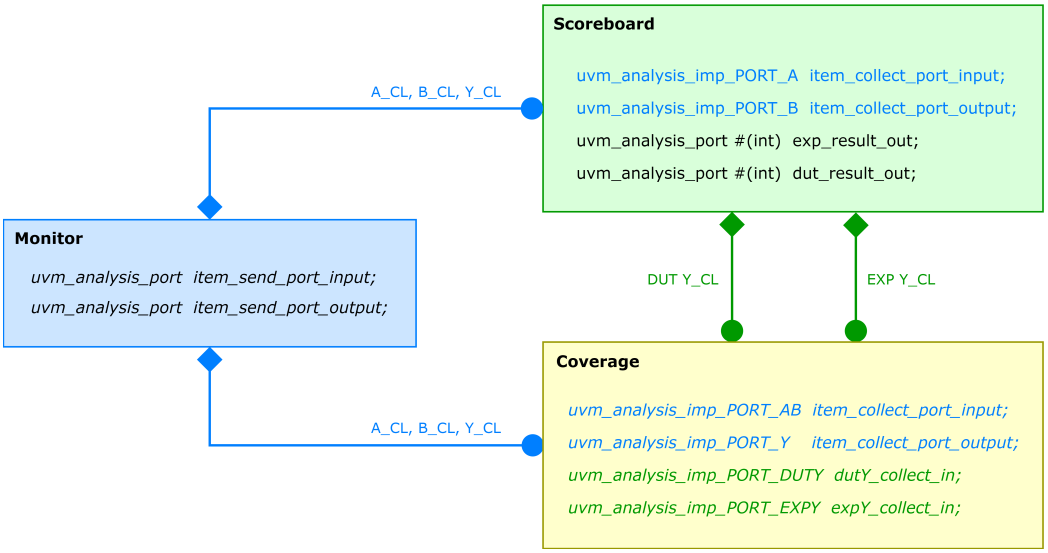

Figure 5.3: TLM Analysis Ports and Imports Interconnecting Cluster Testbench Components

the *run_test()* method, which executes the *pim_cluster_test*. Inside *pim_cluster_test*, the sequence is initiated to drive the test scenarios.

5.3 Multiply-and-Accumulate (MAC)

The "Multiply and Accumulate" operation, often abbreviated as "MAC", is a fundamental arithmetic operation frequently used in signal processing and numerical computing. It combines multiplication and addition in a single step. The MAC operation calculates the product of two operands and adds the result to an accumulated sum. This operation is commonly employed in various applications, such as digital filters, image processing, and vector operations, where efficient processing of large data-sets is required. The MAC operation significantly reduces the number of arithmetic steps, leading to faster and more optimized computations.

The following section presents a detained explanation of the Multiply and Accumulate operation for 8-bit cluster inputs and 4-bit core inputs, showcasing the cluster's capability in handling memory-intensive operations efficiently. It is essential to note that, the cluster and core compo-

nents are designed to be scalable, accommodating a wide range of operand widths. This versatility enables the cluster to execute not only the MAC operation but also other complex computationally heavy instructions.

5.4 Efficient MAC Operation with pPIM Cluster

5.4.1 MAC using Partial Products and Accumulation

Multiplication using partial products meaning breaking down the multiplication of two multi-digit numbers into simpler steps. Each partial product is obtained by multiplying a digit from one number with each digit from the other number and then aligning them in their correct positions. The final product is obtained by summing all the partial products. This technique reduces the complexity of large-scale multiplication and is commonly used in hardware implementations.

In the case of the pPIM cluster, a similar approach is followed to implement MAC. The Cluster operates on double the size of the input compared to the cores inside it. Since each core can handle 4-bit operands, the 8-bit cluster inputs are split into two 4-bit operands: A_H, A_L and B_H, B_L. The subscripts 'H' and 'L' represent the upper and lower nibbles, respectively. The equations for the partial products are as follows:

$$V_0 = A_L B_L \tag{5.1}$$

$$V_1 = A_L B_H \tag{5.2}$$

$$V_2 = A_H B_L \tag{5.3}$$

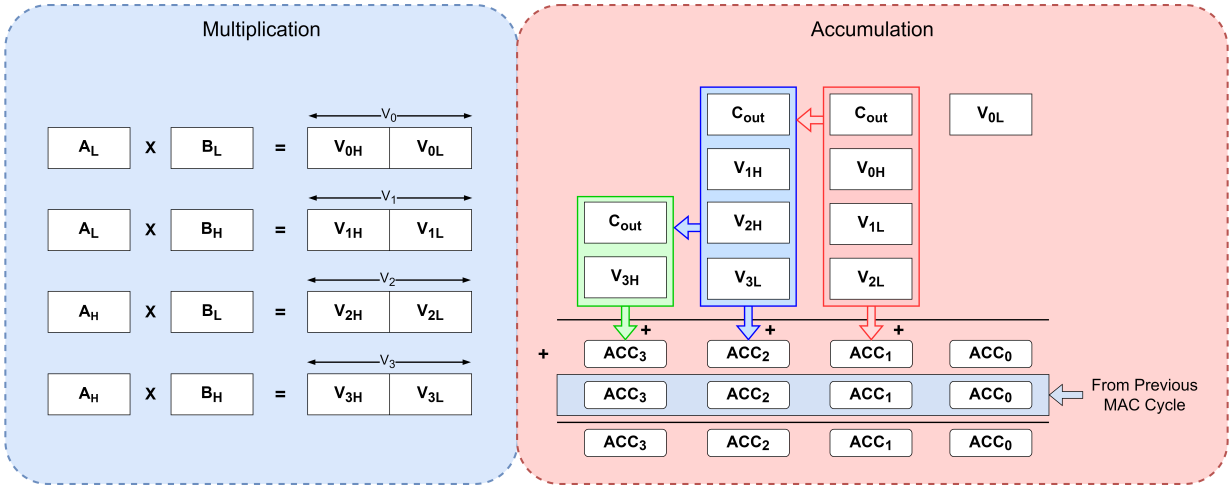

Figure 5.4: Partial Products and Step-wise Accumulation

$$V_3 = A_H B_H \tag{5.4}$$

The partial products obtained in 5.1, 5.2, 5.3, and 5.4. These partial products are split into higher and lower halves and arranged for the summation as shown in the Figure 5.4.

Each summation is accumulated into four individual 4-bit registers (*ACC*0, *ACC*1, *ACC*2, *ACC*3), that further concatenate to form an Accumulator. The result of the MAC operation is obtained as the sum of these partial products.

5.4.2 Mapping MAC Algorithm to pPIM Cluster

To map the above algorithm on the pPIM Cluster, nine cores within the cluster are assigned specific tasks. Among these cores, four are programmed for multiplication, while the remaining five are programmed for addition. In Figure 5.5, each pPIM core (labeled as P0, P1, ..., P8) is represented by small squares, with cores performing multiplication shown in blue and those performing addition shown in red. The time steps are denoted as "t".

During the first clock cycle, four partial products are computed using cores 0, 1, 2, and 3.

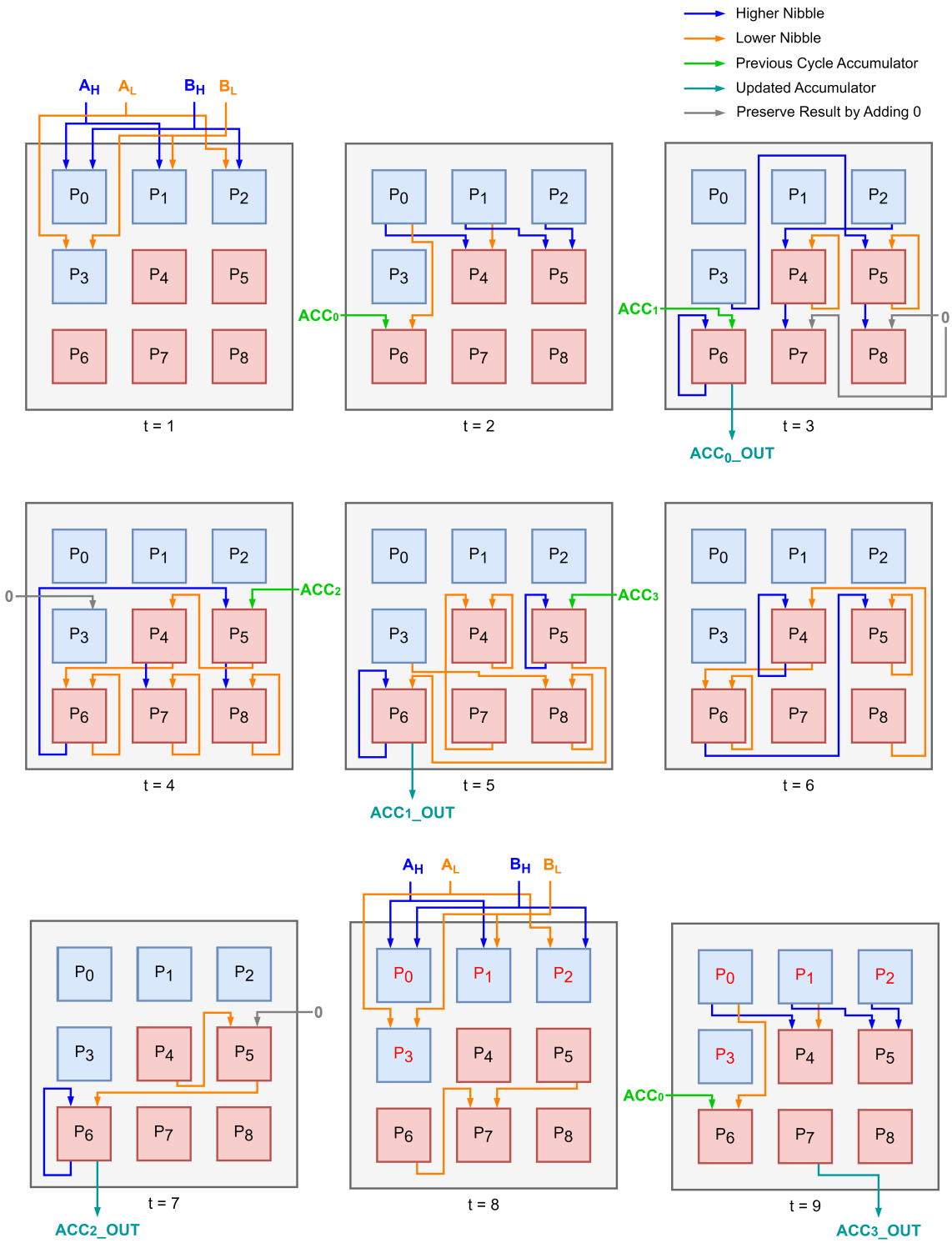

Figure 5.5: MAC Operation Data-flow between Nine Cores inside a Cluster

Starting from the second cycle onward, the addition of these partial products commences, taking into account the previous values of the accumulator for the accumulation process. It is observed that, the updated values of $ACC0$, $ACC1$, $ACC2$ and $ACC3$ are seen at time steps 3, 5, 7, and 9 respectively. The throughput is improved by overlapping consecutive MAC instructions. New sets of inputs at time step 8 without any conflicts. Partial pipelining scales down the MAC operation from 9 steps to 7 steps , except the first iteration. The observed speedup due to the pipelining is $9/7 \approx 1.2857$

An extra clock cycle is required after the results are visible in the accumulator for the final output to be reflected on the output port of the cluster labeled as "Y_CL".

5.4.3 MAC Example with Calculations

Table 5.1: pPIM Cluster MAC Example with step-by-step Calculations

Step	**Active Core**	**Calculation**	**Comment**
At $t = 1$		$A\_CL = 39_{10}, 0010\ 0111_2$, $B\_CL = 74_{10}, 0100\ 1010_2$	Cluster inputs
	PIM 0	$P_0 = A_L \times B_L = 0111 \times 1010 = 0000\,0100$	Partial products 1
	PIM 1	$P_1 = A_L \times B_H = 0111 \times 0100 = 0010\,0000$	Partial products 2
	PIM 2	$P_2 = A_H \times B_L = 0010 \times 1010 = 0000\,0010$	Partial products 3
	PIM 3	$P_3 = A_H \times B_H = 0010 \times 0100 = 0001\,0000$	Partial products 4
At $t = 2$	PIM 4	$P_4 = P_{0H} + P_{1L} = 0100 + 1100 = 0001\ 0000$	
	PIM 5	$P_5 = P_{1H} + P_{2H} = 0001 + 0001 =$ $0000\ 0010$	
	PIM 6	$P_6 = P_{0L} + ACC_0 = 0110 + 0000 =$ $0000\ 0110$	
At $t = 3$	PIM 4	$P_4 = P_{4L} + P_{2L} = 0000 + 0100 = 0000\,0100$	
	PIM 5	$P_5 = P_{3L} + P_{5L} = 1000 + 0010 = 0000\ 1010$	
	PIM 6	$P_6 = ACC_1 + P_{6H} = 0000 + 0000 =$ $0000\ 0000$	
	PIM 7	$P_7 = 0 + P_{4H} = 0000 + 0001 = 0000\ 0001$	Preserve upper nibble of PIM 4

Table 5.1: pPIM Cluster MAC Example with step-by-step Calculations

Step	**Active Core**	**Calculation**	**Comment**
	PIM 8	$P_8 = 0 + P_{5H} = 0000 + 0000 = 0000\ 0000$	Preserve upper nibble of PIM 5
	ACC0	$ACC_0\_OUT = P_{6L} = 0110$	First nibble of the Accumulator
At $t = 4$	PIM 5	$P_5 = ACC_2 + P_{6H} = 0000 + 0000 = 0000\ 0000$	
	PIM 6	$P_6 = P_{4L} + P_{6L} = 0100 + 0000 = 0000\ 0100$	
	PIM 7	$P_8 = P_{8L} + P_{5H} = 0001 + 0000 = 0000\ 0001$	
	PIM 8	$P_7 = P_{7L} + P_{4H} = 0001 + 0000 = 0000\ 0001$	
	PIM 4	$P_4 = 0 + P_{5L} = 0000 + 1010 = 0000\ 1010$	Preserve lower nibble of PIM 5
At $t = 5$	PIM 4	$P_4 = P_{7L} + P_{4L} = 0001 + 1010 = 0000\ 1011$	
	PIM 8	$P_8 = P_{8L} + P_{3H} = 0000 + 0000 = 0000\ 0000$	
	PIM 5	$P_5 = ACC_3 + P_{5H} = 0000 + 0000 = 0000\ 0000$	
	PIM 6	$P_6 = P_{6H} + P_{5L} = 0000 + 0000 = 0000\ 0000$	
	ACC1	$ACC_1\_OUT = P_{6L} = 0100$	Second nibble of the Accumulator
At $t = 6$	PIM 4	$P_4 = P_{4H} + P_{8L} = 0000 + 0000 = 0000\ 0000$	

Table 5.1: pPIM Cluster MAC Example with step-by-step Calculations

Step	**Active Core**	**Calculation**	**Comment**
	PIM 5	$P_5 = P_{5L} + P_{6H} = 0000 + 0000 = 0000\ 0000$	
	PIM 6	$P_6 = P_{4L} + P_{6L} = 1011 + 0000 = 0000\ 1011$	
At $t = 7$	PIM 6	$P_6 = P_{5L} + P_{6H} = 0000 + 0000 = 0000\ 0000$	
	PIM 5	$P_5 = 0 + P_{4L} = 0000 + 0000 = 0000\ 0000$	Preserve lower nibble of PIM 4
	ACC2	$ACC_2\_OUT = P_{6L} = 1011$	Third nibble of the Accumulator
At $t = 8$		$P_7 = P_{5L} + P_{6L} = 0000 + 0000 = 0000\ 0000$	
		$A\_CL = 84_{10}$, $0101\ 0100_2$, $B\_CL = 236_{10}$, $1110\ 1100_2$	2nd set of Cluster inputs
	PIM 0	$V_0 = V_{0H}V_{0L} = A_L \times B_L = 0100 \times 1100 = 0011\ 0000 = P_0$	Step 1 pipelined with Step 8
	PIM 1	$V_1 = V_{1H}V_{1L} = A_L \times B_H = 0100 \times 1110 = 0011\ 1000 = P_1$	
	PIM 2	$V_2 = V_{2H}V_{2L} = A_H \times B_L = 0101 \times 1100 = 0011\ 1100 = P_2$	
	PIM 3	$V_3 = V_{3H}V_{3L} = A_H \times B_H = 0010 \times 0100 = 0001\ 0000 = P_3$	
At $t = 9$	ACC3	$ACC_3\_OUT = P_{7L} = 0000$ $ACC\_OUT = 0000\ 1011\ 0100\ 0110$	Fourth nibble of the Accumulator

Table 5.1: pPIM Cluster MAC Example with step-by-step Calculations

Step	**Active Core**	**Calculation**	**Comment**
	PIM 4	$P_4 = P_{0H} + P_{1L} = 0011 + 1000 = 0000\ 1011$	Step 2 pipelined with Step 9
	PIM 5	$P_5 = P_{1H} + P_{2H} = 0011 + 0011 = 0000\ 0110$	
	PIM 6	$P_6 = P_{0L} + ACC_0 = 0000 + 0110 = 0000\ 0110$	
At $t = 10$	PIM 4	$P_4 = P_{4L} + P_{2L} = 1011 + 1100 = 0001\ 0111$	Step 3
	PIM 5	$P_5 = P_{3L} + P_{5L} = 0110 + 0110 = 0000\ 1100$	
	PIM 6	$P_6 = ACC_1 + P_{6H} = 0100 + 0000 = 0000\ 0100$	
	PIM 7	$P_7 = 0 + P_{4H} = 0000 + 0000 = 0000\ 0000$	Preserve upper nibble of PIM 4
	PIM 8	$P_8 = 0 + P_{5H} = 0000 + 0000 = 0000\ 0000$	Preserve upper nibble of PIM 5
	ACC0	$ACC_0\_OUT = P_{6L} = 0110$	First nibble of the Accumulator
At $t = 11$	PIM 5	$P_5 = ACC_2 + P_{6H} = 1011 + 0000 = 0000\ 1011$	Step 4
	PIM 6	$P_6 = P_{4L} + P_{6L} = 0111 + 0100 = 0000\ 1011$	
	PIM 7	$P_8 = P_{8L} + P_{5H} = 0000 + 0001 = 0000\ 0001$	

Table 5.1: pPIM Cluster MAC Example with step-by-step Calculations

Step	**Active Core**	**Calculation**	**Comment**
	PIM 8	$P_7 = P_{7L} + P_{4H} = 0000 + 0000 = 0000\ 0000$	
	PIM 4	$P_4 = 0 + P_{5L} = 0000 + 1100 = 0000\ 1100$	Preserve lower nibble of PIM 5
At $t = 12$	PIM 4	$P_4 = P_{7L} + P_{4L} = 0001 + 1100 = 0000\ 1101$	Step 5
	PIM 8	$P_8 = P_{8L} + P_{3H} = 0000 + 0100 = 0000\ 0100$	
	PIM 5	$P_5 = ACC_3 + P_{5H} = 0000 + 0000 = 0000\ 0000$	
	PIM 6	$P_6 = P_{6H} + P_{5L} = 0000 + 1011 = 0000\ 1011$	
	ACC1	$ACC_1\_OUT = P_{6L} = 1011$	Second nibble of the Accumulator
At $t = 13$	PIM 4	$P_4 = P_{4H} + P_{8L} = 0000 + 0100 = 0000\ 0100$	Step 6
	PIM 5	$P_5 = P_{5L} + P_{6H} = 0000 + 0000 = 0000\ 0000$	
	PIM 6	$P_6 = P_{4L} + P_{6L} = 1101 + 1011 = 0001\ 1000$	
At $t = 14$	PIM 6	$P_6 = P_{5L} + P_{6H} = 0000 + 0001 = 0000\ 0001$	Step 7
	PIM 5	$P_5 = 0 + P_{4L} = 0000 + 0100 = 0000\ 0100$	Preserve lower nibble of PIM 4
	ACC2	$ACC_2\_OUT = P_{6L} = 1000$	Third nibble of the Accumulator

Table 5.1: pPIM Cluster MAC Example with step-by-step Calculations

Step	**Active Core**	**Calculation**	**Comment**
At $t = 15$		$P_7 = P_{5L} + P_{6L} = 0100 + 0001 = 0000\ 0101$	Step 8 3rd set of Cluster inputs
At $t = 9$	ACC3	$ACC_3\_OUT = P_{7L} = 0101$ $ACC\_OUT = 0101\ 1000\ 1011\ 0110$	Fourth nibble of the Accumulator

The above calculations shows the expected result of the first MAC is seen in the accumulator ACC that is comprised of $(ACC_3, ACC_2, ACC_1, ACC_0)$ at step 9, and it is equal to 2886_{10} equal to $0000\ 1011\ 0100\ 0110_2$. The final result of the second MAC operation, which is the accumulation of $2886 + (84 \times 236)$ is equal to 22710_{10} or $0101\ 1000\ 1011\ 0110_2$ in binary.

The waveform in the Figure 5.6, 5.7 below further confirms above discussed algorithm. For the first iteration, accumulator contents are set to zero with inputs A_CL, B_CL being 39_{10} and 74_{10}. The result is 2886_{10} which gets stored in the accumulator.

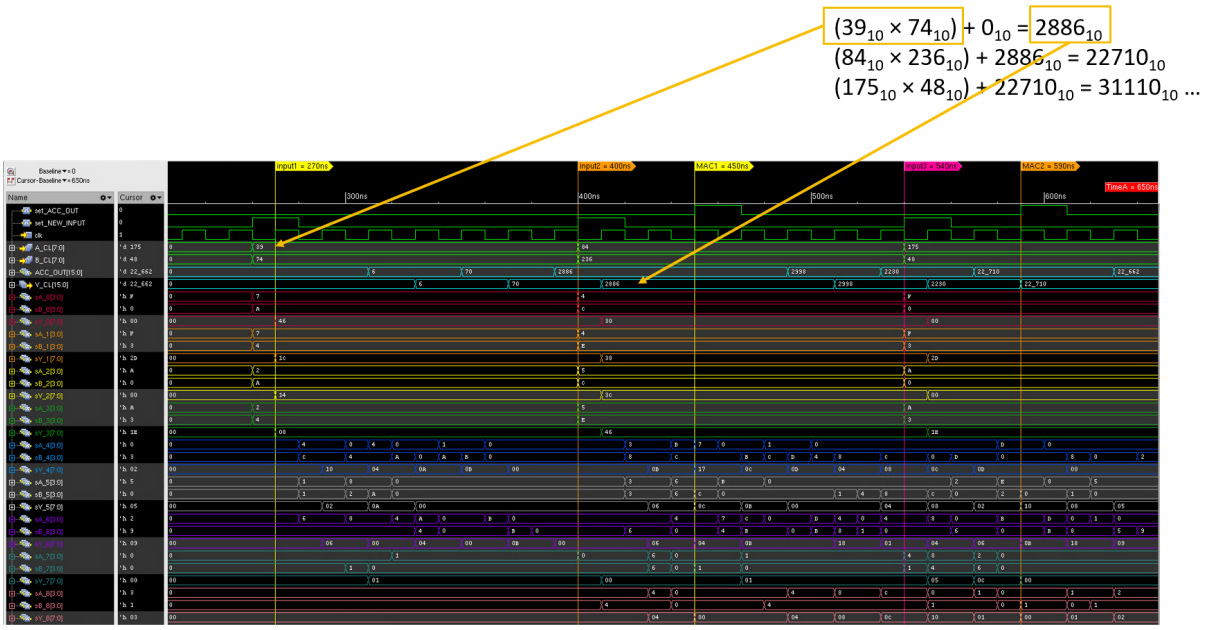

Figure 5.6: Simulation Results Verifying Calculations in Table (Part 1) 5.1

In the second iteration, next pair of inputs, 84_{10} and 236_{10} respectively, is introduced at the time step of 400ns. This occurs even prior to observing the outcome of the first multiplication and accumulation operation within the accumulator. This prompt execution is made feasible by leveraging partial pipelining. By $t = 450$ns, the Accumulator retains the outcome of the first MAC instruction. This value is subsequently combined with the product of the new inputs $(88_{10} \times 236_{10})$ at the time step of $t = 590$ns. Output of the next MAC instruction, 22710_{10}, is then stored as the final result in the accumulator. This process continues for the specified number of iterations which is set in the testbench.

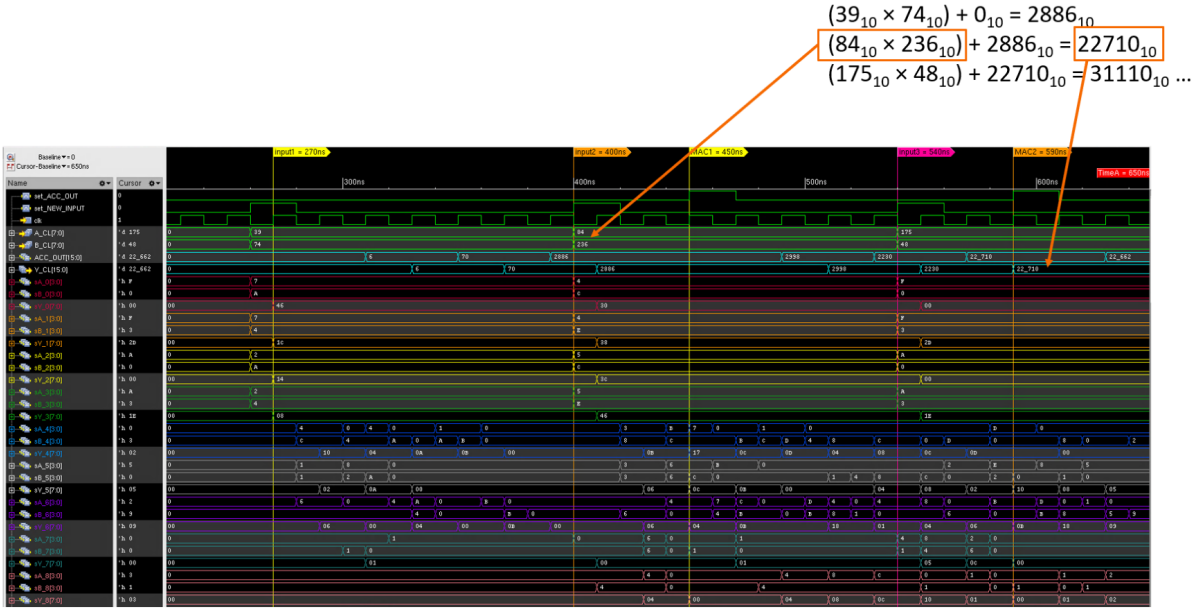

Figure 5.7: Simulation Results Verifying Calculations in Table (Part 2) 5.1

The first MAC instruction yields an output in 180ns after receiving the 1st set of inputs, while each subsequent MAC instruction produces results in 140ns. This optimization is possible because steps 8 and 9 overlap with steps 1 and 2, enabling more efficient processing.

Chapter 6

Tools Development

This chapter presents a toolkit developed using Python for code generation of Gen 2 pPIM Core and Gen 2 pPIM Cluster.

6.1 Methodology

One of the major objectives of this thesis is to create a design-verification suite for user-defined pPIM cores and clusters. The entire process of RTL design and verification for each variant of the design can be time-consuming, error-prone, and inefficient. To address this, the thesis proposes an automated approach using a command-line interface developed in Python.

The development of this tool involves several steps as illustrated in Figure 6.1. First, the Verilog RTL code needs to be generalized by recognizing patterns within the code. This is achieved by simplifying the code and avoiding complex structures to facilitate automation. The pPIM RTL is simplified by replacing generate constructs with module instantiation. Once the RTL is made synthesizable, a UVM testbench is built around the design for verification. Both newly implemented core and cluster RTL designs are verified thoroughly using UVM testbenches,

as detailed in Chapters 4, 5.

After verifying the RTL design with its companion UVM testbench, the next step is to replace hard-coded values with user-specified parameters to allow for design scaling. Specific equations were derived for the pPIM architecture, studying how the components inside the design and port sizes scale with different operand widths. These equations provide a way to adjust every component in the pPIM core and cluster based on the operand width, making the design more flexible and customizable.

This scaling capability ensures that user can generate customized pPIM designs and corresponding testbenches by simply specifying the desired operand width.

- **Equations for pPIM Core Components Sizes**

  1. Width of Core Data Words/ Input Registers $(A, B) = W$
  2. Number of Function Registers $= 2 \times W =$ Number of Multiplexers
  3. Size of a Function Word $= 2^{2W} =$ Size of a Multiplexer
  4. Width of Function Address $= ceil(log_2(2 \times W)$
  5. Core Output $(Y) = 2 \times W$

- **Equations for pPIM Cluster Components Sizes**

  1. Width of Cluster Data Words $(A\_CL, B\_CL) = 2 \times W$
  2. Size of Accumulator $= 4 \times W$
  3. Size of Output Register $= 4 \times W$
  4. Cluster Output $(Y\_CL) = 4 \times W$

Verification is a crucial phase in the design cycle as it plays a pivotal role in identifying and rectifying any errors, whether they stem from RTL or architectural issues. Its primary objective is

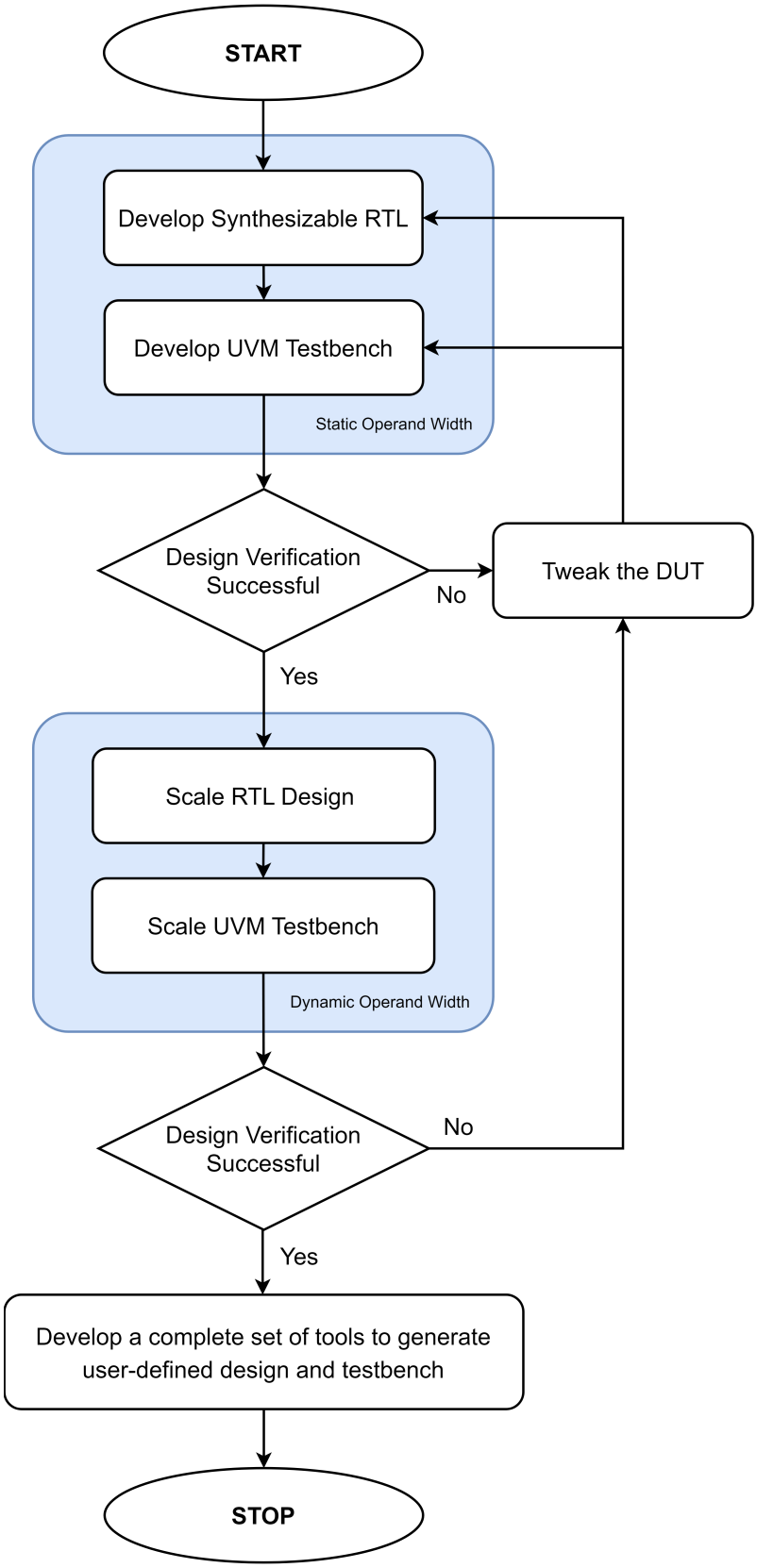

Figure 6.1: Tools Development Methodology

to ensure that the design is an accurate reflection of its specifications. During verification, the testbench rigorously tests the design, and any discrepancies in results are investigated leading to modifications in either the design or the testbench code to address the problems. The simulation is re-run for multiple iterations with different testing scenarios until the required accuracy is achieved.

The next step involves embedding Verilog into Python to enable the generation of user-defined designs. Python aids modularization of the design code into reusable methods to which the user-input data width can be passed as input to produce a custom design. Python's simplicity and platform independence make it a popular choice for scripting in the hardware design process. A Python script is created to generate design and verification testbench files, leveraging string manipulation methods and utility libraries. The generated design and testbench are thoroughly tested to ensure no new errors are introduced during scripting.

To support user-friendly design creation, the Python script is exposed through a Command Line Interface (CLI). The CLI tool accepts three parameters, out of which "width" (data word width) and "techlib" are required. An optional "suffix" parameter is used for module, file, and folder naming based on user input. By specifying these requirements with a single-line command, users can easily create desired designs and testbenches. A *help()* function is included to list the necessary parameters and provide a sample command for generating the required design, enhancing user experience and reducing potential errors.

6.1.1 Core Generation

The ***pPIM_core_toolkit*** takes three command line arguments 1) to specify the core operand size, 2) desired technology library for the synthesis, and 3) an optional module name that will be appended as a suffix in the generated code to uniquify all modules. Users can select between a 180nm, 65nm, or 28nm technology library. The generated files are organized based on the

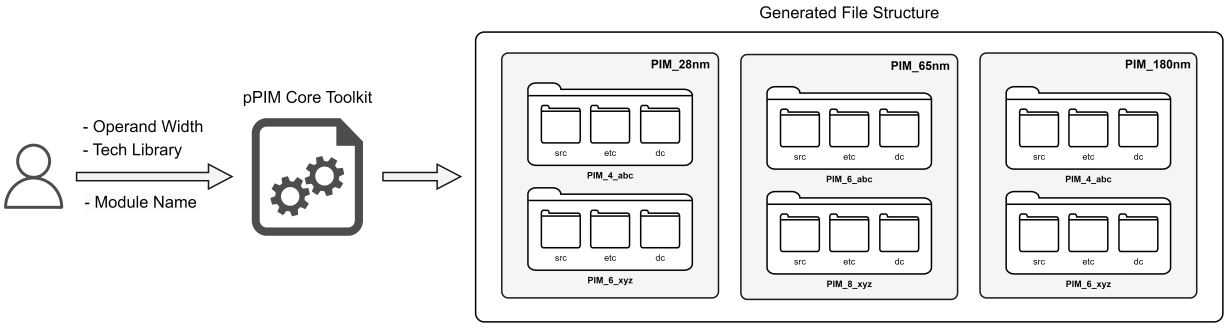

Figure 6.2: Use Case Diagram for pPIM Core Tools

user-defined tech library, specified width, and suffix. Different sized cores with the same techlib are stored under a common top-level folder named 'PIM_<techlib>' (e.g., 'PIM_180' for techlib '180nm'). Refer to a Figure 6.2 illustrating the folder organization done by tools. Subsequent designs with the same techlib will be created inside this top-level folder, creating different design variants. Each variant will have separate folders for source files, control files for gates and RTL, and configuration files.

6.1.2 Cluster Generation

The **pPIM_cluster_toolkit** requires three command line arguments: 1) the PIM core operand size desired within the cluster, 2) the desired technology library for synthesis (options: 180nm, 65nm, and 28nm), and 3) an optional module name to be appended as a suffix in the generated code to uniquify all modules. The folder organization is shown in the Figure 6.3. Similar to the pPIM core toolkit, the generated files are organized based on the user-defined tech library and specified core operand size. The cluster consists of nine cores, each with its operational data word width (W). Therefore, the cluster's operand width will be of resolution 2W. All files and folders generated by the script will have '2W' as their suffix to reflect the cluster's operand width.

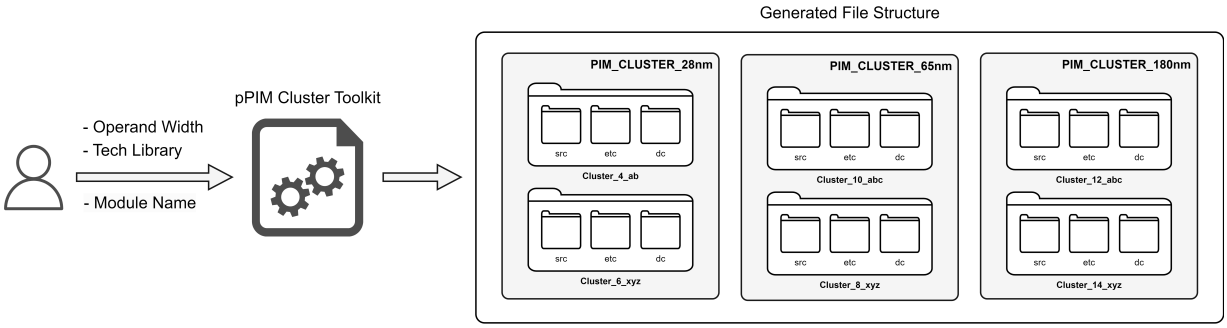

Figure 6.3: Use Case Diagram for pPIM Cluster Tools

6.2 Python for Code Generation

Python is an excellent choice as a scripting language for generating custom designs and testbenches. Its strong string manipulation and formatting capabilities make it well-suited for replacing placeholders with desired values, making code generation efficient and less error-prone. Python has a rich ecosystem of libraries and packages that can be utilized for various tasks, including parsing, data manipulation, and file generation.

The following Python libraries are used in our tools:

- *sys*: The sys library is a fundamental Python library that provides access to system-specific parameters and functions. *sys.argv()* is used to access any command-line options and arguments and *sys.exit()* is used for existing the program.
- *getopt*: The getopt library is used for parsing command-line options and arguments which are often specified with flags (e.g., -h, --verbose).
- math: The math library provides a set of mathematical functions and constants in Python.
- *os*: The os library allows interaction with the operating system. It provides functions for performing operations related to file handling, directory manipulation, and other system-related tasks. This library is used in file generation.

6.3 User Guide

The script takes command line arguments to specify the width of data words, desired technology library for the synthesis, and an optional suffix for module names in the generated RTL.

6.3.1 pPIM Core

- **Usage**
  - *python **pPIM_core_toolkit.py** [arguments]*
- **Arguments**
  1. --help (optional): Displays the help message.
  2. --width or -w (required): Specifies the width of data words for PIM Core.
  3. --techlib or -t (required): Specifies the technology library for the synthesis. The User can select between 180nm, 65nm, and 28nm.
  4. --suffix or -s (optional): Appends a suffix to module names in the generated RTL.
- **Example commands showing long and short arguments**
  1. *python pPIM_core_toolkit.py --width 4 --techlib 180 --suffix A*
  2. *python pPIM_core_toolkit.py -w 4 -t 180 -s A*
- **Generated folder and files hierarchy example (using example commands above)**
  - RTL and Testbench source code: PIM_180/PIM_4_A/**src**
    - Design.v and Testbench.sv files
  - Simulation control files: PIM_180/PIM_4_A/**etc**

    - PIM_4_A.uvm.gate.f
    - PIM_4_A.uvm.rtl.f
  - Synthesis control files: PIM_180/PIM_4_A/**dc**
    - PIM_4_A_config.tcl
    - tech_config.tcl

6.3.2 pPIM Cluster

- **Usage**
  - *python **pPIM_cluster_toolkit.py** [arguments]*
- **Arguments**
  1. --help (optional): Displays the help message.
  2. --width or -w (required): Specifies the width of data words for PIM Core. The PIM Cluster's input size is double the Core's width. For example, with 4-bit PIM cores, the cluster will have 8-bit inputs and a 16-bit output.
  3. --techlib or -t (required): Specifies the technology library for the synthesis. User can select between 180nm, 65nm and 28nm.
  4. --suffix or -s (optional): Appends a suffix to module names in the generated RTL.
- **Example commands showing long and short arguments**
  - *python pPIM_cluster_toolkit.py --width 4 --techlib 180 --suffix xyz*
  - *python pPIM_cluster_toolkit.py -w 4 -t 180 -s xyz*
- **Generated folder and files hierarchy example (using example commands above)**

- RTL and Testbench source code: PIM_180/PIM_CLUSTER_8_xyz/**src**
  - Design.v and Testbench.sv files
- Simulation control files: PIM_180/PIM_CLUSTER_8_xyz/**etc**
  - PIM_CLUSTER_8_xyz.uvm.gate.f
  - PIM_CLUSTER_8_xyz.uvm.rtl.f
- Synthesis control files: PIM_techlib/PIM_width_suffix/**dc**
  - PIM_CLUSTER_8_xyz_config.tcl
  - tech_config.tcl

Chapter 7

Results

This chapter presents the verification results achieved from the UVM verification testbench for both Gen 2 pPIM Core and Gen 2 pPIM Cluster. Also, both the core and the cluster, have undergone RTL as well as gate-level synthesis. Power, area, and scan-test coverage results are discussed.

The results are collected using following tools:

1. Cadence Xcelium for RTL simulations.
2. Synopsys Design Compiler for logic synthesis, test insertion, power, and timing analysis.
3. Synopsys PrimeTime for timing analysis.
4. Synopsys PrimePower for power analysis.

7.1 Gen 2 pPIM Core

This section presents results for the Gen 2 pPIM Core.

7.1.1 Verification Results

As per the verification methodology discussed in Chapter 4, the pPIM Core undergoes comprehensive testing using the UVM testbench with a random seed. Our developed **pPIM_core_toolkit** enables the generation of multiple versions of the pPIM Core design, along with its corresponding UVM testbench. Each design, based on its configuration, is then assigned a different number of passes in the tools. The number of passes is determined as shown in Table 7.1:

Table 7.1: Number of Passes in Simulation for Various pPIM Core Configurations

W (Bits)	**Cycle Multiplier**	**Width of Function Word (Bits)**	**Number of Passes**
2	1000	16	16000
3	1000	64	64000
4	1000	256	256000
5	1000	1024	1024000
6	100	4096	409600
7	10	16384	163840
8	10	65536	655360

7.1.1.1 Simulation Results

This section showcases simulation waveforms for the Gen 2 pPIM Core with different Operand widths (W). During the simulation, four opcodes ADD, SUB, MULT, and DIV are randomly generated and each one is tested for a period of Total Simulation Time/20 or Total Passes/20. The blue markers on the waveform indicate the start and end of the function word loading process. It is observed that as the operand width increases, the time required to program the function word also increases.

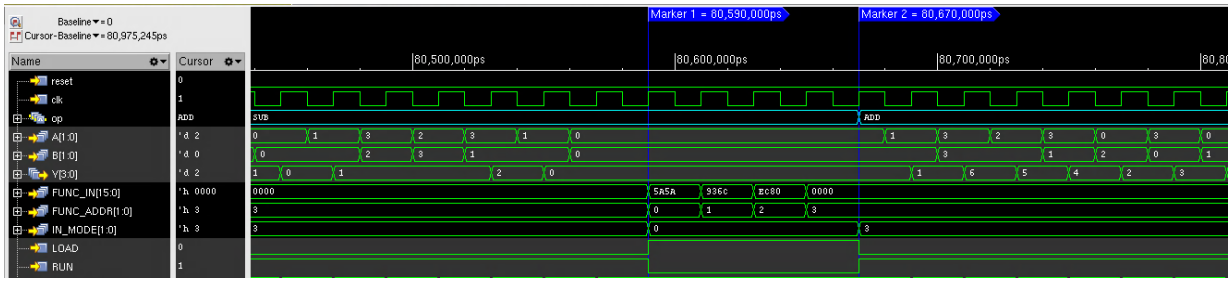

Figure 7.1: Operand Word Width (W) = 2

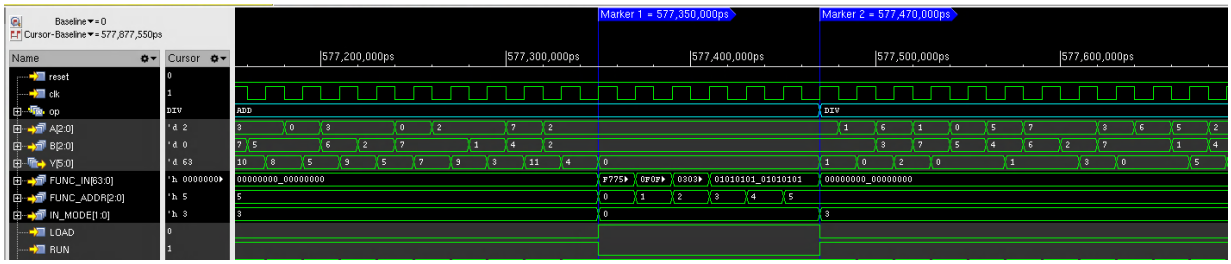

Figure 7.2: Operand Word Width (W) = 3

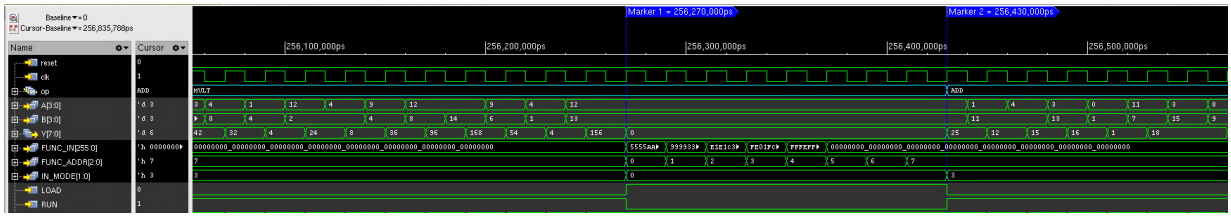

Figure 7.3: Operand Word Width (W) = 4

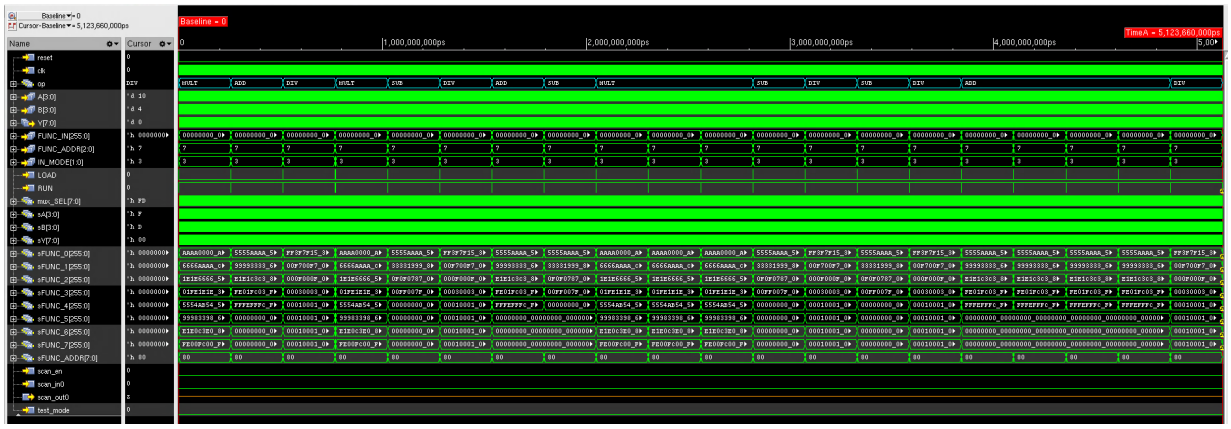

Figure 7.4: Full Simulation View: Operand Word Width (W) = 4

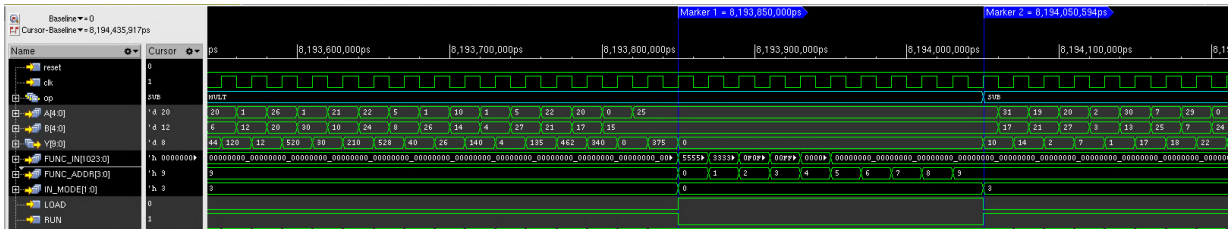

Figure 7.5: Operand Word Width (W) = 5

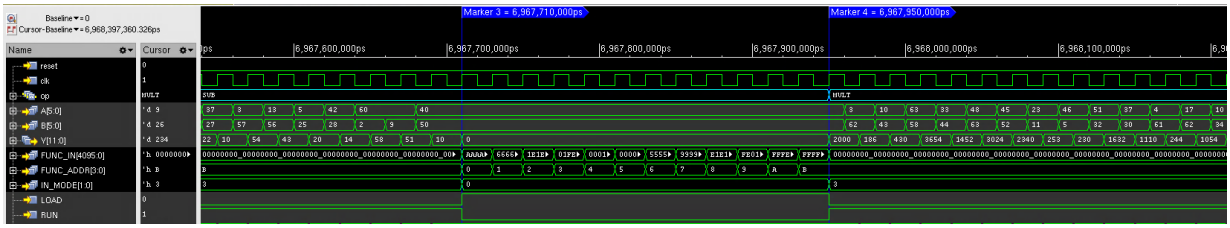

Figure 7.6: Operand Word Width (W) = 6

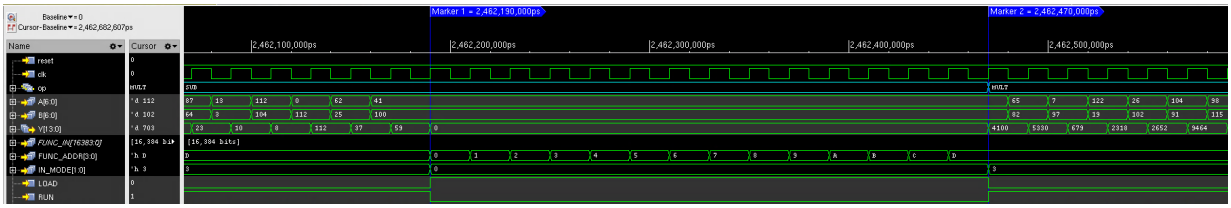

Figure 7.7: Operand Word Width (W) = 7

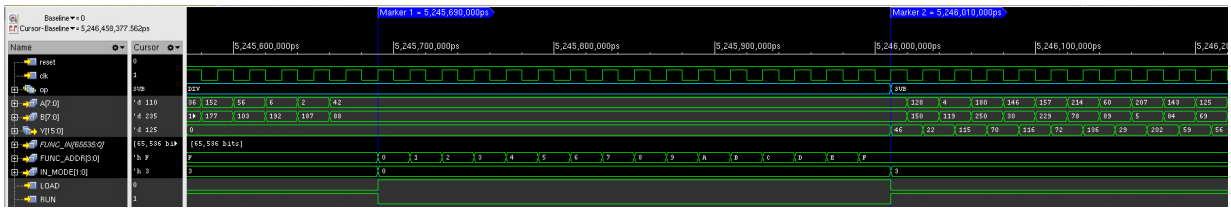

Figure 7.8: Operand Word Width (W) = 8

7.1.1.2 Functional Coverage

Figure 7.9 displays the comprehensive coverage analysis, including both code and functional coverage, for the user-defined Gen 2 pPIM Core utilizing 4-bit operands. This combined coverage evaluation offers a holistic view of the core's verification progress and ensures that both design correctness and functionality are thoroughly examined.

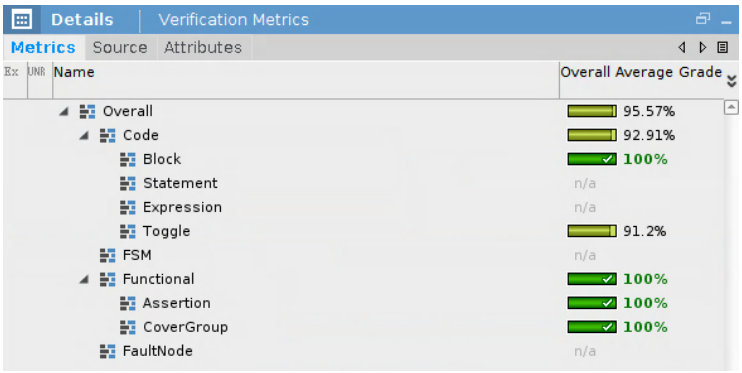

Figure 7.9: Verification Metrics for PIM core with W=4

Figure 7.10 captures Covergroups analysis of the pPIM core using 4-bit operands. It showcases the cross coverage of data words and opcodes, providing valuable insights into the randomness of stimuli.

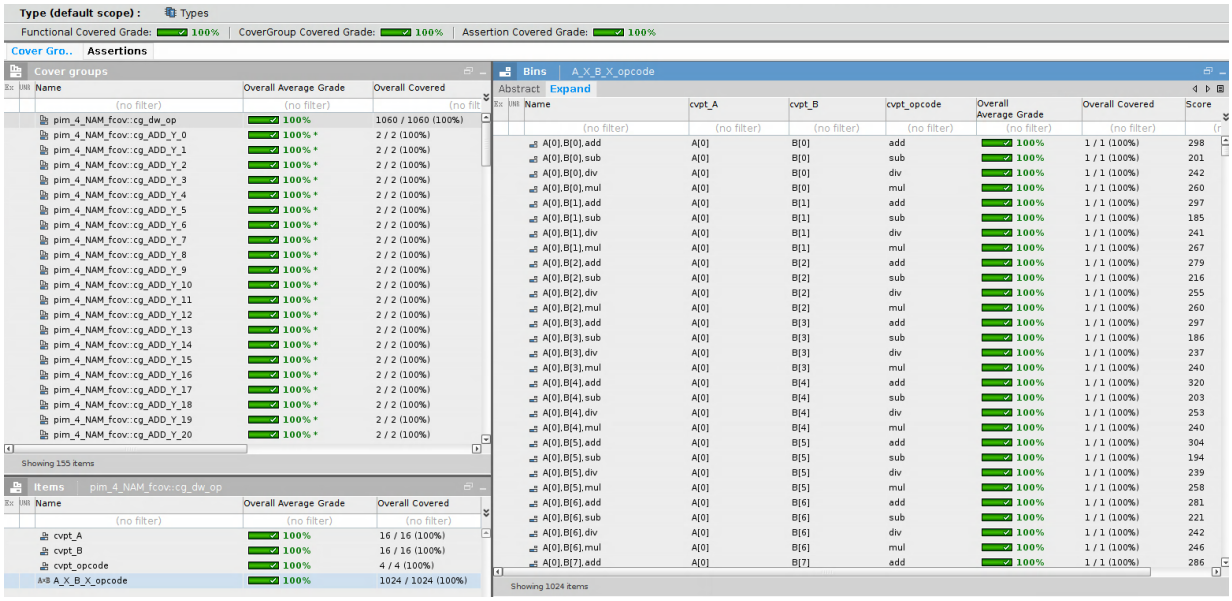

Figure 7.10: Covergroups Analysis for PIM core with W=4

Figure 7.11 presents the Assertion Coverage analysis of the pPIM core using 4-bit operands. The analysis includes both Cover Properties (CP) and Assertion Properties (AP). Each assertion written for the core is tested to ensure it is covered at least once.

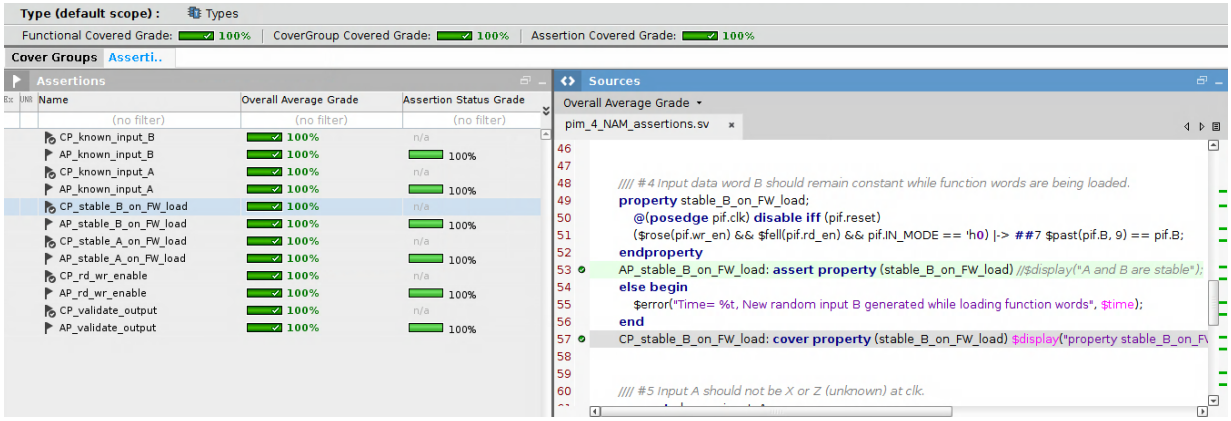

Figure 7.11: Assertion Properties Analysis for PIM core with W=4

Table 7.2 below presents the results of function coverage for each pPIM core variant.

Table 7.2: Functional Coverage for Various Gen 2 PIM Core Configurations

W (Bits)	**Functional Coverage**	
	Assertions	**Covergroups**
2	100%	100%
3	100%	100%
4	100%	100%
5	100%	100%
6	100%	100%
7	100%	100%
8	100%	100%

7.1.1.3 Code Coverage

Table 7.3 presents the code coverage analysis for the pPIM core Verilog code. It includes measurements for both block coverage and toggle coverage. Block coverage indicates the percentage of code blocks executed during verification, while toggle coverage assesses the number of signal transitions exercised by the testbench. Toggle coverage is observed to be not 100%, this is because all the bits in the function word registers do not toggle. The function word registers contain pre-calculated results of all possible combinations of input pairs. These coverage metrics enable engineers to identify untested areas in the code (dead code) and evaluate the thoroughness of their verification efforts.

Table 7.3: Code Coverage for Various Gen 2 PIM Core Configurations

W (Bits)	**Code Coverage**	
	Block	**Toggle**
2	100%	94.72%
3	100%	92.37%
4	100%	92.91%
5	100%	90.75%
6	100%	90.60%
7	100%	92.40%
8	100%	90.59%

Code coverage and functional coverage are both essential aspects of the verification process in hardware design. They complement each other and provide different perspectives on the quality and completeness of the verification effort. Code coverage focuses on analyzing how much of the source code is exercised by the testbench during verification. Code coverage helps identify untested portions of the code, such as unreachable code or dead code, enabling verification engineers to spot potential coding errors or missing functionality. Functional coverage, on the other hand, concentrates on the verification of specific functionality or requirements of the design. It defines coverage goals based on functional specifications and monitors whether these goals are achieved during testing.

7.1.2 Synthesis Results

Synthesis plays a vital role in chip design as it bridges the gap between the high-level RTL description and the actual hardware implementation. During synthesis, the RTL code is transformed into a gate-level netlist, representing the physical hardware components and connections. This process involves mapping the RTL code to specific technology library cells available in the chosen process technology, ensuring the design's feasibility within the process design kit.

For the pPIM Core and Cluster, designers have three options for technology libraries: 28nm, 65nm, and 180nm. Each variant of the pPIM undergoes synthesis, providing valuable insights into design-for-test (DFT) considerations and the total area required by the design. Moreover, prime time and prime power reports are generated, facilitating a comparison of the different pPIM designs in terms of timing and power characteristics.

The test coverage for all cores is observed to be 100%, indicating that all scan cells (flip-flops) are exercised during the scan test. This metric ensures that all flip-flops are reachable and correctly connected in the scan chain, providing comprehensive verification of the design.

The area and power consumption increase significantly as the size of the Lookup Table (LUT) expands with higher data-word widths. The increase in LUT size leads to an increase in area and power requirements.

Data arrival time refers to the timing information related to when valid data becomes available at the input of a digital circuit relative to a reference clock edge. The data arrival time values in the Table 7.4 are measured from register to register.

7.1.2.1 28nm Synthesis Results

Table 7.4: 28nm Synthesis Results for Various PIM Core Configurations

W (Bits)	**Test Coverage**	**Total Area** (μm^2)	**Power** (W)	**Delay** (ns) **(Data Arrival Time)**
2	100%	1446	$1.084e^{-04}$	0.4300
3	100%	8519	$7.350e^{-04}$	0.4380
4	100%	48128	$3.043e^{-03}$	0.5410
5	100%	271459	0.0187	0.5410
6	100%	1574933	0.0899	0.4380

The plot of core input data word width vs. total area is shown in Figure 7.12.The X-axis represents core operand widths, and the Y-axis shows the Total Area measured using synthesis tools. The Total Area is presented on a logarithmic scale, showing a consistent linear growth pattern. This observation indicates an exponential rise in the total area as the operand width expands, reflecting the LUT size's exponential growth in powers of 2.

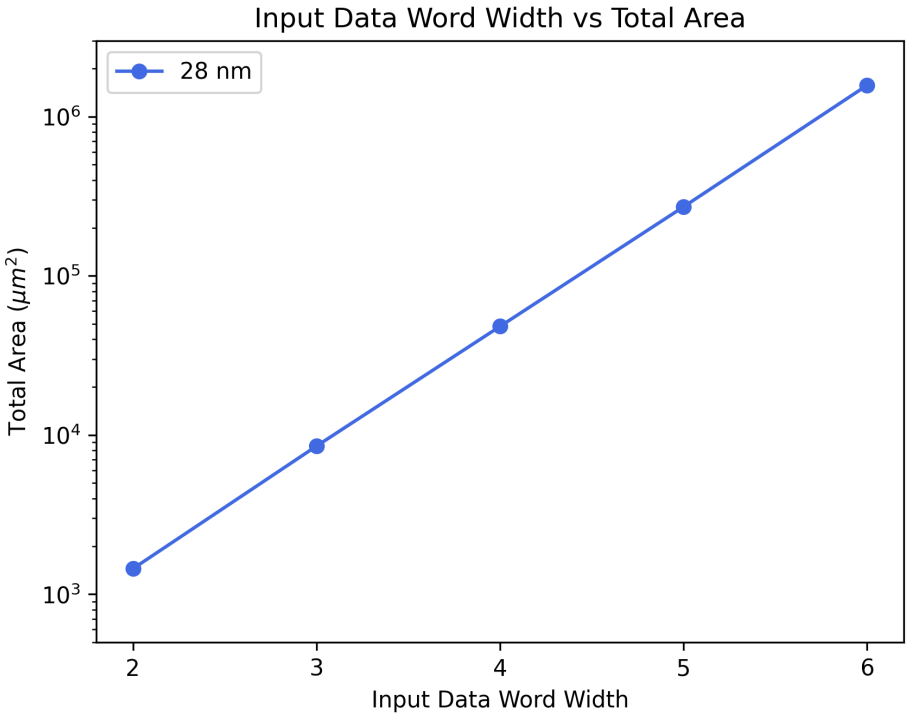

Figure 7.12: Plot of Input Data Word Width (W) vs Total Area for 28nm pPIM Core

7.1.2.2 65nm Synthesis Results

The synthesis results collected for 65nm technology library for the pPIM Core shows similar behavior in area, power and timing.

Table 7.5: 65nm Synthesis Results for Various PIM Core Configurations

W (Bits)	**Test Coverage**	**Total Cell Area (μm²)**	**Power (W)**	**Delay (ns) (Data Arrival Time)**
2	100%	1264	$7.743e^{-05}$	0.1890
3	100%	7128	$4.375e^{-04}$	0.3040
4	100%	36799	$2.295e^{-03}$	0.3740
5	100%	182017	0.0113	0.3740
6	100%	869455	0.0539	0.3450

The plot of core input data word width vs. total cell area is shown in Figure 7.13.

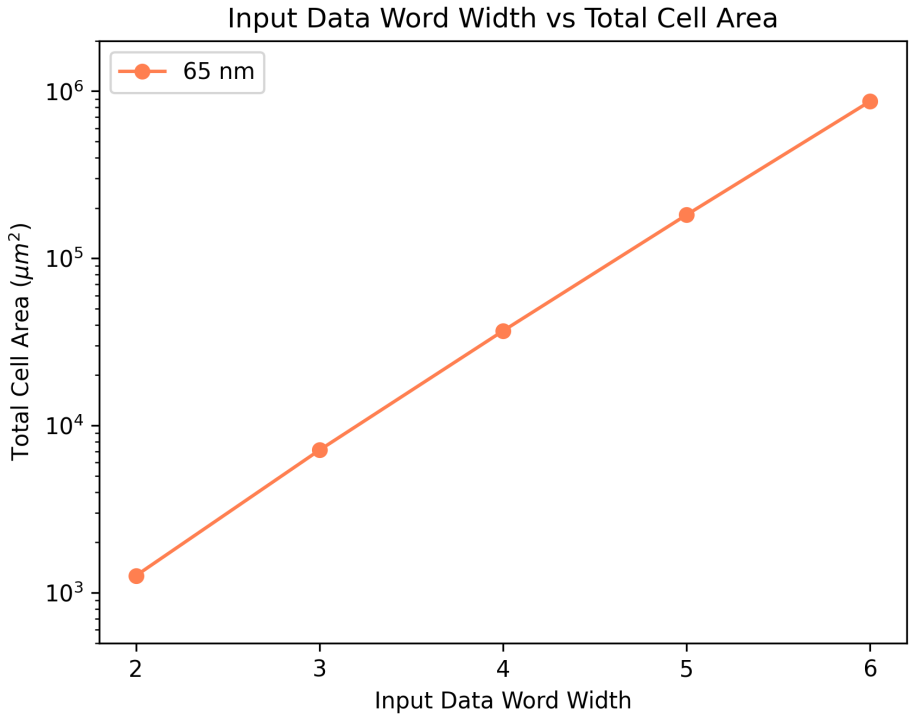

Figure 7.13: Plot of Input Data Word Width (W) vs Total Cell Area for 65nm pPIM Core

The Plot 7.14 below showcases a comparison between core operand width against power, considering both 28nm and 65nm technologies. The power data also follows a consistent linear growth pattern when presented on a logarithmic scale, signifying an exponential surge in total power as the core's operand width increases.

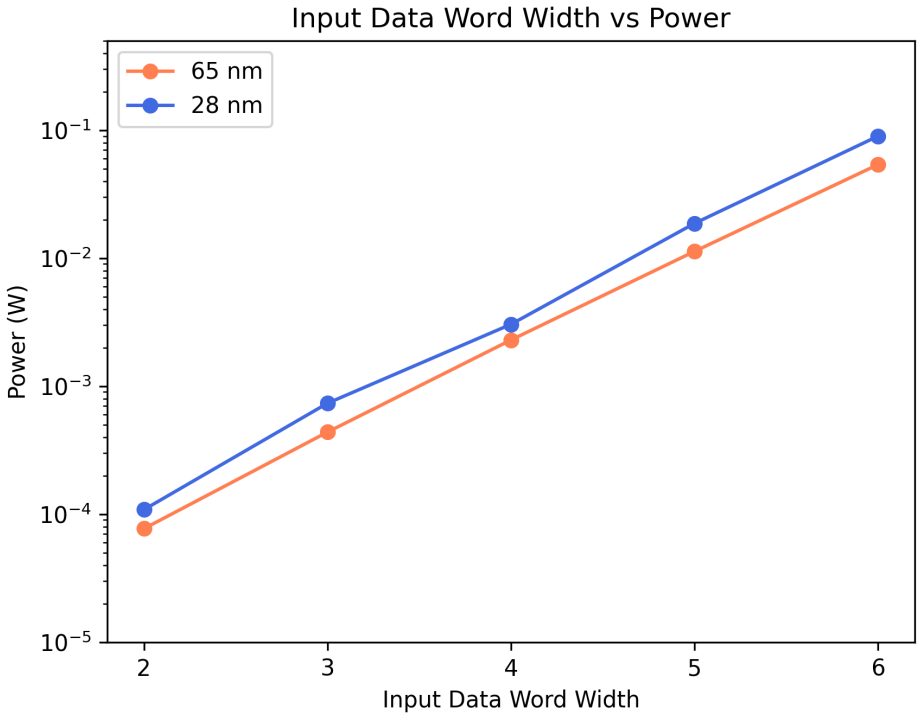

Figure 7.14: Comparison of Input Data Word Width (W) vs Power for 28nm and 65nm pPIM Core

7.2 Gen 2 pPIM Cluster

This section discusses results for the Gen 2 pPIM Cluster.

7.2.1 Verification Results

7.2.1.1 Simulation Results

The waveform diagrams illustrating the MAC (Multiply and Accumulate) operation are depicted in the Figure 7.15 below. The initial MAC instruction yields an output after 180ns from the introduction of the first set of inputs into the processing cluster. Subsequent MAC instructions, following the initial one, generate outputs at intervals of 140ns each. The incorporation of a partial pipeline has resulted in a notable enhancement of 22.22% in the performance of the MAC operation implementation.

Two markers of the same color denote inputs and output of the cluster. The core inputs and outputs produced by pPIM core within the cluster can also be seen in the waveform. The core inputs are seen on internal wires sA_*n*, sB_*n* and resulting core outputs on sY_*n*, where *n* goes from 0 to 8. Each pPIM core is marked with a different color. In the waveform, core 0 is positioned at the top and core 8 at the bottom. In the schematic of the pPIM Cluster depicted in I.3, the colors of the cores match the arrangement shown in the waveform below.

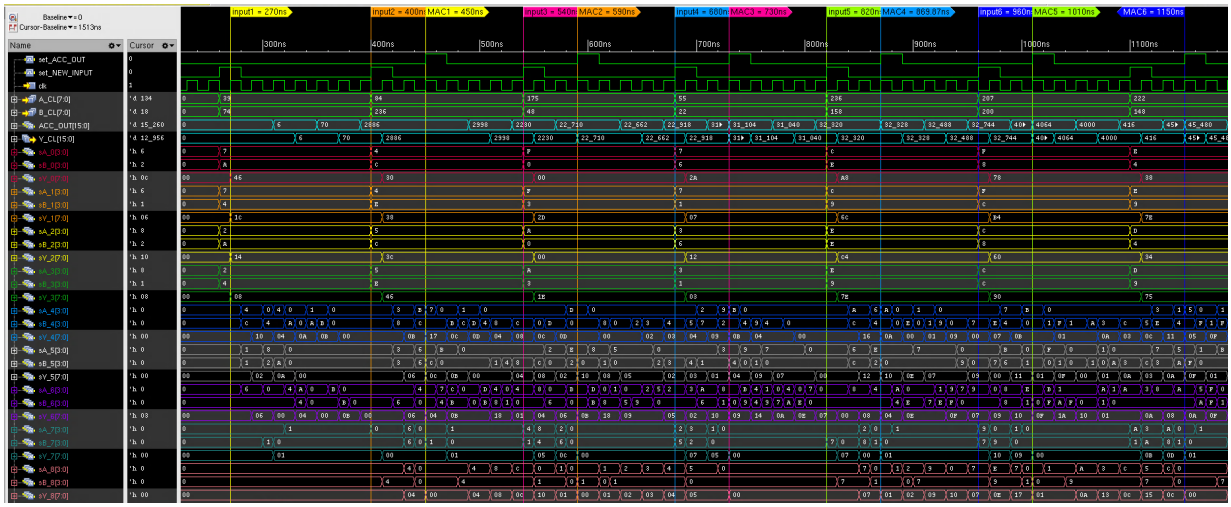

Figure 7.15: MAC Operation: 8-bit Cluster Operands with 4-bit Core Operands

7.2.1.2 Functional Coverage

Table 7.6 displays the functional coverage results for different pPIM Cluster variations. The data collected confirms that MAC instructions can be successfully carried out on any cluster variant. This also implies that Gen 2 user-defined clusters have the potential to efficiently handle data-intensive tasks.

Table 7.6: Functional Coverage for Various PIM Core Configurations

Cluster Operands Width (Bits)	**Core Operands Width (W) (Bits)**	**Functional Coverage (Covergroups)**
4	2	100%
6	3	100%
8	4	100%
10	5	100%
12	6	100%
14	7	100%
16	8	100%

Figure 7.16 illustrates the extent of coverage for the inputs and outputs of the cluster, as well as coverage for the results from the Design Under Test (DUT) and the expected results from the computational model included in the testbench. Coverpoints and Covergroups provide statistical analysis of the observed items and also give insight into the quality of the stimuli.

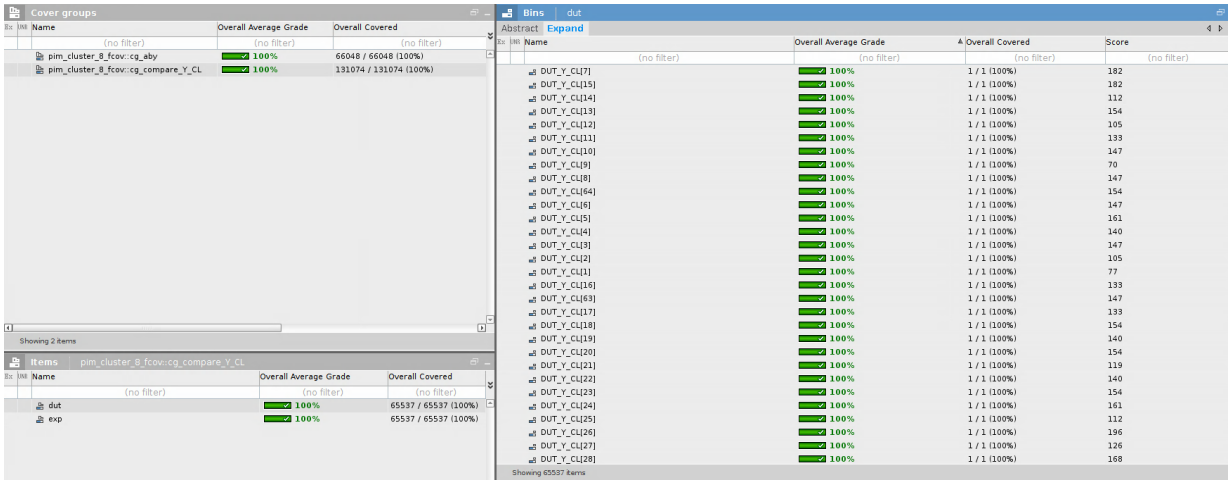

Figure 7.16: Covergroups Analysis for PIM cluster with W=8

7.2.2 Synthesis Results

Synthesis results collected for Gen 2 pPIM Cluster using 28nm technology library are presented in this section.

7.2.2.1 28nm Synthesis Results

The synthesis results collected for 28nm technology library for the pPIM Cluster shows similar behavior in area, power and timing. As the operand width of the individual cores within the cluster increases, there is an exponential increase in the size of LUTs for each core, consequently leading to an expansion in the overall cluster size and power.

Table 7.7: 28nm Synthesis Results for Various PIM Cluster Configurations

A_CL, B_CL (Bits)	**W (Bits)**	**Test Coverage**	**Total Area (μm^2)**	**Power (W)**	**Delay (ns) (Data Arrival Time)**
4	2	100%	17732	$9.94e^{-04}$	0.4300
6	3	100%	90467	$6.75e^{-03}$	0.4380
8	4	100%	520008	0.0341	0.5300
10	5	100%	2804968	0.1692	0.5410

The plot of cluster operand width vs. total area is shown in Figure 7.17.

Figure 7.17: Plot of Cluster Operand Width (W) vs Total Area for 28nm pPIM Cluster

The plot of cluster operand width vs. power is shown in Figure 7.18.

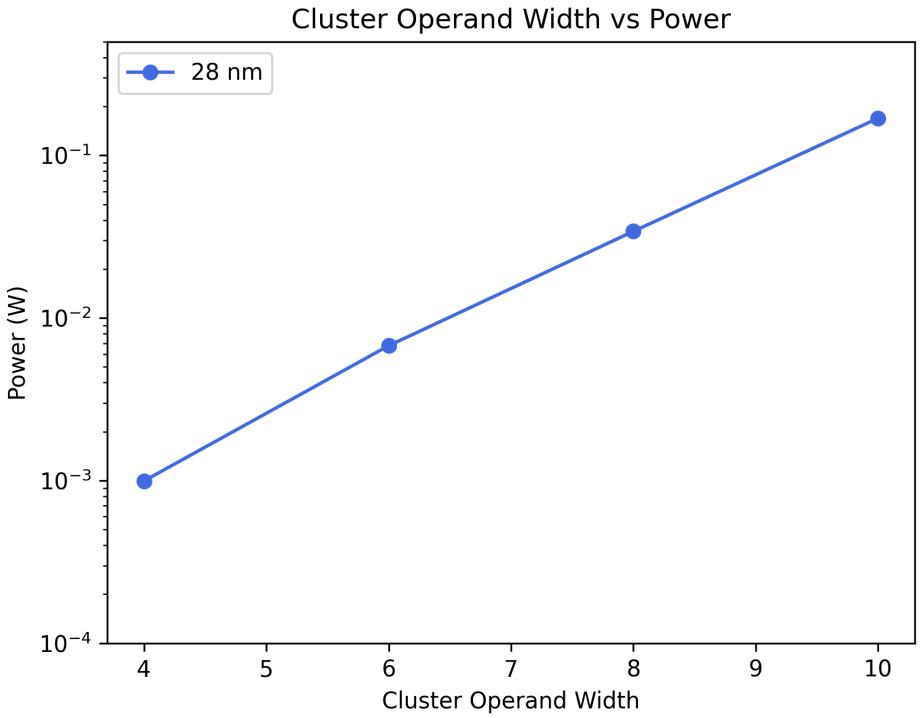

Figure 7.18: Plot of Cluster Operand Width (W) vs Power for 28nm pPIM Cluster

This chapter concludes verification and synthesis results for both Generation 2 pPIM Core and Generation 2 pPIM Cluster.

Chapter 8

Conclusion

In conclusion, this thesis culminates in a comprehensive overview of the contributions and advancements made in of Generation 2 Programmable Processing in Memory (pPIM) architecture:

1. Generation 2 pPIM Core: Thesis presents the Generation 2 pPIM core, a redesigned, advanced version of the previous static architecture, Generation 1 pPIM core, as a key contribution. The redesigned core architecture handles operand width in range 2-bits to 8-bits seamlessly eliminating limitation of only supporting 4-bit inputs. The operand width is fixed at the time of RTL generation. Gen 2 pPIM Core demonstrates successful execution of any programmed operations. The Verilog RTL code for the pPIM core is revised. By simplifying the code, making it scalable, and ensuring synthesizability, the design becomes more versatile and adaptable to synthesis tools.

2. Generation 2 pPIM Cluster: Thesis presents the Gen 2 pPIM Cluster, a redesigned, advanced version of the previous architecture. Flexibility is improved with individually programmable cores. The processing elements inside the cluster such as router, pPIM cores, are redesigned, including non-blocking router, integrated accumulator, and integrated output register. Gen 2 pPIM Cluster operates on operand widths of 4, 6, 8, 10, 12, 14, and 16. Gen 2 pPIM Cluster

inputs are double the size of the core inputs residing in it, and the cluster operand width is also fixed at the time of RTL generation.

3. Gen 2 pPIM Router: As noted above, this work includes a redesigned the router inside the Gen 2 pPIM Cluster, which significantly enhances the cluster's capabilities. The new router includes components such as input multiplexers and output multiplexers that allow non-blocking, all-to-all communication between the cores, accumulator and output register. The router exhibits crossbar architecture. The cores are able to feed themselves back their own output.

4. Accumulator and Output Register: As noted above, this work includes the design and integration of the accumulator and output register into Gen 2 pPIM Cluster enabling it to implement complex operations. The output register is implemented for timing purposes.

5. UVM Testbenches for Gen 2 pPIM Core and pPIM Cluster: This work includes the research and development of advanced UVM testbenches, complete with constrained-random testing, functional coverage, and SystemVerilog Assertions for the Gen 2 pPIM Core and pPIM Cluster. These testbenches provide efficient, and comprehensive validation of both the Gen 2 pPIM Core and Gen 2 pPIM Cluster. Randomized testing of arithmetic operations thoroughly verifies core functionality. Concurrently, the pPIM Cluster testbench demonstrates data-intensive Multiply-and-Accumulate (MAC) operations, revealing 22.22% performance increase through partial pipelining in the MAC algorithm.

6. Python-Based Design Database Generator Tool Suite: A Python-based command-line interface has been developed to automate the generation of user-defined Gen 2 pPIM pPIM Core and pPIM Cluster designs, along with their verification testbenches. This tool suite reduces manual effort, speeds up the design and verification process, and helps explore

different design options more effectively. This tool has proven to enhance design verification accuracy and efficiency by reducing human involvement and errors.

In summary, this research advances the field of high-performance, data-intensive processors by making pPIM processors adaptable for various applications. The contributions presented in this thesis provide a foundation for further development in the area of programmable processing in memory architectures, with potential applications across a wide range of computational tasks and technologies.

8.1 Future Work

The future work in the area of pPIM include several promising directions for research and development.

- Firstly,the MAC operation is demonstrated as one of the use-cases of the Gen 2 pPIM Cluster, although the cluster is not limited to performing only the MAC. To uncover its full potential across diverse computational tasks, the Gen 2 pPIM Cluster UVM testbench can be extended to map and validate other data-intensive instructions on the cluster effectively.

- Efforts can also be directed towards optimizing the synthesis process, particularly for operands with 7-bit and 8-bit pPIM Core. Exploring strategies such as modular synthesis or incremental synthesis could potentially reduce processing time and enhance the overall design workflow.

- The Gen 2 pPIM Cluster generation tool can be enhanced by the addition of user-defined parameters, such as specifying the number of cores in the cluster through command line options, would provide greater customization and flexibility in cluster design.

- Gen 2 pPIM designs can be guided through a physical layout flow (and example using physical synthesis is shown in I. Additionally, the evolution of the Gen 2 Programmable Processing in Memory is facilitated by the implementation of an instruction set architecture (ISA) hardware, which is a step toward development of this innovative computing paradigm.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Amsterdam: Morgan Kaufmann, 2012.

[2] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, 1997.

[3] P. R. Sutradhar, S. Bavikadi, M. Connolly, S. Prajapati, M. A. Indovina, S. M. P. Dinakarrao, and A. Ganguly, "Look-up-Table Based Processing-in-Memory Architecture With Programmable Precision-Scaling for Deep Learning Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 263–275, 2022.

[4] P. R. Sutradhar, M. Connolly, S. Bavikadi, S. M. Pudukotai Dinakarrao, M. A. Indovina, and A. Ganguly, "pPIM: A Programmable Processor-in-Memory Architecture With Precision-Scaling for Deep Learning," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 118–121, 2020.

[5] M. Connolly, P. R. Sutradhar, M. Indovina, and A. Ganguly, "Flexible Instruction Set Architecture for Programmable Look-up Table based Processing-in-Memory," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021, pp. 66–73.

[6] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg,

"Benchmarking Large Language Models for Automated Verilog RTL Code Generation," in *Design, Automation and Test in Europe Conference and Exhibition*, 2023, pp. 1–6.

[7] C. Fun and N. Thulasiraman, "Synthesizable verilog code generator for variable-width tree multipliers," *Journal of Physics: Conference Series*, vol. 1962, p. 012046, 07 2021.

[8] R. Kulkarni, "Automated RTL generator," mathesis, San Jose State University, (2013). [Online]. Available: https://scholarworks.sjsu.edu/etd_projects/305

[9] S. Ghose, A. Boroumand, J. S. Kim, J. Gomez-Luna, and O. Mutlu, "Processing-in-memory: A workload-driven perspective," *IBM Journal of Research and Development*, vol. 63, no. 6, pp. 3:1–3:19, 2019.

[10] X. e. a. Yang, "A Processing-in-Memory Architecture Programming Paradigm for Wireless Internet-of-Things Applications," *Sensors (Basel, Switzerland) vol. 19,1 140.*, 2019.

[11] X. Zou, S. Xu, X. Chen, L. Yan, and Y. Han, "Breaking the von Neumann bottleneck: architecture-level processing-in-memory technology," *Science China Information Sciences*, vol. 64, no. 6, p. 160404, 2021. [Online]. Available: https://doi.org/10.1007/s11432-020-3227-1

[12] D. E. Shaw, S. Stolfo, H. A. H. Ibrahim, B. Hillyer, G. Wiederhold, and J. N. Andrews, "The NON-VON Database Machine: A Brief Overview," *IEEE Database Eng. Bull.*, vol. 4, pp. 41–52, 1981.

[13] J. Šilc, B. Robič, and T. Ungerer, *Processor-in-Memory, Reconfigurable, and Asynchronous Processors*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 299–333. [Online]. Available: https://doi.org/10.1007/978-3-642-58589-0_7

[14] H. S. Stone, "A Logic-in-Memory Computer," *IEEE Transactions on Computers*, vol. C-19, no. 1, pp. 73–78, 1970.

[15] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. E. Kozyrakis, R. Thomas, and K. Yelick, "Intelligent RAM (IRAM): chips that remember and compute," 03 1997, pp. 224 – 225.

[16] D. G. Elliott, M. Stumm, M. Snelgrove, C. Cojocaru, and R. Mckenzie, "Computational RAM: implementing processors in memory. Des. Test Comput. 16(1), 32-41," *Design and Test of Computers, IEEE*, vol. 16, pp. 32 – 41, 02 1999.

[17] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: a computation model for intelligent memory," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, 1998, pp. 192–203.

[18] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: a modular reconfigurable architecture," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, 2000, pp. 161–171.

[19] *Design Verification Challenges*. Boston, MA: Springer US, 2007, pp. 1–16. [Online]. Available: https://doi.org/10.1007/978-0-387-69167-1_1

[20] C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer US, 2012. [Online]. Available: https://books.google.com/books?id=QaWOYTOXy0EC

[21] J. Bromley, "If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language," in *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, 2013, pp. 1–7.

[22] R. Salemi, *The UVM Primer: A Step-By-Step Introduction to the Universal Verification Methodology*. Boston Light Press, 2013. [Online]. Available: https://books.google.com/books?id=h7MLngEACAAJ

Appendix I

Schematics and Layouts

This section presents the schematics and layouts for Generation 2 pPIM Core with 4-bit operands and Generation 2 pPIM Cluster with 8-bit operands. For simulation Cadence Xcelium tools are used and for producing layouts, Synopsys IC Compiler Physical Synthesis tools are utilized.

I.1 Gen 2 pPIM Core (W=4) Schematic

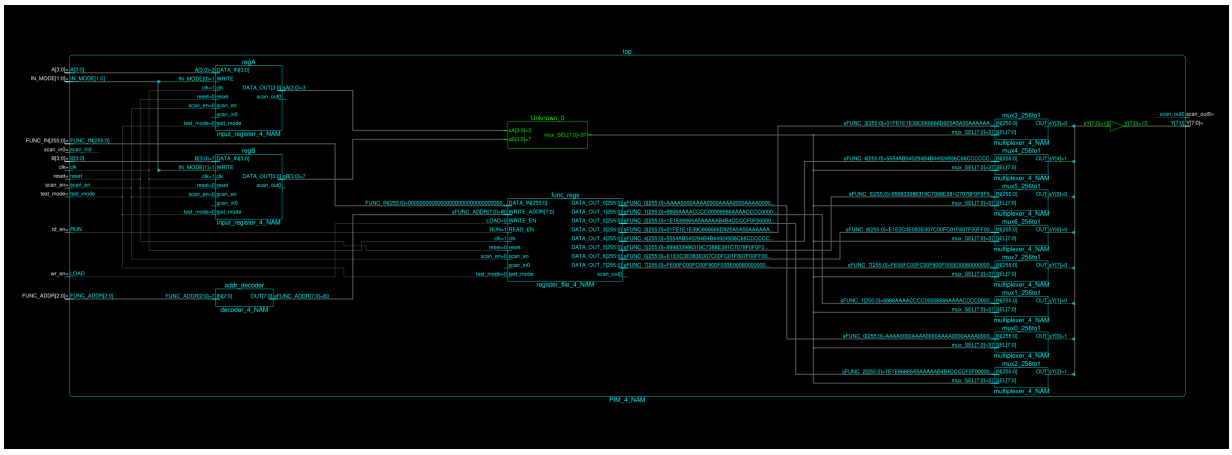

Figure I.1: Schematic for pPIM Core with Operand Width 4-bits

I.2 Gen 2 pPIM Core (W=4) Layout

pPIM Core area (Operand width 4 bits) collected using Synopsys IC Compiler Physical Synthesis tools : 52517 μm^2 in 28nm.

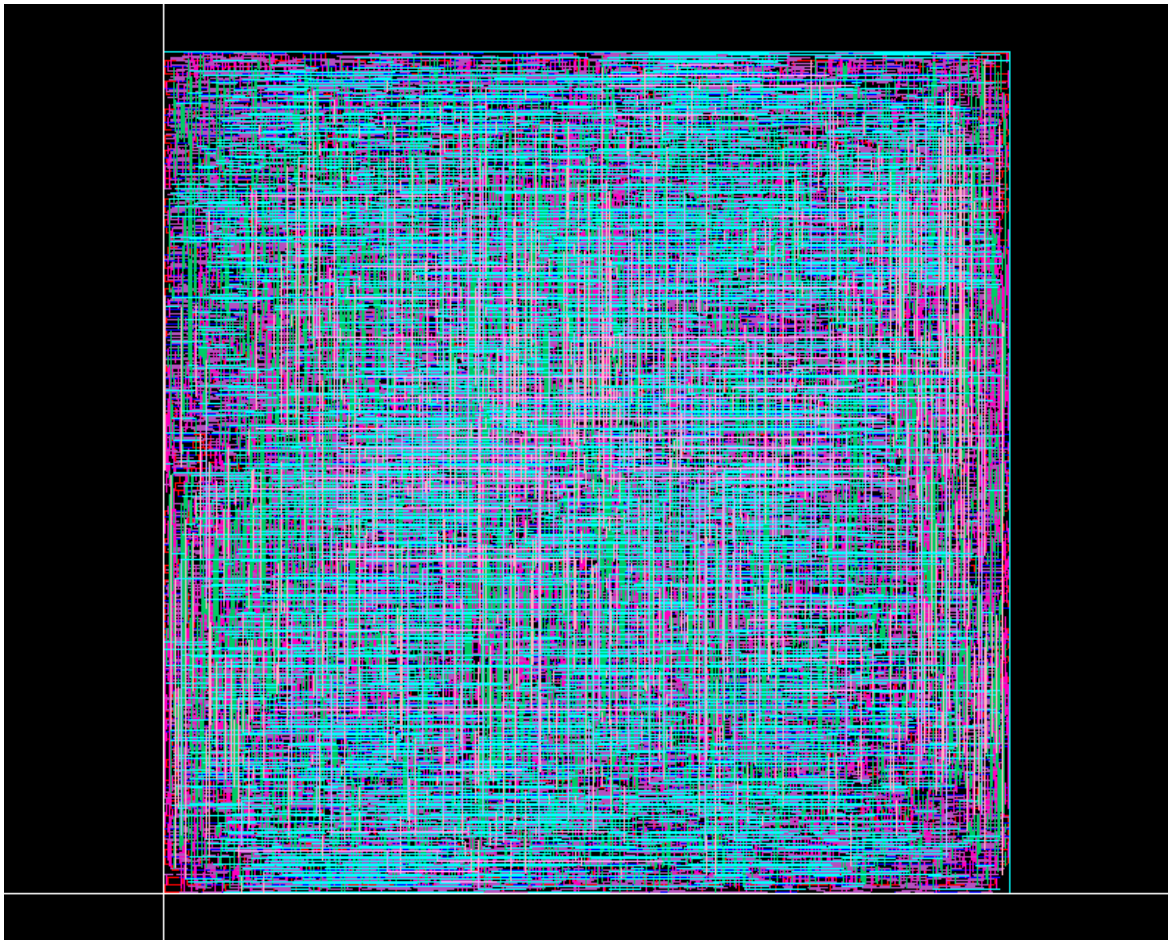

Figure I.2: Synopsys IC Compiler Physical Synthesis Layout for pPIM Core in 28nm with Operand Width 4-bits

I.3 Gen 2 pPIM Cluster (W=8) Schematic

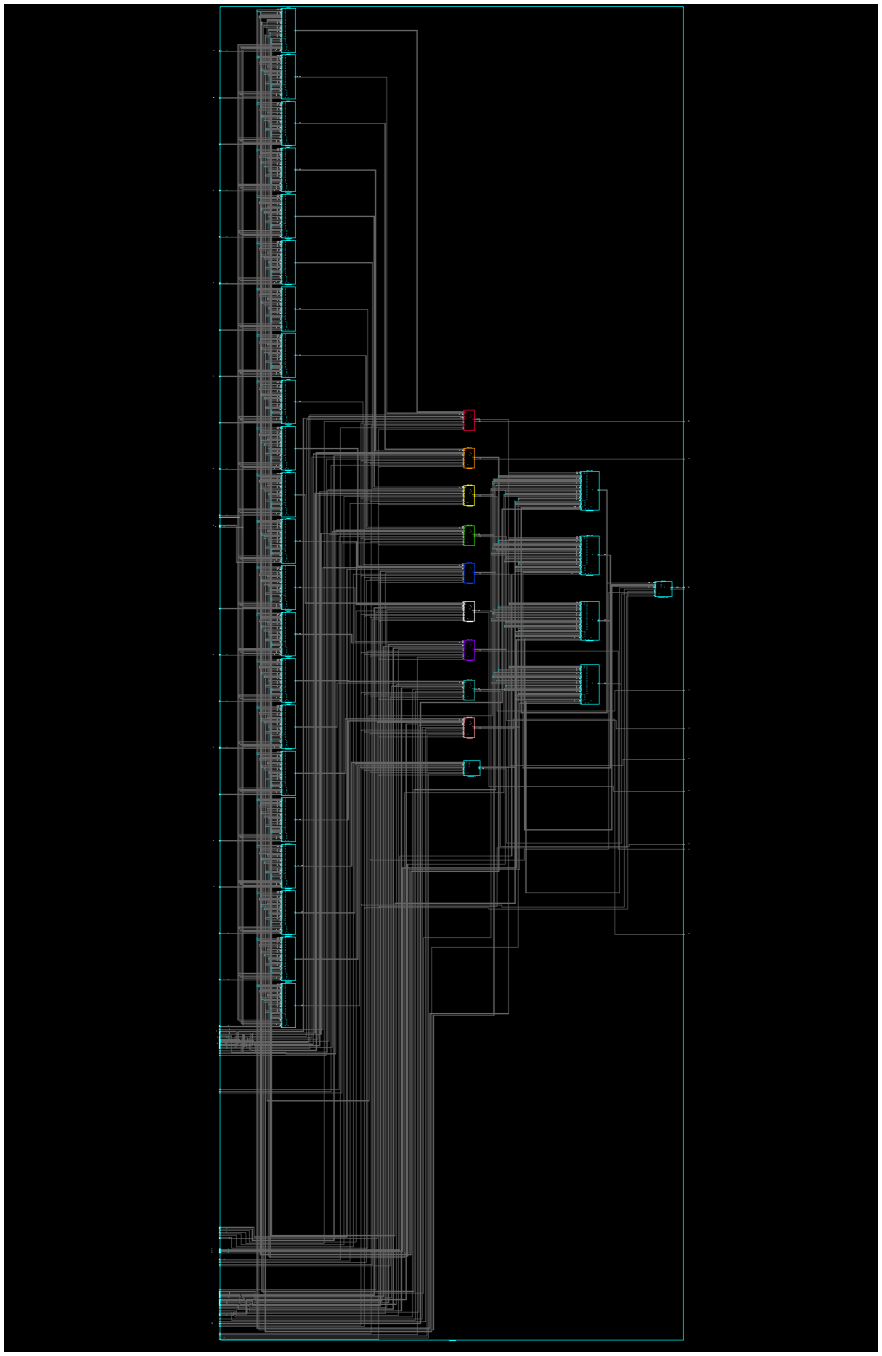

Figure I.3: Schematic for pPIM Cluster with Operand Width 8-bits

I.4 Gen 2 pPIM Cluster (W=8) Layout

pPIM Cluster area (Operand width 8 bits) collected using Synopsys IC Compiler Physical Synthesis tools : 671662 μm^2 in 28nm.

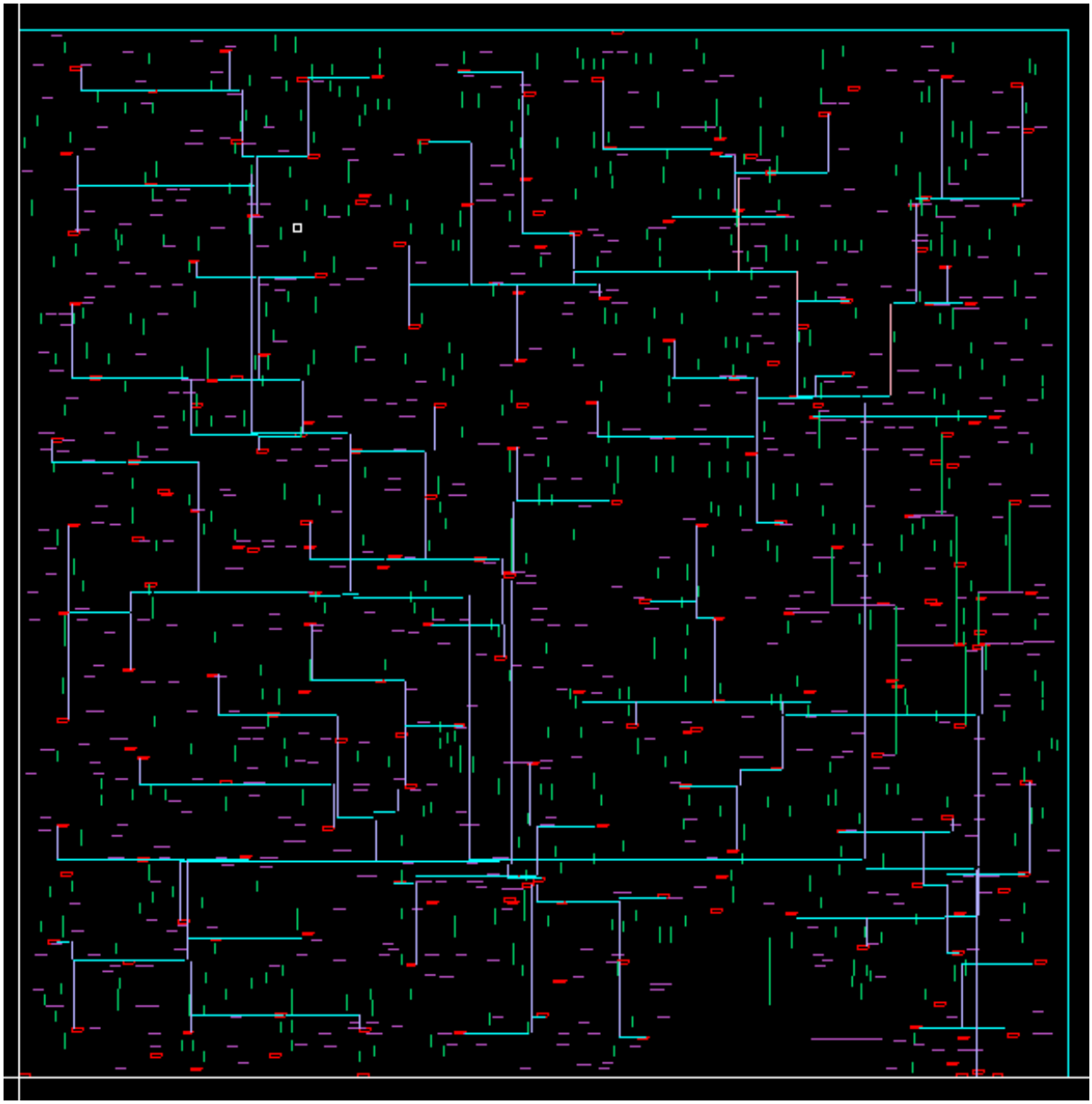

Figure I.4: Synopsys IC Compiler Physical Synthesis Layout for pPIM Cluster in 28nm with Operand Width 8-bits

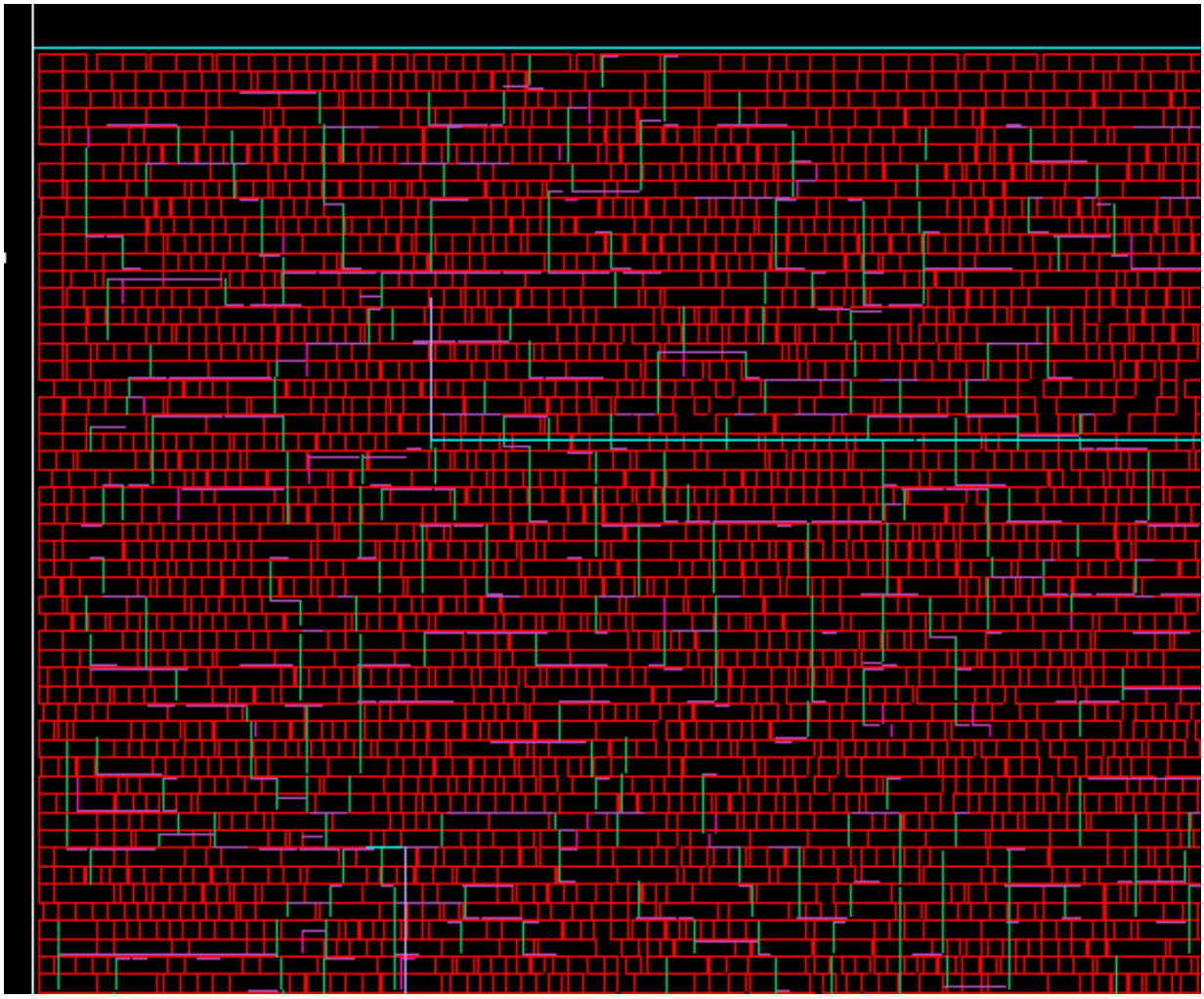

Figure I.5: Top Left Corner View of Synopsys IC Compiler Physical Synthesis Layout for pPIM Cluster in 28nm with Operand Width 8-bits

Appendix II

Source Code Request

All requests for a source code release package should be made by email to Mark Indovina at Rochester Institute of Technology: maieee @ rit . edu